Smart Campus Services Portal - Software Engineering Documentation

Table of Contents

        - To scale, swap lowdb for MongoDB/Postgres with minimal API changes.
        - Use environment variables for DB connection strings and secrets.
    - UML Diagrams
        - **Sequence Diagram: Registration & Error Handling**

```
User -> Frontend: Fill registration form
Frontend -> Backend: POST /api/auth/register
Backend -> DB: Check for existing user/email
DB --> Backend: Exists/Not exists
Backend -> Backend: Hash password, create user
Backend --> Frontend: Success/Error response
Frontend -> User: Show success/error message
```

        - **Activity Diagram: Booking Approval**

```
[Booking Submitted] -> [Check Role]
  -> [If Student] -> [Pending Approval]
  -> [If Faculty/Admin] -> [Auto-Approve]
  -> [Send Notification]
```

        - **Class Diagram (Key Entities)**

```
+------------------+
|      User        |
+------------------+
| _id: String      |
| username: String |
| password: String |
| name: String     |
| email: String    |
| role: String     |
+------------------+
```

    - Database Design
        - **Schema:**
            - Users: _id, username, password, name, email, role
            - Bookings: _id, roomId, userId, date, startTime, duration, purpose, status
            - MaintenanceRequests: _id, location, issueType, description, reportedBy, status, createdAt, resolvedAt, assignedTo
            - Announcements: _id, title, content, category, createdBy, createdAt, target

4. Development
    - **Best Practices**
        - **Code Organization:**
            - Frontend: `src/components`, `src/contexts`, `src/pages`
            - Backend: `routes`, `middleware`, `db`, `models`
        - **Adding a New Feature:**

1. Create a new branch: `git checkout -b feature/your-feature`
2. Add backend route/controller and frontend component.
3. Write unit/integration tests.
4. Open a Pull Request for review.
5. Merge after approval and CI passes.

- **Local Development:**

```
npm install
cd frontend && npm install
cd backend && npm install
npm start
```

- **Extending the API:**
  - Add a new file in `backend/routes/`, update `backend/index.js`, and document the endpoint.
- **Extending the Frontend:**
  - Add a new component in `src/components/`, update routing/context as needed.
- **Branching Strategy:**
  - Follow Git Flow or trunk-based development for larger teams.
- **Code Review:**
  - Use Pull Requests and require at least one approval before merging to main.

- **Running & Testing Locally**
  - **Start Backend:** `cd backend && npm start`
  - **Start Frontend:** `cd frontend && npm start`
  - **Run Tests:** `npm test` (in either frontend or backend)
- **Feature Example: Adding Notifications**
  1. Create `backend/routes/notifications.js` and define endpoints.
  2. Add notification UI in `frontend/src/components/notifications/`.
  3. Update context/provider if global state is needed.
  4. Write tests for both layers.
  5. Document new endpoints and UI in this file.
- Technology Stack
- Frontend Implementation
- Backend Implementation
- API Documentation

5. Testing
   - Unit Testing
   - Integration Testing
   - User Acceptance Testing
6. Deployment
7. Maintenance and Support

Introduction

The Smart Campus Services Portal is a comprehensive web application designed to centralize and streamline various campus services. It provides a role-based access system catering to students, faculty, and administrators, offering features such as room bookings, class schedules, maintenance requests, and announcements.

This document provides detailed technical documentation of the system from a software engineering perspective, covering requirements analysis, design, implementation, testing, and deployment.

Requirements Engineering

Software Design

System Architecture

Why These Technologies?

- **REST:** Stateless, cacheable, and easy for mobile/web clients.
- **JWT:** Enables scalable, stateless authentication (no server-side session storage).
- **lowdb:** Fast prototyping, easy migration to MongoDB/Postgres later.
- **React:** Component-based, fast, and easy to maintain/extend.

Scalability & Modularity

- Backend can be containerized (Docker), load-balanced, and connected to a cloud DB.
- Modular codebase allows separate teams to work on frontend/backend independently.

Security

- All sensitive endpoints require JWT; CORS is locked down to trusted origins.
- Passwords hashed with bcrypt; input validation on both client/server.

Deployment Workflow

```
Developer Push -> GitHub Actions CI -> Build & Test -> Deploy to Cloud/Server
```

Migration Path

- To scale, swap lowdb for MongoDB/Postgres with minimal API changes.
- Use environment variables for DB connection strings and secrets.

Architectural Rationale & Tradeoffs

- **Separation of Concerns:** The split between frontend (React), backend (Express), and storage (lowdb) ensures modularity, easier debugging, and team parallelization.
- **Scalability:** While lowdb is ideal for rapid prototyping and small deployments, the backend is architected such that swapping in a more scalable database (MongoDB, PostgreSQL) would require minimal changes to the API layer.
- **Security:** JWT authentication is stateless and scalable. Passwords are hashed with bcrypt. CORS is configured to restrict cross-origin requests.
- **Extensibility:** The RESTful API design and modular React components make it straightforward to add new features (e.g., notifications, analytics) or integrate with external systems (e.g., campus LDAP, SMS gateways).
- **Deployment:** The project is containerizable (Docker-ready) and can be deployed to cloud platforms or on-premises.

Security Considerations

- **JWT Expiry:** Tokens are set to expire after 24 hours to reduce risk if leaked.
- **Role-based Access Control:** All sensitive endpoints are guarded by middleware that checks user roles.
- **Input Validation:** Both frontend and backend validate user input to prevent injection attacks.
- **Error Handling:** API responds with meaningful HTTP status codes and messages. Server errors are logged for admin review (can be extended with Winston or similar logging libraries).

Error Handling & Logging

- **User Feedback:** Frontend surfaces errors (e.g., failed login, booking conflict) via Bootstrap alerts.
- **Backend Logging:** Errors are logged to the console and can be extended to log files or external monitoring services.
- **API Responses:** Consistent JSON structure for errors: `{ "message": "Error description" }`.

API Versioning & Extensibility

- **Versioning:** All endpoints are prefixed with `/api/` and can be extended to `/api/v2/` etc. for backward compatibility.
- **RESTful Principles:** Resources are nouns, HTTP verbs are used semantically, and statelessness is maintained. This enables easier integration with third-party tools and mobile apps.

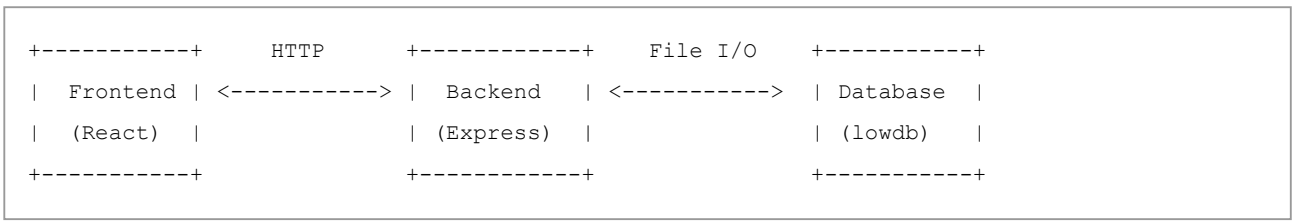
The Smart Campus Services Portal uses a three-tier client-server architecture:

- **Frontend:** React SPA (Single Page Application) for user interaction.
- **Backend:** Node.js/Express REST API for business logic and data access.
- **Database:** JSON file storage via lowdb for persistence (collections: users, bookings, schedules, maintenanceRequests, announcements).

Flow:

1. The frontend communicates with the backend via RESTful API calls (HTTP/HTTPS).
2. The backend authenticates requests (JWT), enforces role-based access, and manipulates data in the JSON database.
3. The database stores all persistent data in collections.

Component Diagram



UML Diagrams

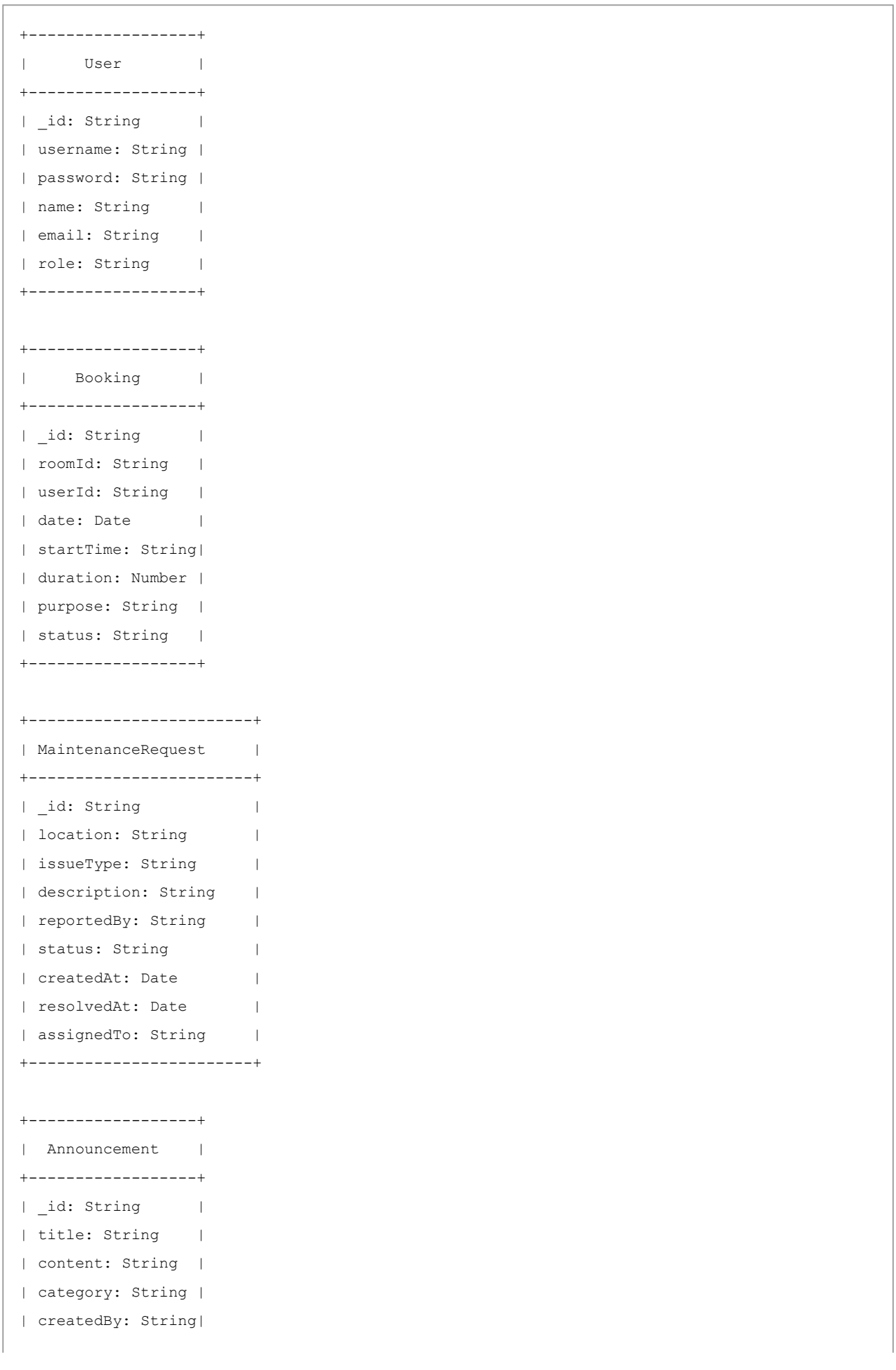
Sequence Diagram: Registration & Error Handling

```
User -> Frontend: Fill registration form
Frontend -> Backend: POST /api/auth/register
Backend -> DB: Check for existing user/email
DB --> Backend: Exists/Not exists
Backend -> Backend: Hash password, create user
Backend --> Frontend: Success/Error response
Frontend -> User: Show success/error message
```

Activity Diagram: Booking Approval

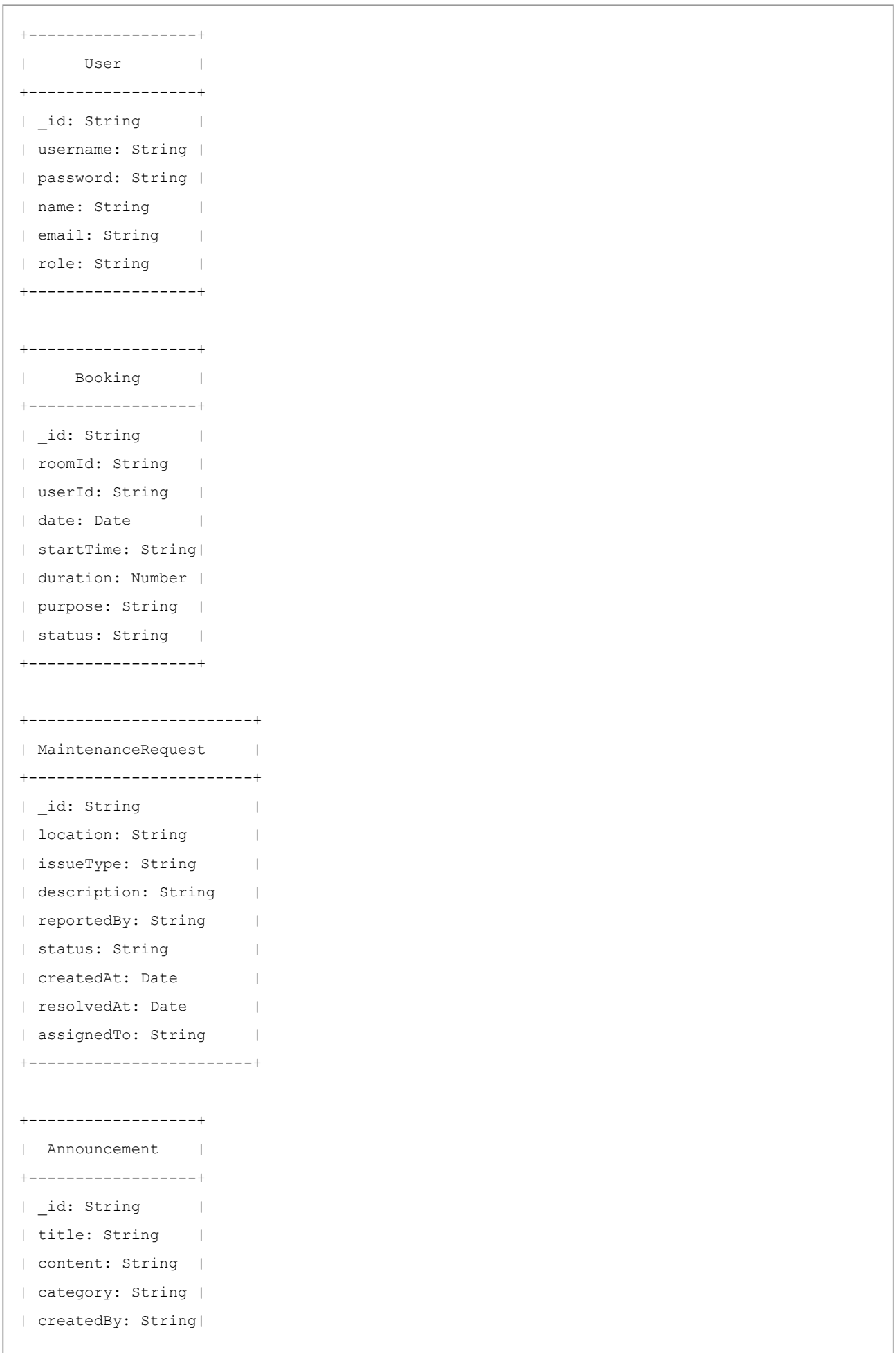
```
[Booking Submitted] -> [Check Role]
  -> [If Student] -> [Pending Approval]
  -> [If Faculty/Admin] -> [Auto-Approve]
  -> [Send Notification]
```

Class Diagram (Key Entities)


User
_id: String
username: String
password: String
name: String
email: String
role: String
Booking
_id: String
roomId: String
userId: String
date: Date
startTime: String
duration: Number
purpose: String
status: String
MaintenanceRequest
_id: String
location: String
issueType: String
description: String
reportedBy: String
status: String
createdAt: Date
resolvedAt: Date
assignedTo: String
Announcement
_id: String
title: String
content: String
category: String
createdBy: String

```
| createdAt: Date  |
| target: String   |
+------------------+
```

Class Diagram (Key Entities)


User
_id: String
username: String
password: String
name: String
email: String
role: String
Booking
_id: String
roomId: String
userId: String
date: Date
startTime: String
duration: Number
purpose: String
status: String
MaintenanceRequest
_id: String
location: String
issueType: String
description: String
reportedBy: String
status: String
createdAt: Date
resolvedAt: Date
assignedTo: String
Announcement
_id: String
title: String
content: String
category: String
createdBy: String

```
| createdAt: Date  |
| target: String   |
+------------------+
```

Sequence Diagram (Room Booking)

```
Student -> Frontend: Request to book room
Frontend -> Backend: POST /api/bookings
Backend -> Database: Check room availability
Database --> Backend: Room status
Backend -> Database: Create booking
Database --> Backend: Success
Backend -> Frontend: Booking confirmation
Frontend -> Student: Show confirmation
```

Use Case Diagram (Text Representation)

- Student: Register, Login, Book Room, View Schedule, Submit Maintenance Request, View Announcements
- Faculty: Login, Approve/Reject Bookings, Manage Schedules, Post Announcements
- Admin: All faculty actions + Manage Users, Generate Reports

Database Design

Indexing & Performance

- For production, migrate to MongoDB/Postgres and add indexes on `userId`, `roomId`, `date` for fast lookups.
- Archive bookings and maintenance requests older than 1 year.
- Schedule regular backups of `db.json` (or use managed DB backup if migrated).

Migration Path

- Abstract DB logic in utility modules (`db/utils.js`).
- Write migration scripts to export/import data to new DB.

Analytics Example

- Count bookings per month:

```
db.get('bookings')
  .filter(b => b.date.startsWith('2025-04'))
  .size()
  .value();
```

Schema Evolution

- Use versioned migrations; keep old fields for backward compatibility, add new fields as optional.

Data Integrity & Migration

- **Unique Constraints:** User emails and usernames are enforced as unique in the backend logic.
- **Referential Integrity:** Foreign keys (e.g., `userId` in bookings) are checked on creation; orphaned records are avoided by cascading deletes (future enhancement).
- **Migration:** Moving from lowdb to a full RDBMS or NoSQL solution is supported by clear data models and separation of database logic into utility modules.

Security & Privacy

- **Password Hashing:** User passwords are never stored in plaintext.
- **Sensitive Data:** Only non-sensitive fields are sent to the frontend. Private fields (like password hashes) are stripped from API responses.

Example Query Patterns

- **Get all bookings for a user:**

```
db.get('bookings').filter({ userId: 'u456' }).value();
```

- **Find open maintenance requests:**

```
db.get('maintenanceRequests').filter({ status: 'open' }).value();
```

Data Growth & Performance

- **Archiving:** Old records can be archived to separate files or collections for performance.
- **Indexing:** For larger deployments, an indexed database is recommended.

The database is a set of JSON collections. Each collection is an array of objects (documents) with unique `_id` fields.

Relationships are managed via foreign keys (e.g., `userId` in bookings).

Collections:

- `users`: User accounts and roles
- `bookings`: Room booking requests
- `schedules`: Class schedules
- `maintenanceRequests`: Facility issue tickets
- `announcements`: Campus-wide messages

Example: Booking Document

```
{
  "_id": "b123",
  "roomId": "R101",
  "userId": "u456",
  "date": "2025-04-22",
  "startTime": "10:00",
  "duration": 2,
  "purpose": "Study Group",
  "status": "pending"
}
```

Stakeholders

Stakeholder	Description	Interests
Students	Primary users of the system	Access schedules, book rooms, submit maintenance requests
Faculty	Teaching and administrative staff	Manage schedules, approve bookings, post announcements
Administrators	IT and facilities management	System oversight, generating reports, user management
IT Department	Technical support team	System maintenance, security, updates

System Requirements Specification

Functional Requirements

1. **User Authentication and Authorization**
    - FR1.1: The system shall allow users to register with a username, password, name, and email.
    - FR1.2: The system shall authenticate users via username and password.
    - FR1.3: The system shall assign one of three roles to users: student, faculty, or admin.
    - FR1.4: The system shall restrict access to features based on user roles.
2. **Room Booking Management**
    - FR2.1: The system shall allow users to create room booking requests.
    - FR2.2: The system shall display available rooms based on date and time.
    - FR2.3: The system shall allow faculty to approve or reject booking requests.
    - FR2.4: The system shall notify users of booking status changes.
3. **Class Schedule Management**
    - FR3.1: The system shall display class schedules to all authenticated users.
    - FR3.2: The system shall allow faculty to create and modify class schedules.
    - FR3.3: The system shall prevent scheduling conflicts for rooms and instructors.
4. **Maintenance Request System**
    - FR4.1: The system shall allow users to submit maintenance requests.

- FR4.2: The system shall allow tracking of maintenance request status.
- FR4.3: The system shall allow admins to assign, update, and close maintenance requests.

5. **Announcement System**

   - FR5.1: The system shall allow faculty and admins to post announcements.
   - FR5.2: The system shall display relevant announcements to users.
   - FR5.3: The system shall support formatting and categorization of announcements.

6. **Reporting**

   - FR6.1: The system shall generate PDF reports for user data and bookings.
   - FR6.2: The system shall restrict access to reports to admin users only.

Non-Functional Requirements

1. **Usability**

   - NFR1.1: The system shall provide an intuitive user interface.
   - NFR1.2: The system shall be responsive and compatible with modern browsers.
   - NFR1.3: The system shall provide clear feedback for user actions.

2. **Performance**

   - NFR2.1: The system shall load pages within 2 seconds.
   - NFR2.2: The system shall support at least 100 concurrent users.

3. **Security**

   - NFR3.1: The system shall encrypt user passwords.
   - NFR3.2: The system shall use JWT for secure authentication.
   - NFR3.3: The system shall implement role-based access control.

4. **Maintainability**

   - NFR4.1: The system shall use modular architecture to facilitate maintenance.
   - NFR4.2: The system shall follow consistent coding standards.

5. **Scalability**

   - NFR5.1: The system shall be designed to scale as the user base grows.
   - NFR5.2: The database shall be designed to handle increasing data volume.

Use Cases

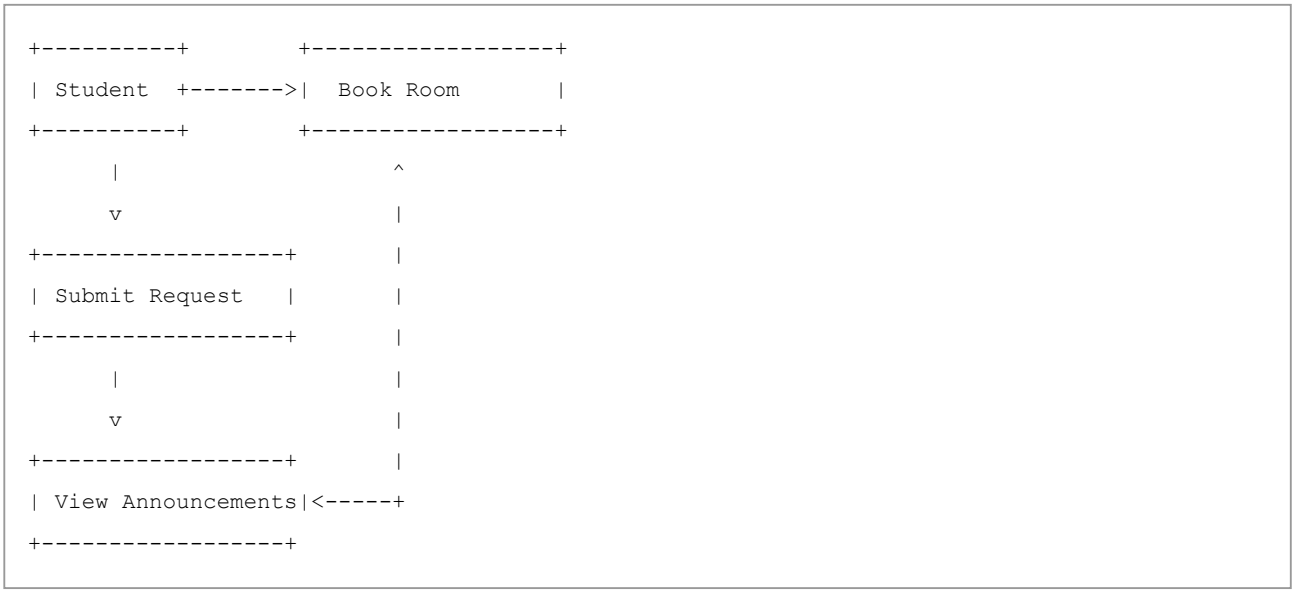
Expanded Use Case Details

- **UC1: User Registration**

  - **Edge Cases:** Email already used, weak password, invalid email format.
  - **Exception Handling:** Return clear error messages for each failure.
  - **Business Rule:** Only valid campus emails allowed (e.g., `@university.edu`).

- **UC2: Room Booking**
    - **Edge Cases:** Double-booking, booking in the past, exceeding room capacity.
    - **Exception Handling:** API returns `409 Conflict` for double-booking.
    - **Alternate Flow:** If booking is for an event, notify admin for approval.
- **UC3: Maintenance Request**
    - **Edge Cases:** Duplicate requests for same issue/location.
    - **Business Rule:** Only allow one open maintenance request per room/issue.
- **UC4: Announcements**
    - **Edge Cases:** Announcement targeting (students, faculty, all).
    - **Business Rule:** Only faculty/admin can post; students can only view.
- **UC5: Generate Reports**
    - **Exception Handling:** If PDF generation fails, return `500` with log.

Use Case Diagram (ASCII):



UC1: User Registration

Actor: Unregistered User

Description: A new user registers in the system.

Preconditions: User is not registered.

Main Flow:

1. User navigates to the registration page.
2. User provides username, password, name, email, and role.
3. System validates the input.
4. System creates a new user account.
5. System generates a JWT token.
6. System redirects user to the dashboard.

Alternative Flow:

- If username or email already exists, system displays an error message.
- If input validation fails, system displays appropriate error messages.

Postconditions: User is registered and authenticated.

UC2: User Login

Actor: Registered User

Description: User logs into the system.

Preconditions: User has a registered account.

Main Flow:

1. User navigates to the login page.
2. User enters username and password.
3. System validates credentials.
4. System generates JWT token.
5. System redirects user to the dashboard.

Alternative Flow:

- If credentials are invalid, system displays an error message.

Postconditions: User is authenticated and has access to role-specific features.

UC3: Create Room Booking

Actor: Student/Faculty/Admin

Description: User books a room for an event or meeting.

Preconditions: User is authenticated.

Main Flow:

1. User navigates to the booking section.
2. User selects a room, date, time, and duration.
3. User enters the purpose of booking.
4. System checks room availability.
5. System creates a booking request.
6. System displays confirmation.

Alternative Flow:

- If room is not available, system displays an error message.
- If user is a faculty/admin, the booking is automatically approved.
- If user is a student, the booking requires faculty approval.

Postconditions: Booking is created and awaiting approval if necessary.

UC4: Approve/Reject Booking Request

Actor: Faculty/Admin

Description: Faculty or admin user reviews and processes booking requests.

Preconditions: User is authenticated with faculty or admin role. Booking request exists.

Main Flow:

1. Faculty/admin navigates to the booking management section.
2. System displays list of pending booking requests.
3. Faculty/admin selects a booking request.
4. Faculty/admin approves or rejects the request.
5. System updates booking status.
6. System notifies the requester.

Alternative Flow:

- Faculty/admin can provide a reason for rejection.

Postconditions: Booking status is updated.

UC5: Submit Maintenance Request

Actor: Any User

Description: User submits a maintenance request for a facility issue.

Preconditions: User is authenticated.

Main Flow:

1. User navigates to the maintenance section.
2. User selects location and issue type.
3. User provides description of the issue.
4. User submits the request.
5. System creates a maintenance ticket.
6. System assigns a reference number.

Alternative Flow:

- User can upload images of the issue.

Postconditions: Maintenance request is created and pending.

UC6: Post Announcement

Actor: Faculty/Admin

Description: Faculty or admin posts an announcement.

Preconditions: User is authenticated with faculty or admin role.

Main Flow:

1. User navigates to the announcements section.
2. User creates a new announcement with title, content, and category.
3. User submits the announcement.
4. System publishes the announcement.
5. System displays the announcement on the dashboard.

Alternative Flow:

- User can schedule the announcement for future publication.
- User can target the announcement to specific user roles.

Postconditions: Announcement is published and visible to target users.

UC7: Generate Report

Actor: Admin

Description: Admin generates system reports.

Preconditions: User is authenticated with admin role.

Main Flow:

1. Admin navigates to the reports section.
2. Admin selects report type (users or bookings).
3. System generates a PDF report.
4. System provides download link.
5. Admin downloads the report.

Alternative Flow:

- Admin can filter report data based on various criteria.

Postconditions: PDF report is generated and downloaded.