FLIGHT FINDER APPLICATION - HIGH-LEVEL ARCHITECTURE

This document provides a comprehensive overview of the high-level architecture and core technology stack for a modern flight search application. The primary objective is to deliver a seamless and efficient experience for users seeking flight information, by leveraging a robust backend, an intuitive frontend, and reliable integration with external API services. The design emphasizes scalability, maintainability, and performance to ensure a responsive user experience.

1. TECHNOLOGY STACK OVERVIEW

The application is built upon a carefully selected, contemporary, and scalable technology stack. Each component is chosen for its proven capabilities and suitability for its specific role in delivering a high-performance and reliable flight search service.

Layer	Technology	Key Responsibilities
Frontend	**React.js**	Develops dynamic and interactive user interfaces (UI), manages client-side routing, handles user interactions, renders data, and manages application state for a rich user experience.
Backend	Node.js with Express	Manages API endpoints, implements business logic, aggregates data from various sources (including external APIs and databases), handles data persistence, and ensures application security.
Database	MongoDB	Provides a flexible NoSQL document database for storing application-specific data, such as user profiles, cached flight search results, historical query data, and configuration settings.
API Provider	Skyscanner (or Mock)	Serves as the primary external service for fetching real-time flight data, including prices, schedules, availability, and airline information, ensuring up-to-date results.

2. ARCHITECTURE FLOW

The architecture flow details the sequential interactions and data pathways from the moment a user initiates a search query to the display of flight results. This flow is meticulously designed to optimize efficiency, minimize latency, and provide a fluid user experience.

- **User Initiates Search via React.js UI:** The journey begins when a user accesses the React.js-powered web application through their browser. On the intuitive user interface, they input their flight search criteria, such as departure and arrival airports, travel dates, and the number of passengers, and then trigger the search action.
- **Frontend (React.js) Requests Backend (Node.js):** Upon form submission, the React.js application constructs an HTTP request (typically a GET or POST) containing the search parameters. This request is then sent to a designated API endpoint on the Node.js backend (e.g., `/api/flights/search`), acting as the primary communication channel between the client and server.
- **Backend (Node.js) Processes Request & Fetches Data:** The Node.js (Express) backend receives and processes the incoming HTTP request.
    - **Validation & Caching Strategy:** The backend first validates the integrity and format of the search parameters. It then checks its internal MongoDB database to determine if a recent, identical, or highly similar search query has been cached. This step significantly reduces redundant calls to external APIs, improving response times and managing API usage limits.
    - **Skyscanner API Integration:** If the requested data is not available in the cache or requires real-time updates, the Node.js backend dynamically constructs and dispatches a secure request to the Skyscanner API. The backend handles all necessary authentication (e.g., API keys) and adheres to any rate limiting policies imposed by Skyscanner.
    - **Data Aggregation & Transformation:** Once the raw flight data is received from the Skyscanner API, the Node.js backend processes it. This involves filtering out irrelevant information, transforming the data format to align with the application's internal data models, and potentially enriching the data with additional context from MongoDB (e.g., user-specific preferences or historical insights).
    - **Cache Update:** For frequently requested or newly fetched data, the processed flight details are then stored in MongoDB. This caching

    mechanism is crucial for enhancing subsequent search performance and optimizing API call costs.

- **Backend (Node.js) Responds to Frontend:** After successfully processing and aggregating the flight data, the Node.js backend constructs a structured JSON response containing all relevant flight details (e.g., flight options, prices, airlines, stopovers, departure/arrival times). This JSON payload is then transmitted back to the React.js frontend via an HTTP response.
- **Frontend (React.js) Displays Results with Filters & Sort:** The React.js application receives the JSON response. It then intelligently parses this data, updates its internal application state, and dynamically renders the flight results on the user interface.
    - **Interactive Display:** Users are empowered with various interactive options to refine their search results, including filtering by criteria such as airline, number of stopovers, price range, and cabin class. They can also sort the results by price, duration, departure time, or arrival time. React.js's reactive data binding (via state management) and component-based architecture enable these operations to be highly responsive, often performed client-side for immediate feedback.
    - **Further Interactions:** Subsequent user interactions, such as clicking on a specific flight for more details, might trigger new client-side routing within React.js or initiate smaller, targeted HTTP requests to the backend for specific data fetches.

The following diagram visually represents this intricate architecture flow, illustrating the precise interaction points and data movements between each core component of the system.

3. ARCHITECTURE FLOW DIAGRAM

This diagram provides a clear visual representation of the entire end-to-end architecture flow, highlighting the sequence of interactions and data exchanges among the user, frontend, backend, database, and external API services.

Figure 1: Flight Search Application Architecture Flow

[Insert Architecture Flow Diagram Here]
(e.g., User → React.js ↔ Node.js ↔ MongoDB / Skyscanner API)

CONCLUSION

This meticulously designed high-level architecture provides a robust, scalable, and highly maintainable foundation for a high-performance flight search application. By strategically decoupling the frontend and backend layers, leveraging the inherent flexibility and power of a NoSQL database like MongoDB, and integrating seamlessly with a specialized external API such as Skyscanner, the system is engineered to handle complex search queries with remarkable efficiency and deliver a highly responsive user experience.

The modular nature of this design ensures that individual components can be developed, deployed, updated, or scaled independently without impacting the entire system. This flexibility is paramount for future enhancements, feature additions, and adaptation to evolving market demands or technological advancements. The synergy between React.js for a dynamic and interactive user interface, Node.js with Express for a fast and efficient server-side layer, and MongoDB for agile data storage, combined with a reliable external API provider, creates a powerful and resilient ecosystem perfectly suited for the demands of a modern travel application.