Technical Assignment

Candidate: Aman Singh **Position:** Full Stack Developer **Duration:** 3 Days

---

Project Overview

Build a **Mini Event Booking Platform** — a web application where users can browse events, register/login, and book tickets for events.

This assignment will help us evaluate your skills in:

- Backend API development with Node.js and Express
- Frontend development with React.js
- Database design and operations with MongoDB
- Authentication implementation
- Problem-solving and logical thinking
- Code organization and quality

Tech Stack

Layer	Technology
Frontend	React.js
Backend	Node.js + Express.js
Database	MongoDB
Authentication	JWT (JSON Web Tokens)

Task 1: Backend API Development

Build a RESTful API server that handles user authentication, event listing, and ticket booking.

Database Schema Design

Create the following collections in MongoDB:

Users Collection

Field	Type	Description
name	String	User's full name (required)
email	String	User's email address (required, unique)
password	String	Hashed password using bcrypt (required)
createdAt	Date	Account creation timestamp

Events Collection

Field	Type	Description
title	String	Event name (required)
description	String	Event details

date	Date	Event date and time (required)
venue	String	Event location (required)
totalSeats	Number	Total capacity (required)
availableSeats	Number	Remaining seats (required)
price	Number	Ticket price in INR (required)
createdAt	Date	Event creation timestamp

Bookings Collection

Field	Type	Description
userId	ObjectId	Reference to Users collection
eventId	ObjectId	Reference to Events collection
seatsBooked	Number	Number of tickets booked
totalAmount	Number	Total price (seats × price)
bookingDate	Date	When booking was made

API Endpoints

Implement the following endpoints:

POST `/api/auth/register`

Register a new user account.

Request Body:

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "securepassword123"
}
```

Expected Behavior:

- Validate that all fields are provided
- Check if email already exists in database
- Hash the password using bcrypt before storing
- Return success message with user details (exclude password)
- Return appropriate error if email already registered

POST `/api/auth/login`

Authenticate user and return JWT token.

Request Body:

```
{
  "email": "john@example.com",
  "password": "securepassword123"
}
```

Expected Behavior:

- Validate credentials against database
- Compare password hash using bcrypt
- Generate JWT token with user ID in payload
- Return token and basic user info on success
- Return error message for invalid credentials

GET `/api/events`

Fetch all upcoming events.

Expected Behavior:

- Return list of all events from database
- Only return events where date is in the future
- Sort by date (nearest first)
- Each event should include all fields

GET `/api/events/:id`

Fetch details of a single event.

Expected Behavior:

- Return complete event details for given ID
- Return 404 error if event not found

POST `/api/bookings` (Protected Route)

Book tickets for an event. Requires authentication.

Request Headers:

```
Authorization: Bearer <jwt_token>
```

Request Body:

```
{
  "eventId": "event_object_id",
  "seatsBooked": 2
}
```

Expected Behavior:

- Verify JWT token and extract user ID
- Check if event exists
- Validate that requested seats ≤ available seats
- Calculate total amount (seats × event price)
- Create booking record in database
- Decrease `availableSeats` in the event document
- Return booking confirmation details
- Return error if not enough seats available

GET `/api/bookings/my` (Protected Route)

Get all bookings for the logged-in user.

Request Headers:

```
Authorization: Bearer <jwt_token>
```

Expected Behavior:

- Extract user ID from JWT token
- Return all bookings made by this user
- Include event details (title, date, venue) in response
- Sort by booking date (most recent first)

Backend Requirements Summary

1. **Authentication Middleware**: Create a middleware function that verifies JWT tokens and protects routes
2. **Password Security**: Use bcrypt to hash passwords before storing
3. **Input Validation**: Validate all incoming data and return meaningful error messages
4. **Error Handling**: Use appropriate HTTP status codes (200, 201, 400, 401, 404, 500)
5. **Seat Availability**: Ensure booking logic prevents overbooking

---

Task 2: Frontend Development

Build a React application that consumes the backend APIs and provides a user interface for the booking platform.

Pages to Implement

Home Page (`/`)

The landing page displaying all available events.

Requirements:

- Fetch events from `/api/events` on page load
- Display events in a card grid layout
- Each card should show:
  - Event title
  - Date (formatted nicely, e.g., "15 Jan 2025, 6:00 PM")
  - Venue
  - Price per ticket
  - Available seats remaining
- Clicking on a card navigates to the event details page
- Show a loading spinner while fetching data
- Display a message if no events are available

Register Page (`/register`)

New user registration form.

Requirements:

- Form fields: Name, Email, Password, Confirm Password
- Client-side validation:
  - All fields required
  - Valid email format
  - Password minimum 6 characters

    - Passwords must match
- Show validation errors below respective fields
- On successful registration, redirect to login page
- Show error message if email already exists

Login Page (`/login`)

User login form.

Requirements:

- Form fields: Email, Password
- Client-side validation for empty fields
- On successful login:
    - Store JWT token (localStorage or Context)
    - Store basic user info
    - Redirect to home page
- Show error message for invalid credentials
- Link to register page for new users

Event Details Page (`/events/:id`)

Detailed view of a single event with booking functionality.

Requirements:

- Fetch event details using the ID from URL
- Display complete event information:
    - Title, Description, Date, Venue
    - Price per ticket
    - Available seats
- Booking Section (visible only if user is logged in):
    - Number input for selecting seat count
    - Show calculated total amount as user changes seat count
    - "Book Now" button
    - Validate seats don't exceed availability
- After successful booking, show confirmation message
- If user is not logged in, show "Login to Book" button that redirects to login page
- Show error if booking fails

My Bookings Page (`/my-bookings`) — Protected

Display all bookings made by the logged-in user.

Requirements:

- This page should only be accessible to logged-in users
- Redirect to login if not authenticated
- Fetch bookings from `/api/bookings/my`
- Display each booking with:
    - Event title
    - Event date and venue
    - Number of seats booked
    - Total amount paid
    - Booking date
- Show message if user has no bookings

- Loading state while fetching

Frontend Requirements Summary

1. **Routing**: Use React Router for navigation between pages
2. **Protected Routes**: Implement route protection for authenticated pages (My Bookings)
3. **State Management**: Use React Context or useState for:
    - Authentication state (logged in/out)
    - User information
    - JWT token storage
4. **API Integration**:
    - Create a service/utility for API calls
    - Include JWT token in Authorization header for protected routes
5. **User Experience**:
    - Show loading indicators during API calls
    - Display meaningful error messages
    - Provide feedback on successful actions (booking confirmed, etc.)
6. **Responsive Design**: Basic responsiveness so it works on both desktop and mobile screens
7. **Navigation**: Include a header/navbar with:
    - Logo/App name (links to home)
    - Login/Register links (when logged out)
    - My Bookings link (when logged in)
    - Logout button (when logged in)

Task 3: Problem Solving

Solve any **ONE** of the following three problems. This tests your logical thinking and coding abilities.

Save your solution in a `/solutions` folder in your repository.

Problem 1: Maximum Bookings

You have multiple booking requests and limited seats. Find the maximum number of bookings you can fulfill.

Scenario:

- You have several booking requests, each requiring different number of seats
- You have a fixed number of total available seats
- You want to maximize the NUMBER of bookings (not seats filled)
- A booking must be fulfilled completely (no partial bookings)

Input:

- `requests` : array of integers (seats needed per booking)
- `totalSeats` : integer (total seats available)

Output:

- Maximum number of bookings that can be fulfilled

Example 1:

```
Input: requests = [3, 2, 5, 1, 4], totalSeats = 10
Output: 4
Explanation: Accept bookings requiring [1, 2, 3, 4] seats = 10 total, 4 bookings
```

Example 2:

```
Input: requests = [5, 5, 5], totalSeats = 10
Output: 2
Explanation: Can only fit two bookings of 5 seats each
```

Write a function: `maxBookings(requests, totalSeats)`

Problem 2: Find Overlapping Events

Given a list of events with time slots, find which events conflict with each other.

Scenario:

- Multiple events are scheduled with start and end times
- Two events overlap if one starts before the other ends
- Find all pairs of overlapping events

Input:

- `events` : array of objects with `{id, startTime, endTime}`

Output:

- Array of pairs `[id1, id2]` representing overlapping events

Example:

```
Input: [
  {id: 1, startTime: 1, endTime: 5},
  {id: 2, startTime: 3, endTime: 7},
  {id: 3, startTime: 6, endTime: 10},
  {id: 4, startTime: 12, endTime: 15}
]

Output: [[1, 2], [2, 3]]

Explanation:
- Event 1 (1-5) overlaps with Event 2 (3-7) because 3 < 5
- Event 2 (3-7) overlaps with Event 3 (6-10) because 6 < 7
- Event 3 does not overlap with Event 4
- Event 1 does not overlap with Event 3
```

Write a function: `findOverlappingEvents(events)`

Problem 3: Rate Limiter

Design a rate limiter that restricts how many requests a user can make in a time window.

Scenario:

- Build a system to prevent API abuse
- Each user can only make N requests within a time window
- Different users have separate limits

Requirements:

- Create a `RateLimiter` class
- Constructor takes: `maxRequests` (number) and `windowSeconds` (number)
- Method `isAllowed(userId)` returns `true` or `false`

Example:

```
const limiter = new RateLimiter(3, 60); // 3 requests per 60 seconds

limiter.isAllowed("user1"); // returns true (1st request)
limiter.isAllowed("user1"); // returns true (2nd request)
limiter.isAllowed("user1"); // returns true (3rd request)
limiter.isAllowed("user1"); // returns false (limit exceeded)
limiter.isAllowed("user2"); // returns true (different user, separate limit)

// After 60 seconds pass...
limiter.isAllowed("user1"); // returns true (window reset)
```

Implement: `class RateLimiter` with constructor and `isAllowed` method

Submission Instructions

Repository Structure

Organize your code as follows:

```
your-repo-name/
│
├── backend/
│   ├── config/
│   │   └── db.js
│   ├── middleware/
│   │   └── auth.js
│   ├── models/
│   │   ├── User.js
│   │   ├── Event.js
│   │   └── Booking.js
│   ├── routes/
│   │   ├── auth.js
│   │   ├── events.js
│   │   └── bookings.js
│   ├── server.js
│   ├── package.json
│   └── .env.example
│
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── context/
│   │   ├── App.jsx
│   │   └── main.jsx
│   ├── package.json
│   └── index.html
│
├── solutions/
│   └── problem.js (whichever one you solved)
```

```
│
└── README.md
```

README Requirements

Your README.md should include:

1. **Project Title and Description**
2. **Setup Instructions**
    - Prerequisites (Node.js version, MongoDB)
    - How to install dependencies
    - Environment variables needed
    - How to run the backend server
    - How to run the frontend application
3. **API Documentation**
    - List of all endpoints
    - Request/response format for each
4. **Screenshots**
    - Home page with event listings
    - Login/Register pages
    - Event details with booking form
    - My Bookings page
5. **Assumptions Made** (if any)

What to Submit

- Share the **public GitHub repository link**
- Ensure the application runs without errors when following your setup instructions

---

Evaluation Criteria

We will evaluate your submission based on:

Backend

- API endpoints work correctly as specified
- Proper authentication and route protection
- Input validation and error handling
- Clean code structure and organization

Frontend

- All pages implemented and functional
- Proper integration with backend APIs
- Good user experience (loading states, error messages)
- Clean component structure
- Basic responsive design

Problem Solving

- Correct solution with proper logic
- Code readability
- Edge cases handled

Overall

- Code quality and readability
- Project organization
- Git commit history (meaningful commits)
- Documentation quality

Important Notes

1. **Original Work**: Write your own code. You may refer to documentation and tutorials but do not copy solutions.

2. **Clarifications**: If any requirement is unclear, make reasonable assumptions and document them in your README.

3. **Partial Submission**: If you cannot complete everything, submit what you have. We evaluate partial work too.

4. **Seed Data**: You can create a script or manually add some sample events to the database for testing.

Questions?

For any queries about this assignment, contact: hr@ubiqtech.ai

---

Deadline: 3 days from the date of receiving this assignment

We look forward to reviewing your work!