Deliverable#3:

Ibtisam Shahzad (22i-1201)
Haider Zia (22i-1196)
Ibrahim Asim (22i-1330)
Company : Smartsphere

A- Software Project Plan

Work Breakdown Structure (WBS)

1. Project Initiation

- 1.1 Requirement Analysis
- 1.2 Feasibility Study
- 1.3 Project Planning (Trello, GitHub Setup)

2. System Design

- 2.1 High-Level Architecture Design
- 2.2 UML Diagrams
    - 2.2.1 Use Case Diagram
    - 2.2.2 Class Diagram
    - 2.2.3 Sequence Diagram
    - 2.2.4 Package Diagram
- 2.3 Database Schema Design

3. Frontend Development

- 3.1 Authentication Pages
    - 3.1.1 Login
    - 3.1.2 Signup
- 3.2 User Dashboards
    - 3.2.1 Admin Dashboard
    - 3.2.2 Organizer Dashboard
    - 3.2.3 Participant Dashboard

- 3.3 Event Management Pages
    - 3.3.1 Create/Edit/Delete Events
    - 3.3.2 Announcements

- 3.4 Feedback & Notifications UI

4. Backend Development

- 4.1 API Development
    - 4.1.1 User Controller
    - 4.1.2 Event Controller
    - 4.1.3 Ticket & Feedback Controller

- 4.2 Services Implementation
- 4.3 Security Configuration (BCrypt, JWT)
- 4.4 Database Integration

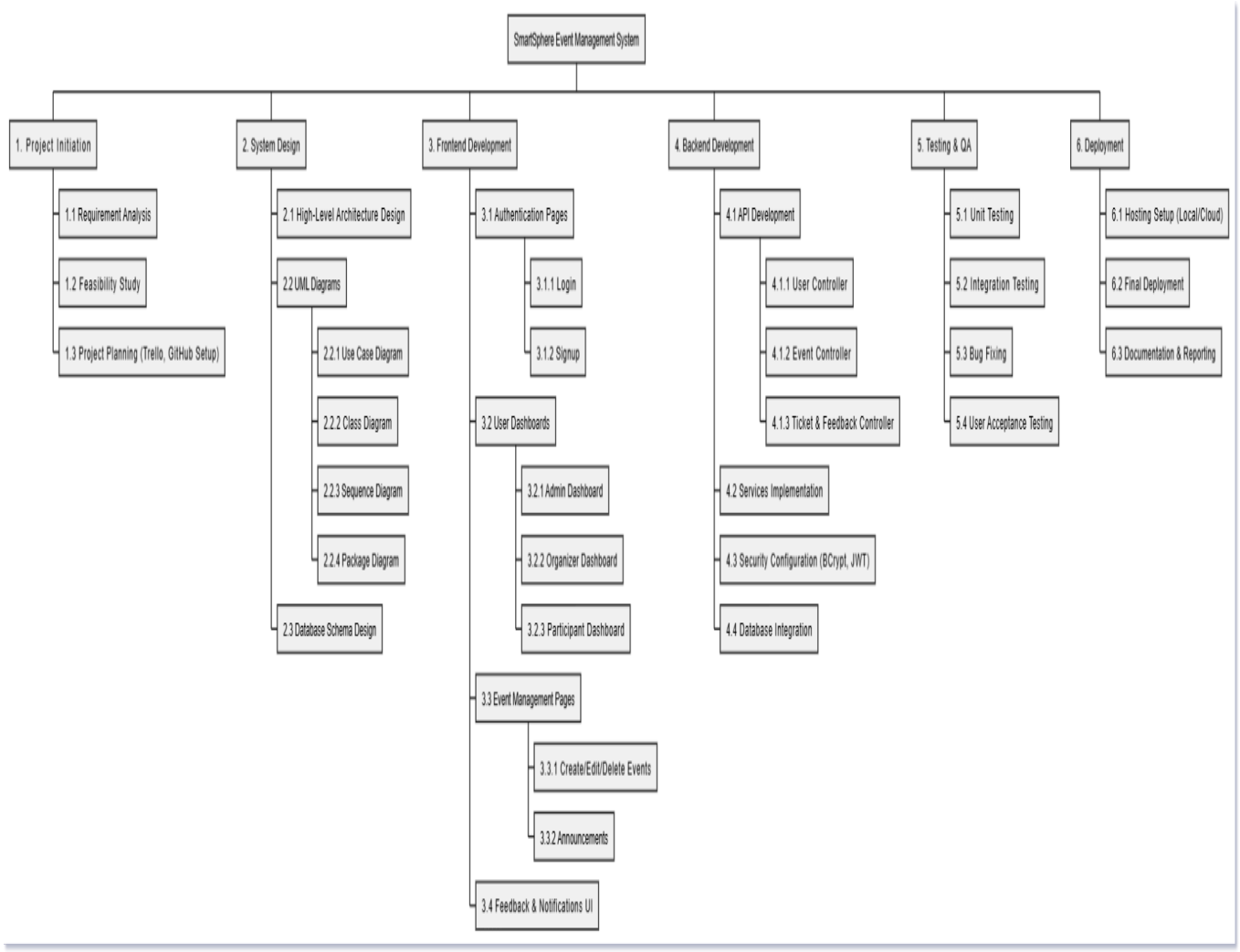
5. Testing & QA

- 5.1 Unit Testing
- 5.2 Integration Testing
- 5.3 Bug Fixing
- 5.4 User Acceptance Testing

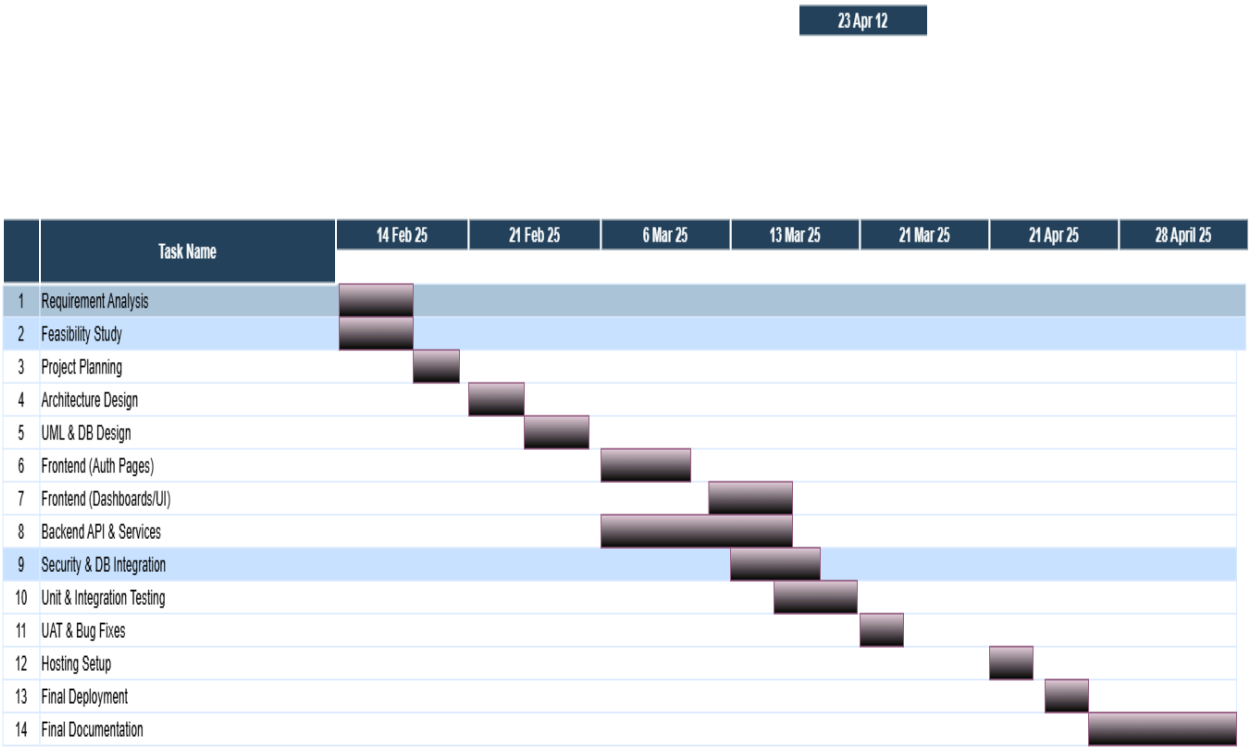
6. Deployment

- 6.1 Hosting Setup (Local/Cloud)
- 6.2 Final Deployment
- 6.3 Documentation & Reporting

WBS Chart:


SmartSphere Event Management System
1. Project Initiation
1.1 Requirement Analysis
1.2 Feasibility Study
1.3 Project Planning (Trello, GitHub Setup)
2. System Design
2.1 High-Level Architecture Design
2.2 UML Diagrams
2.2.1 Use Case Diagram
2.2.2 Class Diagram
2.2.3 Sequence Diagram
2.2.4 Package Diagram
2.3 Database Schema Design
3. Frontend Development
3.1 Authentication Pages
3.1.1 Login
3.1.2 Signup
3.2 User Dashboards
3.2.1 Admin Dashboard
3.2.2 Organizer Dashboard
3.2.3 Participant Dashboard
3.3 Event Management Pages
3.3.1 Create/Edit/Delete Events
3.3.2 Announcements
3.4 Feedback & Notifications UI
4. Backend Development
4.1 API Development
4.1.1 User Controller
4.1.2 Event Controller
4.1.3 Ticket & Feedback Controller
4.2 Services Implementation
4.3 Security Configuration (BCrypt, JWT)
4.4 Database Integration
5. Testing & QA
5.1 Unit Testing
5.2 Integration Testing
5.3 Bug Fixing
5.4 User Acceptance Testing
6. Deployment
6.1 Hosting Setup (Local/Cloud)
6.2 Final Deployment
6.3 Documentation & Reporting

Gantt Chart:



B-System Architecture:

1-Identifying Subsystems:

1. Authentication Subsystem
    a. Handles login, signup, password encryption (BCrypt).
    b. Related Classes: `AuthController`, `SecurityConfig`, `UserService`.

2. User Management Subsystem

    a. Manages user profiles, roles (Admin, Organizer, Participant).
    b. Related Classes: `UserController`, `UserService`, `UserRepository`, `User`.

3. Event Management Subsystem
    a. Creation, modification, deletion of events by organizers.

b. Related Classes: `EventController`, `EventService`, `EventRepository`, `Event`.

4. Notification & Feedback Subsystem

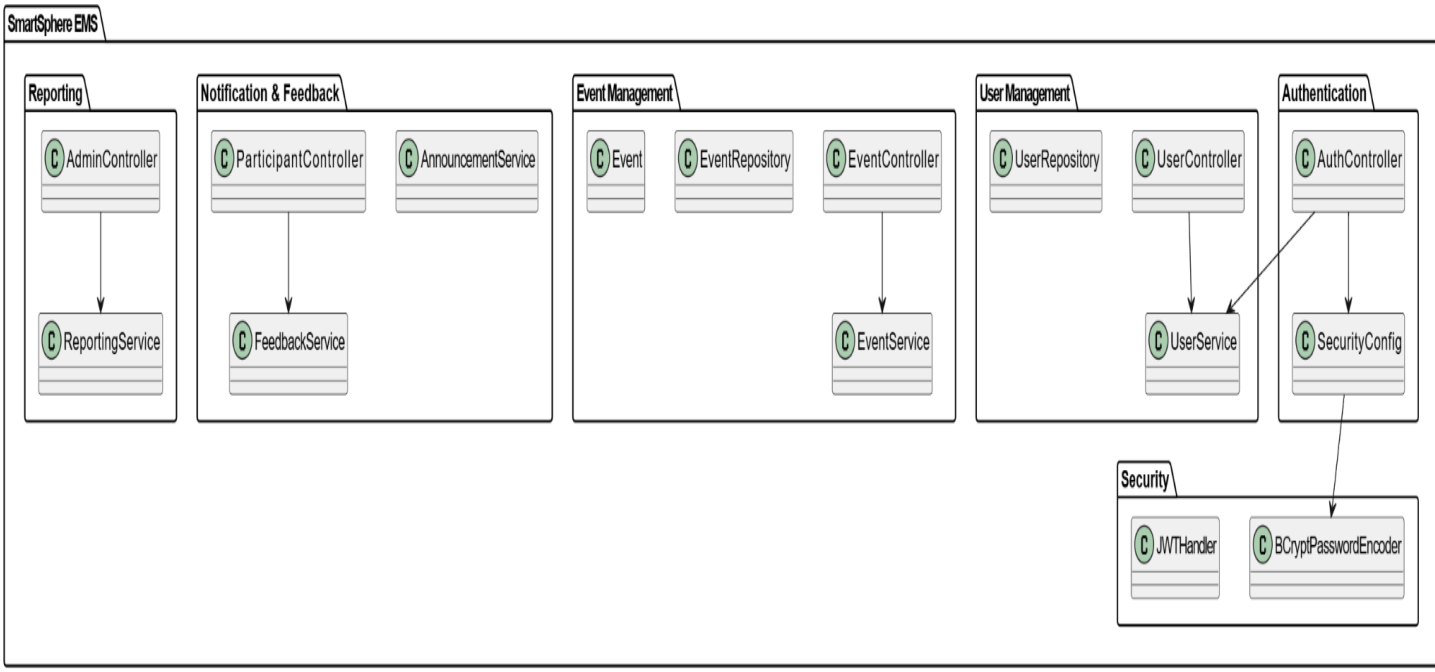
a. Sends announcements, collects feedback from participants.
b. Related Classes: `Announcement`, `Feedback`, `ParticipantController`.

5. Reporting Subsystem

a. Admin views reports, participant lists, feedback.
b. Related Classes: `AdminController`, `Reporting`.

6. Security Subsystem

a. Authentication, authorization, session management.
b. Related Classes: `SecurityConfig`, `JWT` (if used), `Spring Security`.



2-Architecture Styles:

1. Layered Architecture (N-Tier Architecture)

Description:
The system is organized into **separate layers**, each with distinct responsibilities.

Layers in SmartSphere:

- **Presentation Layer**: React.js Frontend UI.
- **Business Logic Layer**: Spring Boot Services handling core operations.
- **Data Access Layer**: Repositories interacting with the database.
- **Database Layer**: MySQL for persistent storage.

Why Used?

- Clear separation of concerns.
- Easier to maintain and scale.
- Independent development of frontend and backend.

2. Client-Server Architecture

Description:
The system follows a **Client-Server model**, where:

- The **client (React.js)** requests resources.
- The **server (Spring Boot)** processes and responds.

Communication:

- Uses **HTTP** protocols.
- **RESTful APIs** serve as the interface between client and server.

Why Used?

- **Scalability**: Clients and servers can scale independently.
- **Modularity**: Frontend and backend can evolve separately.

3. Model-View-Controller (MVC)

Description:
The Spring Boot backend follows the **MVC pattern**:

- **Model**: Data layer (Entities like User, Event, Ticket).
- **View**: Not directly applicable (handled by React), but can include API responses.
- **Controller**: Handles HTTP requests (e.g., `UserController`, `EventController`).

Why Used?

- Makes the backend more organized.
- Separates data, logic, and request handling.

4. RESTful Architecture

Description:
Backend services expose **RESTful APIs**:

- CRUD operations via **HTTP verbs** (GET, POST, PUT, DELETE).

- Stateless interactions.

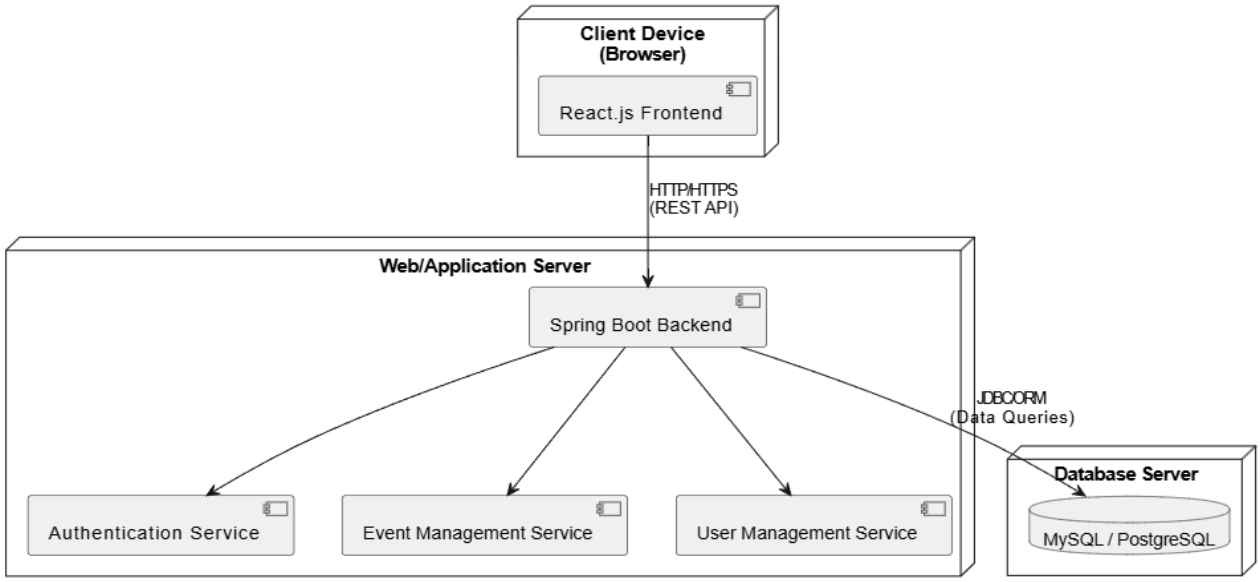
Why Used?

- **Interoperability**: Easily integrates with any frontend.
- **Scalability**: Lightweight, ideal for distributed systems.

3-Deployment diagram for client deployments:

Key Nodes in SmartSphere Deployment:

1. **Client Node** (User's Browser)
    - Runs the **React.js Frontend**.
    - Interacts with the backend via **RESTful APIs**.
2. **Web Server Node**
    - Hosts **static React build** (Optional if hosted separately).
    -
3. **Application Server Node**
    - Runs **Spring Boot Backend** (APIs).
    - Handles business logic, authentication, event management, etc.
4. **Database Server Node**
    - Stores data in **MySQL**
    - Accessed by the Application Server.



Deployment Flow:

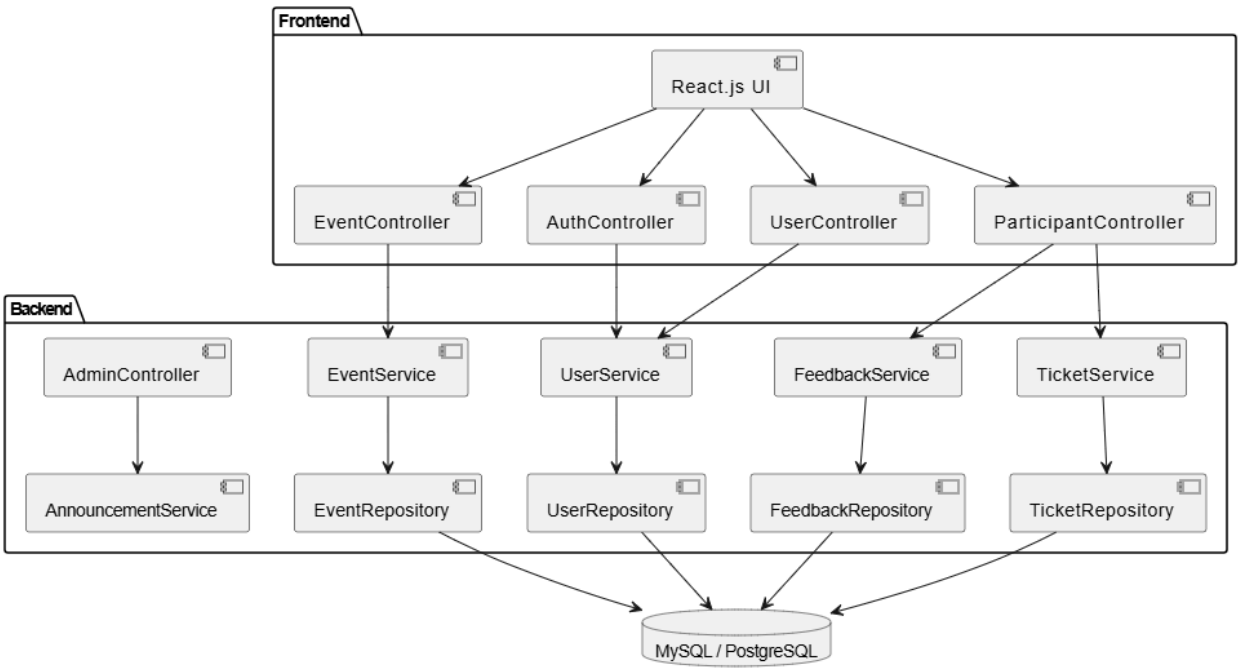
1. User accesses **frontend UI** from the browser.
2. UI sends **HTTP requests** to **Spring Boot REST API**.
3. Backend processes requests and queries **database**.
4. Responses are sent back to the frontend.

4-Component Diagram:

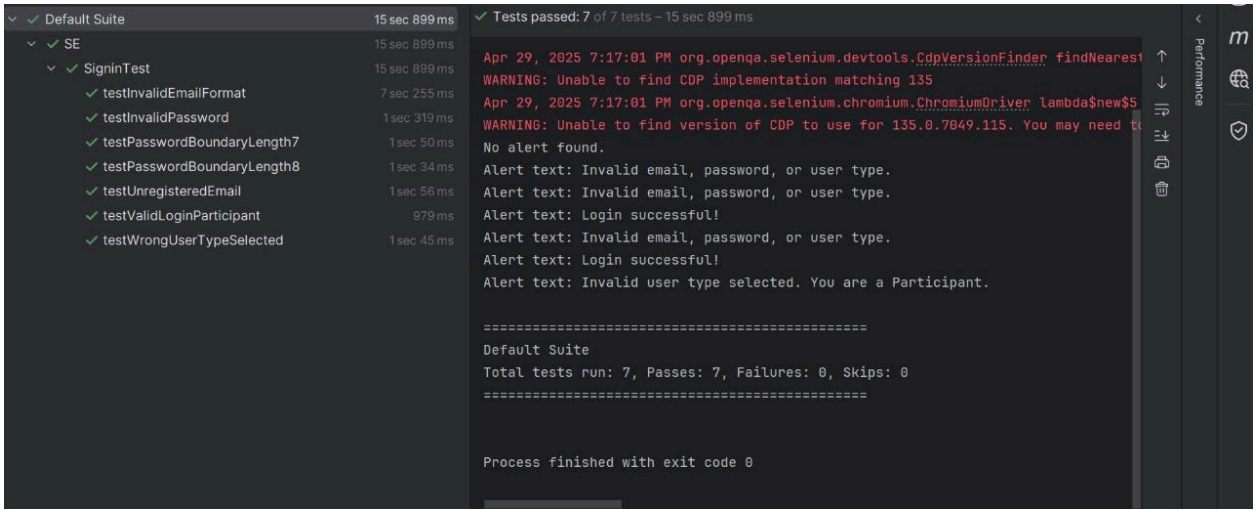
Key Components:

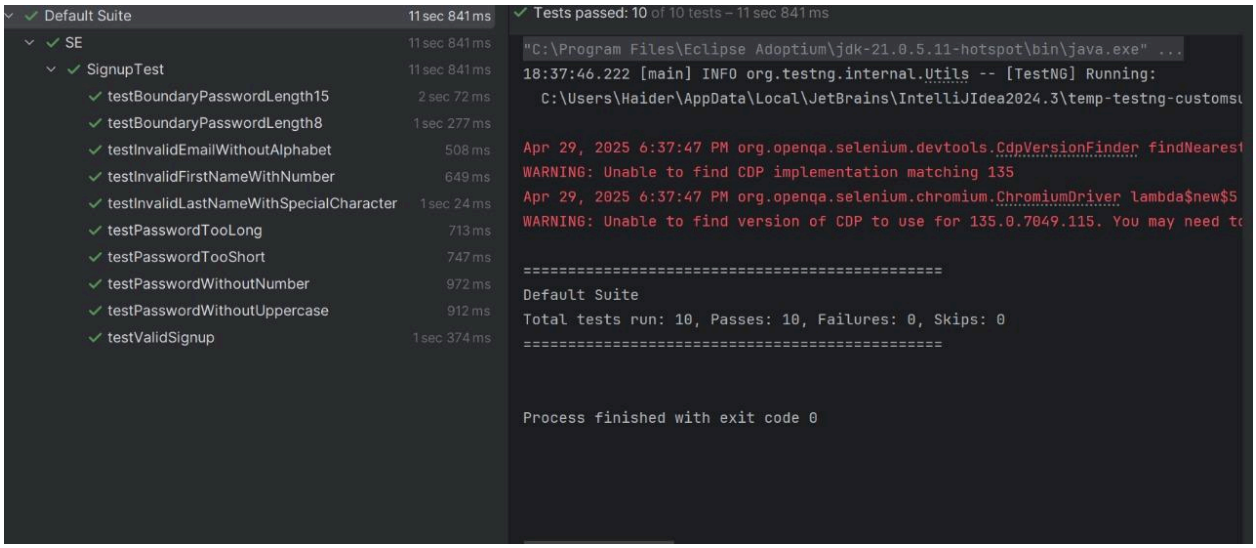
1. **Frontend (React.js)**
    - Interfaces with backend via REST APIs.
2. **Spring Boot Backend Components**:
    - **Controllers**:
        - `AuthController`
        - `UserController`

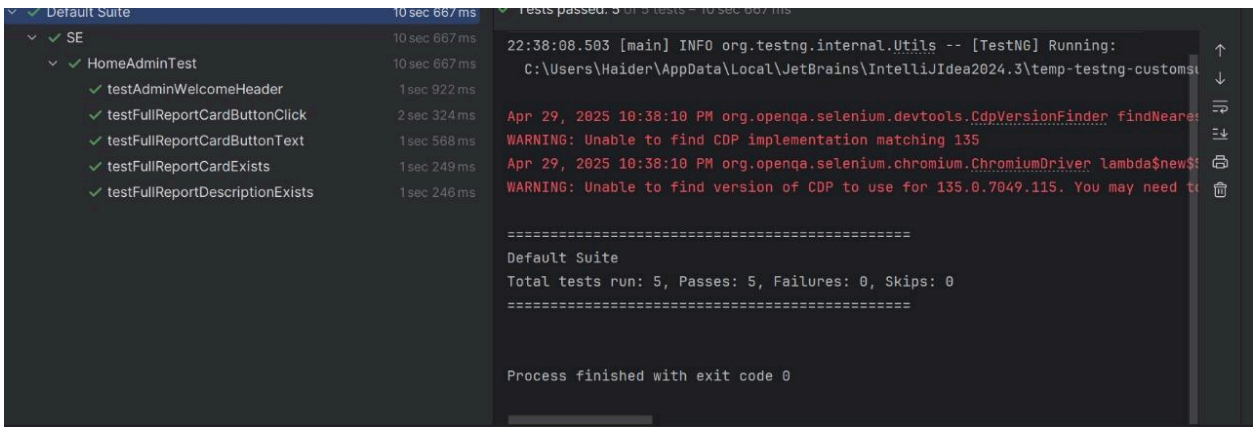
        - `EventController`
        - `ParticipantController`
        - `AdminController`
    - **Services**:
        - `UserService`
        - `EventService`
        - `TicketService`
        - `FeedbackService`
        - `AnnouncementService`
    - **Repositories**:
        - `UserRepository`
        - `EventRepository`
        - `TicketRepository`
        - `FeedbackRepository`
3. **Database**:
    - MySQL



C-TestCases BlackBox:







D-TestCases WhiteBox:

What Is Covered Well:

Controller Layer (controller):
Achieved 100% coverage. All REST endpoints were tested through unit tests, ensuring every path is executed.

Model Layer (models/):
Achieved 100% coverage. Data classes like Event, User, etc., were fully instantiated and used within service/controller tests.

Repository Layer (repository/):
Interface-based with Spring Data JPA – considered covered when interacting through services.

Security Layer (security/):
100% coverage on custom classes (e.g., filters, configs).

Main Application Class (DemoApplication):
Fully covered during application context loading.

Core Logic (EventService & AnnouncementService):
Main methods like createEvent, getAllEvents, and searchEvents were invoked and validated.

What Is Not Covered and Why:

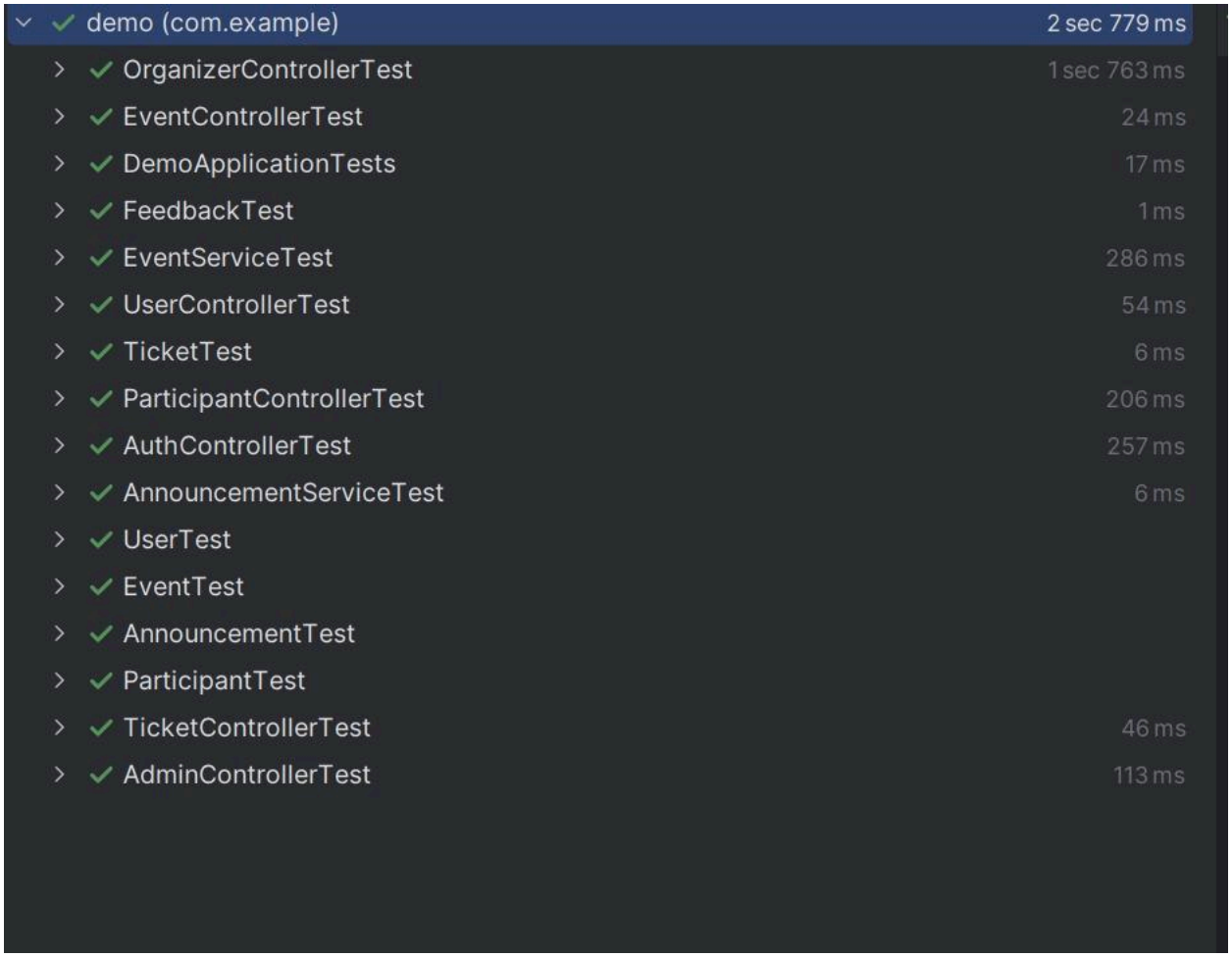
Some Service Methods:
Coverage in service/ layer is around 33%–40% in some files due to incomplete testing of:

Exception branches (e.g., invalid organizer cases).
Optional-based logic where isEmpty() or isPresent() was not triggered for all paths.
update and delete operations were partially tested or skipped.

Branch Coverage Gaps:
if/else and exception-throwing branches need more dedicated negative test cases to boost coverage.

Third-party and Auto-generated Code:
External frameworks (Spring Boot, Hibernate, JWT, etc.) are not included in coverage metrics. These are assumed to work as intended and don't need direct testing.

Unreachable or Rare Execution Paths:
Paths that rely on specific user states, request combinations, or database errors weren't feasible to replicate in unit tests alone.



models	100% (6/6)	100% (77/77)	100% (99/99)	100% (0/0)
Announcement	100% (1/1)	100% (9/9)	100% (12/12)	100% (0/0)
Event	100% (1/1)	100% (18/18)	100% (23/23)	100% (0/0)
Feedback	100% (1/1)	100% (10/10)	100% (13/13)	100% (0/0)
Participant	100% (1/1)	100% (8/8)	100% (10/10)	100% (0/0)
Ticket	100% (1/1)	100% (12/12)	100% (16/16)	100% (0/0)
User	100% (1/1)	100% (20/20)	100% (25/25)	100% (0/0)
controller	100% (10/10)	98% (56/57)	93% (152/162)	67% (39/58)
AdminController	100% (2/2)	100% (18/18)	90% (38/42)	41% (5/12)
AuthController	100% (1/1)	100% (2/2)	92% (12/13)	87% (7/8)
EventController	100% (1/1)	100% (7/7)	100% (14/14)	100% (0/0)
OrganizerController	100% (2/2)	88% (8/9)	82% (24/29)	42% (6/14)
ParticipantController	100% (2/2)	100% (14/14)	100% (44/44)	87% (14/16)
TicketController	100% (1/1)	100% (2/2)	100% (7/7)	75% (3/4)
UserController	100% (1/1)	100% (5/5)	100% (13/13)	100% (4/4)

repository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
AnnouncementRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
EventRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
FeedbackRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ParticipantRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
TicketRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
UserRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)

com.example.demo	80% (21/26)	88% (151/170)	86% (295/343)	61% (49/80)
controller	100% (10/10)	98% (56/57)	93% (152/162)	67% (39/58)
models	100% (6/6)	100% (77/77)	100% (99/99)	100% (0/0)
repository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
security	100% (1/1)	100% (4/4)	100% (8/8)	100% (0/0)
service	33% (2/6)	40% (11/27)	45% (30/66)	45% (10/22)
DemoApplication	100% (2/2)	75% (3/4)	85% (6/7)	100% (0/0)
HelloController	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)