DHANALAKSHMI SRINIVASAN ENGINEERING COLLEGE
(AUTONOMOUS)
(Approved by AICTE & Affiliated to Anna University, Chennai)
Re-Accredited by NAAC with 'A' Grade
Accredited by NBA for AERO, BME, CSE, ECE, EEE, IT & MECH.
PERAMBALUR-621212, TAMILNADU, INDIA.
Website: www.dsengg.ac.in

COMPLAINT MANAGEMENT SYSTEM

PROJECT REPORT

Submitted by: Group 5

Aryan Kumar Gupta	*Backend Lead*
Asina Begam A	*Backend Support & Database*
Gowdham R	*Flutter UI Developer*
Elangovan P	*Flutter Integration & Documentation*

ABSTRACT

In many institutional environments, complaint handling is often decentralized, relying on manual registers, verbal communication, or disjointed email threads. This traditional approach leads to significant inefficiencies, including delayed resolutions, a lack of accountability, and difficulty in tracking the real-time status of grievances.

To address these challenges, Group 5 has developed the **Complaint Management System**, a cross-platform mobile application designed to bridge the communication gap between users (students/staff) and administrators. The system streamlines the entire lifecycle of a complaint—from lodging and tracking to resolution and closure—through a unified digital interface.

Built using Flutter for a high-performance, responsive frontend and Core PHP with MySQL for a lightweight, secure backend, the application ensures real-time status updates and efficient role-based management. This report documents the development lifecycle, architectural decisions, and the technical implementation of the system, highlighting how digital intervention can transform facility management.

INTRODUCTION

Overview

Traditional methods of complaint management suffer from significant inefficiencies. Physical complaint boxes are rarely checked, and manual logbooks make it impossible to generate data on recurring issues or facility performance. When a complaint is recorded on paper, it lacks transparency; the complainant rarely knows if the issue has been seen or acted upon.

The Need for Digitalization

The Complaint Management System introduces a digital centralized platform to replace these archaic methods. The primary objective is to bring transparency, accountability, and efficiency to the process.

Objectives

By digitizing this workflow, our system aims to:

1. **Reduce Turnaround Time:** Eliminate the physical lag time between a complaint being written and a maintenance engineer reading it.
2. **Enhance Visibility:** Provide users with real-time tracking (Pending, In Progress, Resolved) so they are never left in the dark.
3. **Empower Administration:** Give admins tools to assign tasks automatically or manually to the relevant departments and monitor performance.

PROBLEM STATEMENT

Through our initial analysis of existing manual systems in hostels and campus environments, we identified four critical pain points that this project specifically addresses:

1. Lack of Accountability

When a complaint is written in a generic physical book, it is unclear who is specifically responsible for solving it. There is no digital audit trail to show who viewed the complaint or assigned it.

2. Manual Tracking Inefficiencies

Searching for past complaints or verifying if a specific issue was resolved requires sifting through physical pages. This makes it impossible to detect patterns, such as "Which wing has the most electrical failures?"

3. Delay in Resolution

Without digital notifications, maintenance teams often receive complaints days after they are lodged. A plumbing leak reported on Friday might not be seen until Monday in a manual system.

4. Communication Gaps

The most common frustration for users is the "black box" effect. Users rarely receive feedback on whether their complaint was accepted, rejected, or completed, leading to dissatisfaction and repeated complaints for the same issue.

PROPOSED SOLUTION

Our solution is a robust, role-based mobile application that automates the complaint workflow. The system is designed to handle multiple categories of complaints with specific logic for each.

Key Solution Features:

- **Role-Based Access Control (RBAC):**

The system features distinct interfaces for **Users** (to lodge and track complaints) and **Admins** (to view, assign, and resolve them). This ensures security and relevant data visibility.

- **Automated Routing & Prioritization:**

The system uses logic to categorize complaints. For example, complaints tagged as "Critical Infrastructure" or "Electrical" can be automatically flagged with **High Priority**.

- **Transparent Status Lifecycle:**

Every ticket follows a strict lifecycle:

    - **Pending:** Complaint lodged, waiting for admin review.
    - **In Progress:** Admin has assigned the task to an engineer.
    - **Resolved:** The work is complete.

- **Admin Monitoring Dashboard:**

Admins are provided with a high-level dashboard to view pending issues sorted by urgency, ensuring that critical problems are addressed first.

SYSTEM ARCHITECTURE

The system follows a classic **Client-Server Architecture** optimized for mobile performance:

1. Frontend (Client)

- **Technology:** Flutter (Dart)
- **Role:** Runs on the user's mobile device. It handles UI rendering, input validation, and displays data. It communicates with the server via asynchronous HTTP requests.

2. Backend (Server)

- **Technology:** Core PHP (REST APIs)
- **Role:** Hosted on an Apache server. We utilized Core PHP without heavy frameworks to maintain full control over the Request/Response cycle and minimize server overhead. It exposes RESTful endpoints (GET, POST, PUT).

3. Database

- **Technology:** MySQL
- **Role:** Stores all relational data including user profiles, complaints, categories, and audit logs.

4. Security Layer

- **JWT (JSON Web Tokens):** Used for stateless authentication. Every API request includes a token in the header to validate the user's identity.
- **Data Flow:** User Input → Flutter Service → JSON Payload → PHP API → PDO Query → MySQL → Response.

DATABASE DESIGN

We designed the database in **3rd Normal Form (3NF)** to reduce redundancy and ensure data integrity.

Primary Tables:

1. users

- Stores authentication details.
- **Columns:** id (PK), name, email (Unique), password_hash, role (enum: 'admin', 'user'), created_at.

2. categories

- Stores the types of complaints available.
- **Columns:** id (PK), name (e.g., 'Internet', 'Hostel', 'Academic').

3. complaints

- The core transaction table.
- **Columns:** id (PK), user_id (FK), category_id (FK), title, description, priority (High/Medium/Low), status (Default: 'Pending'), created_at.

4. admin_responses

- Stores the resolution notes provided by admins.
- **Columns:** id (PK), complaint_id (FK), admin_id (FK), response_text, updated_at.

Entity Relationships:

- **One-to-Many:** A User can lodge multiple Complaints.
- **Many-to-One:** A Complaint belongs to exactly one Category.
- **One-to-Many:** An Admin can provide multiple Responses.

CORE FEATURES IMPLEMENTATION

1. Complaint Registration

Users can fill out a dynamic form. They select a category from a dropdown (fetched via API) and set a priority level. The Flutter app validates that the description is not empty before sending the data to the create_complaint.php endpoint.

2. Dynamic Status Updates

This is the hallmark feature of the system.

- **Admins:** Have write access to the status field. They can change a ticket from 'Pending' to 'Resolved'.
- **Users:** Have read-only access to the status. They can view the progress but cannot manipulate it.

3. Automatic Assignment Logic

To assist admins, the backend implements basic automation. When a complaint is inserted, the script checks the category. If the category is flagged as **"Emergency"** (e.g., Fire Safety), the priority is automatically forced to **"High"**, regardless of user input.

4. Admin Dashboard

The Admin Dashboard is designed for productivity. It displays a list of complaints filtered to show "Pending" items first, sorted by Date (Oldest First), ensuring that no complaint is ignored for too long.

BACKEND IMPLEMENTATION (PHP)

The backend logic, led by **Aryan** and **Asina**, prioritizes security and a clean folder structure over using a monolithic script.

Folder Structure:

- /config: Contains db.php for the PDO database connection.
- /api: Contains the endpoint scripts (login.php, create_complaint.php, update_status.php).
- /models: PHP classes representing database entities.

Key Technical Decisions:

1. PDO (PHP Data Objects):

We used PDO exclusively instead of mysqli. PDO allows for consistent database access and supports named parameters, which makes the code cleaner and safer.

2. Prepared Statements:

To prevent SQL Injection, every query uses prepared statements.

- *Bad:* SELECT * FROM users WHERE email = '$email'
- *Implemented:* prepare("SELECT * FROM users WHERE email = ?")

3. REST Standards:

Since we did not use a framework, we manually handled HTTP verbs. The scripts check $_SERVER['REQUEST_METHOD'] to ensure that a GET request cannot modify data and a POST request is required for creating records.

FRONTEND IMPLEMENTATION (FLUTTER)

The frontend, developed by **Gowdham** and **Elangovan**, focuses on a clean User Experience (UX).

Architecture:

We adopted the **Service-Repository Pattern**.

- **UI Layer:** Widgets only handle display logic.
- **Service Layer:** Files like auth_service.dart and complaint_service.dart handle the HTTP communication. This separation makes the code modular and easier to debug.

State Management:

We utilized the **Provider** package. This allows us to manage the user's session state globally. When a user logs out, the Provider notifies all listeners, and the app instantly clears the dashboard and redirects to the login screen, preventing data leaks.

Theming & UI Components:

- **Consistency:** A unified Blue/White color palette was used to look professional.
- **Custom Widgets:** We created reusable components, such as StatusIndicator chips (Red for Pending, Green for Resolved), to ensure visual consistency across the User and Admin dashboards.

SECURITY IMPLEMENTATION

Security was not an afterthought but integrated from the design phase.

1. Password Hashing

We strictly adhere to the rule of never storing plain-text passwords.

- **Registration:** We use password_hash($password, PASSWORD_BCRYPT) to create a secure hash.
- **Login:** We use password_verify() to check credentials.

2. SQL Injection Prevention

By strictly using **PDO Prepared Statements**, we ensure that user input is treated as data, not executable code. This neutralizes common attacks where hackers try to manipulate database queries via input fields.

3. CORS Handling (Cross-Origin Resource Sharing)

To allow the Flutter web build (used during testing) and mobile app to communicate with the local server, we implemented specific headers:

- Access-Control-Allow-Origin: *
- Access-Control-Allow-Headers: Content-Type, Authorization

4. Input Validation

- **Client-Side:** Flutter FormKey validation ensures fields are not empty.
- **Server-Side:** PHP checks empty() and sanitizes input strings to prevent malicious scripts (XSS).

ERROR HANDLING & VALIDATION

A robust system must handle failures gracefully without crashing.

Backend Error Handling

We wrapped all database operations in try-catch blocks.

- **Logic:** If a query fails (e.g., duplicate email), the API does not expose the raw SQL error.
- **Response:** It returns a structured JSON error:

{ "success": false, "message": "Email already exists." }

This ensures the app can display a friendly error message to the user.

Frontend Network Handling

Mobile apps often face connectivity issues.

- **SocketException:** We specifically catch SocketException in Dart. If the server is unreachable, the app displays a SnackBar saying "Check Internet Connection" rather than crashing to a grey screen.

Input Sanitization

- We implemented .trim() on all text fields. This prevents users from registering with an email address that has accidental spaces at the end, which is a common user error.

GIT & DEVELOPMENT WORKFLOW

To ensure we met the evaluation rubric for team collaboration, we adhered to strict Version Control protocols.

Branching Strategy

We avoided pushing code directly to the main branch. instead, we used feature branches:

- feature/auth-login: For Aryan’s authentication logic.
- feature/ui-dashboard: For Gowdham’s UI work.
- fix/db-connection: For Asina’s database patches.

Commit Discipline

We enforced a standard commit message format to ensure the history was readable.

- *Bad:* "update code"
- *Good:* "Added JWT token parsing logic to middleware"

Merge Management

Elangovan acted as the integration manager. Before merging any branch into main, he ensured that the Flutter UI changes did not break the existing API integration. This prevented "integration hell" during the final phase of the project.

CHALLENGES FACED

Development is rarely smooth. The team overcame the following realistic technical challenges:

1. Auto-Assignment Case Sensitivity

Issue: The auto-assignment logic initially failed because "Electrical" and "electrical" were treated as different categories.

Fix: We implemented strtolower() in PHP to normalize all category inputs before comparison.

2. JWT Header Extraction

Issue: On our local Apache server, the Authorization header was being stripped out before it reached our PHP script, causing login failures.

Fix: We had to modify the .htaccess file to explicitly allow the Authorization header to pass through to the script.

3. Google Authentication Debugging

Issue: While attempting to add Google Sign-In, we faced 12500 errors.

Fix: We realized the SHA-1 fingerprint generated in our local keystore didn't match the one registered on the Firebase console. Updating the SHA-1 fixed the issue.

4. Data Type Mismatches

Issue: The Flutter app occasionally crashed because it treated numeric status codes (like priority levels) as Strings.

Fix: We strictly defined data models in Dart to map incoming JSON numbers to Integers correctly.

TESTING & EXECUTION

Backend Environment Setup

1. Navigate to the project root folder.
2. Start the built-in PHP server: php -S localhost:8000.
3. Ensure MySQL services (XAMPP/WAMP) are running on Port 3306.

Frontend Environment Setup

1. Connect a physical Android device via USB (USB Debugging must be enabled).
2. Run dependencies command: flutter pub get.
3. Execute the app: flutter run.

Testing Strategy

- **Happy Path Testing:** We verified that a user can successfully register, login, lodge a complaint, and see it appear on the dashboard.
- **Negative Testing:** We intentionally entered invalid emails and wrong passwords to ensure the system rejected them with the correct error messages.
- **Role Validation:** We attempted to access Admin APIs using a User token to verify that the Middleware correctly blocked unauthorized access (Returning HTTP 403 Forbidden).

CONCLUSION

The **Complaint Management System** developed by **Group 5** successfully meets all the objectives outlined in the project proposal. It provides a structured, secure, and user-friendly platform for resolving grievances within an institution.

Key Achievements:

1. **Scalability:** By implementing Role-Based Access Control and distinct dashboards, the system can easily support hundreds of users without UI clutter.
2. **Efficiency:** The automated prioritization logic proves that digital systems can reduce administrative workload.
3. **Integration Success:** The project demonstrated the team's ability to integrate a modern cross-platform frontend (Flutter) with a raw, optimized backend (Core PHP).

This project has not only solved the problem of manual complaint tracking but also provided the team with valuable hands-on experience in Full Stack Development, API Security, and Team Collaboration using Git.

FUTURE ENHANCEMENTS

To evolve this project from a Mini Project to a production-grade commercial application, we propose the following future updates:

1. Push Notifications

Currently, users must open the app to check status. Integrating **Firebase Cloud Messaging (FCM)** would allow us to send real-time push notifications to the user's lock screen immediately when an admin updates the complaint status.

2. Complaint Escalation Logic

We plan to implement a cron job (scheduled task) that checks for complaints that have been "Pending" for more than 7 days. These should be automatically forwarded to higher authorities or flagged as "Overdue."

3. Analytics Dashboard

Admins would benefit from visual data. We aim to add a graphical dashboard (using charts) to show:

- Which category receives the most complaints? (e.g., Pie chart of "Electrical" vs "Plumbing").
- Average time taken to resolve a complaint.

4. Cloud Deployment

Currently running on localhost, the next step is hosting the PHP backend on a cloud provider like **AWS EC2** or **Heroku** to allow global access over the internet.