**DESIGN DOCUMENT**

**STUDENT ATTENDANCE MONITORING SYSTEM WITH QR CODE**

**Prepared By:**

**Camacho, Vien Mabee**

**Galvez, Sean Andrew**

**Java, Marjorie**

**Ortiz, Krisha Mae**

**INTRODUCTION**

Purpose of the Document This design document provides a comprehensive overview of the architecture, components, and design decisions for the Student Monitoring System. It serves as a reference for developers, stakeholders, and future maintainers to understand the system’s structure, functionality, and implementation details.

**Overview of the Software System**

The Student Monitoring System is a web-based application designed to streamline the management of student attendance and class records in educational institutions. The system leverages QR code technology to enable efficient and accurate attendance tracking. It provides interfaces for administrators and teachers to manage students, classes, and attendance records, as well as generate reports and visualize data.

**Key features include**

1. Student and class management
2. Attendance tracking via QR code scanning
3. User authentication and role-based access
4. Data visualization and reporting

**Scope of the Design Document**

**This document covers the following aspects of the Student Monitoring System**

1. System architecture and technology stack
2. Database schema and relationships
3. Frontend and backend design
4. API endpoints and data flow
5. User roles and permissions
6. Security considerations Deployment and environment setup
7. Potential areas for future enhancement

**SYSTEM ARCHITECTURE**

**Overview**

The Student Monitoring System is designed as a web-based, client-server application. It consists of a React-based frontend, a Python Flask backend, and a SQLite database. The system is structured to separate concerns between the user interface, business logic, and data storage, ensuring maintainability and scalability.

**High-Level Components and Their Interactions**

1. **Frontend (Client)**: Built with React.js, responsible for rendering the user interface, handling user interactions, and communicating with the backend via HTTP requests. Provides pages for login, dashboard, student and class management, attendance tracking, and reporting.
2. **Backend (Server):** Implemented using Python Flask. Handles API requests from the frontend, processes business logic, manages authentication, and serves HTML templates for certain routes. Interacts with the database to store and retrieve data.
3. **Database:** SQLite database stores persistent data, including user accounts, students, classes, and attendance records. Managed and migrated using Alembic. Static Assets CSS, JavaScript libraries, images, and QR codes are served as static files by the backend.

**Deployment Architecture**

**Client-Server Model**

1. The frontend and backend are deployed on the same server or separate servers, depending on the environment. In development, both run locally. In production, the backend serves the frontend’s static files or they are served by a web server (Nginx).

**Deployment Options**

**Local**

1. Both frontend and backend run on localhost for development and testing. Production: Backend (Flask) can be deployed using Gunicorn/uWSGI behind a web server (Nginx/Apache). The SQLite database resides on the server.

**Communication Protocols and Interfaces**

**Frontend-Backend Communication**

1. Uses HTTP/HTTPS for all communication.
2. RESTful API endpoints exchange data in JSON format.
3. Some routes may render HTML templates directly (server-side rendering).

**Static File Serving**

1. CSS, JS, images, and QR codes are served over HTTP/HTTPS.

**Database Access**

1. The backend communicates with the SQLite database using SQL Alchemy ORM.

**DATABASE DESIGN**

**Entity-Relationship Diagram**

Below is a textual representation of the ERD for your Student Monitoring System. If you need a graphical diagram, let me know and I can generate a Mermaid diagram for you.

**Entities**

1. User
2. Student
3. Class
4. Attendance

**Relationships**

1. Each Student belongs to one Class.
2. Each Attendance record links a Student, a Class, and a Date.
3. Users (admin/teacher) manage Students, Classes, and Attendance.

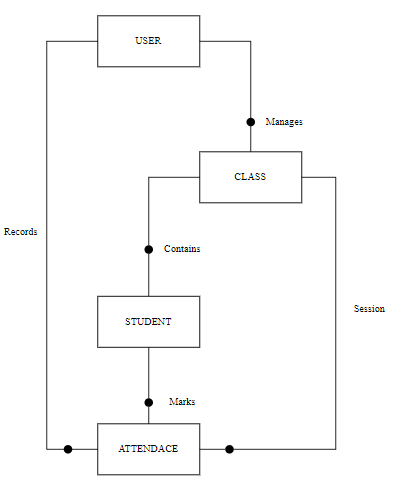


FIGURE 1

**Description of database tables, fields, and relationships**

**User Table**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| Id | Integer PK | Unique User ID |
| Username | String | User’s Login name |
| Password | String | Password |
| Role | String | User role (admin/teacher) |

**Class Table**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| Id | Integer PK | Unique User ID |
| Username | String | Class name |
| Description | String | Class description |

**Student Table**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| Id | Integer PK | Unique student ID |
| Name | String | Student’s full name |
| Class id | Integer FK | References Class(id) |
| Photo | String | Path to student photo |
| QR code | String | Path to student QR code |

**Attendance Table**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| Id | Integer PK | Unique student ID |
| student id | Integer FK | References Student(id) |
| Class id | Integer FK | References Class(id) |
| Date | Date | Attendance date |
| Status | String | Present/Absent/Late/etc. |
| Scan time | Date Time | Time of QR scan (if used) |

TABLE 4

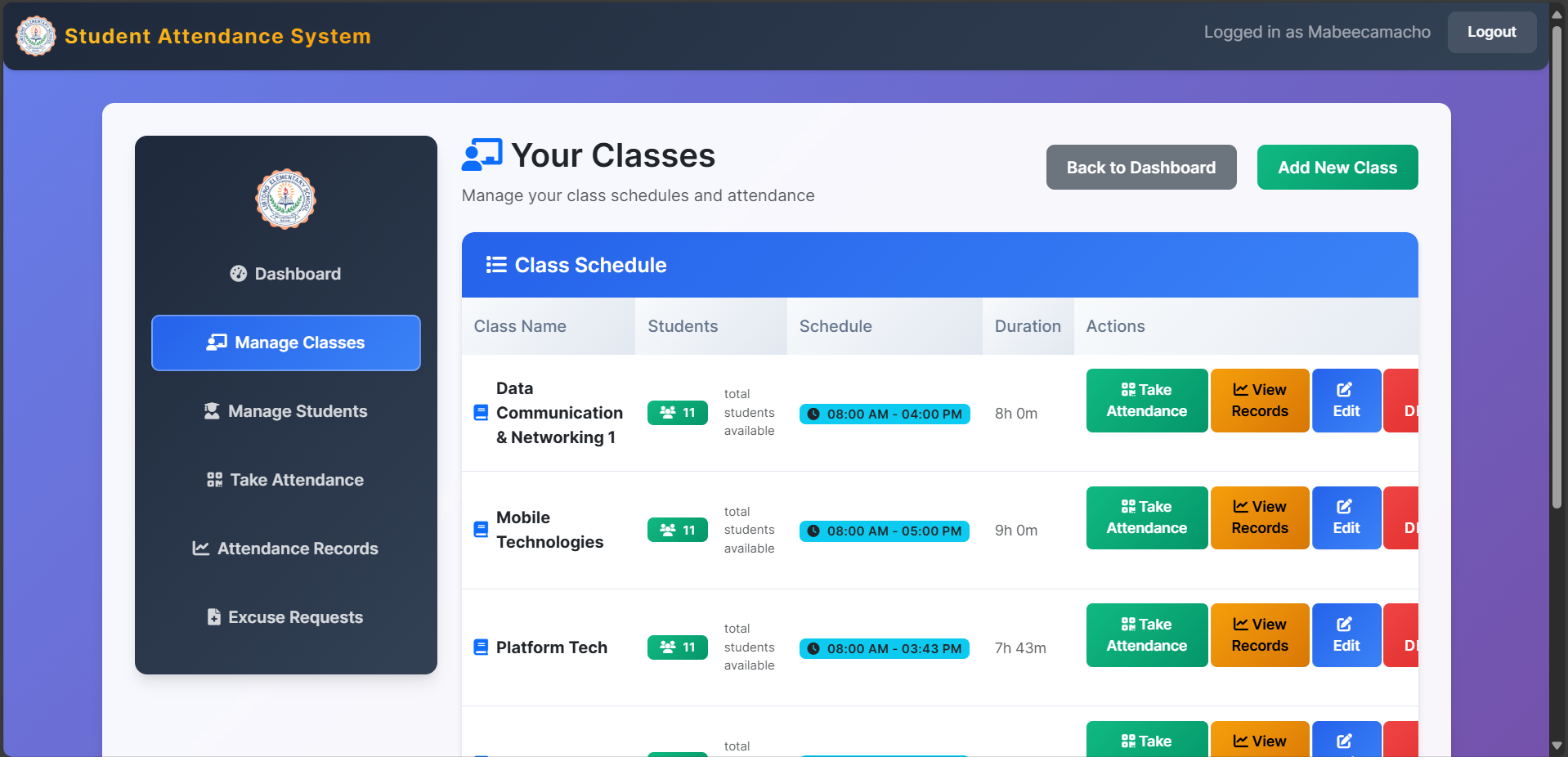
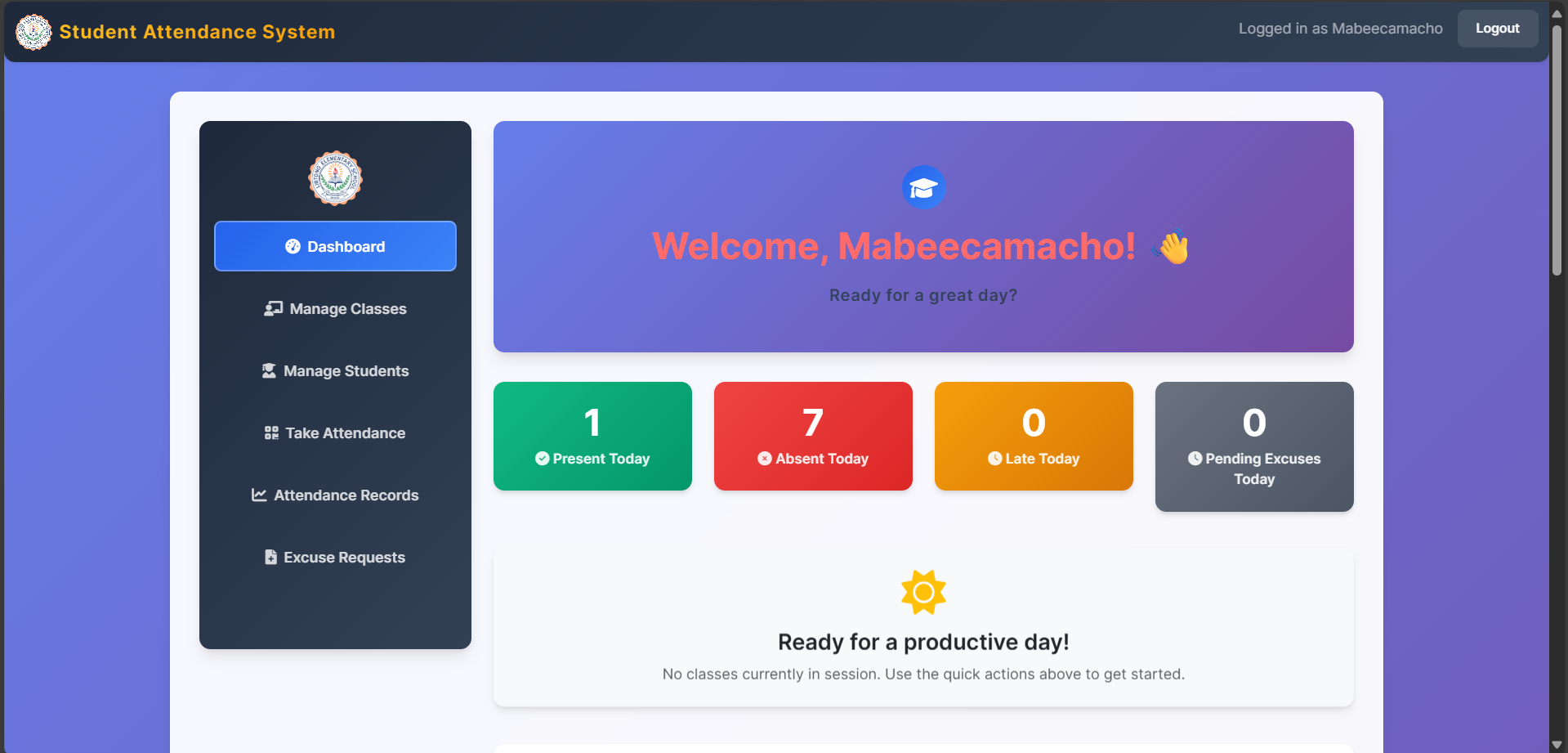
**Relationships**

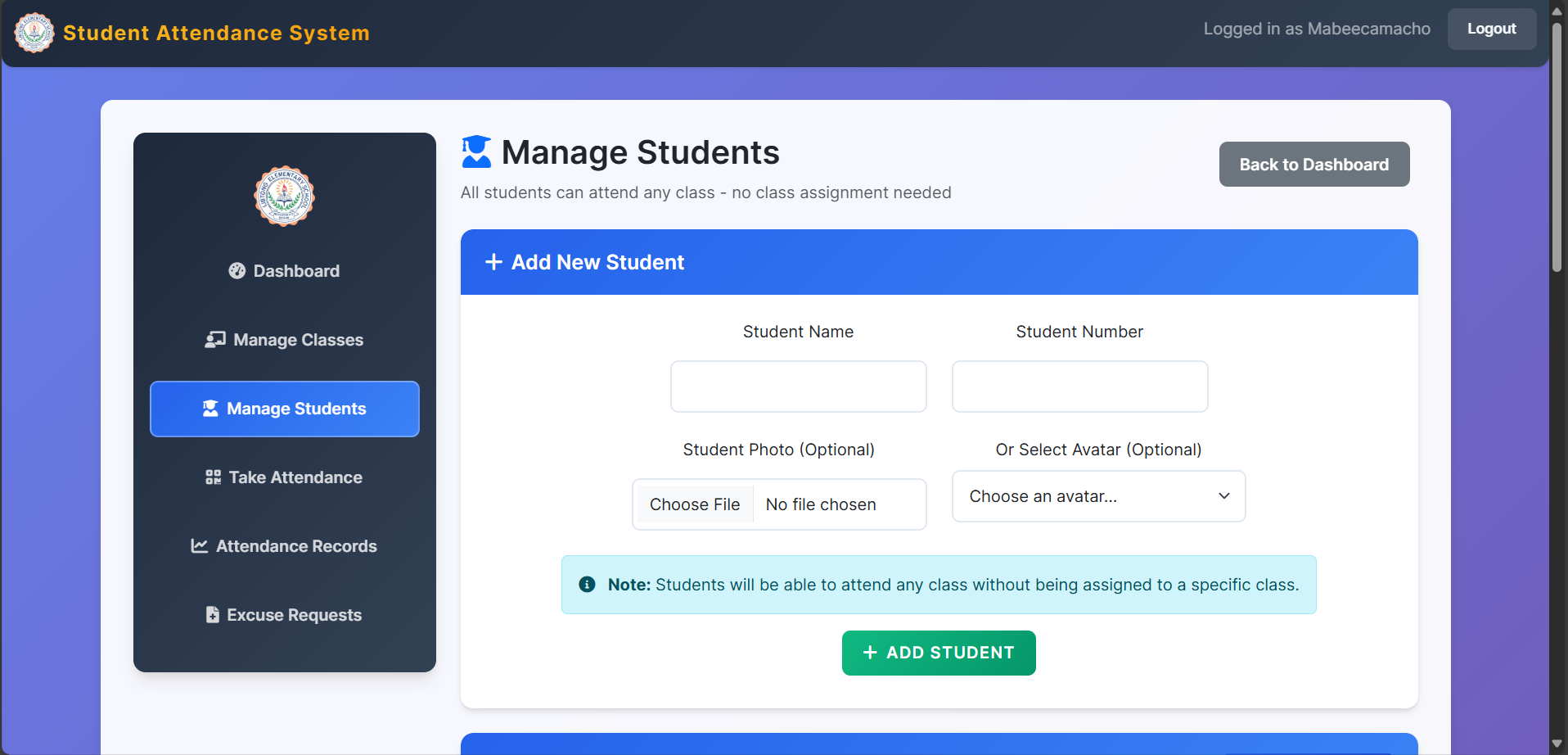
1. Each Student is linked to one Class.
2. Each Attendance record is linked to one Student and one Class.
3. Users can manage multiple Classes and Attendance records.

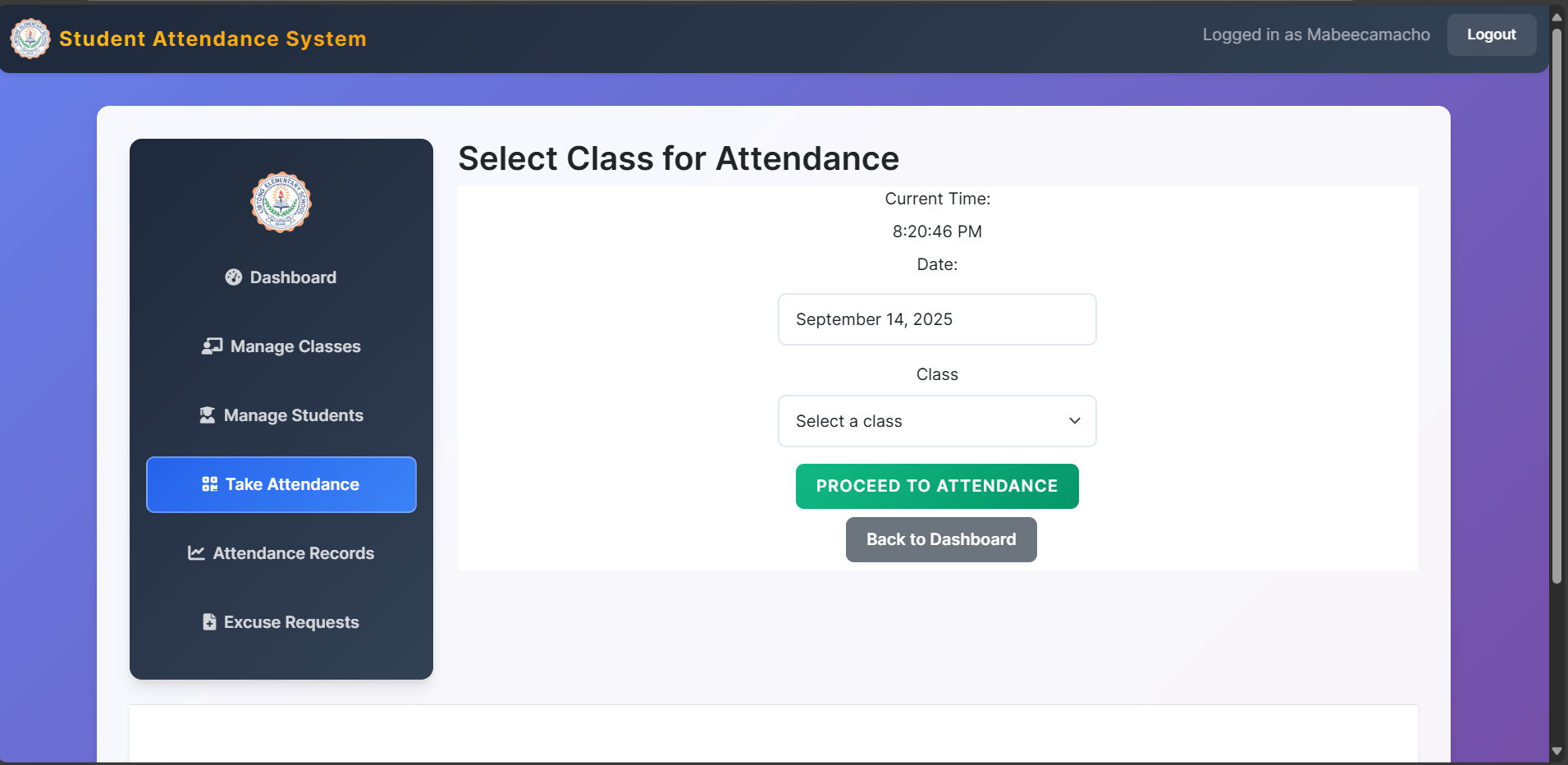
**Data Normalization Techniques Used**

1. **1st Normal Form (1NF):** All tables have atomic values; no repeating groups or arrays.
2. **2nd Normal Form (2NF):** All non-key attributes are fully functionally dependent on the primary key.
3. **3rd Normal Form (3NF):** No transitive dependencies; all fields depend only on the primary key. This normalization ensures data integrity, reduces redundancy, and improves maintainability.

**USER INTERFACE DESIGN**

****

****

****

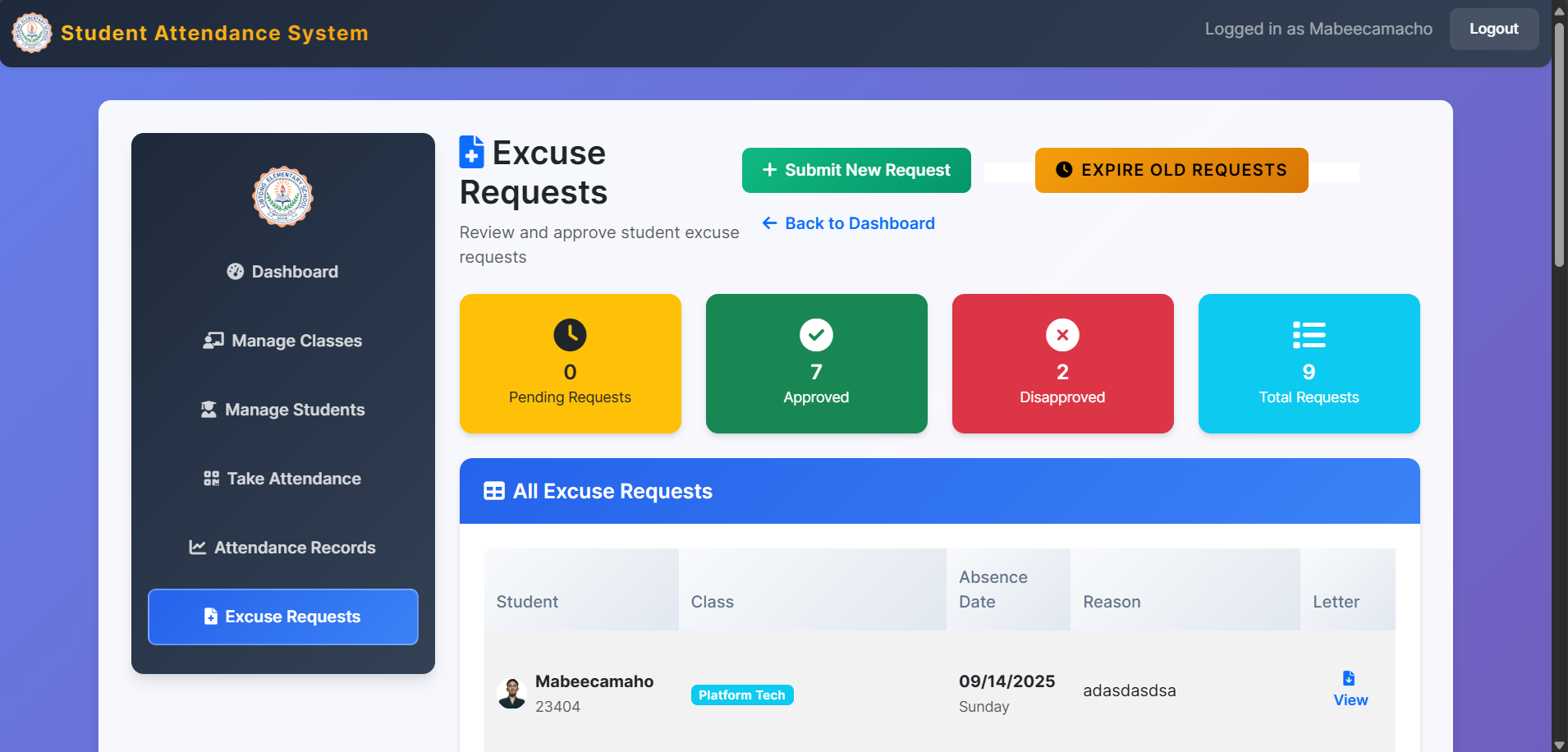
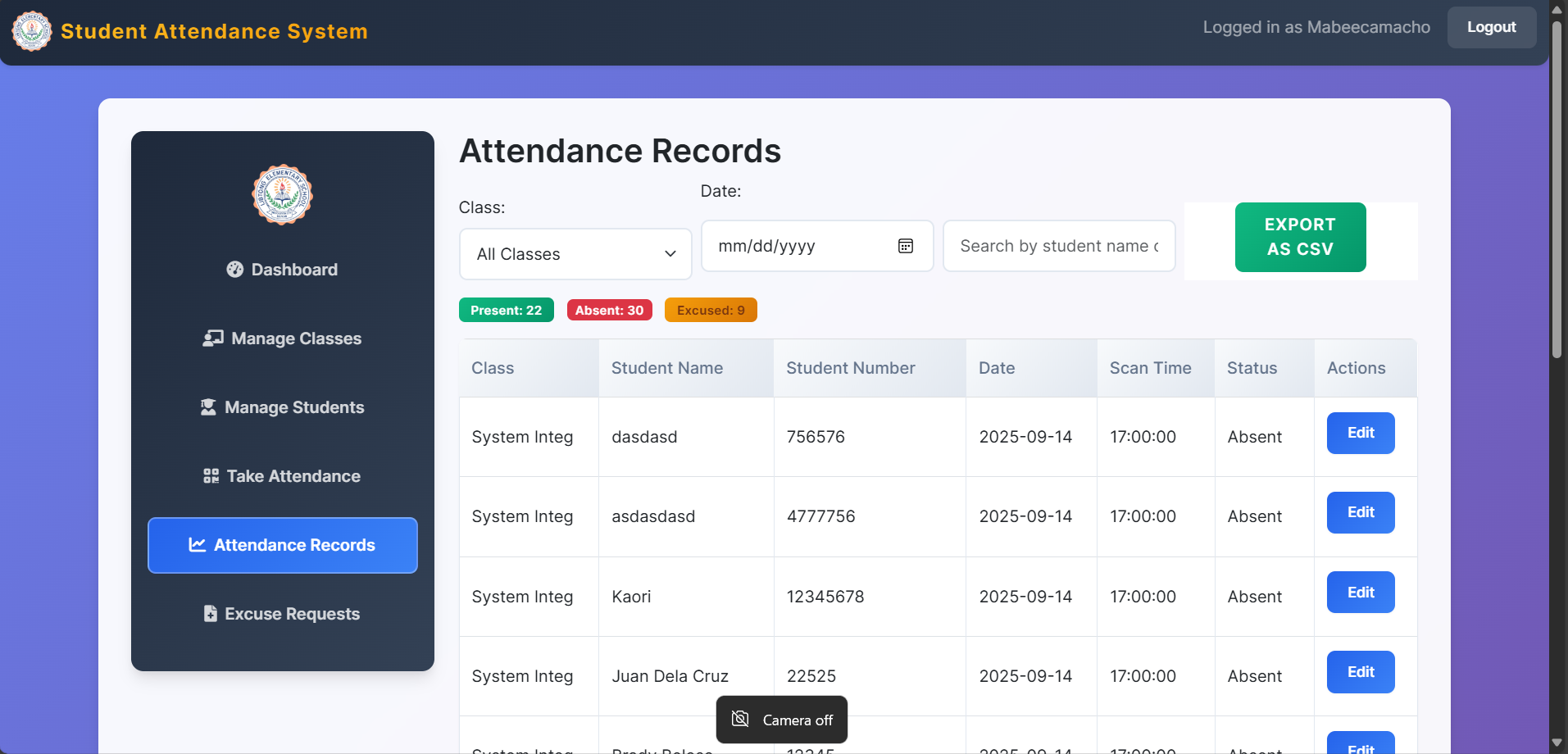
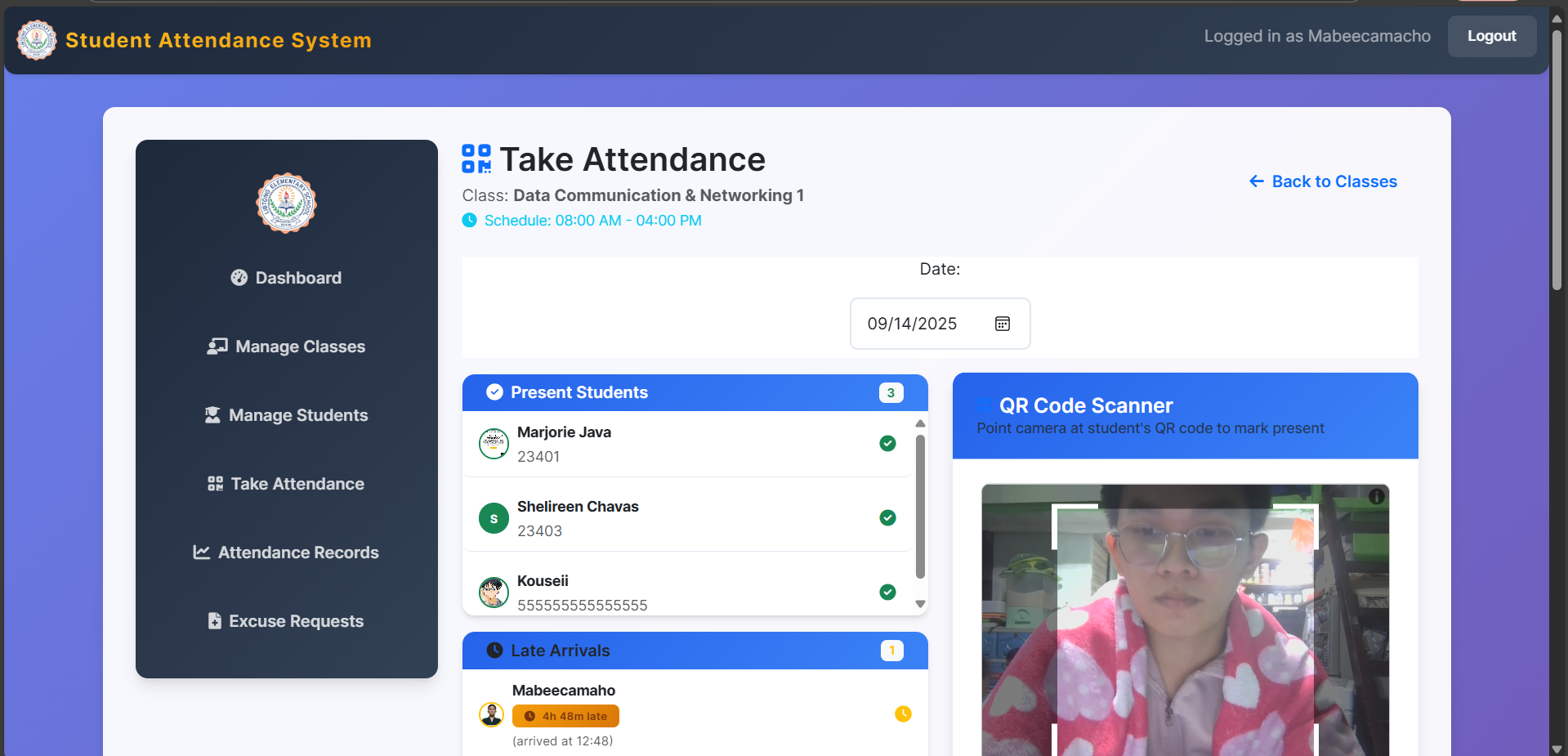
****

FIGURE 1

**Description Components and modules**

**Frontend (React) Purpose**

1. Provides the user interface for interacting with the system.

**Key Modules**

1. **App.js:** Main application component, handles routing and layout.
2. **Pages/Views:** Login, Dashboard, Students, Classes, Attendance, Records.
3. **Components:** Forms, tables, QR scanner, charts, navigation bar.

**Backend (Flask)**

1. **Purpose:** Handles business logic, API endpoints, authentication, and serves static files/templates.

**Key Modules**

1. **pp.py:** Main Flask application, route definitions, API endpoints.
2. **Templates:** Jinja2 HTML templates for server-rendered pages.
3. **Database Models:** ORM models for User, Student, Class, Attendance.
4. **Authentication:** User login, registration, session management.

**Database (SQLite)**

1. **Purpose:** Stores persistent data for users, students, classes, and attendance.

**Key Modules**

1. **Alembic Migrations:** Manages schema changes.
2. **ORM Models:** Define tables and relationships.

**Static Assets**

* 1. **Purpose:** Provides CSS, JavaScript libraries, images, and QR codes.

**Key Modules**

* 1. **style.css: Main stylesheet.**
  2. **login.js, chart.umd.min.js, html5-qrcode.min.js:** JS libraries for UI and QR scanning.

**Interface Specifications for Each Component**

**Frontend-Backend API (Example Endpoints)**

1. **POST /login Request:** (username, password)
2. **Response:** (success, token) GET /students
3. **Response:** (id, name, class id, photo, QR code)
4. **POST /attendance Request**: (student id, class id, date, status)
5. **Response:** success GET /classes
6. **Response:** (id, name, description)

**Backend-Database ORM Models (Example Signatures)**

1. **User** (id: int, username: str, password: str, role: str)
2. **Student** (id: int, name: str, class id: int, photo: str, QR code: str)
3. **Class** (id: int, name: str, description: str)
   1. **Attendance** (id: int, student id: int, class id: int, date: date, status: str, scan time: datetime)

**Frontend Components (Example Props/Functions)**

1. **Student List students** ={students} on Edit = {edit Student}
2. **Attendance Form on Submit** = {mark Attendance}
3. **QR Code Scanner on Scan** = {handle Scan}

**Dependency Management and Interaction Between Components**

**Frontend**

1. Uses npm/yarn for managing React and third-party libraries.
2. Communicates with backend via RESTful API using fetch or axios.
3. Imports CSS and JS libraries for styling and QR code scanning.

**Backend**

1. Uses pip/requirements.txt for managing Python and Flask dependencies.
2. Imports ORM models for database access.
3. Handles CORS and session management for secure API communication.

**Database**

1. Managed via SQL Alchemy ORM and Alembic for migrations.
2. Backend interacts with the database through ORM models, abstracting SQL queries.

**Static Assets**

1. Served by the backend or a web server.
2. Frontend references static files for styling and functionality.

**DATA FLOW DIAGTRAMS**

**Overview**

Data Flow Diagrams (DFDs) illustrate how data moves through the Student Monitoring System, identifying data sources, processing steps, and destinations. The diagrams below show both high-level and detailed flows for key operations.

**High-Level Data Flow Diagram Description**

At a high level, the system consists of Users (Teachers/Class President), the Web Application (Frontend), the Application Server (Backend), and the Database. Users interact with the frontend, which communicates with the backend to process and store data.

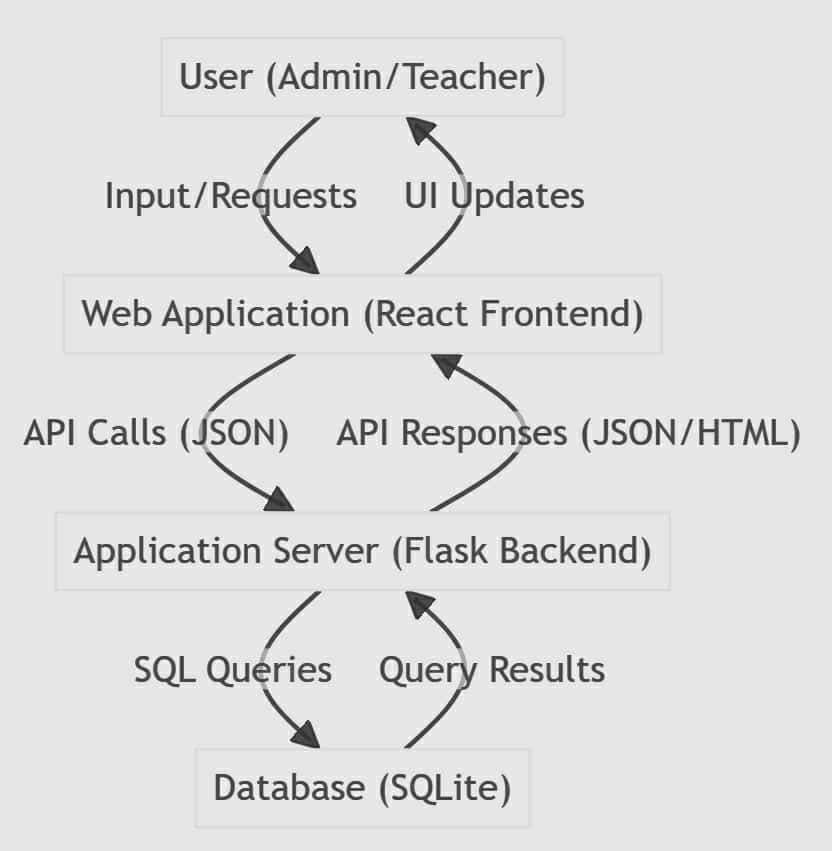


FIGURE 1

**Attendance Marking Data Flow Data Sources**

1. User (scans QR code) Student and Class data in the database Processing Logic
2. Frontend captures QR code and sends student/class info to backend Back end validates and records attendance Destinations.
3. Attendance record stored in the database Confirmatio n/status sent back to frontend

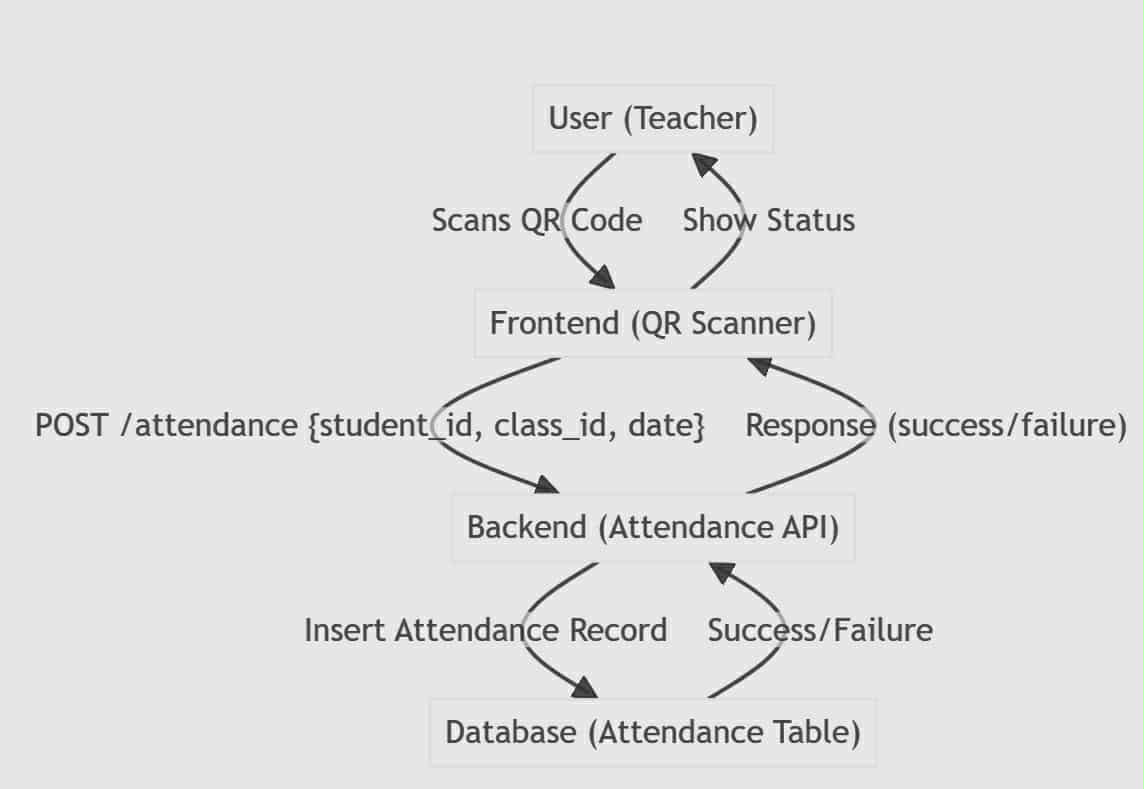


FIGURE 2

**Dentification of Data Sources, Processing Logic, and Destinations Data Sources**

**Data Sources**

1. User input (forms, QR scans) Existing records in the database Processing Logic
2. Frontend validation and formatting Backend authentication, business logic, and database operations Destinations
3. Database tables (User, Student, Class, Attendance) User interface (confirmation messages, updated views)

**SECURITY DESIGN**

Overview of Security Requirements and Considerations the Student Monitoring System is designed to protect sensitive student and attendance data, ensure only authorized users can access or modify records, and prevent common web vulnerabilities. Security is considered at every layer of the application, from user authentication to data storage and transmission.

**Key security requirements**

1. Only authenticated users can access the system.
2. Role-based access control (admin, teacher) restricts sensitive operations.
3. User credentials and sensitive data are protected at rest and in transit.
4. The system is resilient against common attacks (e.g., SQL injection, XSS, CSRF).

**Authentication and Authorization Mechanisms Authentication**

1. Users must log in with a valid username and password.
2. Passwords are never stored in plain text; they are hashed using a secure algorithm (bcrypt, werkzeug security).
3. Session management is handled by Flask’s session mechanism or secure tokens.
4. User sessions are validated on each request to protected endpoints. Authorization: Each user is assigned a role (e.g., admin, teacher).
5. Access to certain routes and actions is restricted based on user role.
6. Backend checks user roles before processing sensitive operations (e.g., managing students, editing attendance).

**Data Encryption and Protection Measures at Rest**

1. Passwords are stored as salted hashes in the database.
2. Sensitive files (e.g., student photos, QR codes) are stored in protected directories with restricted access.
3. Database access is limited to the backend application.

**In Transit**

1. All communication between frontend and backend should be secured using HTTPS in production.
2. API endpoints accept and return data in JSON, with input validation to prevent injection attacks.

**Other Protection Measures**

1. Input validation and sanitization on both frontend and backend to prevent XSS and injection attacks.
2. CSRF protection for forms and state-changing requests.
3. Rate limiting and account lockout mechanisms can be implemented to prevent brute-force attacks.
4. Regular updates and patching of dependencies to address known vulnerabilities.

**PERFORMANCE DESIGN**

**Performance Requirements and Objectives Responsiveness**

The system should provide a smooth and responsive user experience, with page loads and API responses typically under 1 second for standard operations.

**Scalability**

1. The system should be able to handle an increasing number of users, students, and attendance records without significant degradation in performance.

**Resource Efficiency**

1. The application should use server and client resources efficiently, minimizing unnecessary computation and data transfer.

**Strategies for Optimizing System Performance Frontend Optimization**

1. Minimize and bundle JavaScript and CSS files to reduce load times.
2. Use efficient React rendering practices (e.g., component memorization, avoiding unnecessary re-renders).
3. Lazy-load images and non-critical resources.
4. Use browser caching for static assets (CSS, JS, images, QR codes).

**Backend Optimization**

1. Optimize database queries using indexes and query optimization techniques.
2. Use ORM (SQL Alchemy) efficiently to avoid N+1 query problems.
3. Implement pagination for large data sets (student lists, attendance records).
4. Cache frequently accessed data where appropriate (class lists).
5. Serve static files via a dedicated web server (Nginx) in production.

**Deployment/Infrastructure**

1. Use a production-ready WSGI server (Gunicorn, uWSGI) for Flask.
2. Enable load balancing if scaling to multiple backend instances.
3. Monitor server resource usage and scale hardware as needed.
4. Performance Testing Plan

**Performance Testing Plan Unit and Integration Testing**

1. Test individual components and API endpoints for response time and correctness.

**Load Testing**

1. multiple concurrent users performing common actions (logging in, marking attendance, viewing records).
2. Use tools such as JMeter, Locust, or Artillery to measure system throughput and latency.

**Stress Testing**

1. Gradually increase the load to identify the system’s breaking point and observe how it recovers.

**Monitoring**

1. Implement logging and monitoring to track performance metrics (e.g., response times, error rates, resource usage). Set up alerts for abnormal performance or resource exhaustion.

**Optimization Review**

1. Regularly review and profile the application to identify and address performance bottlenecks.

**ERROR HANDLING AND LOGGING**

**Error Handling Mechanisms and Strategies Frontend**

1. User input is validated before submission (required fields, correct formats).
2. API errors are caught and displayed as user-friendly messages (“Failed to save attendance. Please try again.”).
3. React error boundaries are used to catch and display unexpected UI errors.

**Backend**

1. All API endpoints validate incoming data and handle invalid or missing parameters gracefully.
2. Try-except blocks are used to catch exceptions, ensuring the server returns a controlled error response rather than crashing.
3. Custom error handlers are implemented in Flask to return standardized error responses (400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Internal Server Error).
4. Sensitive error details are not exposed to the client; only generic messages are returned, while detailed logs are kept on the server.

**Logging Requirements and Specifications Backend Logging**

1. All errors and exceptions are logged with timestamps, error type, stack trace, and relevant request information.
2. Authentication attempts (success and failure) are logged for security auditing.
3. Important system events (user creation, data import) are logged for traceability.
4. Logs are written to a file (app.log) and/or a centralized logging system in production. Log levels are used (INFO, WARNING, ERROR, CRITICAL) to distinguish the severity of events.

**Frontend Logging**

1. Client-side errors are logged to the browser console for debugging.
2. Optionally, critical client-side errors can be reported to the backend for monitoring**.**

**Error Codes and Messages HTTP Status Codes**

1. **200 OK:** Successful request
2. **201 Created:** Resource successfully created
3. **400 Bad Request:** Invalid input or missing parameters
4. **401 Unauthorized:** Authentication required or failed
5. **403 Forbidden:** Insufficient permissions
6. **404 Not Found:** Resource does not exist
7. **409 Conflict:** Duplicate resource or conflicting request
8. **500 Internal Server Error:** Unexpected server error

**Error Message Guidelines**

**Messages returned to the client are clear, concise, and do not reveal sensitive system details.**

**Example**

1. ("error": "Invalid username or password.")
2. ("error": "Student not found.")
3. ("error": "You do not have permission to perform this action.")

**THIRD PARTY INTEGRATION**

List of Third-Party Services or APIs Integrated html5-qrcode JavaScript library used in the frontend for scanning QR codes via the device camera.

**Chart.js**

1. JavaScript library used for rendering charts and data visualizations in the frontend.

**Flask & Flask Extension**

1. Python web framework and its extensions (Flask-Login for authentication, Flask-Migrate for database migrations).

**Alembic**

1. Database migration tool for managing schema changes in SQLite.
2. React & React Libraries React.js and supporting libraries for building the frontend user interface.

**Description of Integration Points and Data Exchange Formats html5-qrcode (Frontend)** **Integration Point**

1. Used in the attendance marking page to scan student QR codes.

**Data Exchange**

1. Captures QR code data (usually student ID) and sends it to the backend via a REST API call in JSON format. Chart.js (Frontend)

**Integration Point**

1. Used in dashboard and reporting pages to visualize attendance and student data.

**Data Exchange**

1. Receives data from backend API (JSON), which is then formatted and passed to Chart.js for rendering. Flask Extensions (Backend)

**Integration Point**

1. Used for user authentication (Flask-Login), session management, and database migrations (Flask-Migrate).

**Data Exchange**

1. Internal to the backend; manages user sessions, database schema, and migration scripts. Alembic (Backend)

**Integration Point**

1. Used during development and deployment to apply database schema changes.

**Data Exchange**

1. Operates on the SQLite database using migration scripts. React & Supporting Libraries (Frontend)

**Integration Point**

1. Used throughout the frontend for UI rendering, state management, and routing.

**Data Exchange**

1. Communicates with the backend via HTTP/JSON APIs.

**DEPLOYMENT PLAN**

Overview of the Deployment Process The deployment process for the Student Monitoring System involves preparing the backend (Flask), frontend (React), database (SQLite), and static assets for production use. The system can be deployed on a local server, institutional intranet, or a cloud-based virtual machine.

**Deployment Steps**

1. Set up the server environment (install required software and dependencies).
2. Clone the project repository from version control.
3. Install Python and Node.js dependencies.
4. Build the frontend (if using a separate React build process).
5. Apply database migrations to ensure the schema is up to date.
6. Configure environment variables and application settings.
7. Start the backend server (using a production WSGI server like Gunicorn or uWSGI).
8. Serve static files and frontend assets (via Flask or a web server like Nginx).
9. Monitor logs and system health.

**Hardware and Software Requirements for Deployment Hardware Requirements**

**Minimum**

1. 2 CPU cores
2. 2 GB RAM 10 GB disk space

**Recommended**

1. 4+ CPU cores
2. 4+ GB RAM
3. SSD storage for faster access

**Software Requirements**

1. **Operating System:** Windows, Linux, or macOS (Linux recommended for production)
2. **Python:** Version 3.7 or higher
3. **Node.js and npm:** For building the React frontend
4. **SQLite:** (included with Python, no separate install needed)
5. **Web Server:** Nginx or Apache (optional, for serving static files and reverse proxy)
6. **Git:** For version control and deployment WSGI Server: Gunicorn or uWSGI for running Flask in production

Configuration Management and Version Control Procedures Version Control

1. All source code is managed using Git.
2. The main branch is protected feature branches are used for development.
3. Commits are descriptive and atomic.
4. Releases are tagged for deployment.

**Configuration Management**

1. Environment-specific settings (e.g., database URI, secret keys) are stored in environment variables or a .env file (not committed to version control).
2. Sensitive information (passwords, secret keys) is never stored in the codebase.
3. Database migrations are managed using Alembic and tracked in version control.
4. Dependency versions are pinned in requirements.txt (Python) and package.json (Node.js).

**Deployment Automation (Optional)**

1. Deployment scripts or CI/CD pipelines (e.g., GitHub Actions, GitLab CI) can be used to automate testing, building, and deployment steps.

**MAINTENANCE AND SUPPORT**

**Guidelines for System Maintenance and Support**

**Regular Monitoring**

1. Continuously monitor system logs, performance metrics, and user feedback to detect issues early.

**Documentation**

1. Maintain up-to-date technical and user documentation to assist with troubleshooting and onboarding new maintainers.

**Backup Procedures**

1. Schedule regular backups of the database and critical files to prevent data loss.

**Access Control**

1. Limit administrative access to trusted personnel and review access permissions periodically.

**Procedures for Handling Software Updates, Patches, and Bug Fixes**

**Update Process**

1. Pull the latest code from the version control repository.
2. Review the changelog and update notes for new features, bug fixes, or security patches.
3. Test updates in a staging environment before applying them to production.
4. Apply database migrations if required. Deploy the updated application to production.
5. Monitor the system for any post-update issues.

**Patch Management**

1. Apply security patches to dependencies (Python packages, Node modules) promptly.
2. Use dependency management tools (pip, npm) to check for and install updates.
3. Document all changes and patches applied to the system.

**Bug Fixes**

1. Log all reported bugs in an issue tracker.
2. Prioritize and assign bugs based on severity and impact.
3. Develop, test, and review fixes before deployment.
4. Communicate resolved issues to users and stakeholders.

**Escalation Process for Resolving Issues First-Level Support**

1. Routine issues and user questions are handled by the designated support team or system administrator.

**Second-Level Support**

1. Complex technical issues or bugs that cannot be resolved at the first level are escalated to the development team.

**Third-Level Support**

1. Critical system failures, security breaches, or issues requiring vendor or third-party intervention are escalated to senior management or external support providers.

**Issue Tracking and Communication**

1. All issues are logged in an issue tracking system (e.g., GitHub Issues, Jira).
2. Status updates and resolutions are communicated to all relevant stakeholders.
3. Post-incident reviews are conducted for major incidents to prevent recurrence.

**REVISION HISTORY**

|  |  |  |
| --- | --- | --- |
| **VERSION** | **DATE** | **DESCRIPTION** |
| 1.0 | |  | | --- | | 2025-07-15 | | Initial draft created |
| 1.1 | |  | | --- | | 2025-8-15 | | |  | | --- | |  |  |  | | --- | |  |   Flowchart |
| 1.2 | 2025-8-20 | Organize the table |

TABLE 1