Bus Booking System: Design & Reference Document

This document outlines the design for a simple bus booking system. It covers the system architecture, API design, data model, and the core logic as per the assignment requirements.

1. System Architecture

We will employ a standard three-tier architecture, a proven pattern for web-based applications, to ensure separation of concerns, scalability, and maintainability.

- Presentation Layer (Client): A React.js Single-Page Application (SPA) running in the user's web browser. This client will provide an intuitive user interface for searching buses, viewing details, selecting seats, and managing bookings. It will interact with the backend via RESTful API calls.
- Application Layer (Backend Server): This is the core of our system, responsible for all business logic. It will be a Node.js application utilizing the Express.js framework, exposing RESTful API endpoints. It will handle requests for bus searches, booking creation, payment processing, and user management.
- Data Layer (Database): A persistent storage system to securely store all data related to users, bus operators, routes, bus trips, seats, and bookings. A relational SQL database (such as PostgreSQL or SQLite) is chosen for its strong data integrity, transactional capabilities, and structured nature, which is well-suited for managing interconnected booking data.

2. RESTful API Design

The API will serve as the public interface for our system, adhering to RESTful principles for clear, resource-oriented interactions.

Base URL: `/api/v1`

2.1. SEARCH: Search for available bus trips

- Endpoint: `GET /buses/search`
- Description: Retrieves a list of available bus trips based on origin, destination, and desired travel date.
- Query Parameters:
  - `origin`: `string` (e.g., "Mumbai")
  - `destination`: `string` (e.g., "Pune")
  - `date`: `string` (e.g., "YYYY-MM-DD")
- Success Response (200 OK):
  [

```
 {
  "bus_trip_id": "unique_trip_identifier",
  "bus_number": "MH-04-AB-1234",
  "operator_name": "XYZ Travels",
  "departure_time": "ISO_TIMESTAMP",
  "arrival_time": "ISO_TIMESTAMP",
  "duration_hours": "number",
  "base_price": "number",
  "available_seats": "number",
  "bus_type": "AC Sleeper"
 },
 // ... more bus trips
]
```

- Error Response (400 Bad Request): If input parameters are invalid.
- Error Response (404 Not Found): If no buses are found for the given criteria.

2.2. GET_BUS_DETAILS: Retrieve details and seat availability for a specific bus trip

- Endpoint: `GET /buses/{bus_trip_id}`
- Description: Fetches comprehensive details for a specific bus trip, including its seat layout and real-time availability.
- Success Response (200 OK):

```
{
 "bus_trip_id": "unique_trip_identifier",
 "bus_number": "MH-04-AB-1234",
 "operator_name": "XYZ Travels",
 "origin": "Mumbai",
 "destination": "Pune",
 "departure_time": "ISO_TIMESTAMP",
 "arrival_time": "ISO_TIMESTAMP",
 "base_price": "number",
 "bus_type": "AC Sleeper",
 "total_capacity": "number",
 "available_seats_count": "number",
 "seat_layout": [
  { "seat_number": "A1", "status": "available", "price": 800 },
  { "seat_number": "A2", "status": "booked", "price": 800 },
  { "seat_number": "B1", "status": "available", "price": 750 },
  // ... more seats
 ]
}
```

- Error Response (404 Not Found): If `bus_trip_id` does not exist.

2.3. BOOK: Create a new booking

- Endpoint: POST /bookings
- Description: Initiates a new booking for selected seats on a specific bus trip. This step typically precedes payment.
- Request Body:

```
{
 "user_id": "string",
 "bus_trip_id": "string",
 "selected_seats": ["A1", "A2"],
 "passenger_details": [
  { "name": "John Doe", "age": 30, "gender": "Male" },
  { "name": "Jane Doe", "age": 28, "gender": "Female" }
 ]
}
```

- Logic:
  - Validate the bus_trip_id and user_id.
  - Check the real-time availability of selected_seats.
  - Temporarily reserve the seats to prevent double-booking.
  - Calculate the total_amount based on seat prices.
  - Create a new booking record with status: 'PENDING_PAYMENT'.
- Success Response (201 Created):

```
{
 "booking_id": "unique_booking_identifier",
 "user_id": "string",
 "bus_trip_id": "string",
 "total_amount": "number",
 "booked_seats": ["A1", "A2"],
 "status": "PENDING_PAYMENT",
 "message": "Booking initiated. Proceed to payment."
}
```

- Error Response (400 Bad Request): If input data is invalid (e.g., missing fields, invalid seat numbers).
- Error Response (404 Not Found): If bus_trip_id or user_id does not exist.
- Error Response (409 Conflict): If one or more selected_seats are no longer available.

2.4. CONFIRM_PAYMENT: Confirm payment for a booking

- Endpoint: POST /bookings/{booking_id}/confirm-payment
- Description: Finalizes a booking by confirming payment. This would typically be called after a successful interaction with a payment gateway.
- Request Body:

```
{
 "payment_token": "string" // A mock token representing a successful payment
```

```
}
```

- Logic:
  - Validate the `booking_id`.
  - Verify the `payment_token` (in a real system, this would involve calling the payment gateway API).
  - Update the booking `status` from `PENDING_PAYMENT` to `CONFIRMED`.
  - Update the `Seats` table to mark seats as permanently booked.
- Success Response (200 OK):

```
{
 "booking_id": "string",
 "status": "CONFIRMED",
 "message": "Payment successful and booking confirmed."
}
```

- Error Response (400 Bad Request): If `payment_token` is invalid or missing.
- Error Response (404 Not Found): If `booking_id` does not exist.
- Error Response (409 Conflict): If the booking is already confirmed or cancelled.

2.5. VIEW_BOOKING_HISTORY: View all bookings for a user

- Endpoint: `GET /users/{user_id}/bookings`
- Description: Retrieves a summary of all past and upcoming bookings associated with a specific user.
- Success Response (200 OK):

```
{
 "user_id": "string",
 "total_bookings": "number",
 "bookings": [
  {
   "booking_id": "string",
   "bus_trip_id": "string",
   "operator_name": "string",
   "origin": "string",
   "destination": "string",
   "departure_time": "ISO_TIMESTAMP",
   "total_amount": "number",
   "booked_seats": ["A1", "A2"],
   "status": "CONFIRMED"
  },
  // ... more bookings
 ]
}
```

- Error Response (404 Not Found): If `user_id` does not exist or has no bookings.

2.6. CANCEL_BOOKING: Cancel a specific booking

- Endpoint: `POST /bookings/{booking_id}/cancel`
- Description: Allows a user to cancel an existing booking.
- Logic:
  - Validate the `booking_id`.
  - Check if the booking is eligible for cancellation (e.g., not too close to departure time).
  - Update the booking `status` to `CANCELLED`.
  - Mark the corresponding seats in the `Seats` table as `available`.
  - Initiate a refund process (mock in this assignment).
- Success Response (200 OK):

```
{
  "booking_id": "string",
  "status": "CANCELLED",
  "message": "Booking successfully cancelled. Refund initiated."
}
```

- Error Response (400 Bad Request): If the booking is not eligible for cancellation (e.g., already cancelled, past cancellation deadline).
- Error Response (404 Not Found): If `booking_id` does not exist.

3. Data Model / Database Schema

We will use several interconnected tables in our SQL database to manage the bus booking information.

Table: `Users`

- `user_id` (Primary Key, TEXT, UUID)
- `email` (TEXT, UNIQUE, NOT NULL)
- `password_hash` (TEXT, NOT NULL)
- `name` (TEXT, NOT NULL)
- `phone_number` (TEXT)
- `created_at` (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)

Table: `BusOperators`

- `operator_id` (Primary Key, TEXT, UUID)
- `name` (TEXT, UNIQUE, NOT NULL)
- `contact_email` (TEXT)
- `created_at` (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)

Table: `Buses`

- bus_id (Primary Key, TEXT, UUID)
- operator_id (Foreign Key to BusOperators, TEXT, NOT NULL)
- bus_number (TEXT, UNIQUE, NOT NULL)
- type (TEXT, e.g., 'AC', 'Non-AC', 'Sleeper', 'Semi-Sleeper', NOT NULL)
- capacity (INTEGER, NOT NULL)
- seat_layout_config (JSONB/TEXT - Stores a flexible representation of the bus's physical seat layout, e.g., [[1,2],[3,4]] for rows/columns or a more complex schema, NULLABLE)
- created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)

Table: Routes

- route_id (Primary Key, TEXT, UUID)
- origin (TEXT, NOT NULL)
- destination (TEXT, NOT NULL)
- distance_km (DECIMAL)
- estimated_duration_hours (DECIMAL)
- created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)

Table: BusTrips

- bus_trip_id (Primary Key, TEXT, UUID)
- bus_id (Foreign Key to Buses, TEXT, NOT NULL)
- route_id (Foreign Key to Routes, TEXT, NOT NULL)
- operator_id (Foreign Key to BusOperators, TEXT, NOT NULL)
- departure_time (TIMESTAMP, NOT NULL)
- arrival_time (TIMESTAMP, NOT NULL)
- base_price (DECIMAL(10, 2), NOT NULL)
- status (TEXT, e.g., 'SCHEDULED', 'DEPARTED', 'COMPLETED', 'CANCELLED', NOT NULL)
- available_seats_count (INTEGER, NOT NULL) -- Denormalized for quick lookup
- created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)
- updated_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)

Table: Seats

- seat_id (Primary Key, TEXT, UUID)
- bus_trip_id (Foreign Key to BusTrips, TEXT, NOT NULL)
- seat_number (TEXT, NOT NULL) -- e.g., 'A1', 'B2', '1', '2'
- is_available (BOOLEAN, NOT NULL)
- price (DECIMAL(10, 2), NOT NULL) -- Can be different from base_price for premium seats
- booked_by_user_id (Foreign Key to Users, TEXT, NULLABLE)

- booking_id (Foreign Key to Bookings, TEXT, NULLABLE)
- last_updated (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)
- Unique Constraint: (bus_trip_id, seat_number)

Table: Bookings

- booking_id (Primary Key, TEXT, UUID)
- user_id (Foreign Key to Users, TEXT, NOT NULL)
- bus_trip_id (Foreign Key to BusTrips, TEXT, NOT NULL)
- booking_date (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)
- total_amount (DECIMAL(10, 2), NOT NULL)
- status (TEXT, e.g., 'PENDING_PAYMENT', 'CONFIRMED', 'CANCELLED', 'COMPLETED', NOT NULL)
- payment_reference_id (TEXT, NULLABLE) -- Reference ID from payment gateway
- created_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)
- updated_at (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)

Table: Passengers

- passenger_id (Primary Key, TEXT, UUID)
- booking_id (Foreign Key to Bookings, TEXT, NOT NULL)
- seat_id (Foreign Key to Seats, TEXT, NOT NULL)
- name (TEXT, NOT NULL)
- age (INTEGER, NOT NULL)
- gender (TEXT, NOT NULL) -- e.g., 'Male', 'Female', 'Other'
- Unique Constraint: (booking_id, seat_id)

4. Assumptions and Design Decisions

- Seat Reservation During Booking: When a user selects seats and proceeds to book, those seats are temporarily reserved for a short duration (e.g., 10-15 minutes). If payment is not completed within this time, the reservation is automatically released, and the seats become available again.
- Payment Gateway Integration: The system assumes integration with an external payment gateway for processing transactions. For this assignment, a mock payment confirmation step is used to simulate a successful payment.
- Real-time Seat Availability: The system is designed to provide near real-time updates on seat availability to prevent overbooking. This is managed by updating the Seats table and the available_seats_count in BusTrips.
- User Authentication: Basic user registration and login are assumed to manage user-specific booking history.
- Cancellation Policy: A simplified cancellation policy is assumed. Bookings can be

cancelled up to a certain time before departure (e.g., 2 hours), after which cancellations may not be permitted or may incur charges. Refunds for cancellations are simulated.

- Data Integrity and Consistency: A relational SQL database is chosen specifically for its robust support for ACID (Atomicity, Consistency, Isolation, Durability) properties, which are critical for financial transactions and maintaining accurate booking records.
- Stateless API: The backend API is designed to be stateless, meaning each request from the client contains all the information needed to process it, improving scalability and reliability.
- UUIDs for Primary Keys: Universally Unique Identifiers (UUIDs) are used for all primary keys (`_id` fields). This approach is beneficial in distributed systems, avoids potential conflicts with sequential IDs, and simplifies merging data from different sources.

5. Recommended Technology Stack

- Frontend Framework:
  - React.js: A highly popular and powerful JavaScript library for building dynamic, component-based user interfaces. Its declarative nature and efficient DOM updates make it ideal for interactive web applications like a bus booking system.
- Backend Framework:
  - Node.js with Express.js: A fast, unopinionated, and minimalist web framework for Node.js. It's an excellent choice for building RESTful APIs, offering high performance and a vast ecosystem of middleware and libraries.
- Database:
  - PostgreSQL: A powerful, open-source object-relational database system known for its strong compliance with SQL standards, reliability, feature robustness, and performance. It's suitable for production environments requiring high data integrity.
  - SQLite: A serverless, self-contained, file-based SQL database. It's perfect for local development, testing, and smaller-scale applications where a full-fledged database server isn't required, offering simplicity and ease of setup.