**FLIGHT BOOKING APPLICATION USING MERN**

**ABSTRACT:**

This project presents a comprehensive **flight booking application** developed using the **MERN stack** (MongoDB, Express.js, React.js, and Node.js), designed to enhance the booking experience for travellers and streamline service for travel providers. The application leverages the Model-View-Controller (MVC) architecture to establish a clear separation between data management, user interface, and core business logic, ensuring a scalable, maintainable, and high-performing solution.

In today’s digital era, user-friendly, responsive, and scalable web applications are essential in sectors like travel and tourism. This project presents a robust flight booking application developed using the MERN stack, with a strong focus on providing a seamless experience for both travellers and service providers. Through the Model-View-Controller (MVC) architecture, our application separates data management, user interaction, and core business logic, enhancing maintainability and scalability.

Our project is dedicated to addressing the significant challenges faced by both tourists and tourism service providers by creating a comprehensive, integrated platform using the MERN stack (MongoDB, Express.js, React.js, Node.js). This platform is designed to facilitate seamless interactions between tourists and various stakeholders within the tourism industry, aiming to enhance the overall travel experience and streamline service delivery.

Our focus on a responsive user interface extends across devices, ensuring that the platform is accessible on both desktop and mobile, so tourists can conveniently plan their trips from anywhere. The platform design emphasizes ease of use, providing clear navigation, robust search features, and easy access to essential information. Through careful attention to design and user experience, we enable tourists to efficiently find services, make informed decisions, and customize their travel plans. The platform also offers service providers tools to manage listings, interact with customers, and analyses feedback to improve their offerings.

**INTRODUCTION:**

**Vision and Purpose:**

In today’s world, travel has transformed from a simple journey into an immersive, often life-changing experience, with technology playing a crucial role in defining how people explore, connect, and engage with their destinations. Our vision for this MERN (MongoDB, Express.js, React.js, Node.js) project is to leverage cutting-edge technology to reshape the tourism landscape, creating a digital solution that caters to the diverse, evolving needs of modern travellers while empowering tourism stakeholders to craft personalized, impactful, and memorable experiences.

**Enriching the Travel Experience:**

Our project recognizes that travellers today seek more than traditional sightseeing; they desire authentic experiences that connect them to the local culture, environment, and community. By leveraging the MERN stack, this platform aims to provide users with rich, immersive experiences. With interactive features, intuitive search functionalities, and curated content, travellers can easily explore and personalize their journeys, accessing local insights, activity suggestions, and real-time updates that transform travel into a tailored experience.

**Empowering Tourism Stakeholders:**

Through an integrated backend built with Node.js and Express.js, stakeholders can access a powerful suite of tools to manage listings, analyses traveller preferences, and offer services that resonate with their target audiences. By implementing data-driven analytics, the platform enables service providers to make informed decisions, personalize offerings, and stay agile in a competitive market. MongoDB’s flexible, scalable database structure further supports the ability to handle diverse datasets, from user profiles to location-specific content and booking history, ensuring a comprehensive and responsive service.

**History of Online Travel Booking Application:**

The history of online travel booking traces back to the emergence of the internet and the digitalization of the travel industry. Here's a brief overview of its evolution.

**1990s - Emergence of Online Travel Agencies (OTAs):**

In the early days of the internet, a few pioneering companies began to explore the potential of selling travel services online.

One of the earliest online travel agencies was Travelocity, launched in 1996 as a subsidiary of Sabre Corporation.

**Late 1990s to Early 2000s - Rapid Growth and Consolidation:**

The late 1990s and early 2000s witnessed rapid growth in the online travel industry as more consumers embraced the convenience of booking travel services online.

Expedia and Travelocity emerged as leading OTAs, along with other players such as Orbitz, Priceline, and Booking.com.

**Mid-2000s to Present - Innovation and Diversification:**

The mid-2000s saw continued innovation and diversification within the online travel industry.

OTAs expanded their offerings to include a wide range of travel services beyond flights and hotels, such as vacation rentals, activities, and travel insurance.

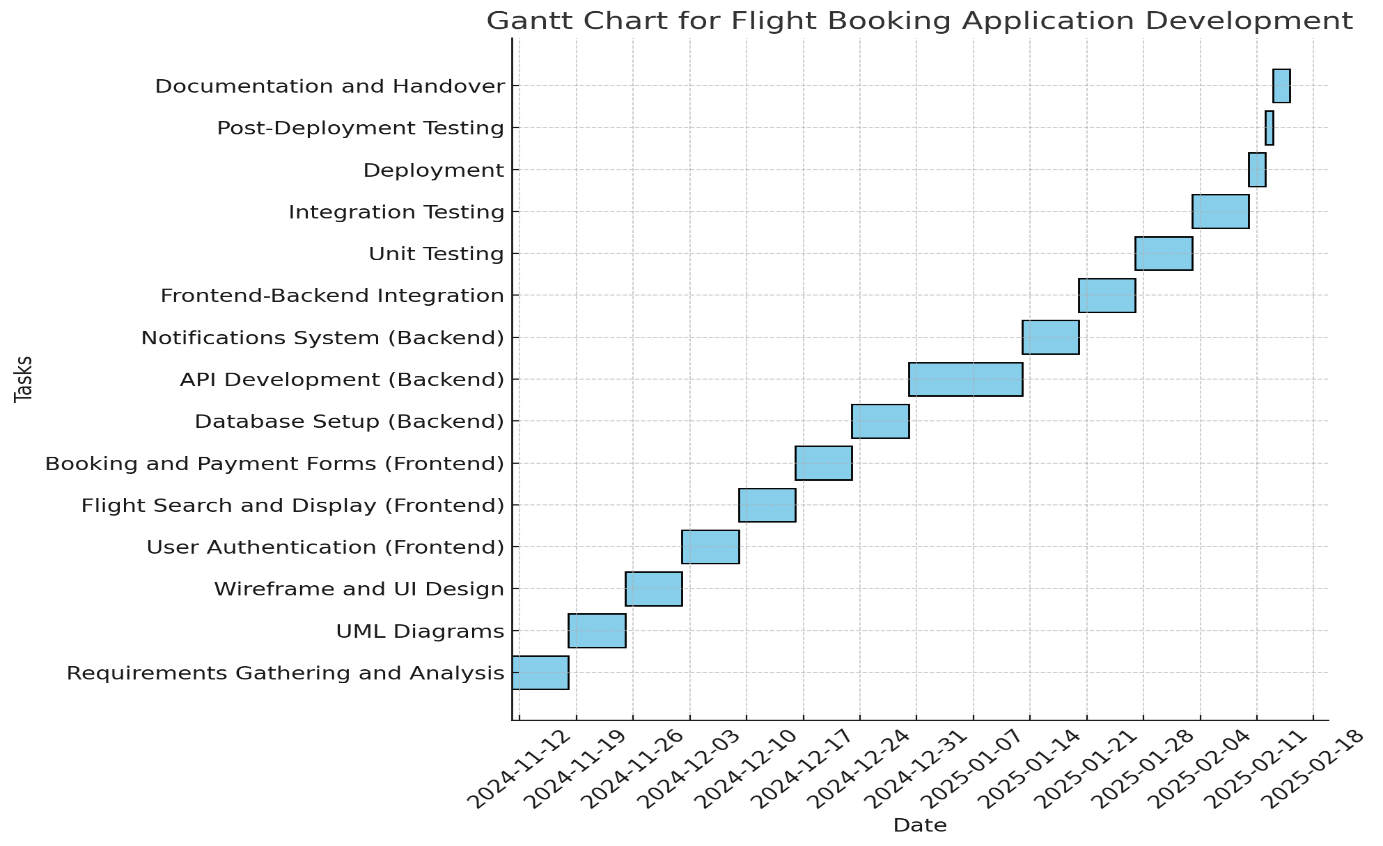
**WHAT IS A GANTT CHART AND HOW IT FOCUSES ON ANANLYSING THE OVERALL IMPLEMENTATION DONE FOR DEVELOPING THE FLIGHT BOOKING APPLICATION:**

A **Gantt chart** is a type of bar chart that visually represents a project schedule over time. It lists project tasks on the vertical axis and time intervals (like days or weeks) on the horizontal axis. Each task is represented by a horizontal bar that shows its start and end dates, making it easy to see task durations and overlaps at a glance.

For developing a **flight booking application**, a Gantt chart can help organize and track progress across the various phases involved. Here's how it would typically work for this kind of project:

1. Defining Project Phases and Tasks
2. Scheduling and Allocating Time for Each Task
3. Tracking Dependencies and Overlaps
4. Monitoring Progress

**The Gantt Chart for the Flight Booking Application:**



**WHAT IS MVC AND WHY ITS IMPORTANT IN THE APPLICATION WE ARE GONNA DEVELOP:**

**Model-View-Controller (MVC)** is a design pattern widely used in software development, particularly in web applications. It divides an application into three interconnected components, each responsible for a distinct aspect of the application’s functionality. This separation helps manage the complexity of the application, improve maintainability, and allow developers to work more efficiently by focusing on one aspect of the application at a time.

**1. Model**

* **Definition**: The **Model** is responsible for managing the data and business logic of the application. It defines how data is stored, retrieved, and manipulated, and it represents the "real-world" entities in the application.
* **Role**: It communicates with the database, applies business rules, and manages the data structure.

**2. View**

* **Definition**: The **View** is the presentation layer of the application. It is responsible for displaying the data to the user and defining the structure and layout of the UI.
* **Role**: The View interacts with the user and reflects the data provided by the Model.

**3. Controller**

* **Definition**: The **Controller** acts as the intermediary between the Model and View. It processes user inputs, communicates with the Model to fetch or update data, and determines which View to display.
* **Role**: The Controller interprets user inputs from the View, updates the Model, and then selects the appropriate View for the user.

**HOW THE MVC ENHANCES THE OVERALL FUNCTIONING OF OUR APPLICATION?**

Since our application is built using React for the frontend and Node.js for the backend, it follows the Model-View-Controller (MVC) design pattern to organize the code effectively and improve functionality.

**Model:** We use MongoDB as our database to store and manage information such as flight data, user profiles, and booking details. MongoDB’s flexible, schema-less structure allows us to store data in a variety of formats, making it adaptable and scalable. The Model layer handles all database interactions, providing a seamless and optimized data experience for the application.

**View:** Our React frontend serves as the View layer, creating an interactive and responsive user interface for users to explore flights, manage bookings, and view details. React' s component-based design enables a modular approach, ensuring that the UI is engaging and easy to update. This layer solely focuses on the presentation and does not handle data directly, making it easier to maintain and scale the UI independently.

**Controller:** The Node.js and Express backend functions as the Controller, connecting the user interface with the database. It processes user requests, communicates with the Model to retrieve or update data as needed, and sends responses to the View to update the display accordingly. This layer handles the business logic, ensuring data flows smoothly between the frontend and the backend.

**By following the MVC pattern, we’ve created a structure that:**

1. Separates concerns between data management, UI presentation, and business logic:

By isolating the different concerns (data, UI, and user input), MVC makes your codebase more organized and easier to maintain. Changes to one part of the system (e.g., updating the UI) can be made without affecting the business logic or data processing.

1. Enhances the app’s scalability and maintainability, making it easy to introduce new features:

The modularity of MVC allows the application to grow over time by adding new features or supporting more complex interactions without significant refactoring. Each part can be worked on independently.

1. Provides a responsive and efficient user experience, with a backend that ensures data consistency and security:

MVC allows developers to maintain a clear structure where each component is responsible for a specific part of the functionality, making it easier to fix bugs, implement changes, or add new features.

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| **Section** | **Description** |
| What’s a Flight Booking App and the Reason for Choosing It? | Describe what a flight booking app is, and why your team chose to develop it. We have also explained about the importance of this application with that we have also mentioned how our application |
| What Makes Our Application Different from Other Applications? | The unique features, functionalities, or design elements that distinguish our app from competitors. |
| Implementations Done | The key implementations we have made, such as user interface design, payment integration, and flight search optimization. |
| Installation Commands | List the commands needed to install and set up the project, including dependencies and any required setup steps. |
| Workflow Diagram | Include or describe the flow of how users interact with your application, from searching flights to booking. |
| ER Diagram | Provide an Entity-Relationship (ER) diagram to illustrate the database structure and relationships between entities. |
| Architecture of Our Application | Describe the architecture, such as MVC (Model-View-Controller), microservices, or other patterns used in your app. |
| Technology Stacks We Have Used | List all technologies, frameworks, and libraries used in the project, like React, Node.js, MongoDB, etc. |
| Importance of Each Tech Stack Component | Explain why each technology was chosen and how it contributes to the overall app development. |
| Frontend Functionality | Describe how the frontend part of your app works, such as React components, routing, and styling. |
| Backend Functionality | Explain how the backend handles requests, processes data, and connects to the database. |
| Database Functionality | Outline how the database stores, retrieves, and organizes flight and user data. |
| Importance of API and Its Main Functionality | Discuss the significance of APIs in your app and their role in connecting frontend and backend functionalities. |
| Importance of JSON web token | We have provided a clear and detail explanation of why we have used this and how it enhances the overall authentication of the application. |
| Importance of Git and GitHub | Explain how Git and GitHub were used in collaboration, version control, and maintaining code integrity. |
| Coding Part | Share some coding snippets or describe key pieces of code that were essential to the project. |
| Execution Part | Describe the steps for executing the application, from running the server to testing functionalities. |
| Overall Journey for Building the Application | Reflect on the process, challenges faced, and milestones achieved throughout the project. |
| Conclusion | Summarize the project outcomes, lessons learned, and potential future improvements. |
| References | Cite any resources, tutorials, documentation, or guides that helped in building the project. |

We have actually discussed in detail about each and every content within the table of content.

Let’s dive into the actual project part.

**What’s a Flight Booking App and the Reason for Choosing It?**

**What’s a Flight Booking Application?**

A flight booking application is a digital platform that allows users to search, compare, and book flights online.

Typically, it provides a user-friendly interface for travelers to:

* Search for flights based on destinations, dates, and preferences (like non-stop flights or preferred airlines).
* Compare flight options, including ticket prices, flight times, and layover details.
* Book tickets and pay securely, often offering multiple payment methods.
* Manage bookings, which includes making changes, checking flight status, or canceling reservations.
* Access additional services, such as seat selection, baggage options, and travel insurance.

**Importance of using a Flight Booking Application:**

Have you ever wondered why a Flight Booking Application was developed well it was just developed to reduce the overall time taken for any person to book their own tickets and to get disappointed if their tickets were not booked properly. In order to reduce all these stuffs our project works in reducing the time period for a particular person to go the airport and to book their ticket, they can do within a short period of time with this application.

**How our application works?**

This application works with two main features:

* 1. The user who is going to book tickets
  2. The admin who is going to update the flight details to the user
  3. The Operator who is going to add or remove the flight details.

Feeling Kinda curious, then why are you waiting let’s get into the actual functioning of the Flight Booking Application and know more about it’ s workflow.

**What Makes Our Application Different from Other Applications?**

As mentioned above we have developed this application in such a way the User, Admin, the flight operator can use it without any authentication problem in it. For this we have used the **Json Web Token** which focuses on maintaining authentication for the overall flight booking application.

**What’s this JSON WEB TOKEN and how it handles those authentication features?**

**JSON Web Token (JWT)** is a compact, **URL-safe token** format used for securely transmitting information between parties as a **JSON object**. In the context of authentication, JWT allows a server to verify the identity of a client and authorize access to resources without needing to manage session data on the server side, making it popular for web applications, particularly in stateless architectures like REST APIs.

In a **flight booking application**, **JSON Web Tokens (JWT)** play a crucial role in handling authentication and securing user interactions with the system. Here’s how JWT authentication typically works within this type of application:

**Key Uses of JWT in a Flight Booking Application:**

1. **User Authentication**:
   * When a user logs in (e.g., as a passenger, admin, or airline staff), the backend server verifies their credentials (like username and password).
   * Upon successful login, the server generates a JWT containing key information (e.g., user ID, roles, and any permissions), then sends this token back to the client (frontend).
2. **Token Storage**:
   * The client (e.g., a web or mobile app) stores the JWT in a secure location, typically in local storage or cookies.
   * This token will be attached to every subsequent request that requires authentication, making the user’s session “stateless” on the server, as there is no need for session storage.
3. **Authorization**:

* Once authenticated, the JWT helps control access to specific resources based on user roles or permissions. For example:
  + **Passengers** can search for flights, make bookings, and view their booking history.
  + **Admins** can access booking records, manage flights, and oversee customer service requests.
  + **Airline Staff** might access check-in and boarding information for passengers.
* This separation ensures that only users with appropriate roles can access certain features or data, protecting sensitive information.

1. **Session Expiration and Token Refreshing**:

* JWTs usually have an **expiration time** to improve security. For example, a token might expire after 24 hours.
* If a user’s session expires during usage, the client might request a new token using a **refresh token** (if implemented) or prompt the user to log in again.

1. **Security for Sensitive Data**:

* In a flight booking app, sensitive information (like payment details, booking confirmations, and personal data) is exchanged. JWT’s secure signature (using algorithms like HMAC or RSA) ensures that token data is tamper-proof.
* Only a valid JWT signed by the server allows access to protected endpoints, adding a layer of security for customer information.

**Implementations Done:**

Developing a flight booking app involves implementing various features and functionalities across the frontend, backend, database, and security layers. Here are key implementations often included in a flight booking application:

**1. User Authentication and Authorization**

* **User Registration and Login**: Implement user registration, login, and password reset functionalities.
* **Role-Based Access Control**: Different roles (e.g., passengers, admins, and airline staff) are created to restrict access to certain sections of the app.
* **JWT (JSON Web Token) Authentication**: For secure, stateless user sessions and to protect routes with authentication requirements.

**2. Flight Search and Filtering**

* **Search Flights**: Allow users to search for flights by entering departure and destination airports, dates, and the number of passengers.
* **Filter and Sort Options**: Implement filters for direct/indirect flights, airlines, departure/arrival times, and sorting options based on price, duration, or convenience.
* **Dynamic Pricing**: Integrate pricing data from different sources or APIs, as ticket prices may fluctuate based on demand, season, or other factors.

**3. Booking and Payment System**

* **Flight Booking**: Implement booking forms where users can input passenger details, seat preferences, and other information.
* **Seat Selection**: Display seating arrangements and allow passengers to choose seats (if applicable).
* **Payment Integration**: Integrate secure payment gateways (e.g., Stripe, PayPal) to process transactions.
* **Booking Confirmation and Email Notification**: After successful payment, generate a booking confirmation and send it via email, possibly including an e-ticket.

**4. Flight Management for Admins**

* **Flight Schedules and Availability**: Admins should be able to manage flight schedules, add new flights, and update availability.
* **Pricing Management**: Allow admins to adjust pricing, apply discounts, or manage seat classes.
* **User Management**: Admin functionality for managing user accounts, handling customer service inquiries, and reviewing booking history.

**5. Database Design and Management**

* **Entity-Relationship (ER) Diagram**: Design an ER diagram representing key entities like Users, Flights, Bookings, Payments, and Notifications.
* **Database Indexing and Optimization**: Optimize the database with indexing and caching for quick access to frequently queried data, such as flight schedules and availability.
* **Data Backup and Security**: Regularly back up the database and ensure data security through encryption, access control, and periodic audits.

**6. Backend Development and APIs**

* **RESTful API Development**: Build APIs for handling core functionalities like flight search, booking, user management, and payments.
* **Data Validation and Error Handling**: Implement robust validation checks for incoming data and manage errors gracefully.
* **Scalability and Load Balancing**: Design the backend to handle high traffic volumes, especially during peak travel seasons.

**8. Testing and Deployment**

* **Automated and Manual Testing**: Perform unit testing, integration testing, and end-to-end testing to ensure stability.
* **Deployment Pipeline**: Set up CI/CD (Continuous Integration and Continuous Deployment) pipelines to streamline testing and deployment.
* **Monitoring and Logging**: Integrate monitoring (e.g., using tools like New Relic or Data Dog) to track application performance and error logs.

**Installation Commands:**

The installation commands for the application focuses on how to run the application in terminal and how to generate outputs.

Our project is basically divided into three main types:

**1. Frontend part:**

The very first part focuses on generating the **User Interface (UI)** which focuses on how the user interacts with the web page.

**2. Backend part:**

The next part is the server part which focuses on handling all the responses from the server side on the requests given by the user.

**3. Database part:**

The last part is something which focuses on handling the database. This is the part where the stored data in Database is fetched by the **Application Programming Interface (API)** for enhancing the overall backend part.

**Installation command part for Frontend:**

The frontend part usually focuses on enhancing the User Interface part and then this also analyses how well the application is developed for the user to enhance the user interactions and to update changes on the way they interact.

We have developed the application using **React** which focuses on **enhancing the overall user experience.**

Apart from that we have used **JSX (JavaScript XML)** which focuses on providing the **best User Interface UI experience** for the user who are using our application.

The commands we have used for the installation of Frontend part include the following:

We have created a folder **Flight-Booking-App-MERN-main**. This includes two main sub folders within it which includes **the client folder** and **the server folder.**

**The Client folder:** This was created to update the installations of all the **Frontend part.**

**The server folder:** This was created to update all the **backend and the database part** of the overall application.

The very first step is to direct to the respective folder.



Once when it is done, we need to redirect to the client folder.



**Command 1:** This focuses on installation of **React Vite**.

npm install vite@latest ./

**Command 2:** This focuses on installation of **all the dependencies** in the frontend part.

npm install @testing-library/jest-dom @testing-library/react @testing-library/user-event axios bootstrap react react-dom react-router-dom react-scripts vite web-vitals

**Command 3:** This focuses on running the client part.

npm start

**Installation command part for Backend:**

Command 1: This focuses on verifying the installations of all commands

npm init -y

Command 2: This focuses on installation of all the necessary packages for the server to run efficiently.

npm install bcrypt body-parser cors express mongoose

Command 3: This focuses on installation of nodemon.

npm install nodemon -g

Command 4: This focuses on providing the successful connection of database.

nodemon index.js

**Installation command part for Database:**

The database part focuses on connecting the database to the server.

For this, we have focused on connecting the MongoDB Atlas cluster Connection string into the MongoDB compass. Alternatively, we have also focused on installation of **Mongodb for VS code** and then, we have entered our connection string on connecting to the database.

**WORKFLOW DIAGRAM:**

The very first thing our users do is to get signed in to our Web page on directing directly to the Login page.

Once when this is done, then the next part is to deal with Registering to the Sign up page. This sign up page is open to

**Admin**

**Flight Operator**

**Customer.**

**Role of Admin:**

The admin is the one who focuses on managing the booked and cancelled tickets.

**Role of Flight operator:**

The Flight operator is the one who ensures on the number of successful bookings and the number of failed bookings.

**Role of the Customer:**

The Customer is the one who books the tickets and cancels those when they don’t need it.

**The workflow of both User and the Admin are given below:**

**WORK FLOW OF THE USER:**

**STEP 1:** Create their account.

**STEP 2:** Search for the destination.

**STEP 3:** Search for flights as per his time convenience.

**STEP 4:** Book a flight with a particular seat.

**STEP 5:** Make payment.

**STEP 6:** To cancel bookings.

**WORKFLOW OF THE ADMIN:**

**STEP1:** Manages all bookings.

**STEP 2:** Adds new flights and services.

**STEP 3:** Monitor User activities.

**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:

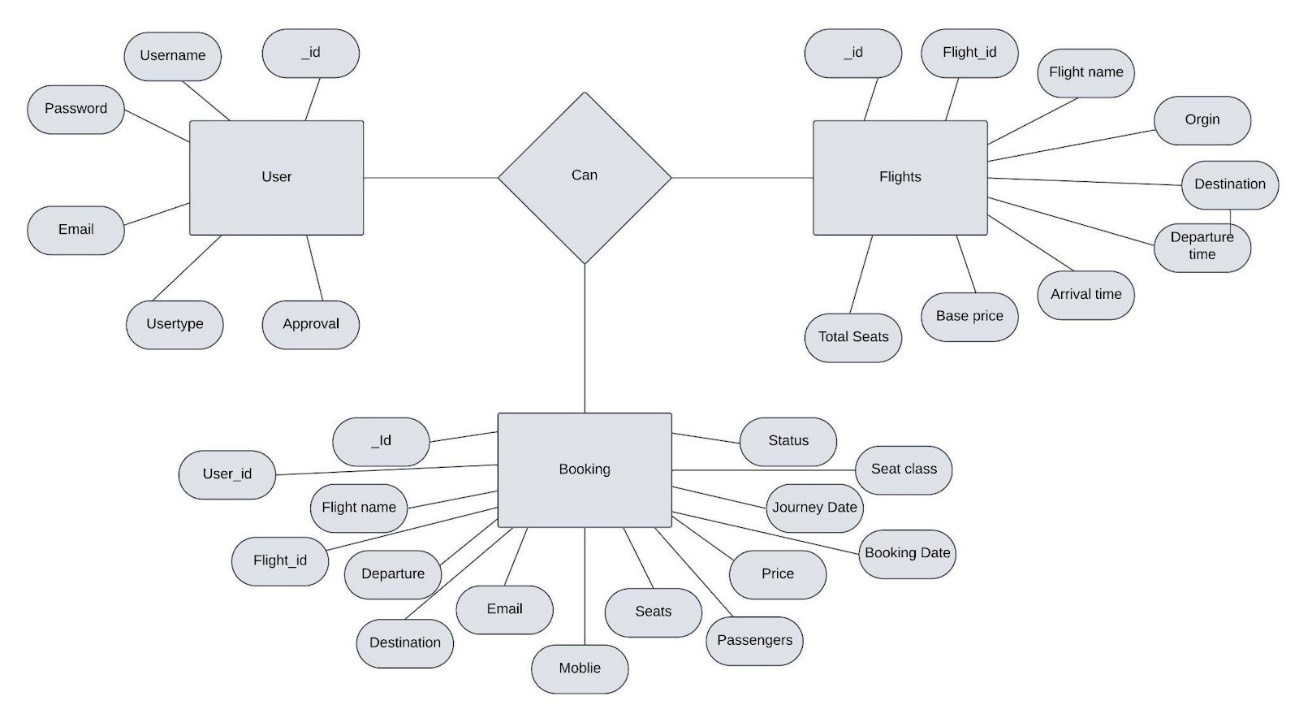
**USER:** Represents the individuals or entities who book flights. A customer can place multiple bookings and make multiple payments.

**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.

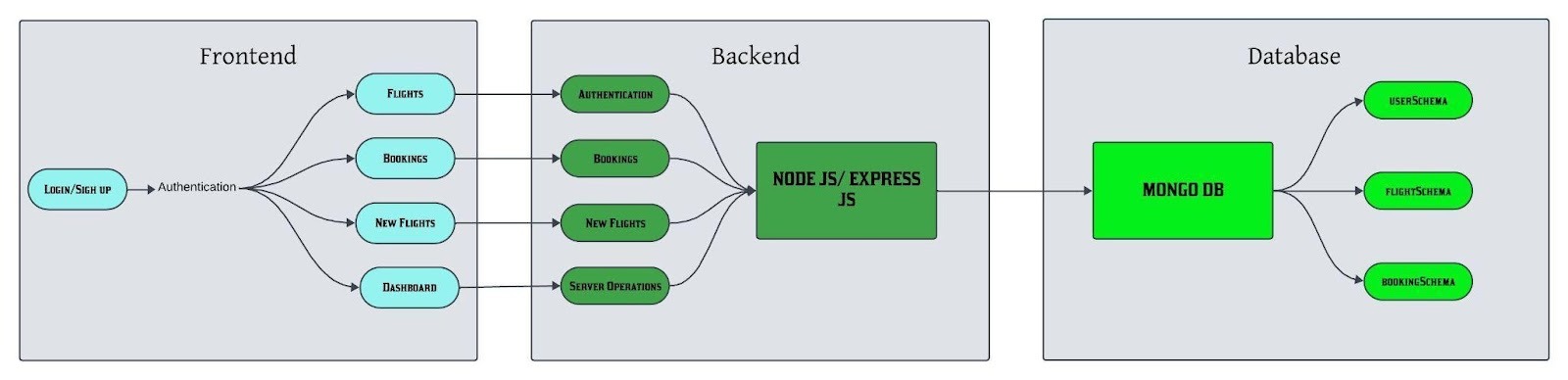
**THE ER DIAGRAM FOR OUR APPLICATION HAS BEEN ILLUSTRATED BELOW:**

****

**APPLICATION FLOW:**

This is a simpler demonstration of how we have used our Frontend, Backend and the Database part for building our entire application.

**THE APPLICATION FLOW HAS BEEN ILLUSTRATED BELOW:**

****

The user is capable of accessing the Dashboard of the application, they can sign in or login to that particular application.

All the available flights, the booking and the new flights are illustrated at the frontend part of the website.

In the Backend part, the bookings done by the user, the new flights, the server operation and all the authentication functionalities are handled.

The database focuses on storing the respective user data, the flights data and finally the bookings of how it is done.

**Architecture of our application:**

**1. Model (MongoDB Schema):**

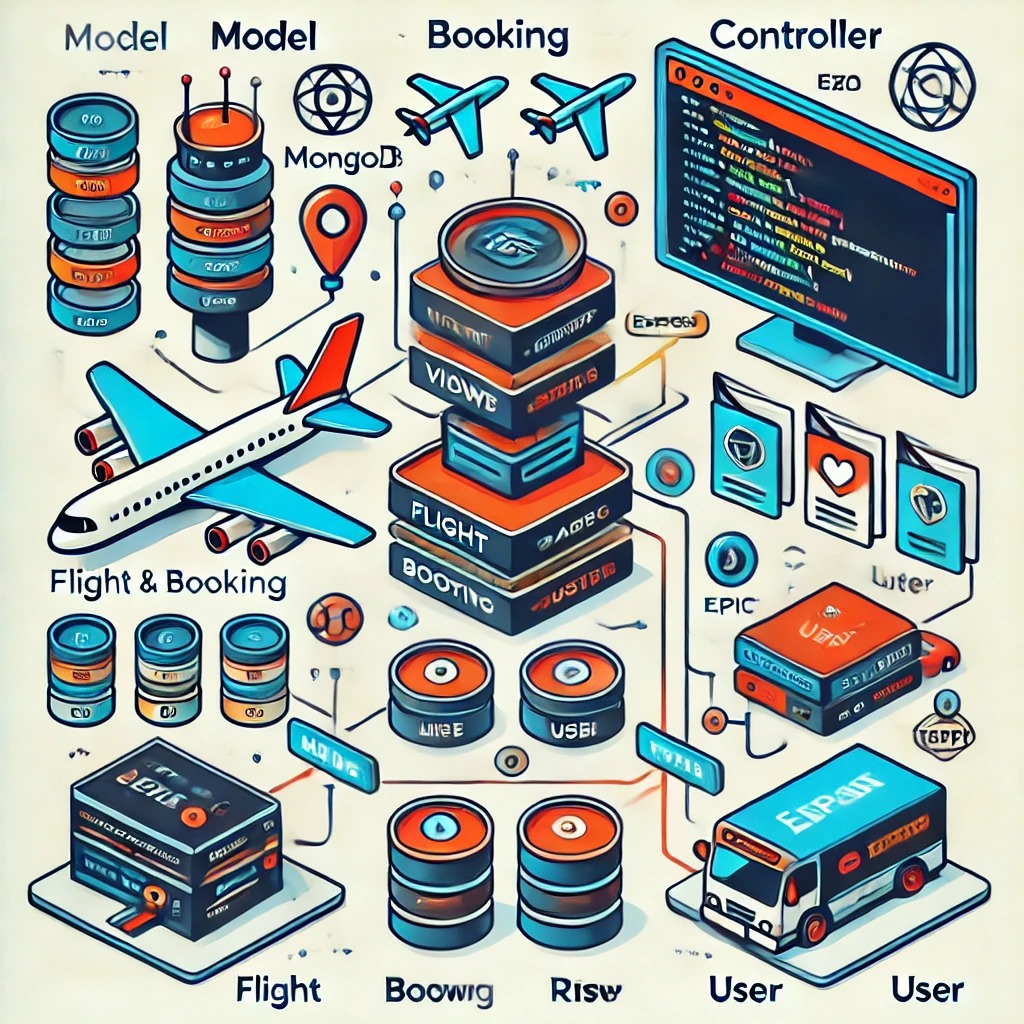
The Model represents the data structure and the business logic of the application. In MongoDB, models are represented using schemas with Mongoose.

**2. View (Frontend - React):**

The view will be responsible for rendering the UI. For the flight booking system, you will have various components such as flight listings, booking forms, and user profiles.

**3.Controller (Backend - Express):**

The controller handles requests, interacts with models, and returns the response.



**TECHNOLOGY STACKS WE HAVE USED:**

**The frontend part:**

Frontend refers to the part of a web application or website that users interact with directly. It encompasses everything that users experience visually and interact with, including the design, layout, and the elements of a webpage. The frontend is responsible for how the content is presented to the user and how users can interact with the application.

**Why we have used React?**

React is a popular JavaScript library for building user interfaces, primarily for single-page applications where you need a fast and dynamic user experience. Developed by Facebook, React allows developers to create reusable UI components and manage the state of an application in an efficient way. Here are some of its key features:

1. One way data binding.
2. Virtual DOM.
3. Component-based architecture.
4. Performance Optimization.

**The backend part:**

The backend is the part of an application that operates behind the scenes, managing how the system works and handling data storage, security, and business logic. It acts as the intermediary between the frontend (what users see) and the database (where data is stored), and it ensures the application functions smoothly and securely.

**Why we have used Node js?**

Node.js is an open-source, cross-platform JavaScript runtime environment that enables developers to build server-side applications using JavaScript. Built on Chrome’s V8 JavaScript engine, Node.js allows JavaScript to run on the server side, extending its use beyond client-side scripting. Node.js is known for its non-blocking, event-driven architecture, which makes it highly efficient and ideal for building scalable applications.

**The database part:**

A database plays a central role in any application, particularly in systems that need to store, retrieve, update, and manage data effectively. In applications like a flight booking system or e-commerce platform, the database acts as the backbone, ensuring data integrity, accessibility, and security. Here's a breakdown of the database's role in applications:

1. Data Storage and Management
2. Data Retrieval and Querying
3. Data Consistency and Integrity
4. Data Scalability and Availability
5. Data Security and Access Control

**The API part:**

The main function of an API (Application Programming Interface) is to facilitate communication between different software applications, enabling them to share data and functionality without needing to know each other’s inner workings. APIs define a set of rules, protocols, and tools that allow applications, servers, and services to interact seamlessly, enhancing flexibility and integration.

**1. Data Access and Communication**

* APIs allow applications to request and retrieve data from other services, acting as a bridge for data exchange. For example, a weather app can request data from a weather API to show forecasts.
* **REST APIs** (most common) follow HTTP protocols, sending data in formats like JSON or XML, making it easy to read and process on different platforms.

**2. Functionality Exposure**

* APIs allow applications to use each other’s functions or features. For instance, a flight booking app can use an API to access an airline's flight search and booking functions.
* By exposing only necessary functions, APIs let applications access a service's capabilities without exposing the underlying code or infrastructure.

**3. Interoperability**

* APIs enable interoperability between different systems, platforms, and programming languages, allowing diverse systems to communicate. This makes APIs ideal for creating cross-platform applications, where multiple services work together in a coordinated way.

**4. Automation**

* APIs allow tasks to be automated by enabling applications to make requests to each other without manual input. For instance, a content management system (CMS) might use an API to automatically publish content to social media platforms at scheduled times.

**5. Modularity and Reusability**

* APIs promote modularity by letting developers access specific functions or data without rebuilding these from scratch. This approach saves development time and improves code reuse, as functionalities are accessed and used as components.

**6. Enhanced User Experience (UX)**

* APIs enable real-time data updates and provide responsive interactions. For instance, in a flight booking application, APIs can retrieve real-time flight data and availability, improving user satisfaction by presenting current information instantly.

**7. Security and Controlled Access**

* APIs act as a layer between the application and data, often requiring authentication (like API keys, OAuth) to secure data. This controlled access ensures only authorized users or applications can interact with the API and access sensitive information.

**FRONTEND FUNCTIONALITY:**

User Interface (UI):

Search Form: Users can enter information such as the departure and destination cities, dates, number of passengers, and class (economy, business, etc.).

Flight Listings: After searching, users see a list of available flights with details like flight numbers, departure times, prices, and seat availability.

Filters and Sorting: Users can filter the results by airline, price, departure time, or other preferences, and sort them based on their criteria.

Booking Form: Once a user selects a flight, they are prompted to fill out personal information, payment details, and other necessary details to complete the booking.

Design and Layout:

CSS/Styling: The look and feel of the application, including fonts, colors, layout, and responsiveness across devices, would be created using CSS or CSS frameworks like Bootstrap (which you're using).

Interactive Elements: Buttons, input fields, date pickers, dropdown menus, and modal windows (like showing flight details) would be designed to be interactive using JavaScript.

JavaScript & Frameworks:

React (or other frameworks): If you're using React or similar frameworks, they would handle the dynamic loading of flight data, form validation, user input handling, and making requests to the backend to fetch or submit data.

AJAX Requests: When a user submits a search or booking form, AJAX or fetch API calls would be made to the backend to get flight data or store the booking details without reloading the page.

User Interaction:

Real-time Updates: If the application shows real-time flight availability or updates on booking status, it might use WebSockets or polling to receive live data from the backend without needing to refresh the page.

Confirmation Screens: After a successful booking, users would be shown a confirmation screen or email with flight details, which is all handled on the frontend.

**BACKEND FUNCTIONALITY:**

1. User Authentication and Authorization

Sign Up / Log In: The backend manages user authentication (login and sign-up processes) using secure methods like password hashing and token-based systems (JWT).

User Sessions: It maintains user sessions to track login states across different requests.

Authorization: Once logged in, the backend ensures users can only access data or perform actions they are authorized for, such as booking a flight, managing bookings, or viewing their history.

2. Search and Display Flights

Flight Search: The backend handles the core search logic. When a user enters their search criteria (e.g., departure city, destination, dates), the backend queries the database to find available flights.

Filters and Sorting: After retrieving the results, the backend processes filters like price, airlines, or departure times, and sorts them accordingly before sending them back to the frontend.

Real-Time Updates: Flight availability and pricing can change, so the backend regularly updates the frontend with the latest data.

3. Flight Booking Process

Selecting Flights: After a user selects their desired flights, the backend prepares to manage the booking. It checks availability in real-time to ensure the flights are still available.

Seat Reservation: The backend may lock seats for a short period to prevent overbooking. This typically involves checking seat availability and managing inventory.

Payment Integration: The backend integrates with third-party payment gateways (like Stripe or PayPal) to process payments. It also ensures the transaction is secure and handles any payment failures or errors.

4. Booking Management

Create Booking: Once payment is confirmed, the backend creates a booking in the database and generates booking references.

Cancellation/Modification: The backend allows users to cancel or modify their bookings, updating both the flight inventory and user data accordingly.

Ticket Generation: The backend generates the flight ticket details and sends them to the user, either via email or in the user’s account.

5. User Profile and History

User Profile Management: The backend allows users to update their personal information (e.g., name, contact details, passport info).

Booking History: The backend maintains a history of all past bookings for each user, enabling them to view and track previous flight details.

6. Admin Panel (Optional)

Flight Management: Admins can add, update, or delete flights from the database (e.g., adding new routes, setting prices, or managing seat counts).

User Management: Admins can manage user accounts, view their activity, and handle support-related tasks.

7. Notifications and Email System

Confirmation Emails: After a successful booking, the backend sends confirmation emails with flight details and ticket numbers.

Reminders: The backend may send reminder emails or SMS notifications for upcoming flights, changes, or cancellations.

8. Analytics and Reporting

Flight Popularity and Analytics: The backend can aggregate data to identify trends, such as most popular routes, peak travel times, or revenue generation.

Admin Reports: Provide reports on booking status, revenue, cancellations, or customer behaviour to assist with business decisions.

**DATABASE FUNCTIONALITY:**

In a flight booking application, the database is central to managing and storing all the essential data required to ensure smooth operations. The database handles everything from user profiles to flight schedules, bookings, payments, and even user history. Here's a breakdown of the core database functionality:

1. Database Structure

The database is typically organized using multiple tables (or collections in MongoDB, since you're using the MERN stack) to represent different entities within the system. These entities include:

Users

Flights

Bookings

Payments

Flight Schedules

Airlines

Airport Codes

Notifications

Each of these entities can be represented as a document (in MongoDB) or table (in SQL-based systems). They are all interconnected, allowing the system to maintain relationships between the data.

2. Core Database Entities

Users

The Users collection/table stores information about customers who use the flight booking system.

Attributes:

user\_id (unique identifier)

email, password\_hash (for authentication)

name, phone, address (contact details)

passport\_number (if applicable)

booking\_history (an array of booking references or IDs)

Functionality:

Enables user authentication (login/signup).

Stores user profile data and preferences.

Manages the user's booking history, which can be referenced later.

Flights

The Flights collection/table holds information about available flights.

Attributes:

flight\_id (unique identifier)

departure\_airport, destination\_airport (airport codes or names)

departure\_time, arrival\_time

price, currency

seats\_available (number of remaining seats)

airline\_id (foreign key to Airlines table)

Functionality:

Stores flight schedules, including departure and arrival details.

Tracks flight availability, prices, and the number of seats available.

Can be updated by the admin when flight schedules change.

Bookings

The Bookings collection/table manages flight reservations made by users.

Attributes:

booking\_id (unique identifier)

user\_id (foreign key to Users table)

flight\_id (foreign key to Flights table)

booking\_date

payment\_status (e.g., pending, completed, failed)

passenger\_details (name, passport number, contact details)

seat\_number (assigned seat)

Functionality:

Stores booking records once a user confirms a reservation.

Links users to the flights they’ve booked.

Tracks booking status (confirmed, cancelled, etc.).

Allows for modifications or cancellations of bookings.

Generates booking confirmation with flight details and ticket number.

Payments

The Payments collection/table handles all payment-related transactions.

Attributes:

payment\_id (unique identifier)

booking\_id (foreign key to Bookings table)

user\_id (foreign key to Users table)

payment\_method (e.g., credit card, PayPal)

amount

status (pending, completed, failed)

payment\_date

Functionality:

Records payment transactions related to bookings.

Stores payment statuses, such as successful payments or failed attempts.

Enables easy tracking of user payments for refunds or disputes.

Airlines

The Airlines collection/table holds information about the airlines operating the flights.

Attributes:

airline\_id (unique identifier)

name (airline name)

code (IATA or ICAO code)

contact\_info (website, phone, email)

Functionality:

Tracks which airline operates each flight.

Helps in organizing flights by the airline for search and display purposes.

Flight Schedules

The FlightSchedules collection/table stores specific details related to scheduled flights, which could vary based on time.

Attributes:

schedule\_id (unique identifier)

flight\_id (foreign key to Flights table)

departure\_date, departure\_time

arrival\_date, arrival\_time

seat\_capacity

Functionality:

Allows tracking of specific instances of flights that occur at different times (e.g., same flight but different dates).

Enables filtering of available flights by schedule.

Airport Codes

The AirportCodes collection/table stores information about airports.

Attributes:

airport\_code (unique identifier, e.g., JFK, LHR)

airport\_name

city, country

Functionality:

Provides a quick reference for airport names and codes for flights.

Helps with flight search by airports.

Notifications

The Notifications collection/table holds notifications sent to users.

Attributes:

Notification \_id (unique identifier)

User \_id (foreign key to Users table)

message

notification \_type (e.g., flight reminder, booking confirmation)

status (read/unread)

Functionality:

Tracks notifications sent to users, such as booking confirmations, reminders, or cancellations.

Manages the read/unread status of notifications.

3. Database Relationships

In a flight booking application, certain entities are related to each other. In MongoDB, this would be handled using references (via Object Id) or embedded documents. Here are some key relationships:

User → Bookings: A user can have multiple bookings, so the Bookings collection has a user \_id field referencing the Users collection.

Booking → Flight: A booking is linked to a specific flight through the flight \_id field.

Flight → Airline: Each flight is associated with one airline through the airline \_id field.

Flight → Airport Codes: A flight will have a departure and destination airport, linked to the Airport Codes collection.

4. Database Operations:

CRUD Operations: The system performs basic CRUD (Create, Read, Update, Delete) operations to manage flights, bookings, users, and payments.

Create: A user can create a booking, which is added to the Bookings collection. Similarly, a flight can be added by an admin.

Read: Flight search queries retrieve relevant flight details, user data, or booking information.

Update: Flight schedules or bookings can be modified, such as updating seat availability or cancelling a booking.

Delete: Admins or users may delete a booking or cancel a flight.

Search Queries: The backend executes queries to search for flights based on user preferences (e.g., destination, dates, price range) by querying the Flights collection with relevant filters.

Aggregation: MongoDB allows aggregation pipelines to compute statistics or summarize data, like fetching the most popular flights or calculating the total number of bookings for a particular route.

5. Real-Time Data Handling:

In a dynamic environment like flight booking, real-time data handling is essential. For example, flight availability or prices may change frequently, and the database needs to reflect these changes immediately.

Stock Management: When a user books a flight, the backend updates the seats \_available field of the corresponding flight in the database.

Caching: For high-demand data, such as popular flight routes, caching mechanisms (e.g., Redis) can help reduce database load and speed up retrieval.

6. Data Security and Integrity

Data Validation: Ensuring data integrity by validating inputs (e.g., correct email format, valid passport number) before storing them.

Transactional Integrity: Using transactions (or MongoDB’s write concern) to ensure that operations like booking a flight and processing payments are completed successfully or rolled back in case of failure.

7. Performance Optimization:

Indexing: Key fields (e.g., flight \_id, user \_id, departure \_time) should be indexed for faster query performance, especially when dealing with large datasets.

Sharding: If the database grows large, sharding can be used to distribute data across multiple servers for scalability.

**IMPORTANCE OF API AND ITS FUNCTIONALITY:**

Importance of REST API:

Standardized Communication: REST APIs use standard HTTP methods (GET, POST, PUT, DELETE) for communication, which makes them easy to understand and implement.

Statelessness: REST APIs are stateless, meaning each request from a client to the server must contain all the information the server needs to process that request. This simplifies server design and makes it easier to scale.

Separation of Concerns: REST APIs separate the client and server, allowing the backend logic to evolve independently of the frontend user interface.

Scalability: REST APIs can easily scale because they can handle large amounts of traffic by distributing requests across multiple servers.

Interoperability: REST APIs allow different systems, written in various programming languages, to communicate with each other. This is especially important in a microservices architecture where multiple services need to interact.

Mobile and Web Integration: REST APIs are commonly used in mobile applications and single-page web applications (SPAs), enabling them to communicate efficiently with backend servers and databases.

Caching: REST supports caching of responses, reducing server load and improving performance.

**IMPORTANCE OF JSON WEB TOKEN:**

JSON Web Token (JWT) is an open standard for securely transmitting information between parties as a JSON object. It is widely used in authentication and authorization systems. Below are the key reasons why JWT is important:

1. Stateless Authentication:

No Server-Side Sessions: JWT enables stateless authentication, meaning the server doesn’t need to store session data. Once a JWT is issued, the client holds the token, and each subsequent request is authenticated by verifying the token. This reduces server load and improves scalability because no session data needs to be stored on the server.

2. Secure Information Exchange:

Data Integrity: JWTs are signed using a secret key (HS256) or a public/private key pair (RS256). This ensures that the data has not been tampered with during transit. The recipient can verify the integrity of the token by checking its signature.

Confidentiality (Optional): JWTs can be encrypted to ensure confidentiality. While the token is typically signed for verification, it can also be encrypted (using JWE - JSON Web Encryption) to prevent unauthorized parties from reading the data.

3. Compact and URL-Safe:

Compact: JWTs are compact, making them easy to send over the network as part of HTTP headers, URLs, or even in cookies. This compact size makes JWTs efficient in scenarios like REST API calls.

URL-Safe: The JWT format is URL-safe, meaning it can be transmitted easily over HTTP as part of a URL (e.g., as a query parameter or in a header).

4. Flexibility:

Custom Claims: JWTs can include custom claims, allowing you to store additional data in the payload. For example, you can add user roles, permissions, or metadata about the user to customize the authentication and authorization processes.

Expiration Control: JWTs can include an expiration date (exp) in the payload, allowing tokens to automatically expire after a specified period. This ensures that a compromised token cannot be used indefinitely.

5. Cross-Domain Authentication:

Cross-Origin Resource Sharing (CORS): JWTs are particularly useful in scenarios where authentication is required across different domains or microservices. Since the token is passed in HTTP headers, it allows for single sign-on (SSO) or decentralized authentication systems that work across different services.

6. Support for Authorization:

Role-Based Access Control (RBAC): JWTs can be used for authorization by including user roles or permissions in the payload. This helps control access to specific resources or endpoints based on the claims within the token, enabling fine-grained control over access.

7. Improved Security with Short-Lived Tokens:

Short-Lived Access Tokens: JWTs can be issued with short expiration times for enhanced security. In case a token is compromised, it will only be valid for a limited time. A refresh token can be used to issue new access tokens without requiring the user to log in again.

8. Widely Adopted and Supported:

Industry Standard: JWT is widely adopted and supported across most modern web frameworks, libraries, and services. This makes it easy to implement across multiple platforms, both on the client and server side.

9. Single Point of Authentication:

No Need for Repeated Authentication: With JWT, after a user logs in and receives a token, they don't need to repeatedly authenticate with the server. This is particularly useful in single-page applications (SPAs) and mobile apps, where maintaining state on the server is inefficient.

Key Components of a JWT:

Header: Contains the metadata about the token, such as the signing algorithm (e.g., HS256 or RS256).

Payload: Contains the claims (information) like user ID, roles, and expiration time. This can be any data you want to transmit.

Signature: Ensures the token has not been altered. It’s created by combining the encoded header, payload, and a secret key (or a private key for asymmetric encryption).

Example Use Cases of JWT:

User Authentication: After a user logs in, a JWT can be issued and used to authenticate subsequent requests to secure endpoints.

API Security: APIs use JWTs to authenticate and authorize users, ensuring that only authorized users can access specific resources or perform certain actions.

Single Sign-On (SSO): JWTs are often used in SSO systems, where a user can log in once and access different services without re-authenticating.

**IMPORTANCE OF GIT AND GIT HUB:**

Git and GitHub are essential tools for developing a flight booking application, as they streamline collaboration, version control, and deployment processes. Here's how they contribute:

1. Version Control with Git: Git allows you to track changes in your codebase over time. For a complex project like a flight booking app, multiple developers may work on different parts (frontend, backend, etc.). With Git, you can:

Save snapshots of your code at various stages, making it easy to revert to previous versions if issues arise.

Branch off to develop new features or fix bugs without affecting the main codebase. Once these changes are tested, you can merge them back safely.

2. Collaborative Development on GitHub: GitHub, as a remote repository and collaboration platform, offers essential tools for teamwork:

Pull Requests: These allow developers to propose changes, review code, and discuss modifications before merging. This ensures code quality and helps detect issues early.

Issue Tracking: GitHub provides issue tracking, allowing you to document bugs, features, or enhancements. It’s helpful for organizing tasks, especially in a large application with multiple components.

CI/CD Integration: GitHub integrates well with CI/CD pipelines, enabling automated testing and deployment. This can help ensure each new feature or fix for the booking app is tested and deployed without manual intervention.

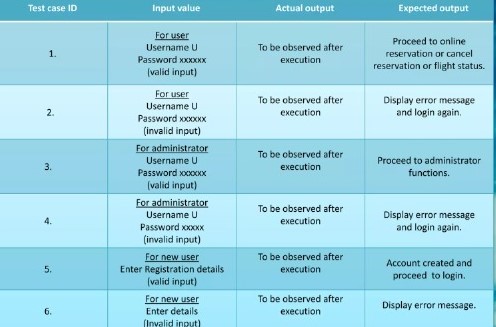
3. Documentation and Transparency: GitHub's README and Wiki sections can be used for documentation. This helps developers understand the architecture, setup, and usage of the app, and aids in the onboarding of new developers.

4. Backup and Access Control: Storing code on GitHub acts as a secure backup. Additionally, you can control who has access to your repositories, making it easy to manage contributors.

For a flight booking app, these features ensure you can maintain an organized, efficient development process, even as the codebase grows and the team scales.

**TEST CASES:**

We have generated the following test cases based on the observations



**CODING PART:**

Before moving on to the coding part, let’ s explore our project directory.

The project directory is divided into two main folders: The first one is **Frontend part** which is the **Client** and the **Backend part** which is the **Server.**

****

The next one is to explore package.json file in each folder(for both the client and the server folder).

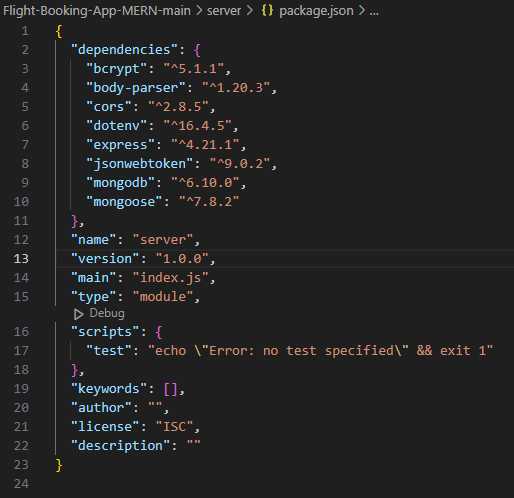
**What’s package.json file?**

This is a configuration file used in Node JS. This file is basically used to manage the dependencies, metadata and scripts.

**The package.json for frontend is illustrated below:**

**** ****

**The package.json for backend is demonstrated below:**

****

**The installation commands for front end include the following:**

**STEP1:** To redirect to the particular folder.





**STEP 2:** To install **React vite**.



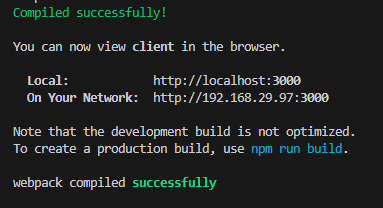
After entering this command we encountered some errors and we actually focused on clearing those by providing the **npm audit fix** and we have reduced the vulnerabilities.

**STEP 3:** To install the necessary dependencies:



**STEP 4:** To run the **Client** part:

Once when we have provided the **npm start**, we have got the following and the application started to load in the browser.



**THE BACKEND PART:**

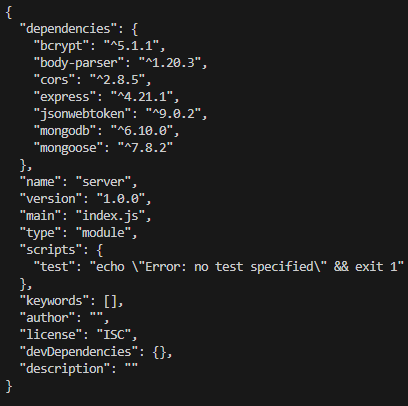
**STEP 1:** To redirect to the **server** folder.



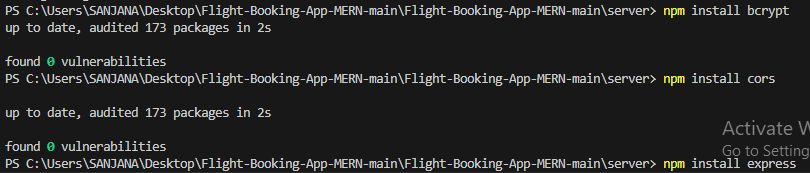
**STEP 2:** To ensure the installation of node in the server folder, we have provided the following command:

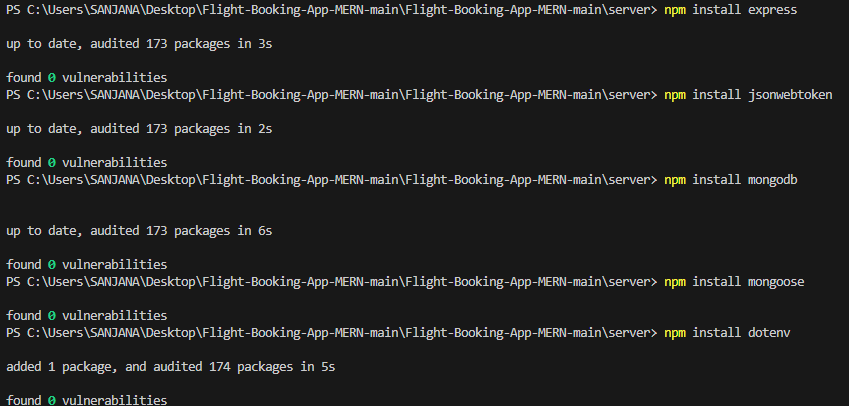


We have got the following on providing the above command:



**STEP 3:** To install the respective dependencies for the backend folder.





**STEP 4:** To install nodemon for running the server.

**What’ s nodemon?**

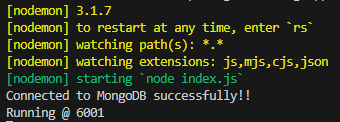
Nodemon is a utility for Node.js applications that automatically restarts the server when it detects changes in the code. This makes development more efficient by saving you the step of manually restarting the server each time you make changes to your code.



**STEP 5:** To run the server.



This generates the following output.



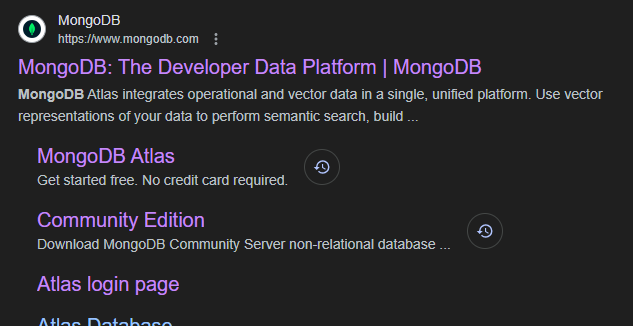
**STEP 3:** Connecting the server to the backend:

For this, we have used the **MongoDB Atlas.**

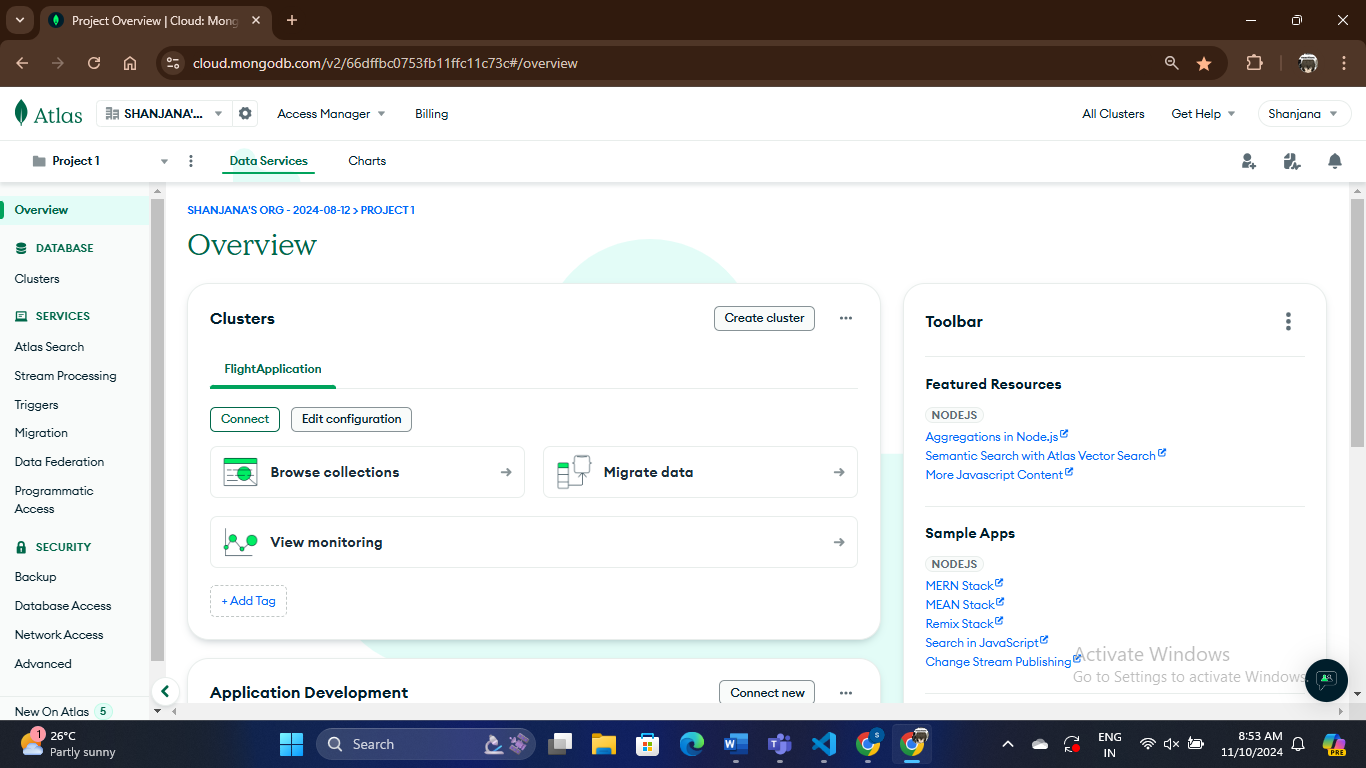
**STEP 1:** Redirect to the Google chrome and search for MONGODB.

**STEP 2:** Go to the **MongoDB developer** platform and then **Sign up** by providing your details.

The following is the **MongoDB developer** platform. Make sure that you are clicking the correct link.

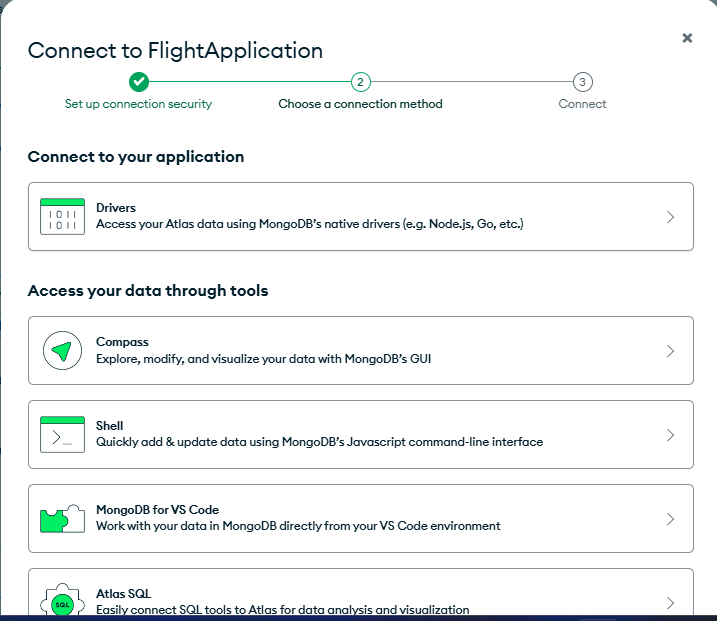


**STEP 2:** Once when you have successfully created your identity and when you sign in, you observe that you would receive a separate dashboard.



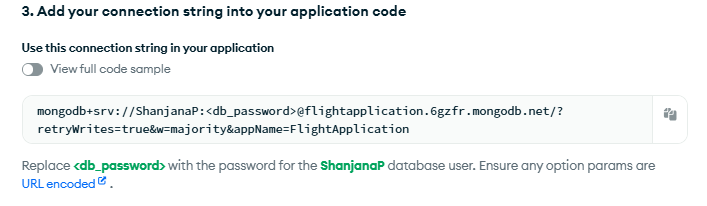
As I have already created the cluster, it was easy for me to work with the additional changes and to develop results efficiently.

**STEP 2:** Let’s connect to the drivers on clicking the connect button.

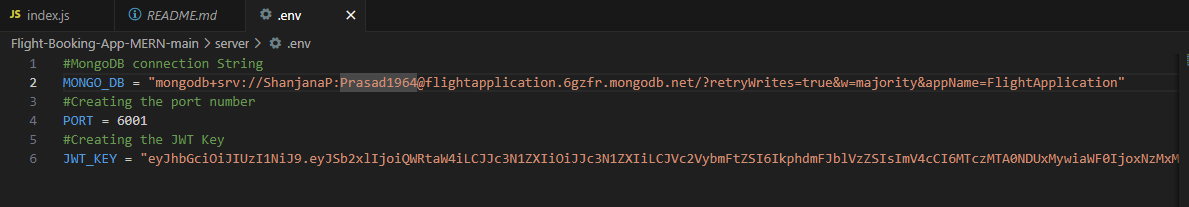


**STEP 3:** Copying the connection string.

For this make sure you are copying the connection string properly and also providing your correct password.



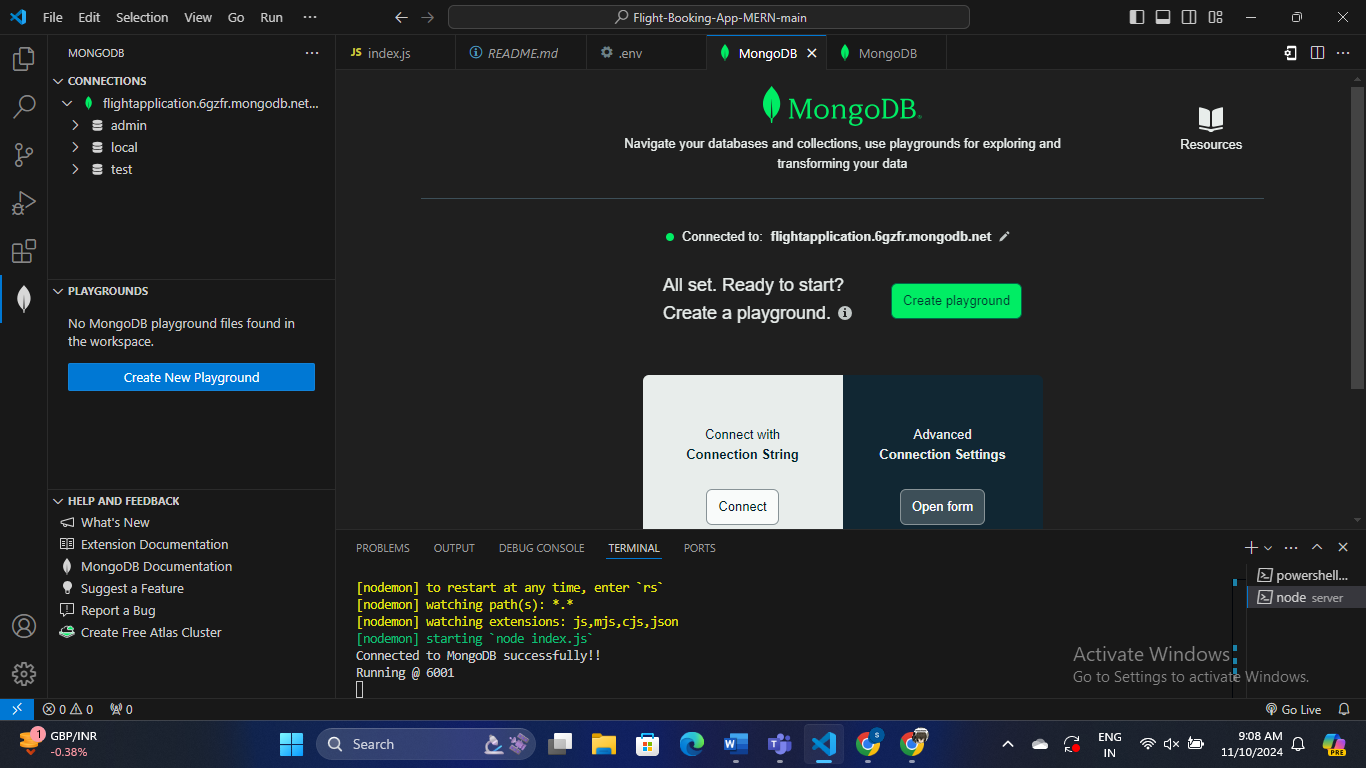
**STEP 4:** Make sure you are entering your details properly within the **.env** file which you are creating within the **server** folder.



**STEP 5:** As we have used **Extension for MongoDB** in the **VS code,** we have to provide the connection string on connecting with the **MongoDB** database.

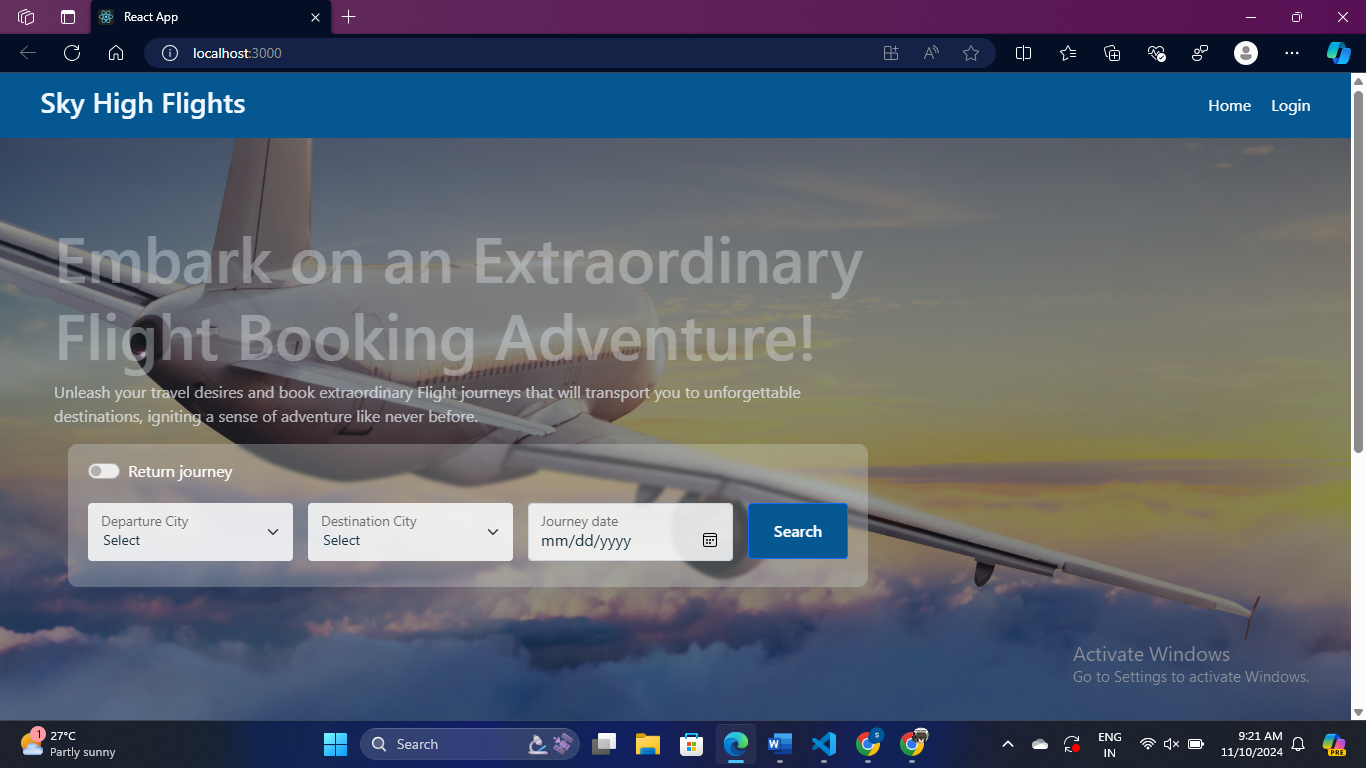




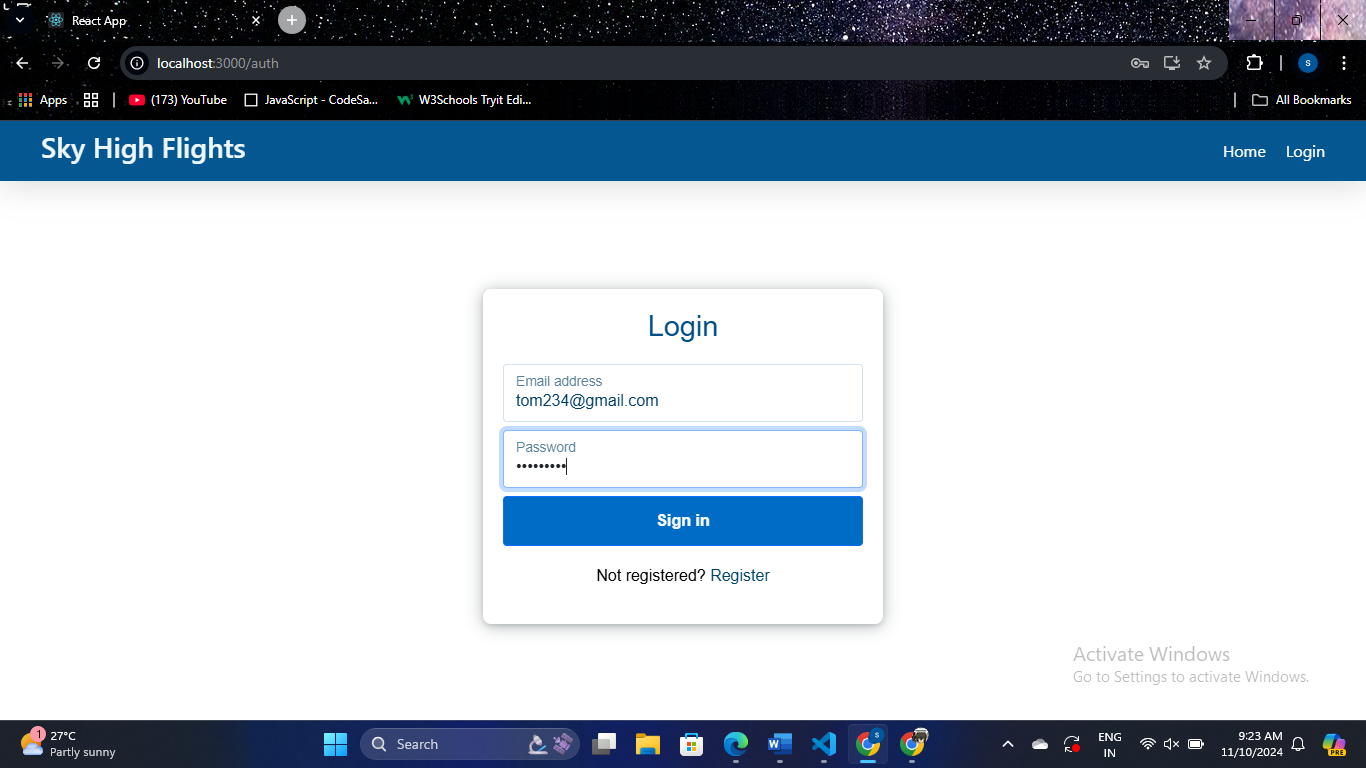


**EXECUTION PART:**

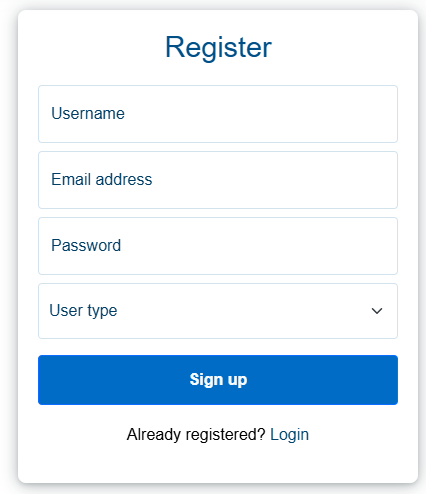
The frontend server gets started on providing the **npm start** command.



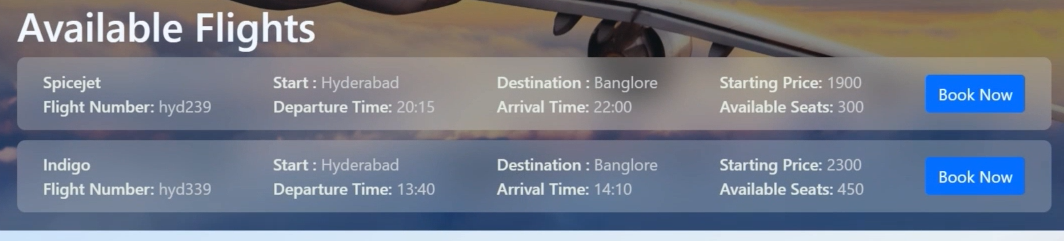
The login page will be looking like this:

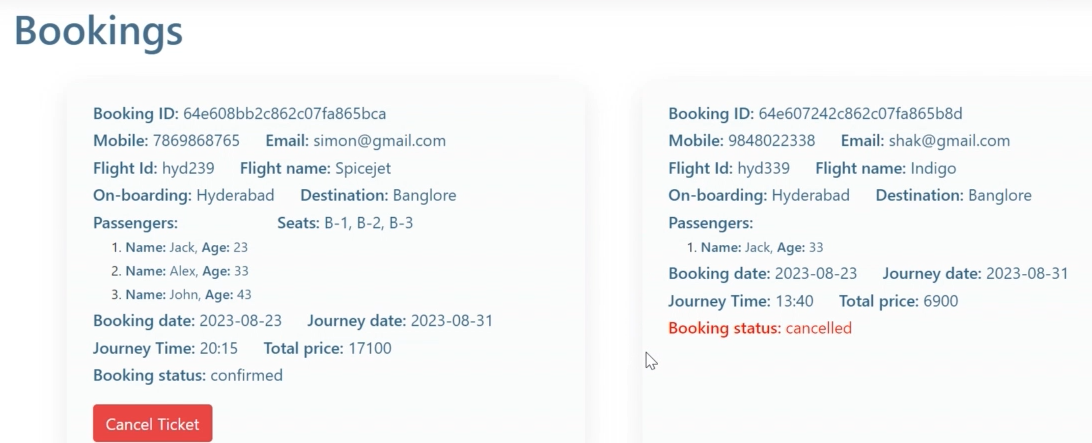


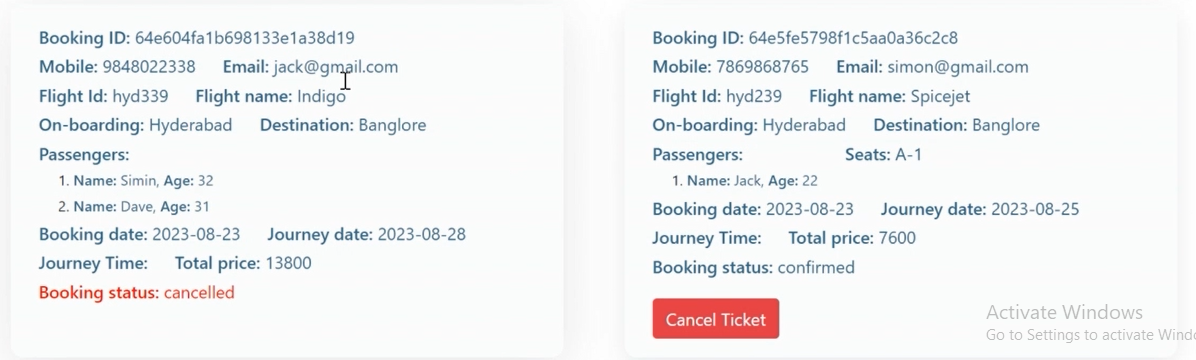
The Register page will be looking like this:

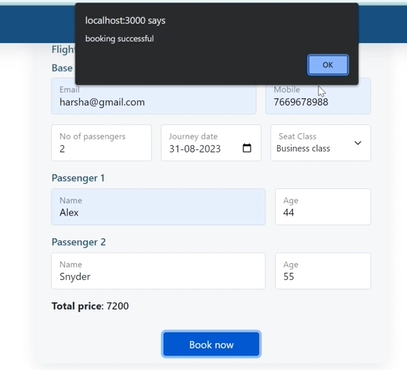
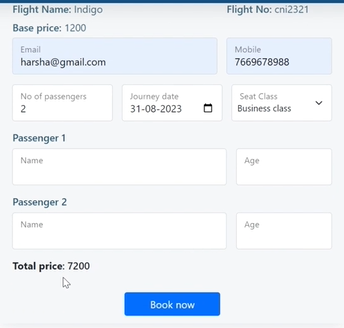


**Work of the Customer:**

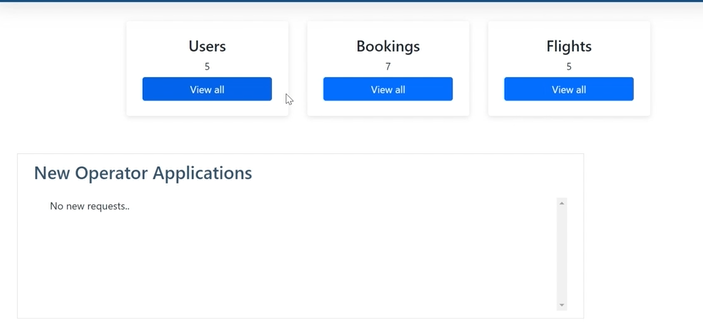


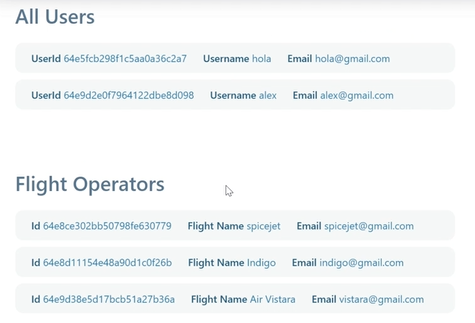


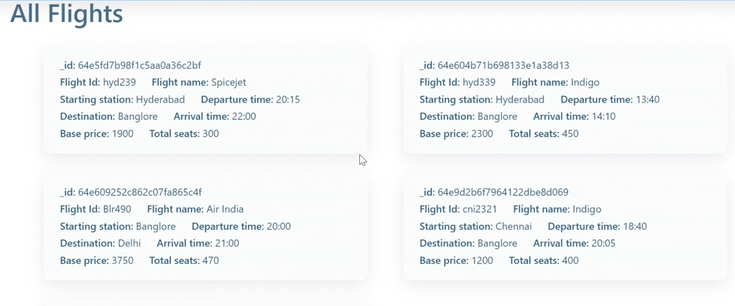




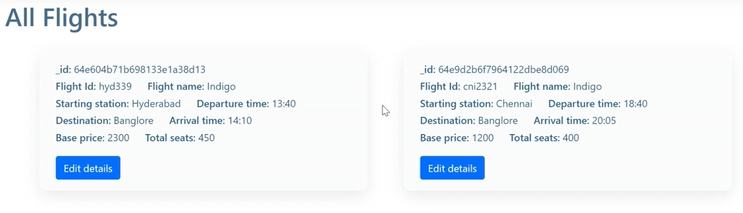
**Work of the Admin:**

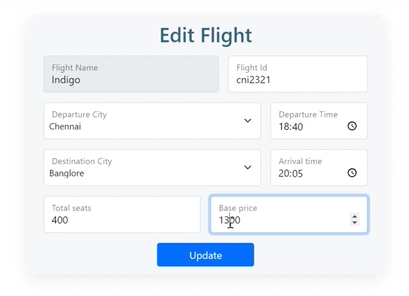
****

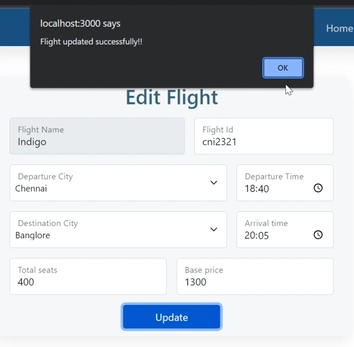
****

****

**Work of the Flight Operator:**

****

****

****

**Overall Journey for Building the Application:**

**1. Planning and Requirements Gathering**

* **Feature Identification**: Define core features like flight search, booking, user registration/login, payment, and booking management.
* **UI/UX Design**: Create wireframes and prototypes to map out the user journey. Consider mobile responsiveness, intuitive layouts, and user experience.
* **Tech Stack Decisions**: Choose the tools and libraries you’ll use, including MongoDB, Express.js, React, Node.js, and any additional libraries (e.g., Redux for state management, Axios for API calls).
* **Project Structure**: Outline the folder structure and database schema (collections for Users, Flights, Bookings, Payments, etc.).

**2. Setting Up the Development Environment**

* **Initialize Git Repository**: Set up version control for collaborative development and tracking.
* **Install Node and Express**: Set up the backend server (index.js) using Express.js, and define the port (e.g., 6001) for local development.
* **React App Initialization**: Bootstrap a new React app for the frontend with create-react-app.
* **Database Setup**: Install and set up MongoDB, either locally or using a cloud service like MongoDB Atlas. Define the initial schema for flights, users, and bookings.

**3. Backend Development (Express + MongoDB)**

* **Set Up Routes**:
  + **User Authentication**: Create routes for login, signup, and logout.
  + **Flight Search**: Develop API endpoints to handle flight searches based on criteria like destination, dates, and passenger count.
  + **Booking and Payments**: Add endpoints for booking flights and processing payments.
* **Database Models**: Create Mongoose schemas and models for the primary entities (Users, Flights, Bookings, Payments).
* **Implement Controllers**: Write controller functions for handling API requests (e.g., searchFlights, createBooking, processPayment).
* **Middleware**:
  + **Authentication**: Implement JWT or session-based authentication.
  + **Validation**: Use middleware to validate incoming requests and protect routes.
* **Testing API**: Test each API endpoint with tools like Postman to ensure correct functionality.

**4. Frontend Development (React)**

* **UI Components**:
  + **Homepage**: Create a welcoming homepage with a flight search form.
  + **Search Results Page**: Display available flights after a user searches, with filtering and sorting options.
  + **Booking Page**: Create a booking form to collect passenger and payment information.
  + **User Profile**: Allow users to view their booking history and profile details.
* **State Management**: Use Context API or Redux to manage global states such as user authentication, flight results, and bookings.
* **Integrate with Backend**: Use Axios or Fetch API to connect the React frontend to the backend API, managing routes for searching flights, handling bookings, and displaying user data.
* **Styling**: Use CSS frameworks like Bootstrap (since you’re already working with it) or custom CSS for a polished UI.

**5. Implementing Additional Features**

* **Payment Integration**: Integrate a payment gateway (such as Stripe or PayPal) for secure payment processing.
* **Notifications**: Implement email confirmations for bookings using a service like Nodemailer.
* **Admin Panel** (optional): Add an admin dashboard to manage flights, view bookings, and generate reports if needed.

**6. Testing and Debugging**

* **Unit Testing**: Write unit tests for both backend and frontend logic (using tools like Jest, Mocha, or Chai).
* **End-to-End Testing**: Use tools like Cypress to test the entire user journey, from searching flights to completing bookings.
* **Load Testing**: Simulate high traffic using tools like Apache JMeter to ensure your app can handle the expected load.

**7. Deployment and Scaling**

* **Backend Deployment**: Deploy the backend server on a platform like Heroku, DigitalOcean, or AWS. Ensure it connects with the MongoDB database securely.
* **Frontend Deployment**: Deploy the React app using services like Vercel, Netlify, or directly on AWS S3 with CloudFront.
* **Environment Variables**: Use .env files or environment settings on your hosting platforms for sensitive information (like database URIs, API keys, and secret keys).
* **Continuous Integration/Deployment (CI/CD)**: Set up CI/CD pipelines for smooth deployment of updates.

**8. Post-Deployment and Monitoring**

* **Analytics and Monitoring**: Use tools like Google Analytics, Sentry (for error monitoring), and server logs to monitor usage and diagnose issues.
* **User Feedback**: Collect user feedback to identify areas for improvement and potential feature enhancements.
* **Scaling**: If demand grows, consider scaling your database and server resources, load balancing, and using a Content Delivery Network (CDN) for faster load times.

**CONCLUSION:**

Building a flight booking application using the MERN stack is a comprehensive journey that covers planning, development, testing, and deployment. By following a structured approach, you can create a robust application that meets user needs, from simple flight searches to secure bookings and payments. Each phase, from backend API development to frontend integration, plays a critical role in the application's success.

Upon completion, the application will not only offer users a seamless booking experience but also provide room for scaling and feature enhancements. Incorporating user feedback, monitoring, and iterative improvements will keep the app reliable, competitive, and adaptable to future needs. This project journey not only delivers a valuable tool for users but also strengthens your skills in full-stack development, preparing you for more complex projects in the future.

**REFERENCES:**

<https://medium.com/@viditban/building-a-comprehensive-flight-management-system-with-the-mern-stack-c4c955dc6725>

<https://github.com/seshandonan/Flight-Booking-App>

<https://www.geeksforgeeks.org/ticket-raising-platform-using-mern-stack/>

<https://medium.com/@nur_islam/mern-stack-a-complete-mern-guide-part-2-7ff2320da7f>