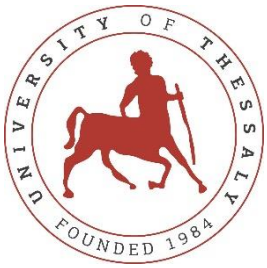


UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

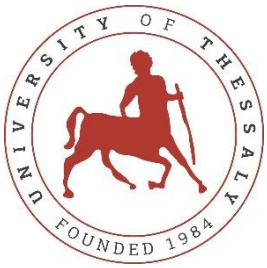
A System for Fault Detection and Management in a Fleet of Commercial Vessels

Diploma Thesis

Gerontidis Georgios

Supervisor: Athanasios Fevgas

September 2025



UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

A System for Fault Detection and Management in a Fleet of Commercial Vessels

Diploma Thesis

Gerontidis Georgios

Supervisor: Athanasios Fevgas

September 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ολοκληρωμένο σύστημα ανίχνευσης και διαχείρισης βλαβών σε στόλο εμπορικών πλοίων

Διπλωματική Εργασία

Γεροντίδης Γεώργιος

Επιβλέπων: Αθανάσιος Φεύγας

Σεπτέμβριος 2025

Approved by the Examination Committee:

Supervisor	**Athanasios Fevgas** Laboratory Teaching Staff member, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Panagiota Tsompanopoulou** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Eleni Tousidou** Laboratory Teaching Staff member, Department of Electrical and Computer Engineering, University of Thessaly

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελούν αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλουν οποιασδήποτε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχουν έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Δηλώνω επίσης ότι τα αποτελέσματα της εργασίας δεν έχουν χρησιμοποιηθεί για την απόκτηση άλλου πτυχίου. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο Δηλών

Γεροντίδης Γεώργιος

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The Declarant

Gerontidis Georgios

Acknowledgments

This thesis marks the closure of a long journey, one filled with demanding work, setbacks, small victories, and many lessons. I am grateful for every step along the way.

First and foremost, I would like to thank my supervisor, Dr. Athanasios Fevgas, for his steady guidance and unwavering support. His mentorship pushed me to think more clearly, build more robustly, and grow into a better engineer. I am equally thankful for his thoughtful advice about life after graduation and his ideas on it.

I would also like to thank the entire examination committee Panagiota Tsompanopoulou and Eleni Tousidou for their valuable time, for their comments and the assistance on this thesis.

I am also indebted to Mr. Dimitrios Fountoukidis for his invaluable insights into the maritime domain. His long experience, both at sea and with technical applications, shaped the project's direction and sharpened its relevance.

My sincere thanks go to my fellow students and future colleagues. Our conversations, shared ideas, and everyday collaboration enriched this work and, more importantly, helped me mature as both a scientist and an engineer.

To my friends, thank you for your patience, your humor, and your constant encouragement. Your support made the difficult stretches lighter and the good moments brighter.

Finally, to my whole family, my grandparents, aunts, uncles, cousins, my brother Giannis, and my sister Athina, thank you for believing in me. A special and heartfelt thanks to my parents Niki and Theodorhs, who really are the true heroes of this story. Your unwavering support, in every practical and emotional sense, made all this effort possible. The way I grew up thanks to you, taught me above all to be a good human before anything else. For that, and for everything else, I am deeply grateful.

Διπλωματική Εργασία

Ολοκληρωμένο σύστημα ανίχνευσης και διαχείρισης βλαβών σε στόλο εμπορικών πλοίων

Γεροντίδης Γεώργιος

Περίληψη

Η παρούσα διπλωματική εργασία εμπίπτει στον επιστημονικό τομέα της ναυτιλιακής τεχνολογίας και έχει ως στόχο την ανάπτυξη ενός ολοκληρωμένου συστήματος για την ανίχνευση και πρόληψη βλαβών σε εμπορικά πλοία μέσω μιας εφαρμογής αξιολόγησης που βασίζεται σε ερωτηματολόγια. Το σύστημα υποστηρίζει τρεις τύπους χρηστών: διαχειριστές στόλου, αρχιμηχανικούς και μέλη πληρώματος. Ο αρχιμηχανικός μπορεί να δημιουργεί, να επεξεργάζεται και να διαγράφει ερωτηματολόγια πολλαπλής επιλογής, να παρακολουθεί τις απαντήσεις του πληρώματος και να δημιουργεί στατιστικά στοιχεία. Τα μέλη του πληρώματος απαντούν στα ερωτηματολόγια αξιολογώντας την κατάσταση των συστημάτων του πλοίου, με την επιλογή να προσθέσουν φωτογραφίες και σχόλια. Η εφαρμογή Android συγχρονίζεται με την πλατφόρμα διαχείρισης μέσω API και προσφέρει λειτουργικότητα εκτός σύνδεσης για χρήση εν πλω. Οι διαχειριστές στόλου μπορούν να δημιουργήσουν τα πλοία και τους λογαριασμούς των χρηστών τους και να παρακολουθούν οποιαδήποτε διαδικασία λαμβάνει χώρα στο πλοίο. Η υλοποίηση βασίζεται σε σύγχρονες τεχνολογίες, όπως React, Node.js, MySQL και Firebase, με αρχιτεκτονική που επιτρέπει τον ασφαλή συγχρονισμό μεταξύ τοπικών και cloud βάσεων δεδομένων, συμβάλλοντας στην αποτελεσματική συντήρηση και ασφάλεια των πλοίων στη σύγχρονη ναυτιλία.

Λέξεις-κλειδιά: ναυτιλιακής τεχνολογίας, ανίχνευση και πρόληψη βλαβών, ερωτηματολόγια, API, React, Node.js, MySQL,Firebase βάσεις δεδομένων

Diploma Thesis

A System for Fault Detection and Management in a Fleet of Commercial Ships

Gerontidis Georgios

Abstract

This thesis falls within the scientific field of maritime technology and aims to develop an integrated system for fault detection and prevention in commercial vessels through a questionnaire-based evaluation application. The system supports three types of users: fleet administrators, chief engineers, and crew members. The chief engineer can create, edit, and delete multiple-choice questionnaires, monitor crew responses, and generate statistics. Crew members respond to the questionnaires by assessing the condition of ship systems, with the option to add photos and comments. The Android application synchronizes with the management web platform via an API and offers offline functionality for use at sea. The fleet administrators manage to create their ships and their users accounts and can watch any process that happens in the ship. The implementation is based on modern technologies including React, Node.js, MySQL, and Firebase, with an architecture that enables secure synchronization between local and cloud databases, contributing to efficient vessel maintenance and safety in contemporary shipping.

Keywords:

Maritime Technology, Fault Detection, Commercial Vessels, Android, API, Firebase, SQL, React, Questionnaires, Ship Crews, Fleet, cloud databases

Table of Contents

List of Figures

List of Shapes

List of Tables

Abbreviations

i.e.	id est
e.g.	exempli gratia
etc.	et cetera
API	Application Programming Interface
ML	Machine Learning
Db	Database
SQL	Structured Query language
App	Application
UI	User Interface
CRUD	Create Read Update Delete
DAO	Data Access Object
REST	Representational State Transfer
URI	Uniform Resource Identifiers
SDK	Software Development Kit
IoT	Internet of Things
SCADA	Supervisory Control and Data Acquisition
HMR	Hot Module Replacement
ESM	ECMAScript modules
MVVM	Model-View-View Model
KPI	Key Performance Indicator
SVMs	Support Vector Machines
MLP	Multilayer perceptron
PdM	predictive maintenance
KNN	k-Nearest Neighbors
RF	Random Forest

Chapter 1 Introduction

In the modern era technology plays a vital role in our everyday lives. As it evolves very quickly, engineers should likewise try to adapt in order to modernize and improve processes for dealing with practical challenges. An example of a sector that can benefit significantly from technological advancement is the shipping industry.

Despite widespread digitalization across many industrial sectors, the shipping sector especially in the inspection and maintenance part, often relies on handwritten records or non-automated processes [1] This reliance not only makes the current technological capabilities slower but also frequently results in inefficient maintenance management and limited capacity for fault prediction in commercial vessels. The consequences include operational delays, increased running costs and sometimes safety risks [1]

1.1 Target of the system creation

This thesis tries to confront these shortcomings by proposing the development of a digital application for managing questionnaires aimed at preventing and recording faults in commercial ships. The application, exploits modern software development technologies, database systems and APIs for data synchronization. It is designed to support three user roles which are the administrators, the chief engineers, and the crew members. It includes an Android application with offline capabilities, enabling usage even in environments without internet connectivity, that are very common for ships, a server side application that processes questionnaires for an entire vessels' fleet and a web dashboard that facilitates the management of the fleet.

Through this implementation, the project tries to modernize the shipping industry, aligning it with the demands and expectations of the current technological situation [2]

1.2 Thesis Structure

The thesis is organized into seven chapters that contribute to a better understanding of, development of, inspection of, and detection of possible failures in fleets of commercial ships. The thesis is structured as follows:

Chapter 2 shows the system design, from the system's structure to the selection of the technologies used.

Chapter 3 introduces the creation of the data models of the system.

Chapter 4 presents how the Android application of the system was created.

Chapter 5 describes the web dashboard and how it manages fleets, questionnaires, etc.

Chapter 6 provides information about the web dashboard backend, the forming of the API connection and details about synchronizing the system.

Chapter 7 conducts research on a future ML model and how it could work within the system.

Chapter 8 summarizes the entire thesis project, presents the results, and suggests ways to improve the project in the future.

Chapter 2 System Design and Requirements

2.1 System Structure

2.1.1 Introduction

The system design process was initiated with an assessment of the needs of users in the commercial shipping sector. The primary objective of this initiative was to enhance fault prevention and management by modernizing and automating workflows. The approach adopted was predicated on the principle of continuous data flow from the vessel to the management system, taking advantage of cloud technologies and mobile applications.

First, an exhaustive analysis was conducted to ascertain deficiencies in the sector, with a particular focus on conventional practices for recording and managing technical reports. The analysis showed that the part of automation and digitization was the cause of many of the sector's challenges, indicating the necessity to modernize inspection procedures and ship condition monitoring.

Collaborating with Mr. Fountoukidis from Devmat Solutions PC proved to be a pivotal element in comprehending the market's requirements, as his considerable experience in the shipping industry and knowledge of the requirements of the commercial maritime sector contributing with invaluable insights (having contributed in programs of the Devmat Solutions PC company like Ocean Anax Maritime ERP). Discussions revealed the necessity for a practical, mobile-accessible solution that could be used directly by every crew member.

The proposed solution is predicated on the development of an Android application with offline capabilities, which would enable crew members to complete questionnaires and store their responses locally. After the restoration of connectivity, the data will undergo automatic synchronization with the central system. Furthermore, the implementation of a dedicated management interface for each ship's chief engineer was deemed essential. This interface will facilitate real-time monitoring of the vessel's technical condition, oversight of crew submissions, and management of inspection processes.

To complete the system, a web-based management dashboard has been designed for shipping companies to use. This platform enables the creation, organization and monitoring of an entire fleet, and provides access to statistical analyses of questionnaire responses and facilitating the identification of potential malfunctions. Such insights support long-term planning and preventive maintenance.

Given that the system depends on interaction between the mobile application and the management platform, an intermediate communication layer was needed and was covered by using a RESTful API. This API conducts the synchronization between the mobile device's local database and the central server database, while guaranteeing both data security and reliability (Figure 2. 1).

Figure 2. 1 System Structure

So for the full project plan a comparison research for possible technologies frameworks and tools needed to be made.

2.2 Comparison of technologies

2.2.1 Android (Kotlin + Firestore)

For the Android application, Firestore DB, was evaluated against two other popular solutions the SQLite, and MySQL db's. SQLite was deemed unsuitable because it was more appropriate for smaller applications and lacks built-in synchronization capabilities (although a version of it was used inside the android application). Subsequently, a decision had to be made between MySQL and Firestore DB, with the latter being chosen due to its greater compatibility and real-time support. So, as shown (Figure 2. 2), firebase was selected for the android app db [3] [4] [5]

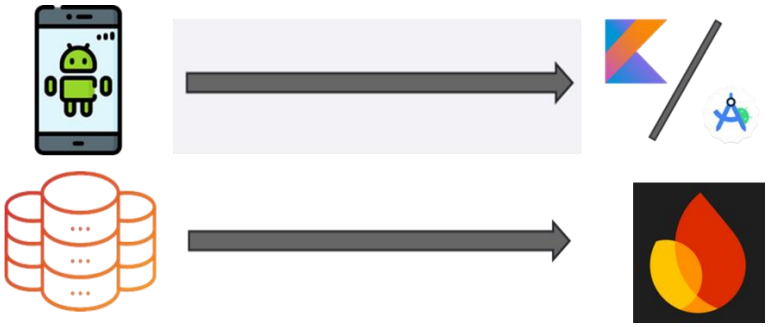


Figure 2. 2 Selection Kotlin and Firestore db for the android app

To provide clearer insight, the tables below present a detailed comparison of the tools evaluated for use (Table 2. 1,Table 2. 2).

Feature	Firestore (NoSQL)	SQLite (SQL)
Type	Cloud, NoSQL	Local, SQL
Real-time support	Yes	No
Offline support	Yes (built-in)	Partial
Scalability	High	Limited
Best Use Cases	Cloud apps, real-time collaboration, mobile apps	Local storage, offline-first apps, lightweight single-user projects

Table 2. 1 Comparison between Firestore (NoSQL) and SQLite databases

Feature	Firestore	MySQL
Type	NoSQL, document-based	RDBMS
Real-time support	Yes	No (requires polling)
Offline support	Yes	Possible but requires extra setup (e.g. local replication)

Feature	Firestore	MySQL
Scalability	Automatic	Manual
Use cases	Cloud apps, real-time collaboration, mobile apps	Complex relational data, enterprise systems, reporting-heavy apps

Table 2. 2 Comparison between Firestore and MySQL databases

Regarding the choice of programming language in Android Studio, the option of using Java was considered. However, Kotlin was selected as a more modern language that has become preferred for Android development in recent years, offering high performance with very simple syntax. The comparison is presented below (Table 2. 3) and the selection is presented in the (Figure 2. 2).

Parameter	Kotlin	Java
Syntax	Concise	Verbose
Null safety	Built-in	Not native
Modernity	High	Established
Performance	High	High
Use cases	New Android projects	Legacy applications

Table 2. 3 Comparison between Kotlin and Java programming languages

2.2.2 Backend (Node.js + Express)

The most popular development choices for the backend were Node.js and Django since they are widely spread and known for this kind of development. Node.js works faster than Django for real-time tasks because it handles many requests without waiting. Django is good for simple apps but slower with heavy data flow. For this project Node.js was better for its speed and ease in building real-time APIs (Figure 2. 3). For better insight (Table 2. 4) shows exact comparison between them.

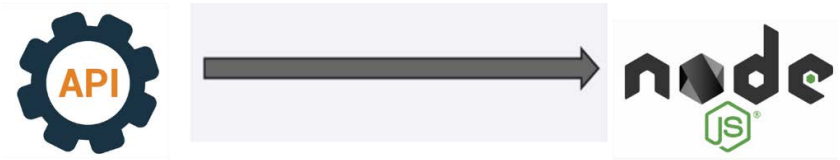


Figure 2. 3 Node.js the API programming language

Parameter	Node.js/Express	Django
Real-time performance	Very high	Moderate
Learning curve	Gentle	Gentle to moderate
Use cases	APIs, microservices	CRUD applications
Scalability	High	High

Table 2. 4 Comparison between Node and Django programming languages

2.2.3 Web Dashboard (Node.js/Vite/React + MySQL)

A web dashboard is an interface that presents data, metrics, and controls in a clear and organized way, allowing users to monitor and manage key information in real time. Building a dashboard typically requires a responsive frontend, fast data updates, and seamless communication with a backend. Thats why a fast building tool needed to be chosen for a quick build and a library needed to be added that would make a well-structured UI.

To avoid delays knowing Vite is significantly faster than Webpack as a building tool, offering near-instant hot reloads and reduced build times, Vite was the clear choice for a frontend tool (Table 2. 5). It was also certain that React was going to be used in this project because it has many libraries , many choices for creating a decent UI (Figure 2. 4).

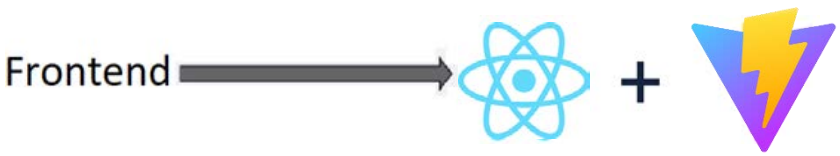


Figure 2. 4 Frontend using React.js and Vite.js

Parameter	Vite	Webpack
Build speed	Very high	Lower
Hot reload	Almost instant	delay
Ease of use	High	Moderate

Table 2. 5 Comparison between Vite and Webpack build tools

After conducting research, MySQL seemed to be more suitable for the projects backend. Although PostgreSQL was considered whether it might offer any advantages, the decision ultimately came easily, especially given a prior experience with MySQL (Table 2. 6).

Figure 2. 5 MySQL as the Web app db

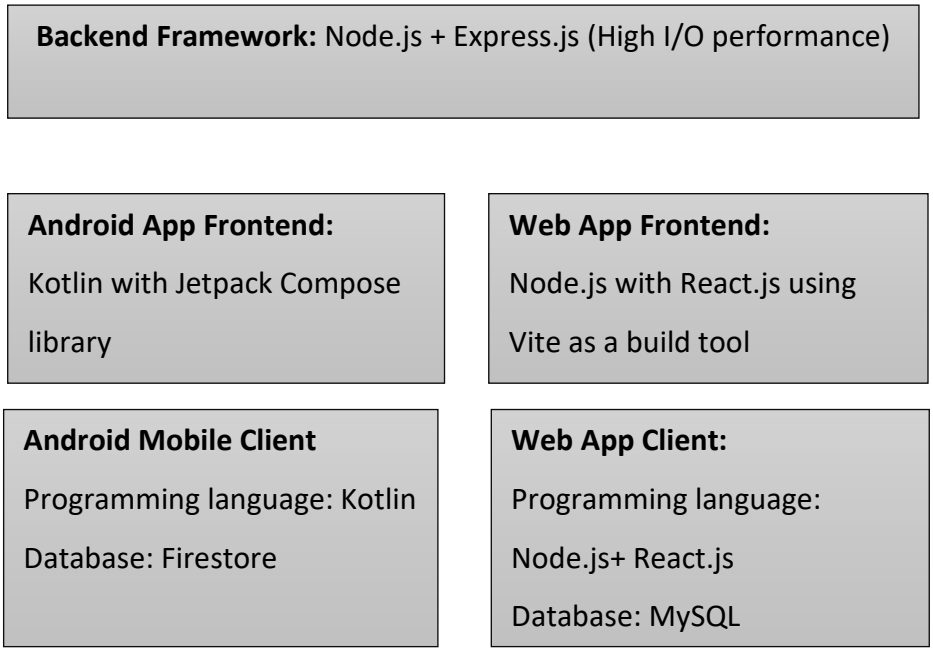
Feature	MySQL	PostgreSQL
Installation ease	High	Moderate
JSON support	Limited	Advanced
Community	Large	Large & specialized
Learning curve	Gentle	Steeper

Table 2. 6 Comparison between MySQL and PostgreSQL db's

For a collective view and a better understanding of the choices that were made for the implementation of this system the (Table 2. 7) presents the result of this Comparative Study (Shape 2. 1).

Component / Decision Area	Option 1	Option 2	Selected Technology	Key Reason for Selection
Android Database	Firestore (NoSQL, Cloud, Real-time)	SQLite (Local SQL)	**Firestore**	Real-time sync, offline support, scalable for many users ([3] ,[4])
Database Alternative	Firestore (NoSQL)	MySQL (RDBMS)	**Firestore**	Automatic scaling, native real-time updates ([3] , [5])
Android Language	Kotlin	Java	**Kotlin**	Modern syntax, null-safety, better suited for new Android projects ([6])
Backend Framework	Node.js + Express	Django (Python)	**Node.js + Express**	High performance for real-time apps, easy to learn ([7])
Front-end Build Tool	Vite	Webpack	**Vite**	Faster builds, instant hot reloads ([8] ,[9])
Web DB Choice	MySQL	PostgreSQL	**MySQL**	Easier setup, simpler learning curve, sufficient for project needs ([3] ,[10])

Table 2. 7 Final Comparison Results and Selections



Shape 2. 1 Architecture of Main Technologies selected

The architecture of the main technologies that was developed after the comparative study of them is appeared in the shape above (Shape 2. 1).

Building on this foundation, the next stage of the project was to design the database. The goal was to support the required functionalities, optimize data storage, and accurately represent the relations between the system's entities.

Chapter 3 Database Model

3.1 Introduction

The system had to have a database Model that could fulfill both the requirements of the fleet management web application and the Android application. The schema was constructed as an ER of a MySQL schema, that would be the schema for the server side (Figure 3. 1). The android application would have the same entities and attributes as the

MySQL schema but it would be structured and organized differently based on collections, documents and subcollections (Table 3. 1). Collections are like tables, the top-level group of documents. Each document has a unique ID and contains its fields. Fields can be simple values like string, int etc., and more complex like arrays, maps, nested objects etc. A document can also have its own subcollection .

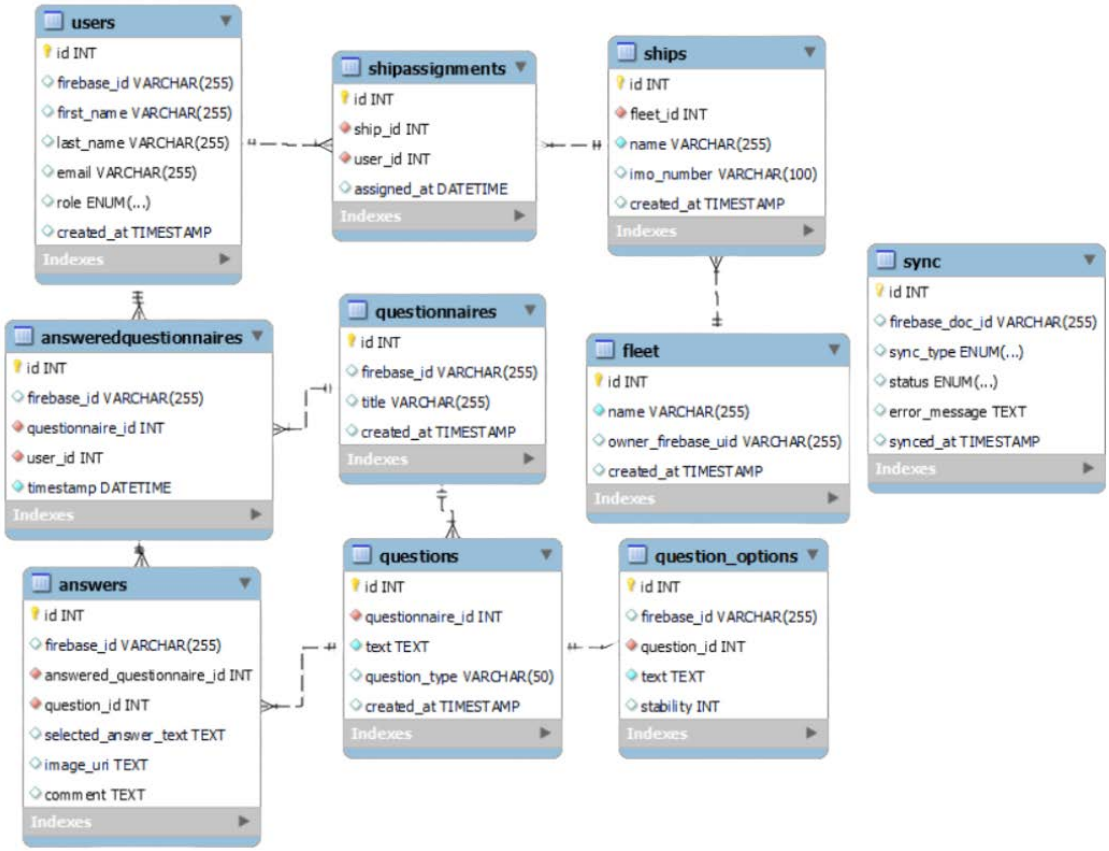


Figure 3. 1 ER Diagram database Schema

users	ships	fleets
{uid}	{shipId}	{fleetId}
└ email	└ fleetId	└ name
└ firstName	└ name	└ ownerFirebaseUid
└ lastName	└ imoNumber	└ createdAt
└ role	└ createdAt	└ mysqlId
└ createdAt	└ mysqlId	

└ shipId └ mysqlId		
answeredQuestionnaires {submissionId} └ questionnaireId └ shipId └ userId └ username └ timestamp └ mysqlId answers (array) [k] └ questionId └ selectedAnswerText └ imageUri └ comment	questionnaires {questionnaireId} └ title └ shipId └ stability └ createdAt └ mysqlId questions (array) [i] └ id └ text └ stability └ questionType answers (array) [j] └ id └ text └ stability	shipAssignments {assignmentId} └ userId └ shipId └ assignedAt └ mysqlId

Table 3. 1 Data Model Diagram

3.1.1 Database Schema Description

The database is designed to be suitable for needs of the maritime fleet management platform. The schema is structured into multiple tables, that have a distinct functional purpose while maintaining referential integrity through foreign key constraints.

3.1.2 Users Table

The Users table is used as the central directory of all individuals who interact with the platform, including administrators, chief engineers, and crew members. Administrators

are the company leaders that manage the fleet manager platform. Chief engineers are the chief mechanics assigned to a ship to oversee the Inspection. Crew Members are the ones that give reports by filling the questionnaires. Each record stores essential profile details and assigns a unique Firebase ID to every user, through his addition to the system from an admin at the web application. This identifier not only ensures secure authentication but also supports direct integration with Firebase's identity management services, enabling smooth coordination between the web platform, mobile application, and backend systems.

3.1.3 Fleet and Ships Tables

The fleet table represents the highest organizational unit in the system, as it consolidates multiple ships under a common entity. Each fleet may comprise one or more ships, the number of which is documented in the ships table. The ship table stores ship-specific details, including the ship's name and IMO number. Furthermore, it facilitates a foreign key relationship, which links each ship to its parent fleet.

3.1.4 Shipassignments Table

The shipassignments table is responsible for the management of crew assignments, acting as a junction table between users and ships. This enables the tracking of user assignments to specific vessels and the determination of the temporal context of such assignments, thereby facilitating dynamic crew management.

3.1.5 Questionnaires , Questions and Question_options Tables

The inspection and survey functionality are supported by three core tables: questionnaires, questions, and question_options. Questionnaires serve as the primary survey instrument and comprise multiple questions. The inquiries posed may assume the form of text-based, multiple-choice, or image-based questions. The question_options table is used as the repository for available answer choices.

3.1.6 AnsweredQuestionnaires and Answers Table

Completed inspections are saved in the answered_questionnaires table, which records the answered questionnaire, the submitting user, and the timestamp. Also, the responses to each question are saved in the answers table, which supports a user's comment, choice selections, and image attachments.

3.1.7 Sync Table

The sync table records all synchronization events between the Firestore database and the MySQL database. This log includes the synchronization type, status, and any errors encountered.

3.1.8 Entity Relationships in the Database

This is a relational database and this way it has entities that are connected through relations between them. These relations allow the system to link operational data, inspection records, and user activity for the application.

The **users** table is central to the system and is one of the most important as it connects to lots of other entities. Each user can be assigned to a ship through the **shipassignments** table. This is a *many-to-many* relationship between users and ships, as one user may serve on different ships over time, and one ship may have multiple crew members.

One fleet has many ships and that's why it has an *one-to-many* relationship between **fleet** and **ships**: a single fleet can manage multiple vessels, but each vessel is associated with only one fleet.

Inspection-related data follows a hierarchical relationship. The **questionnaires** table are in the top-level structure, which connects to **questions** table in a *one-to-many* relationship. Each question has four answer choices in the **question_options** table. That way another *one-to-many* connection is created.

When a questionnaire is completed by a user, the action is logged to the **answeredquestionnaires** table, which connects back to both the users and questionnaires tables. This creates a *many-to-one* relationship in both cases, as many completed questionnaires are linked to one user and one questionnaire.

The responses go to the **answers** table, which connects to the answeredquestionnaires table as well as the questions table. This way each recorded answer can be traced back to both the question it belongs to and the specific inspection in which it was recorded.

Finally, the **sync** table operates independently in terms of core business logic but is related conceptually to all data entities. It logs the synchronization process between the Firestore mobile data source and the MySQL backend, providing an audit trail for data consistency and system reliability.

Chapter 4 Android application

4.1 Introduction

The android application was the first inspiration of this thesis. The app needed a name and a logo, and it was called SafeShips, because this app was made to establish the safety of the ship (The logo appears beside this text Figure 4. 1).This app was designed to detect quickly malfunctions and planned to support two Android application views for two types of users the “user” and the “chief”[11] The “user” would be the crew members of the ship that were supposed to fill the handwritten Inspection in the past , but now with this app they would fill the application questionnaire, having the opportunity to attach photos and comments to each answer. The “chief” on the other side would be the chief engineer of the ship, that creates reviews and sees stats of the answered questionnaires.

Figure 4. 1 SafeShips logo

The app supports multiple fleet companies that have multiple ships. It was also essential for the app to have an offline functionality since the signal is often lost in the open sea.

4.2 Technologies used

The key technologies and components that were used for the implementation of the Android app :

- **Kotlin:** The app was developed in Kotlin, which is the official recommended language for Android. Kotlin is very simple and has strong safety features, which helped increase development efficiency and reduce bugs. Its clear code structure leads to fewer lines of code compared to Java [12]

- **Android Jetpack Compose Libraries:** Android Jetpack components utilized to implement an efficient architecture. The Room persistence library was used for local data storage (providing an abstraction over SQLite to store questionnaire data and pending submissions), and ViewModel + LiveData**/**Flow for managing UI state in a lifecycle-aware manner. The WorkManager library was a crucial choice for scheduling background synchronization of data when internet connectivity is available. WorkManager provides a reliable way to defer API calls and retry them, even if the app is closed or the device restarts, which is ideal for our offline-sync requirements.

- **Firebase Services:** The application integrates directly with Google Firebase for several cloud-based features (Figure 4. 2). Firebase Authentication is utilized to manage user login and credentials securely, simplifying the auth implementation by using Firebase's hosted identity management. Additionally, Firebase Cloud Storage is used to upload and store photos taken within the app. This choice was made because Firebase Storage seamlessly handles large binary file uploads and provides caching. Thus crew photos are first saved locally and later uploaded to Firebase when the smartphone is online. At this point the backend retrieves photos via secure URLs. Firebase's offline capabilities were also exploited for example, if using Cloud Firestore (a NoSQL cloud database) to store some temporary configuration data , it automatically caches data on the device so that reads can succeed offline [13] These Firebase components work in tandem with the main backend, enriching the app with cloud synchronization features without requiring complex custom server code for each function.

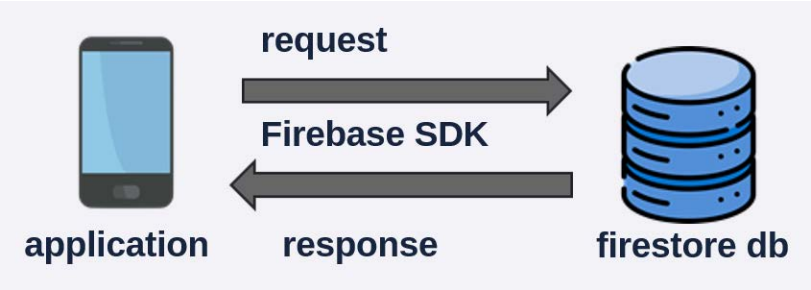


Figure 4. 2 Communication between application and firestore DB

- **Security and Data Protection:** The app handles connections using SSL/TLS for all API communication, which is then routed through HTTPS endpoints on the Node.js server. User credentials and authentication tokens are stored using Android's secure storage mechanisms, such as EncryptedSharedPreferences and the Android Key Store, to prevent sensitive information leakage. Firebase Auth tokens issued by the backend are cached and included in request headers for authorization to protect endpoints from unauthorized access. A two-level security system was implemented using the combination of Firebase Auth and server-side session validation. Firebase verifies identity, and the server maintains session roles and permissions, ensuring that only authorized crew members can submit data for a particular vessel.

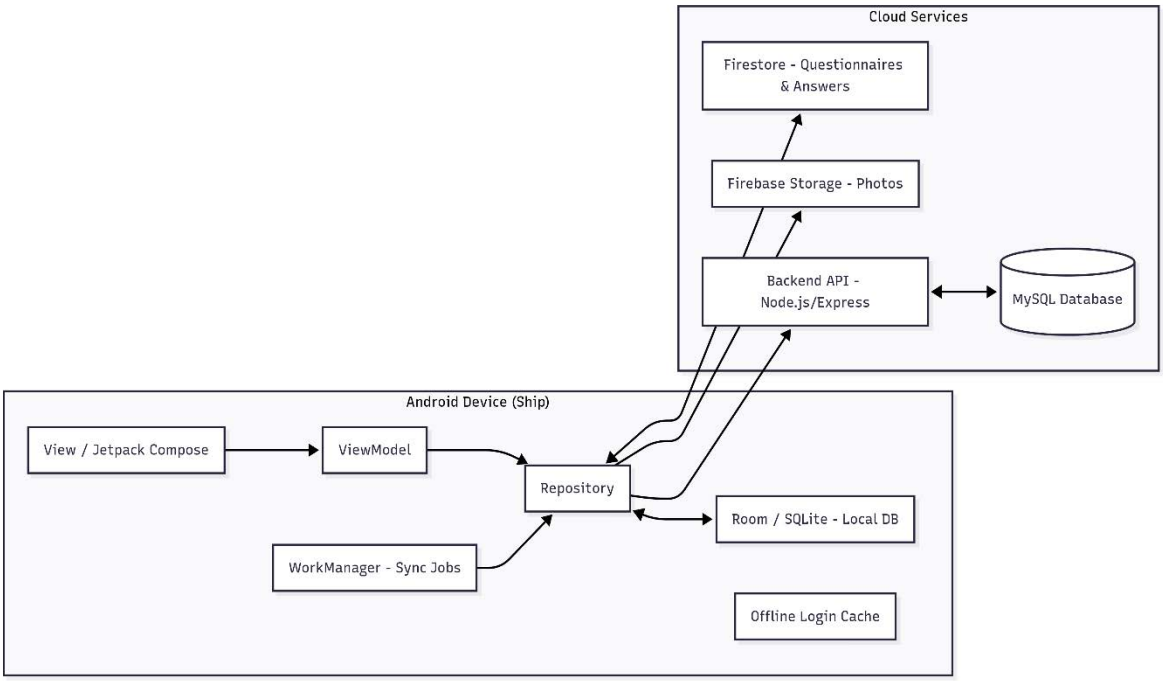
4.3 Architecture

The Android app was built to work offline, handle data safely, and link with the backend when the network is available. The idea was to keep things reliable, because on a ship network connection is not always there. To make this possible, the app was split into layers. Each layer does its own part. The layers handle information flow from the user to the backend. This way the app keeps working even if the internet drops for a long time Shape 4. 1.

Essentially, the app architecture forms a two-way bridge: it uplinks crew inputs to the server, and downlinks new content or instructions to the crew. This two-way communication is designed to be robust and near-real-time whenever connectivity is present, but tolerant to disconnections.

The result is an application that can function independently offline yet seamlessly synchronize with the fleet management system. By following an offline-first architecture with local storage and carefully orchestrated sync (using WorkManager and Firebase

messaging), the app achieves a high level of reliability. This architecture directly contributes to the project's fault detection goals: data about potential issues is captured on-site and relayed efficiently, ensuring decision-makers have the necessary information with minimal delay.



Shape 4. 1 System Context Diagram

4.3.1 Layered architecture

The structure follows the MVVM pattern (Shape 4. 2). The first part, the View, is the interface. It is made with Activities, Fragments, or Jetpack Compose functions. Its role is limited: it shows forms and records taps, text, or other inputs. The View is not supposed to handle the logic of the data. It is kept simple on purpose. The ViewModel sits in the middle. Each screen has its own ViewModel. It asks the repository for data, changes the data if needed and then gives it back to the View. It also reacts to actions from the user. For example, when someone saves a questionnaire, the ViewModel handles that. LiveData as an observable data holder class and Flow as a Kotlin Coroutine stream API are used so the interface updates when the data changes. The Model layer is about data and communication. Room is used here as an easier way to work with SQLite. The repositories, are design patterns that act as a single source of truth for data, determine if the app should read from local storage or fetch fresh data from the server. Retrofit is used to talk to the

backend API. The offline-first idea is built into the repository, so the app always tries to use local data first and then sync later when a connection is available.

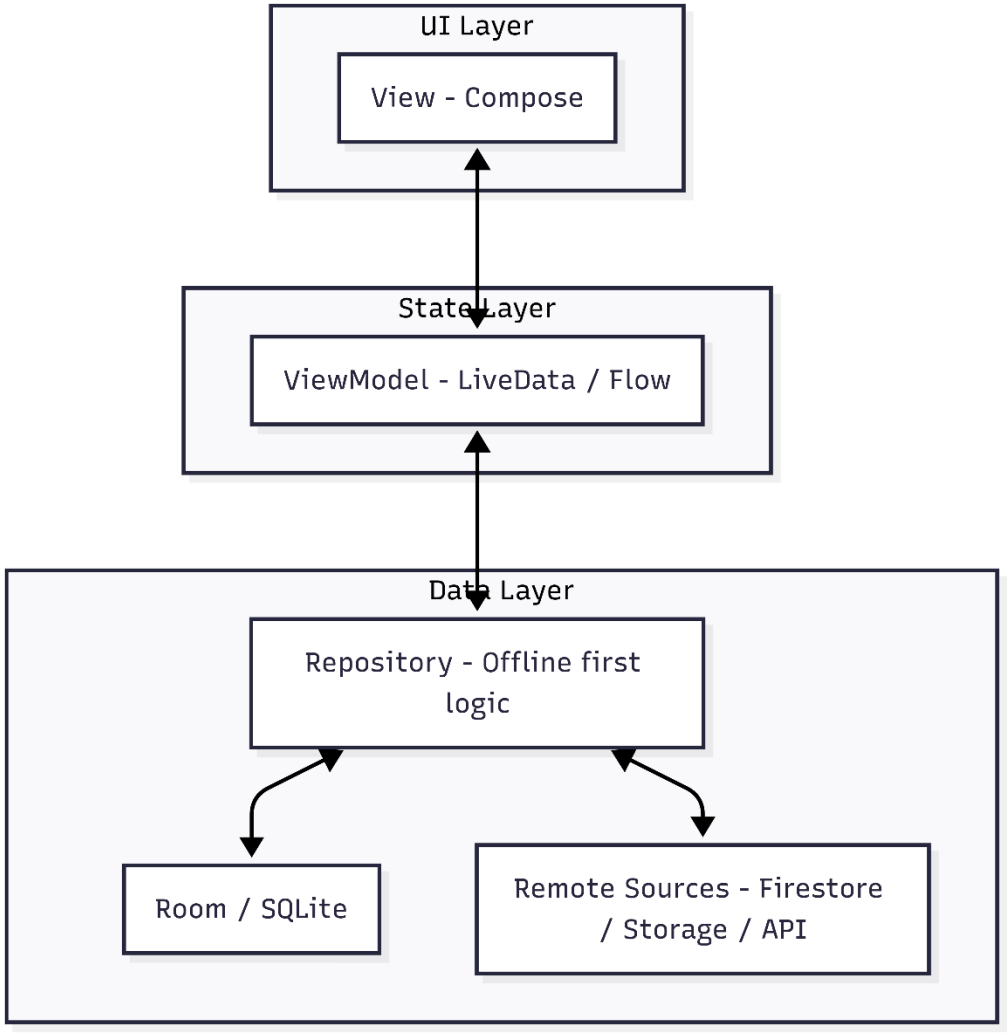
4.3.2 Local storage & offline data

Local storage is important. The app keeps a SQLite database through Room Library as the main source of truth. Every time the app starts, or when it downloads updates, it saves important information locally, that includes the questionnaires. This setup lets the crew continue using the app without needing internet connection. New or unfinished data stays in the database until the next sync sends it to the server.

By doing so, the app can read and display data without needing a network [14] According to Android's official guidance, an offline-first app should be able to perform read operations entirely from a local data source [14] and our architecture follows this principle. Each questionnaire response entered by the crew is saved as a local record marked with a "pending" status. Binary data like photos are saved as files in local storage (in a private app directory) with their paths referenced in the local DB. This ensures that even large media attachments are available for viewing on the device immediately after capture and are not lost if the app closes before upload.

To manage local data consistency, the app designates certain operations that trigger only when it restores network connection. For example, login authentication is generally an online operation (to verify credentials with the server or Firebase Auth). However, to support use cases where the device might be offline at login time (e.g.the ship's internet is down at the time of daily inspection), we implemented an offline login cache: if a crew member has logged in successfully before, the app securely stores a token or hashed credential such that it can allow access in offline mode (with limited functionality) and later re-validate when back online. This approach, while used cautiously, prevents scenarios where crew are completely blocked from using the app due to lack of internet. It trades a bit of strict security for practical usability — a reasonable compromise

given the controlled environment (the device is typically company-issued and used by authorized crew only).



Shape 4. 2 Layered MVVM Diagram

4.3.3 Synchronization mechanism

A cornerstone of the architecture is the sync service that handles data exchange with the backend. We utilized Android's WorkManager to schedule job's synchronization, encapsulating the logic in a SyncWorker class. The synchronization process works as follows:

1. **Triggering conditions:** Sync can be triggered either by a periodic schedule (e.g., attempt to sync every hour), or connectivity status changes (e.g., device goes from offline to online). WorkManager allows constraints to run a task only when certain conditions are met (like “requireNetwork=ANY”). We take advantage of that by enqueueing a sync task that will run as soon as a network is available.

2. **Data batching:** The sync process batches all pending operations. For example, if three questionnaires were filled offline, and two new ones were fetched from the server in the meantime, the sync job will handle all these in one go. The repository gathers all local records marked “pending upload” and calls the appropriate API endpoints (e.g., a POST request for each new submission). Similarly, the API will check if any new questionnaires or updates are available from the server by calling a “get updates” endpoint.

3. **Conflict avoidance:** Because the app mostly deals with user-generated reports that don’t overlap (each questionnaire submission is independent), conflicting scenarios are avoided. However, we still consider potential conflicts in data synchronization. In worst-case, if a conflict is detected (e.g., a question was deleted on server but an answer exists locally for it), the app will flag it and not overwrite blindly. These situations are rare due to the systems operation procedure (crew only fill data, they don’t modify the questionnaire structure), but the architecture includes provisions for conflict resolution to ensure data integrity.

We plan to use the Cloud Messaging firebase in a future version to send a silent push notification to devices when certain events occur. Receipt of such a message wakes the app (via a broadcast receiver or FirebaseMessagingService) and prompts it to initiate a sync for that specific item. This ensures timely updates: for example, if a new critical checklist is issued ahead of a port state inspection, the crew’s device can receive it as soon as they have any connectivity, rather than waiting for the next periodic sync.

During synchronization, network reliability is a significant issue. Therefore, the architecture employs a retry logic. If an upload fails due to a lost connection or server timeout, the WorkManager task will use an exponential backoff and retry automatically [14] We also designed the API idempotently: repeated submission of the same data will not create duplicates on the server (each submission has a unique ID or client timestamp, thus the backend ignores duplicates). This design choice prevents the

"double-reporting" problem in cases that a sync is performed twice due to retries or uncertain acknowledgments.

4.3.4 Integration with central platform

On successful sync, data flows into the central Node.js/MySQL system. The architecture ensures that once the sync is done, the local records are marked as synced (and could be purged or archived locally if storage is a concern, though typically we kept them for history). Any new data from the server (like updated templates) are saved to local DB and immediately reflected in the UI via LiveData updates.

4.4 Functionalities

The Android application provides a range of functionalities carefully tailored to support vessel maintenance operations and fault prevention workflows. In this section, we detail the major features and how they work from a user's perspective, highlighting how each functionality supports the crew's tasks:

- **User Authentication (Login/Logout):** When a crew member launches the app, they are presented with a login screen. The login requires entering credentials (such as a email, password , fleet and ship IMO [15] This ensures that only authorized personnel can access and submit data, which is important for data integrity and accountability. The app uses Firebase Authentication to validate credentials online. On successful login, an authentication token is stored for subsequent requests. The app also caches the user's basic profile (name, role, assigned vessel) for offline display. If the vessel is offline and a user has previously logged in, the app will allow access using the cached credentials, as described in the Architecture. This login function also ties into role-based content: an engineer on Vessel A will, upon logging in, only receive questionnaires relevant to Vessel A, as determined by the server.

- **Questionnaire Download and Access**: Once logged in, the user can download the latest questionnaires from the central system. The app's home screen typically lists available questionnaires that the crew needs to perform. These could be routine

daily checklists (covering engine parameters, safety equipment checks, etc.). Each questionnaire entry shows a title (e.g., “Engine Room Inspection”). Crew can tap a questionnaire to open it. The questionnaire screen displays a series of questions or checkpoints regarding various ship systems. Questions are multiple-choice (categorizing an observed issue), with a choice of an open-ended text for comments. The crew can move through the questions, providing answers as they perform the inspection procedure. Templates are stored in the local database so even if the ship goes offline after downloading, the questionnaire content is fully accessible.

- **Photo Capture and Upload:** A crucial functionality is the ability to attach photographs to a report. In any question or section of the form where a visual record is useful, the app offers a camera integration. Tapping “Add Photo” triggers the device camera to take a picture. After capturing, the photo is shown as a thumbnail in the form and saved to the device’s local storage. The user can typically add multiple photos but only one to each question, each stored and listed. They can also add a short caption or comment for each photo if needed (e.g., “Oil leak near pump”). All images are stored offline first – meaning the image file is saved on the device and a reference is added to the form data. This offline-first approach for media ensures that even large images are handled without needing immediate upload (which could fail or take long at the sea). Later, during synchronization, these images are uploaded to the Firebase Storage. The ability to include pictures greatly improves the accuracy and richness of maintenance reports [16] – for example, shore engineers can see the exact part that was corroded or the reading on an instrument, reducing ambiguity. The app interface shows any attached images and enables reviewing them (e.g., tapping the thumbnail opens a full-screen view). It also allows deletion of an image before submission in case the user took a bad shot or attached the wrong file. All photos and data remain stored on the device until successful sync: the crew does not need to manually manage files.

- **Adding Comments and Annotations**: In addition to structured question answers, the app provides free-text comment fields for crew to record observations or suggestions. Each question typically has an optional comment box (“Comment”)

where the user can type additional details if the pre-defined answers don't capture everything. These comments are important for context – e.g., if a reading is abnormal, the engineer might note "reading higher than normal, will monitor closely". The app uses multi-line text fields for this purpose. It supports offline entry and stores these with the rest of the form data. The goal is to make capturing qualitative data as easy as possible, since those narrative comments often provide insight to fault causes that numeric data alone might not.

- **Offline Saving:** A fundamental feature is that the app allows saving work in progress offline. Crew might not complete a long inspection in one go; they might perform it over several hours or shifts. The app enables saving a partially filled questionnaire as a draft locally. Users can explicitly save a draft, or the app auto-saves periodically. This way, crew can resume later. Once a questionnaire is fully filled out, the user marks it as complete (by hitting a "Submit" button). If online at that moment, the app will attempt an immediate sync (submitting the data to the server). If offline, the submission is stored and tagged as "Pending Sync".
- **Synchronization (Upload/Download) Functionality**: When internet connectivity is available, the app's sync mechanism kicks in. During sync, any new submissions or photos are uploaded to the central system. The app provides feedback, by an icon indicator. After a successful upload, the local status of that report changes from "Pending" to "Synced" and it is removed from the pending queue. Simultaneously, the app checks for any new assignments or updates from the server. New questionnaires that have been assigned to the vessel (or updates to existing templates) are downloaded and stored. The sync also fetches any results or feedback from the server. In a full-featured system, after the crew submits a report, the shore team might review it and send back comments or create a follow-up task.

One of the notable advantages of this sync design is that it eliminates manual data transfer. In older workflows, crew filled forms and then email them or physically hand them over. The proposed solution automates this: with one tap, the entire report including images is shared to the shore management system in a structured way. Conversely, sending new tasks from shore to ship is just as easy, removing the need for lengthy email exchanges

or spreadsheets. In effect, the app and its sync feature foster direct communication between onboard crew and onshore personnel through the central platform.

Each of these functionalities was designed with the maritime user in mind, emphasizing reliability and simplicity. Importantly, every function works offline to the extent possible. By compartmentalizing features this way, the app ensures that day-to-day use (performing inspections and reporting) is uninterrupted by connectivity problems. This set of features transforms the maintenance workflow: routine inspections become guided digital checklists, data capture is richer (with photos and comments), and reporting is instantaneous and standardized. According to industry examples, such mobile tools raise the quality of safety inspections and reporting by making it easier to capture accurate results and by standardizing templates across the fleet[16] . This app's functionalities were developed in line with these best practices, ultimately contributing to safer and more efficient vessel operations.

4.5 Implementation Details

The original code was separated in a lot of parts for both simplicity of the code and better understanding. So, the main files/folders of the app that will be discussed below are the data Classes, the MainActivity of the app, the Screens, the ViewModel files, the Repositories , and the offline-Sync files such as WorkManagerUtils.

4.5.1 Data Classes

In the data classes folder there are the data classes corresponding to the db schema entities.

1. Answer

```
@Parcelize
data class Answer(
    val id: Int = 0,
    val text: String = ""
) : Parcelable
```

2. AnsweredQuestion

```
data class AnsweredQuestion(
    val questionId: Int = 0,
    val selectedAnswerText: String? = null,
    val comment: String? = null,
    var imageUri: String? = null,
    val remoteUrl: String? = null
)
```

3. AnsweredQuestionnaire

```
data class AnsweredQuestionnaire(
    var id: Int = 0,
    val userId: String = "",
    val questionnaireId: Int = -1,
    val username: String ="",
    val answers: List<AnsweredQuestion> = emptyList(),
    val timestamp: Long = 0L,
    val shipId: Int? = null
)
```

4. AnsweredQuestionnaireEntity(This was created for the offline storage)

```
@Entity(tableName = "answered_questionnaires")
data class AnsweredQuestionnaireEntity(
    @PrimaryKey(autoGenerate = true) val localId: Int =
0,
    val userId: String,
    val questionnaireId: Int,
    val username: String,
    val answers: List<AnsweredQuestion>,
    val timestamp: Long,
    val isSynced: Boolean,
    @ColumnInfo(name = "shipId", defaultValue = "0")
val shipId: Int = 0
)
```

5. AnswerOption

```
@Parcelize
data class AnswerOption(
    val id: String = "",
    val text: String = "",
    val stability: Int = 0
) : Parcelable
```

6. Fleet

```
data class Fleet(
    val id: Int = 0,
    val name: String = "",
```

```
    val firebaseUid: String? = null,
    val docId: String = ""
)
```

7. Question

```
@Parcelize
data class Question(
    val id: Int = 0,
    val text: String = "",
    val answers: List<AnswerOption> = listOf(
        AnswerOption(id = "1", text = ""),
        AnswerOption(id = "2", text = ""),
        AnswerOption(id = "3", text = ""),
        AnswerOption(id = "4", text = "")
    )
) : Parcelable
```

8. Questionnaire

```
@Parcelize
data class Questionnaire(
    val id: Int = 0,
    val title: String = "",
    val questions: List<Question> = emptyList(),
    val shipId: Int? = null
) : Parcelable
```

9. Ship

```
data class Ship(
    val id: Int = 0,
    val name: String = "",
    val fleetId: Int = 0,
    val imoNumber: String? = null
)
```

10. User

```
data class User(
    val firstName: String = "",
    val lastName: String = "",
    val email: String = "",
    val role: String = "user"
)
```

4.5.2 MainActivity.kt

This file is the entry point of the application. It extends ComponentActivity and uses Jetpack Compose to set the app's content. Inside the setContent() block, the layout is built with a background and a Surface that contains the NavigationGraph. A NavHostController was used to navigate and make it possible to move between different screens, such as the Login Screen, the Admin Main Screen, and the User Main Screen. In this activity, the main view models and repositories (AuthViewModel, FirestoreRepository and StorageRepository) provide services for user authentication and Firebase data management. User roles are checked here, so the app knows whether to show admin options, (like creating or editing questionnaires) or user options (like answering them). Image handling is also included through the StorageRepository, which makes it possible to upload and store photos linked to questionnaire responses. Overall, this file ties together the navigation flow and the core services of the rest of the app depends on.

4.5.3 Screen.kt

This file defines the navigation routes. It is implemented as a sealed class, meaning each possible screen , the UI of each page that someone can navigate, is declared as a fixed subclass with its own route string. Some screens use simple, fixed (e.g., LoginScreen and AdminMainScreen), while others accept parameters (e.g., AnswerQuestionnaireScreen), which requires a questionnaire ID. Having all the routes in one place makes navigation more reliable and easier to maintain. The full set of screens defined includes the loginScreen , the AdminMainScreen, the UserMainScreen, the EditQuestionnaireScreen, the AnswerQuestionnaireScreen the AnsweredQuestionnaireDetailScreen and the AdminViewAnswersScreen.

4.5.4 FirestoreRepository

The FirestoreRepository acts as the central data layer of the application, bridging the Android client with Firestore, Firebase Storage, and the local Room database. It ensures a seamless offline-first experience: questionnaires arrive live from the office, answers are

stored locally when the device is offline, and all data is synchronized with the cloud once connectivity is restored.

Through its methods, the repository handles every major aspect of the workflow: creating and retrieving questionnaires, managing answered data, syncing unsynced submissions, attaching images, and enforcing correct fleet-ship assignments Table 4. 1.

Category	**Methods / Features**	**Description**
Questionnaires	saveQuestionnaire	Writes a new questionnaire into the questionnaires collection.
	getHighestQuestionnaireId	Retrieves the maximum ID to avoid collisions.
	getQuestionnaireById	Loads a questionnaire by its ID for detail screens.
	listenQuestionnairesByShip (shipId)	Listens for questionnaire updates for a specific ship (live UI updates).
	deleteQuestionnaireById	Deletes a questionnaire by its ID.
Answered Questionnaires (Offline)	saveAnsweredOffline	Saves answers locally in Room when offline.
	getUnsyncedAnswers	Retrieves the queue of unsynced answers.
	syncOfflineAnswers	Uploads offline answers/photos once network is restored.
	markAsSynced	Flags a questionnaire as synced to prevent duplicates.
Answered Questionnaires (Online)	saveAnsweredQuestionnaire	Directly saves a submission when document ID is provided.
	addAnsweredQuestionnaire	Adds a new answered questionnaire.
	submitAnsweredQuestionnaire	Submits answers directly to Firestore.

Category	Methods / Features	Description
	getAnsweredQuestionnaires	Fetches all submitted answered sets.
	getAnswersForQuestionnaire(questionnaireId)	Gets answers for a specific questionnaire.
	getAnsweredQuestionnaireById	Loads a submission by document ID.
	deleteAnsweredQuestionnaire	Deletes a submitted answered questionnaire.
	fetchStatsForQuestionnaire	Scans and aggregates stored answers for statistics.
Fleet and Ships	fetchFleets, fetchShipsByFleet	Populates fleet/ship selectors for admins.
	getUserShipId, getCurrentUserShipId	Finds the ship assigned to the current user.
	setUserCurrentShip	Updates the user's assigned ship.
	isUserAssignedToShip	Verifies that a user belongs to a ship before showing forms.
	getShipHeader	Builds a readable header string for ships.
Media Handling	uploadImage	Stores photos in Firebase Storage and returns a download URL.
Mappers	Firebase → App Models	Converts Firestore documents into local models (Questionnaire, Question, AnswerOption).

Table 4. 1 Repository Responsibilities and Methods

4.5.5 ViewModels

When the user taps, the screen calls a ViewModel function. The ViewModel performs the work and updates the state. All ViewModels here were created using the following principles:

- Keep UI state as LiveData or StateFlow.
- Do network and database work inside viewModelScope.
- Expose read-only state to the UI, keep mutables private.
- Call repository methods, do not talk to Firebase or Room directly from the screen.

This keeps screens simple. Compose/Fragment observes LiveData and collects StateFlow, and renders whatever the ViewModel exposes.
Below are the ViewModels created with some info about them:

4.5.6 AdminViewAnswersViewModel.kt

Its purpose is to show all the submitted answers for one questionnaire to an admin. It is used in the admin screen that lists who answered, when answered, and basic details. The screen collects answers and renders a list. When the admin navigates in, the screen calls fetchAnswers(...) to fetch the answers of the user.

4.5.7 AuthViewModel.kt

Its purpose is to sign up users, log them in, and redirect the current user to the UI. The UI traces the authResult to show success or error messages. It checks currentUser type to route either to admin or crew home. The logging tags (TAG_AUTH) help trace flows during testing.

4.5.8 LoginViewModel.kt

Its purpose is to let a user pick fleet and ship before entering the app, and verify assignment. The screen loads fleets and ships, binds dropdowns, and shows loading or error messages based on the state.

4.5.9 MainViewModel.kt

Its purpose is to load questionnaires for the current user and keep a live list on the screen. The screen calls resolveShipAndListen() in a LaunchedEffect(Unit). The LiveData updates the list view whenever Firestore changes.

4.5.10 QuestionnaireDetailViewModel.kt

Its purpose is to load one questionnaire by id and submit a set of answers. The detail screen requests the questionnaire once, then binds the result to a form. On submit, the ViewModel sends the payload and triggers WorkManagerUtils.scheduleSync(context) from the UI layer.

4.5.11 Offline Sync files

The offline Sync path starts with the WorkManagerUtils file. This is a function that makes an one-time request for the network. It works with a unique name, in order to have only one sync at a time. After the user submission the scheduleSync method is called through this file and files and photos are uploaded by other files. Specifically , syncWorker is the file that runs in the back as coroutine to try and upload everything. First thing it does it to ask the repository for unsynced data that exist in the local storage Room library. Then after counting the number of the syncs to run it identifies if there is a photo inside the answers and makes a url after uploading the photo. Then when the sync is finished on success it marks it as synced in the local row.

The Room is the one that has every offline activity stored in it. From the appDatabase file we get the declaration of the AnsweredQuestionnaireEntity as long as the expose of the AnsweredQuestionnaireDao. The DAO file is the one that inserts the new rows inside the Room initializing the isSynced flag to zero.

4.6 Results/Screens

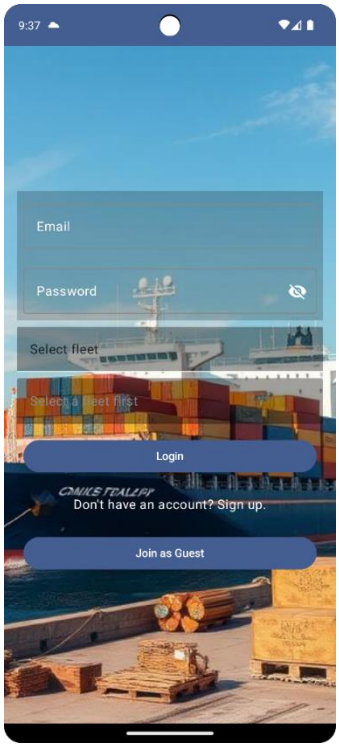


Figure 4. 4 Login Screen

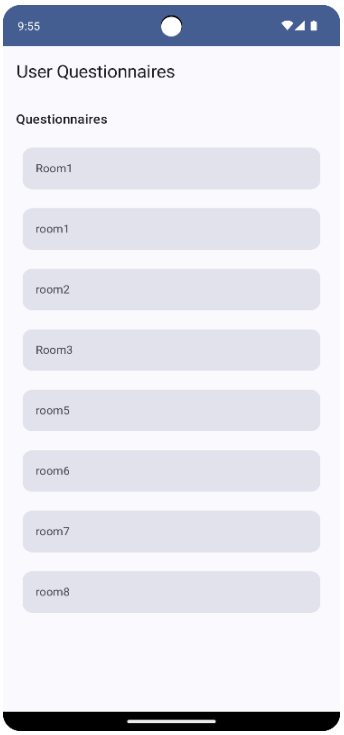


Figure 4. 3 User Main Screen

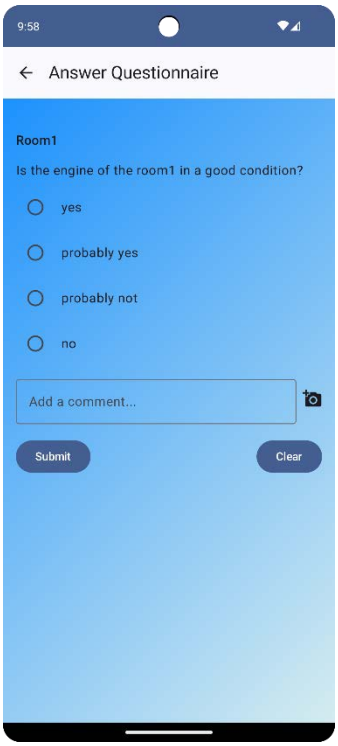


Figure 4. 6 Answer Questionnaire Screen

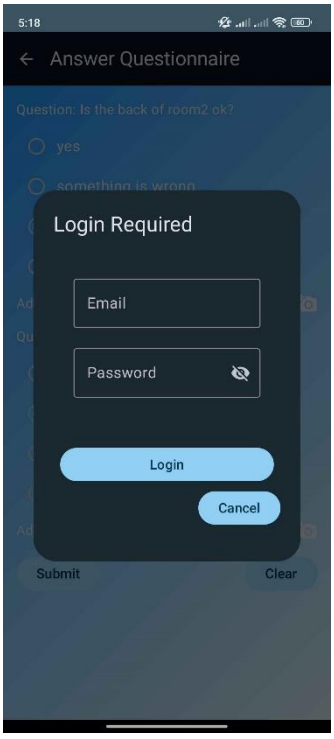


Figure 4. 5 Alert Dialog

4.6.1 Login Screen

The Login Screen (Figure 4. 4) allows users to connect, by only filling their credentials (email, password) and also by selecting their correct fleet and ship they work on. When clicking the select fleet the user can see all possible fleets that use this app. After selecting a fleet a query is running and brings only the ships of that fleet for selection in the next dropdown Menu. So it is easier and very simple for someone to identify his ship. That ensures a quick and simple connection to the application. After the credentials go to the firebase authentication center and the user is identified and if the cross reference between the user and the ship is also correct the user is finally connected to the main Screen. If the user is a simple “user” it navigates to the UserMainScreen else as a “chief” engineer it navigates to the AdminMainScreen.

4.6.2 User Main Screen

After navigating to the UserMainScreen (Figure 4. 3) the app immediately if it has connection tries to load as many questionnaires of the ship as possible. If there is no connection or no questionnaires are uploaded yet the user will get a message of no questionnaires are available. When questionnaires exist and are loaded the user can see a list of questionnaires to answer, as it is depicted in the Figure 4. 3. Then, a questionnaire is selected and navigates to the Answer Questionnnaire Screen.

4.6.3 Answer Questionnaire Screen

The Answer Questionnaire Screen (Figure 4. 6) shows the selected questionnaire by showing its title, the four multiple choices that a “user” can select and a space to add a comment and a photo. It has also a clear button to clear the previous selections and reselect to submit the correct values. After submission if the user has not logged in due to

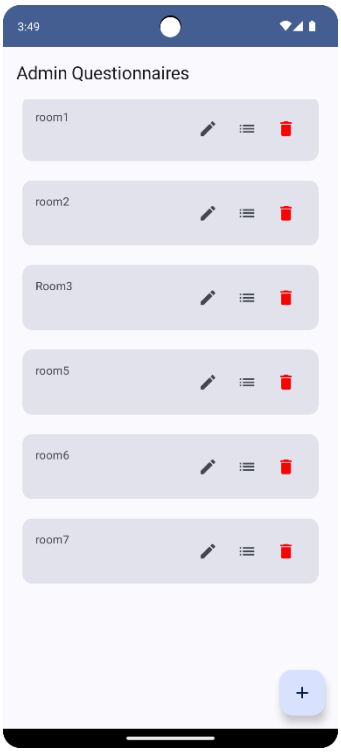


Figure 4. 9 Admin Main Screen

Figure 4. 8 View Answers Screen

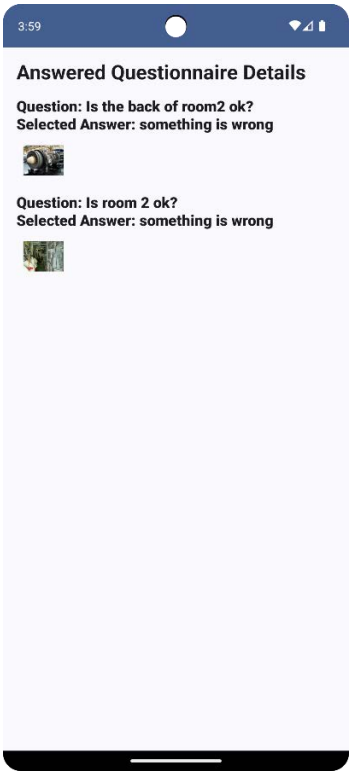
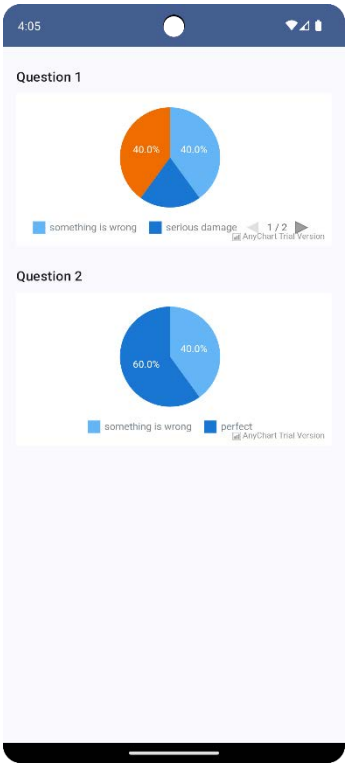


Figure 4. 7 Answered Questionnaire Detail Screen



Questionnaire Stats Screen

connection errors or due to timeouts a reconnection with his credentials is required for

security reasons. Then the questionnaire will be submitted offline-first and when the phone connects the answers will be synced.

3.6.4 Admin Main Screen

The admin Main Screen (Figure 4. 9) is the one that manages questionnaires and allows the admin to see, edit, delete or add new questionnaires. The admin can see the existing questionnaires select one of them and process it. Additionally he has the option to see the users' answers. By clicking to add a questionnaire (the + sign) or to edit (the pencil) the screen navigates to the Edit Questionnaire Screen. If the block of lines is selected the person is navigated to the View Answers Screen.

4.6.4 View Answers Screen

The View Answers Screen (Figure 4. 8) loads a list of submitted answers. It has the information of the submition date and time and the email of the user that submitted the exact questionnaire. Its obvious that someone can answer multiple times a questionnaire and it is the way the app is structured right now. By selecting a questionnaire the admin navigates to the Answered Questionnaire Detail Screen and if the admin selects show stats he navigates to the Questionnaire Stats Screen.

4.6.5 Answered Questionnaire Detail Screen

The Answered Questionnaire Detail Screen (Figure 4. 7) presents user answers. It icludes the details of the answered questionnaire like the Question Title and the selected answer. The comment and the photo that a user may has uploaded also appears on it. That way an admin can quickly see exactly what happens in a room that is inspected without being in it , and safely take the proper decisions

4.6.6 Questionnaire Stats Screen

The Questionnaire Stats Screen (Figure 4. 10) shows pie charts regarding statistics of the user's input. Depending on the answers admin can detect any trouble easier. That happens because when the answers of multiple crew members for the same question follow the same patterns, he knows the inspection went well. This has as an outcome chief engineer to go to only specific places on the ship that really need his attention and not waste time to surely fine Inspected areas.

Chapter 5 Web Application

5.1 Introduction

In complex systems like maritime operations, web-based dashboards are crucial for monitoring and managing faults. These dashboards provide real-time system status updates, enabling operators to swiftly detect anomalies and respond to faults. Fleet administrators need a remote way to monitor their vessels' status, since they can't be on board every ship, especially in large fleets. For that reason, the web dashboard and backend are essential, since the Android app alone cannot manage an entire fleet. Modern implementations show that custom web dashboards can deliver alerts and visualizations comparable to those of traditional SCADA control rooms, but with better accessibility and lower cost. For instance, a recent IoT-based power grid monitoring system used a web dashboard alongside mobile notifications to effectively offer a viable alternative to an expensive SCADA setup in remote or cost-sensitive deployments [25] Web dashboards also can provide remote supervision and decision support. A cloud fault diagnosis framework describes a cloud platform accessible via a web dashboard that allows engineers to collect sensor data, label anomalies, and even orchestrate edge-device algorithms [26]

This dashboard developed users to check questionnaires, ships, and reports, as well as view stats about the entire fleet that could provide future stats for fault detection, through ML algorithms.

5.2 Core Mechanics of Dashboard

5.2.1 User Interaction Layer

To make a more consistent look for the admins to manage their fleets effectively, the interface is composed of reusable UI components like navigation menus, data tables and visualization cards. Also some visual indicators are used to prevent the admin from being uncertain of what happens when data are loading. There are also a lot of confirmation messages and error notifications to improve performance and ensure to give the best

feedback to the administrators. For example when creating a user or a ship there is a pop-up message indicating it.

5.2.2 Authentication

The authentication starts from the login page where the admin fills his credentials, then it is handled through Firebase. The web app listens to any changes in login status and with the help of the backend and firebase makes sure that only administrators gain access. Then the app tries to fetch every information about the administrators fleet and distributes this information to the rest of the pages.

5.2.3 Navigation

Navigation is implemented by routing mechanisms that make sure that the administrator sees smooth transition between the pages. It works with the authorization system that runs in the backend, since only authenticated admin users are allowed through several pages. Each major feature (ships, users etc.) has its own Route making it possible for the web dashboard navigation to grow further.

5.2.4 Main Views and Functional Modules

This dashboard is implemented around several views of pages that correspond to the operational needs of the fleet administrators:

Login: Provides secure sign-in with error handling and redirection on success.

Dashboard Overview: Presents high-level statistics as summary cards, including users, ships, questionnaires, and synchronization logs.

Ships: Allows new vessels to be added and displays the fleet's current composition.

Users: Manages personnel records with features for creating, searching, updating roles, and linking users to ships.

Questionnaires: Displays a list of ships that have their own list of questionnaires , that when pressed can navigate to the Questionnaire Detail Screen.

Questionnaire Details: Offers a detailed breakdown of questions, options, and response, showing the exact answers comments and images of the respondent.

Reports: Displays recent submissions sorted by recency, with options for manual refresh.
Synchronization Logs: Provides paginated history of synchronization events, showing their document IDs, statuses, errors, and timestamps.

5.2.5 Backend Integration

As far as communication with the backend is concerned, some service functions are handling it. These functions utilize secure HTTP requests to request resources or to update any data, with the proper authentication Tokens. Then the web dashboard is able to present the data with its smooth UI.

5.2.6 Services

The services folder connects the dashboard to the backend and Firebase.
It uses Axios with an interceptor that automatically attaches a new Firebase token to every request, keeping calls secure. There are helper functions for fetching users, ships, questionnaires, reports, sync logs, and dashboard stats. It also includes actions like creating ships, assigning users, updating roles, and renaming fleets. This way, the UI can stay clean and only call simple functions instead of writing requests everywhere.
On the Firebase side, the file handles login and logout, relying on Firebase Auth.

5.3 Technologies

Without further analysis, since they will be presented in next Chapter (**Error! R eference source not found.** the backend of the Web application was implemented with node.js using the MySQL db.

For the implementation of the Web dashboard UI the use of the React.js library and the Vite.js tool were selected. React.js as a component-based library manages to structure the dashboard with modular and reusable elements and Vite as the most recent tool that provides hot-reloads and faster compilation tools[27] making everything work faster. That way not only compatibility was successful but also speed. When a developer edits a React component, Vite updates the module immediately in the browser without a full reload. Unlike older tools where HMR performance degrades as the app grows, Vite's HMR is implemented over native ESM and precisely invalidates only the changed modules [27] .

The structure and the architecture of the UI was organized was modular, by having folders like components for making the UI elements, Context, Pages, Routes and Services for the purposes of this project. It also contained separate files such as obviously the index.js, App.js that were the main files of the app, firebase.js for connection to firebase.

5.4 Pages Results

The result dashboard presents a clean and fast interface for managing a fleet with ships, users and questionnaires. It provides the application with a clear UI with quick filtering and review. This has as a result a stable tool that streamlines quality checks across fleets, reduces manual aggregation, and surfaces actionable insights at a glance.

5.4.1 Login Page

The login page (Figure 5. 1) presents a centered form with a title labeled Admin Login. It includes two input fields, one for the email and one for the password, along with an area to display an error message if authentication fails. At the bottom, a submit button allows the administrator to log in and access the dashboard.

5.4.2 Dashboard Page

The Dashboard Page (Figure 5. 2) begins with a title and a row of summary boxes, also known as KPI cards (Key Performance Indicator cards), which highlight the main numbers for users, ships, questionnaire submissions, and sync logs. Just below, a section presents the submission trend over the last thirty days, where each date is shown with a small horizontal bar and the corresponding count, together with a running total. Another section lists the top ships over the past seven days, ranking vessels by how many submissions they have made. Finally, the sync health area for the last twenty-four hours is displayed in four compact boxes showing total attempts, successes, errors, and the calculated success rate. The page also has simple loading and error messages that appear in place of the content when data cannot be retrieved.

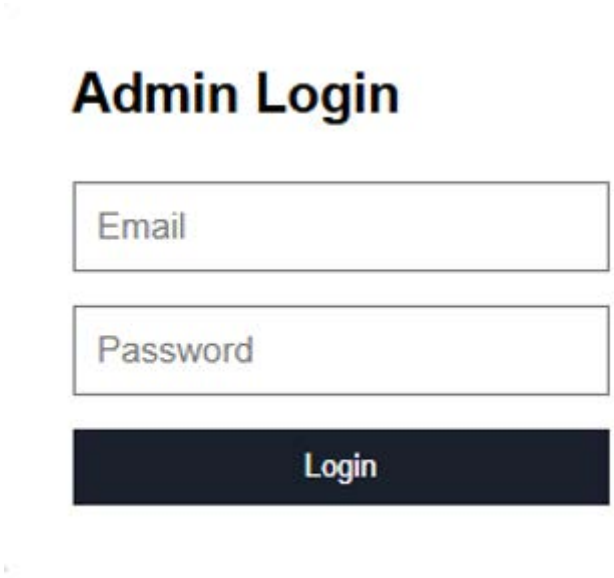


Figure 5. 1 Login Page

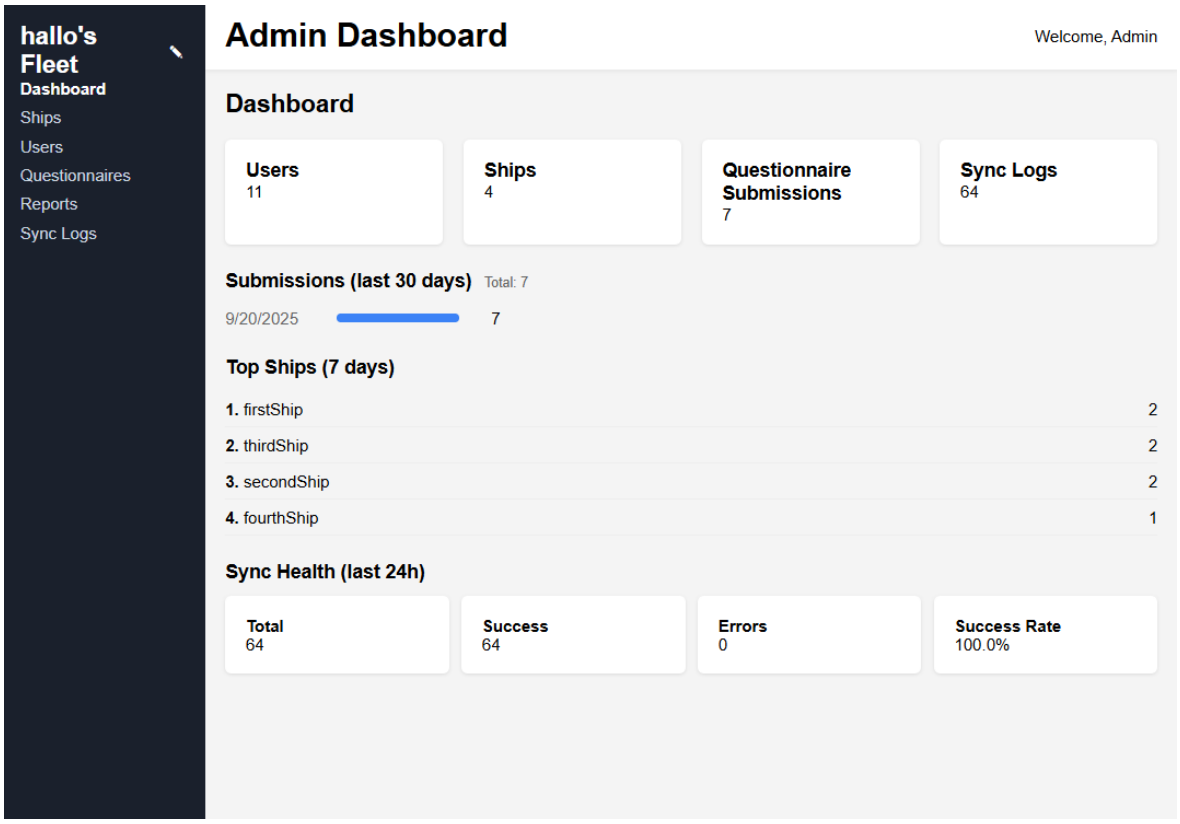


Figure 5. 2 Dashboard Page

5.4.3 Ships Page

The ships page (Figure 5. 3) displays a simple management view for the fleet. At the top, a heading introduces the section, followed by a form where the administrator can add a new ship by entering its name and its IMO number. Below the form, a table lists all the ships currently registered, showing their names and IMO numbers in two columns. If no ships are available, the table instead shows a single row with a message indicating that no ships were found.

5.4.4 Users Page

The users page (Figure 5. 4) shows a “Fleet Users” heading with an optional loader, followed by a compact form to add a new user with fields for first name, last name, email, password, ship selection, and role, plus a submit button. Beneath the form, a small search box filters the list. The main area is a table that lists users with columns for name, email, assigned ship, role, and an action button that toggles between promoting to chief or demoting to user. When the list is long, simple pagination controls appear at the bottom to move between pages.

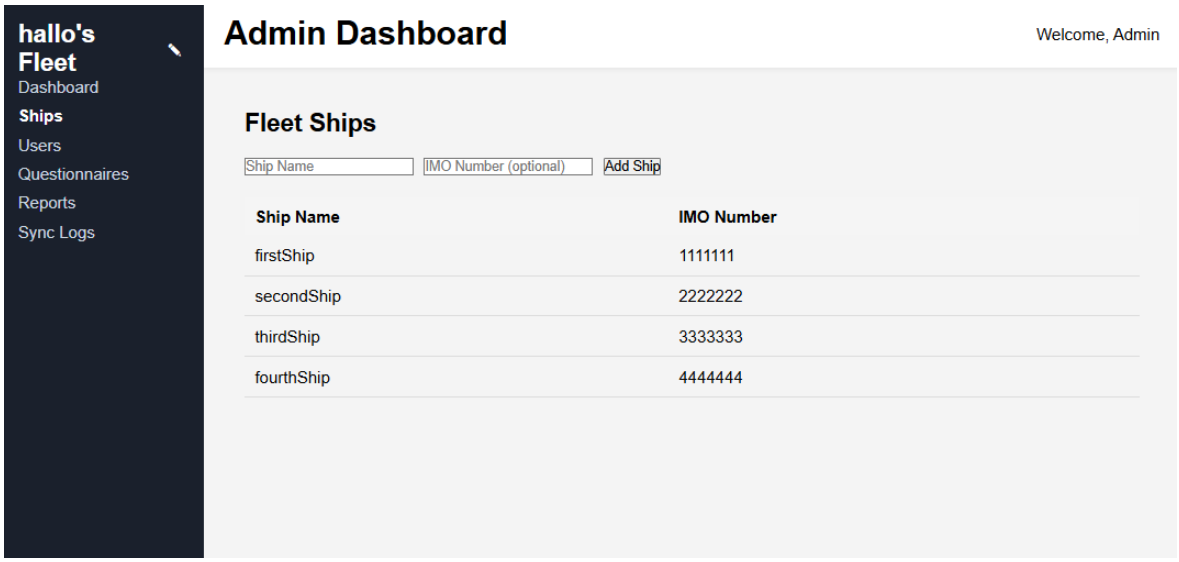


Figure 5. 3 Ships Page

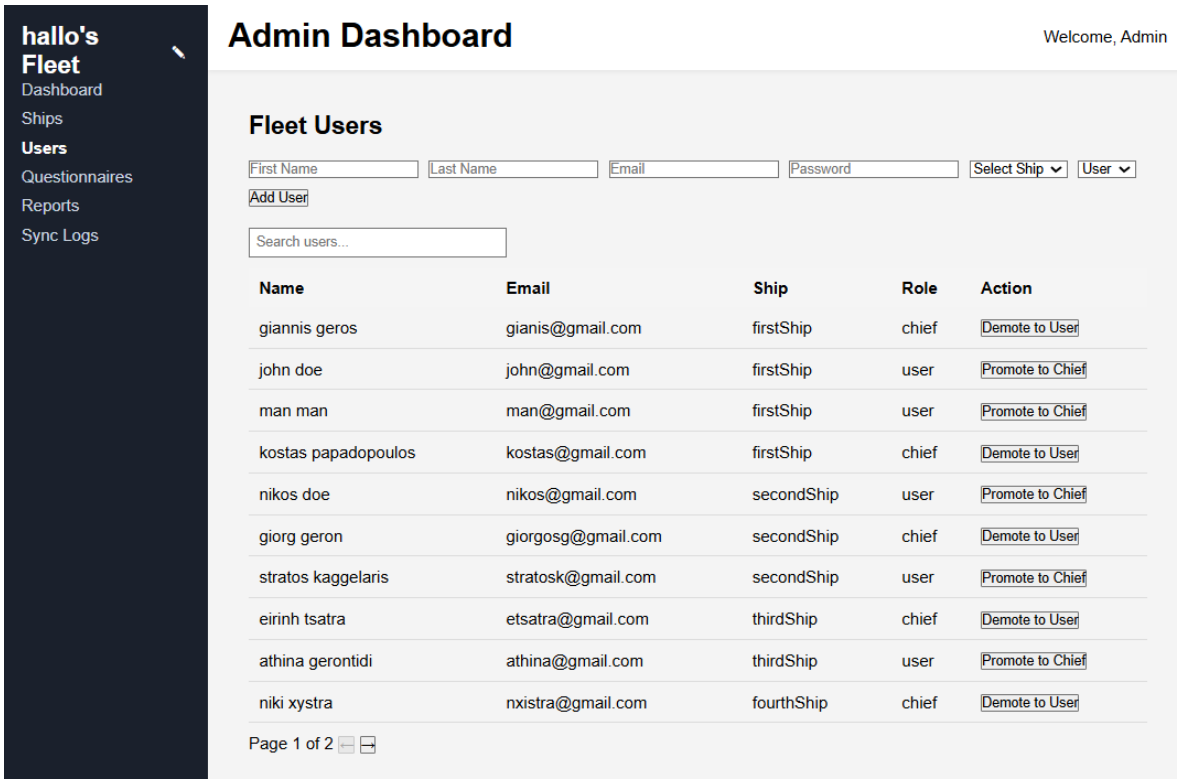


Figure 5. 4 Users Page

5.4.5 Questionnaires Page

The Questionnaires page (Figure 5. 5) shows a “Questionnaires by Ship” heading with simple loading or error messages when needed. After loading, it lists each ship by name with its IMO number, and under each ship it either displays a note that no questionnaires are linked or a list of questionnaire entries showing the title and ID. Each entry provides a button to view its details and another to delete it with confirmation.

5.4.6 Questionnaires Detail Page

The Questionnaires Detail page (Figure 5. 6) shows a back link and the questionnaire title, with a small note if no ship is linked. The main view is split in two columns. On the left, a searchable list of questions displays each question’s options with response counts and, when a respondent is selected, their answer, comment, and any image link. On the right, a searchable respondents list shows names, emails, and completion time, and selecting one updates the details. Loading and error messages appear while data is fetched, and clear empty states are shown when no questions or respondents match the filters.

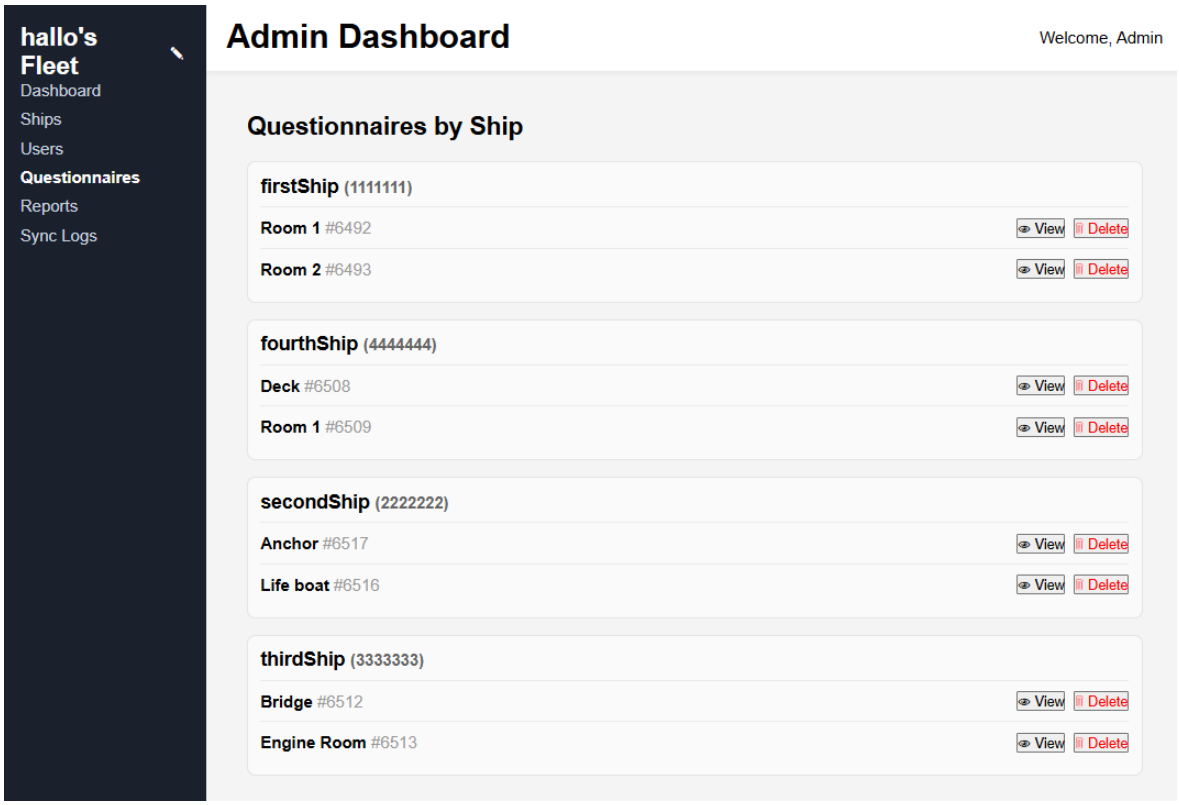


Figure 5. 5 Questionnaires Page

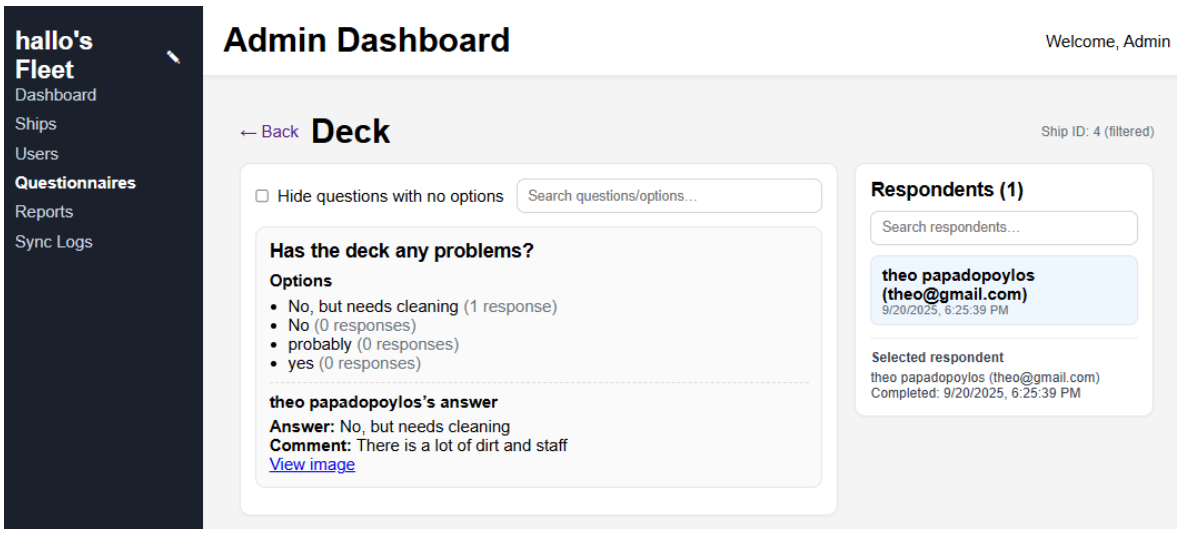


Figure 5. 6 Questionnaires Detail Page

5.4.7 Reports Page

The reports page (Figure 5. 7) shows a “Recent Submissions” title with a refresh button above the list. When data loads or fails, a short status message appears; if there are no results, a simple “No submissions found” message is shown. Each submission is displayed as a text row summarizing the user, questionnaire title, ship name with IMO, and

the submission time. At the bottom, centered pagination controls provide Previous and Next buttons with a page indicator.

5.4.8 Sync Logs Page

The sync logs page (Figure 5. 8) presents a “Sync Logs” heading followed by simple status messages for loading or errors. When data is available, a full-width table lists each log with columns for document ID, type, status, error message, and the timestamp. If there are no entries, a brief “No logs found” message is shown instead. Pagination controls appear below the table with Previous and Next buttons and a page counter to navigate through the log pages.

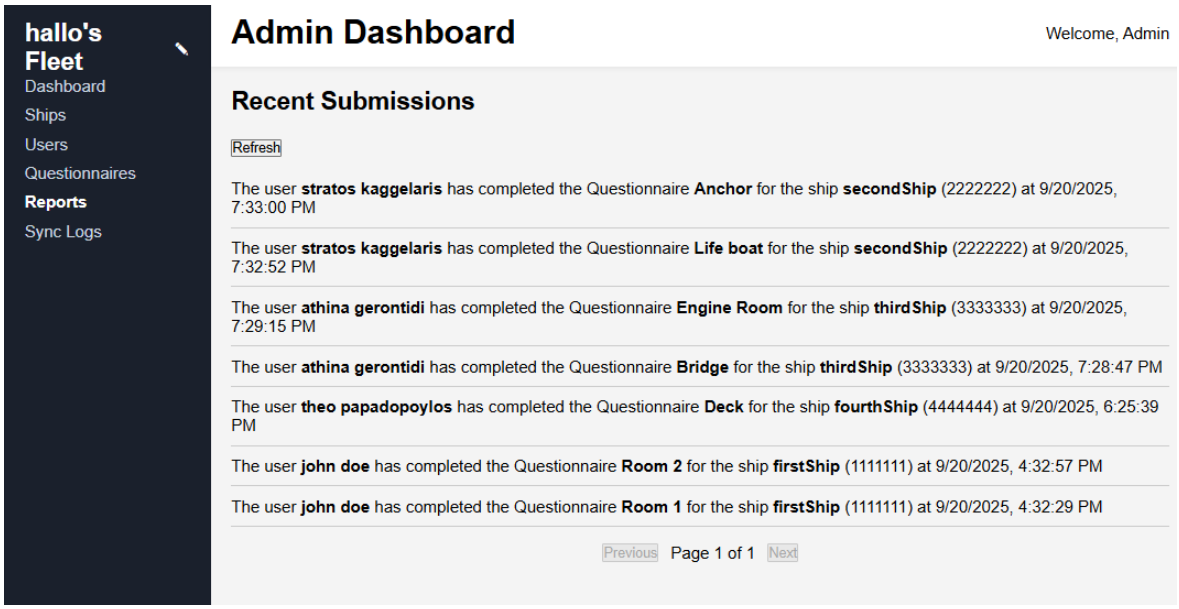


Figure 5. 7 Reports Page

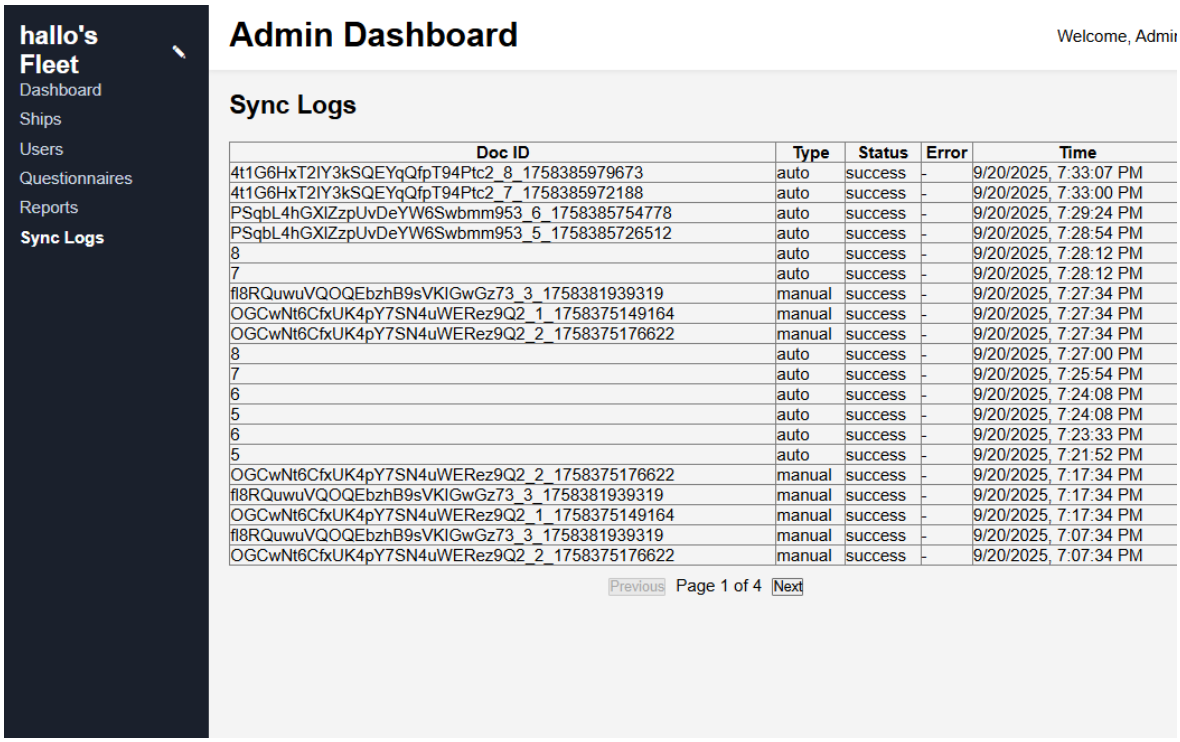


Figure 5. 8 Sync Logs Page

Chapter 6 Backend Server

6.1 Introduction

The backend server is the main component of the system. It works as the bridge between Android Application, the web dashboard and the databases ensuring that data travels securely between all components. The backend server offers a set of mechanisms that provide communication to the system, synchronization among the two databases and reliability with the secure authentications and authorizations. Its design is based on web site principles, where the clients in our case the web dashboard don't have to directly interact with the databases. Instead they communicate with the backend to offer security, scalability and consistency.

An API creation was a great need for the system to work, and the reason for this could simply be understood by just the definition of it. An API is a tool that allows software components to communicate with each other using a set of rules and protocols [17] .

The API is used as a middleman between the clients (which in our case are the web and the android application), and the server. It also manages the data flow between them and

handles data requests and how data are returned. The API also works as a gateway between the user interface and the backend database. All operations on system data, including user accounts, fleet and ship records, and submitted answers, are performed by sending HTTP requests to the API instead of accessing the database directly. This structured approach ensures that clients can consistently and safely interact with the server.

6.1.1 RESTful API

The API in our system is structured as a RESTful API. In REST (Representational State Transfer) data like users, questionnaires and ships are identified as “resources” and “ are identified by URIs and can be manipulated using standard HTTP methods” according to this article [18] So it uses GET requests to retrieve data, POST requests to create new records, PUT to update existing records and DELETE to remove them.

A basic characteristic of a RESTful API is the statelessness, it means that all the necessary information should be sent through the HTTP request as the server does not retain any session state between requests [18]

6.2 Technologies

The key technologies and components that were used for the implementation of the API :

- **Node.js and Express.js:**

  Node.js is a JavaScript runtime built on Chrome’s V8 JavaScript engine [19] It works fast and is the most well spread server code language. So it is best for web servers, like the one needed for the system, can make concurrent connections , and backend applications like the API. Also the node.js provides the runtime for the server allowing the server logic in JavaScript taking advantage of the Node’s asynchronous capabilities to manage I/O operations.

  Express.js is a fast and minimal framework for Node.js [20], provides a robust set of features for building APIs. In our system, Express is used to set up the RESTful API routes, define middleware for tasks like authentication and error handling, and implement controller functions that process incoming requests and produce responses. The project was organized perfectly due to the Express routing mechanism and middleware support. It was separated to use route modules for the API logic and controllers for the Endpoints.

- **MySQL database:**

  MySQL is a widely used, open-source, relational database management system (RDBMS) known for its reliability, performance, and scalability [21] Inside this db data are stored in structured tables that can be managed and retrieved from the users of the db. For this project the ACID-compliant transactions were needed for critical data consistency and the fact that this db is able to perform complex JOINs. Communication with MySQL is established by the mysql2 library of the node.js server. The server maintains a connection pool to the MySQL database to face concurrent queries.

- **Firebase SDK and Admin Cloud Firestore:**
  Firebase Admin SDK is a set of server libraries provided by Google Firebase that allows backend systems to interact with Firebase services with admin privileges [22] Of course Firestore in our system is used for its authentication system and also for synchronization, its cloud firestore and its storage for images.
  By using the Admin SDK reads and writes happen automatically through its connection.

- **Postman:**
  Postman is an all-in-one API platform building and working with APIs [23] It was used for testing communication as it replies with JSON visuals after specifying the body of the URL and the request that needs to be made. It obviously can accept any type of query any POST ,GET ,PUT and DELETE request can be made through this API. Until a working version of the UI all the visual testing was done through this tool. Some examples are these *POST /fleets* (to create a new fleet), *GET /ships?fleetId=X* (to list ships in a fleet), *POST /answers* (to submit questionnaire answers), *GET /reports/:id* (to fetch a specific report), etc.

6.3 Functional Architecture

The structure of the backend server was inspired [24] It uses folders as logical containers that help separate responsibilities in the codebase, making the backend easier to understand and more scalable. The REST-API defines rules about resources, endpoints,

HTTP methods and representations and that's how these folders where organized based on those rules.

6.3.1 Core Application Layer

The core application of the system begins with the server.js. It starts the backend application, it sets up the MySQL pool establishing a connection, gives the flag to start the Firestore Listeners for synchronization and schedules the periodic Syncs of MySQL with Firebase. This file could be named also as the "heart" of the backend, since it makes sure that all the other components remain connected.

6.3.2 Database Connection Management

The initialization of the MySQL database was successful and made by creating a pool that contains inside the host the user the database and the password that are required to a successful connection with the mySQL db. This guaranteed that many requests could be handled effectively.

6.3.3 Security and Role management

The security and the Role management was implemented by the definition of two important middleware functions. The authentication that happens through the firebase UID and the authorization, which allows the beginning of a role-based system as this one. As a method was called, depending on if there was an authorized token or an authenticated token could give you the result. Thus, it could build a security wall through these constant checks.

6.3.4 Logic Layer

Controllers are the core logic of the API. They are located between the clients (e.g. the Android app or Postman) and the databases services. Their role is to receive a request, apply the appropriate rules or permissions, communicate with the MySQL or Firestore, and send back a clean, structured response. By keeping this logic in the controllers, the project achieves two important goals. First, the code stays organized because each controller

focuses only on its own domain, such as users, fleets, or questionnaires. Second, the system becomes easier to maintain and extend because new features can be added by enhancing the relevant controller without affecting unrelated parts.

In this API, each controller has a clear responsibility. The User Controller manages users in the system by creating accounts, updating details, assigning users to ships, and syncing with Firebase to ensure consistent login and data. The Fleet Controller manages groups of ships, allowing administrators to create fleets, add ships, and see which users belong to their fleet. The Questionnaire Controller manages the inspection forms used on board and provides both simple CRUD operations and detailed breakdowns of answers and response rates. The Reports Controller generates insights from those questionnaires, providing summaries and detailed views that managers can use. The Dashboard Controller provides a quick "snapshot" of system activity by counting key resources, making it easy to monitor usage at a glance. Finally, the Sync Controller keeps track of database synchronization and logs whether data has successfully moved between MySQL and Firestore.

6.3.5 Routing and Endpoint definition

Routes groups the API in endpoints. A route is defining the endpoint of the API method that we call. The main route groups are : /users, /fleets, /ships, /questionnaires, /answers, /reports, and /sync. All the modules are being exported to the router. A better understanding of what each method of the endpoints does can be seen to the table below (Table 6. 1).

6.3.6 Synchronization Mechanisms

The synchronization modules are one of the most critical parts of the API, ensuring consistency between MySQL, and Firestore.

The Sync files contain Firestore live listeners and batch exporters that successfully transmit data. Whenever a document is added, modified, or deleted, the corresponding record goes in MySQL and notifies it to make the change. This guarantees that submissions made from mobile clients, which may work offline, eventually flow into the central SQL

database. Conversely, some sync scripts push data from MySQL into Firestore using the following scripts: usersToFirestore.js, shipsToFirestore.js, fleetsToFirestore.js, and questionnairesToFirestore.js., and shipAssignmentsToFirestore.js. These scripts package relational data into Firestore-friendly JSON documents. Each script focuses on a specific domain: users, ships, fleets, questionnaires with their questions and options, and users to ships assignments. Together, they make sure that mobile devices always see the latest authoritative data when connected.

There is also a sync mechanism, that calls the proper queries for synchronization and periodically refreshes all collections at once. This periodic sync acts as a safety net, ensuring consistency, even if some real-time events are missed. This design is very important in the maritime setting, where connectivity is unreliable. However, data about crews, ships, and inspection results must eventually converge for safety management and reporting purposes.

Endpoint	**Method**	**Description**
/users	GET	Retrieve list of all users
/users	POST	Create a new user account
/users/{id}	GET	Get specific user profile
/users/{id}	PUT/PATCH	Update user information or role
/users/{id}	DELETE	Delete a user account
/fleets	GET	List all fleets
/fleets	POST	Create a new fleet
/fleets/{id}	GET	Get details of a specific fleet
/fleets/{id}	PATCH	Update fleet information
/ships	GET	List ships (optionally by fleet via query param)
/ships	POST	Register a new ship in a fleet
/ships/{id}	GET	Get details of a specific ship
/ships/{id}	PATCH	Update ship information (e.g., status)
/questionnaires	GET	List available questionnaire templates
/questionnaires	POST	Create a new questionnaire template
/questionnaires/{id}	GET	Get full details of a questionnaire (questions)

Endpoint	Method	Description
/questionnaires/{id}	PATCH	Update questionnaire (e.g., edit questions)
/answers	POST	Submit answers for a questionnaire on a ship
/answers	GET	Query submitted answers (by ship, questionnaire)
/reports	GET	List reports (filterable by fleet/ship)
/reports/{id}	GET	Retrieve a specific fault report
/reports/{id}	PATCH	Update report status (e.g., mark resolved)
/sync	POST	Trigger full synchronization (MySQL → Firestore)
/sync/{resource}	POST	Sync specific resource category (e.g., /sync/ships)

Table 6. 1 Method and description of each of the most important endpoints of the API

6.4 Results

After implementing each method, a way to test it had to be found. The Postman API is the most popular and suitable choice for visual results.

A Firebase ID token is needed to verify most requests due to the authorization that ensures that only admin-role users can retrieve specific data. The base URL for the request is also needed, and then the key "Authorization", that is applied in Headers with the value "Bearer Firebase ID token" (Figure 6. 1).

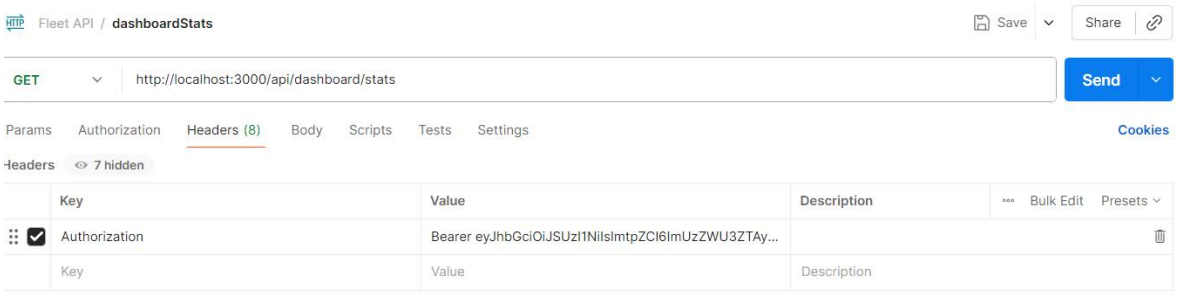


Figure 6. 1 Postman GET Method request

Below there are some of the main created Methods from the API:

Dashboard stats http://localhost:3000/api/dashboard/stats

```
{
  "users": 14,
  "ships": 3,
  "questionnaires": 8,
  "syncLogs": 4392
}
```


Get Fleets http://localhost:3000/api/fleets


```
[
  {
    "id": 1,
    "name": "hallo's Fleet",
    "owner_firebase_uid": "Ibh0zUIeOSclAlhIofrznhDSLdC2",
    "created_at": "2025-07-16T22:37:42.000Z"
  }
]
```


Get ships http://localhost:3000/api/fleets/1/ships


```
[
  {
    "id": 1,
    "name": "firstShip",
    "imo_number": "1111111"
  },
  {
    "id": 2,
    "name": "secondShip",
    "imo_number": "2222222"
  },
  {
    "id": 3,
    "name": "thirdShip",
    "imo_number": "3333333"
  }
]
```


POST AddShips http://localhost:3000/api/fleets/1/ships

If trying to add ships to a fleet with this post method
I got this error {


```
  "error": "Forbidden: Admins only"
}
```


Since administrator privileges required to add a vessel.

Get Users http://localhost:3000/api/users

This API endpoint call returns all users of the system


```
[
  {
    "id": 5,
    "firebase_id": "Ibh0zUIeOSclAlhIofrznhDSLdC2",
    "first_name": "hallo",
    "last_name": "hallo",
    "email": "hallo@gmail.com",
    "role": "admin",
    "created_at": "2025-07-16T22:37:02.000Z"
  },

  {
    "id": 99,
    "firebase_id": "OGCwNt6CfxUK4pY7SN4uWERez9Q2",
    "first_name": "john",
    "last_name": "doe",
    "email": "john@gmail.com",
    "role": "user",
    "created_at": "2025-07-17T13:36:45.000Z"
  },
  {
    "id": 108,
    "firebase_id": "KCvBszsB6KefUDZuLew7OensQNt1",
    "first_name": "man",
    "last_name": "man",
    "email": "man@gmail.com",
    "role": "user",
    "created_at": "2025-07-17T13:44:44.000Z"
  }
]
```


Logging that appears in the Terminal after each sync was added for better testing. As one can observe the Figure below (Figure 6. 2), sync is called when the server side application starts and after some periodic of time it is re-called and it syncs again. It shows in logs everything that synced in counting style and the time it needed for sync.

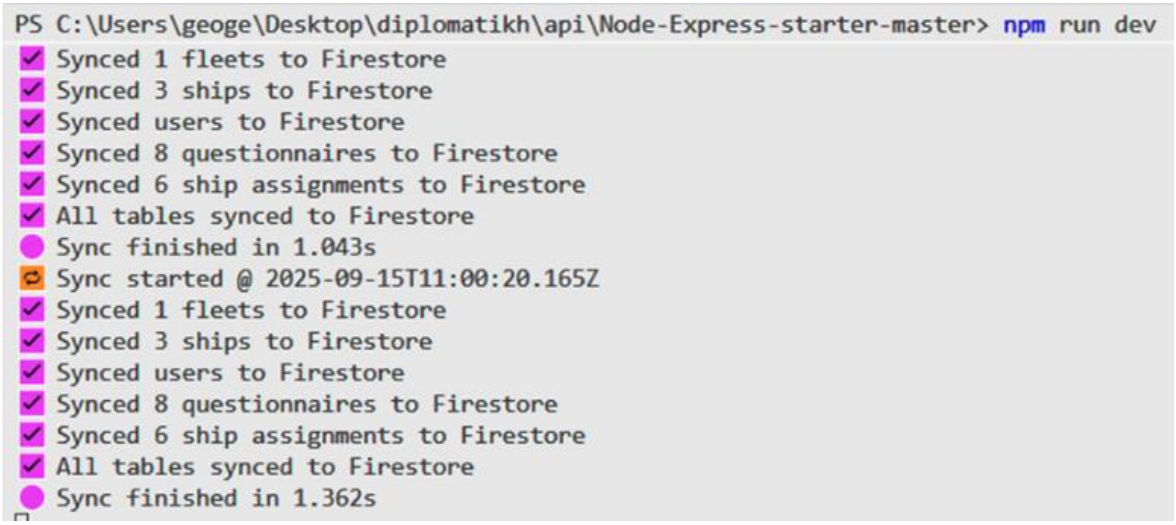


Figure 6. 2 Sync when starting

Chapter 7 ML Algorithms Research

7.1 Introduction

In order to create a fully functional system that detects faults in commercial ships, a prediction component had to be added to prevent future faults. Thus, research was conducted to determine the most suitable machine learning (ML) algorithms for this application. However, since no real data could be assembled without releasing the application.

The system collects answers to an inspection questionnaire, (each question has four categorical choices) and seeks to predict when a fault will occur next. In other words, this is a predictive maintenance problem, using past inspection data to forecast future failures[28]

The questionnaire answers are discrete categorical features, and the goal is to output a time range until the next detected fault. Such a task can be framed as either a **regression** (predicting a continuous time) or a **classification** (predicting for example < 1 week until fault) problem. In practice, machine learning models learn patterns linking the inspection responses to failure timing. As [28] notes, integrating inspection data with analytics "can uncover patterns and predict when equipment might need attention". Here, we treat the questionnaire answers as features (inputs) and the fault timing as the target outcome.

Each questionnaire answer is categorical (4 choices). Before modeling, the inputs should be encoded first with one-hot or ordinal encoding for ML algorithms. For example, one-hot encoding would transform a 4-choice question into 3 binary variables [28] This preprocessing ensures the model can handle discrete survey answers alongside any numerical features.

7.2 Models

Some of the learning algorithms could be applied are:

7.2.1 Decision Trees and Random Forests

Tree-based models manage mixed data and can model non-linear relationships. A single decision tree can perform classification or regression. Random Forests are very popular for predictive maintenance tasks because they reduce overfitting and capture complex patterns. Random Forests also manage categorical features (after encoding) well and are robust to noise [29] .

7.2.2 Gradient Boosting (XGBoost, LightGBM)

Boosted tree ensembles often achieve state-of-the-art performance on tabular data. In one machinery case, XGBoost outperformed other classifiers, with Random Forest being a close second. The analytics example found "XGBoost has the best performance for the given scenario" of fault classification [30] Boosting models can handle encoded categorical inputs and typically give high accuracy, making them a strong choice for predictive maintenance.

7.2.3 Support Vector Machines (SVMs)

SVMs are used for classification or regression on structured data[29] .They work well with moderate-sized feature sets, especially if the data is well-scaled, but require categorical features to be numerically encoded. SVMs have been used in fault detection scenarios and can serve as a solid benchmark.

7.2.4 Neural Networks (Multi-Layer Perceptron)

Feed-forward neural networks can come up to complex, nonlinear mappings, and they can manage both classification and regression [28] A survey of PdM literature notes that a multilayer perceptron (MLP) often achieves high accuracy and one study found that a K-nearest-neighbor classifier and an MLP gave the best accuracy on a classification task [29] Neural nets need a high amount of training data and tuning, but they can exploit patterns in questionnaire responses.

7.2.5 k-Nearest Neighbors (kNN)

A non-parametric method that classifies or regresses based on similarity to past cases. In one review, kNN matched or exceeded more complex models in predictive maintenance classification accuracy [29] KNN is easy to implement but can be sensitive to the feature scaling and the size of the dataset. It may serve as a quick baseline, but maybe not the ideal for this case.

7.3 Results best model

In practice, ensemble tree models tend to work very well on structured inspection data. **XGBoost (gradient boosting)** often achieves top performance, as reported in predictive maintenance case studies [30]. In the cited example, XGBoost “has the best performance” for failure prediction and was chosen over Random Forest. Random Forest itself is also a strong candidate (it “is a popular ML algorithm used for classification or regression” [29] and handles categorical inputs effectively).

Thus, a sensible strategy is to start with **tree-based ensembles**. For example, build an XGBoost model (or LightGBM) on the encoded questionnaire data. This model can be trained for either classification (if you discretize the failure time into categories) or regression (to predict continuous time to fault). It will capture nonlinear interactions

between questionnaire responses. A XGBoost or a Random Forest model that is well-tuned, is more likely to achieve higher predictive accuracy [30] , [29] Ensure using cross-validation or hold-out testing to verify the model's performance on new data.

By doing a quick summary we have:

- Encoding all categorical answers so models can use them [28]
- **Tree ensembles** (XGBoost, LightGBM, Random Forest) are highly effective for tabular PdM data and should be tested first. In one study, XGBoost significantly outperformed others, with RF a close runner-up [28]
- **Neural nets** (MLP) and **kNN** are also promising; e.g. KNN and an MLP each achieved top accuracy in a PdM anomaly task[29] These require careful tuning and adequate data.
- Ultimately, the "best" model depends on data specifics. A practical workflow is to try several (e.g. XGBoost, RF, SVM, MLP) and compare performance via cross-validation. Given prior work, **XGBoost (gradient boosting)** is a strong initial choice for forecasting faults from questionnaire data[29]

Recommendation for future work

For a future work someone should begin with an XGBoost (or Random Forest) model on the encoded questionnaire responses. This approach is well-supported by PdM research [29] and should give a robust prediction of when the next fault will occur. Validate its accuracy, and consider other models if needed.

Chapter 8 Conclusions

8.1 Summary and Conclusions

This chapter aims to provide a comprehensive summary and reflection of the key results of the system developed throughout the thesis work. This thesis focused on the design and implementation of an integrated digital system for fault detection and prevention in commercial vessels. After heavy research into this maritime sector, this project came as an answer to the problems that were discovered and overcame basic everyday ship problems that as now exist. This thesis helped understanding these problems, evolve better consciousness as a mechanic, since the purpose of mechanics are to find solutions to every problems and improve basic skills in coding and researching. The results confirm that the system contributes meaningfully to the modernization of maritime maintenance practices. The Android app allows crew members and engineers to complete inspections and questionnaires in real time, even in conditions without internet connectivity. The backend API ensures that synchronization between ship and cloud occurs securely and consistently, while the web dashboard offers administrators an overview of ships, users, and questionnaire reports at a fleet-wide level. Some of the main pros of this project is the reduction in manual paperwork, improved access to inspection data, structured storage of maintenance records, and a foundation for analytics. At the same time, some challenges remain. The reliance on continuous user input means the system's effectiveness depends on adoption by crews, and its current design does not yet include predictive fault detection. It also is not possible, if the questionnaires are not pre-loaded to work and still have to work hybrid and use paperwork till internet loads questionnaires. Nevertheless, the thesis demonstrates that modern technologies such as React, Node.js, MySQL, Firebase, and Android Jetpack Compose can be effectively integrated to solve practical problems in the shipping industry.

8.2 Future Extensions

There are several directions where this project can grow further. The most popular is the one already mentioned of the predictive character of the application. Since the application offers a history of the answered questionnaires, now its possible to keep track of the answers separate them to numeric or categorical values and with classification have a decent result.

Another improvement for the system would be a more advanced data visualization, by adding interactive charts, trend analysis, and anomaly detection on the dashboard. That could be truly helpful for admins to detect patterns even quicker than before. Real-time sensor data from ship equipment could also be linked to the platform, which would automate part of the inspection process and this way workload for the crew members would be reduced.

Additional Android app Utilities**:** Depending on project scope, the app could have additional features that complement the core functionalities. For instance, a search function to quickly find a specific report or question (useful if looking up how a problem was previously solved). Another possible utility is barcode/QRA code scanning to identify equipment – e.g., the crew could scan a QR code on a machinery part and the app jumps to the section of the questionnaire for that part. It was also considered an analytics dashboard on the app for crew – giving them some insight such as “10 reports filed this month, 2 pending review,” though most analytics were to be done on the web management portal. Nonetheless, small features like counting open issues can motivate crew by showing the impact of their reporting.

To conclude, the project succeeds in delivering a working system that improves how maintenance data is collected and managed. However, one of the most important assets of this system, lies in the potential for future extensions, especially predictive analytics, which could make fault detection faster, smarter, and more reliable for the shipping industry.

Bibliography

[1] G. Singh, "Don’t hold back on the switch to eLogs," ABS Wavesight, 24 12 2020. [Online]. Available: https://www.maritimemagazines.com/marine-news/202412/dont-hold-back-on-the-switch-to-elogs/. [Accessed 7 9 2025].

[2] "Digitalisation uncovered: Whats next for shipping?," Inmarsat, [Online]. Available: https://www.inmarsat.com/content/dam/inmarsat/corporate/documents/maritime/insights/Digitalisation_Uncovered_What_s_Next_for_Shipping.pdf#:~:text=The%20findings%20also%20point%20to,the%20potential%20for%20greater%20savings [Accessed 07 09 2025].

[3] os-system, "Comparison of Firestore with Alternative Databases," os-system, [Online]. Available: https://os-system.com/blog/firestore-vs-alternative-databases/#:~:text=Differences%20between%20Firestore%20and%20MySQL,source%20database [Accessed 13 08 2025].

[4] "Best Database for Mobile Apps in 2025," [Online]. Available: https://www.aalpha.net/articles/best-database-for-mobile-apps/#:~:text=1.%20%23%23%20Cloud [Accessed 07 09 2025].

[5] "Firebase Database vs MySQL," [Online]. Available: https://blog.back4app.com/firebase-vs-mysql/#:~:text=%2A%20Realtime%20Updates%20%E2%80%93%C2%A0Default%20real,approaches%20to%20get%20this%20feature [Accessed 07 09 2025].

[6] "Kotlin vs Java: A Comprehensive Comparison for Developers," [Online]. Available: https://www.imaginarycloud.com/blog/kotlin-vs-java#:~:text=Kotlin%20vs%20Java%2C%20which%20is,depends%20on%20your%20project%20needs [Accessed 07 09 2025].

[7] "Spring Boot vs. Node.js vs. .NET Core vs. Django: a “celebrity face-off”," [Online]. Available: https://www.frameworktraining.co.uk/news-insights/spring-boot-vs-node-js-versus-dot-net-core-vs-django#:~:text=Performance [Accessed 07 09 2025].

[8] "Vite vs Create React App in 2024," [Online]. Available: https://blog.bitsrc.io/vite-vs-create-react-app-326e8cc2c46b [Accessed 07 09 2025].

[9] "Vite Vs. Webpack: The Best Bundler For React Applications," [Online]. Available: https://www.axelerant.com/blog/vite-vs-webpack-the-best-react-bundler#:~:text=Speed [Accessed 07 09 2025].

[10] "Postgres vs. MySQL: a Complete Comparison in 2025," [Online]. Available: https://www.bytebase.com/blog/postgres-vs-mysql/ [Accessed 07 09 2025].

[11] "Seafarer's professions and ranks," [Online]. Available: https://en.wikipedia.org/wiki/Seafarer%27s_professions_and_ranks [Accessed 07 09 2025].

[12] "Pros and Cons of Kotlin for Android App Development [2025 Update]," [Online]. Available: https://www.netguru.com/blog/kotlin-pros-and-cons#:~:text=Less%20buggy [Accessed 07 09 2025].

[13] "Firebase Access data offline," [Online]. Available: https://firebase.google.com/docs/firestore/manage-data/enable-offline#:~:text=Cloud%20Firestore%20supports%20offline%20data,your%20app%20is%20actively%20using [Accessed 09 09 2025].

[14] "Android Developers," [Online]. Available: https://developer.android.com/topic/architecture/data-layer/offline-first#:~:text=match%20at%20L687%20Note%3A%20At,perform%20reads%20without%20network%20access [Accessed 09 09 2025].

[15] "IMO number," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/IMO_number [Accessed 09 09 2025].

[16] "DNV launches app for efficient safety inspections and reporting in ShipManager's QHSE software," [Online]. Available: https://www.dnv.com/news/2021/dnv-launches-app-for-efficient-safety-inspections-and-reporting-in-shipmanager-s-qhse-software-201201/#:~:text=managers%2C%20including%20planned%20inspections%20and,risk%20of%20missing%20important%20information [Accessed 09 09 2025].

[17] "What is a RESTful API?," [Online]. Available: https://aws.amazon.com/what-is/restful-api/#:~:text=RESTful%20API%20is%20an%20interface,applications%20to%20perform%20various%20tasks. [Accessed 07 09 2025].

[18] "RESTful APIs: Principles and Best Practices," [Online]. Available: https://api7.ai/learning-center/api-101/restful-api-best-practices [Accessed 07 09 2025].

[19] "What is Node.js?," DigitalOcean, [Online]. Available: https://www.digitalocean.com/community/tutorials/what-is-node-js [Accessed 12 09 2025].

[20] "Express.js," TutorialsTeacher, [Online]. Available: https://www.tutorialsteacher.com/nodejs/expressjs [Accessed 12 09 2025].

[21] "MySQL: Understanding What It Is and How It's Used," oracle, [Online]. Available: https://www.oracle.com/mysql/what-is-mysql/#:~:text=MySQL%20is%20an%20open%20source,com [Accessed 12 09 2025].

[22] "Firebase SDK," [Online]. Available: https://opensource.google/projects/firebasesdk#:~:text=Firebase%20is%20an%20app%20development%20platform%20that%20provides%20integrated%20tools,idiomatic%20manner%20on%20several%20platforms

[Accessed 12 09 2025].

[23] "What is Postman?," [Online]. Available: https://www.postman.com/use-cases/api-development/ [Accessed 12 09 2025].

[24] "Build Restful CRUD API with Node.js, Express," [Online]. Available: https://www.youtube.com/watch?v=9OfL9H6AmhQ [Accessed 12 09 2025].

[25] A. K. N. S. S. A. S. R. S. Adithya P Shetty, "Smart Fault Detection and Real-Time Alert System with cloud integration for Power Distribution Networks," [Online]. Available: https://tijer.org/tijer/papers/TIJER2505162.pdf [Accessed 15 09 2025].

[26] F. D. V. Dario Bruneo, "Detecting Faults at the Edge via Sensor Data Fusion Echo State Networks," [Online]. Available: https://www.mdpi.com/1424-8220/22/8/2858 [Accessed 15 09 2025].

[27] "Why Vite," Vite, [Online]. Available: https://vite.dev/guide/why [Accessed 15 09 2025].

[28] "How to Use Inspection Data for Predictive Maintenance," [Online]. Available: https://www.thechecker.net/stories/blog/how-to-use-inspection-data-for-predictive-maintenance [Accessed 07 09 2025].

[29] "Machine Learning Algorithms for Predictive Maintenance: A Systematic Literature Mapping," [Online]. Available: https://www.researchgate.net/publication/388624453_Machine_Learning_Algorithms_for_Predictive_Maintenance_A_Systematic_Literature_Mapping [Accessed 07 09 2025].

[30] "Predictive Maintenance For Constant Flow Machinery Using XGBoost," [Online]. Available: https://medium.com/analytics-vidhya/predictive-maintenance-for-constant-flow-machinery-0d3c4bb0defc [Accessed 07 09 2025].