Sports Day HLD

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.

2. Assumptions

- “Select” button is “Register” button. Similarly, “Unselect” button is referred to as “Un-register”.
- Each user is assumed to have a single active session at a time and will not be logged in from multiple devices simultaneously.
- The events are pre-populated in the system and do not require dynamic creation or management through an administrative interface.
- Event timings are assumed to be accurate and do not require real-time updates or synchronization with an external time source.
- The system is assumed to handle a moderate number of concurrent users, with scalability considered for future growth.

3. Functional Requirements

3.1. User Management

3.1.1 User Registration

- Users should be able to create an account by providing a unique user ID, password and an email address or phone number.
- Users should be able to go to the login page from the account creation page if the user already has one account.

3.1.2 User Login

- Users should be able to login using their unique user ID and password.
- Users should be able to go to the account creation page from the login page if the user already does not have any account.

3.2. Event Registration

3.2.1 Register for an Event

- Users should be able to register for an event by clicking a "Select" button associated with the event on the left hand side of the screen.
- Users should not be able to register for the same event more than once.
- Users should not be able to register more than 3 events at once.
- Users should not be able to register for events with conflicting timings.
- Users should not be able to register the same event twice.

3.2.2 View Registered Events

- Users should be able to view all the events they registered on the right hand side of the screen.
- The registered events list shall include the same details as the event listing.

3.2.3 Unregister from an Event

- Users should be allowed to unregister from an event by clicking a "Deselect" button associated with the registered event.

3.3. User Interface

3.3.1 Event List Display

- The UI shall display all available events in a card format in the left hand side of the screen.
- Each card shall show the event name, category and timings, along with a button to select the event.

3.3.2 Selected Events Display

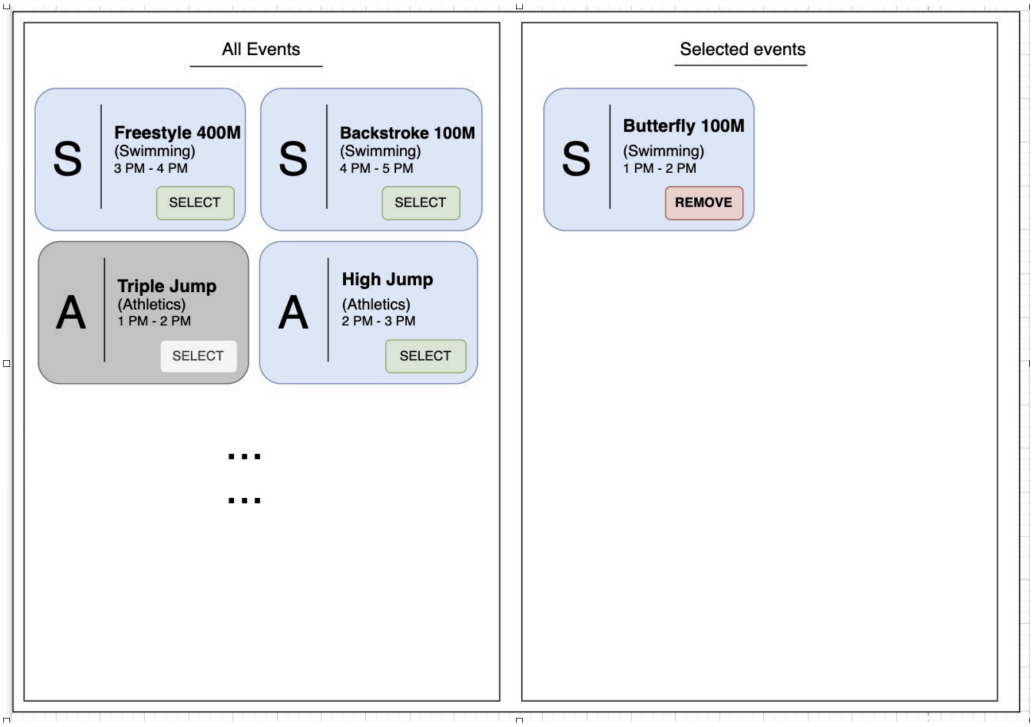
- The UI shall display a separate section for the user's registered events in a card format on the right hand side of the screen.
- Each registered event shall include the event name, category, and timings and a button to deselect the event.

3.4. Error Handling

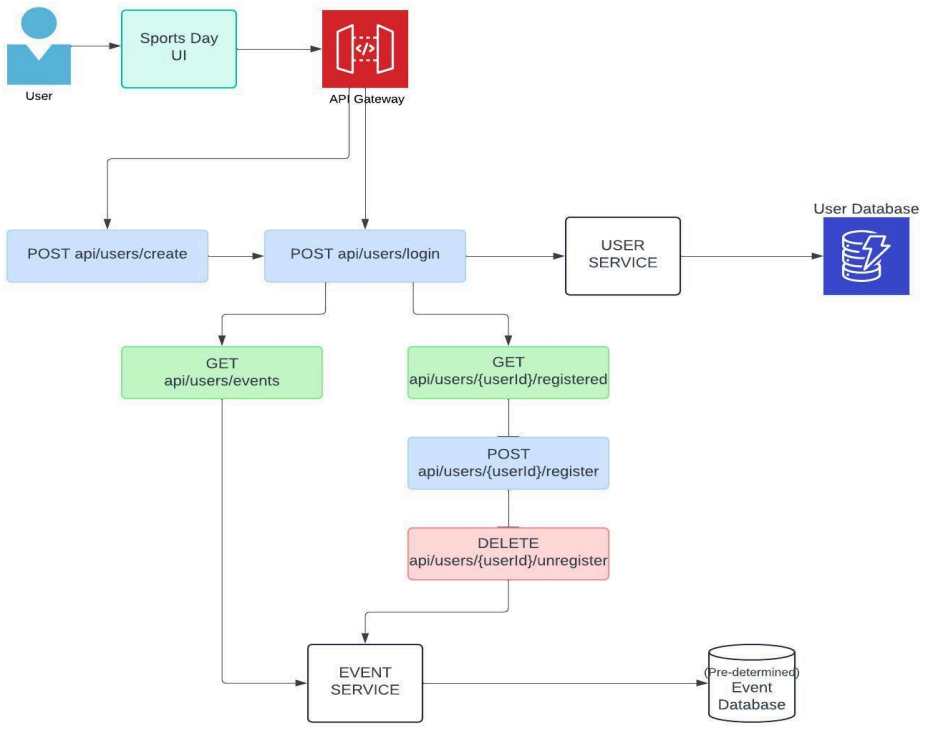
3.4.1 User Feedback

- The system shall provide appropriate error messages for various scenarios, including:
    - Attempting to register for an already registered event.
    - Providing a duplicate user ID during registration.
    - Attempting to log in with an unrecognized user ID.
    - Attempting to create another account with the same email/phone number.
    - Selecting more than 3 events per user
    - Selecting 2 events of overlapping time range.

4. UI Mocks



5. Architectural Design



5.1 Components

5.1.1 Client Layer:

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

5.1.2 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.

5.1.3 Database Layer:

- **DynamoDB**: Block representing the database where user and event data is stored.

5.2 Flow

- User will be interacting with the Sports day UI.
- User will be first redirected to the Login Page. If the user is not already registered go to the login page with a url link in the page that will point to Create User API.
- If the user Creates an account, the username and password gets stored in User Database along with email/phone number.
- When the user logs in, we authenticate the password stored in User Database.
- After the user logs in, the user will be able to see all the events happening on sports day on the left hand side of the screen using the Get All Events API . (4)
- If the user registers to any of the events, the events will be stored in the User Database using Register Events API.
- We will be fetching all the Registered Events from Get All Registered Events API which will be shown on the right hand side of the screen. (4)
- If the user deregisters any event then the deleted event will be deleted from the User Database using Unregister Events API.

6. Database Design

6.1 Overview

For the project, we are evaluating two popular database options: **PostgreSQL** and **DynamoDB**. The goal is to determine which database is best suited for our application's requirements, considering factors such as performance, scalability, future usability and ease of use.

6.2 Comparison Criteria

We compared PostgreSQL and DynamoDB based on the following criteria:

1. Data Model
2. Scalability
3. Performance
4. Transaction Support
5. Cost
6. Community and Support
7. Integration

6.3 Comparison Table

Feature	**PostgreSQL**	**DynamoDB**
Data Model	Relational (SQL)	NoSQL (Key-Value, Document)
Scalability	Horizontal scaling	Fully managed, auto-scaling
Performance	Optimized for complex queries	Optimized for high-throughput workloads
Transaction Support	Yes (ACID compliant)	Limited (supports transactions with some restrictions)
Cost	Open-source (free)	Pay-per-request pricing (can become costly with high traffic)
Documentation and Support	Strong community support	Supported by AWS, extensive documentation
Integration	Excellent with various technologies	Seamless integration with AWS services

6.4 Detailed Analysis

1. **Data Model**:
    - PostgreSQL is a relational database that uses structured data and supports SQL. It is ideal for applications requiring complex queries and relationships.
    - DynamoDB is a NoSQL database that offers a flexible schema, allowing for key-value and document storage. This makes it suitable for applications with unstructured or semi-structured data.
2. **Scalability**:
    - PostgreSQL can scale horizontally but often requires more manual intervention for clustering and replication.
    - DynamoDB is designed for seamless horizontal scaling and can automatically handle increases in traffic without manual configuration.
3. **Performance**:
    - PostgreSQL excels in handling complex queries and transactions, making it suitable for applications that need strong consistency and data integrity.
    - DynamoDB is optimized for high-throughput workloads and provides low-latency performance, particularly for read and write operations.
4. **Transaction Support**:
    - PostgreSQL is fully ACID compliant, ensuring reliable transactions which is crucial for applications that require data integrity.
    - DynamoDB supports transactions, but there are some limitations on how they can be used (e.g., constraints on the number of items).
5. **Cost**:
    - Both PostgreSQL is open-source and free to use, although hosting and management costs apply.
    - DynamoDB uses a pay-per-request pricing model, which can become costly as traffic increases, particularly if not managed properly.
6. **Documentation and Support**:
    - PostgreSQL has a strong community with extensive documentation, forums, and third-party resources available.
    - DynamoDB is backed by AWS, providing robust documentation and support, as well as integration with other AWS services.
7. **Integration**:
    - PostgreSQL integrates well with various technologies and frameworks, making it versatile for different application architectures.
    - DynamoDB offers seamless integration with AWS services, making it a good choice for applications already within the AWS ecosystem.

6.5 Recommendation

Based on the evaluation, it is recommended using DynamoDb for this project.

6.6 Justification

- DynamoDB's ability to automatically scale with traffic ensures that our application can handle unpredictable loads efficiently, making it ideal for high-traffic environments which will help us in future usability.
- Its flexible data model allows for quick adjustments to the data schema without downtime, which is advantageous for projects that may evolve over time.
- The seamless integration with AWS services can enhance our application's capabilities, allowing us to leverage other AWS features like Lambda for serverless architectures or S3 for storage.

In conclusion, DynamoDB provides the scalability and performance necessary to meet the demands of our project effectively.

7. Scalability

Microservices Architecture:

- Split your application into microservices to handle different functionalities independently (e.g., user service, event service). This allows for scaling specific services based on demand.

Load Balancing:

- Implement load balancers to distribute incoming traffic evenly across multiple server instances, ensuring no single server becomes a bottleneck.

Caching:

- Implement caching mechanisms (e.g., Redis or Memcached) to reduce database load by storing frequently accessed data in memory.
- Use a Content Delivery Network (CDN) to cache static assets, improving load times for users globally.

Asynchronous Processing:

- Use message queues (e.g., RabbitMQ, AWS SQS) for tasks that can be processed asynchronously, such as sending notifications or processing registrations, allowing your application to remain responsive.

API Rate Limiting:

- Implement rate limiting to prevent abuse and ensure fair usage of APIs. This helps maintain performance during peak times.

Auto-scaling:

- Utilize cloud features that allow for auto-scaling of resources based on traffic, ensuring that you have enough resources during high-demand periods and scaling down during low usage times.

Monitoring and Performance Metrics:

- Implement monitoring tools (e.g., AWS CloudWatch, Prometheus) to track performance metrics and system health, allowing you to identify and address bottlenecks proactively.

Database Indexing:

- Optimize your database queries by implementing proper indexing, ensuring fast data retrieval as your dataset grows.

Micro-Frontends:

- Consider using a micro-frontend architecture if you have a complex UI, enabling independent development and scaling of UI components.

8. Future Scope

- Implement multi-factor authentication (MFA) for increased security.
- Allow users to update their profiles
- Provide users with the ability to create and manage their own events.
- Include features for event organizers to send notifications or updates to registered users.
- Implement push notifications for upcoming events or changes.
- Send email alerts for registration confirmations or reminders.
- Enhance the events list with advanced search and filtering options (e.g., by date, category, location).
- Allow users to bookmark or favorite events for quick access.
- Develop a dedicated mobile application to enhance user experience.
- Consider integrating with third-party services for payment processing, ticketing, or event promotion.
- Explore API integrations with platforms like Calendars or social media for event sharing.