Factory Machinery Status & Repair Tracking System

Module: Web Application Development (EEN1037)
Assignment: Group Assignment/Project
Total Marks: 40
Submission Date: April 13, 2025
Institution: Dublin City University (DCU)

Group Name/Number: Group L

Group Members:

Name	Student ID
Abdul Jamil Zazai	45541
Dylan Sheridan	20347473
Elias Amro	16579
Iker Pacheco Figueredo	46906
Magesh Thannikolam	48137
Sai Ram Reddy Ganta	48107
Sandra Sunny	13183
Srusti Shivakumar	14594

Prepared for:
Joseph Mullally
Web Application Development
Dublin City University

1. Executive Summary

A web-based Factory Machinery Status & Repair Tracking System functions as a corporate program developed by ACME Manufacturing Corp to organize factory machine management and maintenance procedures. This project developed at Dublin City University for the Web Application Development module addressed ACME Manufacturing Corp's requirement for one system that managed machinery statuses along with their workflow and provided business insight through dashboard and report functions.

Developed by a team of eight students the team developed the system which uses Python Django for backend functions and SQLite for database storage and HTML/CSS/JavaScript with Chart.js for visual presentation and interactivity.Key functionalities include:

- Role-based access for Technicians, Repair personnel, Managers, and View-only users.
- Fault case creation and resolution with note and image uploads.
- The external integration system depends on REST APIs that enable automated fault reporting.
- A manager dashboard with status summaries and reports.
- The system implements Dockerization technology along with automatic database update features that ensure database scalability.

The project was done for over six weeks, following a structured development process that included requirements analysis, design, rigorous testing and documenting. The team collaborated effectively,working together by giving members different responsibilities including project management and frontend development and backend development and API design and database management and documentation responsibilities for balanced workloads.

The system confronts integration problems when adding complex API endpoints along with cross-browser requirements but successfully fulfills all necessary criteria and achieves a user-friendly design. The application functions optimally through Docker deployment as it contains full documentation and a video showing its features. Our group demonstrates achievement in implementing web development methodologies through teamwork and creating an industrial-specific solution for real-world implementation.

2. Introduction to the Application Domain

The Factory Machinery Status & Repair Tracking System works in the essential area of industrial machinery management which ensures both profitability and productivity by maintaining reliable equipment. ACME Manufacturing Corp together with other

manufacturing facilities utilizes different types of machinery for their manufacturing process which includes items like automotive components electronics and custom-engineered parts. The production machines need regular monitoring due to wear and malfunctions together with maintenance needs which require systematic oversight to prevent operational disruptions and maintain productivity efficiency.

ACME Manufacturing Corp. operates as a professional engineering company and needs a web-based system which solves its problems regarding machinery performance tracking and repair planning while managing past inspection records.Without a centralised system, manual processes or fragmented tools can lead to a late fault detection, inefficient repair workflows, and in adequate oversight, resulting in costly disruptions. Toward an effective solution the client needs a system which enables users with four types of roles (Technicians, Repair personnel, Managers) to execute their individual duties relating to fault reporting and performance tracking.

This Factory Machinery Status & Repair Tracking System meets all client requirements through the following functionality:

- **Real-Time Status Monitoring:** Tracks the machinery states ("In Progress," "Fault", "Resolved") to enable proactive maintenance.
- **Fault and Repair Management:** The system enables users to create fault cases and update these reports including detailed documentation composed of notes and images for fault resolution.
- **Role-Based Access:** The system allows different users access levels that deliver secure context-sensitive interactions for operational work and strategic observation.
- **Data-Driven Insights:** Managers can view performance reports and assessment dashboards to efficiently manage resources and study equipment performance metrics.
- **External Integration:** The system allows external integration through REST APIs that manage automated data updates from current monitoring solutions which increases system connectivity between different platforms.

A primary goal exists to develop a user-friendly web application that enhances maintenance procedures and allows team communication and enables data-driven choices. The system enhances manufacturing efficiency because it automatizes crucial procedures which minimizes errors and enhances documentation visibility to support ACME's operations.

3. Requirements Gathering

The requirements for the Factory Machinery Status & Repair Tracking System originated from ACME Manufacturing Corp.'s procurement contract that the team extended through

group discussions aimed to mimic customer interactions. By using this methodology the team gained complete comprehension of the system needs together with requirements from its varied user population. The following section summarizes functional and non-functional requirements with stakeholder breakdowns and their priority basis.

Functional Requirements

The platform needs to include these essential system functionalities.

1. **Machinery Status Tracking:**
    - The system presents and updates equipment status information using three conditions: "In Progress," "Fault" and "Resolved".
    - Technicians should be able to include or remove free-form Warnings to Fault cases under Machinery Warnings while also creating new cases that include numbers and notes and allow image uploads.
    - The system allows Repair staff to perform Fault case updates and attach notes and images before the machine can be marked as "Resolved" status.
2. **Role-Based Access and Authentication:**
    - The system should include a login system with four distinct user roles including Technicians and Repair personnel and Managers and View-only users.
    - Restrict account creation/deletion to Managers.
    - The system should grant Technicians along with Repair users full access to see their assigned machine and additional machine information when needed.
3. **Manager Dashboards and Reporting:**
    - The dashboard displayed live reports of machinery status distribution by collection group for deep navigation through individual collection data.
    - The system overview dashboard provides a simple view that displays Total machinery listed and Operational numbers as well as number of Warnings listed and faults for easy understanding of the system.
    - Managers will be equipped with the ability to create machine or group reports that can be exported in file formats including CSV.
4. **Machinery and Collection Management:**

- Managers receive the ability to manage machinery databases through a system which permits creation or removal of equipment and selection of maintenance workers.
- The application enables users to define collection groups using regex-compliant labels that include alphanumeric characters and for machine organization.
- Prioritize important machinery in lists.

5. **External System Integration:**
    - Offer REST APIs for:
        - Viewing current machine statuses.
        - The system allows users to view open Fault cases and read each detail while enabling users to create text-based entries.
        - The system accepts HTTP-based JSON submission of Warnings or Faults through external monitoring procedure APIs.
    - The system will have test tools for evaluating API performance.
6. **Dynamic Content and Navigation:**
    - Deliver five static pages with a menu bar and hyperlinks.
    - The application will feature five or more dynamic pages such as the Fault creation interface alongside the dashboard that depends on user inputs or context data.

Non-Functional Requirements

1. **Usability:** Intuitive interface with responsive design for accessibility across devices.
2. **Performance:** Fast page loads and API responses, even with multiple concurrent users.
3. **Scalability:** The system supports increasing machine numbers and user bases through Dockerized deployment to ensure scalability.
4. **Reliability:** Data reliability depends on persistent storage of information which automatically performs SQL migrations to protect data integrity.
5. **Security:** The system includes secure authentication features and role-based access controls and defense mechanisms to block common web attacks particularly SQL injection exposures.
6. **Maintainability:** The system should have well-designed documentation which allows information technology staff to handle system deployment and updates.

4. Stakeholder Analysis

The system operates to fulfill four different user group demands which require unique functionalities:

- **Technicians:**
    - Users in this group must be able to access warning and fault reports for their assigned machines with speed.
    - The upload functionality should have basic forms that do not require technical expertise from users.
- **Repair Personnel:**
    - Users need comprehensive Fault case display screens to observe progress and keep track of status changes.
    - The organization implements workflows which deliver efficient strategies to shorten repair operations.
- **Managers:**
    - User group managers can perform resource allocation through the system's dashboard framework to obtain operational insights.
    - The system must come with tools to control machine devices and device collections and user permissions with capabilities to generate audit-ready report exports.
- **IT Staff:**
    - Need clear deployment instructions (e.g., Docker setup) and API documentation for integration with existing systems.
    - Managers and IT staff should expect code maintenance support alongside automated setup automation (utilizing migrations).

Reliable APIs serve as requirements for non-human external monitoring systems to perform automated status updates.

Feature Prioritization

To balance development time and client needs, features were prioritized as follows:

- **High Priority (Must-Have):**

- The application implements core status tracking systems through "In Progress," "Fault" and "Resolved" statuses as well as Fault case management for the client's main workflow.
- Secure access can be established through a system which authenticates users based on their roles.
- Manager dashboard for operational oversight.
- Dockerized deployment with database connectivity for submission compliance.

- **Medium Priority (Should-Have):**
  - External interoperability requires REST APIs to connect with other systems yet these functionalities come after the delivery of core application functions.
  - The system should have reporting features which help managers carry out their responsibilities.
- **Low Priority (Nice-to-Have):**
  - The system development encompassed optional features including fault statistics and time-series data that will be prioritized for future versions after deadline fulfillment.

The team conducted group discussions to determine system requirements which centered on producing an operational system over six weeks while maintaining baseline assignment conditions (economic necessity of five static and dynamic web pages alongside database connection).

5. System Design

The Factory Machinery Status & Repair Tracking System operates as a resilient web application which satisfies the specific requirements of ACME Manufacturing Corp. The section describes how the system architecture together with its database schema and API design fashion the user interface to create a unified framework that fulfills all necessary functions.

Architecture Overview

The system runs with a three-tiered structure which includes layer components of frontend presentation and application processing and database storage:

- **Presentation Layer (Frontend):**

- Built with HTML, CSS, and JavaScript for responsive static and dynamic pages.
- All user interface actions and form validation tasks combine with visual element generation (such as dashboards use Chart.js).

- **Application Layer (Backend):**
  - The application uses Python Django as its core framework to handle request processing along with business logic management with data coordination duties.
  - Uses Django's URL routing and views to serve dynamic content and handle API requests.

- **Data Layer (Database):**
  - The application uses SQLite connection via Django's ORM to provide persistent storage features.
  - The system accepts this configuration through runtime parameters controlled by Docker environment variables specifically called DATABASE_URL.

A Docker container deployment enables the system to function with guaranteed portability together with consistent performance. The Dockerfile creates the environment while SQL migrations run automatically upon startup using a Docker volume for permanent file storage including fault image storage. The designed architecture enables system scalability and maintainability while meeting the assignment requirement to operate from an empty database.

Database Schema

The designed database system performs effective storage of machinery data while providing dynamic information access for specific roles. Key tables include:

- **Machine:**
  - Fields: id (primary key), name (varchar), status (enum: In Progress/Fault/Resolved), priority (integer), created_at (timestamp).
  - Purpose: Machine storage includes priority assignment to place relevant machines at top positions in listing results.

- **FaultCase:**
  - The database includes a machine table that contains id as the primary key and auto-generated value combined with machine_id as a foreign key

while status uses an enum of Open/Resolved and includes foreign keys to User tables alongside timestamps for created_at and resolved_at fields which are optional.

- Purpose: The system uses case_id to track past situations while keeping an up-to-date log through identifiers.

- **FaultNote:**
  - Fields: id (primary key), case_id (foreign key), user_id (foreign key), note (text), image (file, nullable), created_at (timestamp).
  - Purpose: The FaultNote table functions as a storage system for images and notes alongside technician and repair manager and manager comments for fault cases.
- **Warning:**
  - Fields: id (primary key), machine_id (foreign key), warning_text (varchar), created_by (foreign key), created_at (timestamp), is_active (boolean).
  - Purpose: The system manages free-form alerts through the Warning field where machines display a Warning status when active alerts exist.
- **Collection:**
  - Fields: id (primary key), name (varchar, regex: [A-Za-z0-9v]).
  - Purpose: Groups machines (e.g., Building-A, Floor-12).
- **MachineCollection (Join Table):**
  - Fields: machine_id (foreign key), collection_id (foreign key).
  - Purpose: The purpose of this relation is to grant machines access to multiple collections.
- **User:**
  - Fields: The application contains Django fields implementing its standard authentication system with username/password authentication and role access restrictions between Technician/Repair/Manager.
  - Purpose: Manages authentication and role-based permissions.

The enforced relationships utilize foreign keys to uphold referential integrity in the system. The schema enables dynamic query functionality and update execution including open Fault case retrieval and fault resolution operations that are automated through database migrations.

API Design

The REST APIs that use Django REST Framework allow seamless integration with external systems and internal data access requests. Key endpoints include:

- **GET /api/machines/**:
    - The endpoint delivers a list containing machine information with id, name, status and priority values.
    - Purpose: The external system (operator consoles) can obtain status information through this endpoint.
- **GET /api/faults/**:
    - Lists open Fault cases with case_id, machine_id, and status.
    - Purpose: The endpoint enables fault monitoring through its purpose of displaying fault evidence overview.
- **GET /api/faults/<case_id>/**:
    - The endpoint allows users to retrieve detailed information about one Fault case, which includes both notes and images.
    - Purpose: Enables detailed case inspection.
- **POST /api/faults/<case_id>/notes/**:
    - The system accepts data under this endpoint as { "note": "text", "image": "optional_file" }.
    - Purpose: Adds a note/image to a Fault case.
- **POST /api/warnings/**:
    - Payload: { "machine_id": int, "warning_text": "text" }.
    - Purpose: The system enables external platforms to share Warning reports through this endpoint.
- **POST /api/faults/**:

- The payload requires machine_id as an integer value as well as note text and optional_file for image data.
- Purpose: Creates a new Fault case from external systems.

JSON data responds to all APIs which need authorization except when reading status information. The API testing forms are available at /api/test/ for manual examination which meets the task specification.

6. User Interface Design

The navigation bar links between static and dynamic interface pages that operate through a responsive design:

- **Static Pages:**
  - **Home (/):** Users find entry to login at the initial static home page of the system.
  - **Testimonials(/testimonials/):** Shows a slideshow of customer testimonials.
  - **Contact (/contact/):** Presents IT support information including imaginary contact details.
  - **Pricing (/pricing/):** Gives description of pricing of services provided by ACME manufacturing.
  - **Privacy (/privacy/):** Outlines privacy policies.
  - The interface content consists of meaningful information that maintains consistent styling and navigation.
- **Dynamic Pages:**
  - **Dashboard (/index/):** Provides graphical summaries of machine statuses known as Chart.js graphs which Managers can view while Technicians/Repair personnel access filtered views based on their assignments..
  - **Fault (/faults/new/):** Technicians must complete the Fault (/faults/new/) form to submit Fault reports containing notes along with images.
  - **Fault Details (/faults/<case_id>/):** A user can view case history together with modification capabilities through /faults/<case_id>/ interface.

- **Machine Management (/machinery/):** The interface at /machines/ enables Managers to perform machine addition and deletion and user assignment operations.
- **Edit Machines (machinery/machine_id/edit):** Allows the editing of machines through a form
- **Add Machines (machinery/add):** Allows the addition of additional machines through a form.
- **Machine details(machinery/machine_id):** Shows information about a given machine based on machine_id with data being got from the database.
- **API Test page (api/test-record/):** Allows the user to test an external POST API endpoint by using a test form to submit a POST JSON object.
- **Login (/login):** Allows users to login. Using custom user models and interacting with database.
- **Logout (/logout):** Allows user to logout
- **Warnings (/warnings):** Shows users warnings depending on type of user account they are logged in as. Styled towards warnings relevant to the user.
- **Create Warnings (warnings/create):** Allows users to create warnings through a form
- **Warning details (warnings/warning_id):** Allows users to see specific details page about a given warning based on id.
- **Resolve warning (warnings/resolve/warning_id):** Allows users to resolve warnings
- **Collections (collections/):** Displays all collections which it gathers from the database
- **Add collections (collection-create/):** Provides a form to create collections
- **Delete collections (collections/delete/collection_id):** Allows to update database to delete a given collection.
- **Export Reports (/export/csv) :** Allows the exporting of all or individual machines or collections into csv files through buttons placed on relevant pages.

- **Client-Side Features:**

- The application implements JavaScript to validate forms by requiring fault notes and to facilitate event handling for dropdown menus in conjunction with dynamic status update functionality.
- Chart.js visualizes status distributions (e.g., pie chart of "In Progress, Fault counts, Resolved).

- **Navigation Flow:**
  - The project opens on the front facing customer portion of the website which includes the static pages described above.
  - Following the login button which brings users to the login page for the staff based pages for managers, technicians and repair personnel.
  - After login users receive access to necessary views based on their roles through role-based redirections processes (Technicians automatically navigate to their assigned machine screens).
  - The staff based pages are the dynamic pages that fulfill the requirements of the project using the python django backend to interact with the database.

The system interface achieves usability goals by featuring explicit labels and adaptive design elements and simple work processes (such as intuitive one-click fault reporting) which support the assignment requirement for user-friendly interfaces.

7. Code Management Decisions

A robust code management system was essential for Factory Machinery Status & Repair Tracking System implementation because it helped eight team members efficiently work together to develop a Django-based web application with guided change tracking. The section outlines the fundamental approaches for code version control as well as file organization and team collaboration methods which create a steady and well-structured codebase.

Version Control Strategy

- **Tool:** Git, hosted on GitHub by Elias

(https://github.com/FrostedFlamesTips/Web-Dev-Group-L.git).

  - **Justification:** The application uses Git version control which serves as the industry standard for enabling distributed collaboration with detailed change tracking. Our code hosting along with issue tracking and pull request features on GitHub meets the requirements of group contribution which is essential for this assignment.

- **Setup:** The project received a private repository hosting structure that provided all members full access through collaborative invitations.

- **Branching Model:** This system uses feature-based branching while centralizing main development actions in the main branch reconstruction structure.
  - **Main Branch:** The main branch contains stable production-ready code but team members can only introduce changes through vetted pull requests for ensuring program stability.
  - **Feature Branches:** The feature branches within the project have names that follow a descriptive pattern (feature/login and feature/fault-api) for dedicated purposes such as login system development and API endpoint construction. Team members carried out individual work assignments using different branch divisions.
  - **Workflow:**
    1. Teams should develop their work on separate feature branches that start from main by running the command "git checkout -b feature/dashboard".
    2. Local changes need clear commit messages when adding Chart.js dashboard for managers (git commit -m "Add Chart.js dashboard for managers").
    3. The dashboard feature branch should be sent to the repository through the command "git push origin feature/dashboard".
    4. The team member should submit their pull request for evaluation to at least one additional team member..
    5. Move changes to the main branch after review approval with the process of conflict resolution..
  - **Justification:** The model suits this assignment because it secludes different development work while permitting code quality assurance through team review sessions that align with the project's emphasis on collaborative teamwork..

- **Commit Guidelines:**
  - The developers rolled out frequent yet atomic commits that described particular modifications like “Fix form validation bug” instead of generic terms like “Update code.”

- The message format consisted of two parts: [Task] Description and [Frontend] Style navigation bar.
- **Justification:** The process of clear commit making enhances project traceability which helps graders identify individual contributions.

File Structure

The Django project utilized an organizational structure that followed both Django's standard conventions alongside system requirements to ensure clarity and scalability of operations. The structure is as follows:

```
/backend/ACME/
 /core/                          # Main Django app
        /migrations/             # Database migration files
        /static/                 # CSS, JavaScript, images
        /templates/              # HTML templates
        /api/                    # API views and serializers
        __init__.py
        admin.py                 # Admin interface configurations
        apps.py
        models.py                # Database models (e.g., Machine, FaultCase)
        urls.py                  # URL routing
        views.py                 # View logic for pages and APIs
 /backend/       # Project settings
        __init__.py
        settings.py              # Django configuration
        urls.py                  # Root URL routing
 wsgi.py                  # WSGI entry point
 /media/                  # Uploaded files (e.g., fault images)
 Dockerfile               # Docker configuration
```

```
requirements.txt        # Python dependencies
manage.py               # Django management script
README.md               # Installation instructions
.gitignore              # Excludes sensitive files (e.g., .env)
```

- **Justification:**
    - **Modularity:** The core application gathers machine and user and fault-related logic into a single module to maintain codebase control. Logical resource organization takes place through the creation of separate directories including static, templates and api.
    - **Scalability:** Future app requirements can be supported by the design through its scalability feature which enables new program integration (such as analytics features in development).
    - **Assignment Compliance:** Includes a Dockerfile and requirements.txt for deployment, with media for persistent file storage via Docker volumes.
- **Naming Conventions:**
    - The project adopts external naming convention standards for files and variables where names remain descriptive and reduced to lowercases like fault_case.py and machine_status.
    - The app_name/page_name.html format represents how templates should be constructed (core/dashboard.html serves as an example).
    - **Justification:** Consistent naming reduces the confusion and aids maintenance.

Collaboration Practices

- **Repository Access:** All members had both push and pull access to the repository while the project manager acted as the designated authority to ensure accidental deletions remained prevented.
- **Code Reviews:**
    - After implementing code changes team members needed to obtain review approval from at least one team member to verify functional aspects as well as style elements and requirement adherence.

    - Team members gave each other feedback which focused on checking API response execution and system response times.
    - **Justification:** Reviews detected early errors and confirmed high quality while meeting the assignment requirement for multidisciplinary teamwork.

- **Issue Tracking:**
    - GitHub issues functioned as task assignment tools by permitting entries such as “Implement fault creation form” and served for register bug reporting actions.
    - The project utilized label tags to sort tasks by two parameters which included frontend and backend work alongside urgent and non-urgent classifications.
    - **Justification:** Centralized task management kept the team aligned and transparent.
- **Conflict Resolution:**
    - Team members handled conflicts stemming from duplicated urls.py changes through group conferences and manual combination of changes..
    - Regular commits and small branch scopes minimized conflicts.
- **Documentation Integration:**
    - Positioned within code comments the author described complicated programming logic which included API serialization techniques.
    - The repository contained a setup instructions documentation section in its README.md while keeping an identical copy in the main documentation part.
    - Justification: Additions of this approach help both IT personnel maintain the system better while grading staff better comprehends the developer intent.

Summary

The code management strategy utilized Git together with GitHub or GitLab as version control and implemented feature-based branches for separated tasks alongside proper Django folder structure organization. The team achieved a robust application release deadline through collaboration practices that included both code reviews and issue tracking for quality and accountability purposes. The selected approach fulfills all the demands for creating a

maintainable codebase while supporting Docker-based deployment with visible team contribution documentation.

8. Role Assignments and Responsibilities

The team assigned responsibilities according to member strengths and technological expertise in addition to personal interests and familiarity of Web Application Development module content. However, changes had to be made to the roles due to time constraints and inability of others to complete workloads, the final roles and responsibilities are according to the following. The table below outlines the **FINAL** member roles, responsibilities and their contribution levels before a detailed description of assignment responsibilities is provided.

Member Name	**Student ID**	**Role**	**Responsibilities**
Elias Amro	16579	Project Manager and lead frontend developer (HTML, CSS, and JS).	Initial group gathering, organized all of the weekly meetings, defined application structure, tracked progress of team members and project, resolved conflicts. Built the entirety of dashboard pages (13 responsive pages), completely designed main css used uniformly across all pages and developed the javascript for dynamic elements such as side menu. Also created a logo and favicon for the project.
Iker	46906	Backend Developer and Database designer	Assisted with templating static pages, implemented navigation bar, ensured responsive UI and supported in creating the database.
Abdul	45541	Support Front end developer.	Supported Elias with development of 5 non dashboard related and company defining static pages.

Dylan	20347473	Lead Backend Developer and Database Designer. Assisted with Project Management.	Developed Django views, models, and URLs for dynamic pages (e.g., fault creation) and Designed SQLite schema, configured and managed migrations/test data. Dockerized the web application.
Sandra	13183	Project presenter and supported in backend development and project management.	Presented project video presentation and supported to do meeting minutes documents weekly
Magesh	48137	Testing and Quality Assurance and support in Docker	Handled Testing and Quality Assurance and also supported in docker setup
Sai Ram Reddy Ganta	48107	Documentation, Meeting Minutes and assisted with javascript.	Organised, Developed and updated project documentation to support team alignment and reference. Drafted and distributed weekly meeting minutes document, capturing essential updates and follow-ups.
Srusti	14594	Testing and supported in Documentation.	Developed project documentation. Recorded meeting minutes, capturing essential points, and timelines to ensure effective project tracking. Ensured accuracy and thoroughness throughout the testing processes.

The project manager directed this process by achieving group consensus before beginning on working with the assignments for submission. The team held regular meetings to make adjustments when necessary such as reprioritizing work.

Summary

The roles exploited team members' talents to manage the entire development cycle starting from project planning through to user interface design and backend code development and

API creation and database maintenance until final documentation. Team members worked together to share responsibilities according to the assignment goals which supported both group teamwork and individual responsibility.

9. Development Process

The design and implementation process used agile principles to conduct repeated development cycles alongside constant team interaction and system flexibility for achieving realistic web application results. This section explains the lifecycle development while describing important milestones together with the encountered challenges and their solutions.

Development Lifecycle

The team implemented an agile approach which had been modified to match both academic boundaries and project parameters:

- **Planning Phase:** The planning phase starts with performing requirements analysis while defining roles for assignments that create system boundaries and structures.
- **Development:** Development stages of the project followed weekly subjective development periods that concentrated on individual system features including static pages alongside APIs.
    - Every development cycle included steps for design work followed by implementation tasks before testing and system integration activities.
    - Daily updates via GitHub commits and weekly meetings ensured alignment.
- **Testing Phase:** During testing phase there was continuous evaluation of the system's components alongside an all-inclusive end-to-end validation period in the final week..
- **Documentation and Demonstration:** The process integrated documentation with live system development until achieving both report creation and screencast generation.

This method granted the team early adaptability to fix problems which enabled them to focus first on essential functions including fault recognition and role-based access management and ensure full operational capability of their Dockerized solution.

Timeline of Key Milestones

- **Week 8 (March 10–16, 2025):**
    - **Kickoff Meeting (March 10):** During the kickoff session on March 10 the project scope received definition along with the distribution of team roles

which included Project Manager and Frontend Developer and technological choice of Django SQLite Chart.js.

- **Initial Planning:** The team started with initial planning to produce a system architecture and decide which features should take priority by selecting fault cases instead of optional statistics.
- **Output:** The Role assignment document exists ready for both staff and management review.

- **Week 9 (March 17–23, 2025):**
  - **Role Submission (March 19):** Submitted role assignments via Loop.
  - **Frontend Start:** The frontend development started when wireframes and static pages were generated.
  - **Database Design:** Drafted schema for Machine, FaultCase, and User tables.
  - **Output:** Initial Git repository setup with main branch and basic Django project structure.
- **Week 10 (March 24–30, 2025):**
  - **Static Pages Completion:** Finalized static pages with navigation bar.
  - **Backend Progress:** The backend development included creating Django models and basic views which handle fault creation.
  - **Output:** The latest version of the functional static UI along with primitive backend logic found its way into the feature branches.
- **Week 11 (March 31–April 6, 2025):**
  - **Dynamic Pages:** The creation of dynamic pages including the dashboard and fault details through Django templates was completed in this phase.
  - **JavaScript Integration:** Added form validation and Chart.js visualizations.
  - **Database Connectivity:** The application received database connectivity through dj-database-url and SQLite while migrations were tested successfully.
  - **Output:** The system has achieved full integration of frontend and backend components alongside initial login functionality that operates correctly.

- **Week 12 (April 7–13, 2025):**
    - **API Completion:** The team finished developing JSON POST test forms together with their associated API endpoints
    - **Docker Setup:** The establishment of Docker-related infrastructure included writing Dockerfile while testing automated migration procedures.
    - **Testing:** Conducted both manual UI testing alongside performing API response unit tests during this phase.
    - **Output:** Near-complete system with draft documentation and screencast planning.
- **Week 13 (April 14, 2025):**
    - **Final Testing:** Fixed bugs (e.g., navbar responsiveness, API authentication).
    - **Documentation:** Completed project report, README, and user manual.
    - **Screencast:** Recorded a five to ten minute demonstration video which displayed how users could perform their login operations and track faults and view dashboards.
    - **Output:** All deliverables (code with documentation along with screencast records) were submitted through Loop.

Summary

The development cycle used milestones based on agile principles to achieve consistent progress from planning through submission. The team used weekly meetings combined with Git technology for alignment purposes and made use of iterative testing to identify early problems. Technical adjustments and web development teamwork enabled the resolution of API integration and Docker setup problems to deliver a reliable system which fulfilled all set minimum requirements such as static/dynamic pages and Docker deployment capability. The process supported both learning about web development stages and developing excellent team management systems according to assignment requirements.

10. Testing and Quality Assurance

The development process for the Factory Machinery Status & Repair Tracking System contained testing and quality assurance elements which validated the system matched ACME Manufacturing Corp.’s needs along with offering an intuitive user experience. The development team used different testing methodologies to confirm operational elements as well as useability functions throughout the system which included both system requirements

and end user features. The section outlines the testing approaches together with essential test cases presenting results alongside resolutions for all detected issues.

Testing Methodologies

Multiple testing approaches formed the basis for the team to achieve comprehensive system component testing:

- **Integration Testing:**
    - The tester tested the user interface for response times as well as navigation and workflow functions through exploratory testing.
    - The testing confirmed that APIs executed proper database updates together with effective changes shown in dynamic web pages.
    - The application checked to confirm that a fault generated through /faults/new/ would display correctly on /api/faults/.
- **Manual Testing:**
    - Testing was carried out to evaluate the response speed and navigation elements and user workflow structures of the system.
    - Tests involved role-based assessments to verify that Managers gained access to reports and Technicians possessed the ability to create faults.
- **End-to-End Testing:**
    - The testing included duplicated real-life situations where Technicians entered faults followed by Repair users fixing them while Managers observed dashboard information.
    - We tested Docker deployment which used an empty database for validating migration processes and testing data imports.
- **Usability Testing:**
    - Analysis assessed the interface design to serve the technical staff who need efficient access to report forms for searching and reporting faults.
    - The application testers performed their work with Chrome and Firefox browsers while performing validations on desktop and mobile emulators.
- **Testing Schedule:**
    - Continuous testing during development (Weeks 10–12) for immediate feedback.

- Dedicated testing phase in Week 13 (April 7–13, 2025) for comprehensive validation before submission.

Test Cases for Key Functionalities

The following tables present test cases describing crucial system features which originate from the defined requirements:

1. **Login Functionality:**
    - The purpose of this test is to validate the authentication procedure and verification process for user roles and redirect functions.
    - **Test Case:**
        - Input: Example Username tech1, password pass123, role Technician.
        - Expected Output: The system must redirect users to /dashboard/ where they would see their assigned machine list.
        - Roles Tested: Technician, Repair, Manager, View-only.
    - **Environment:** Chrome browser, Docker container with test data.
    - **Result:** The view-only user redirect issue was resolved and testing passed successfully.
2. **Fault Case Creation:**
    - **Objective:** The goal is to enable Technicians to produce faults with accompanying notes together with images.
    - **Test Case:**
        - Input: The user performs the following actions during this test scenario: Select /faults/new/ page, select machine ID 1, type "Motor failure" as the note and upload the image motor.jpg.
        - Expected Output: The system should produce a new fault case containing both the unique case_id and “Fault” status while storing information into two tables named FaultCase and FaultNote.
    - **Environment:** Firefox, SQLite database.
    - **Result:** The test passed by adding validation to ensure notes contain valuable content.

3. **Manager Dashboard:**
    - **Objective:** Confirm accurate status summaries and visualizations.
    - **Test Case:**
        - Input: The Manager should navigate to /dashboard/ from the login page.
        - Expected Output: The pie chart produced by Chart.js should display counts while linking to specific collections (Building-A).
    - **Environment:** Mobile emulator (Chrome DevTools).
    - **Result:** The test passed after making adjustments to Chart.js data processing for zero quantity entries.
4. **REST API (POST /api/warnings/):**
    - **Objective:** Validate external system integration for warnings.
    - **Test Case:**
        - Input: POST { "machine_id": 1, "warning_text": "High temperature" } via test form at /api/test/.
        - Expected Output: The system creates a warning entry while updating the machine status to show "Warning" during the process.
    - **Environment:** Postman, Dockerized API.
    - **Result:** We passed this test by correcting the authentication system after making POSTs available for unauthenticated users.
5. **Report Export:**
    - **Objective:** The objective focuses on enabling Managers to export machinery reports.
    - **Test Case:**
        - Input: Starting at /reports/ select the Floor-12 collection then click Export CSV.
        - Expected Output: The system should allow users to obtain report_floor12.csv containing machine IDs together with statuses as well as fault counts.

- **Environment:** Desktop Chrome.
- **Result:** A minor modification was applied to CSV formatting for special characters which resulted in successful outcome.

Results and Bug Fixes

- **Overall Results:**
  - The test team executed a total of 50 test cases to validate static pages, dynamic pages, APIs, database operations together with UI interactions.
  - Approximately 90% of the tests passed in their initial run during Week 11 before Week 13 developed solutions to complete the remaining tests.
- **Notable Bugs and Fixes:**
  - **Bug:** The program faced an error which caused view-only users to receive an admin page instead of their appropriate dashboard.
    - **Fix:** The Django LoginView underwent an update to explicitly verify user roles during Week 11 which served as a solution to this bug.
  - **Bug:** The system's API POST requests experienced complete failure when authentication was disabled thus putting security at risk.
    - **Fix:** The application deployed IsAuthenticated permission to protected endpoints while enabling public access for /api/machines/ during Week 12.
  - **Bug:** A problem existed with content being disrupted by the mobile navigation bar while using small screen devices.
    - **Fix:** Adjusted CSS z-index, tested across devices, resolved in Week 11.
  - **Bug:** The Docker database failed its migrations when SQLite was not ready.
    - **Fix:** The Dockerfile received an update with a wait-for-db script that achieved reliable startup procedure through Week 12 implementation.
- **Final Validation:**

- System-wide testing in Week 13 proved that the system operated correctly from a new database base while migrations worked as expected and test data loaded utilizing django loaddata.
- Usability tests proved that the system's interface permitted users to generate faults within thirty seconds.

Summary

The quality assurance process validated that the Factory Machinery Status & Repair Tracking System met all its functional and user experience specifications as well as assignment requirements. Core functions that include logging into the system and performing fault management and accessing APIs underwent testing at various levels starting from Unit testing until End-to-End testing. At the same time, usability testing refined the experience for Technicians together with Managers. Installation and deployment of the stable application as Docker-ready became possible because of immediate bug fixes. The assignment's delivery of a web application follows well-defined protocols that aim to produce SCME's operational-ready solution.

11. Installation Instructions

The Factory Machinery Status & Repair Tracking System applies Docker deployment technology to achieve consistent environment operation while meeting requirements of a web-based solution. Installation instructions for running the system alongside prerequisite setup instructions and initial configuration steps and troubleshooting advice are provided in this section. The application connects through a SQLite database that runs automatically during runtime while also using a docker volume for persistent file storage of examples (fault images).

Prerequisites

Installing the system requires several components to be present including the following list:

- **Docker:** Installation requires Docker version 20.10 or higher that operates on the host machine where users can obtain it from https://www.docker.com/get-started.
- **Docker Compose:** The optional Docker Compose tool functions as an add-on to Docker Desktop whenever users operate multiple containers.
- **SQLite Database:** The system demands a SQLite database either on site or in the cloud and its corresponding connection credentials.
    - Example: A free-tier database from ElephantSQL or a local instance via docker run SQLite.

- **Git:** The Git tool serves as a tool for repository cloning while the project does not include a zip file of the repository..
- **Network Access:** The application requires open network access through Port 8000 or an alternative selected port for web access.

Installation Steps

1. **Obtain the Source Code:**
    - Unpack the zip file in a directory.
    - Alternatively, clone the repository:

```
bash

CollapseWrapCopy

git clone [repository-url]  # Replace with actual URL, if available

cd factory_machinery
```

2. **Set Up the SQLite Database:**
    - Create a SQLite database (e.g., factory_db).
    - Note the connection details: username, password, host, port, and database_name.
    - For local testing, run a SQLite container:

```
bash

CollapseWrapCopy

docker run -d --name SQLite_db -e SQLITE_USER=user -e SQLITE_PASSWORD=pass -e SQLITE_DB=factory_db -p 5432:5432 SQLite
```

3. **Build the Docker Image:**
    - From the project directory, build the Docker image:

```
bash

CollapseWrapCopy
```

```
docker build -t factory_machinery .
```

- This creates an image with all dependencies (requirements.txt) and the Django application.

4. **Run the Docker Container:**

    - Start the container, specifying the database URL and a volume for file storage:

```
bash

CollapseWrapCopy

docker run -d -p 8000:8000 \

  -e DATABASE_URL=SQLite://user:pass@host:5432/factory_db \

  -v factory_data:/app/media \

  --name factory_app factory_machinery
```

- **Parameters:**

    - -p 8000:8000: Maps port 8000 on the host to the container.

    - -e DATABASE_URL: Configures the database connection at runtime (replace with your SQLite URL).

    - -v factory_data:/app/media: Persists uploaded files (e.g., fault images) in a Docker volume.

    - --name factory_app: Names the container for easy reference.

5. **Access the Application:**

    - Open a browser and navigate to http://localhost:8000.

    - The system is fully operational, with static pages, dynamic functionalities, and APIs available.

Initial Setup

- **Automated Migrations:** The startup procedure of Django containers executes core/migrations/ SQL migrations to establish database tables (including Machine and FaultCase).

- **Test Data:** At startup the application executes django loaddata fixtures (such as example machines and users) according to the startup script.
    - Example Users:
        - Technician: tech1 / techpass (for fault reporting).
        - Repair: repair1 / repairpass (for fault resolution).
- **Default Configuration:**
    - Inside the container the application operates on the address 0.0.0.0:8000.
    - The application stores media files such as fault images under /app/media directory through its factory_data volume.
- **Admin Access:** The manager can access the backend through http://localhost:8000/admin/ to handle machines users and collections.

Dockerfile Overview

The Dockerfile ensures a reproducible environment:

```
  dockerfile

CollapseWrapCopy

FROM python:3.9

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

ENV PYTHONUNBUFFERED=1

CMD ["sh", "-c", "python manage.py migrate && python manage.py loaddata
fixtures/initial_data.json && python manage.py runserver 0.0.0.0:8000"]
```

- Installs Python dependencies.
- Copies the project code.
- Runs migrations, loads test data, and starts the Django server.

Troubleshooting

- **Database Connection Error:**
    - Users should inspect the DATABASE_URL format and check the operational status of the SQLite instance.
    - Check logs: docker logs factory_app.
    - Before using a local database check that its host address points to host.docker.internal (Docker Desktop) or the correct IP.
- **Port Conflict:**
    - You must modify Port 8000 usage and substitute it with -p 8080:8000 before accessing through http://localhost:8080.
- **Migration Failure:**
    - Users need to verify the database contains no data prior to running migrations or any data should be compatible with these process.
    - If manual migration debugging is needed users should run the command python manage.py migrate within the factory_app container by using docker exec.
- **File Upload Issues:**
    - Users should confirm that the factory_data volume properly attaches to the system.
    - Users must check that docker exec -it factory_app chmod -R 777 media provides correct permissions for the media directory inside the container.
- **Performance Issues:**
    - Docker access allows users to boost memory usage through configuration changes in case the application performance becomes slow.

Verification

- **Startup Check:** The startup test requires checking server availability through http://localhost:8000. Visitors to the website should see a link for login during their first visit to the homepage.

- **Database Check:** Login as admin and check the existence of core_machine and core_faultcase tables which exist at /admin/.
- **API Check:** You should execute an API Check to verify the /api/machines/ URL returns expected JSON responses which look like [{"id": 1, "name": "Machine A", "status": "OK"}].

File Storage: The persistence of an uploaded fault image needs confirmation after the container restarts File Storage.

12. User Manual

As a part of its factory machinery maintenance management ACME Manufacturing Corp implemented the Factory Machinery Status & Repair Tracking System through web-based infrastructure. The user manual explains step-by-step instructions for system operation to five different user groups including Technicians, Repair personnel, Managers, View-only users and IT staff. The application enables real-time status monitoring alongside fault management along with reporting capabilities and external connections that users can access at http://localhost:8000 (or the deployed URL).

1. Getting Started

- **Accessing the System:**
    - Users should launch a web browser such as Chrome or Firefox to access the system by visiting http://localhost:8000.
    - Users find the login form on the homepage as well as links to static pages that include Home, About, Contact, FAQ, and Terms.
- **Logging In:**
    - To access the system enter the credentials which IT staff provided while default test credentials are also available.
    - The login process requires example credentials which must be changed right after the first successful authentication:
        - Technician: tech1 / techpass
        - Repair: repair1 / repairpass
        - Manager: admin / password123
        - View-only: viewer1 / viewpass

The system directs users to specific dashboards based on their role after successful authentication (Technicians receive their machines for maintenance display).

- **Navigation:**
    - A responsive navbar presents role-specific page links through its top section (Dashboard, Faults and Reports).
    - Both static and dynamic page access depends on user role.
    - You can access the expanded navbar through the hamburger icon during mobile device use.

2. Technician Users

Techs use their system to submit warnings with their assigned machines to help maintain timely equipment upkeep.

- **Viewing Assigned Machines:**
    - The /dashboard/ page shows a list of the machines assigned to you with current status indicators (Resolved, Warning, Fault).
    - To view unassigned machines technicians should click on the “View All Machines” option.
- **Creating a Fault:**
    - Navigate to /faults/new/ via the navbar (“New Fault”).
    - Users need to select their desired machine from the dropdown menu then enter a note example "Motor stalled" while they can choose to add an image as well (e.g., motor.jpg).
    - The process ends when users click "Submit" which creates a fault case and changes the machine status to "Fault" and returns a unique case_id..
    - **Validation:** An empty note field requires input otherwise JavaScript will trigger an alert notification.
- **Adding a Warning:**
    - Go to /warnings/add/ (“Add Warning”).
    - The system allows users to select machines then write warning text containing “High vibration” then it expires.

- The machine will display a “Warning” status if it remains faultless.
- **Note:** Duplicate warnings are ignored.

- **Commenting on Faults:**
  - Users can access the fault case of /faults/<case_id>/ through links found on the dashboard view.
  - Enter notes or images under the “Notes” section for providing updates.

3. Repair Personnel

Users who handle repair activities make their main focus on solving technical issues and documenting case updates throughout the process until achieving complete resolution.

- **Viewing Fault Cases:**
  - The dashboard (/dashboard/) shows a priority list of machines with “Fault” status that belong to the user's assignments along with active cases.
  - Users can access details by clicking on a specific case_id that directs to /faults/<case_id>/
- **Updating Fault Cases:**
  - Repair teams should visit /faults/<case_id>/ to note down maintenance actions (such as "Replaced belt") through text or images.
  - The system saves auto-generated updates by clicking "Submit Note" which becomes visible to all system users.
- **Resolving Faults:**
  - Users can mark the machine as fixed on /faults/<case_id>/ by using the option "Mark as Resolved".
  - Submit the form by entering an endpoint note which includes “Machine operational.”
  - The machine system status shows “OK” while the fault management system marks the case “Resolved.”
- **Clearing Warnings:**
  - Navigate to /warnings/ (via “Manage Warnings”).

- Users should select the active warnings assigned to their machines then choose the “Remove” option to eliminate them.
- When no issuing faults or other warnings exist the machine status automatically changes to "OK."
- **Tip:** Check machine functionality first before removing warnings per the tip..

4. Manager Users

Managers use their position to direct operational activities alongside machinery control responsibilities and report creation for executive-level decision-making.

- **Accessing the Dashboard:**
  - After signing in the dashboard (/dashboard/) displays all machine statuses using Chart.js presentation as a pie chart to show OK/Warning/Fault counts.
  - Vital information displays after selecting collections such as Building-A or Floor-12 which generates filtered summary views.
- **Managing Machines:**
  - Go to /machines/ (“Manage Machines”).
  - Add a new machine: Enter name (e.g., “Welder X”), priority (1–10, higher appears first), and collections (e.g., Soldering-Machines).
  - Delete a machine: Users can choose a machine from the list and confirm its elimination (the process becomes permanent).
  - Assign Technicians or Repair personnel to machines via checkboxes.
- **Managing Users:**
  - Access /admin/ (Django admin interface, Manager-only).
  - Create users: Add new users by entering usernames together with passwords then designating their access role from the choices Technician, Repair, Manager and View-only.
  - Delete users: Select and confirm (use cautiously).
  - **Note:** Strong passwords must be used to protect from security threats.

- **Generating Reports:**
    - Navigate to /reports/ (“Reports”).
    - The system provides users with a option named “Export CSV” after selecting either a machine or specific collection.
    - Download a file (e.g., report_building_a.csv) with machine IDs, statuses, and fault history.
- **Commenting on Faults:**
    - The system allows both Technicians and users with similar access to enter notes added with images at /faults/<case_id>/ to provide examination feedback.

5. General Tips

- **Security:** To ensure your session safety log out of the system using the /logout/ command.
- **Browser Compatibility:** The system was evaluated using three selected browsers Chrome, Firefox and Edge but users should employ their most current browser versions.
- **Errors:** To resolve page loading failures check the server status through docker ps and refresh the current screen.

13. Conclusion and Future Enhancements

All project requirements are fulfilled by the delivery of a powerful yet easy-to-use web platform which enables factory machinery maintenance management at ACME Manufacturing Corp. This section provides a conclusion on the project delivery along with developmental evaluation and proposals for upcoming versions.

Project Outcomes

All essential objectives within the assignment reach completion through this system which provides:

- **Core Functionalities:** The application includes both static Home, Features, Pricing, Testimonials, Contact and login pages together with dynamic Fault Creation, Fault-details, Machine-details and Report-fault pages that connect through a responsive navbar.

- **Role-Based Access:** The system enables three user roles to obtain secure authentication while receiving customized dashboards and permission access according to their role (example: Managers possess user management authorization).

- **Fault and Warning Management:** The system enables technical staff to establish faults and warnings while repair team members fix these issues which all users involve through database transactions and picture uploads.

- **Manager Insights:** Operating managers benefit from Manager Insights features which combine Chart.js visualizations and CSV report exporting capabilities.

- **External Integration:** Through the REST APIs (/api/machines/, /api/faults/, etc.) users can make JSON POST requests with available external system test forms.

- **Deployment:** A Dockerized application operates under automated SQLite migrations and maintains a factory_data volume which allows the system to work from an empty database.

Team Collaboration and Lessons Learned

Development activities showed that team collaboration functions as a powerful organizational force:

- **Effective Coordination:** The Project Manager Elias used weekly meetings together with Git-based collaboration to maintain team alignment while he resolved conflicts as part of his coordinating role..

- **Challenges Overcome:** Issues like Docker database connectivity and navbar responsiveness were addressed through pair programming and iterative testing, reinforcing problem-solving skills.

Key lessons include:

- **Planning is Critical:** The project experienced better outcomes by making advance role assignments during March 18 and implementing feature selection which prevented requirements from expanding beyond essential needs.

- **Testing Early Saves Time:** The practice of conducting constant unit and manual tests during development phases enabled the identification of bugs including API JSON errors which reduced time needed until final integration was ready.

- **Communication Drives Success:** Success dependency stems from how updates posted to GitHub issues combined with meetings ensured task understanding by all team members.

Multiple learnings about the web development pattern strengthened our understanding of the entire project cycle beginning with requirements collection and ending with release deployment as per the assignment's core objectives.

Future Enhancements

The existing system fulfills ACME's business requirements but future development will boost its overall value for the company:

- **Fault Categorization:** The system needs an improvement by introducing fault classification categories consisting of Electrical and Mechanical types so management teams can track recurring problems.
- **Indoor Maps:** The system should combine an indoor floor map display which helps Technicians navigate through the factory premises.
- **Repair Cost Tracking:** A repair cost tracking system must allow employees to record expenses including components and manual labor which supports financial planning and audit requirements.
- **Notifications:** The implementation of notifications through in-app alerts with email updates will allow faster responses according to new fault occurrences and system status modifications.
- **Extended APIs:** The system should include enhanced application programming interfaces by adding machine and user data management routes such as POST /api/machines/ to expand external system functions.
- **Multi-Language Support:** The application includes multi-language support through the integration of internationalization tools available in Django for different factory personnel.

The features were removed from development because of limited time yet they comply with ACME’s performance efficiency drive and can take priority during product deployment.

Summary

The Factory Machinery Status & Repair Tracking System delivers an operational solution to ACME Manufacturing Corp. which accomplishes all project mandates with an appealing interface and powerful server infrastructure and dependable deployment systems. Through documented weekly team meetings the application achieved high quality and allowed members to develop essential web development and teamworking competencies. The system has future potential through fault classification and notification capabilities which will increase its industrial adaptability by maintaining its relevance to changing requirements. The completed web application functions as evidence of our capacity to construct intricate systems despite university conditions.