UIT2739 – FULL STACK DEVELOPMENT

A PROJECT REPORT

on

Flight Ticket Reservation System

Submitted by

SRIHARI J - 3122225002134
TUFAIL M - 3122225002150
VENKATAGANAPATHISUBRAMANIAN V -312225002153

Department of Information Technology

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

Rajiv Gandhi Salai (OMR), Kalavakkam – 603 110

NOVEMBER 2025

SRI SIVASUBRAMANIYA NADAR COLLEGE OF ENGINEERING

Department of Information Technology

CERTIFICATE

Certified that this project titled "FLIGHT TICKET RESERVATION SYSTEM" is the bonafide work of "SRIHARI J (3122225002134), TUFAIL M (3122225002150) and VENKATAGANAPATHISUBRAMANIAN V (3122225002153)" and is submitted for UIT2739 Full Stack Development Project Review

Place : Kalavakkam

Date : **Internal Examiner**

TABLE OF CONTENTS

1. Project Overview

1.1 Problem Statement

Air travel booking is often marked by fragmented interfaces, lack of transparency, and complex booking management for both customers and administrators. The need persists for a modern, secure, and user-friendly system streamlining flight discovery, booking, and management for both end-users and administrative personnel.

1.2 Motivation

The motivation behind this project is to address the limitations of legacy flight booking platforms by providing an intuitive, end-to-end booking solution. The aim is to enhance transparency, efficiency, and reliability, offering users greater control over their bookings while empowering administrators with comprehensive management tools.

1.3 Objectives

- To implement a modern flight booking web application with a clean, responsive user interface.
- To facilitate secure user registration, authentication, and verification.
- To allow users to search, filter, and book flights with real-time seat availability.
- To enable users to view, manage, and cancel their bookings.
- To provide administrative capabilities for flight and booking management via a dedicated dashboard.
- To ensure data consistency, security, and reliability across the stack.

1.4 System Description

The system is a full-stack web application developed using the MERN stack (MongoDB, Express.js, React.js, Node.js). It is composed of a React frontend, a RESTful backend API built with Express and Node.js, and a MongoDB database for persistent storage.

Users can register for an account, verify their email via a code-based mechanism, log in, search for flights, book tickets for one or more passengers, and manage or cancel their bookings. Administrators have access to an admin dashboard where they can create, edit, or delete flights and monitor all bookings in the system.

1.5 User Roles and Their Capabilities

1. **Guest (Unauthenticated User):**
    - Browse available flights and view flight details.
    - Register a new account.
    - Log in to the system.
2. **Registered User:**
    - Book available flights by specifying passenger details and number of seats.
    - View their personal booking history.
    - Cancel their bookings (with seats returned to flight availability).
    - Log out securely.
3. **Administrator (Admin):**
    - Manage the entire list of flights: create, edit, and delete flights.
    - View all bookings made by all users within the system.
    - Access a dedicated admin dashboard with summary statistics (total flights, total bookings).
    - Cancel any user's booking if necessary.

1.6 Overall Workflow Summary

- **Registration:** Users sign up, providing their details and receive an email with a verification code. The account must be verified before accessing authenticated features.
- **Authentication:** Users log in with email and password; JWT-based authentication secures protected routes.
- **Searching Flights:** Users can search and filter flights by criteria such as departure, destination, date, and airline.
- **Booking Process:** Authenticated users select a flight, specify the number of seats and passenger details, and confirm the booking. The system automatically updates seat availability and calculates the total cost.

- **Booking Management:** Users access their booking history, view details, and may cancel bookings. Cancelling a booking returns the reserved seats to flight availability.
- **Admin Management:** Admins can add, edit, or remove flights and view every booking. All admin operations occur within a secured dashboard.

1.7 High-Level System Behaviour

- Only verified users can book flights and manage their bookings.
- Each booking is linked to a specific user and flight; booking statuses include “confirmed” and “cancelled”.
- Real-time seat availability is enforced: users cannot book more seats than available, and cancellation instantly updates the count.
- Admin actions (flight CRUD and all bookings view) are strictly limited to those with admin privileges.
- All sensitive operations (flight management, booking management, booking cancellation) are protected using JWT authentication and role-based access control.
- The frontend communicates exclusively with the backend API for all data operations, ensuring a clear separation of concerns.

2. Project Requirements

2.1 Wireframes / Screenshots

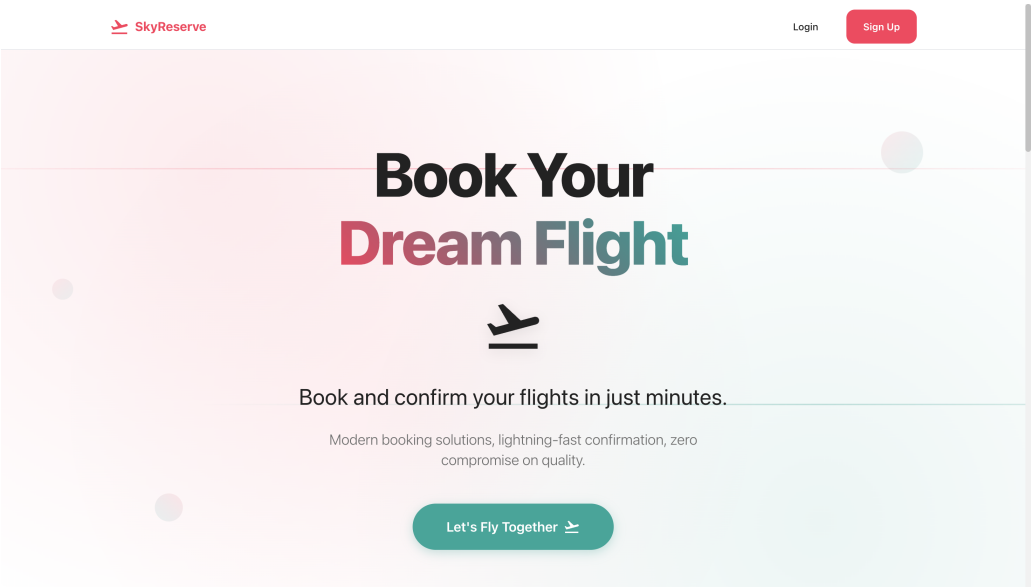

Figure 2.1.1. Home page — welcome message, login and sign up navbar

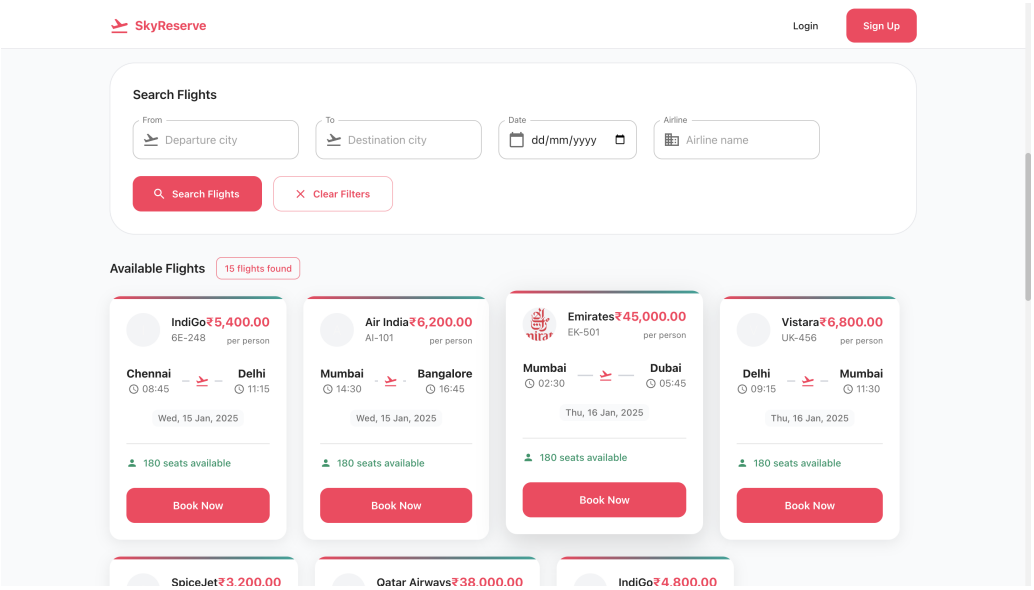

Figure 2.1.2. Flight booking — flight details, fare breakdown and ‘Book Now’ button

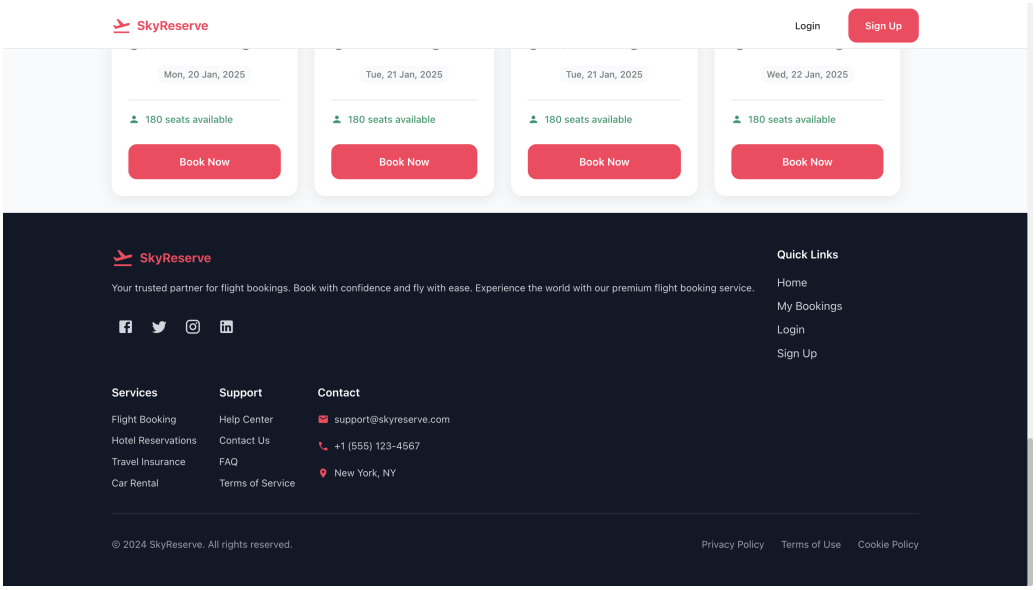

Figure 2.1.3. Contact & footer — company contact info, social links and footer navigation

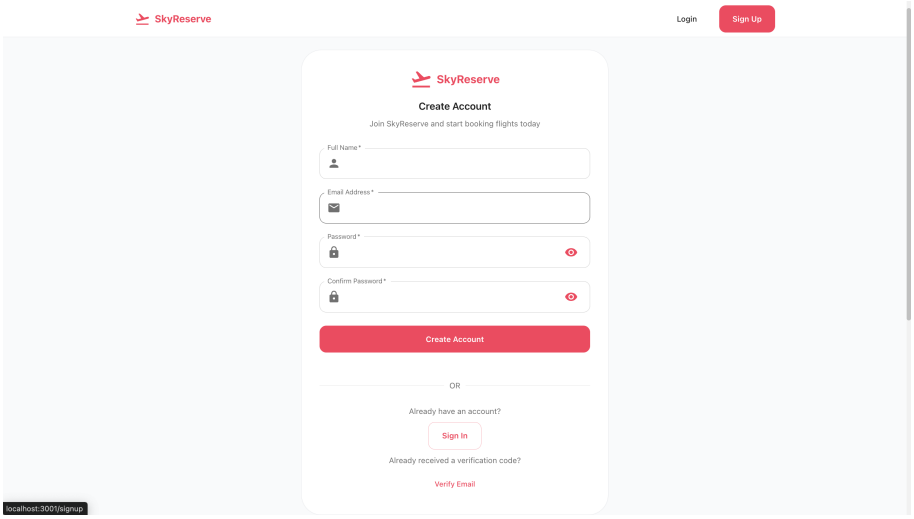

Figure 2.1.4. Sign up page — user registration form with email verification prompt

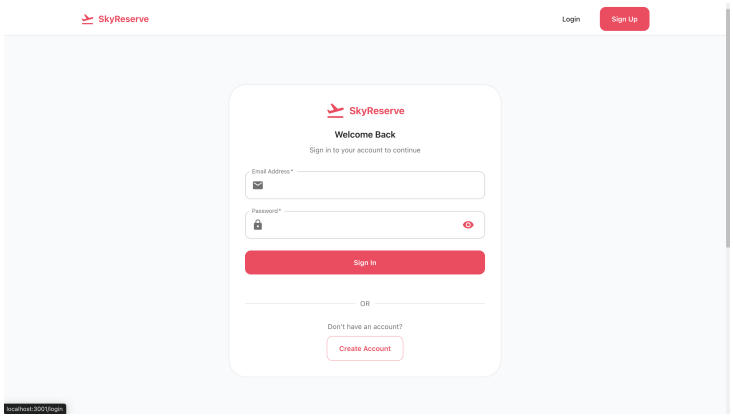

Figure 2.1.5. Login page — email/password authentication screen

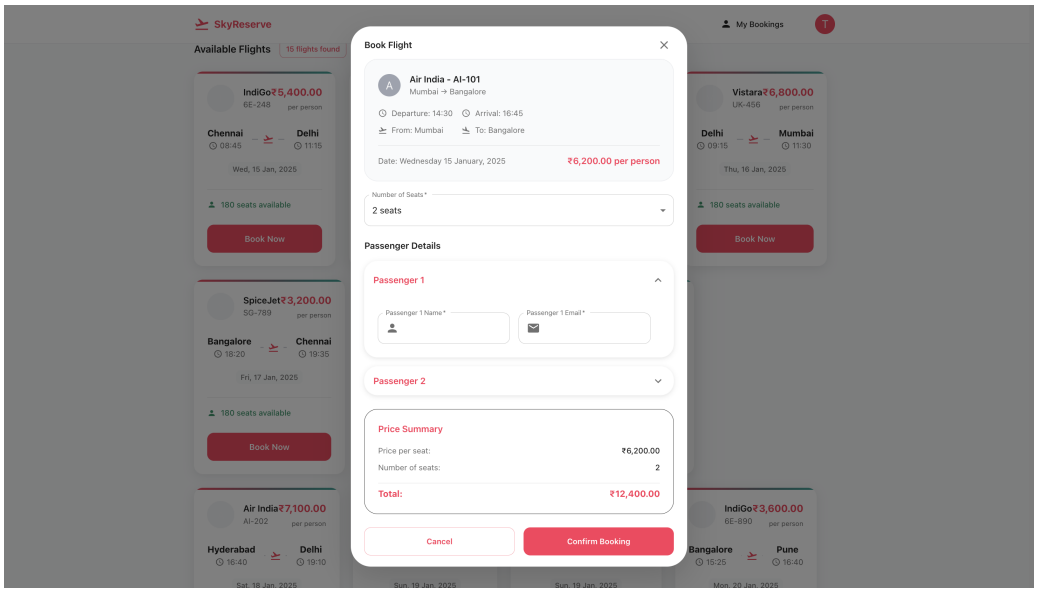

Figure 2.1.6. Booking form — passenger details entry

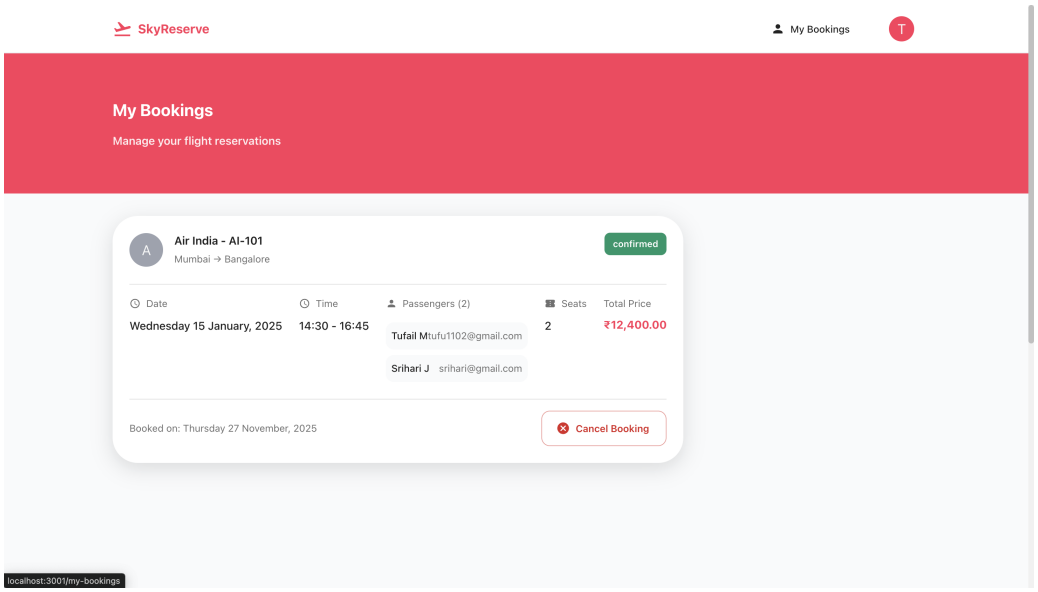

Figure 2.1.7. User bookings — personal booking history and actions (view/cancel)

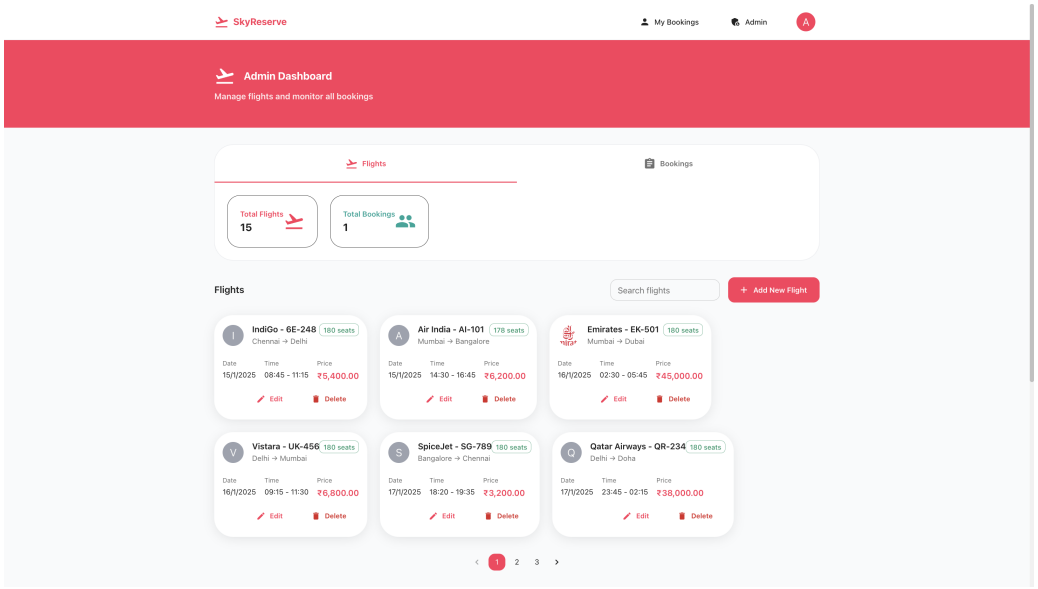

Figure 2.1.8. Admin dashboard — summary statistics, quick actions and navigation

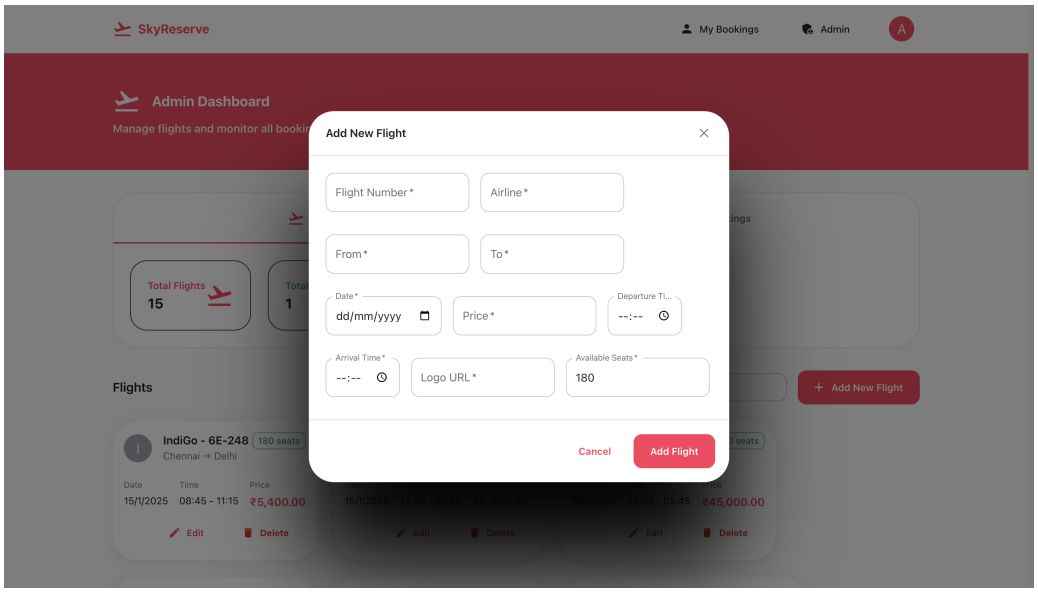

Figure 2.1.9. Admin — add flights form with fields for flight number, route, timings

2.2 Functional Requirements

2.2.1 User Registration and Email Verification

New users must be able to register by providing their name, email, and password. Upon registration, a verification code is sent to the user's email. The user must successfully submit this code to activate their account, ensuring that only valid and accessible email addresses are used for authentication.

2.2.2 User Authentication (Login/Logout)

Users authenticate using their registered email and password. Only those with verified email addresses may log in. Successful authentication produces a JWT token, which is required for subsequent access to protected resources. Users may securely log out, and tokens are not reused after logout.

2.2.3 Flight Search and Filtering

The system enables all users—including guests—to search and filter available flights by criteria such as departure city, destination city, date, and airline. The frontend provides filter input fields, and the backend processes matching queries and returns results sorted chronologically.

2.2.4 Display of Flight Details

For each flight displayed in search results, the application shows comprehensive information including airline, flight number, route, date, departure and arrival times, ticket price, seat

availability, and the airline's logo.

2.2.5 Flight Booking (Authenticated Users)

Authenticated and verified users can book available flights by selecting the number of seats (from one to nine) and entering details for each passenger. The backend validates that the requested number of seats does not exceed availability, creates the booking, and decreases available seats atomically.

2.2.6 Price Calculation and Display

During booking, the system automatically calculates and displays the total price based on the flight's per-seat cost and the number of seats selected. This calculation is performed on both the frontend and backend to ensure consistency and correctness.

2.2.7 Passenger Information Collection

For each seat booked, the application requires the passenger's name and email address. The UI provides individual entry forms for each passenger, and the backend validates completeness and correctness of the information before creating the booking.

2.2.8 Booking Confirmation and History

Upon successful booking, users receive immediate confirmation and a unique booking ID. Verified users can view their complete booking history, including details of flights booked and booking status, organized by booking date.

2.2.9 Booking Cancellation and Seat Return

Users can cancel their confirmed bookings. The backend ensures that only the owner of the booking (or an administrator) may perform this action. Canceled bookings update their status and immediately return the reserved seats to the corresponding flight's availability.

2.2.10 Admin Authentication and Authorization

The system implements a separate administrator role. Administrative actions—such as managing flights and viewing all bookings—are restricted to authenticated users with admin privileges, enforced using JWT and role-based middleware.

2.2.11 Admin: Flight Management (CRUD Operations)

Administrators can create, modify, and delete flights. Form-based validation ensures all required fields are provided. The backend enforces data uniqueness, such as unique flight numbers, and

returns meaningful error messages for conflicts or validation failures.

2.2.12 Admin: Overview and Management of All Bookings

Through the administrator dashboard, authorized users can view all bookings made in the system, including user and flight details. The interface supports filtering and searching of booking records. All sensitive operations are role-checked.

2.2.13 Database Schema Enforcement

The backend defines strict Mongoose schemas for all major entities (User, Flight, Booking), enforcing required fields, data types, unique indexes, and structured validation. Invalid or incomplete records cannot be persisted.

2.2.14 User Input Validation and Error Reporting

All user inputs at both the frontend and backend are validated for completeness and type correctness (e.g., valid email format, password length, seat limits). Validation failures trigger clear error messages in the UI and return appropriate HTTP error responses.

2.2.15 Secure Password Management

User passwords are hashed using industry-standard cryptographic techniques (bcrypt) before being stored in the database. During authentication, password comparison uses secure cryptographic functions, preventing exposure of plain-text credentials.

2.2.16 Email Notification Service (Registration Verification)

During registration, the system sends a verification code to the user's email address using a secure email transport utility. Delivery errors are handled gracefully to prevent unintended registration failures.

2.2.17 JWT-Based Session Management and Protected Routes

The system issues JWT tokens after successful authentication. Protected API endpoints require a valid token, and the backend verifies tokens on every request. Expired or invalid tokens deny access to authenticated features.

2.2.18 Data Consistency and Atomic Updates

Operations such as seat deduction during booking and seat restoration upon cancellation are implemented atomically to maintain data integrity and prevent race conditions or over-booking.

2.2.19 Comprehensive Error Handling and API Responses

All API routes implement structured error handling and return informative error messages for situations such as missing data, validation errors, unauthorized access, resource not found, and internal server errors.

2.2.20 Mobile-Responsive, Modern UI/UX

The frontend is fully responsive, offering a seamless experience across desktops, tablets, and mobile devices. Feedback such as error messages, success notifications, and informational alerts is presented through modern UI components.

2.3 Non-Functional Requirements

2.3.1 Performance

The system provides timely responses to all user actions and API requests. The backend uses non-blocking, asynchronous operations in Node.js and Express for efficient request handling. Client-side workflows such as flight search and booking are optimized for minimal latency and smooth user experience.

2.3.2 Scalability

The MERN architecture (MongoDB, Express.js, React.js, Node.js) supports horizontal scalability. MongoDB facilitates distributed storage through sharding and efficiently handles increasing data and user load.

2.3.3 Security

Security is enforced at multiple layers. Passwords are hashed using bcrypt, and session management uses JWT tokens. Role-based access control restricts sensitive administrative operations. Email verification is required before account activation. Input validation and sanitization mitigate injection attacks. CORS is configured to control cross-origin requests, and sensitive data such as passwords or tokens is never exposed.

2.3.4 Reliability

Robust error-handling ensures reliable system behavior, with clear messaging for failures. Atomic operations maintain consistent state during booking and cancellation. MongoDB and Mongoose reduce database inconsistencies by enforcing schema validation.

2.3.5 Maintainability

The codebase follows modular patterns with controllers, models, routes, middleware, and utilities separated logically. Backend follows the MVC pattern; the frontend maintains organized component structures. Standard libraries, comments, and environment-based configuration improve long-term maintainability.

2.3.6 Usability

The frontend follows modern UI/UX principles and responsive design. Form validations, clear alerts, intuitive workflows, and organized dashboards support easy navigation for users and administrators.

2.3.7 Compatibility

The React frontend ensures cross-browser compatibility and responsive layouts. The backend API follows RESTful design, enabling interaction from any compliant HTTP client. Environment variables allow flexible deployment across different platforms.

2.3.8 Data Integrity

Mongoose schema validation prevents malformed or incomplete records. Atomic updates ensure seat counts and booking statuses remain accurate despite concurrent operations. Unique indexes and required fields further enforce data consistency across the system.

2.4 Use Cases

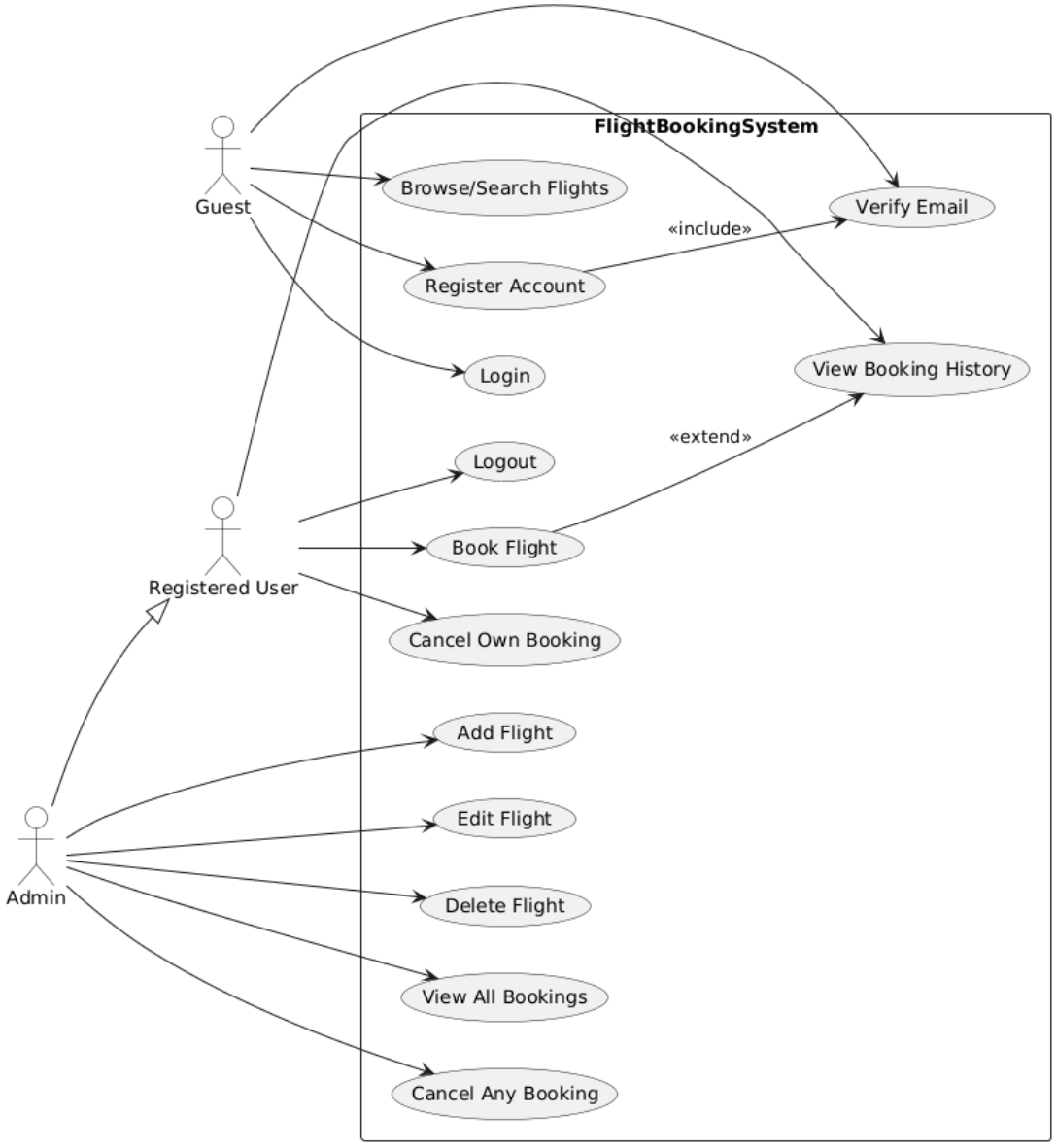

Figure 2.4.1. Use Case Diagram

2.4.1 Use Case 1: User Registration and Email Verification

Field	**Details**
Use Case ID	UC01
Use Case Name	User Registration and Email Verification
Actor(s)	Guest User

Description	A user creates an account by providing details. A verification code is sent to their email. Account is activated upon correct code entry.
Preconditions	User is not logged in. Email address is not already in use.
Postconditions	Verified user account is created and ready for login.
Main Flow	1. User submits registration form with name, email, and password. 2. System validates uniqueness and required fields. 3. System saves unverified user. 4. System sends verification code to email. 5. User enters code to verify. 6. System checks code and marks email as verified.
Alternative / Exception Flows	• Email already exists: Error message. • Validation fails: Form errors. • Code incorrect or expired: Error and retry. • Email sending fails: Account still created (user can retry verification).

2.4.2 Use Case 2: User Login and Logout

Field	**Details**
Use Case ID	UC02
Use Case Name	User Login and Logout
Actor(s)	Registered User
Description	A registered and verified user logs in with email and password, receiving a JWT for protected actions. Logout removes the session locally.
Preconditions	User is registered and has verified their email.
Postconditions	User is logged in and receives authentication token, or is fully logged out.

Main Flow	1. User enters email and password on login screen. 2. System verifies credentials and email status. 3. On success, JWT is returned and stored. 4. User accesses protected resources. 5. User logs out, clearing token.
Alternative / Exception Flows	• Credentials incorrect: Error message. • Email not verified: Prompt for verification. • Token expired/invalid: Forced logout.

2.4.3 Use Case 3: Search and View Flights

Field	**Details**
Use Case ID	UC03
Use Case Name	Search and View Flights
Actor(s)	Guest, Registered User, Admin
Description	Any user searches available flights using filters (from, to, date, airline) and views detailed information for each.
Preconditions	None. Accessible by all.
Postconditions	List of flights matching search criteria displayed.
Main Flow	1. User navigates to homepage. 2. User fills in search/filter fields. 3. System queries backend for flights. 4. Matching flights are displayed with details.
Alternative / Exception Flows	• No matches: “No flights found” message. • Server error: “Failed to fetch” alert.

2.4.4 Use Case 4: Book Flight

Field	**Details**
Use Case ID	UC04

Use Case Name	Book Flight
Actor(s)	Registered User
Description	Authenticated user books seats on a selected flight, entering all required passenger details.
Preconditions	User is logged in and has verified email. Flight exists with enough available seats.
Postconditions	Booking is saved, seats are reserved, confirmation shown, history updated.
Main Flow	1. User clicks “Book Now” on a flight. 2. Booking form appears. 3. User selects number of seats (1–9). 4. User enters details for each passenger. 5. User submits booking. 6. System validates input, checks seat availability, and creates booking. 7. System responds with confirmation including booking ID and updated seats.
Alternative / Exception Flows	• Passenger details incomplete: Form errors. • Seats requested exceed availability: Error message. • Booking fails (e.g., server/database error): Error notification.

2.4.5 Use Case 5: View Personal Booking History

Field	**Details**
Use Case ID	UC05
Use Case Name	View Personal Booking History
Actor(s)	Registered User
Description	Authenticated user views all bookings made under their account, along with up-to-date status, details, and cost.
Preconditions	User is logged in and verified.
Postconditions	User sees booking list with all relevant details.

Main Flow	1. User clicks “My Bookings” in navigation. 2. System fetches user’s bookings. 3. Display booking list with statuses, passenger info, price, and dates.
Alternative / Exception Flows	• No bookings: “No Bookings Found” message. • Fetch fails: Error alert.

2.4.6 Use Case 6: Cancel Booking

Field	**Details**
Use Case ID	UC06
Use Case Name	Cancel Booking
Actor(s)	Registered User, Admin
Description	User or admin cancels a booking, returning reserved seats to the flight. Status updates to “cancelled.”
Preconditions	Booking exists and is “confirmed.” User is owner or admin.
Postconditions	Booking status is “cancelled”; seats returned; visible in history.
Main Flow	1. User selects booking. 2. Clicks “Cancel Booking.” 3. System prompts for confirmation. 4. User confirms. 5. System marks booking as cancelled and restores seats. 6. Success message displayed.
Alternative / Exception Flows	• Not owner/admin: Access denied. • Already cancelled: Error message. • Cancellation fails: Server error message.

2.4.7 Use Case 7: Admin Login and Access Control

Field	**Details**

Use Case ID	UC07
Use Case Name	Admin Login and Access Control
Actor(s)	Admin
Description	Admin authenticates and is granted access to privileged features, which are protected by backend middleware and UI guards.
Preconditions	Admin account exists; user has admin credentials; is logged in and verified.
Postconditions	Admin can access the admin dashboard and functions; non-admins are denied.
Main Flow	1. Admin logs in normally. 2. Backend checks `isAdmin` flag. 3. UI renders "Admin" link and dashboard.
Alternative / Exception Flows	• Not admin: Access denied on protected admin APIs and dashboard UI.

2.4.8 Use Case 8: Admin – Manage Flights (Create, Update, Delete)

Field	**Details**
Use Case ID	UC08
Use Case Name	Admin – Manage Flights
Actor(s)	Admin
Description	Admin adds, edits, or removes flights, with validations for input fields and flight uniqueness.
Preconditions	Admin is logged in and authorized.
Postconditions	Flight changes are reflected in the system; affected users see updates instantly.
Main Flow	1. Admin opens dashboard. 2. Navigates to "Flights" tab. 3. Fills/adds/edits form and submits. 4. API validates and applies changes. 5. UI refreshes flight list with confirmation.

Alternative / Exception Flows	• Form errors: UI and API validation messages. • Flight number duplicate: Error message. • Flight not found (edit/delete): Error response.

2.4.9 Use Case 9: Admin – View All Bookings

Field	**Details**
Use Case ID	UC09
Use Case Name	Admin – View All Bookings
Actor(s)	Admin
Description	Admin reviews all bookings system-wide, with ability to search, filter, and inspect booking/user/flight info.
Preconditions	Admin is logged in and authorized.
Postconditions	Admin views up-to-date summary and details of all booking records.
Main Flow	1. Admin opens dashboard. 2. Switches to “Bookings” tab. 3. System retrieves all bookings. 4. Admin filters/searches as desired. 5. Admin reviews contents.
Alternative / Exception Flows	• Fetch fails: Error message. • No records: “No bookings found” message.

3. Technical Details

3.1 Tech Stack

3.1.1 Frontend Stack

- **React**
  Serves as the core frontend framework for building the single-page application (SPA), supporting modular, component-driven development and stateful interfaces.

- **React Router DOM**
  Enables declarative client-side routing, allowing seamless navigation between pages such as Home, Login, Signup, My Bookings, Admin Dashboard, and Email Verification.

- **Material-UI (MUI)**
  Provides pre-built, customizable UI components for consistent, modern interface styling and responsive layouts.

- **axios**
  Handles all HTTP requests from the frontend to the backend API, including authenticated requests with JWT token headers.

- **@emotion/react, @emotion/styled**
  Used in conjunction with MUI for CSS-in-JS theming, supporting dynamic styling based on theme settings.

- **@mui/icons-material**
  Supplies iconography used throughout the interface for enhanced usability and visual feedback.

3.1.2 Backend Stack

- **Node.js**
  Executes the backend server code, providing an asynchronous, event-driven foundation for scalable API serving.

- **Express.js**
  Functions as the primary server framework for building RESTful APIs, route handling, middleware integration, and error management.

- **dotenv**
  Loads and manages environment variables securely, such as database URIs, JWT secrets, and server ports.

- **bcryptjs**
  Implements secure, salted hashing and verification of user passwords for registration and authentication processes.

- **jsonwebtoken (JWT)**
  Issues, verifies, and validates tokens used for user authentication, access control, and session management.

- **cors**
  Enables and configures Cross-Origin Resource Sharing, permitting controlled access to the backend API from the frontend client.

- **nodemailer**
  Sends verification emails to users during registration, facilitating secure, code-based email activation flows.

- **Mongoose**
  Provides object data modeling (ODM) for MongoDB, enforcing schemas, performing queries, and supporting middleware for data validation and transformation.

3.1.3 Database Layer

- **MongoDB**
  Acts as the project's NoSQL document-oriented database, storing persistent data for users, flights, and bookings. Supports atomic updates for data consistency.

- **Mongoose (ODM)**
  Enforces strict schema definitions, validation rules, and relationship mapping among database collections corresponding to application entities.

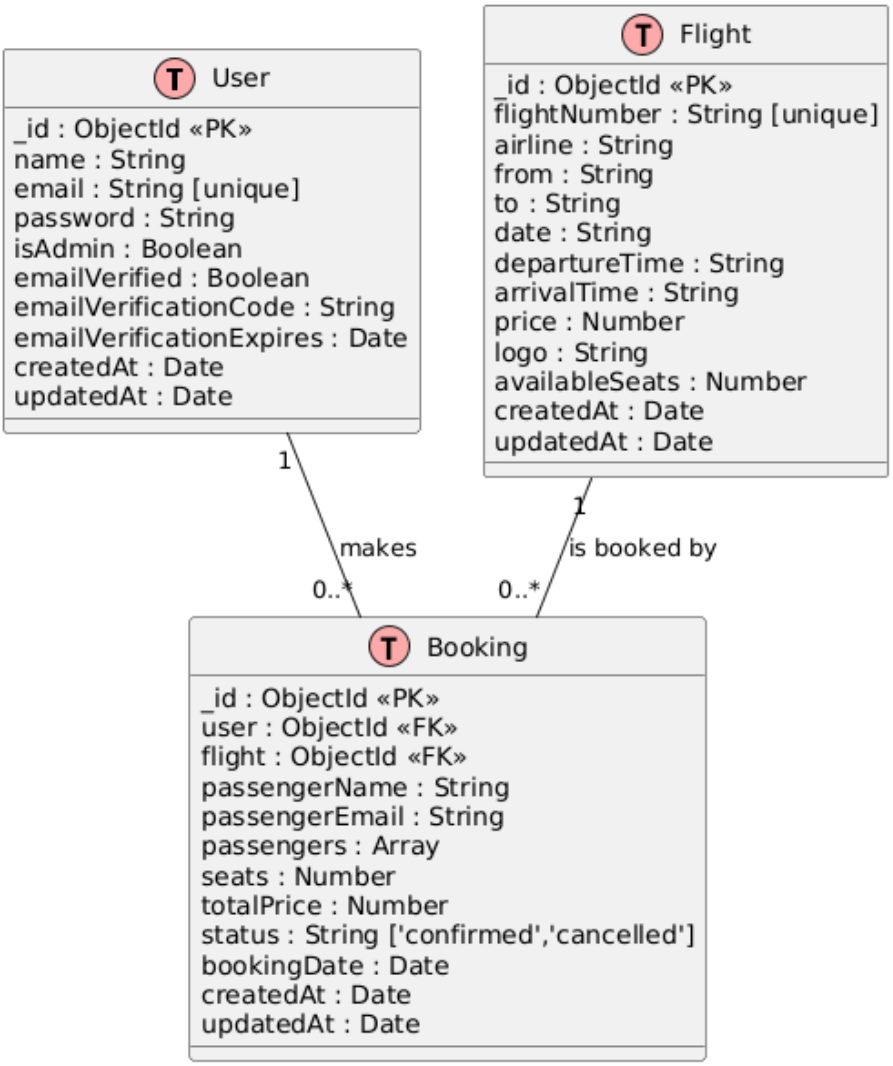

Figure 3.1.1. Entity-Relationship Diagram

3.1.4 APIs and Communication Methods

- **RESTful JSON APIs**
  The backend exposes endpoints using REST principles. All requests and responses utilize JSON, supporting operations for flights, users, authentication, and bookings.

- **HTTP (via axios)**
  All frontend-backend client communication is performed over HTTP, with authentication headers for protected API access.

3.1.5 Development Tools & Dependencies

- **npm (Node Package Manager)**
  Manages project dependencies, scripts, and versioning.

- **nodemon**
  Automatic server restarts during development.

- **concurrently**
  Run frontend and backend concurrently in development.

- **cross-env**
  Cross-platform environment variable handling.

- **eslint**
  Linting and code-style enforcement.

- **react-scripts**
  Create React App build tooling.

- **mui-related dev dependencies**
  Theming, typings, and build-time checks for Material-UI.

3.1.6 Build / Hosting Tools

- **react-scripts (Create React App)**
  Handles building, bundling, and serving the frontend.

- **Node/Express server script**
  Entry point (e.g., `server.js`) for running the backend API.

- **npm scripts**
  Automate setup, development, and production workflows.

3.2 System Architecture

The system is architected according to a layered client–server paradigm, adopting both the Model–View–Controller (MVC) and RESTful API architectural styles. The project is divided into three primary tiers: a React-based presentation layer (frontend), an Express.js application/business logic layer (backend), and a MongoDB persistent storage layer (database). This separation of concerns enhances maintainability, scalability, and secure handling of data.

Communication between frontend and backend is exclusively via RESTful HTTP calls exchanging JSON payloads. The React frontend uses `axios` to perform requests including authentication, flight searches, booking actions, and account operations. Protected requests include a JWT in the `Authorization` header; backend middleware verifies the token and enforces role-based access controls for actions such as booking creation/cancellation and administrative operations.

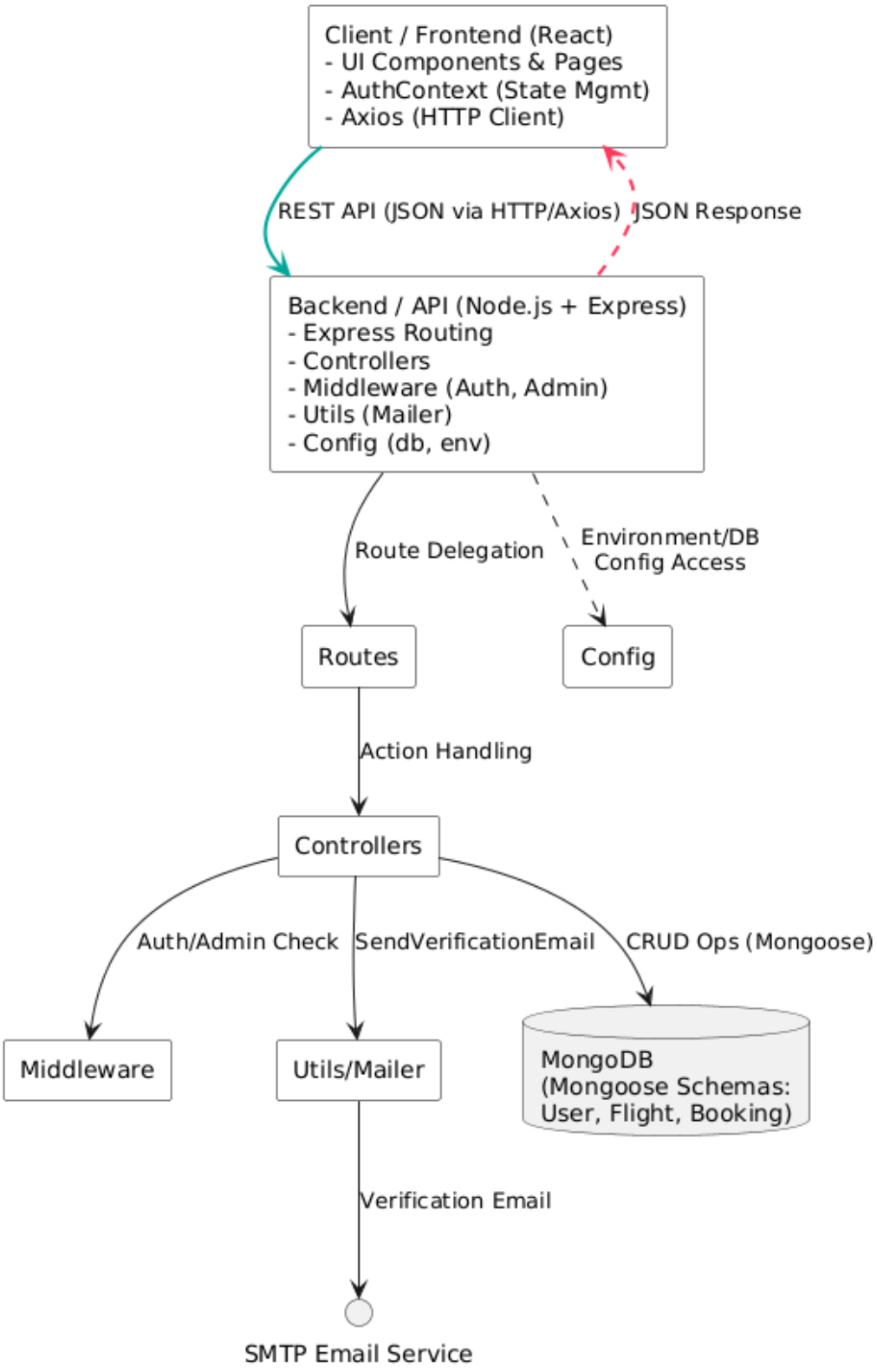

Figure 3.2.1. System Architecture Diagram

Requests from the frontend are received by Express routes, forwarded to controller functions that encapsulate business logic (input validation, authorization checks, transactional operations), and interact with Mongoose models to perform CRUD operations. Controllers coordinate

complex tasks (e.g., atomic seat updates during booking) and return structured HTTP responses. Error conditions are consistently handled and returned as informative error objects.

At the data layer, Mongoose schemas define the shape and constraints of core domain entities (User, Flight, Booking). Data validation is applied at both API and schema levels. Atomic updates (e.g., `findOneAndUpdate` with appropriate operators or transactions where supported) ensure consistency for seat counts and booking records, preventing race conditions.

3.3 Design Patterns

3.3.1 Model-View-Controller (MVC) Pattern

Where it is used: Backend directory structure (e.g., `backend/models/`, `backend/controllers/`, `backend/routes/`).
Why: Separates data schema and persistence (Models), business logic (Controllers), and routing/response handling (Routes/Views).
Benefit: Improves maintainability and modularity; allows independent evolution of layers.

3.3.2 Modular Routing Pattern

Where it is used: Express route modules (e.g., `authRoutes.js`, `flightRoutes.js`, `bookingRoutes.js`). Frontend uses React Router for SPA routes.
Why: Keeps route concerns separated by domain.
Benefit: Easier additions and isolated changes to API endpoints.

3.3.3 Singleton Pattern (Configuration / DB Connection)

Where it is used: Database connection module (e.g., `backend/config/db.js`) and environment configuration.
Why: Ensure a single shared instance of resources such as DB connections.
Benefit: Avoids redundant connections and ensures consistent configuration.

3.3.4 Observer Pattern (Frontend Auth State)

Where it is used: React Context in `frontend/src/context/AuthContext.jsx`.
Why: Propagate authentication state changes to subscribed components.
Benefit: Eliminates prop-drilling and ensures UI updates on auth changes.

3.3.5 Factory Pattern (Model Instantiation)

Where it is used: Mongoose model creation and schema-based instantiation.
Why: Centralize object creation with validation.
Benefit: Simplifies instance creation and enforces schema-level constraints.

3.3.6 Middleware Pattern

Where it is used: Express middleware for authentication/authorization, error handling, logging (e.g., `backend/middleware/authMiddleware.js`).
Why: Encapsulate cross-cutting concerns in a pipeline.
Benefit: Reusable, composable request-processing steps.

4. Conclusion

The presented system is a robust and modern full-stack flight booking application developed using the MERN (MongoDB, Express, React, Node.js) technology stack. It successfully implements a layered client–server architecture with distinct frontend and backend components interacting through secure RESTful APIs. The platform enables seamless workflows for flight search, reservation, user authentication, booking management, and comprehensive administrative oversight, thereby addressing key challenges in traditional flight booking processes—namely, usability, transparency, and operational efficiency.

Through close integration of advanced features, the system delivers essential user functions such as secure account registration with email verification, JWT-based authentication, detailed flight search and filtering, multi-passenger booking with real-time seat validation, booking history tracking, and the ability to cancel reservations. Admin users benefit from privilege-checked features for the creation, modification, and removal of flights, as well as the monitoring of all bookings system-wide via a dedicated dashboard. The project's backend enforces robust validation, error handling, and atomic data operations, while the frontend offers a responsive, accessible user interface supported by Material UI and React Context for state management.

From a technical standpoint, the project demonstrates strong application of architectural best practices and software design patterns, including Modular Routing, MVC, Singleton, Observer, Factory, and Middleware. Schemas are rigorously enforced at both the database and API levels, and asynchronous programming techniques are adopted to achieve responsive and scalable performance. The codebase exhibits modularity, clarity, and extensibility, positioning the application as not only a reliable solution for current use but also a solid foundation for further development.

However, certain limitations are apparent within the current codebase. For instance, the application does not incorporate payment gateway integration, advanced user profile management, detailed audit logging, or internationalization. Additionally, while testing and deployment scripts exist for local and development scenarios, there is no evidence of comprehensive automated testing or DevOps tooling for CI/CD pipelines. Logical future enhancements may include integrating online payment systems, adding multi-language support, strengthening auditing and analytics, implementing automated testing suites, and scaling the system for multi-tenancy or increased airline partnerships. Embracing these improvements would further increase the platform's real-world applicability, security, and commercial viability.