**Product Requirements Document (PRD)**

**1. Introduction**

**1.1 Purpose**

This document outlines the requirements for a **web-based Student/RSO Event Management System** to be developed using **Node.js, Express, MySQL**, and a **React** front-end. The system aims to centralize university-related events, manage RSOs (Registered Student Organizations), and facilitate user engagement via event listings, comments, and ratings.

**1.2 Scope**

The system will allow:

* **Super Admins** to manage university profiles and approve public events.
* **Admins** (affiliated with an RSO) to create and manage events for their organizations.
* **Students** to discover events, join RSOs, rate events, and comment on events.

**1.3 Definitions, Acronyms, and Abbreviations**

* **RSO**: Registered Student Organization.
* **PRD**: Product Requirements Document.
* **SRS**: Software Requirements Specification.
* **UI**: User Interface.
* **API**: Application Programming Interface.

**2. Product Overview**

**2.1 Product Description**

The **Student/RSO Event Management System** is a **three-tier application** comprising:

1. **Front-End**: A React-based web interface.
2. **Server-Side**: Node.js/Express REST API for business logic.
3. **Database**: MySQL, where all data (users, RSOs, events, etc.) is stored and constrained.

**2.2 Product Goals**

1. Provide a centralized platform for all **campus-related events** (public, private, RSO-only).
2. Facilitate quick **creation, approval, and discovery** of events.
3. Enable **social interaction** through commenting and event rating.
4. Support **social network integration** for easy event sharing (optional advanced feature).
5. Enforce **university-specific** or RSO-specific visibility rules (public, private, RSO-only).

**3. Functional Requirements**

**3.1 User Roles & Account Management**

1. **User Registration & Login**
   * Users must register with a valid **university email** domain (e.g., @knights.ucf.edu).
   * Authentication is required for all actions except browsing public events.
   * Passwords must be stored in a secure, hashed format (e.g., bcrypt).
2. **Role Management**
   * **Super Admin**:
     + Can create a new university profile (name, location, description, number of students, pictures).
     + Approve or reject new public events.
   * **Admin**:
     + Can create events for the RSO(s) they manage.
     + Can manage RSO membership and details (once the RSO is active).
   * **Student**:
     + Can view events (public, private for their university, RSO events for RSOs they belong to).
     + Can rate and comment on events.
     + Can request to create or join an RSO.

**3.2 RSO (Registered Student Organizations) Management**

1. **RSO Creation**
   * A new RSO requires at least **5 users** with the same university domain. One of these must be designated as the RSO Admin.
   * The system must remain in pending status until at least 5 members join; once the trigger is activated, the RSO becomes active.
2. **RSO Membership**
   * Students can join an existing RSO or request to create a new RSO.
   * System enforces membership constraints via a **database trigger** (minimum 5 members).

**3.3 Event Management**

1. **Event Creation**
   * Admin can create events with details:
     + **Name, Category (social, fundraising, tech talk, etc.)**
     + **Description**
     + **Date & Time**
     + **Location** (map coordinates, address, building, etc.)
     + **Contact Info** (phone, email)
   * Public events require Super Admin **approval**.
   * Private events are visible only to users with the same university domain.
   * RSO events are visible only to members of that RSO.
2. **Event Discovery & Viewing**
   * Students can browse events filtered by type (public, private, RSO) based on their role and university affiliation.
   * Students can see event details including date, time, location, description, etc.
3. **Rating & Commenting**
   * **Comments**: Users can add, edit, or remove their comments.
   * **Ratings**: Each user can rate an event on a 1-5 star scale.
   * All ratings and comments are stored in the database.
4. **Social Media Integration** (Advanced)
   * Users can share or post event links to social platforms (Facebook, Google, etc.).

**3.4 Event Feeds from External Sources**

* System should optionally pull event data from **university event feeds** (e.g., https://events.ucf.edu), parse the RSS/XML/JSON feed, and insert or update the internal event database.

**4. Non-Functional Requirements**

**4.1 Performance**

* The system should support **multiple concurrent users** without significant performance degradation.
* Responses to user queries (e.g., listing events) should generally return within **2 seconds** under average load.

**4.2 Security**

* All passwords must be **hashed** (bcrypt recommended).
* All client-to-server traffic should use **HTTPS** in a production environment.
* Database triggers must enforce business rules (e.g., RSO membership constraints).

**4.3 Reliability & Availability**

* The system should be available **24/7** with minimal downtime (except for planned maintenance).
* Consider using a **connection pool** to handle database requests efficiently.

**4.4 Maintainability**

* Codebase should be organized by **layer** (front-end, back-end, data access) with clear separation.
* Follow best practices (e.g., consistent naming conventions, linting, code reviews).

**4.5 Scalability**

* The architecture should allow for scaling horizontally or vertically:
  + Potential to **move the database** to a managed cloud database (AWS RDS, Azure Database for MySQL).
  + Potential to run **multiple Node.js instances** behind a load balancer.

**5. System Architecture**

**5.1 Overview**

1. **Front-End (React)**
   * SPA (Single Page Application) served on port 3000 (by default).
   * Communicates with the server through a RESTful API (JSON).
2. **Back-End (Node.js & Express)**
   * Listens on port 5000 (by default).
   * Handles REST endpoints for authentication, event management, RSO management, etc.
   * Contains business logic for role checks, input validation, etc.
3. **Database (MySQL)**
   * Stores all core entities: Users, RSOs, Events, Comments, Ratings, etc.
   * Enforces constraints (foreign keys, triggers, indexes) to maintain data integrity.

**5.2 Data Flow**

1. **User** sends a request to the **React** front-end (e.g., /login).
2. React calls the **Express** server endpoint (e.g., POST /api/auth/login).
3. Express server **validates** credentials, queries the **MySQL** database.
4. Server returns **JSON** response (success/failure), and front-end updates UI.

**6. Data Model**

**6.1 Key Entities**

1. **User**
   * user\_id (PK), username, email, password, role (superadmin, admin, student), university\_id (FK), etc.
2. **University**
   * university\_id (PK), name, location, description, num\_students, pictures, etc.
3. **RSO**
   * rso\_id (PK), name, description, status (pending or active), university\_id (FK), etc.
4. **RSO\_Members**
   * A junction table: rso\_id (FK), user\_id (FK), plus membership timestamps.
   * **Trigger** to activate RSO once membership count >= 5.
5. **Event**
   * event\_id (PK), name, category, description, time, date, location, contact\_phone, contact\_email, visibility (public, private, rso), rso\_id (FK, nullable), university\_id (FK), approved\_by (FK to user\_id if public), etc.
6. **Comment**
   * comment\_id (PK), event\_id (FK), user\_id (FK), text, timestamp, etc.
7. **Rating**
   * rating\_id (PK), event\_id (FK), user\_id (FK), stars (1-5), etc.

**6.2 Constraints & Triggers**

* **RSO Activation Trigger**: If RSO\_Members for a given RSO >= 5, the system updates rsos.status to active.
* **Foreign Key Constraints**: Ensure referential integrity across all entities.
* **Check Constraints** or validations for rating range (1 to 5), event date/time (must be valid future date), etc.

**6.3 Indexing**

* **Indexes** on fields commonly used in queries:
  + user\_id, university\_id in users, rso\_members, etc.
  + Composite index on (date, location) for the events table to speed up queries by date + location.

**7. User Interface & UX**

**7.1 Front-End Pages**

1. **Home / Event Listing**:
   * Lists upcoming events (filter: public, private, user’s RSO events).
   * Search by date, category, location.
2. **Event Details**:
   * Shows event name, date/time, location, description, comments, ratings.
   * Allows logged-in users to add/edit comments or post a rating.
3. **RSO Management**:
   * Join an existing RSO or create a new one.
   * View RSO details, list of members, etc. (Admins have extra controls).
4. **Admin Dashboard**:
   * Create new events or edit existing events.
   * Approve new members.
   * For super admin: Approve public events, manage universities.
5. **User Profile**:
   * Manage personal information, see user’s comments, ratings, and RSO memberships.

**7.2 Responsiveness**

* UI must be **responsive** to support desktop and mobile views.
* React or a UI library (e.g., **Material-UI**, **Bootstrap**) can simplify styling.

**8. Security & Privacy**

1. **User Authentication**
   * Passwords stored via hashing (bcrypt).
   * Sessions or JWT-based tokens for stateful/stateless authentication.
2. **Role-Based Authorization**
   * Middleware checks (Node/Express) to ensure only Admins can create events, only Super Admin can approve public events, etc.
3. **HTTPS** in Production
   * Protect data in transit from interception or tampering.
4. **Data Privacy**
   * Limit the personal data stored to only what is necessary.
   * Enforce standard practices (SQL injection prevention, parameterized queries).

**9. Deployment & Environment**

**9.1 Development Environment**

* **Local** with Node.js v16+ or v18+, MySQL 8+.
* **React** dev server on port 3000, Node/Express on port 5000.

**9.2 Staging/Testing Environment**

* May use Docker containers for reproducibility:
  + **node** container + **mysql** container.
* Automated CI/CD pipeline recommended (GitHub Actions, GitLab CI, etc.).

**9.3 Production**

* Deploy Node/Express on a suitable host (AWS EC2, Heroku, Azure App Service, etc.).
* Deploy MySQL on a managed service (e.g., AWS RDS) or a secure VM.
* Serve React’s static bundle either via Node or a separate CDN.

**10. Testing & Validation**

**10.1 Unit Testing**

* **Backend**: Use frameworks like **Jest** or **Mocha** for controllers, models.
* **Frontend**: Use **React Testing Library** or **Enzyme** for UI components.

**10.2 Integration Testing**

* Test **API endpoints** with a tool like **Postman** or **supertest** (Node library).

**10.3 Acceptance Testing**

* Validate all functional requirements:
  + RSO creation requires 5 members.
  + Public event approvals require Super Admin.
  + Students can comment and rate, etc.

**10.4 Performance & Load Testing**

* Tools like **JMeter** or **k6** to simulate concurrent users.
* Ensure the system meets response time targets.

**11. Timeline & Milestones**

| **Milestone** | **Deliverables** | **Target Date** |
| --- | --- | --- |
| **1. Design & DB Setup** | ER diagrams, DB schema, initial MySQL setup, triggers | TBD |
| **2. Basic Backend Endpoints** | Node/Express project, basic CRUD for events, users, RSOs | TBD |
| **3. Front-End Integration** | React app with event listing, user login, RSO page, API calls | TBD |
| **4. Role & Auth** | Authentication (JWT or session), role-based access | TBD |
| **5. Comments & Ratings** | Endpoints & UI for adding/editing/removing comments, star ratings | TBD |
| **6. Admin & Super Admin** | Dashboards for event approval, RSO management, university profiles | TBD |
| **7. Testing & QA** | Unit, integration, performance testing | TBD |
| **8. Deployment** | Production deployment, domain name, SSL setup | TBD |
| **9. Maintenance** | Ongoing patching, feature enhancements | - |

**12. Risks & Assumptions**

**12.1 Risks**

1. **Data Security**: Sensitive user data might be exposed if not properly secured.
2. **Complex Role Logic**: Potential errors if role-based rules (super admin vs. admin vs. student) are not implemented carefully.
3. **Trigger Constraints**: Misconfigured triggers or foreign keys could lead to data integrity issues.

**12.2 Assumptions**

1. Users have stable internet access; offline mode is not required.
2. The system does not handle payments (e.g., for event tickets).
3. Only a single university domain is handled at a time, or each user belongs to exactly one university domain. (If multi-university is needed, the design can be extended.)

**13. Approvals**

| **Name / Role** | **Signature** | **Date** |
| --- | --- | --- |
| **Project Sponsor** |  |  |
| **Technical Lead** |  |  |
| **Development Team** |  |  |
| **Quality Assurance** |  |  |

**Conclusion**

This document captures the **functional and non-functional requirements** for the Student/RSO Event Management System, outlining the **Node.js + Express + MySQL** and **React**-based architecture. By adhering to these requirements, the development team ensures a robust, secure, and user-friendly platform that meets the needs of super admins, admins, and student users alike.