Sports Day LLD

1. Overview

The Sports Day Event Registration System is a web application designed to facilitate the registration process for various sporting events. It allows users to view available events, register for their chosen events, and manage their selections in an intuitive and user-friendly interface. This system aims to enhance the experience of participants during sports day events by providing a seamless way to interact with event information.

1.1 Scope

The scope of the project includes the following key functionalities:

1. **User Management**:
    - **User Registration**: Allows users to create an account by providing a unique username, password, and email or phone number.
    - **User Login**: Enables users to log in using their unique username and password.
2. **Event Management**:
    - **Event Listing**: Displays a list of available sports events in a user-friendly card format, including details such as event name, category, and timings.
    - **Event Registration**: Permits users to register for events, with constraints such as limiting registrations to a maximum of three events and preventing overlapping time registrations.
    - **Event Unregistration**: Allows users to unregister from events they no longer wish to participate in.
3. **User Interface**:
    - **Responsive Design**: Provides a visually appealing and responsive web interface for users to navigate seamlessly.
    - **Error Handling**: Incorporates robust error handling to provide meaningful feedback for various user actions, such as duplicate registrations and login failures.
4. **Backend Services**:
    - **API Development**: Develops RESTful APIs to support user and event management functionalities.
    - **Database Integration**: Utilizes a relational database DynamoDb to store user data, event details, and registration information efficiently.
5. **Security Considerations**:
    - Ensures secure communication through HTTPS.
    - Implements user authentication and data validation to protect sensitive information.

1.2 Out Of Scope

The project will not cover:

- Advanced features like user role management or administrative dashboards.
- Integration with external payment systems or complex event analytics.
- Advanced user authentication and data validation.
- Mobile application development; the focus is solely on a web-based solution.

2. Architecture Overview

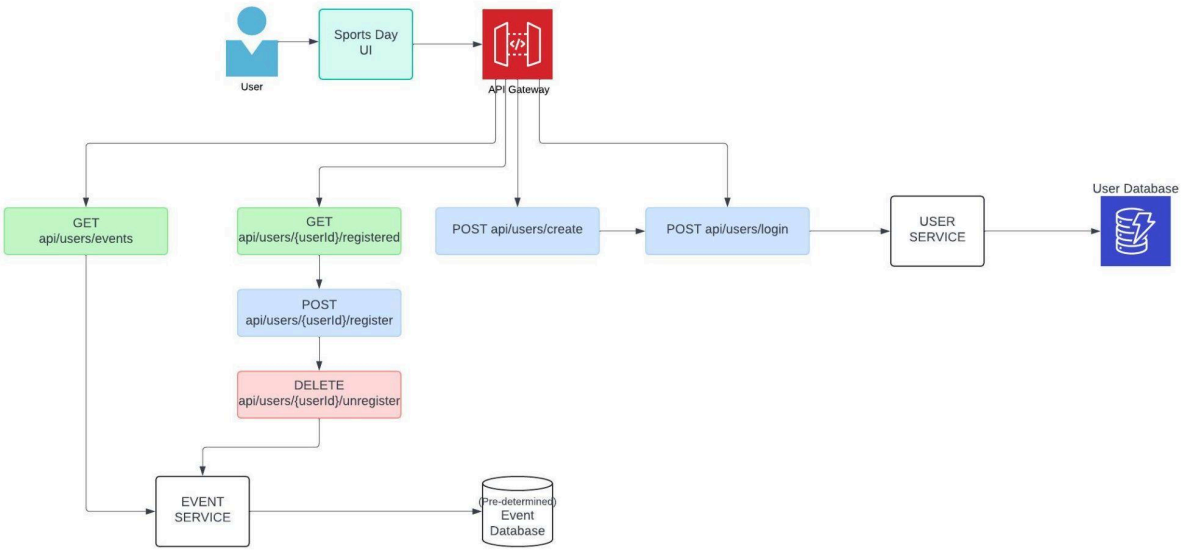

Fig1: High Level Flow of Sports day Application

2.1 Client Layer:

- **User**: Represented as user icons indicating end-users accessing the application.
- **Sports Day UI**: A block representing the React JS application making HTTP requests.

2.2 API Layer:

- **API Gateway**: A block representing the API Gateway that routes requests.
- **Create User API**: POST /api/users/create
- **User Login API**: POST /api/users/login
- **Get All Events API**: GET /api/users/events
- **Get All Registered Events API**: GET /api/events/{userId}/registered
- **Register Events API**: PUT /api/events/{userId}/register
- **Unregister Events API**: PUT /api/events/{userId}/unregister
- **Microservices**: Block representing the User Service and Event Service handling their respective functionalities.

2.3 Database Layer:

- **User Database**: Block representing the database where user data is stored such as username, password, email, phone number and the events registered by the user.
- **Event Database:** Predetermined database that returns a list of events and its details.

3. UI Components

3.1. Sports Day UI (ReactJS)

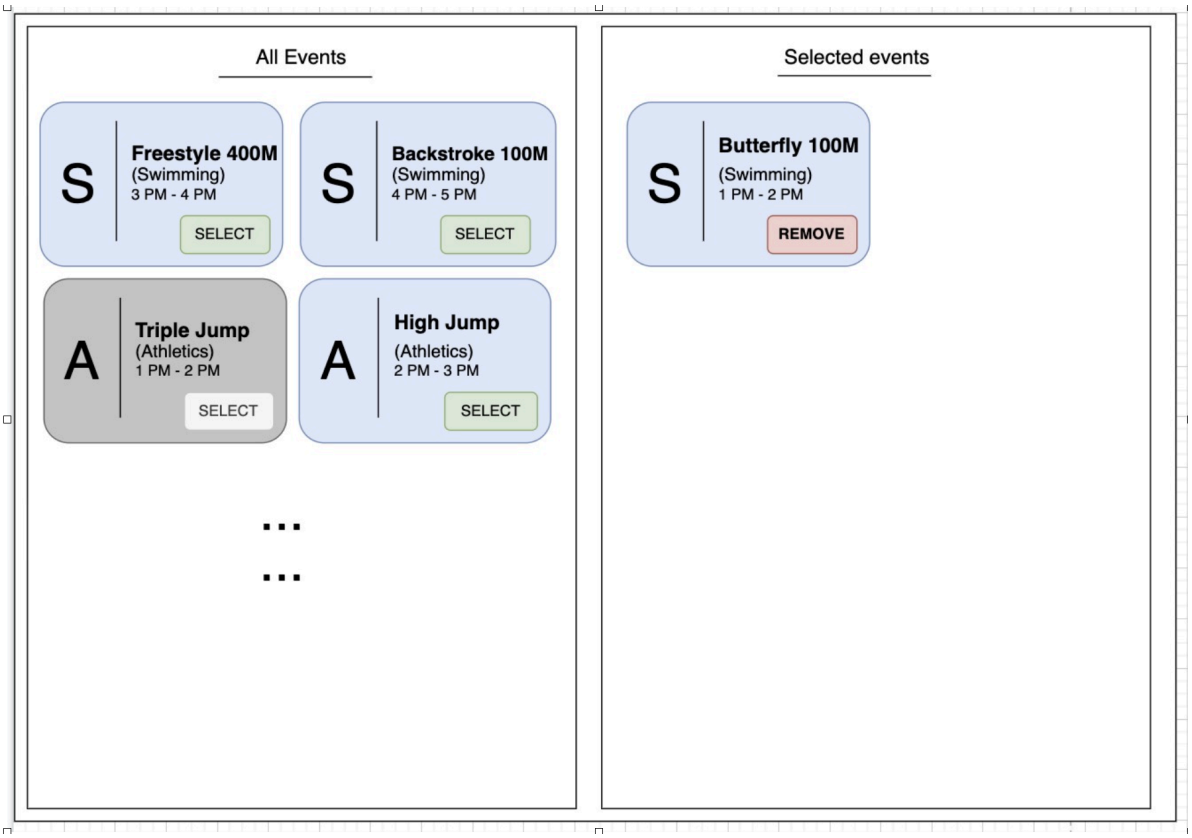

Fig2: UI Mocks

3.1.1 Components:

- **CreateUser**: Handles user registration.
  - Inputs: Username, Password, Email/Phone
  - Actions: Submit registration, navigate to login
- **LoginUser**: Handles user login.
  - Inputs: Username, Password
  - Actions: Submit login, display error messages
- **All Events**: Displays all available events.

        - Properties: List of events
        - Actions: Select an event
    - **Selected Events/Registered Events**: Displays the events a user has registered for.
        - Properties: List of registered events
        - Actions: Deselect an event
    - **EventCard**: Represents each event in a card format.
        - Properties: Event details (name, category, timings)
        - Actions: Select/Deselect event

4. Api Design

4.1 User Service APIs:

- **POST /api/users/create**: Create user.

  Request: Username, Password, Email/Phone, First Name, Last Name

```
{
"username": "john_doe",
"email": "john@example.com",
"password": "securePassword123",
"firstName": "John",
"lastName": "Doe",
"phoneNumber": "5678938393",
}
```

  Response: Success/Error message

```
{ "status": "success", "message": "User registered successfully"}
```

```
{
"status": "error",
"message": "Email already in use"
}
```

```
{
"status": "error",
"message": "username already exists"
}
```

- **POST /api/users/login**: Login user.

Request: Username, Password

```
{ "usernameOrEmail": "john_doe", "password": "securePassword123" }
```

Response: Success/Error message

```
{
"status": "success",
"message": "Login successful"
}
```

```
{ "status": "error", "message": "Invalid username or password" }
```

4.1 Event Service APIs:

- **PUT /api/events/{userId}/register**: Register events.

  Request: username, eventId .

```
{ "username": "123456", "eventId": "1" }
```

  Response: List of registered events.

```
{
"status": "success",
"message": "Successfully registered for the event"
}
```

- **GET /api/events/{userId}/registered**: Get registered events for a user.

  Response: List of registered events.

```
{
"status": "success",
"registeredEventsList": [ {
"event_name": "Butterfly 100M",
"event_category": "Swimming",
"start_time": "2022-12-17 13:00:00",
"end_time": "2022-12-17 14:00:00"
},
{
```

```
"id": 5,
"event_name": "Triple Jump",
"event_category": "Athletics",
"start_time": "2022-12-17 16:00:00",
"end_time": "2022-12-17 17:00:00"
} ]
}
```

```
{ "status": "error", "message": "User not found" }
```


- **DELETE /users/{userId}/events/{eventId}**: Unregister from an event.

  Request: Username, Event ID


```
{ "username": "123456", "eventId": "1" }
```


  Response: Success/Error message


```
{
"status": "success",
"message": "Successfully unregistered from the event",
"eventId": "1"
}
```

```
{
"status": "error",
"message": "User not registered for this event" }
```


- **GET /events**: Get all pre-determined events.

  Response: List of events.


```
[
{
"id": 1,
"event_name": "Butterfly 100M", "event_category": "Swimming", "start_time":
"2022-12-17 13:00:00",
"end_time": "2022-12-17 14:00:00"
},
{
"id": 2,
```

```
"event_name": "Backstroke 100M", "event_category": "Swimming",
"start_time": "2022-12-17 13:30:00",
"end_time": "2022-12-17 14:30:00"
},
{
"id": 3,
"event_name": "Freestyle 400M", "event_category": "Swimming", "start_time":
"2022-12-17 15:00:00",
"end_time": "2022-12-17 16:00:00"
},
{
"id": 4,
"event_name": "High Jump", "event_category": "Athletics", "start_time":
"2022-12-17 13:00:00",
"end_time": "2022-12-17 14:00:00"
},
{
"id": 5,
"event_name": "Triple Jump", "event_category": "Athletics", "start_time":
"2022-12-17 16:00:00",
"end_time": "2022-12-17 17:00:00"
},
{
"id": 6,
"event_name": "Long Jump", "event_category": "Athletics", "start_time":
"2022-12-17 17:00:00",
"end_time": "2022-12-17 18:00:00"
},
{
"id": 7,
"event_name": "100M Sprint", "event_category": "Athletics", "start_time":
"2022-12-17 17:00:00",
"end_time": "2022-12-17 18:00:00"
},
{
"id": 8,
"event_name": "Lightweight 60kg", "event_category": "Boxing", "start_time":
"2022-12-17 18:00:00",
"end_time": "2022-12-17 19:00:00"
},
{
"id": 9,
"event_name": "Middleweight 75 kg", "event_category": "Boxing",
```

```
"start_time": "2022-12-17 19:00:00",
"end_time": "2022-12-17 20:00:00"
},
{
"id": 10,
"event_name": "Heavyweight 91kg", "event_category": "Boxing", "start_time":
"2022-12-17 20:00:00",
"end_time": "2022-12-17 22:00:00"
}
]
```


5. Database Design

Since Get All Events API has a predetermined response. We will be creating a Database for Users only.

Users

- **Table Name**: `users`
- **Attributes**:
  - `username` (Primary Key, VARCHAR, Unique)
  - `email` (VARCHAR, Unique)
  - `password_hash` (VARCHAR)
  - `first_name` (VARCHAR)
  - `last_name` (VARCHAR)
  - `created_at` (DATETIME)
  - `updated_at` (DATETIME)
  - `Registered_events_list` (JSON) : A field to store event IDs as a JSON array or comma-separated list.

6. Error handling and Validations

6.1 User Registration

Validations:

- **Username**:
  - Must be between 3 and 30 characters.
  - Must be unique (check against the database).
  - Must not contain special characters (except underscores).
- **Email**:
  - Must be a valid email format (use regex for validation).
  - Must be unique (check against the database).

- **Password**:
    - Must be at least 8 characters long.
    - Must contain at least one uppercase letter, one lowercase letter, one number, and one special character.
- **First Name & Last Name**:
    - Must not be empty and should only contain alphabetic characters.

Error Handling:

- **Duplicate Username or Email**:
    - Return an error response with a message: `"Username or email already in use."`
- **Validation Errors**:
    - Return a 400 Bad Request status with a structured error response detailing which fields failed validation.

6.2 User Login

Validations:

- **Username or Email**:
    - Must not be empty.
- **Password**:
    - Must not be empty.

Error Handling:

- **Invalid Credentials**:
    - Return an error response with a message: `"Invalid username or password."`
- **User Not Found**:
    - Return a 404 Not Found status with a message: `"User not found."`

6.3 Event Registration

Validations:

- **User ID**:
    - Must be a valid user (check existence in the `users` table).
- **Event ID**:
    - Must ensure the event is not full (check the current number of registrations against the capacity).

Error Handling:

- **User Not Found**:

    - Return a 404 Not Found status with a message: `"User not found."`
- **Event Full**:
    - Return a 400 Bad Request status with a message: `"Event is full, registration failed."`
- **Already Registered**:
    - Return a 400 Bad Request status with a message: `"User is already registered for this event."`

6.4 Unregistering from an Event

Validations:

- **User ID**:
    - Must be a valid user (check existence in the users table).
- **Event ID**:
    - Must check if the user is currently registered for the event.

Error Handling:

- **User Not Found**:
    - Return a 404 Not Found status with a message: `"User not found."`

7. Testing Strategy

7.1 User Registration Testing

- **Automated Unit Testing**:
    - Using frameworks **Jest** (for React) and **JUnit** (for Java) to write unit tests for each validation function (e.g., username uniqueness, password strength).
- **Integration Testing**:
    - Use **Postman** or **REST-assured** to test the API endpoints for user registration. Validate responses for valid and invalid inputs.
- **Manual Testing**:
    - Perform manual tests for user registration, simulating various scenarios to verify the UI behavior and messages.

7.2 User Login Testing

- **Automated Unit Testing**:
    - Write unit tests to validate the authentication logic and error handling.
- **Integration Testing**:
    - Use **Postman** or **Insomnia** to test the login API with valid and invalid credentials, checking for correct status codes and messages.

- **Manual Testing**:
    - Perform manual tests to verify the login process, including testing with different browsers and devices.

7.3 Event Management Testing

- **Automated Functional Testing**:
    - Use tools like **Cypress** or **Selenium** to automate end-to-end tests for viewing, registering, and unregistering events.
- **API Testing**:
    - Use **Postman** to test the API endpoints for registering and unregistering from events, validating that the expected outcomes occur.

8. Scalability

- We can use load balancers to distribute incoming traffic across multiple server instances, ensuring no single server becomes a bottleneck.
- We can implement rate limiting to control the number of requests a user can make to the API within a specific timeframe. This helps protect the application from abuse and ensures fair resource usage among all users.
- We can utilize message queues (e.g., RabbitMQ, Kafka) to decouple services and handle high volumes of requests efficiently. This allows for better handling of spikes in traffic.
- We can implement monitoring tools (e.g., Prometheus, Grafana) to track application performance and resource usage. Set up alerts for critical metrics (e.g., response times, error rates) to proactively address potential issues.
- We can implement API versioning to allow for backward compatibility as the application evolves. This ensures that existing users are not disrupted by new changes while enabling the introduction of new features and improvements.