DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

EXPERIMENT-10

Student Name: Yash Goel
UID: 23BCS11498
Branch: CSE
Section/Group: KRG-2B
Semester: 5th
Subject Code: 23CSH-304
Subject Name: Project Based Java Learning

1. Aim:

To develop a scalable, secure, and real-time attendance tracking web application that allows administrators to generate time-sensitive QR codes for specific class sessions and enables authenticated students to mark their presence instantly by scanning the code

2. Objective:

- Implement a robust, atomic database strategy in MongoDB to prevent data corruption and concurrency issues during simultaneous student check-ins.
- Establish a clean, decoupled MVC architecture using Spring Boot (Controller/Service/Repository) and React (View).
- Utilize token-based authentication principles (simulated `X-User-Id` header) and robust network handling to ensure secure data transfer.
- Provide a dynamic, real-time administration dashboard for session creation, monitoring, and attendance record retrieval.

3. Technology Stack

Component	Technology	Role
Backend Framework	Java (Spring Boot)	Provides the RESTful API endpoints, handles business logic, and manages security integration.
Database	MongoDB	Stores user profiles and attendance sessions, used for high-performance atomic array updates.
Frontend Framework	React.js	Presents the user interface (Admin Dashboard, Student Profile) and manages client-side state.
Styling	Tailwind CSS	Provides rapid, utility-first styling for a responsive and modern user interface.
QR Generation	qrcodegen (JS Library)	Client-side utility for reliable, offline generation of session QR codes.
QR Scanning	html5-qrcode (JS Library)	Client-side utility for activating the student's camera to decode the attendance token.

4. Methodology

The project follows a standard layered architecture with critical focus paid to data integrity and concurrency handling:

1. Data Modeling and Persistence (MongoDB)

- **User Model:** Contains nested data, specifically the **attendanceRecords** array, where each element stores the session ID, presence status, and join time. This complex structure necessitated advanced database handling.
- **Session Model:** Stores the unique token (UUID), section, name, and, critically, the **expiresAt** timestamp to enforce time limits.

2. Atomic Update Strategy (Concurrency Solution)

The most significant technical challenge was preventing duplicate records or lost updates when multiple students check in simultaneously or when the session is initialized for many students.

- **Token Generation Fix:** The **AttendanceService.generateToken()** method uses a single, atomic MongoDB operation (mongoTemplate.updateMulti) with a negative query condition ($ne: token). This ensures that the absent record is only added to a student's array *if it does not already exist*, successfully solving the duplicate insertion problem.
- **Check-in Update:** The **AttendanceService.checkIn()** method uses the MongoDB positional operator ($) via mongoTemplate.updateFirst to update only the specific array element matching the token. This avoids loading and saving the entire large user document, guaranteeing speed and atomicity for the check-in transaction.

3. Frontend Reliability and Experience

- **Anti-Concurrency Measures:** The **fetchWithRetry** utility was modified to perform **only one attempt** for POST requests (like token generation and check-in). This prevents the browser from automatically retrying a write operation on network failure, which was identified as a source of duplicate records.
- **Real-time Monitoring:** The SessionTab.jsx component implements a client-side useEffect hook to run a **countdown timer** every second, providing the administrator with a real-time view of remaining session time.
- **Dashboard Sorting:** Sessions are sorted dynamically by status, ensuring **Active Sessions** always appear above Expired Sessions for immediate oversight.

5. Implementation

I. Backend Implementation (Spring Boot & MongoDB)

The backend is built around the **Model-Service-Controller** pattern, focusing heavily on ensuring data integrity in a highly concurrent environment (many students checking in simultaneously).

A. The Atomic Attendance Service (AttendanceService.java)

The service layer contains the complex logic that manages database transactions, session validation, and, most critically, the solution to the concurrency problem.

1. Token Generation and Roster Initialization (generateToken)

This method is executed when an administrator starts a new session (POST /api/admin/generate-token). The implementation uses MongoDB's native update functionality via **MongoTemplate** to guarantee that only one absent attendance record is created per student, even if the method is called multiple times due to a network glitch or a client error.

- **Database Operation:** mongoTemplate.updateMulti(conditionalPushQuery, pushUpdate, User.class);
- **The Conditional Query (conditionalPushQuery):** This is the anti-duplicate guard. It uses the **$ne (Not Equal)** operator on the nested array:
- **The Update (pushUpdate):** It uses the **$push** operator to add the new AttendanceRecord object (initialized to present: false) to the attendanceRecords array.
- **Result:** By combining the query and the update into one atomic operation (updateMulti), the system ensures that **no student document receives a duplicate entry** for the newly generated session token.

2. Student Check-in Execution (checkIn)

This method is executed when a student scans the QR code (POST /api/attendance/check-in).

1. **Validation Check:** The method first performs essential business validations: checking the token against the SessionRepository and verifying that the current time is **not after** the session's expiresAt timestamp.
2. **Concurrency-Safe Update:** After preliminary checks, the attendance status is updated directly in the database, avoiding the problematic Load-Modify-Save pattern:
    - **The Query:** Targets the specific user ID (_id) and the specific nested array element (attendanceRecords.sessionId).
    - **The Update:** Uses the **positional operator ($)**:
    - **Result:** mongoTemplate.updateFirst(...) atomically updates only the two specified fields within the specific matching attendance record, ensuring high performance and integrity for concurrent check-ins.

B. Controller Layer Organization

The controller layer focuses purely on handling HTTP requests, headers, and responses, delegating all complex logic to the service.

- **Authentication Simulation:** Both controllers read the authenticated student/admin ID via the **X-User-Id header** sent by the frontend, demonstrating the separation of authentication data from the core request body.

- **Response Mapping:** The AttendanceController uses the AttendanceResponse DTO to map the business result (status code and message) directly to the final HTTP response status, ensuring the frontend receives accurate feedback (e.g., 409 Conflict if already checked in).

II. Frontend Implementation (React)

The React application focuses on security robustness, asynchronous operations, and optimal user experience for both admins and students.

A. Anti-Concurrency Network Logic (fetchWithRetry)

The fetchWithRetry utility was structurally modified to solve a common concurrency issue related to write operations.

- **Write Safety:** For non-idempotent methods (POST, PUT), the maximum number of network attempts (maxAttempts) is explicitly set to **1**. This prevents the browser from automatically resubmitting the token generation request if the initial request succeeds on the server but the client loses the response (which would otherwise create duplicate records).
- **Read Reliability:** For GET requests, the utility maintains exponential backoff, ensuring data retrieval is reliable despite intermittent network hiccups.

B. Student Check-in Logic (Profile.jsx)

The student profile is built around integrating the external camera hardware:

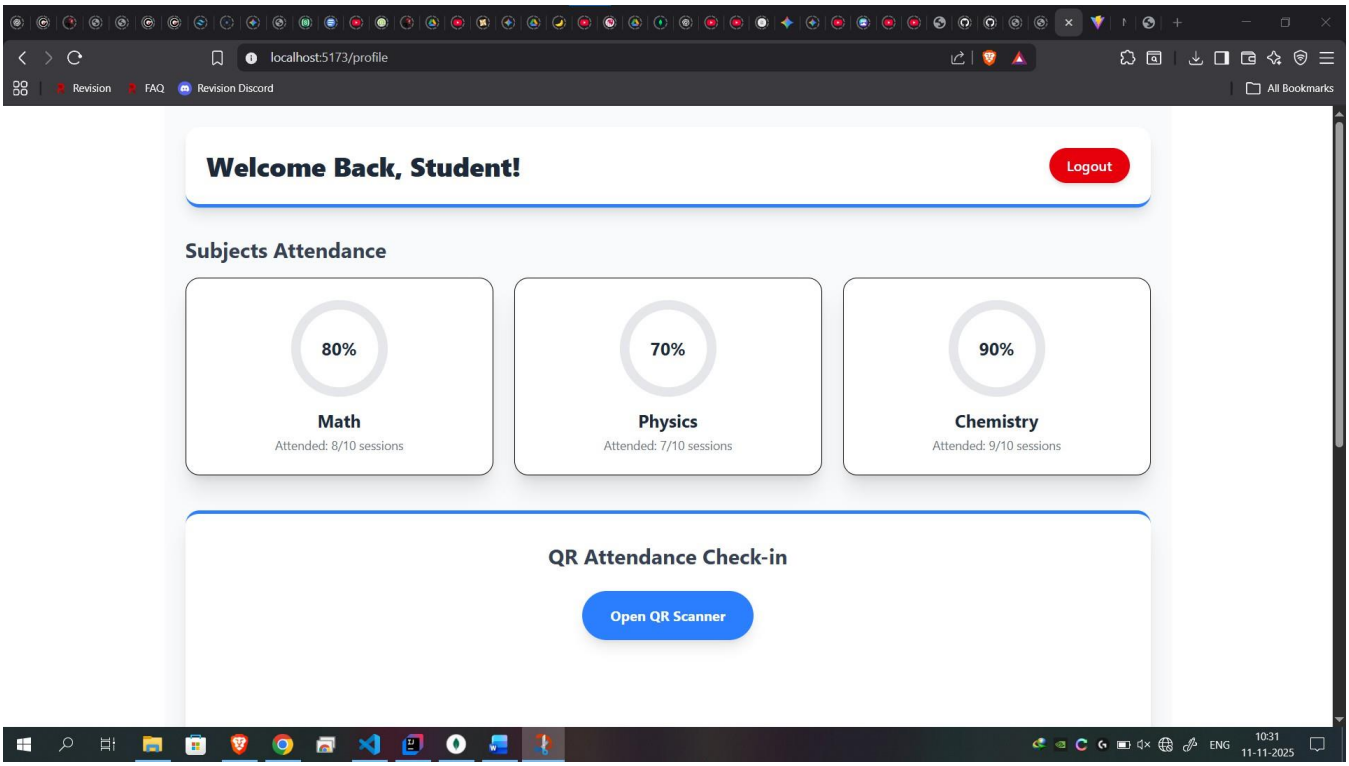
- **Scanner Lifecycle (useEffect):** Uses html5-qrcode to mount the camera scanner when cameraOpen is true and ensures the vital **.clear()** method is called to safely shut down the camera when the user closes it or navigates away.
- **Check-in Sequence:** The handleScan function immediately stops the scanner and sets isScanning = true upon decoding a QR code. It then sends the token via a POST request, attaching the user's ID via the X-User-Id header, which is essential for server-side authorization.

C. Admin Dashboard Logic (SessionTab.jsx)

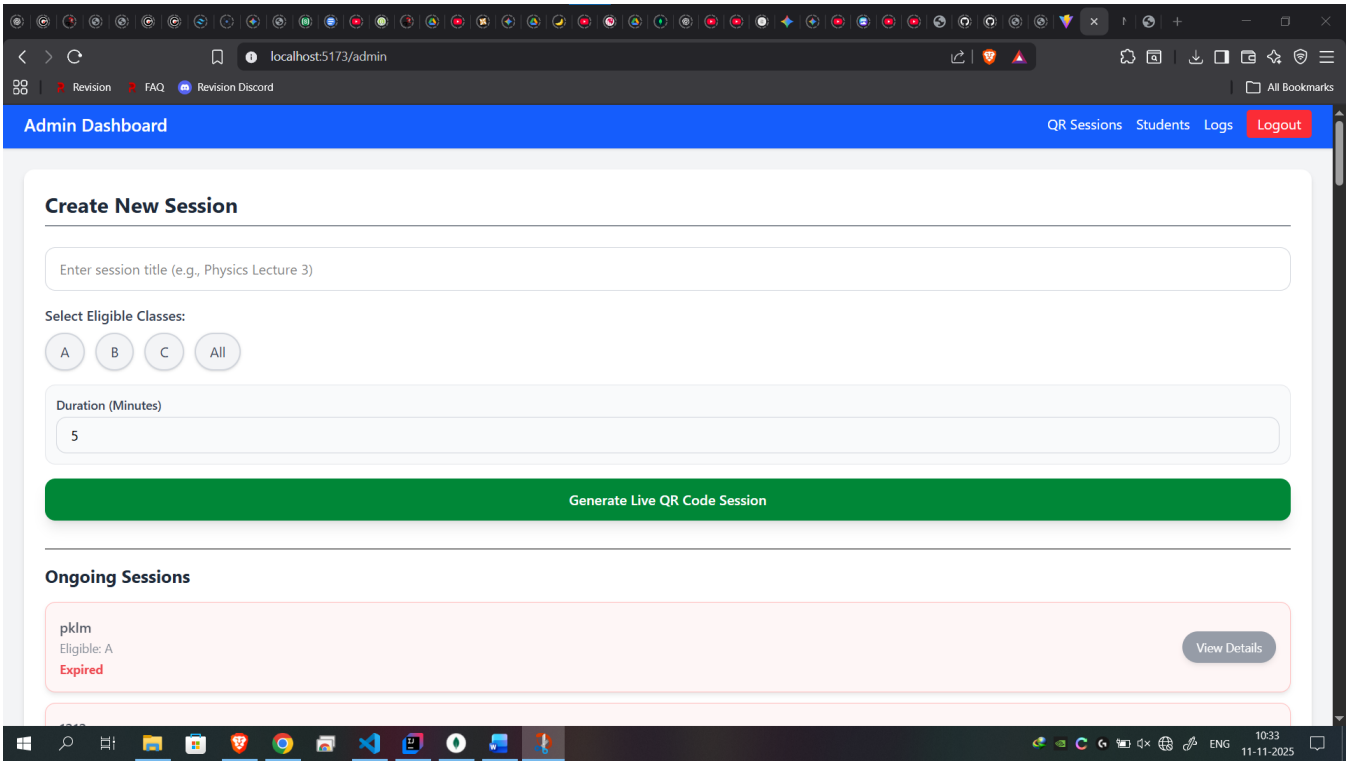
- **Real-time Clock:** A central useEffect hook runs a setInterval every second to decrement the timeLeft property of all active sessions, providing a live countdown without requiring constant server polling.
- **Dynamic Sorting:** Sessions are sorted before rendering: Active sessions are prioritized above Expired sessions, sorted internally by time left.
- **QR Generation:** The QRCodeSVG utility uses client-side JavaScript generation first (via qrcodegen) to maximize speed and minimize dependency on external network calls.

6. Output:

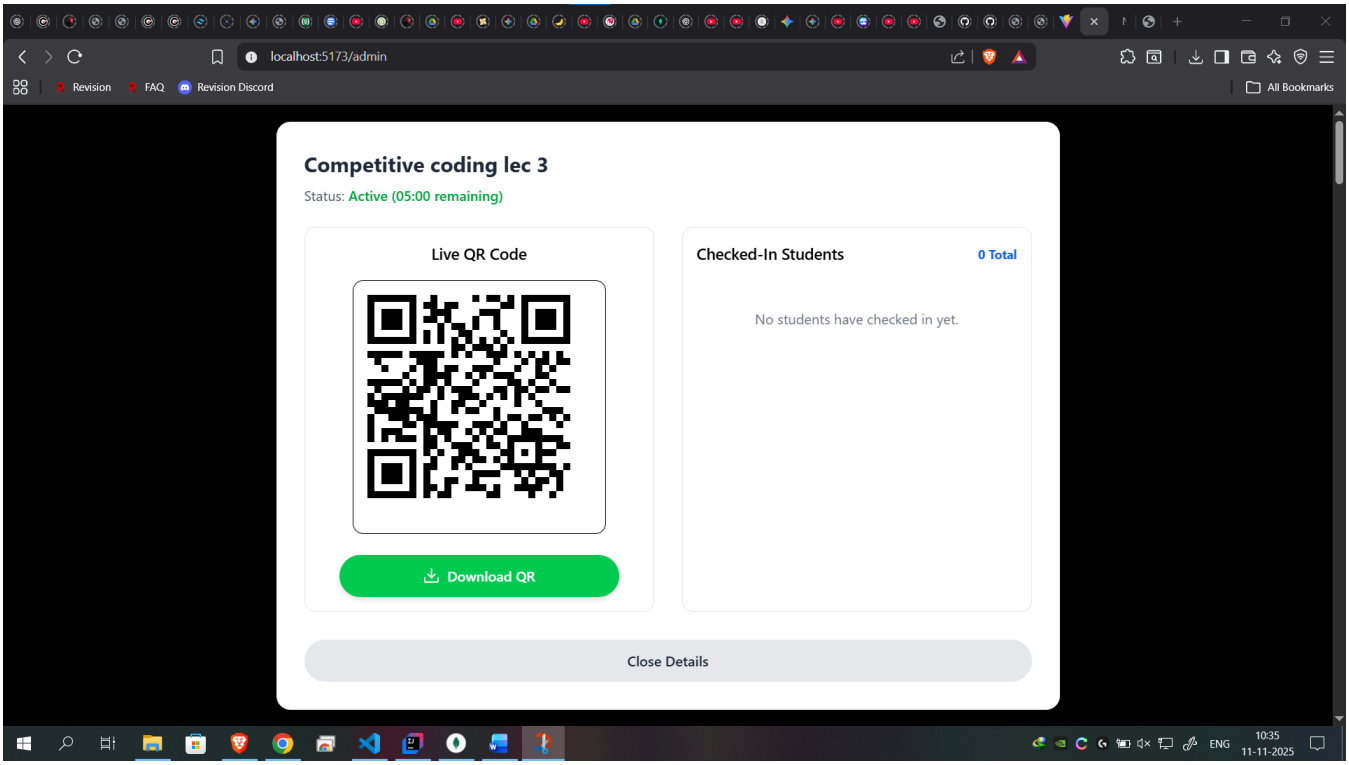
Student Interface



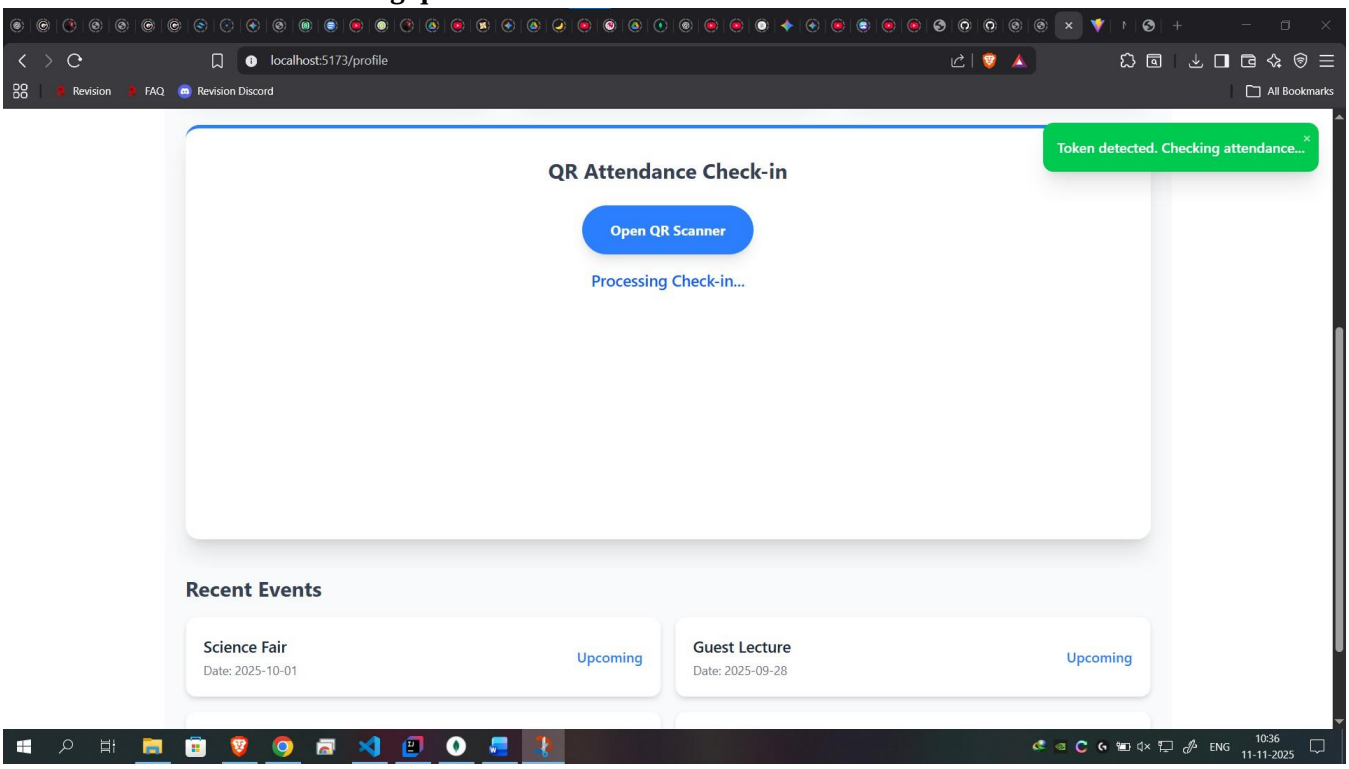
Admin Interface



Admin Creates New Session



Student Interface after scanning qr code



7. Conclusion and Future Scope

The project successfully delivered a fully functional, high-integrity attendance system. The core objective of implementing atomic updates for nested MongoDB arrays was achieved, creating a reliable foundation for scaling concurrent attendance operations.

Future Scope:

1. **Full JWT Authentication:** Replace the simulated X-User-Id header with secure, industry-standard **JSON Web Tokens (JWTs)**, integrating user validation and token management directly into the Spring Security filter chain.
2. **Role-Based Access Control (RBAC):** Implement Spring Security roles (ROLE_ADMIN, ROLE_STUDENT) to strictly enforce which users can access the AdminController versus the AttendanceController.
3. **Data Persistence for Student Profile:** Implement API endpoints to fetch a student's *actual* attendance history from the database instead of using frontend mock data, populating the Profile.jsx dashboard dynamically.