

UNIVERSITY OF BUEA

**Faculty of Engineering and Technology**

CEF440

**INTERNET PROGRAMMING AND MOBILE PROGRAMMING**

**Design and Implementation of a Mobile-Based Archival and Retrieval of Missing Objects Application using Image Matching**

*Submitted to:*

Dr. Nkemini Valery

2023/2024

**Group 17 Members**

1. DJIALEU TANKOUA KEANE RANDY FE21A170
2. MBELI NGO NOEL  FE21A229
3. NANGLEFACK LEODIA FIETSOP FE21A244
4. NKEMZI FOLEFACK GIL FE21A276
5. NSIELA SUHBISI DENZEL FE21A286

**TABLE OF CONTENTS**

[I. Introduction 4](#_Toc169815103)

[A. Project Overview 4](#_Toc169815104)

[B. Importance of Database Design and Implementation 4](#_Toc169815105)

[II. Entity-Relationship Diagram(ERD) 6](#_Toc169815106)

[Entities …………..**Error! Bookmark not defined.**](#_Toc169815107)

[Relationships 7](#_Toc169815108)

[Database Constraints 10](#_Toc169815109)

[III. Firebase Database Implementation 12](#_Toc169815110)

[1. Collections and Documents 12](#_Toc169815111)

[2. Specify Data Types 15](#_Toc169815112)

[3. Implementing Security 15](#_Toc169815113)

[IV. Additional Considerations 16](#_Toc169815114)

[V. Conclusion 17](#_Toc169815115)

[VI. References 18](#_Toc169815116)

# I. Introduction

## A. Project Overview

This report documents the design and implementation of a database system for our mobile application dedicated to the archival and retrieval of missing objects. This application aims to bridge the gap between individuals who have lost personal belongings and those who might have found them. By leveraging a user-friendly mobile interface and a robust database backend, the application will facilitate:

* Efficient reporting of lost objects: Users can easily report lost items, providing details like description, category, and location (optional).
* Secure storage of information: Lost object information will be securely stored in a centralized database, ensuring data integrity and accessibility.
* Enhanced search capabilities: Found object details can be searched based on various criteria, allowing users to find potentially matching items.
* Streamlined communication: The application can facilitate communication between finders and potential owners, expediting the return process.

## B. Importance of Database Design and Implementation

The success of this application hinges on a well-designed and implemented database. The chosen database solution will be responsible for:

* **Storage and Management:** Securely storing detailed information about lost and found objects, including user data and potentially associated images.
* **Scalability:** Efficiently handling a growing number of entries as the user base expands.
* **Search Functionality:** Enabling efficient and accurate searches based on various criteria to find potential matches between lost and found objects.
* **Data Retrieval:** Providing a mechanism to retrieve relevant information about lost and found objects based on user needs.

## C. Importance of Database Design and Implementation

For organizing data in our project, choosing between relational databases (SQL) and NoSQL databases depends largely on the project's requirements. Here’s an explanation of why Firebase (NoSQL) was more suitable for the project:

1. **Flexibility:** NoSQL databases like Firebase are schema-less, allowing for flexible data models. This is beneficial when dealing with data that doesn’t fit well into a structured, predefined schema or when the schema evolves frequently during development.

2. **Scalability**: Firebase is designed to scale horizontally with ease. It can handle large amounts of data and concurrent user connections effectively, making it suitable for our project which expects growth in data size and user base.

3. **Real-time Database:** Firebase offers a real-time database, which is particularly useful for applications that require real-time synchronization across clients. This means changes made by one client are immediately reflected across all clients subscribed to that data. This will help us gain more trust from users, as they get notified on new uploaded missing and found object.

4. **Integration with Other Firebase Services:** For our project that needs backend services such as authentication, storage, or analytics, Firebase provides seamless integration. This can simplify development and maintenance, as all services are managed under one platform.

5. **Development Speed:** NoSQL databases often reduce the time required for initial setup and development iterations due to their flexible nature. This is advantageous for our project with tight deadline.

# II. Data Models

Underlying the structure of a database is the data model, a collection of conceptual tools for describing data, data relationships and consistency constraints. In this report, two data models are outlined.

# 1. Relational model

The relational model is a conceptual way of organizing and structuring data in a database. It uses tables consisting of rows and columns, where each row represents a unique record and each column represents a specific attribute or field of that record.

NoSQL databases like Firebase do not use the relational model primarily because they prioritize schema flexibility, horizontal scalability, and performance characteristics that are more aligned with the needs of modern web and mobile applications. These databases are optimized for handling large volumes of diverse data in a flexible and scalable manner, which often contrasts with the structured and rigid schema requirements of relational databases.

Therefore, this data model was not implemented in our project

# 2. Entity-Relationship Model

An Entity-Relationship Diagram (ERD) is a visual representation of the data entities (tables) in a relational database and the relationships that exist between them. It consists of a base object called entities, which contain attributes and the relationships between this entities. Below are the entities alongside their attributes for our project.

## Entities

Below are the various entities in our system alongside their attributes.

1. **Users:**

-userID,

-username,

-password,

1. **MissingObject:**

-objectID (Pk),

-userID (Fk),

-objectName,

-objectDescription,

-objectImage,

-dateReported,

-locationLastSeen,

-status (e.g “missing” or “found”)

1. **FoundObjects**:

foundObjectId (Pk),

userId (FK),

objectId,

objectImage,

timestamp,

locationFound,

status("claimed", "unclaimed").

1. **Image:**

**-**imageID (Pk),

**-**imageFile,

**-**imageDescription

1. **MatchedImage**:

-matchID (Pk),

-imageID (Fok),

-dateMatched,

-confidenceLevel,

-locationFound

1. **Category:**

-categoryID (Pk),

-categoryName,

-description

1. **Tag**:

-tagID (Pk),

-tagName,

-objectID (Fk)

1. **User Feedback:**

-feedbackID (PK),

-userID (Fk),

-objectID (Fk),

-feedbackText,

-feedbackDate

## Relationships

**1. Users and MissingObject:**

- Each User can report multiple MissingObjects.

- Each MissingObject is reported by exactly one User.

- Relationship: One-to-Many (1 User to Many MissingObjects).

**2. Users and FoundObjects:**

- Each User can claim multiple FoundObjects.

- Each FoundObject is claimed by exactly one User.

- Relationship: One-to-Many (1 User to Many FoundObjects).

**3. MissingObject and FoundObjects:**

- A MissingObject can have zero or one corresponding FoundObject.

- A FoundObject must correspond to exactly one MissingObject.

- Relationship: One-to-One (1 MissingObject to 0 or 1 FoundObject).

**4. Image and MissingObject:**

- Each MissingObject can have multiple images associated with it.

- Each Image can be associated with multiple MissingObjects (if reused for different objects).

- Relationship: Many-to-Many (Many Images to Many MissingObjects via Tag entity).

**5. Image and FoundObjects:**

- Each FoundObject can have one image associated with it.

- Each Image can be associated with multiple FoundObjects (if reused for different objects).

- Relationship: Many-to-One (Many FoundObjects to One Image).

**6. Image and MatchedImage:**

- Each Image can have multiple matched instances in MatchedImage (e.g., for recognition matches).

- Each MatchedImage corresponds to exactly one Image.

- Relationship: One-to-Many (1 Image to Many MatchedImages).

**7. Category and MissingObject:**

- Each MissingObject belongs to one Category.

- Each Category can have multiple MissingObjects.

- Relationship: One-to-Many (1 Category to Many MissingObjects).

**8. Tag and MissingObject:**

- Tags are used to associate additional metadata with MissingObjects.

- Each MissingObject can have multiple Tags.

- Each Tag can be associated with multiple MissingObjects.

- Relationship: Many-to-Many (Many Tags to Many MissingObjects via Tag entity).

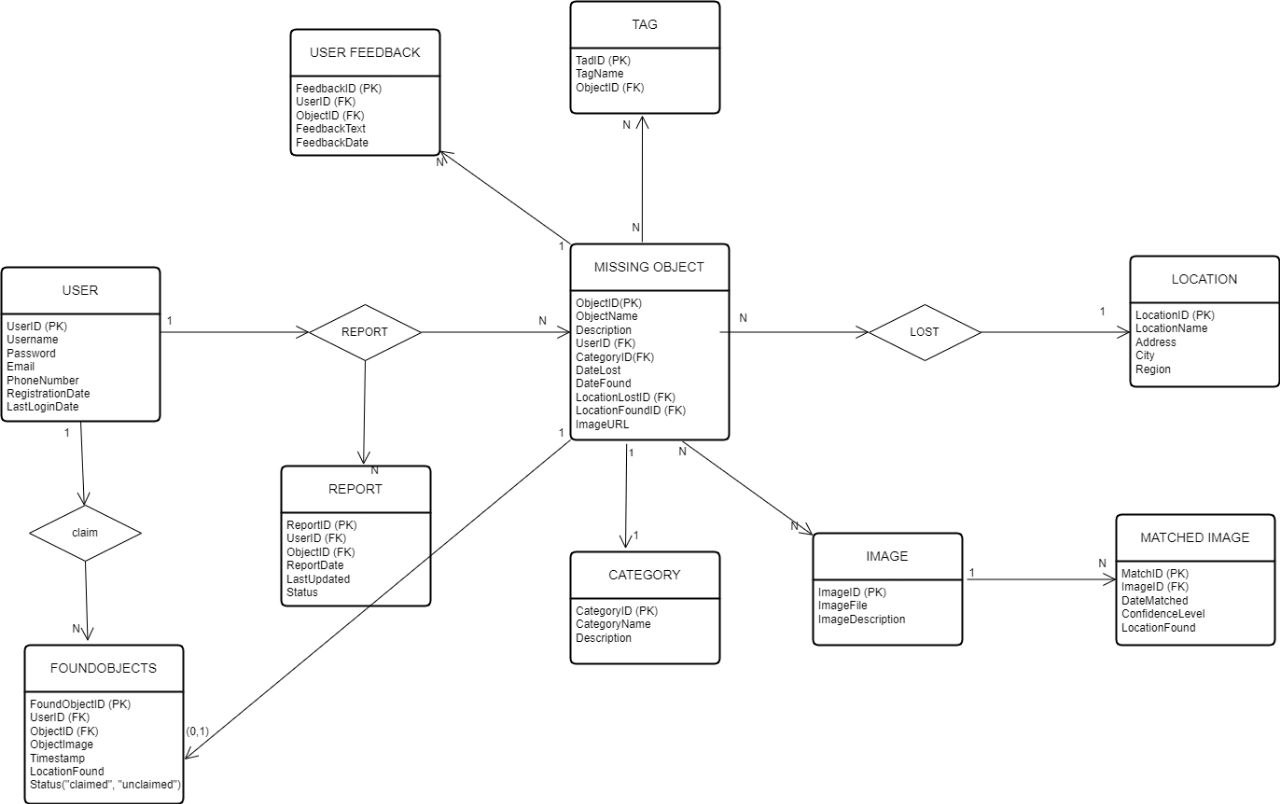
**9. User Feedback and MissingObject:**

- Each MissingObject can have multiple User Feedback entries.

- Each User Feedback entry is linked to exactly one MissingObject.

- Relationship: One-to-Many (1 MissingObject to Many User Feedback entries).

The ER diagram for the above entities and relationships for our project is shown below



## Database Constraints

Based on the entities and relationships above, here are the database constraints

* Users:

- Primary Key Constraint: userID (PK) uniquely identifies each user.

* MissingObject:

- Primary Key Constraint: objectID (PK) uniquely identifies each missing object report.

- Foreign Key Constraint: userID (FK) references Users(userID), ensuring that each MissingObject is reported by a valid user.

* FoundObjects:

- Primary Key Constraint: foundObjectId (PK) uniquely identifies each found object report.

- Foreign Key Constraint: userID (FK) references Users(userID), ensuring that each FoundObject is claimed by a valid user.

- Foreign Key Constraint: objectId (FK) references MissingObject(objectID), ensuring that each FoundObject corresponds to a valid MissingObject.

* Image:

- Primary Key Constraint: imageID (PK) uniquely identifies each image.

- Unique Constraint: imageFile, ensuring each image file is unique.

* MatchedImage:

- Primary Key Constraint: matchID (PK) uniquely identifies each matched image entry.

- Foreign Key Constraint: imageID (FK) references Image(imageID), ensuring that each MatchedImage corresponds to a valid Image.

* Category:

- Primary Key Constraint: categoryID (PK) uniquely identifies each category.

* Tag:

- Primary Key Constraint: tagID (PK) uniquely identifies each tag.

- Foreign Key Constraint: objectID (FK) references MissingObject(objectID), ensuring that each Tag is associated with a valid MissingObject.

* User Feedback:

- Primary Key Constraint: feedbackID (PK) uniquely identifies each user feedback entry.

- Foreign Key Constraint: userID (FK) references Users(userID), ensuring that each User Feedback entry is associated with a valid user.

- Foreign Key Constraint: objectID (FK) references MissingObject(objectID), ensuring that each User Feedback entry is associated with a valid MissingObject.

# III. Firebase Database Implementation

Implementing this database using Firebase was done as follows:

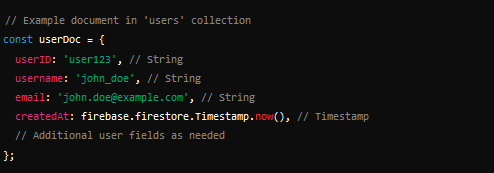
Implementing the database design described using Firebase involves several steps to create collections, define document structures, specify data types, and optionally implement security rules. Here’s a structured approach to implement this in Firebase:

## 1. Collections and Documents

Firebase firestore uses collections to group related documents. Each document contains fields that represent attributes of the entity. We created collections for Users, MissingObject, FoundObjects, Image, MatchedImage, Category, Tag, and User Feedback.

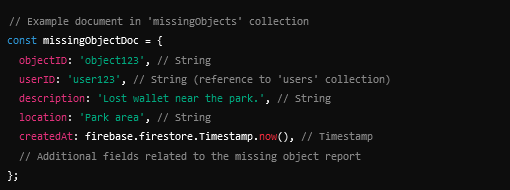
- **Users Collection:**

Each document in the Users collection will represent a user. Fields such as userID, username, email, etc., will be defined.



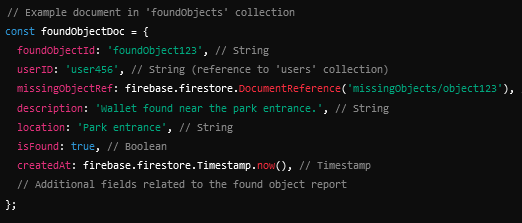
- **MissingObject Collection:**

Documents here represent missing object reports. Fields like objectID, userID, description, location, etc., will be included.



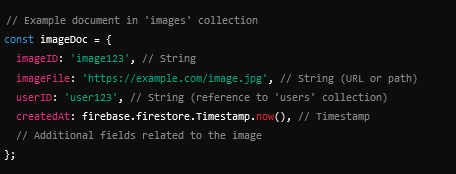
- **FoundObjects Collection:**

Documents represent found object reports. Fields include foundObjectId, userID, objectId (referencing MissingObject), description, location, etc.



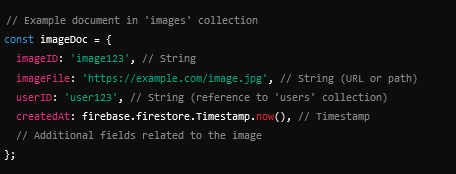
- **Image Collection:**

Each document represents an image. Fields include imageID, imageFile, userID, etc.



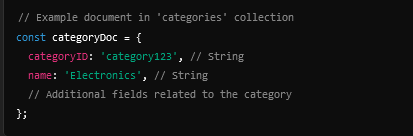
- **MatchedImage Collection:**

Documents represent matched images. Fields include matchID, imageID (referencing Image), etc.



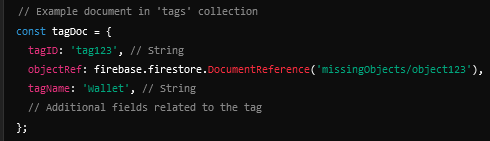
- **Category Collection:**

Documents represent categories. Fields include categoryID, name, etc.



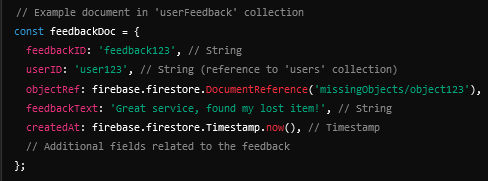
- **Tag Collection:**

Documents represent tags associated with missing objects. Fields include tagID, objectID (referencing MissingObject), tagName, etc.



- **User Feedback Collection:**

Documents represent feedback from users. Fields include feedbackID, userID, objectID (referencing MissingObject), feedbackText, etc.



## 2. Specify Data Types

In Firestore, each field in a document has a data type. It's crucial to define these data types to ensure consistency and validation:

- String: For userID, objectID, imageFile, description, location, username, email, tagName, feedbackText, etc.

- Boolean: For boolean fields if necessary (e.g., isFound in FoundObjects).

- Timestamp: For dates and times, Firebase provides a native Timestamp type.

- Reference: To establish relationships between entities, Firestore supports Reference fields that point to other documents or collections.

## 3. Implementing Security

Firebase allows setting security rules to control access to data based on authentication, user roles, or conditions. Implementing security rules is crucial to ensure data privacy, integrity, and compliance with project requirements:

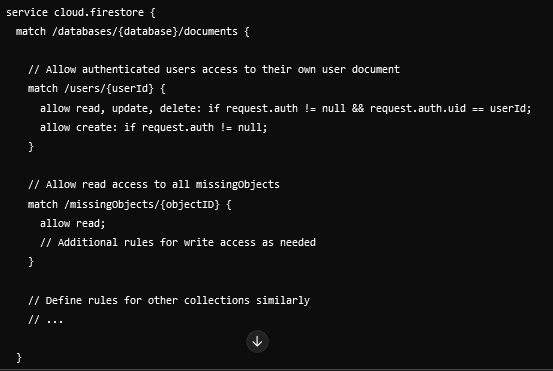
- **Authentication**: Ensure users are authenticated before accessing certain collections or documents (Users, User Feedback, etc.).

- Authorization: Control access based on user roles or specific conditions. For example, allowing users to read FoundObjects only if they are the owner (userID matches the authenticated user) or have specific permissions.

- **Validation**: Ensure data being written meets certain criteria (e.g., non-null values, correct data types).

- **Auditing**: Optionally, logging access or modification attempts for auditing purposes.

This was implemented as shown below



# IV. Additional Considerations

There are some potential future enhancements to consider for the database design implemented in Firebase Firestore. They include:

1. **Advanced Analytics**: Utilize Firebase Analytics or integrate with Google Analytics to gain insights into user behavior, object reporting trends, and system usage. This data can inform future optimizations and feature enhancements. This will be of great use to the admin.

2. **Versioning and History**: Implement versioning or history tracking for objects and feedback to track changes over time. This can be useful for auditing purposes or to revert to previous states if necessary within the application.

3. **Multi-region Support:** When our application grows globally, it is wise to consider configuring Firestore for multi-region support to improve latency and reliability for users in different geographic regions.

# V. Conclusion

Designing the database for our mobile project involves structuring tables, defining relationships, and optimizing for performance and security. This provides a complete structure for implementing the archival and retrieval system efficiently. Adjustments will be made were necessary as time goes on.

# VI. References

1. [*https://inria.hal.science/hal-01092440/document*](https://inria.hal.science/hal-01092440/document)
2. [*https://www.academia.edu/20295246/Database\_Design\_And\_Implementation*](https://www.academia.edu/20295246/Database_Design_And_Implementation)
3. *https://www.researchgate.net/publication/253203812\_Design\_and\_Implementation\_of\_Electronic\_Archive\_of\_Documents*