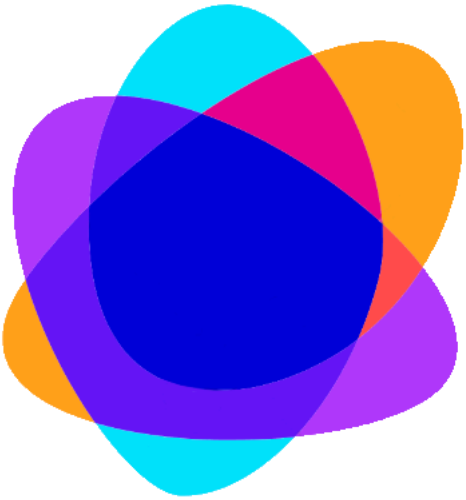
הנדסת תוכנה
שנקר
מהנדסים טאלנטים בתוכנה



Fixit

THE EASY WAY TO FIX

Methods in Software Engineering
Project Book

Authors:

Eliya Samary 206484628
Asaf Zafrir 205929029

31/03/2024

Abstract

In the current era where we use a variety of systems for the benefit of maintenance, the concept of facility maintenance is still in the previous era.

In most cases, the maintenance personnel focused on the fault itself and not on the big picture, this is due to an inability to remember the extensive collection of faults they encountered in the past, or that their predecessors in the position experienced.

Today, the proposed solution is to publish the phone number of the maintenance personnel throughout the organization so that as soon as a malfunction is detected, it will be possible to contact the maintenance personnel to report it. Often this solution creates a mess in managing all the faults together.

Contacting the wrong maintenance agent, incomplete reporting, and unclear location are just some of the problems created by this solution.

Our goals in the project are to create an efficient reporting and monitoring interface for users while ensuring a short filling time while reporting with no complexity at all, in addition to creating a standardization for reports so that the maintenance personnel have all the relevant information for handling the problem.

Our approach to the solution is to model the existing infrastructures, and the types of possible faults from an approach that there are always new problems, thereby helping users to report faults easily and allowing the maintenance personnel to manage all the faults effectively.

Table of Contents

Table of Figures

1 INTRODUCTION

In different environments identifying and resolving faults in time is essential to maintaining a safe and efficient living or working space.
The fault management and mapping system seeks to provide a user-friendly platform for fault reporting and management, while ensuring a quick and organized response to problems in a designated area. In the current era where we use a variety of systems for the benefit of maintenance, the concept of facility maintenance is still in the previous era. In most cases, the maintenance personnel focused on the fault itself and not on the big picture, this is due to an inability to remember the extensive collection of faults they encountered in the past, or that their predecessors in the position experienced .
Today, the proposed solution is to publish the phone number of the maintenance personnel throughout the organization so that as soon as a malfunction is detected, it will be possible to contact the maintenance personnel to report it. Often this solution creates a mess in managing all the faults together. Contacting the wrong maintenance agent, incomplete reporting, and unclear location are just some of the problems created by this old solution.

1.1 Problem Description

The current approach to fault management often lacks efficiency and clarity. Users face challenges in reporting faults, and administrators struggle to prioritize by urgency and address these issues promptly. The lack of a centralized system leads to delays in fault resolution and may contribute to an inefficient use of resources.

The motivation behind the Fault Management and Mapping System is to address these shortcomings by introducing a solution that enables fault reporting, classification, tracking, and conclusions drawn from the data.

1.2 Fixit Motivation

Our motivation at Fixit is to create a user-friendly platform for fault reporting and tracking, while concurrently providing maintenance teams with a centralized platform that consolidates essential information.

1.3 Fixit Goals

We have two main goals:

- Simplify fault reporting for frontline personnel - immediate and accurate.
- Enable maintenance teams to efficiently prioritize and address faults - based on location and urgency.

1.4 Overview of the Fixit Approach

The system will display all existing faults, divided by status or location.

A user will be able to report a new fault and track an existing one.

A maintenance user will be able to see all the faults that exist in his area, confirm pending faults, and close resolved faults. In addition, can also open a fault and follow it.

1.5 Usage scenarios

In order to explain the ideas behind Fixit, three scenarios are introduced in the following sections.

1.5.1 Scenario 1 – Open New Fault

A user encounters a fault or issue within the facility and initiates the process of reporting it through the Fixit application. The user accesses the application interface and navigates to the fault reporting section, where they input detailed information about the fault, including a description of the issue, its location within the facility, and its urgency level (figure 2). Upon submission, the system registers the new fault report and assigns it a unique identifier.

After opening the fault, the user can track its status through the Fixit application. They can access the "Fault Status" section of the application, where they can view a list of all faults reported by them. For each fault, the user can see its current status, such as "Open", "In Progress" or "Closed" (figure 2).

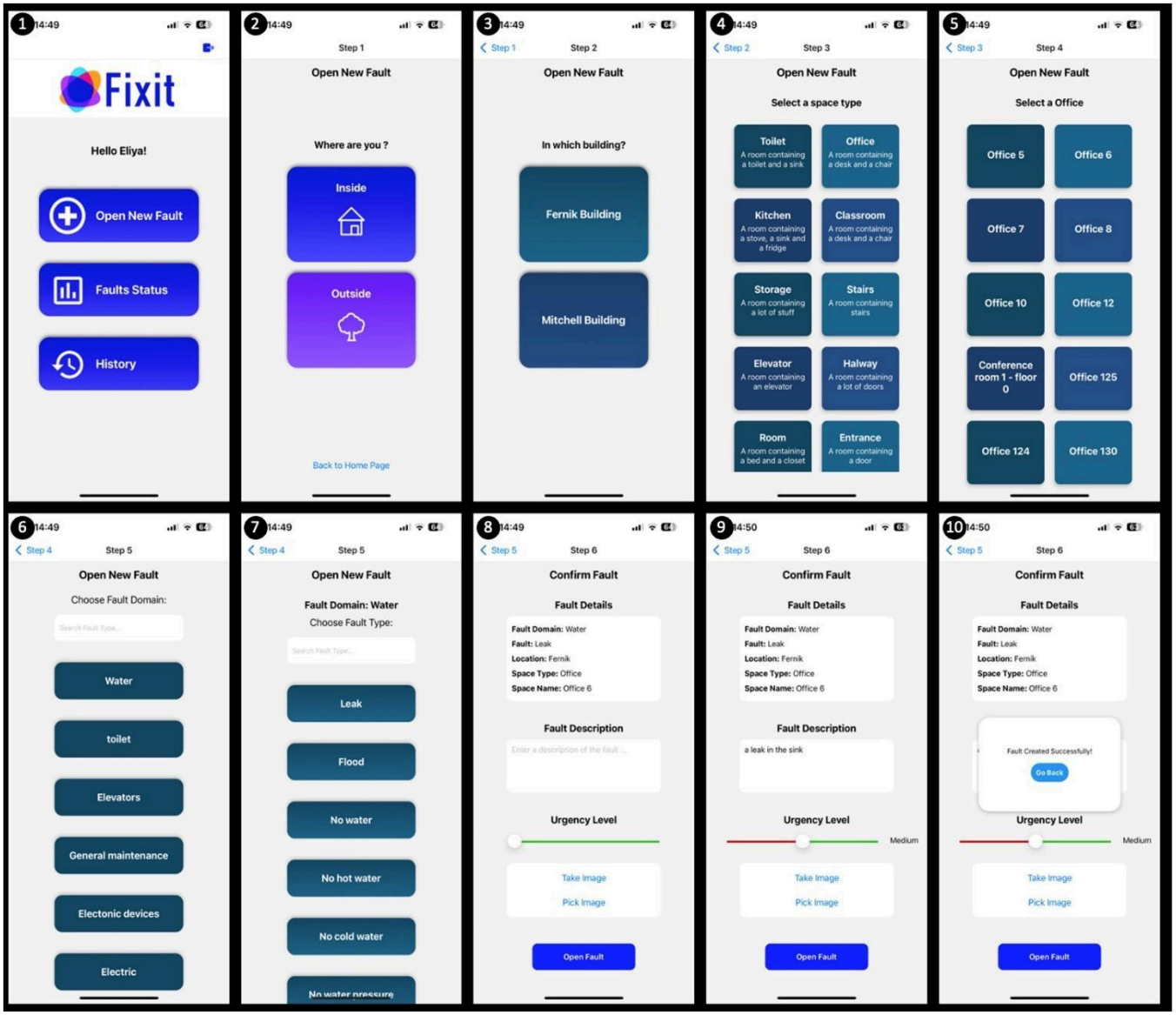


Figure 1: Scenario 1 – open new fault

In this scenario, a simple user wants to report a leak in a sink located in office 6 in Fernik Building:

1 – In home screen choose **“Open New Fault”**

2 – The system asks if the fault is inside or outside, in this case chose **"Inside".**

3 - The system asks in which building, in this case chose **"Fernik Building".**

4 – Need to select a space type, in this case chose **"Office".**

5 – Need to select a specific Office from the list, in this case chose **“Office 6”.**

6 – Need to select a fault domain, in this case chose **“Water”.**

7 - Need to select a fault sub-domain, in this case chose **“Leak”.**

8 – A confirmation form, which details the details of the fault so far, and in addition need to add a description of the fault, select urgency level and there is an option to upload a photo.

9 - Filling of the description and selecting urgency, without adding a photo in this case, click on **“Open Fault”.**

10 - A success message which confirms that the fault has been opened successfully.

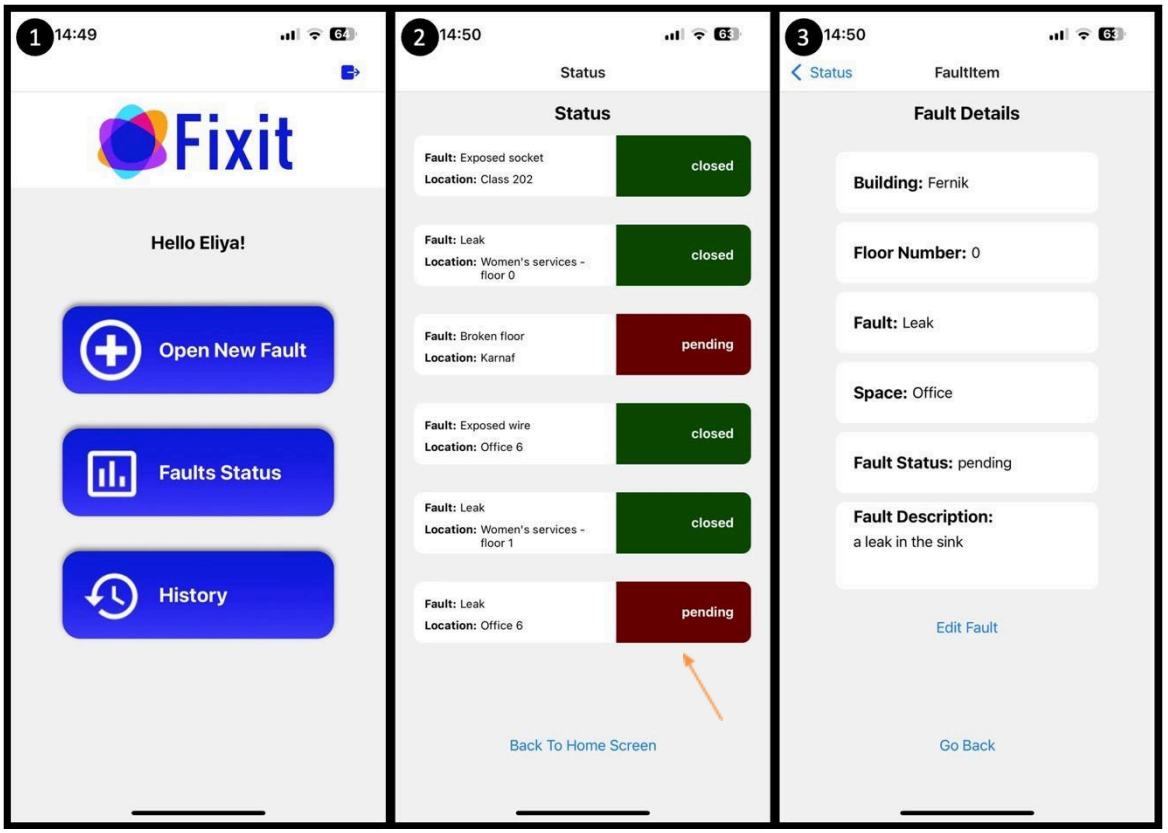


Figure 2: Scenario 1 – fault tracking

1 – Navigate to the "Faults Status" option on the home screen.

2 – Access a list displaying all reported faults by the user, then click on a specific fault to view its detailed information.

3 – View the detailed fault page, showcasing all pertinent information and specifics related to the selected fault.

1.5.2 Scenario 2 - Fault Approval by Maintenance User

A maintenance user logs into the Fixit system and navigates to the list of pending faults awaiting approval.

The maintenance user selects a specific fault from the list, reviews the details provided by the user who reported it, and assesses its urgency and severity.

Based on their assessment, the maintenance user decides to approve the fault for resolution.

The system updates the fault's status from "pending" to "in-progress".

1.5.3 Scenario 3 - Fault Resolution and Closure

The maintenance user proceeds to address the reported fault by performing necessary repairs, maintenance, or corrective actions.

Once the fault has been successfully resolved, the maintenance user updates its status in the system to "closed."

The system records the status change and updates the fault's status to "closed," indicating that it has been resolved.

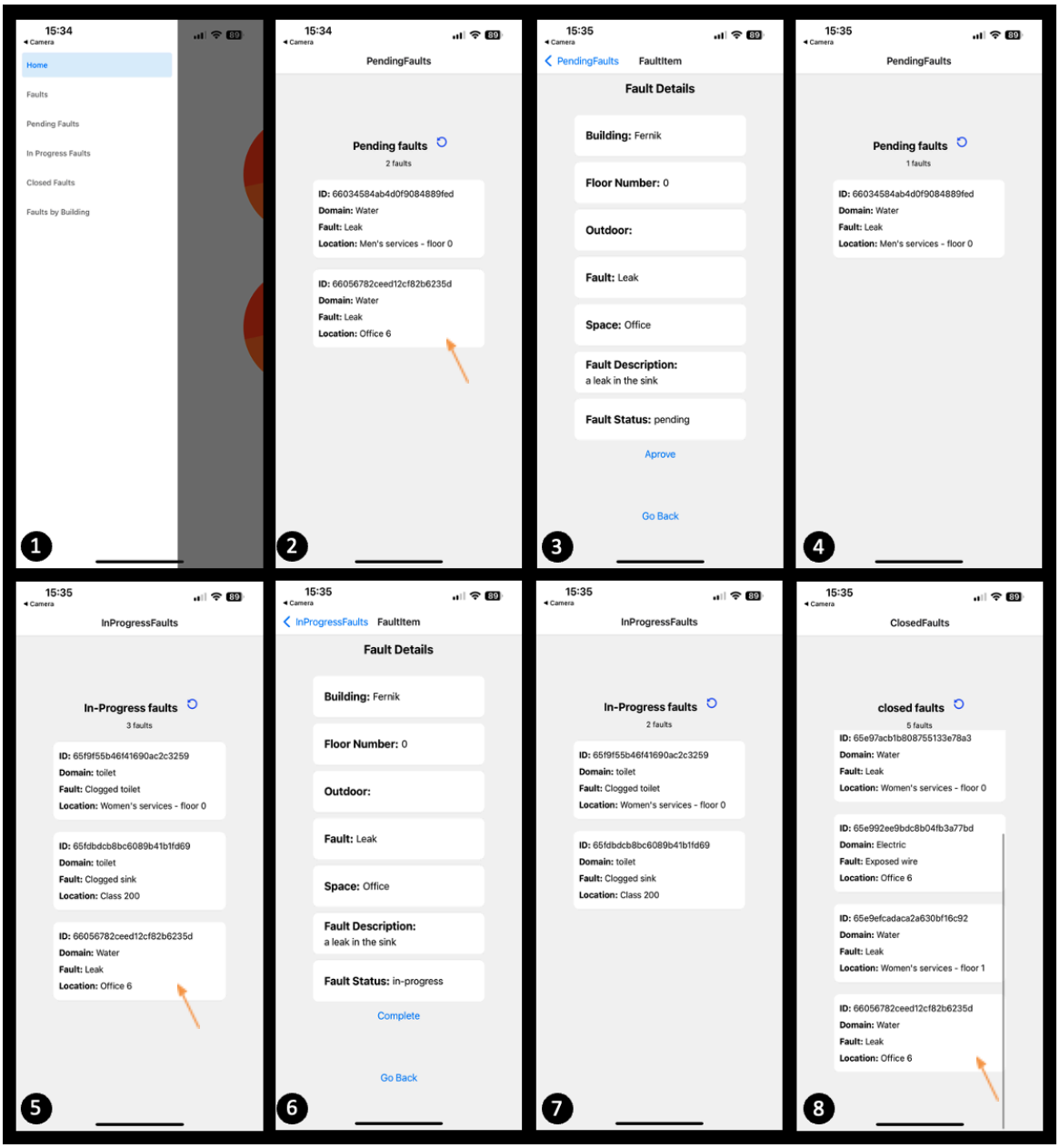


Figure 3: Scenario 2-3 – Fault Approval, Resolution and Closure

In this scenario, a maintenance user reviews a newly opened fault in "pending" status, confirms it to transition to "in-progress", and upon resolving the issue, closes the fault, updating its status to "closed":

1 – Navigate to the "Pending Faults" option on the menu.

2 – View all pending faults and select a specific one to see its details.

3 - Review the fault details and proceed to confirm it by clicking the "Approve" button.

4 – Notice that the fault has been moved out of the "Pending faults".

5 – After addressing the fault, go to the "In-Progress Faults" option in the menu to view all active faults. Select the one that has been resolved.

6 – Review the fault details and proceed to close by clicking the "Complete" button.

7 - Notice that the fault has been moved out of the "In-Progress faults".

8 – Navigate to the "Closed Faults" option on the menu and view all closed faults.

1.6 Fixit audience

The audience of Fixit comprises large-scale facilities grappling with the complexities of efficiently tracking, managing, and rectifying faults. These challenges often result in delays in addressing frontline personnel-reported issues and inefficiencies in the prioritization and resolution of maintenance tasks.

The user audience includes both individuals within the facility and the maintenance teams responsible for its upkeep.

1.7 Glossary

The following terms are divided into two categories: generic and Fixit specific terms.

1.7.1 Generic terminology

1. **Web application** - Software program accessed via a web browser that performs specific tasks or provides functionality to users over the internet.
2. **FE** – front-end. The client side of the web application.
3. **BE** - back-end. The server side of the web application.

1.7.2 Fixit specific terminology

- **Type Of Users:** The system has two types of users:
  1. **Simple User** - Individuals that associated with the organization who encounter faults and use the system to report them.
  2. **Maintenance user** - Maintenance personnel that are responsible for a specific type of service / specific area within the organization.

- **Level Of Urgency:** Fault urgency can be categorized into one of five states:
  1. **Lowest**: Provides information without requiring action.
  2. **Low**: Minimal impact on operations or safety.
  3. **Medium**: Needs attention but not immediate action.
  4. **High**: Urgent issues needing prompt resolution to prevent major disruptions.
  5. **Critical**: Requires immediate attention to prevent severe risks.

- **Fault Status:** Fault status can be categorized into one of three states:
  1. **Pending**: When a user opens a new fault, it enters a "pending" status until it is approved by the maintenance personnel.
  2. **In Progress**: After approval by the maintenance personnel, the fault transitions to an "in progress" status until it has been repaired and subsequently closed by the maintenance personnel.
  3. **Closed**: Once the fault has been successfully addressed, the maintenance user closes it within the system.

2 TECHNOLOGICAL SURVEY

In this section will delve into the technologies employed during the system's development and elucidate the rationale behind their selection.

2.1 Backend

We opted for Express (Node.js) as our backend technology due to its lightweight, flexible nature, and rich middleware ecosystem. These qualities streamline routing and database interactions and guaranteeing efficient error handling.

Alternative: Django (Python)

Reasons it may be less suitable:

- Django, while powerful, can be more opinionated and have a steeper learning curve compared to Express, particularly for developers less familiar with Python.
- It may not provide as much flexibility in terms of middleware options compared to Express, which could be limiting for the project's specific requirements.

Why we chose Express:

- Express offers a lightweight and flexible framework, ideal for rapidly building RESTful APIs and handling complex routing.
- Express is writing in java script thing that enables the use of open source (npm) and makes it easier to implement complex aspects of the application's capabilities.
- Its extensive middleware ecosystem provides flexibility in integrating with various components, which aligns well with the project's need for streamlined database interactions and routing.

2.2 Frontend

For the frontend development, we opted for **React Native** because we wanted a framework for building mobile applications that enable rapid development, code reuse across platforms, and the creation of native-like mobile applications with strong performance.

Alternative: React

Reasons it may be less suitable:

- While React provides strong typing and static analysis benefits, it may require additional configuration and setup compared to React Native.
- Integrating React with TypeScript into a mobile application development workflow may involve more overhead and complexity compared to using React Native, particularly for developers primarily focused on mobile app development.

Why we chose React Native:

- React Native offers a robust component-based architecture and a vast ecosystem of libraries, facilitating rapid development and code reuse across iOS and Android platforms.
- React Native is specifically designed for mobile app development, providing seamless integration with native components and APIs, which aligns well with the project's requirements for building a user-friendly mobile application.

2.3 Database

We chose **MongoDB** for its flexibility and scalability, its schema-less data model and document-oriented storage simplify development, allowing us to accommodate dynamic data requirements effectively. This choice aligns well with our need for a non-relational database, given the complexity of our information systems, providing the flexibility required for our data handling and schema design.

Alternative: Couchbase

Reasons it may be less suitable:

- While Couchbase offers NoSQL capabilities similar to MongoDB, it may have a different approach to data storage and querying, potentially requiring a learning curve for developers already familiar with MongoDB.
- Couchbase may have fewer available resources and community support compared to MongoDB, which could pose challenges in troubleshooting and development assistance.

Why we chose MongoDB:

- MongoDB's flexibility, scalability, and ease of use make it well-suited for accommodating the project's dynamic data requirements.
- Its document-oriented storage and schema-less data model align with the project's need for agile development and adaptable data structures, allowing for seamless integration with modern web application frameworks like Express and React Native.
- MongoDB's horizontal scalability and native support for JSON-like documents ensure optimal performance and seamless integration with modern web application frameworks.

3 REQUIREMENTS AND SPECIFICATION

Functional requirements of the project are the necessary conditions that relate to the functionality of the system. **Non-functional requirements** of the project are requirements which do not relate to functionality but to aspects that influence the manner in which the functionality is provided.

3.1 Fixit functional requirements

The following requirements relate to the basic functionality of Fixit, divided by types of users, system, and database.

Functional requirements for Simple users also apply to Maintenance users.

3.1.1 Simple User

1. **Open New Fault** –

   1.2. A user should be able to open a new fault by inputting all the necessary details. The system ensures all necessary information is provided.

   When initiating a fault report, the user navigates through sequential screens, with each screen presenting specific questions regarding the fault. The user is required to select or search for the appropriate details at each step, such as location, fault domain, urgency.

   This approach helps prevent errors and ensures comprehensive information entry.

   In addition, a user can add "unnecessary details" such as uploading a photo and writing a description.

   1.3. Opening a fault anywhere possible in the facility - the capability for users to report issues or faults from any location within the facility.

2. **Edit Fault** – user should be able to edit a fault he opened as long as it is not confirmed by the Maintenance user (still in a pending status).
3. **Tracking Fault** – user should have the ability to monitor the status of faults they have reported.
4. **History** – user should have access to view a comprehensive list of all the faults he reported.
5. **Notifications** - user should receive notifications when a fault he reported has been resolved and closed.

3.1.2 Maintenance User

1. **Faults Displaying** - user should have access to a real-time display of faults categorized by urgency and location. This includes viewing fault status, history, and filtering by specific buildings or outdoor areas. A users will have access to view only the fault reports relevant to their specific area.
2. **Change Fault Status** - user will have the capability to update the status of a fault. If a fault is 'pending,' the user can transition it to 'in-progress.' Similarly, if a fault is 'in-progress,' the user can then update it to 'closed.
3. **Edit Fault** - user should have the ability to edit a fault as long as it remains in a pending status and has not yet been confirmed.
4. **Delete Fault** – user will have the option to delete a fault. This functionality is useful in scenarios where a fault is not understood or has been mistakenly reported.
5. **Fault Analysis** - user should have the ability to review analytical information derived from fault reports, such as the frequency of failures categorized by location or urgency levels.
6. **Data Analysis** - Data analysis from real-time data and presenting the insights derived from it to the user.

3.1.3 System

1. **Duplicate fault** - When inserting a new fault, the system will check whether such a fault already exists in the system, and instead of duplicating it, we will add the user who opened the new fault as a joiner to the old fault that has already been opened.
2. **Versatile Adaptability** - The system is generic and can be adapted to any facility after mapping the data (buildings and outdoors).

3.1.4 Database

1. **Fault Identification** - The database system must support the identification and documentation of every conceivable fault, organized into fault domains and sub-domains (via fault domains collection).
2. **Facility Identification** - The database system must facilitate space identification and mapping by cataloguing all facility spaces and their spatial representation.

3.2 Fixit non-Functional requirements

The following is a list of non-functional requirements:

1. **Display** – The system will be optimized for IOS operating system for 6.7 inches iPhones.
2. **User-Based Access Control** - System functionalities and data access should be restricted based on user type and permissions.
3. **Customized System Configuration** - The system should be tailored to the unique specifications of the designated space - area mapping.
4. **User Experience** - The system should have an intuitive and user-friendly interface to facilitate efficient fault reporting and tracking.
5. **Optimized Interface and Scalability** - The system should be scalable to accommodate a growing number of users, faults, and administrators without a significant degradation in performance.
6. **Internet Connection** - The system requires internet connection.

3.3 Specification - the scope of the Fixit project

After a detailed consideration of the requirements in the context of the suggested approach, and considering the resources available, it was decided that:

- **Inside the scope of the project**:
  - Functional requirements:
    - Simple User :1-4.
    - Maintenance User : requirements 1-5.
    - System : requirement 2.
    - Database : all.
  - All the non-functional requirements.
- **Outside the scope of the project**:

These topics were omitted from consideration due to time constraints:

- Simple User – requirement 5 **Notifications.**
- Maintenance User – requirement 6 **Data Analysis.**
- System – requirement 1 **Duplicate fault.**

3.4 The Fixit approach

Our approach focuses on user-centric fault management, streamlined reporting, maintenance user features, and scalable architecture.

- **Fault Reporting and Tracking:**
  - Implement functionality for users to report new faults easily, providing fields for detailed information and categorization.
  - Develop features for users to track the progress of existing faults, including updates on status changes and resolutions.

- **User Interface Design:**
  - Develop an intuitive and user-friendly interface for easy navigation and interaction.
  - Implement clear categorization of faults based on status (e.g., pending, in progress, closed) and location.
  - Provide accessible options for reporting new faults and tracking existing ones.

- **Maintenance User Features:**
  - Create specialized views for maintenance users, displaying faults specific to their assigned areas.
  - Enable maintenance users to confirm pending faults, mark faults as resolved, and close them once resolved.
  - Provide options for maintenance users to open new faults and monitor their progress through the system.

- **Notifications and Alerts:**
  - Implement a notification system to alert users about updates on faults they are involved in, such as status changes or resolution.

- **Security and Access Control:**
  - Implement role-based access control to restrict access to sensitive functionalities based on user roles (e.g., regular user, maintenance user, administrator).

- **Scalability and Performance:**
  - Design the application architecture to be scalable, capable of handling a growing number of users and faults without compromising performance.
  - Optimize database queries and system processes to minimize response times and ensure smooth operation, even under heavy loads.

- **Testing and Quality Assurance:**
  - Conduct thorough testing of all functionalities to identify and address any bugs or issues.
  - Perform usability testing to ensure the application meets user expectations and provides a seamless experience.

- **Continuous Improvement:**
  - Gather feedback from users to identify areas for improvement and implement enhancements to the application over time.
  - Stay updated with industry trends and best practices to incorporate new features and technologies that can enhance the application's functionality and usability.

By following this approach, the fault management application can effectively meet the needs of users, streamline fault reporting and tracking processes, and contribute to efficient facility maintenance and management.

4 THE ARCHITECTURE

In order to describe the architecture of the Fixit application we start by describing the data that is needed, where it is originates and how it is transformed. This is followed by explaining the roles that are involved with the application. Finally, the activities that take place when using the system are explained resulting in a description of the system life cycle – the stages through which the system goes when it is being used.

4.1 Fixit Data

The Fixit application employs a MongoDB database, featuring several essential collections: faults, fault domains, space types, buildings, outdoors, users, and maintenance.

Customization and Preparation:

- The buildings and outdoors collections are tailored to represent the facility's layout, while the space types collection encompasses all possible space classifications within the facility.
- The fault domains collection encompasses all conceivable fault domains and sub-domains, requiring meticulous preparation.

Data Management:

- The faults collection serves as a repository for all registered faults, capturing their respective statuses. Each newly opened fault is seamlessly integrated into this collection.
- User data, inclusive of maintenance-specific details, is stored in the users and maintenance collections, respectively.

Data Access and Flow:

- The system's server modules facilitate efficient data access, utilizing Mongoose for database queries. This approach ensures a structured and traceable data flow, with data exclusively accessed and exported through these modules for subsequent processing by the controllers.

4.2 Fixit Processes

Based on the description provided, the main processes of Fixit can be outlined as follows:

1. **Fault Reporting and Monitoring Process:**
   - This process revolves around opening a fault and monitoring its status.
   - Initiated primarily by simple users who encounter faults or issues within the facility.
   - Steps include:
     - Opening a Fault: Users report faults they encounter through the system, providing necessary details such as fault description, location, and urgency.
     - Monitoring Fault Status: Users can track the progress of their reported faults, from submission to resolution, through the system interface.
     - Objective: Ensure prompt reporting and monitoring of faults by simple users to facilitate timely resolution by maintenance personnel.

2. **Fault Handling and Closure Process:**
   - This process involves following up on all open faults, handling them, and closing them once resolved.
   - Primarily attributed to maintenance users responsible for resolving reported faults.
   - Steps include:
     - Following Up on Open Faults: Maintenance users access the system to review all open faults assigned to them or within their designated areas.

- Handling Faults: Maintenance users take necessary actions to address reported faults, which may include confirming, investigating, and resolving issues.
- Closing Faults: Once faults are successfully resolved, maintenance users update their status in the system, marking them as closed.
- Objective: Ensure efficient handling and closure of reported faults by maintenance personnel, thereby maintaining facility functionality and user satisfaction.

These two main processes of Fixit work collaboratively to facilitate fault management within the facility, ensuring seamless communication between simple users reporting faults and maintenance personnel responsible for addressing and resolving them.

4.3 Roles

- **End Users:** the people who will ultimately use the system, divided into two types of users:
  1. **Simple User** - Individuals that associated with the organization who encounter faults and use the system to report them.
  2. **Maintenance user** - Maintenance personnel that are responsible for a specific type of service / specific area within the organization.

4.4 The life-cycle view of a system

The life-cycle view of Fixit can be described as follows, considering the information provided:

1. **Initiation:**

The life-cycle begins when a user encounters a fault or issue within the facility and initiates the fault reporting process through the Fixit application.

2. **Fault Reporting and Monitoring:**
   - Upon initiation, the user opens a new fault report, providing detailed information about the issue, such as description, location, and urgency.
   - The fault report enters the system and is stored in the faults collection in the MongoDB database.
   - The user can monitor the status of the reported fault through the Fixit interface, tracking its progress from submission to resolution.

3. **Assignment and Handling:**
   - Maintenance personnel, responsible for resolving reported faults, access the system to review all open faults assigned to them or within their designated areas.
   - They follow up on open faults, confirming, investigating, and taking necessary actions to address the reported issues.
   - The maintenance users handle the faults, updating their status in the system as they progress through the resolution process.

4. **Resolution and Closure:**
   - Once the reported fault is successfully resolved, maintenance users update its status in the system, marking it as closed.
   - The fault report remains stored in the faults collection for historical tracking purposes, reflecting its status change from open to closed.

5. **Continued Monitoring and Maintenance:**

- Even after closure, the Fixit application continues to monitor facility conditions and track faults as they arise, ensuring ongoing maintenance and operational efficiency.
- The life-cycle repeats as new faults are reported, handled, and resolved, maintaining a continuous loop of fault management within the facility.

In summary, the life-cycle view of Fixit encompasses the entire process of fault management within the facility, from initiation and reporting to handling, resolution, and ongoing monitoring, with a focus on user engagement, efficiency, and continuous improvement.

5 SYSTEM DESIGN

This section discusses the design of Fixit; it will explain the mapping of the architecture and its elements to the components that will implement.

5.1 Data components

This section will present the components we used, and how we used them, divided into three main topics:

- **Users** – All user data, including maintenance user details, is stored in the 'Users' collection. Additional information specific to maintenance users is also stored in the 'Maintenance' collection.

Related components: Users, Maintenance.

- **Facility** – All the facility information, including space types and specific areas within the facility.

Related components: Buildings, Outdoors, Space Types.

- **Fault Frame** - The system stores all opened faults along with relevant information previously saved in the system pertaining to each fault.

Related components: Faults, Fault Domains.

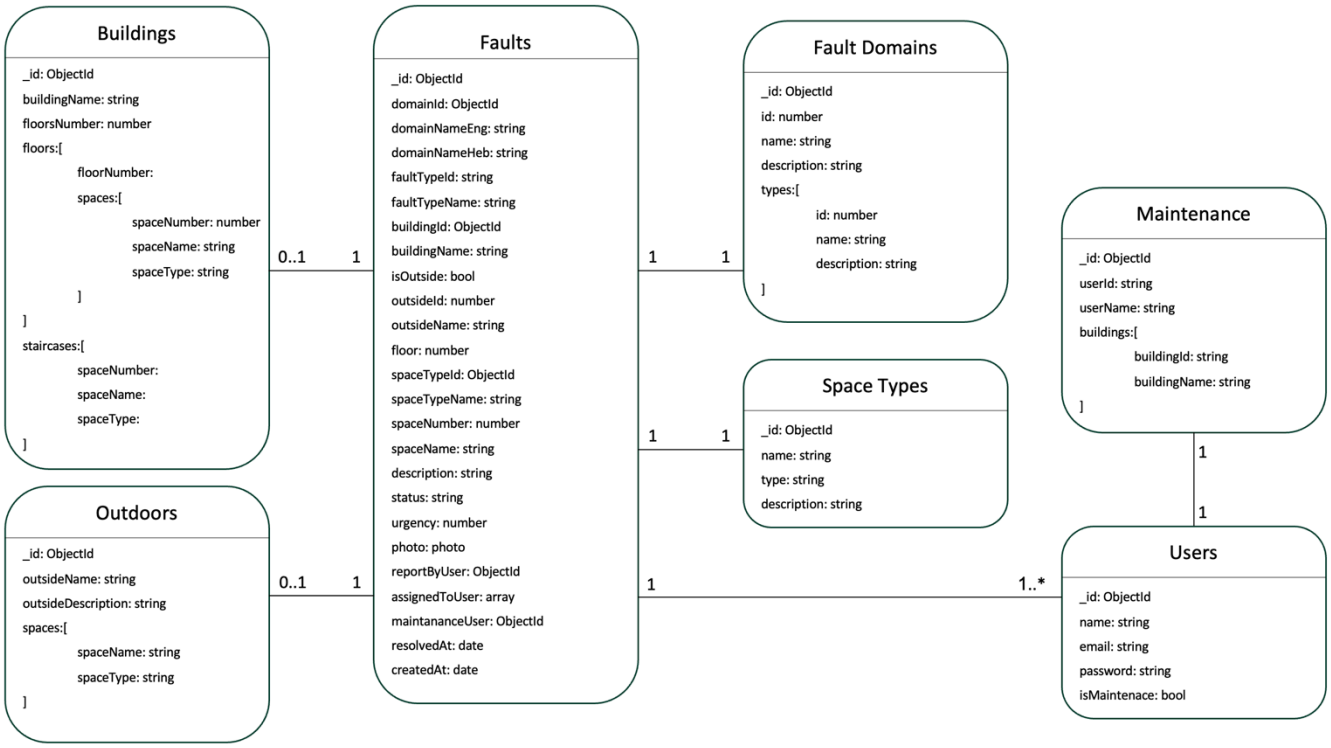


Figure 4: Fixit data components

5.2 Process components

The Fixit system refers to the key elements that make up the various processes involved in it operation. These components include:

1. **User interface:** The user interface is the component that enables users to interact with the system. It includes the visual design, layout, and navigation features that make it easy for users to access the system's features and functionality.
2. **Image upload function**: An essential function integrated into the fault description process, leveraging an external package (Cloudinary) for efficient and secure image storage.
3. **Error handling:** Robust and comprehensive error handling implemented on the server-side, encompassing HTTP requests, CRUD operations and database communication.

5.3 Communication components (user-computer interaction)

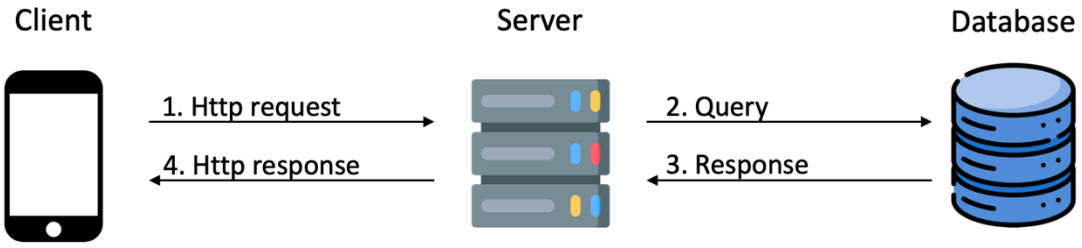


Figure 5: Fixit communication components

5.4 Interactions

This section will present different diagrams in order to describe the Fixit design.

5.4.1 State diagram

The Fixit state diagram (Figure 6) represents the behaviour of an object during its life in response to user events, together with its responses and actions.

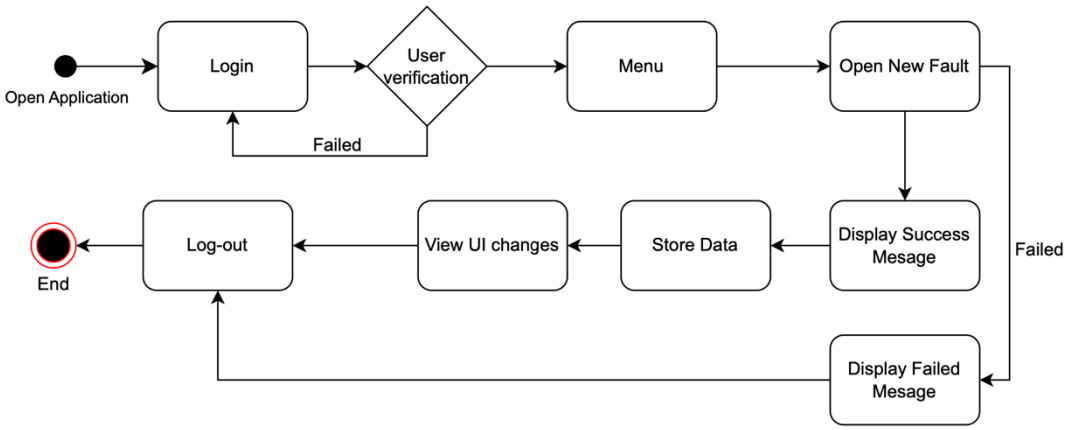


Figure 6: Fixit state design

5.4.2 Design sequence diagram

Figure 7 shows the possible branches of interaction between the user and the system.

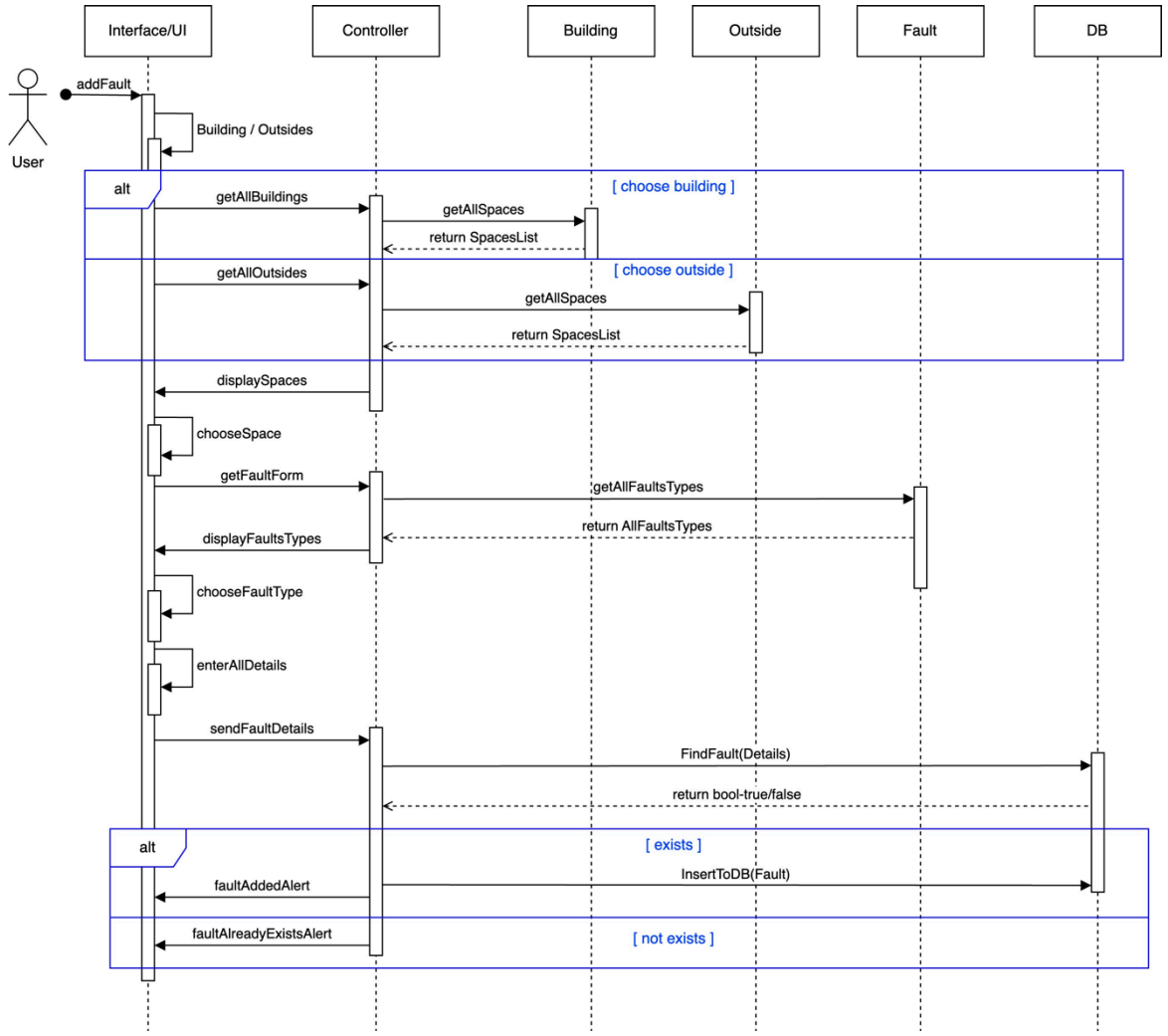


Figure 7: Fixit sequence diagram

5.5 System Architecture

The overall design describes the components that are necessary to carry out the Fixit processes and the data needed for those processes.

The configuration of the system is the result of mapping the design onto specific components using the 'Three tier software architecture':

- **Data Layer:** A MongoDB database stores all relevant system data, encompassing facility mapping, user information, and overall system data.
- **Logic Layer:** An Express server which has featuring dedicated modules for each system entity. These modules encompass CRUD operations, robust error handling, and middleware to ensure the integrity of the logic layer.
- **Presentation Layer:** A React Native application serves as the user interface, providing access to all data and operations available within the system.

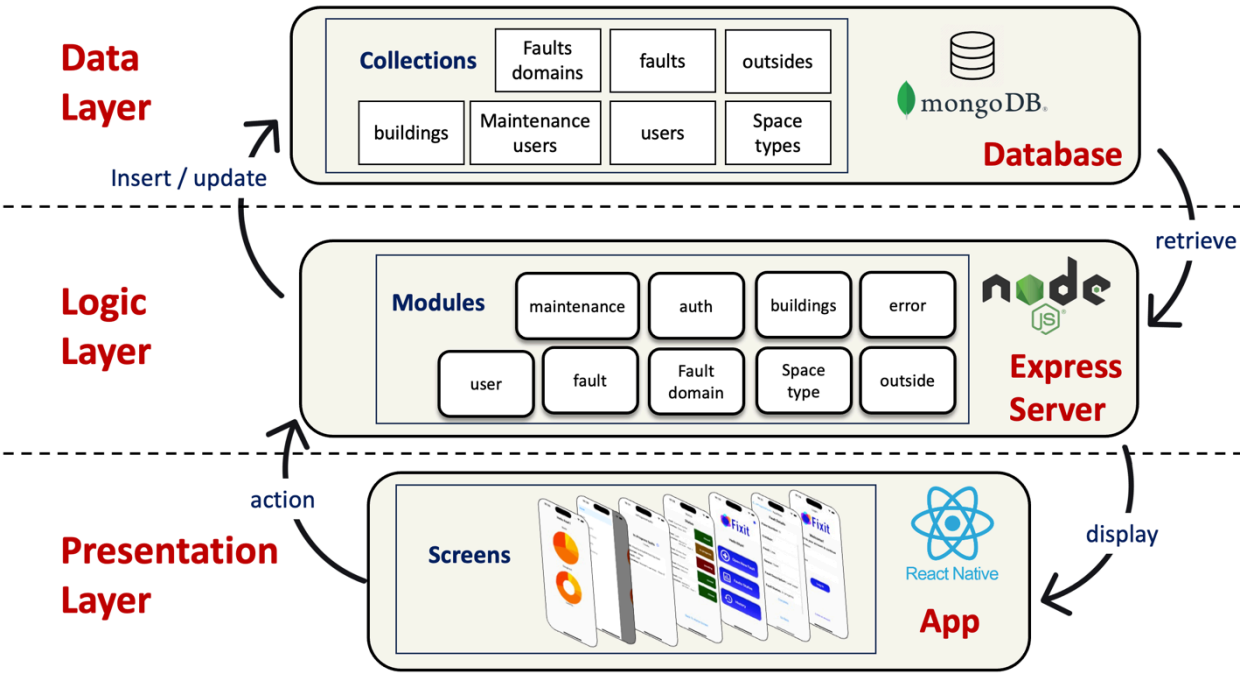


Figure 8: Fixit system architecture

6 IMPLEMENTATION

Fixit is a client-server application designed to streamline fault reporting and maintenance tracking within large-scale facilities. The decision to develop Fixit as a web-based platform was made early in the development process to harness the benefits of web technologies, including accessibility and robust GUI support.

Given Fixit's objective to manage a vast array of faults and facilitate seamless communication between users and maintenance teams, ensuring scalability and optimal performance is paramount. This becomes particularly pertinent when operating within a web-based environment interfacing with an external database.

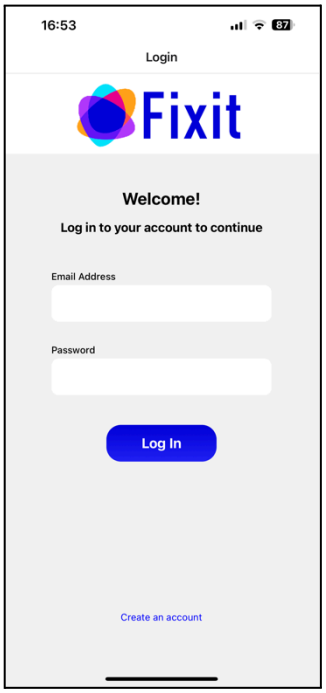
To tackle these complexities, Fixit's backend is built on Node.js, offering a non-blocking architecture capable of efficiently handling a substantial volume of user requests and delivering prompt responses. Complementing this, MongoDB serves as the database management system for Fixit, supporting horizontal scaling and ensuring high availability. This strategic technological pairing enables Fixit to dynamically adjust to fluctuating user demands while upholding rapid and consistent performance.

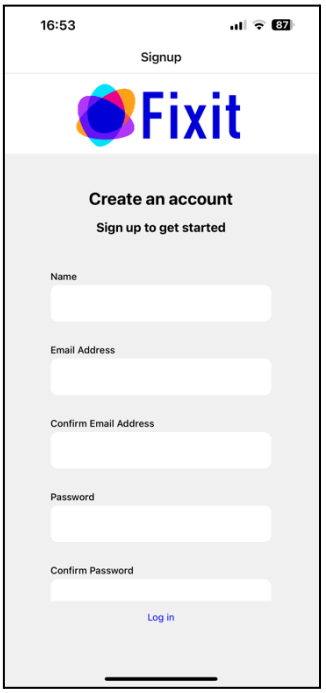
By harnessing these advanced technologies, Fixit stands as a scalable and efficient web application, empowering users to report faults, track maintenance statuses, and facilitate effective communication with maintenance teams.

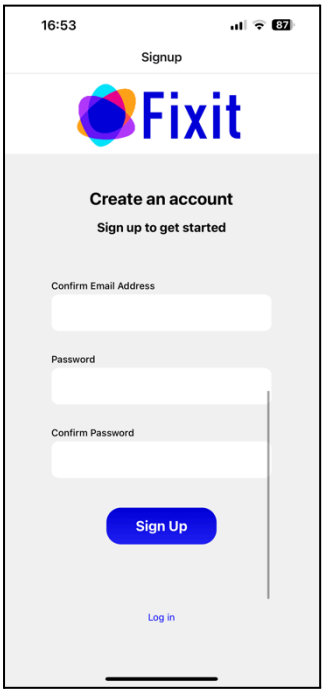
6.1 Interfaces

This section will present the user interface, which is the mobile application Fixit. The section will include screenshots of the different screens of the GUI.

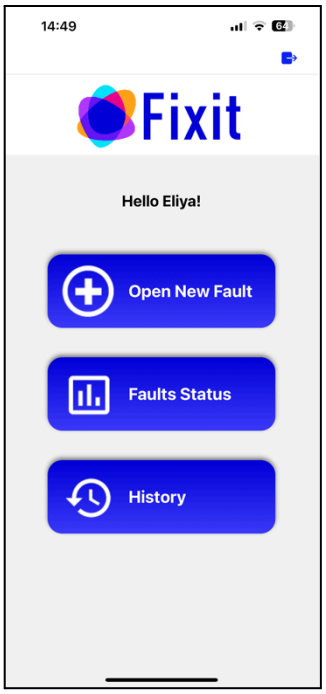
First screen – Login & Sign-up:



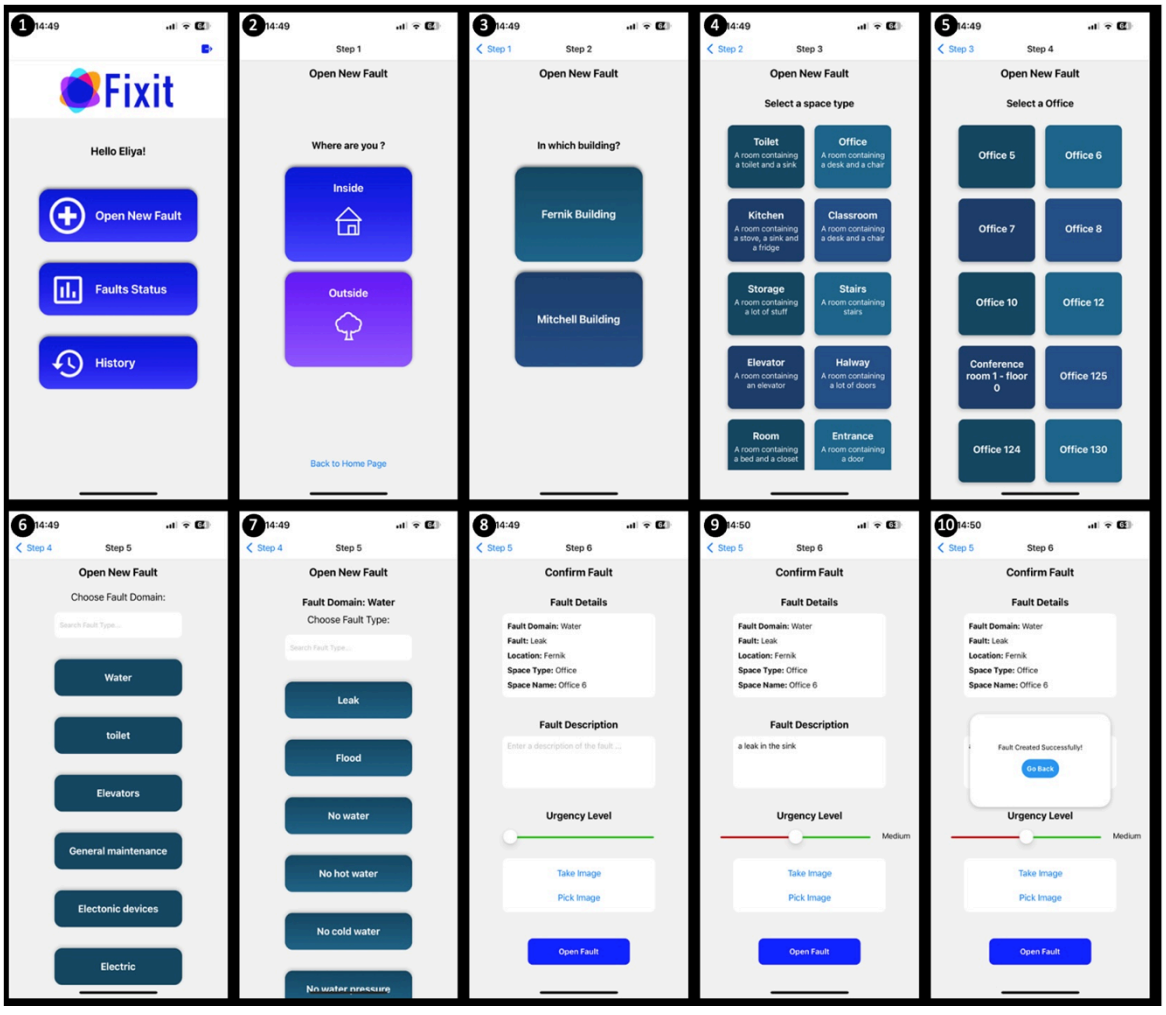




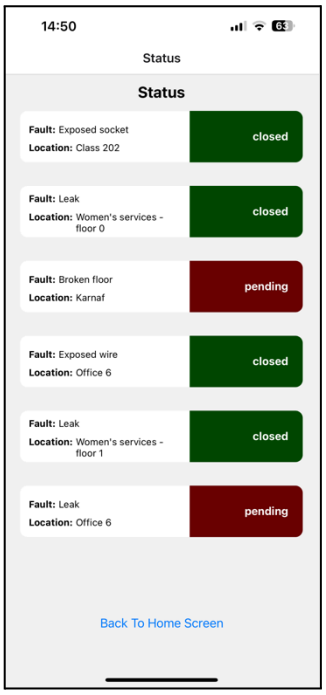
Simple user Home Screen



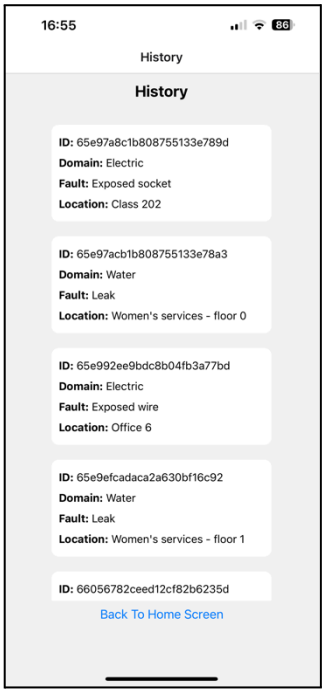
Open New Fault Screens



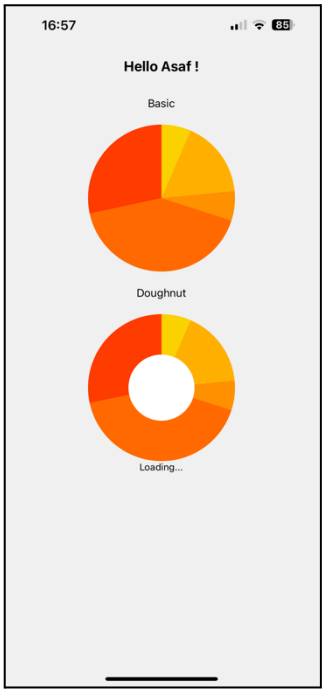
Faults Status Screen



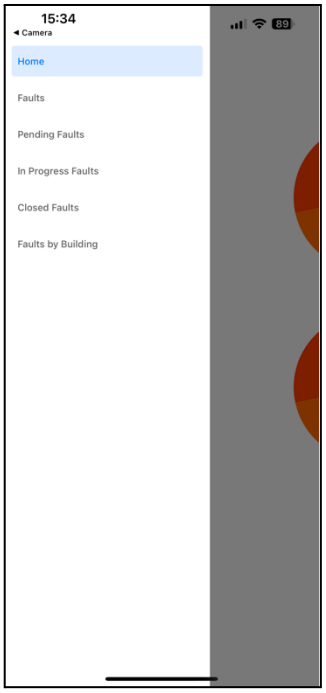
History Screen



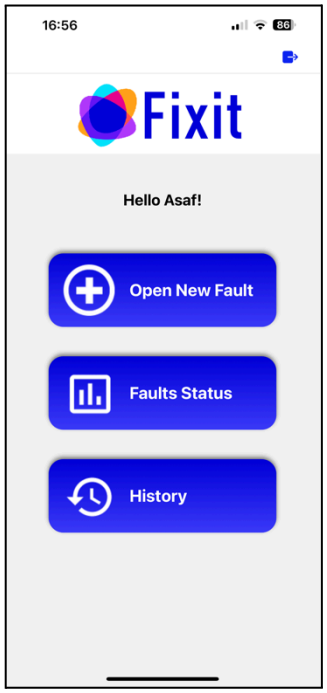
Maintenance user Home Screen



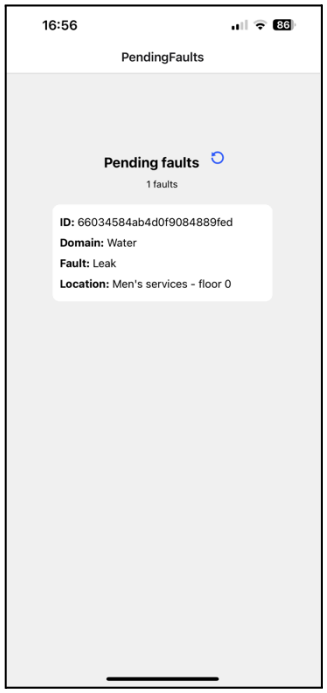
Maintenance user Navigation Menu



Maintenance user Faults Screen

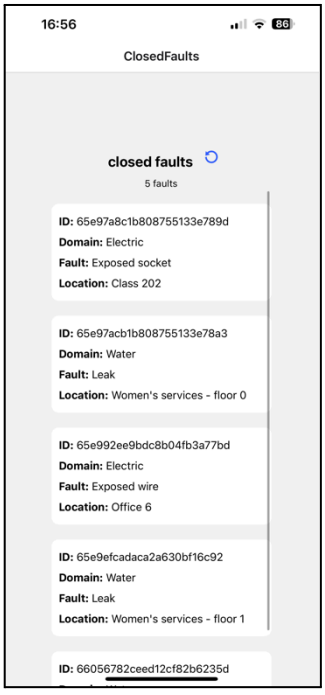


Maintenance user Pending Faults Screen



Maintenance user InProgress Faults Screen

Maintenance user Closed Faults Screen



Maintenance user Faults By Building Screen

Fault Details Screen

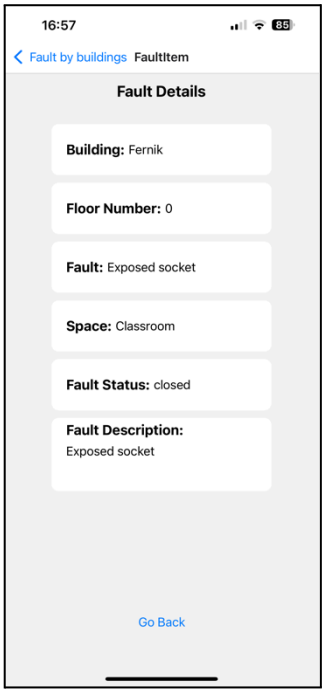


Figure 9: Fixit interfaces

6.2 Development environment

The Fixit project was developed within the Visual Studio Code environment, a widely acclaimed integrated development environment (IDE) known for its robust features tailored for coding, debugging, and testing.

Additionally, we utilized an iOS simulator and Expo platform for emulation on iPhone devices. Given that our system targets mobile applications, these technologies offered us a streamlined and efficient method to rigorously test and optimize our application.

6.3 Programming languages

Based on the selected technologies specified in the technology survey section, there are two parts of the project and each has its own environment, languages, and libraries:

- The client-side (GUI) - this part is composed of components that were developed using **React Native** (JavaScript).
- The backend side - this part is composed of components that were developed using **Express** (Node.js – JavaScript).

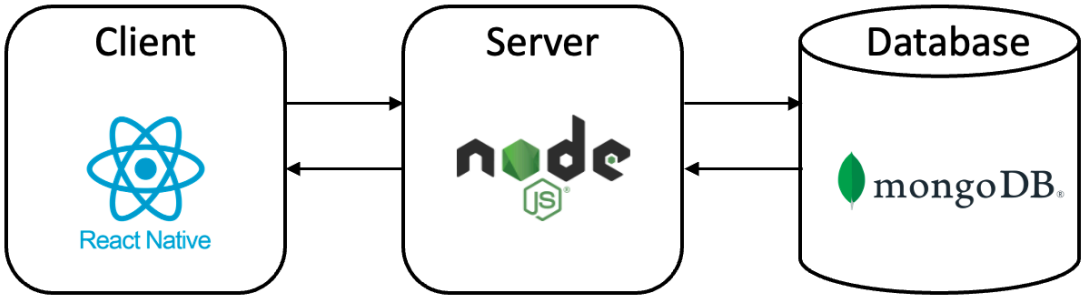


Figure 10: Fixit programming languages

6.4 Risk Management

In this segment, we will delineate potential risks and their corresponding coping strategies.

1. **Fault Type Misclassification**
   - **Description A**: Faults might be inaccurately classified, potentially leading to delays in addressing critical issues
   - **Description B:** Users may encounter challenges in selecting the correct fault type during the reporting process, potentially leading to misidentification
   - **Risk assessment:** Low
   - **Coping strategy A:** A comprehensive study will be conducted to identify and categorize all potential fault types based on their urgency levels.
   - This classification will enable the system to automatically determine and assign the appropriate severity level when users select a fault type from the closed list, thus enhancing the accuracy of fault severity assessment.
   - **Coping strategy B:** Users will be required to choose the specific fault type from a closed list encompassing all possible issues. Additionally, users can provide further details using free text to describe the fault comprehensively.
2. **Incorrect Location Reporting**
   - **Description:** Users may face difficulties accurately identifying and reporting the precise location of faults, potentially causing confusion in the resolution process
   - **Risk assessment:** Low
   - **Coping strategy:** The system fault reporting form will be customized to fit the designated space. When a user submits a fault report, they must choose a location from a closed list alongside providing free-text details.
3. **Fault Urgency Misclassification**

- **Description:** The range of possible fault types is very large, and the data may not cover all of them, so it is necessary to let the user who opens a fault enter the level of urgency.
- An issue may arise when users while reporting a fault, fail to accurately specify the correct level of urgency associated with the reported fault.
- **Risk assessment:** High
- **Coping strategy:** When a new fault is opened, the maintenance user in charge of that area will receive a notification, and must confirm the fault and the level of urgency - if it was classified incorrectly the user will correct the level of urgency.

6.5 Exceptions Management

Open new fault:

- While on the flow of opening new fault, user can't pass forwards without select/type the fault parameters (location & type).
- On the confirmation page (end of the flow), user can't send parameters to the server without set the urgency of the fault.
- User can select only one photo to be added to each fault.

Register:

- User enter Email not in the correct format: an error colour (Red) will appear on the text input.
- User doesn't enter Password: an error colour (Red) will appear on the text input.
- Password and Confirm Password are different: an error colour (Red will appear on the text input.

Login:

- If the user enters wrong details: an error message will appear.
- User enter Email not in the correct format: an error colour (Red) will appear on the text input.
- User doesn't enter Password: an error colour (Red) will appear on the text input.

6.6 Versions Control

The version management of the project was carried out using GitHub, and splitted into to different repositories, one for the FE (React Native), and the other to BE (Express).

(Links below)

6.7 Project Management

In this section we will present the project management and work division.

- **Main milestones:**

January 21 - SRS submission

February 4 - SDD submission

February 2 - Starting work on the code (front and back)

March 7 - Presentation and alpha

- **Work division:**

In the frontend, the tasks were assigned based on clear user roles since the system caters to two types of users. Meanwhile, on the back end, we first prioritized setting up the

infrastructure and database connectivity. The tasks were then divided by specific factors within the system to streamline development and ensure cohesive functionality.

- **Detailed Gantt for the month of March**:

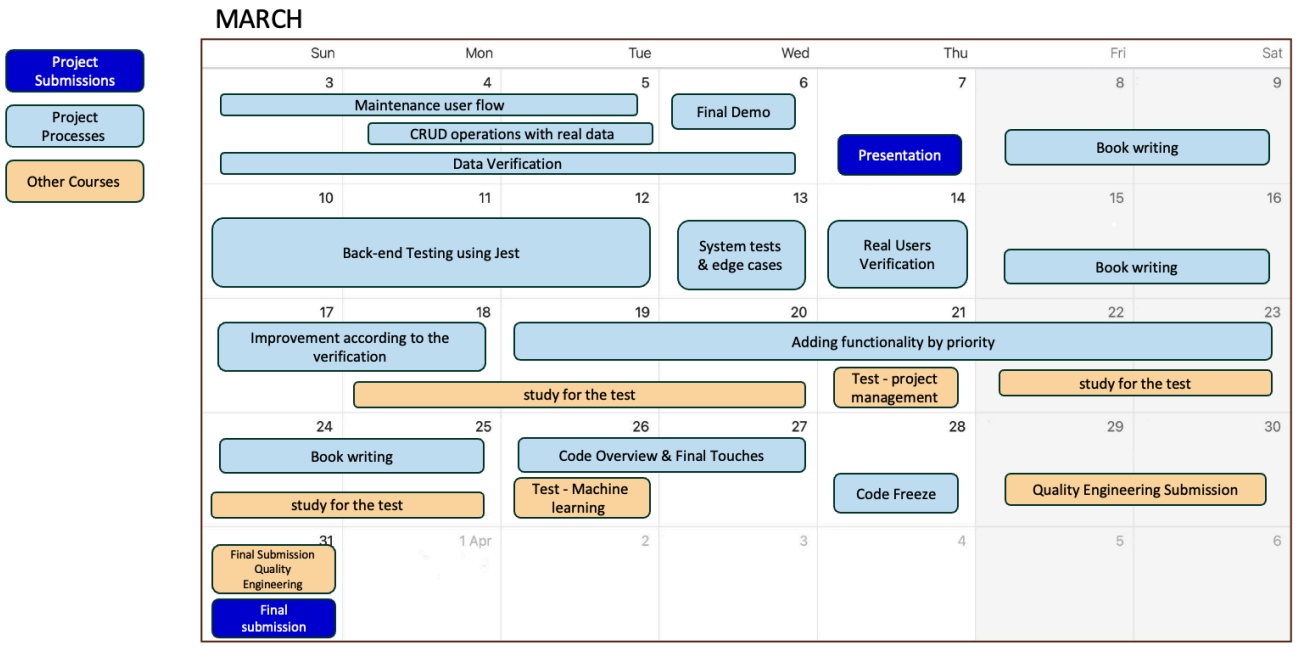


Figure 11: Fixit Gantt

6.8 Code

1. **Link to GitHub:**

Fixit FE - https://github.com/asafzaf/Fixit

Fixit BE - https://github.com/asafzaf/fixit-server

2. **Link to the recording:**

https://www.youtube.com/watch?v=FRdKx2AG3_M

3. **Using the system:**
   - Download the Expo app on your mobile device.
   - Scan the QR or type this link:

exp://u.expo.dev/update/7d0eddff-f8dc-4861-ad0b-d88c53face63

7 SYSTEM VALIDATION

The purpose of this validation report is to verify the functionality and performance of Fixit.

7.1 Backend

Purpose: Testing the server through comprehensive testing, encompassing error handling, edge cases, and CRUD operations for each system entity.

Execution: Utilizing Jest for testing, and establishing dedicated test files for each system entity to ensure thorough and structured validation.

Results: From 27 test cases, 22 have passed successfully:

- **Buildings**: 7 of 7.
- **faultDomain**: 3 of 3.
- **Outside**: 2 of 2.
- **spaceType**: 2 of 2.
- **User**: 3 of 4.
- **Fault**: 5 of 9.

Please see Appendix A for Postman documentation and Appendix B for further details on the tests.

7.2 Users Verification

Purpose: Conducting real-user validations to assess and refine system functionality and usability.

Execution: This verification is divided into two according to the types of system users:

1. **Simple Users** - To ensure the Fixit system meets the needs of its primary users – those associated with the facility – we engaged two students to evaluate the platform. Their feedback focused on assessing the interface's appropriateness, responsiveness, and overall user experience. This user-centric approach enables us to identify areas for improvement and refine the system based on real-world usage and user preferences.
2. **Maintenance Users** - To validate the practicality and effectiveness of the Fixit system from a maintenance perspective, we have enlisted a maintenance professional to actively use the platform. Their hands-on experience and insights will provide valuable feedback on system functionality, usability, and relevance to daily maintenance operations. This direct engagement with a maintenance user ensures that the system is tailored to meet the specific needs and challenges faced by maintenance teams, allowing for targeted improvements and optimizations.

Results:

1. **Simple Users**

Positive Feedback:

- The flow interface is well-organized and user-friendly, making navigation intuitive and easy.
- The ability to upload pictures significantly enhances clarity, enabling users to precisely illustrate the nature of the issue at hand.

Areas for Improvement:

- Implement a search feature across multiple pages, not just limited to the fault domain page.

2. **Maintenance Users**

Positive Feedback:

- The system's location-based display effectively groups faults by area, facilitating streamlined preparation and management when multiple faults occur in the same location.

Areas for Improvement:

- Incorporating real-time data analysis capabilities to provide immediate and insightful information for proactive decision-making.

Conclusion and Recommendations Based on User Validation: Following the comprehensive user validation process, several key insights have emerged regarding the Fixit system's functionality and user experience. Users have positively acknowledged the system's intuitive interface and the utility of features such as picture uploads for clear fault illustration and location-based fault displays for maintenance users. However, areas for enhancement were also identified, notably the need to expand the search functionality across multiple pages and the integration of real-time data analysis capabilities for maintenance users. These insights provide valuable direction for future development, emphasizing the importance of continuous improvement to meet user needs and enhance overall system performance and usability.

7.3 Data Verification

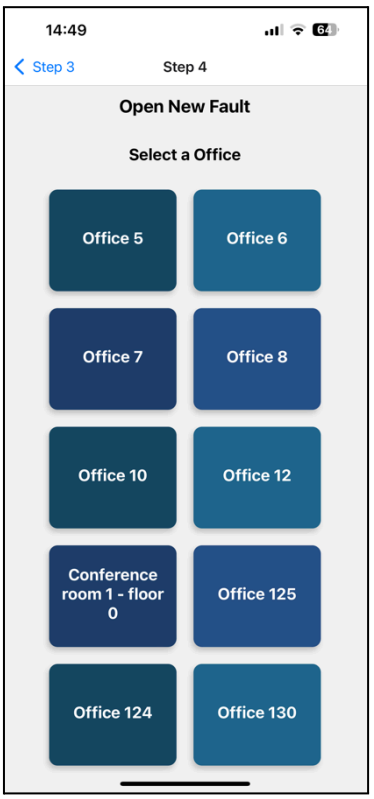
Purpose: Conducting comprehensive data validation by mapping both interior (buildings, floors, rooms, etc.) and exterior (outdoor areas, cafeteria, etc.) facility components, establishing real-time data connections to the database, and integrating the data with the system.

Execution: Integrating the facility mapping into the database and executing queries to verify data accuracy and seamless system integration.

Results: Accurate representation of the facility components within the system.

For example, we can see (1) a screenshot from the Fernik building details database, and (2) a list of all the offices in the building.

I t's evident that there is consistency between the displayed data and the actual records.



8 SUMMARY, EVALUATION, CONCLUSIONS AND FUTURE WORK

8.1 Summary

The Fixit system is designed to streamline the fault reporting and tracking process within a facility, catering to both simple users and maintenance teams. It offers a user-friendly platform for reporting faults, monitoring their status, and facilitating efficient communication between users and maintenance personnel. With features such as fault domain documentation, space identification, and real-time status updates, the system aims to enhance operational efficiency, improve maintenance prioritization, and ensure timely resolution of reported issues.

8.2 Evaluation and conclusions

The Fixit system demonstrates a comprehensive approach to fault management, addressing the diverse needs of facility users and maintenance teams. Its modular architecture, leveraging technologies like Express (Node.js), React Native, and MongoDB, provides scalability, flexibility, and robustness to support the evolving requirements of large-scale facilities. The integration of user-specific functionalities, such as fault approval and closure by maintenance users, ensures accountability, transparency, and effective collaboration throughout the fault resolution process.

However, to further enhance the system's capabilities and user experience, there is a need to focus on optimizing the user interface, incorporating advanced analytics for fault analysis, and enhancing communication channels between users and maintenance teams. Additionally, continuous feedback from users and stakeholders is essential to identify areas for improvement, address emerging challenges, and adapt the system to changing facility needs and technological advancements.

8.3 Future work

Looking ahead, there are several key areas where the Fixit system can further evolve:

- **Enhanced Fault Editing for Maintenance Users:** Enable maintenance users to modify fault details before approving them. This feature will empower maintenance teams to make necessary adjustments or corrections to reported faults, ensuring accurate and relevant information is captured from the outset.

- **Advanced Data Analysis and Insights:** Implement comprehensive data analysis capabilities to derive meaningful insights from real-time data stored in the database. By leveraging data analytics tools and techniques, the system can identify patterns, trends, and anomalies, enabling informed decision-making, predictive maintenance, and proactive fault management.

- **Staircase Fault Reporting:** Address the unique challenge of representing staircases, typically depicted as a Y-axis, within a system designed around an X-axis framework. Develop a specialized module or interface to accurately capture and manage faults associated with staircases, ensuring seamless integration and alignment with the existing system architecture and user experience.

- **Fault Registration to Existing Entries:** Introduce an option for users to register or link their report to an existing fault entry, rather than creating a duplicate entry. This feature will promote data integrity, reduce redundancy, and streamline the fault reporting process by consolidating related issues and facilitating easier tracking and management.

- **Real-time Notification Integration via Socket**: Implement Socket-based notifications to enable real-time communication and updates within the Fixit system. By leveraging Socket technology, users and maintenance teams can receive instant notifications about fault status changes, updates, and other relevant information, enhancing transparency, responsiveness, and collaboration across the platform.

9 REFERENCES

Courses taken to learn the technology:

- Node.js, Express, MongoDB

https://www.udemy.com/course/nodejs-express-mongodb-bootcamp/?couponCode=LETSLEARNNOWPP

- React Native

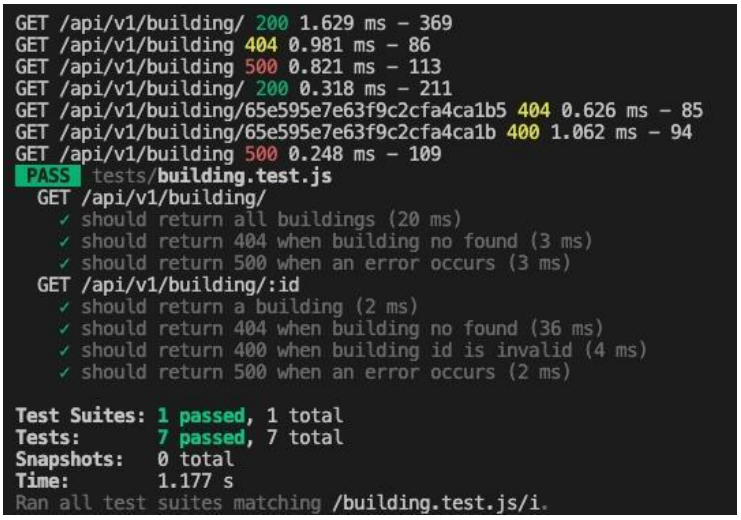
https://www.udemy.com/course/react-native-the-practical-guide/?couponCode=LETSLEARNNOWPP

10 APPENDIX A: Postman documentation

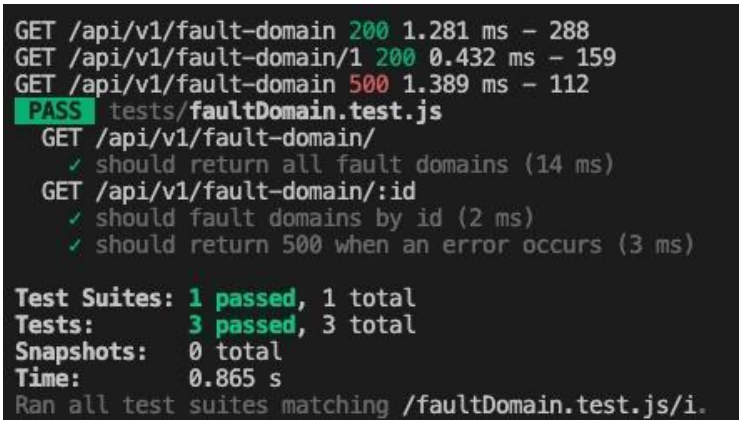
Postman documentation link
https://documenter.getpostman.com/view/31102943/2sA2rAyMWe

11 APPENDIX B: Test summary screenshots

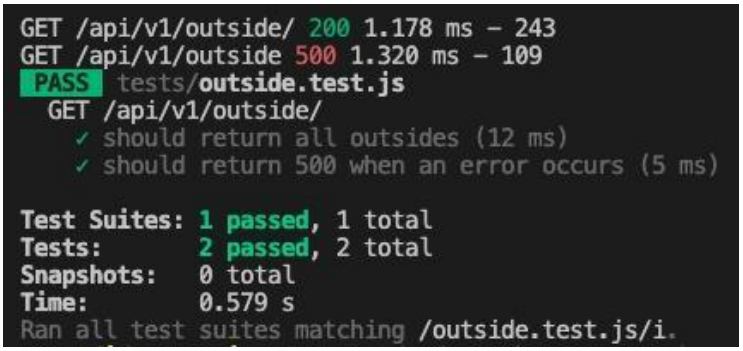
Building:



faultDomain:



:Outside



:SpaceType

```
GET /api/v1/space-type 200 1.178 ms - 302
GET /api/v1/space-type 500 1.496 ms - 111
 PASS  tests/spaceType.test.js
  GET /api/v1/space-type/
    ✓ should return all space types (12 ms)
    ✓ should return 500 when an error occurs (33 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.632 s
Ran all test suites matching /spaceType.test.js/i.
```

:Fault

```
GET /api/v1/fault 200 1.185 ms - 270
GET /api/v1/fault 404 0.712 ms - 83
GET /api/v1/fault/user/65de252ea3d05e2037bf355b 200 0.430 ms - 350
GET /api/v1/fault/user/65de252ea3d05e2037bf355b 404 0.932 ms - 82
GET /api/v1/fault/65de252ea3d05e2037bf355d 200 0.277 ms - 167
GET /api/v1/fault/1 200 0.230 ms - 167
POST /api/v1/fault 400 1.733 ms - 96
PUT /api/v1/fault/:id 400 0.258 ms - 109
DELETE /api/v1/fault/65de252ea3d05e2037bf355d 404 0.574 ms - 82
 FAIL  tests/fault.test.js
  GET /api/v1/fault
    ✓ should return all faults (12 ms)
    ✓ should return not found error (404) (31 ms)
  GET /api/v1/fault/user/:id
    ✓ should return all fault by user id (2 ms)
    ✓ should return not found error (404) (3 ms)
  GET /api/v1/fault/:id
    ✓ should return a fault by fault id (2 ms)
    × should return not found error (404) (3 ms)
  POST /api/v1/fault/
    × should create a fault (12 ms)
  PUT /api/v1/fault/:id
    × should update a fault (2 ms)
  DELETE /api/v1/fault/:id
    × should delete a fault (5 ms)
```

```
Test Suites: 1 failed, 1 total
Tests:       4 failed, 5 passed, 9 total
Snapshots:   0 total
Time:        0.671 s
Ran all test suites matching /fault.test.js/i.
```

:User

```
GET /api/v1/user 401 3.166 ms - 93
GET /api/v1/user/65de252ea3d05e2037bf355c 200 0.750 ms - 128
GET /api/v1/user/65de252ea3d05e2037bf355c 404 1.249 ms - 81
GET /api/v1/user/123 400 0.582 ms - 94
 FAIL  tests/user.test.js
  GET /api/v1/user/
    × should return all users (17 ms)
  GET /api/v1/user/:id
    ✓ should return a user (35 ms)
    ✓ should return 404 when user not found (4 ms)
    ✓ should return 400 when user id is invalid (2 ms)
```

```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   0 total
Time:        0.678 s
Ran all test suites matching /user.test.js/i.
```