Event Registration System

1. Introduction

1.1 Purpose

This Software Design Document (SDD) outlines the design and architecture of the Event Registration System, a web-based application that enables event organizers to create and manage events and allows participants to register for them. The document serves as a blueprint for developers, detailing the system's components, architecture, and implementation guidelines.

1.2 Scope

The Event Registration System supports the following core features:

- Create events with limited seats.
- Register to attend an event.
- View upcoming events.
- Track registered participants.
- Cancel or reschedule events.

The system targets two primary user roles: Event Organizers and Participants. This SDD includes placeholders for Software Requirements Specification (SRS) diagrams, detailed project documentation, implementation details, Object Constraint Language (OCL) specifications, (Api's) Application Programming Interface, Aspect-Oriented Programming (AOP) considerations, and microservices architecture.

1.3 Definitions, Acronyms, and Abbreviations

- **SRS**: Software Requirements Specification
- **SDD**: Software Design Document
- **OCL**: Object Constraint Language
- **AOP**: Aspect-Oriented Programming
- **API**: Application Programming Interface
- **ERD**: Entity-Relationship Diagram
- **REST**: Representational State Transfer

1.4 References

- REST API Guidelines: https://restfulapi.net
- OCL Specification: https://www.omg.org/spec/OCL
- Microservices Architecture: https://microservices.io

2. System Overview

The Event Registration System is a web-based application built using a microservices architecture to ensure scalability and maintainability. It allows event organizers to create and manage events with limited seats and participants to register for events. The system provides a user-friendly interface for viewing upcoming events, tracking registrations, and handling event modifications (cancellation or rescheduling).

2.1 Target Users

- **Event Organizer**: Creates and manages events, tracks participants, cancels or reschedules events.
- **Participant**: Views upcoming events, registers for events.

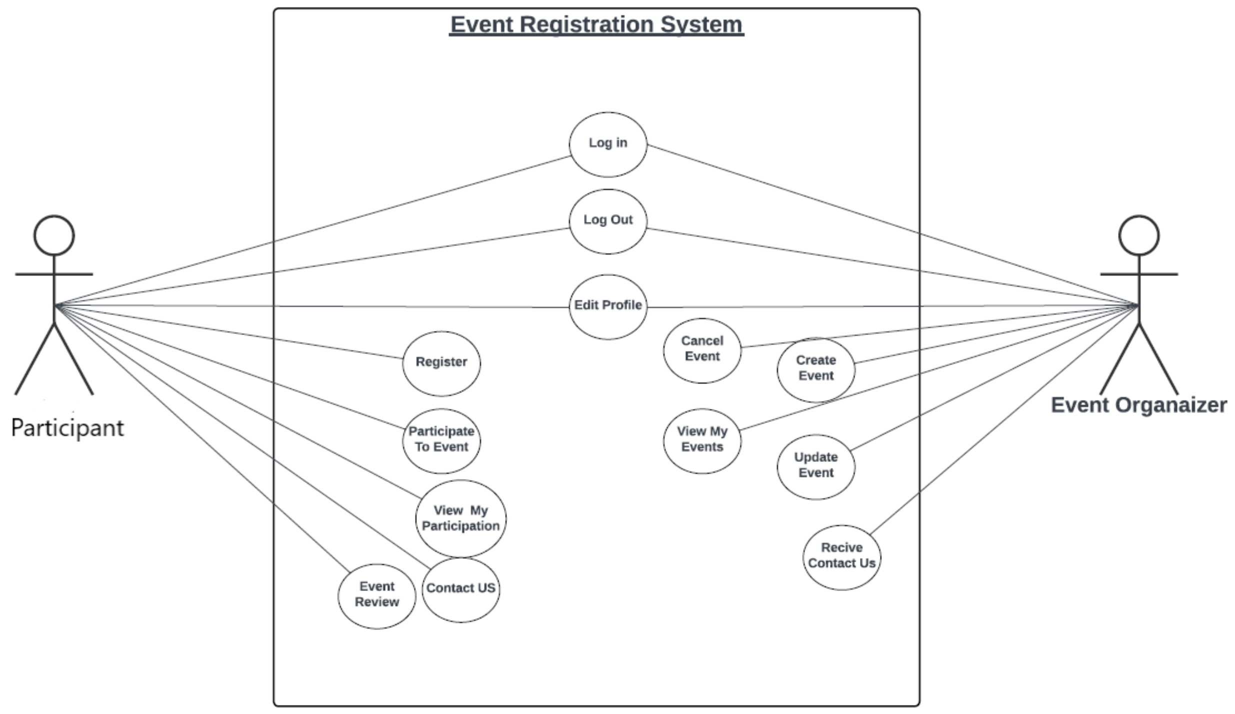
2.2 Core Features

- **Event Creation**: Organizers can create events with details such as name, date, location, and seat limits.
- **Event Registration**: Participants can register for events, with validation for seat availability.
- **Event Viewing**: Users can view a list of upcoming events.
- **Participant Tracking**: Organizers can view registered participants for each event.
- **Event Management**: Organizers can cancel or reschedule events, notifying registered participants.

3. SRS Components (Placeholders)

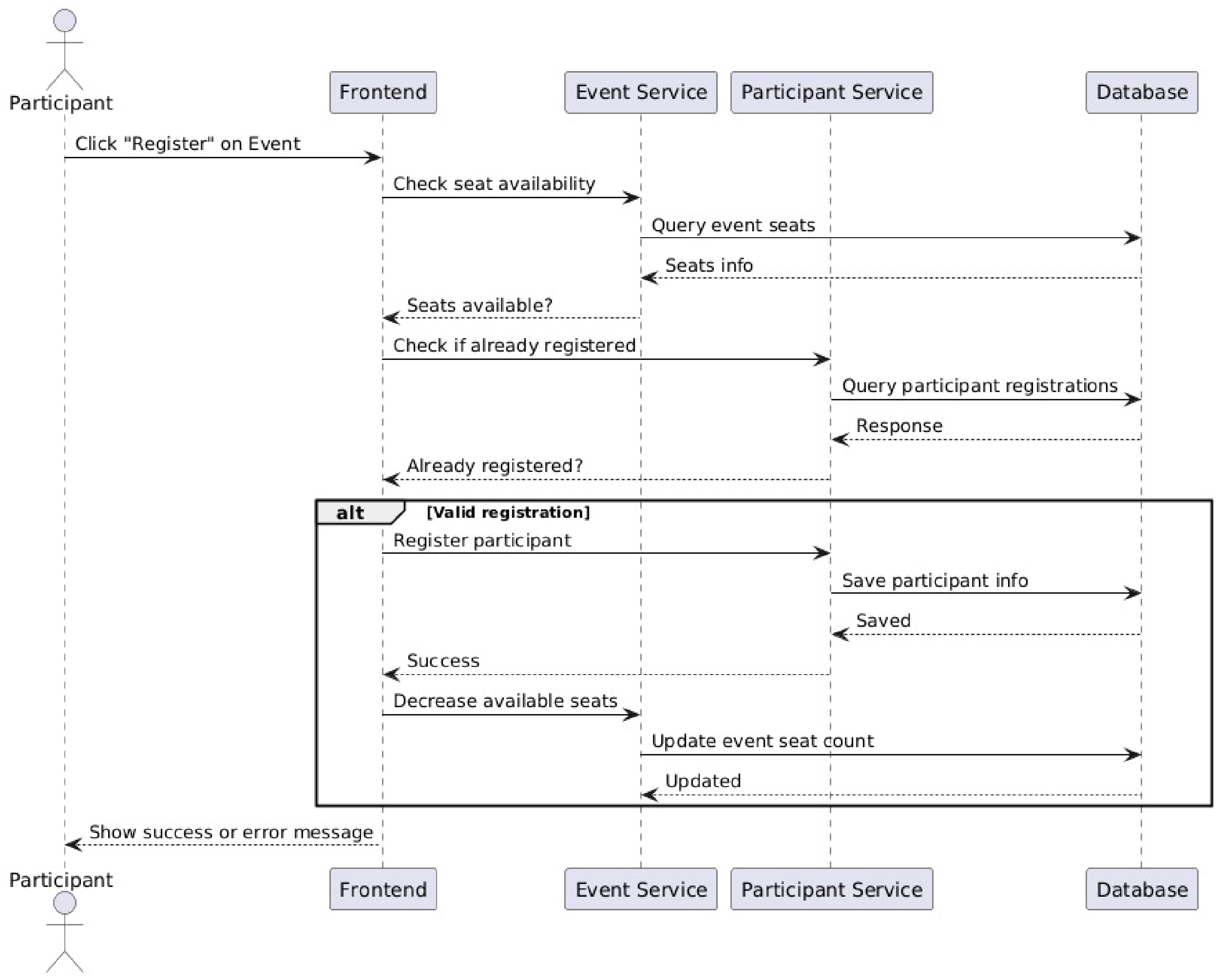
The following diagrams are part of the Software Requirements Specification (SRS) and will be developed to capture the system's requirements. Placeholders are provided here, to be replaced with actual diagrams during the detailed design phase.

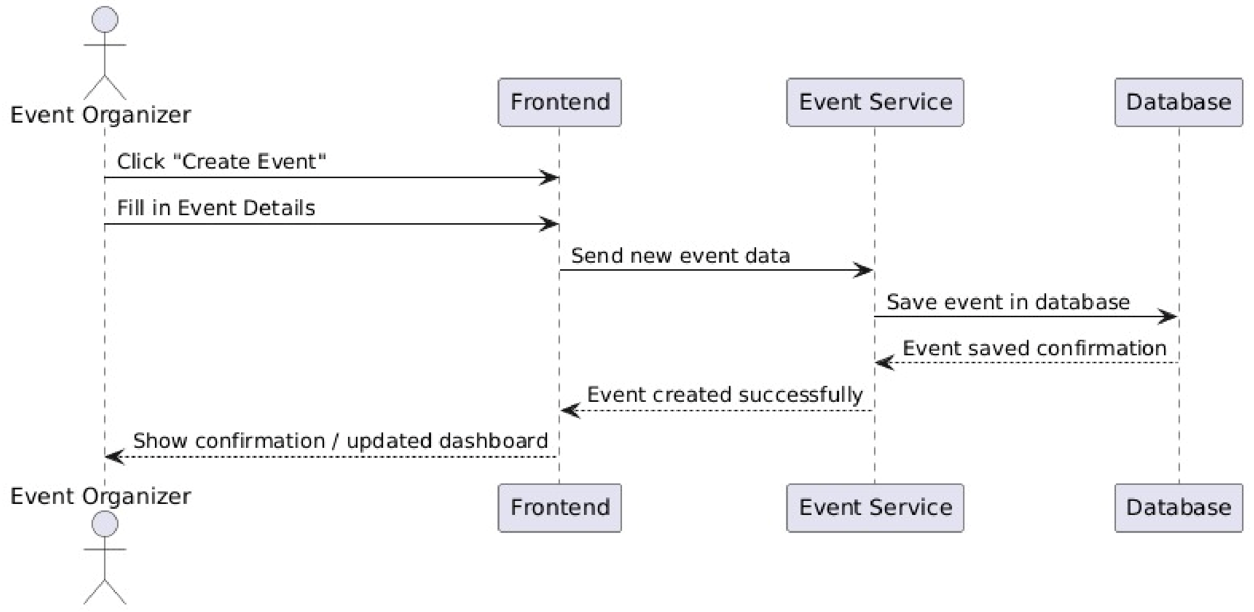
3.1 Use Case Diagram

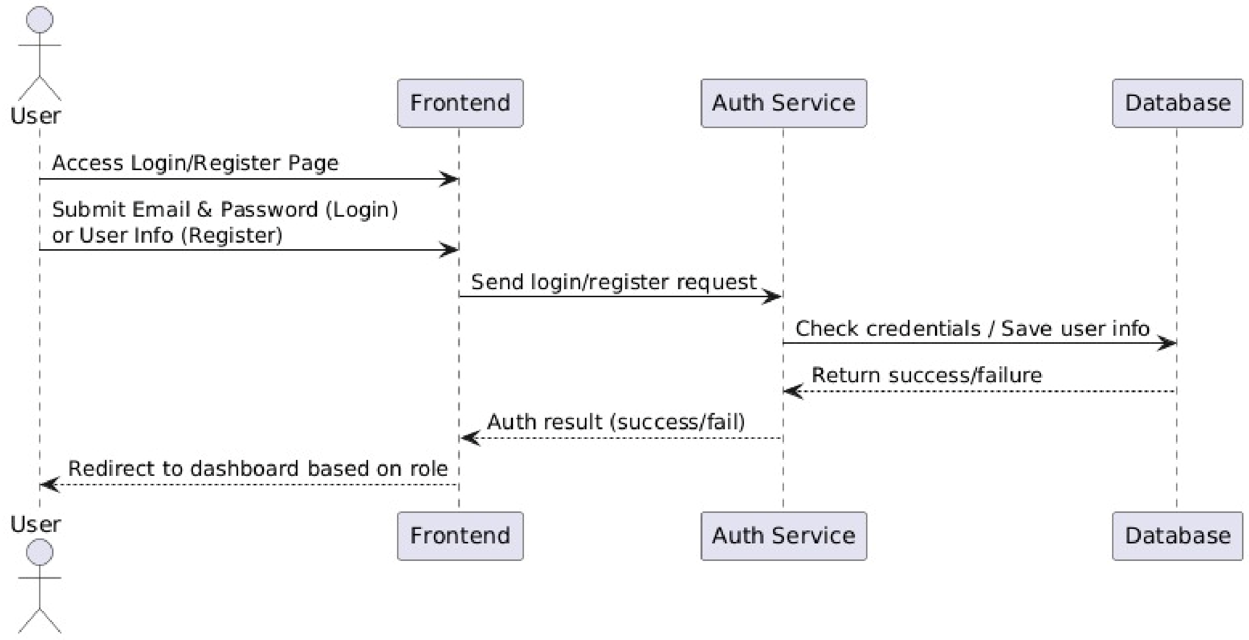


3.2 Activity Diagram

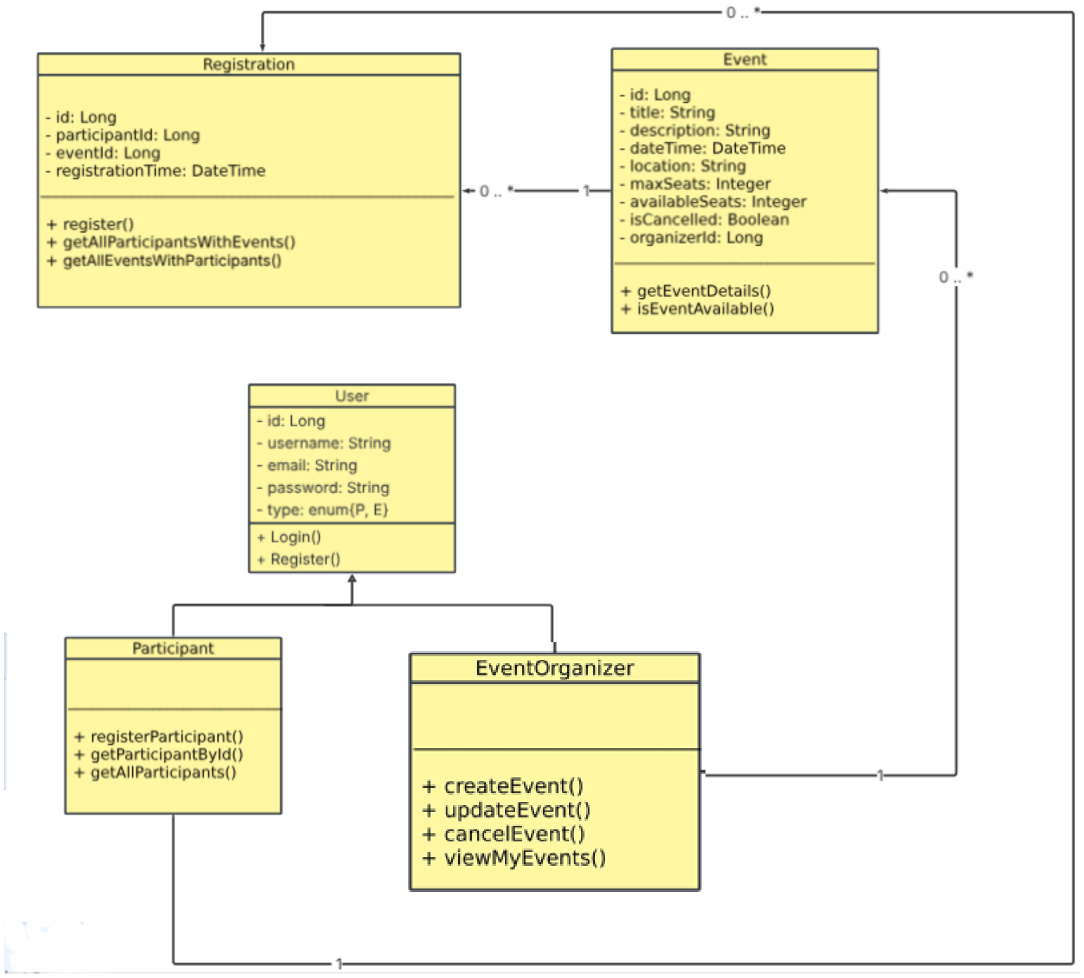
3.3 Sequence Diagram


Participant
Frontend
Event Service
Participant Service
Database
Click "Register" on Event
Check seat availability
Query event seats
Seats info
Seats available?
Check if already registered
Query participant registrations
Response
Already registered?
alt
[Valid registration]
Register participant
Save participant info
Saved
Success
Decrease available seats
Update event seat count
Updated
Show success or error message
Participant
Frontend
Event Service
Participant Service
Database

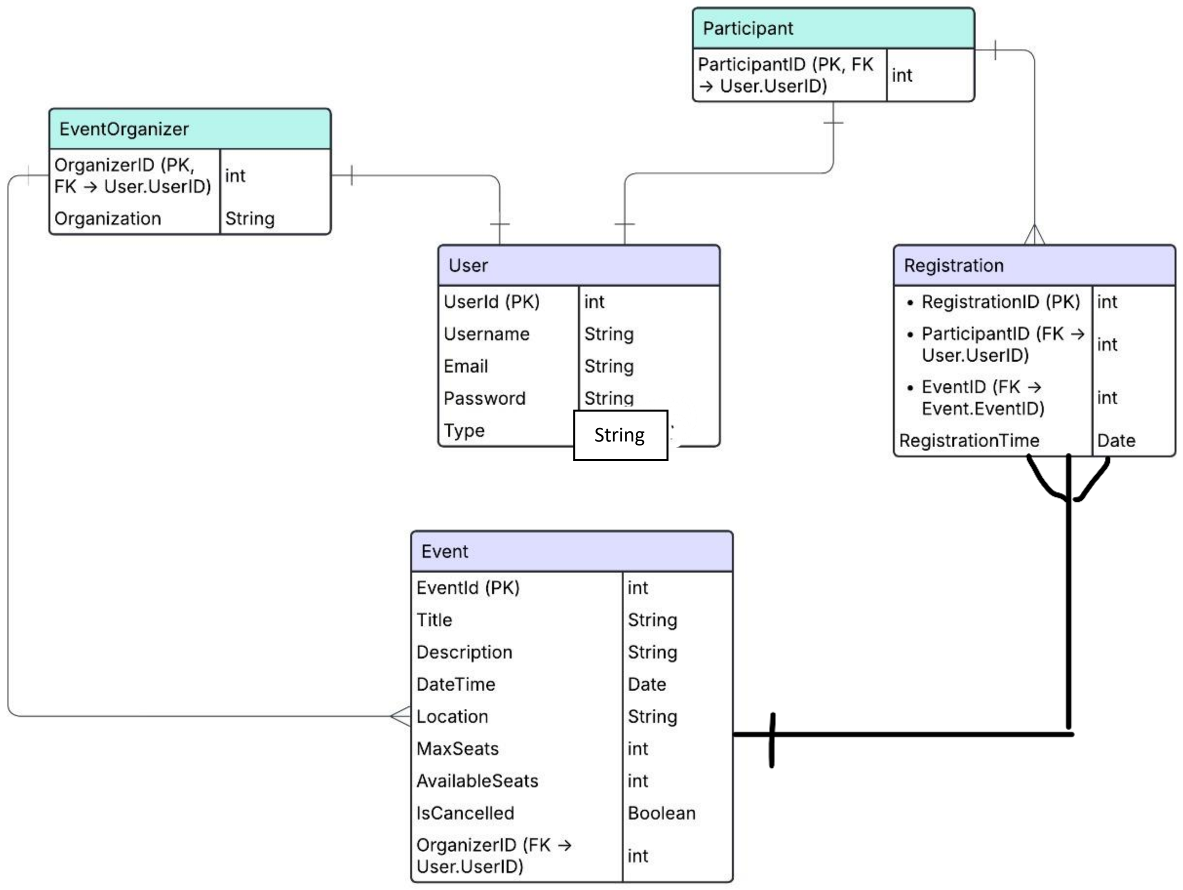

Event Organizer
Frontend
Event Service
Database
Click "Create Event"
Fill in Event Details
Send new event data
Save event in database
Event saved confirmation
Event created successfully
Show confirmation / updated dashboard
Event Organizer
Frontend
Event Service
Database


User
Frontend
Auth Service
Database
Access Login/Register Page
Submit Email & Password (Login)
or User Info (Register)
Send login/register request
Check credentials / Save user info
Return success/failure
Auth result (success/fail)
Redirect to dashboard based on role
User
Frontend
Auth Service
Database

3.4 Class Diagram



3.5 Entity-Relationship Diagram (ERD)


Participant
ParticipantID (PK, FK → User.UserID)
int
EventOrganizer
OrganizerID (PK, FK → User.UserID)
int
Organization
String
User
UserId (PK)
int
Username
String
Email
String
Password
String
Type
String
Registration
RegistrationID (PK)
int
ParticipantID (FK → User.UserID)
int
EventID (FK → Event.EventID)
int
RegistrationTime
Date
Event
EventId (PK)
int
Title
String
Description
String
DateTime
Date
Location
String
MaxSeats
int
AvailableSeats
int
IsCancelled
Boolean
OrganizerID (FK → User.UserID)
int

4. System Architecture

4.1 Architectural Pattern

The system adopts a **microservices architecture** to ensure modularity, scalability, and independent deployment of services. Each core feature is implemented as a separate microservice, communicating via RESTful APIs.

4.2 Microservices

The system is divided into the following microservices:

1. **Event Service**: Manages event creation, cancellation, and rescheduling.
2. **Registration Service**: Handles participant registration and seat availability checks.
3. **User Service**: Manages user authentication and profiles (organizers and participants).

4.3 Technology Stack

- **Frontend**: Native JS with Tailwind CSS for responsive UI.
- **Backend**: Java Spring boot for microservices.
- **Database**: MySQL and MySQL Workbench for relational data storage.
- **API**: RESTful APIs with JSON payloads.

5. API Documentation

The system exposes RESTful APIs for interaction between microservices and the frontend. Below is a sample API documentation for key endpoints.

Event Service APIs

- **GET http://localhost:8080/events/api/events**
  Retrieves a list of all events.
- **GET http://localhost:8080/events/api/events/{id}**
  Retrieves a specific event by its ID.
- **POST http://localhost:8080/events/api/events**
  Creates a new event.

- **PUT http://localhost:8080/events/api/events/{id}**
  Updates an existing event.

- **DELETE http://localhost:8080/events/api/events/{id}**
  Cancels an event by its ID.

- **GET http://localhost:8080/events/api/events/organizer/{organizerId}**
  Retrieves all events organized by a specific organizer.

- **GET http://localhost:8080/events/api/events/available**
  Retrieves all events with available seats.

Registration Service APIs

- **POST http://localhost:8080/api/registrations/register**
  Registers a user for an event.

- **GET http://localhost:8080/api/registrations/events**
  Retrieves all events with their registered participants.

- **GET http://localhost:8080/api/registrations/participants**
  Retrieves all participants with their registered events.

Authentication Service APIs

- **POST http://localhost:8080/api/auth/register**
  Registers a new participant user.

- **POST http://localhost:8080/api/auth/register-EO**
  Registers a new event organizer user.

- **POST http://localhost:8080/api/auth/login**
  Authenticates a user and returns their details.

- **GET http://localhost:8080/api/auth/users/email/{email}**
  Retrieves a user by their email.

- **GET http://localhost:8080/api/auth/users/{id}**
  Retrieves a user by their ID.

- **GET http://localhost:8080/api/auth/users/email**
  Retrieves all users.

6. Database Scheme

The system uses MySQL as the relational database to store and manage data. JPA (Java Persistence API) is used for object-relational mapping, allowing seamless interaction between Java objects and database tables. MySQL Workbench was used to design the database schema and visualize relationships during development.

7. Object Constraint Language (OCL)

OCL is used to define constraints on the system's objects to ensure data integrity and business rules. Below are sample OCL constraints.

Prevents duplicate registrations for the same event by the same participant.

ملحوظة بس
P -> Participant | E -> EventOrganizer
مفيش أنواع لليوزرس بحرف بس ده للتبسيط بس

Defining user validation rules

- **context User**
- An event must be of type "E" and that user type "P". type = "E" or type = "P".

User must have a non-empty username

```
context User
inv UniqueUsername:
    self.username.size() > 0
```

User must have a non-empty email

```
context User
inv UniqueEmail:
    self.email.size() > 0
```

User must have a password set

```
context User
inv PasswordSet:
    self.password.size() > 0
```

User type must be either Participant or EventOrganizer

```
context User
inv ValidUserType:
    Set{'P', 'E'}->includes(self.type)
```

Registration Constraints

Ensuring registration integrity rules

- **context Registration**
- A user of type "P" must be a participant to register for an event.

Participant can register only once per event

```
context Registration
inv NoDuplicateRegistrations:
    self.participant.registrations->forAll(r1 |
        self.participant.registrations->select(r2 | r2.event = r1.event)-
>size() = 1
    )
```

Registration must be linked to a valid event

```
context Registration
inv ValidEventLink:
    self.event.title.size() > 0
```

Registration date must not be in the future

```
context Registration
inv ValidRegistrationTime:
    self.registrationTime <= LocalDateTime.now()
```

Available seats must not exceed max seats

```
context Registration
inv SeatLimit:
    self.event.availableSeats <= self.event.maxSeats
```

Event Constraints

Ensuring event-related rules

- **context Event**
- An event must be organized by a user of type "E".

Event must have a non-empty title

```
context Event
inv NonEmptyTitle:
    self.title.size() > 0
```

Event must have a non-empty description

```
context Event
inv NonEmptyDescription:
    self.description.size() > 0
```

Event must have a date in the future

```
context Event
inv FutureDate:
    self.dateTime > LocalDateTime.now()
```

Event must have at least one available seat when active

```
context Event
inv AvailableSeatsWhenActive:
    self.isEventAvailable() = true implies self.availableSeats > 0
```

Event must have a valid organizer

```
context Event
inv ValidOrganizer:
    self.organizer.type = 'E'
```

Participant Constraints

Ensuring participant-related rules

- **context Participant**
- A user of type "P" must have at least one registration to be considered a participant.

Participant must have at least one registration

```
context Participant
inv HasRegistration:
    self.registrations->notEmpty()
```

Participant can view only their registrations

```
context Participant
inv OwnRegistrations:
    self.registrations->forAll(r | r.participant = self)
```

Participant cannot register for cancelled events

```
context Participant
inv NoCancelledEvents:
    self.registrations->forAll(r | r.event.isCancelled = false)
```

EventOrganizer Constraints

Ensuring organizer-related rules

- **context EventOrganizer**
- An organizer must have at least one event created.

EventOrganizer must have at least one event

```
context EventOrganizer
inv HasEvents:
    self.events->notEmpty()
```

All events by EventOrganizer must have non-empty titles

```
context EventOrganizer
inv EventsWithTitles:
    self.events->forAll(e | e.title.size() > 0)
```

EventOrganizer can cancel only their own events

```
context EventOrganizer
inv CancelOwnEvents:
    self.events->forAll(e | e.cancelEvent() implies e.organizer = self)
```

All events by EventOrganizer must have future dates

```
context EventOrganizer
inv FutureEventDates:
    self.events->forAll(e | e.dateTime > LocalDateTime.now())
```

General Constraints

Ensuring system-wide administrative rules

- **context Event**
- An event must be created by an organizer with a valid organization.

Active events must not exceed 100 registrations

```
context Event
inv RegistrationLimit:
    self.isEventAvailable() = true implies self.registrations->size() <= 100
```

Event must be created by a valid organizer

```
context Event
inv OrganizerValidation:
    self.organizer.organization.size() > 0
```

No duplicate events with the same title and date

```
context Event
inv UniqueEvent:
    self.organizer.events->forAll(e1 |
        self.organizer.events->select(e2 | e2.title = e1.title and
e2.dateTime = e1.dateTime)->size() = 1
    )
```

8. Aspect-Oriented Programming (AOP)

AOP is used to handle cross-cutting concerns such as logging, authentication, and error handling. Below are examples of aspects applied to the system.

- ➔ **TO BE DONE LATER**

9. Microservices Design

9.1 Service Boundaries

Each microservice is responsible for a specific domain:

- **Event Service**: CRUD operations for events.
- **Registration Service**: Manages event registrations and seat limits.
- **User Service**: Handles user authentication and profiles.

9.2 Communication

- **Synchronous**: REST APIs over HTTP for inter-service communication.
- **API Gateway**: A single entry point for all client requests, routing them to appropriate microservices.
- **Eureka Server**: an application that contains information about all micro services including the name of the service, port, and IP address.

9.3 Scalability

- **Load Balancing**: Use an API gateway to distribute traffic.