Project Report : FIXIT (Faculty Issue eXchange and Information Tracker)

Problem Statement

The current campus maintenance reporting process involves actual face-to-face reports and sometimes rely on more paper-based forms submitted to the services departments. This manual approach causes delays, lacks transparency, and is prone to human errors. Misplaced reports can also introduce assurance and security risks to consider as it presents significant information. On paper-based forms, personally identifiable information (PII) are easily accessible and may even be unintentionally altered. There is no access control to restrict who has access to or handle the reports, no audit trail to keep track of modifications or administrative actions, and no authentication mechanism to verify who made the request. Without data integrity and security, the likelihood of maintenance records being lost, altered, or tampered with without permission could rise. These restrictions compromise the overall security of the campus's maintenance data as well as operational efficiency and transparency.

Proposed Solution

To address these challenges, the FIXIT (Faculty Issue eXchange and Information Tracker) introduces a flet-based maintenance reporting system designed with information security and assurance principles. The application integrates CSPC Google OAuth sign-in, role-based authentication, and controlled administrative access that ensures the authorized access for submitting, viewing, or managing reports. The users data and maintenance reports are securely secured within the system, it also has audit log tracks that records all administrative actions for tracing and accountability. Through digital submission, reducing unauthorized access through well-defined user roles, and enhancing transparency by allowing the users to have privilege in monitoring their report status, the data integrity of the platform improves. The categorization using an

AI-powered model ensures accurate and consistent processing. Through these security-focused functions, FIXIT transitioned from a vulnerable paper-based workflow to a protected, efficient, and reliable digital solution for maintaining a safer and well-functioning campus environment.

Framework Chosen & Rationale

The framework used for the development of the application was **Flet** that is Python-based cross-platform framework for mobile, web, and desktop applications. Flet offers a simple architecture that eliminates the code complexity for using a single codebase and context switching and speeds up prototyping for mobile-first UIs. This framework uses Flutter to ensure a high-quality and professional user interface. This also aligned with the team's Python-based Machine Learning and backend stacks. Flet integrates well with the OAuth floe and local server approach that is used for Google sign-in, and it gets along with the event-driven UI code that are already present in the project.

Trade-offs: Flet offers a significant productivity advantage for a semester-long project, but it is less advanced than some mobile-native frameworks (React Native, Flutter) and would need further testing across various device families.

Implemented Features

Baseline Features (Core Requirements)

- User Authentication& Profile with:
    - Secure login and logout functionality that once a user has logged out, their tokens are invalidated. This prevents unauthorized reusing session access.
    - The system used a secure and standard Google Oauth 2.0.

- Role-based Access Control (RBAC) (Admin, Student, Faculty) with:
    - Local admin credential validation (validate_admin_credentials) for demo/admin account.
    - Google OAuth aimed for CSPC school email restrictions for Student/Faculty login via google_ought_login().

- Report submission for capturing description and campus location.
- Profile Management Functionality that allows users to:
    - View & edit profile fields (name, email, etc.)
    - Change password (with current password verification)
    - Profile picture upload (validate type/size)
- Data Layer that used SQLite (may abstract to allow future DB migration)
- Logging baseline security that allows:
    - Authentication success/failure
    - Administrative actions (user create/delete/role change)
- Secure Configuration that practices by ensuring sensitive values such as: SECRET_KEY, database credentials, API keys (if any) are not hard-coded into the source code.
- User dashboard that allows status tracking for submitted issues (Pending, On-going, Fixed, Rejected) with UI tabs.
- Admin dashboard that has the report list, filters, and the status update actions (entry points such as admin_dashboard)
- Persistent storage of reports.

Enhancements/Value-Adds

1. **AI-assisted Categorization**

A trained Multinomial Naive Bayes Model (CounterVectorizer to MultinomialNB) that classifies free-text issue descriptions into categories such as Electrical, Plumbing, Furniture, and many more, The model also include heuristic for 'Uncategorized' classification with gibberish detection and unknown token.

2. **Google OAuth 2.0 Authentication:**

Aims to secure login via google_auth.py, strictly enforcing institutional email domains. Eliminates the risk of storing passwords locally; ensures only verified campus members can access the system.

3. **Automated Audit Logging:**

An immutable CSV logger records every critical system action such as Logins and Status Changes. This provides a security trail for accountability and dispute resolution.

4. Session Security:

An activity monitor tracks user interaction and auto-logs out inactive sessions. This aimed to protect sensitive data on shared campus computers.

Architecture & Module Overview

High-level Components

1. **Client Layer (The User Interface):**
    - Built entirely using Flet (Flutter for Python), ensuring a unified codebase for Web, Desktop, and Mobile (Android/iOS) platforms.

- Provides consistent UI/UX for both User (Reporting/Tracking) and Admin (Management/Analytics) roles.

2. **Presentation Layer (Views):**
   - Located in `app/views/`, this layer handles the UI rendering and user interaction logic.
   - Key components include the `loginpage.py` for authentication, `dashboard/` modules for role-specific views, and `components/` for reusable widgets like the Session Timeout UI.
3. **Business Logic Layer (Services):**
   - Located in `app/services/`, this layer contains the core application rules.
   - Auth Service: Manages Google OAuth 2.0 and Admin credential validation.
   - AI Service: Runs the Naive Bayes model for report categorization.
   - Audit/Activity: Handles logging and user monitoring.
   - Session Manager: Controls token validation and timeouts.
4. **Data Layer (Persistence):**
   - Primary: Google Firestore (NoSQL) for scalable, real-time cloud storage of users, reports, and logs.
   - Secondary: Local Storage (`app_database.db` / JSON) for offline caching and development fallback.

Data Flow & Component Interaction

The application relies on a strict data flow where Views never communicate directly with the Database; they must pass through Services.

Example: Authentication Flow

1. **User Action:** User clicks "Login with Google."
2. **Service Call:** `loginpage.py` invokes `google_auth.py` in the Service Layer.

3. **External Verification:** The app redirects to Google Identity Services for validation.
4. **Session Creation:** Upon success, `session_manager.py` creates a secure session token and stores it in Firestore.
5. **Logging:** `activity_monitor.py` records the `LOGIN_SUCCESS` event in the Audit Log.
6. **Routing:** The user is redirected to the appropriate Dashboard based on their role.

Security Architecture

FIXIT implements a Defense-in-Depth strategy, securing the application at multiple layers of the stack.

- **Layer 1 (Input):** Sanitization of all form inputs to prevent HTML/Script injection.
- **Layer 2 (Identity):** Strict OAuth 2.0 enforcement and Admin password hashing **(SHA-256).**
- **Layer 3 (Access Control):** RBAC enforcement at the route level; credential stuffing protection via account lockouts (3 failed attempts = 15 min lock).
- **Layer 4 (Transport):** All data in transit is encrypted via HTTPS/TLS.
- **Layer 5 (Audit):** Comprehensive logging of all system modifications to Firestore and CSV backups.

Deployment Architecture

The system is designed for cloud deployment to ensure high availability and data redundancy.

- **Production Environment: Hosted on Google Cloud Platform (GCP).**
    - Compute: Cloud Run (or similar container service) hosts the Flet Python application.
    - Database: Firestore provides a serverless, auto-scaling NoSQL database.

    - CDN: Google Cloud CDN ensures fast load times for static assets.
- **Development Environment:**
    - Runs locally on Python 3.x using SQLite/JSON fallback for rapid iteration without incurring cloud costs.
    - Uses Firebase Emulator for testing database triggers.

Module Responsibilities

The project follows a standard industry layout to separate concerns:

- `app/views/`: Contains all UI code.
- `app/services/`: Contains logic (AI, Auth, Database).
- `storage/`: Handles local caching for offline capabilities.
- `assets/`: Stores fonts and static media.

System Architecture Diagram

Click : System Architecture Diagram and Project Structure

Threat Model & Security Controls

Key Threats

Threat	Risk Level	Implemented Control
Unauthorized Access	High	**Google OAuth 2.0:** No local password storage. Authentication is offloaded to Google Identity Services.

Privilege Escalation	High	**Role Verification:** On every page load, the system re-verifies the user's role against the Admin Allow-list (admin_account.py).
Session Hijacking	Medium	**Auto-Timeout:** The session_timeout_ui.py component forces logout after 5 minutes of inactivity.
Data Tampering	Medium	**Immutable Audit Logs:** Critical actions are written to a separate CSV stream. While not cryptographically signed, they provide a forensic trail.
SQL Injection	Medium	**ORM usage:** Interaction with the database is handled through parameterized queries/ORM methods, preventing raw SQL execution.

Design Decisions / Trade-offs

1. **Single-language stack (Python + Flet):** Chosen for speed of development and compatibility with ML components. Trade-off: may present performance or packaging challenges on some mobile devices.
2. **Multinomial Naive Bayes for categorization:** Fast, explainable, and easy to train on small datasets. Trade-off: lower accuracy compared to Transformer-based models, but much lighter and appropriate for an on-device or small-server setup.
3. **Online-First Architecture :** the "Offline Sync" requirement was removed in favor of a strictly online model. Secure authentication (Google OAuth) and real-time AI processing were prioritized. Implementing secure offline auth is complex and risky for this scope. The app cannot be used in campus "dead zones" without Wi-Fi.

4. **Local admin credential for demo use:** Simplifies grading and demo scenarios. Trade-off: must be replaced with a secure credential store for production.
5. **Offline-first UX with later sync:** Prioritizes usability for intermittent connectivity; adds complexity in conflict resolution and data integrity.
6. **Google OAuth with school-email restriction:** Gives a familiar SSO flow for users but depends on availability of Google services and requires safeguarding client secrets.
7. **CSV for Audit Logs:** Audit logs are written to flat CSV files rather than the main database. Performance and Separation of Concerns. Writing logs shouldn't slow down the main database transactions. It also makes it easier for non-technical admins to download and view logs in Excel. Less secure than a dedicated logging server; files could theoretically be deleted if the server is compromised.

Limitations & Future Work

Current Limitations

- **Model maintenance:** The Naive Bayes model is trained on a static CSV at import. It does not support safe, automated re-training from live inputs (risk of poisoning).
- **Dependency on Connectivity: T**he system is rendered unusable without an internet connection due to the reliance on Google Auth.
- **Admin account security:** Demo admin credentials are stored in code (empty password placeholder in example). This is insecure for production and intended only for development/demonstration.
- **Scalability constraints:** A simple SQLite-backed server is suitable for small-scale deployments only.
- **Device compatibility testing:** Additional QA is needed across Android/iOS devices and screen sizes.
- **Single-Tenant:** The system is currently hardcoded for a single campus domain.

Future Enhancements

- **Model improvements:** Replace or augment Naive Bayes with a small, fine-tuned Transformer or embed semantic search (embedding + kNN) for better categorization, plus a safe retraining pipeline.
- **Offline Mode (v2.0):** Implement local caching for report drafting and a background sync worker.
- **Production authentication:** Replace hard-coded admin credentials with server-managed accounts (hashed passwords), and SSO enforcement with a managed identity provider.
- **Push notifications:** Real-time updates to users when status changes (via FCM/APNs).
- **Role management:** Fine-grained RBAC for multiple admin/maintenance roles (triage staff, technicians).
- **Reporting & analytics:** Exportable reports, maintenance KPIs, SLA tracking, and workload balancing.
- **Stronger local data encryption:** Protect offline cache with device-backed encryption keys.
- **Automated testing & CI/CD:** Add unit/integration tests, pipeline for model validation, and continuous deployment.

Closing summary

FIXIT delivers a pragmatic, mobile-first maintenance reporting platform built with Flet and Python to align with the project's existing ML and OAuth components. The system balances fast development and usability (offline-first UX, Google SSO, admin dashboard) with essential security controls (role enforcement, OAuth restrictions, input validation). Future work should focus on production-hardening of authentication, ML lifecycle management, and device/scale testing to make the system suitable for institutional deployment.