THE UNIVERSITY OF BUEA
P.O Box 63,
Buea, South West Region
CAMEROON
Tel: (237) 3332 21 34/3332 26 90
Fax: (237) 3332 22 72

REPUBLIC OF CAMEROON

Peace-Work-Fatherland

FACULTY OF
ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

CEF440: INTERNET PROGRAMMING (J2EE) AND MOBILE PROGRAMMING

DESIGN AND IMPLEMENTATION OF A MOBILE-BASED ARCHIVAL AND RETRIEVAL OF MISSING OBJECTS APPLICATION USING IMAGE MATCHING

TASK SIX: **DATABASE DESING AND IMPLEMENTATION**

Course Supervisor:
Dr. NKEMENI VALERY
University Of Buea

GROUP 1 MEMBERS

	NAME	MATRICULE
1	MEWOABI NGUEFACK DORE	FE21A239
2	ALEANU NTIMAEH ENOW	FE21A134
3	OJONG-ENYANG OYERE	FE21A297
4	FONGANG KELUAM PAUL DIEUDONNE	FE21A193
5	TANWIE BRUNO ADEY	FE21A316

Table of Contents

1. INTRODUCTION

1.1. BACKGROUND:

In an era characterized by rapid technological advancement and an ever-increasing reliance on mobile applications, the need for innovative solutions to everyday challenges has never been more pressing. One such challenge is the distressing experience of losing personal belongings, be it a cherished item of sentimental value or a practical necessity crucial for daily routines. The emotional toll and inconvenience caused by such losses are undeniable, often compounded by the arduous process of attempting to retrieve or replace the missing objects.

Recognizing the urgency of addressing this issue, the proposed project sets out to develop a groundbreaking mobile application aimed at revolutionizing the archival and retrieval of missing objects through the power of image matching technology. This application will not only streamline the process of reporting lost items but also enhance the likelihood of successful recovery, thereby alleviating the stress and frustration associated with such incidents.

1.2. DATABASE DESIGN AND IMPLEMENTATION PROCESS:

Designing and implementing the database of a mobile-based application for archival and retrieval of missing objects involves several critical steps to ensure that the data is organized efficiently, supports required operations, and maintains integrity. This process can be broken down into the following stages:

- Logical design
- Physical design
- Implementation
- Testing
- Deployment
- Maintenance and Monitoring

1.2.1. Factors taken into consideration

Our database requirement analysis involves analyzing all requirements gotten from the requirements gathering phase about the database and bringing out a conducive flow of design and implementation of the database.
The following below include our analysis made on the requirements gotten.
User requirements identify the data-related needs of the users based on initial project requirements. These encompass the types of data the system will handle, the volume of data, and the patterns of access. As such we have to design a database that will ease the storage and retrieval of data with respects to functionality that involve:

A. Functional Attributes:

1) **User Registration and Authentication:**
   The database must allow for storing users data such that must be able to sign up with unique email addresses and secure passwords. And this data should be retrieved when Users want to login in to the platform

2) **Image Upload and Capture:**
   Users should be able to upload images of items they found or capture images directly using their device's camera or from their device gallery and our database system should be able to store this images of various formats.
3) **Image Matching:**
   The database should be able to store matching results which can them be sent to users or be used later for analysis
4) **Notifications:**
   It should be able to store notifications sent to uses about information concerning particular item matches (this is not a must be can be used for future reviews and investigations)
5) **Search Capability:**
   Users must be able to search the database for items using different criteria, such as description, metadata, or image matching. Hence the database system should allow for easy retrieval of items or user information as well as other forms of data in the application.
6) **Data Management:**
   The database should be in a way that It allows for seamless modification of data in cases such as user account editing or posts editing and so on.

B. Non-Functional Attributes:

Non-functional requirements discuss performance expectations, security needs, scalability considerations, and other critical non-functional requirements that the database is expected to meet. The involve

1. **Performance**:
   - Low latency for image uploads, searches, and retrievals.
   - High throughput to handle many concurrent users and operations efficiently.

2. **Security:**
   - Data encryption both in transit and at rest to protect sensitive information.
   - Implement strong authentication and authorization mechanisms.
   - Regular security audits and compliance with relevant standards.
3. **Scalability:**
   - Horizontal scalability to handle increasing amounts of data and users.
   - Efficient scaling strategies for both the database and application servers.
4. **Availability:**
   - High availability with minimal downtime. Data should always be available by to users
5. **Maintainability:**
   - Modular design to simplify updates and maintenance.
   - Clear logging and monitoring to track system performance and detect issues early.
6. **Data Integrity:**
   - Ensure data is accurate, consistent, and reliable.
   - Use constraints and validation rules to enforce data integrity.
7. **Compliance:**
   - Adherence to data protection regulations such as GDPR, HIPAA, etc.
   - Implement user data privacy and consent mechanisms.

These functional and non-functional considerations ensure that the database meets user needs while being performant, secure, and scalable. These requirements form the basis for the detailed design and implementation of the system, ensuring a robust and user-friendly application.

2.DATABASE DESGN:

Here, we show how our database will be designed showing the relationships between each table

2.1. Conceptual Design

The conceptual design here involves our ER diagram/relational schema detailing entities, attributes and relationships between these entities

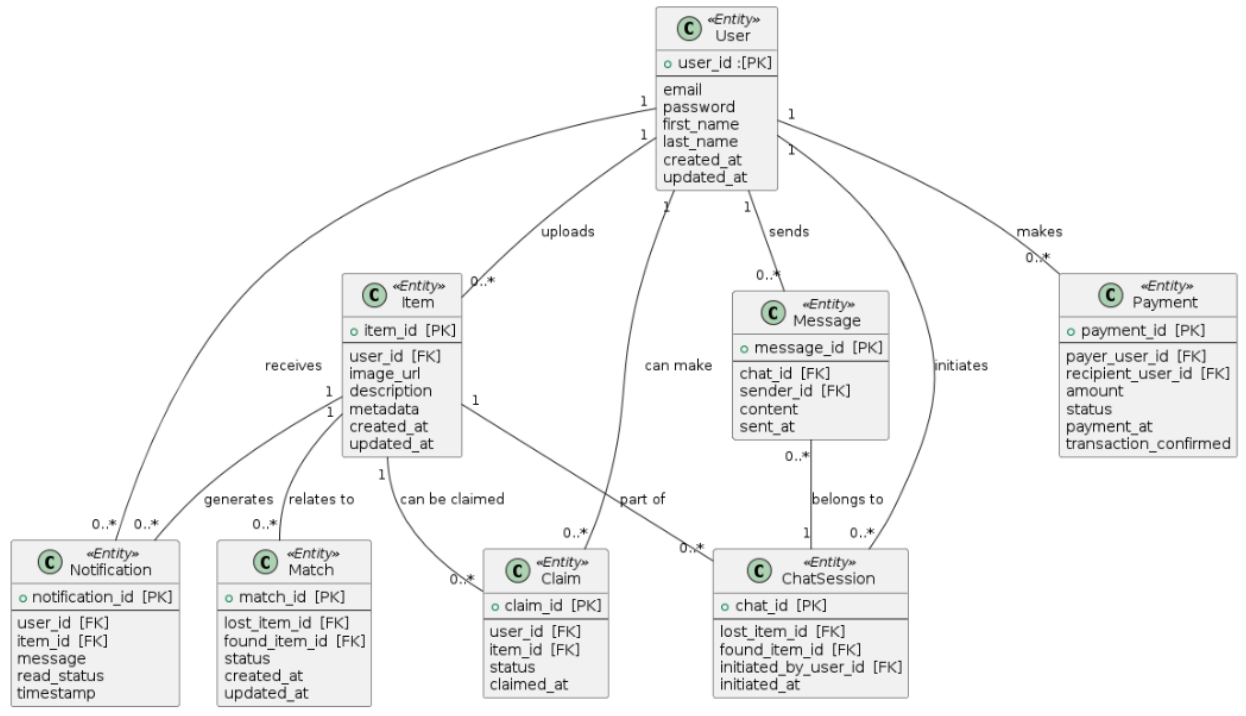


Fig1.1 relation schema

2.2. Logical Design

Schema Definition: Here, we describe the structure of the database including the table definitions column specification and constraints

Here is a detailed description of the database structure, including table definitions, column specifications, and constraints based on the provided ER diagram:

i. USER ENTITY:

- **Columns:**
  - user_id: A unique identifier for each user (Primary Key).
  - email: The email address of the user (must be unique).
  - password: The hashed password of the user.
  - first_name: The first name of the user.
  - last_name: The last name of the user.
  - created_at: The timestamp when the user was created.
  - updated_at: The timestamp when the user was last updated

- **Constraints:**
  - user_id is the primary key.
  - email must be unique and not null.
  - password must not be null.

ii. ITEM ENTITY

- **Columns:**
  - item_id: A unique identifier for each item (Primary Key).
  - user_id: A foreign key referencing the user_id in the User table.
  - image_url: The URL of the item's image (must not be null).
  - description: A description of the item.
  - metadata: Additional metadata about the item, stored as JSON.
  - created_at: The timestamp when the item was created.
  - updated_at: The timestamp when the item was last updated.

- **Constraints:**
  - item_id is the primary key.
  - image_url must not be null.
  - user_id is a foreign key referencing User(user_id).

MATCH ENTITY

- **Columns:**
  - match_id: A unique identifier for each match (Primary Key).
  - lost_item_id: A foreign key referencing the item_id in the Item table.
  - found_item_id: A foreign key referencing the item_id in the Item table.
  - status: The status of the match.
  - created_at: The timestamp when the match was created.

- updated_at: The timestamp when the match was last updated.

- **Constraints:**
  - match_id is the primary key.
  - lost_item_id and found_item_id are foreign keys referencing Item(item_id).

iii. NOTIFICATION ENTITY

- **Columns:**
  - notification_id: A unique identifier for each notification (Primary Key).
  - user_id: A foreign key referencing the user_id in the User table.
  - item_id: A foreign key referencing the item_id in the Item table.
  - message: The notification message.
  - read_status: A boolean indicating whether the notification has been read.
  - timestamp: The timestamp when the notification was created.

- **Constraints:**
  - notification_id is the primary key.
  - user_id and item_id are foreign keys referencing User(user_id) and Item(item_id) respectively.

iv. CLAIM ENTITY

- **Columns:**
  - claim_id: A unique identifier for each claim (Primary Key).
  - user_id: A foreign key referencing the user_id in the User table.
  - item_id: A foreign key referencing the item_id in the Item table.
  - status: The status of the claim.
  - claimed_at: The timestamp when the claim was made.

- **Constraints:**
  - claim_id is the primary key.
  - user_id and item_id are foreign keys referencing User(user_id) and Item(item_id) respectively.

v. CHAT ENTITY

- **Columns:**
  - chat_id: A unique identifier for each chat session (Primary Key).
  - lost_item_id: A foreign key referencing the item_id in the Item table.
  - found_item_id: A foreign key referencing the item_id in the Item table.
  - initiated_by_user_id: A foreign key referencing the user_id in the User table.

- initiated_at: The timestamp when the chat session was initiated.

- **Constraints:**
  - chat_id is the primary key.
  - lost_item_id, found_item_id, and initiated_by_user_id are foreign keys referencing Item(item_id) and User(user_id) respectively.
  -

vi. MESSAGE ENTITY

- **Columns:**
  - message_id: A unique identifier for each message (Primary Key).
  - chat_id: A foreign key referencing the chat_id in the ChatSession table.
  - sender_id: A foreign key referencing the user_id in the User table.
  - content: The content of the message.
  - sent_at: The timestamp when the message was sent.

- **Constraints:**
  - message_id is the primary key.
  - chat_id and sender_id are foreign keys referencing ChatSession(chat_id) and User(user_id) respectively.

vii. PAYMENT ENTITY

- **Columns:**
  - payment_id: A unique identifier for each payment (Primary Key).
  - payer_user_id: A foreign key referencing the user_id in the User table.
  - recipient_user_id: A foreign key referencing the user_id in the User table.
  - amount: The amount of the payment.
  - status: The status of the payment.
  - payment_at: The timestamp when the payment was made.
  - transaction_confirmed: A boolean indicating whether the transaction has been confirmed.

- **Constraints:**
  - payment_id is the primary key.
  - payer_user_id and recipient_user_id are foreign keys referencing User(user_id).

These definitions provide a comprehensive structure for the database, ensuring all relationships and constraints are properly defined to maintain data integrity and support the system's functionalities.

2.2.1. Normalization

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. The goal is to divide large tables into smaller, related tables and ensure that the relationships among these tables are well-defined. Here, I'll explain how normalization can be applied to the database schema provided for the image matching system.

Normalization Steps:

i. First Normal Form (1NF)

1NF dictates that a table must contain only atomic (indivisible) values and each column must contain values of a single type.

- User Table:

- Ensure each attribute (email, password, first_name, etc.) contains only a single value per record.

- Item Table:
  - Each item has a single image_url and description.
  - metadata can be stored as JSON, ensuring it's a single entity per record.

- Match Table:
  - Contains atomic values for lost_item_id, found_item_id, etc.

- Notification Table:
  - Each notification has a single message, read_status, etc.

- Claim Table:
  - Contains single values for status, claimed_at, etc.
  - ChatSession Table:
  - Each chat session has atomic values for initiated_by_user_id, etc.

- Message Table:
  - Each message contains atomic values for content, sent_at, etc.

- Payment Table:
  - Contains single values for amount, status, etc.

ii. Second Normal Form (2NF):

2NF requires that the table is in 1NF and that all non-key attributes are fully dependent on the primary key. This eliminates partial dependency.

- User Table:
  - Already in 2NF as all non-key attributes are dependent on user_id.

- Item Table:
  - image_url, description, metadata, created_at, and updated_at are fully dependent on item_id.

- Match Table:
  - lost_item_id, found_item_id, status, created_at, and updated_at are fully dependent on match_id.

- Notification Table:
  - message, read_status, and timestamp are fully dependent on notification_id.

- Claim Table:
  - status and claimed_at are fully dependent on claim_id.

- ChatSession Table:
  - initiated_by_user_id and initiated_at are fully dependent on chat_id.

- Message Table:
  - content and sent_at are fully dependent on message_id.

- Payment Table:
  - amount, status, payment_at, and transaction_confirmed are fully dependent on payment_id.

iii. Third Normal Form (3NF):

3NF requires that the table is in 2NF and that all the attributes are not only fully functionally dependent on the primary key, but also non-transitively dependent. This means that no non-key attribute is dependent on another non-key attribute.

- User Table:
  - Already in 3NF as there are no transitive dependencies.

- Item Table:
  - Already in 3NF, no transitive dependencies.

- Match Table:
  - Already in 3NF, no transitive dependencies.

- Notification Table:
  - Already in 3NF, no transitive dependencies.

- Claim Table:
  - Already in 3NF, no transitive dependencies.

- ChatSession Table:
  - Already in 3NF, no transitive dependencies.

- Message Table:
  - Already in 3NF, no transitive dependencies.

- Payment Table:
  - Already in 3NF, no transitive dependencies.

By applying the normalization process, the database schema ensures the following:

1. Reduction of Redundancy: By splitting data into related tables and ensuring atomic values, redundancy is minimized. For example, user information is stored in a single User table, avoiding repetition in other tables.

2. Elimination of Update Anomalies: By ensuring that all non-key attributes are fully and non-transitively dependent on the primary key, update anomalies are eliminated. For instance, updating a user's email address only requires a change in the User table.

3. Data Integrity: Using foreign keys, relationships between tables are maintained, ensuring referential integrity. For example, each item_id in the Item table is linked to a user_id in the User table, maintaining a valid relationship.

By adhering to these normalization rules, the database design becomes robust, efficient, and easier to maintain, ensuring data consistency and integrity.

3.DATABASE IMPLEMENTAION

3.1. choice of technologies (MongoDB & Firebase)

When using MongoDB and Firebase as databases for the image matching and image processing system, several considerations need to be considered for physical design, including storage considerations, indexing strategies, and partitioning and sharding.

3.1.1. Storage considerations:

1. Data Storage in MongoDB:

- Document Model: MongoDB stores data in a flexible, JSON-like format called BSON (Binary JSON). Each record is stored as a document, which is more aligned with the object-oriented programming paradigm, allowing for complex data structures to be stored easily.

- Collections: Like tables in relational databases, collections in MongoDB group documents. Each collection will represent entities such as Users, Items, Matches, Notifications, Claims, ChatSession, Messages, and Payments.

2.Data Storage in Firebase:

- Firebase Storage is used for storing images and other binary data. It is optimized for storing and serving user-generated content. Firebase also provides a database called Firetore database. But in our case since our application has a lot of backend logic to be implemented on a backend server side, and firebase by nature is backend as a service, meaning interactions with firebase would need to either be done on the frontend, or on the backend with and since we wanted to keep the frontend clean for UI only, we decided to use a mongoDB database for normal storage and firestore only for image storing.

3.1.2. Partitioning and Sharding

1. MongoDB Indexing:

- Single Field Index: Common for fields that are frequently queried, such as user_id, email, and item_id.

- Compound Index: Useful for queries that filter on multiple fields. For example, an index on {user_id: 1, created_at: -1} would speed up queries that retrieve a user's items ordered by the creation date.
- Text Index: For fields that require text search capabilities, such as item descriptions or messages.
- TTL Index: Time-To-Live indexes can be used for data that needs to expire after a certain period, such as notifications.
- Geospatial Index: If location-based features are reintroduced, geospatial indexes can be used for efficient querying of spatial data.

3.1.3. Partitioning and Sharding

1. MongoDB Sharding:

- Sharding: MongoDB supports horizontal scaling by sharding, which involves distributing data across multiple machines. This is particularly useful for handling large datasets and high-throughput operations.
- Shard Key: Choosing an appropriate shard key is crucial. For example, sharding the Item collection by user_id ensures that user-specific data is distributed across shards, improving performance for user-centric queries.
- Replica Sets: Each shard in MongoDB is a replica set, providing redundancy and high availability. This ensures that even if a shard goes down, data is still accessible.

2. Firebase Partitioning:

- Storage Buckets: Firebase Storage allows for partitioning data into different storage buckets, which can be used to segment data logically or geographically.

We can recapitulate by stating that:

- Storage Considerations: Ensure data is stored efficiently using the appropriate document model, collections, and file storage options in MongoDB and Firebase.
- Indexing Strategies: Implement single field, compound, text, TTL, and geospatial indexes in MongoDB, and utilize automatic and composite indexes in Firestore to optimize query performance.

- Partitioning and Sharding: Use sharding in MongoDB with carefully chosen shard keys and replica sets for high availability. In Firebase, organize data into subcollections and leverage automatic sharding while ensuring even data distribution.

By carefully considering these physical design aspects, the database can be optimized for performance, scalability, and reliability, ensuring it meets the needs of the image matching and image processing system.

3.2. Setting up the environments

3.2.1. Setting up MongoDB Atlas

MongoDB Atlas in an Online Database service which provides with free already hosted database creation services for both development and production purposes. To setup mongoDB for usage in our application, the following steps where followed

1. Creating an account on the mongoDB official website at www.mongodb.com
2. After creating the account by and successfully accessing the dashboard one has to go to the side menu by the left and access the database section, where they have to create a new cluster

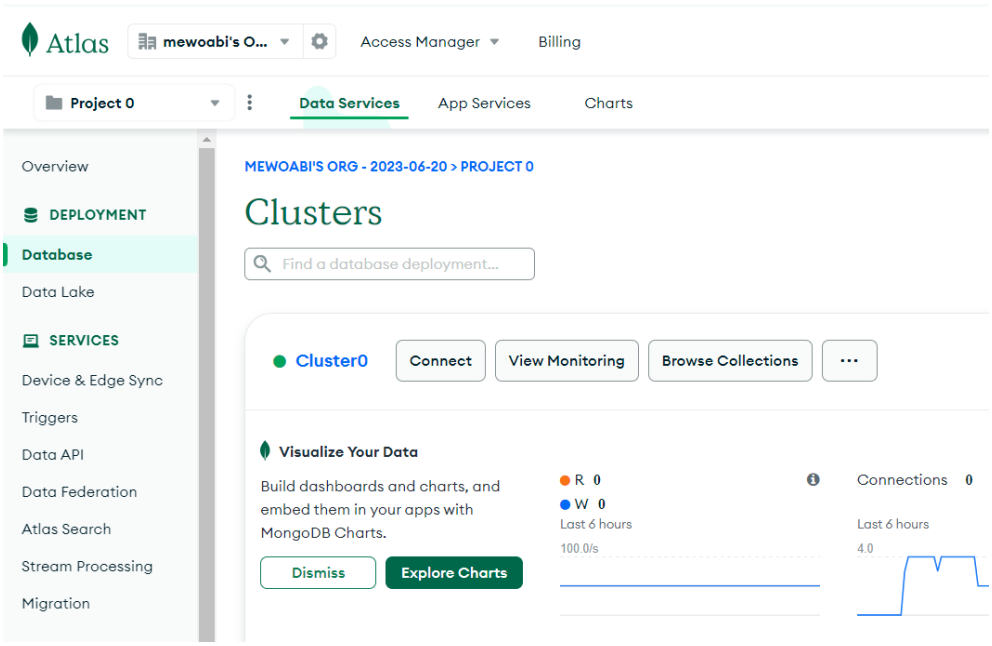


Fig3.1 mongoDB dashboard showing the database section and the created cluster as well as the collections sections and the connect button

3. After the cluster created, you are going to get instructions to create a new database followed by a first collection.

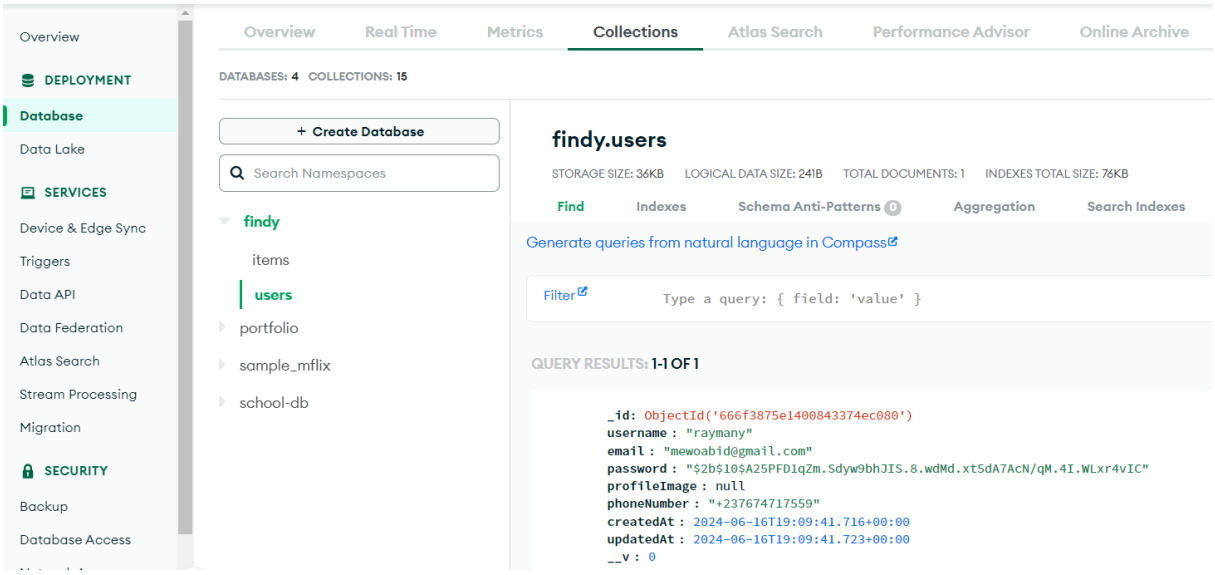


Fig3.2. the collections screen showing our database (findy) and it's collections

4. When the new database and the first collection is created, one has to return to the overview tab on the databases section still and click on the connect button, after which a string will be provided (the connection string) for use in the application.
5. This connection string requires your username and password from mongodb be substituted at some regions for appropriate connection in your application.
6. The connection string can the be used in your backend code at a private location such as the .env file for secure access to the database.

Fig3.3. the .env file in backend with the connection string and port number

7. This connection string can then be used in the main file of the backend to create a connection to the online mongoDB instance

```
// connect to db
mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => {
    // listen for requests
    app.listen(process.env.PORT, () => {
      console.log('connected to db & listening on port', process.env.PORT)
    })
  })
  .catch((error) => {
    console.log(error)
  })
```

8. When the backend development server is started, if all was done correctly the application should be successfully connected to the mongodb backend as seen below.

```
PS E:\projects\SchoolProjects\CEF440-group-1\findy_backend> nodemon server
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`

  New version of nodemon available!
  Current Version: 2.0.22
  Latest Version: 3.1.4

connected to db & listening on port 5500
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
connected to db & listening on port 5500
```

3.2.2. Setting up Firebase Storage

1. If one doesn't already have an account, then a new account has to be created at the official firebase website
2. After the account is created, you are going to be redirected to the console from where one can see all their projects as well as the option to create a new project

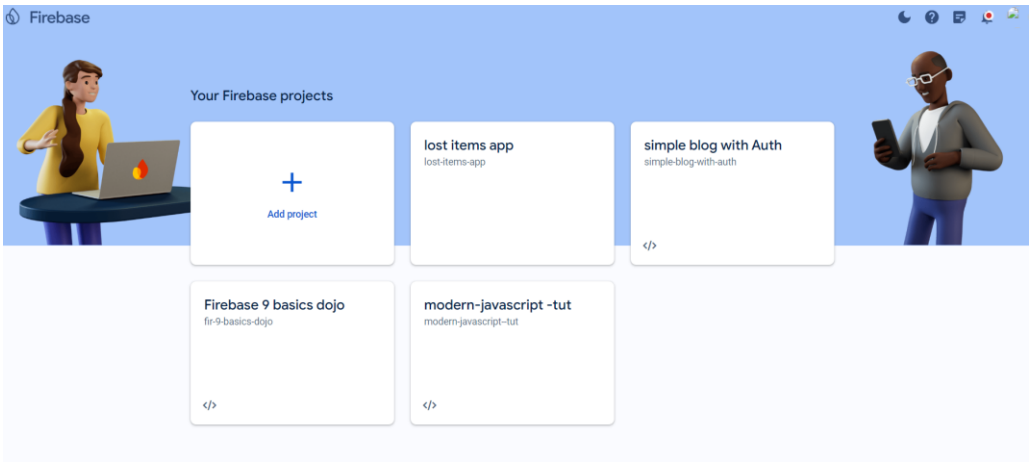


Fig3.6 firebase console

3. When the option to create a new project is chosen, you are going to be prompted to enter the project name after which the project is going to be created.
4. From the side bar at the left, a list of features or services can be chosen for usage in your project, in this case, firebase storage is the chosen service since it is the image storing functionality we are striving to archieve

5. When this is done, a list of buttons with icons can be seen on the dashboard for the service which are indeed connection options to different kinds of projects, in this case the web option shall be chosen (not the native options since we are not writing native code).
6. When this is done, a configuration snippet shall be given for us to paste in our code base

Fig3.7 firebase config snippet

7. This configuration object can then be pasted in a file in our frontend directory and import the firebase storage package from the firebase library installed using npm as seen below

findy_frontend > connections > firebaseConfig.ts > ...

```ts
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
import {getStorage} from "firebase/storage"
import { getAnalytics } from "firebase/analytics";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
const firebaseConfig = {
  apiKey: "AIzaSyBVsw-I0CrSnepdFbSHWHvyGAwKzaJ1bHU",
  authDomain: "lost-items-app.firebaseapp.com",
  projectId: "lost-items-app",
  storageBucket: "lost-items-app.appspot.com",
  messagingSenderId: "375746041352",
  appId: "1:375746041352:web:77622976fceb8fd10963bd",
  measurementId: "G-MHBS710ZGN"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
const analytics = getAnalytics(app);
export const storage = getStorage(app)

```

Fig3.8. Importing the firebase storage and creating a storage object using firebase SDK and config

8. In this case one has to install the package “firebase” using the command “npx expo install firebase” to get this dependency ready for usage.
9. And last but not the list one has to create and configure he metro.config.js file using the command
   “npx expo customize metro.config.js”

findy_frontend > metro.config.js > ...

```js
// Learn more https://docs.expo.io/guides/customizing-metro

const { getDefaultConfig } = require('@expo/metro-config');

const defaultConfig = getDefaultConfig(__dirname);
defaultConfig.resolver.sourceExts.push('cjs');

module.exports = defaultConfig;

```

Fig3.9 metro.config.js file configuration

10. With this done, firebase storage is ready for use in the application

3.3. Code implementation.

3.3.1. Implementation on the backend (mongodb)

As far as database implementation on the backend is concerned, there are three main points of focus which are to be elaborated on. They include,

1. The folder organization structure,
2. Defining the models
3. Defining the controllers
4. Setting up of routes
5. Route consumption

I. Folder Structure

This is simply the way folders and files where organized in the project and the relationship between the different folders that facilitate over all backend communication.

In this project, in the root backend directory where the package.json file and the .env files are, the main folders or directories are

- Models
- Controllers
- Routes
- Middleware

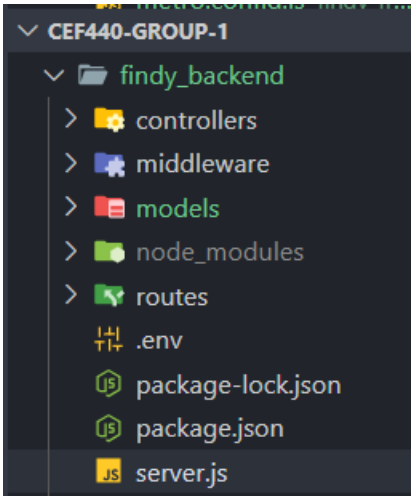


Fig3.10 backend file structure

II. Models :

Each file in this folder corresponds to a collection in the database and describes or enforces the structure of each document in that collection. It defines the properties, their types, constraints and other validation aspects of these documents and all the properties in it. It also creates a reference that can be used to access that particular collection from the controller. It can also include some functions that are commonly used to perform general actions on that collection (such as singup and login functions incase of the users collection)

Defining the structure of the documents is done by creating a schema. This was done using a package called Mongoose as will be seen in the image below.

For our project in question, the models file for the users collection is as shown in the figure below

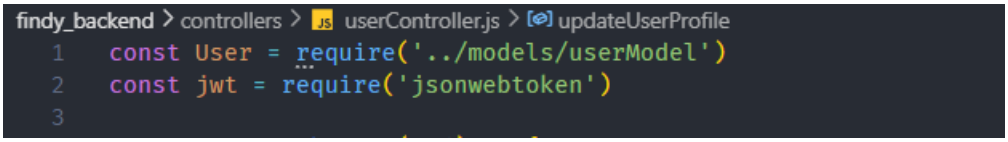
```
const mongoose = require('mongoose')
const bcrypt = require('bcrypt')
const validator = require('validator')

const Schema = mongoose.Schema

const userSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    minlength: 3
  },
  email: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    lowercase: true,
    match: [/\S+@\S+\.\S+/, 'is invalid']
  },
  password: {
    type: String,
    required: true
  },
  profileImage: {
    type: String,
    default: null // URL to the profile image
  },
  phoneNumber: {
      type: String,
      trim: true,
      default: null
  },
  createdAt: {
    type: Date,
    default: Date.now
  },
  updatedAt: {
    type: Date,
    default: Date.now
  }
})
```

Special attention has to be taken to make sure that the schema matches the logic database derived in the design phase as can be seen above.

At the bottom of the models file, the created schema or model is exported for use in the controller.

```
module.exports = mongoose.model('User', userSchema)
```

It can then be imported in the controller and used as references for performing various queries



III. controllers:

In this folder, each file contains the function for querying a particular collection of the database. In the user's controller file for example, we can see function for creating a new user and in the item's collections. All functions in the controller can later be exported used in the route file for that same collection to be called when an API request from the frontend end is made to that route.

findy_backend > controllers > itemController.js > getAllUserPosts

```
// create new item
const createItemPost = async (req, res) => {
  const { title, name, description, additionalInfo, category, location, reporter, type, imageUrl } = req.body
  let userId
  if (req.user && req.user._id) {
    userId = req.user._id
  } else {
    userId = reporter;
  }

  let emptyFields = []

  if (!title) {
    emptyFields.push('title')
  }
  if (!name) {
    emptyFields.push('name')
  }
  if (!description) {
    emptyFields.push('description')
  }
  if (!category) {
    emptyFields.push('category')
  }
  if (!location) {
    emptyFields.push('location')
  }
  if (!imageUrl) {
    emptyFields.push('imageUrl')
  }
  if (emptyFields.length > 0) {
    return res.status(400).json({ error: 'Please fill in all the fields', emptyFields })
  }

  // add doc to db
  try {
    const item = await Item.create({ title, name, description, additionalInfo, category, location, reporter: userId, type, status: 'unreturned', imageUrl })
    res.status(200).json(item)
  } catch (error) {
    res.status(400).json({ error: error.message })
    console.log(error.message)
  }
}
```

Fig 3.14. item controller file showing the function for creating new item post

```
const updateUserProfile = async (req, res) => {

  try {
    const {id} = req.params
    console.log("before update: ", req.body,req.params.id)
    const user = await User.findOneAndUpdate({_id: id}, {...req.body}, {new: true, runValidators: true})
    // create a token
    // console.log(user)
    res.status(200).json(user)
  } catch (error) {
    res.status(400).json({error: error.message})
    console.log(error.message)
  }
}



module.exports = { signupUser, loginUser, updateUserProfile }
```

Fig 3.15. item controller file showing the function for updating a user profile and showing the function exports

IV. Routes :

Each file in this folder contains the endpoints which are to be called from the frontend to perform different queries. Each endpoint is associated to a particular controller function for that same entity or collection in the database. These endpoints are then exported and can be imported in the main backend file to be properly exposed as API routes to the frontend for frontend requests to be made. The users routes file is a seen below

Fig3.16 users route file showing the controller function import and the endpoints for the different request types for login in a user, signing up and updating a user profile

V. API exposure :

This is done in the running backend file, called “server.js” in our project where the connection to the mongodb database is done. Here prefixes are attached to the endpoints from the distinct route folders to group them depending on concerns. This makes querying from the frontend clearer and more logical . In this folder, other measures are taken such as implementing CORS policy to ensure that there is no cross domain restriction policy when trying to query the database from the frontend through the backend. The server.js file can be seen below:

```
require('dotenv').config()
const cors = require("cors")
const express = require('express')
const mongoose = require('mongoose')
const itemRoutes = require('./routes/item')
const userRoutes = require('./routes/user')

// express app
const app = express()
mongoose.set('strictQuery', false)

// middleware
app.use(express.json())
app.use(cors());
app.use((req, res, next) => {
  console.log(req.path, req.method)
  next()
})

// routes
app.use('/api/item', itemRoutes)
app.use('/auth/user', userRoutes)

// connect to db
mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => {
    // listen for requests
    app.listen(process.env.PORT, () => {
      console.log('connected to db & listening on port', process.env.PORT)
    })
  })
  .catch((error) => {
    console.log(error)
  })
```

Fig3.17. server.js file showing connection to mongo dB database and consumption of api routes for exposure to the frontend.

3.3.2. Implementation on the frontend (firebase storage)

This has to deal with connecting the application to the firebase storage such that images can be successfully sent and retrieved from the database. The main procedure in completing this has to deal with setting up the environment which was already seen above. Test code was later on written to ensure that connection and storing of images was actually possible with the firebase database. This shall be seen in the testing section coming up next

3.4. Testing .

Testing was done both for the mongo db database and the firebase storage database to ensure that queries and communication could be properly done between our application and these database

3.4.1. Testing the mondo db database

Here “Postman” was used as testing software to hit the different endpoints by sending sample data for creating different documents and performing different operations on these databases

The images below demonstrate the testing process with postman on the user and item collections using postman on the database for querying and performing different operations in our applicaton.

a. signup a user

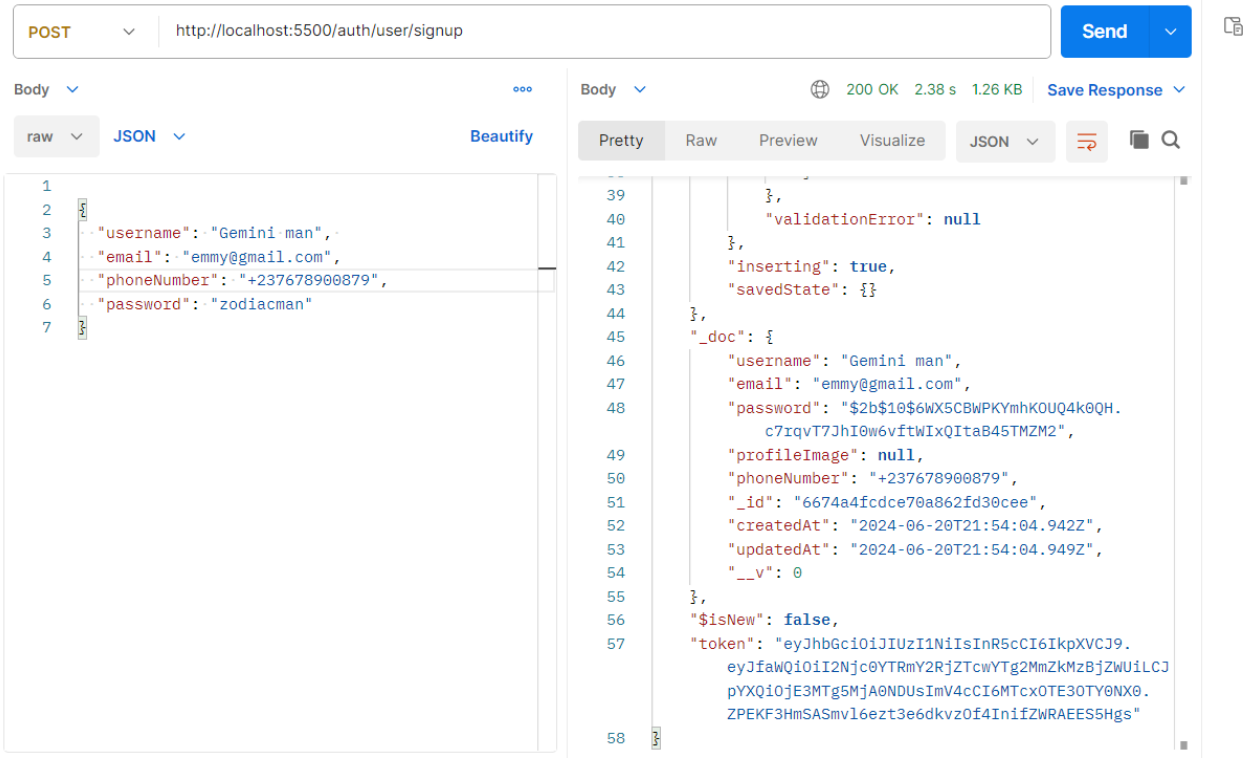


Fig3.18. showing signup test with user information and return user object plus authentication token

b. *create Item post*

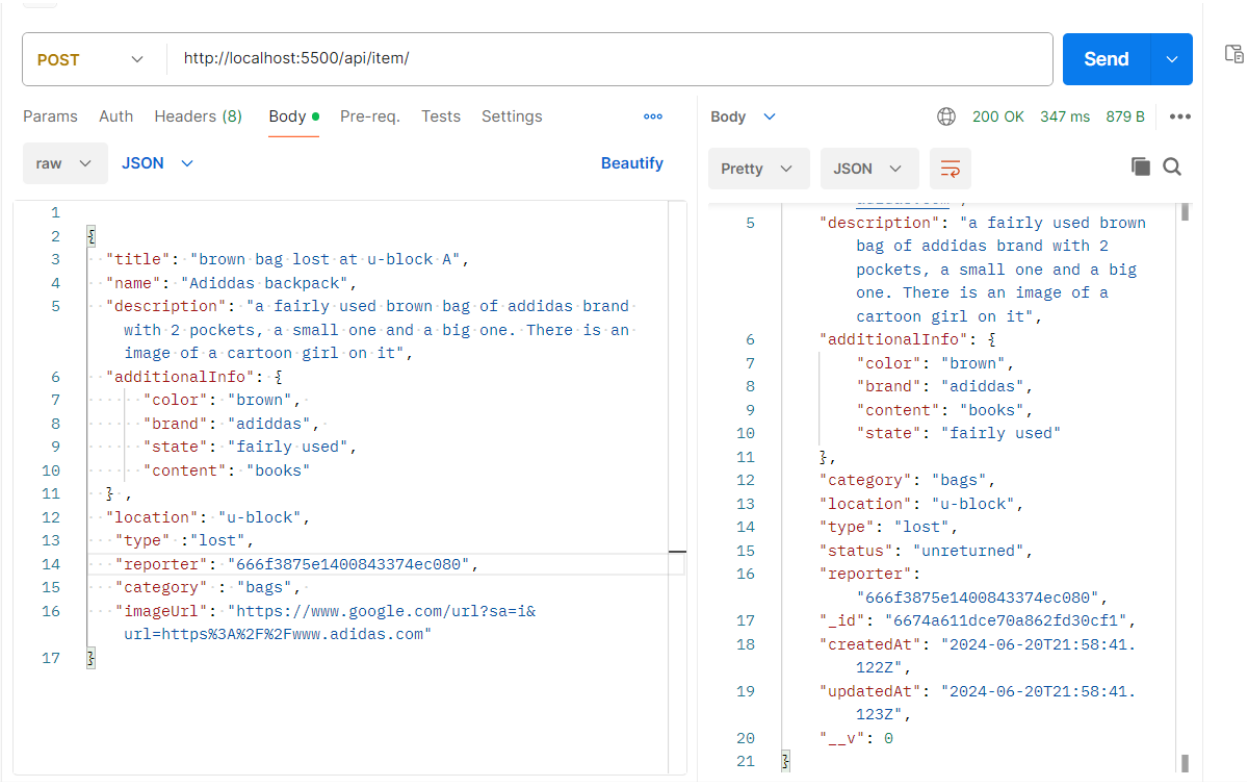


Fig 3.19. showing successful createItemPost test with the return item by the right (includes the item _id from mongodb)

b. confirming from mongodb atlas

This was later on done to confirm that the test data sent was actually received by the database and as seen below it was accurately stored onto mongodb atlas

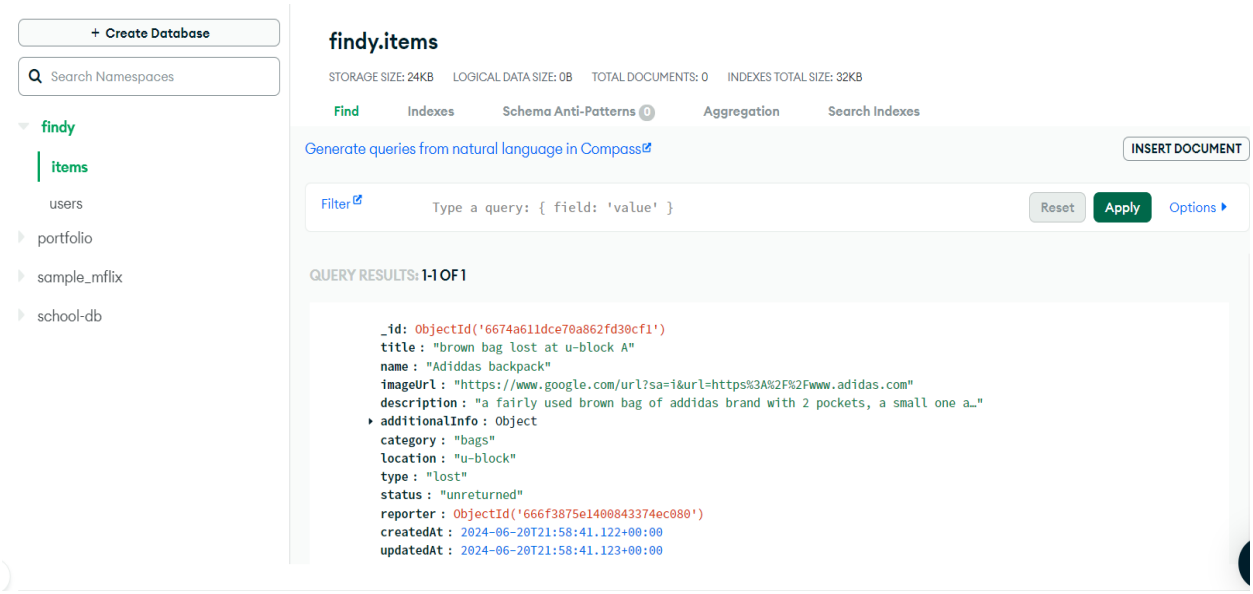


Fig 3.20. showing successful createItemPost stored on mongo db atlas in the items collection

3.4.2. Testing the firebase frontend

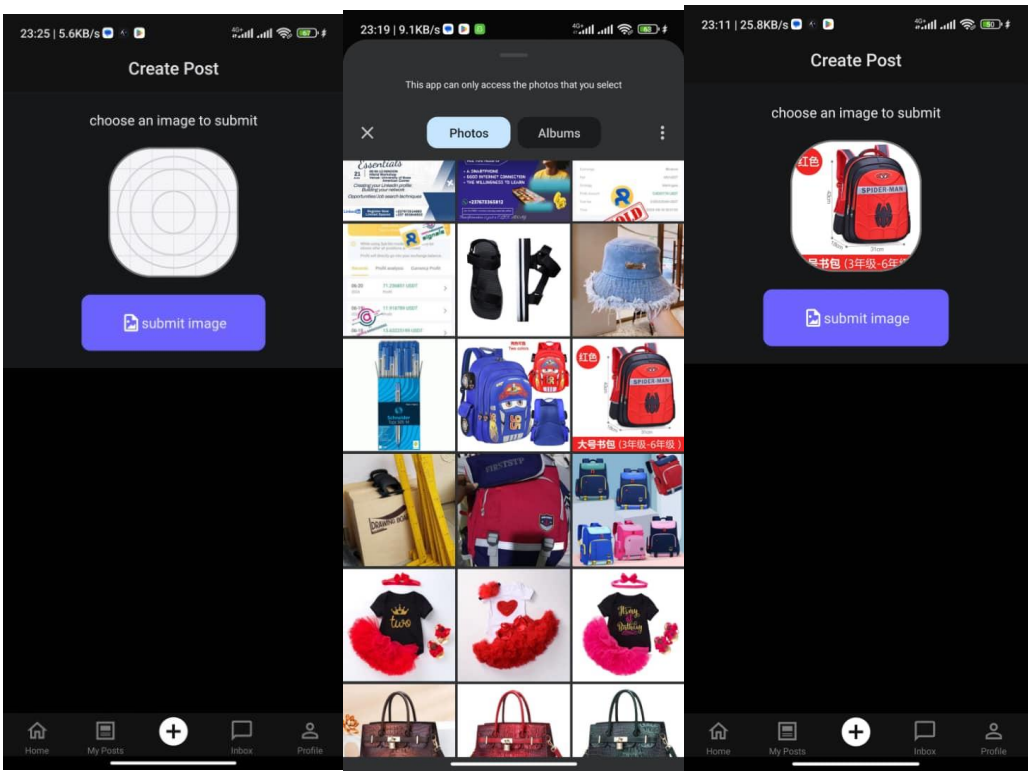
Here we use “Expo-Go” and created a dummy application to simulate the process of uploading an image. Where we first get the image from the device, then upload it to firebase storage and use the url provided by firebase storage to display the image

The code is as seen below

```
export default function createPage() {
    //state for storing the image once it is gotten from the camera
    const [image, setImage] = useState<string>('')
    const chooseFromGallery = async() => {
       try {
        await imagePicker.requestMediaLibraryPermissionsAsync();
        let result = await imagePicker.launchImageLibraryAsync({
            mediaTypes: imagePicker.MediaTypeOptions.Images,
            allowsEditing: true,
            aspect: [1,1],
            quality: 1
        })
        if(!result.canceled) {
            //save image
            console.log("this is the gallery image result: ",result)
            setImage(result.assets[0].uri)
        }
       } catch (error: any) {
        console.log("image choosing was cancelled",error)
        alert("error uploading image " + error.message)
       }
    }
    const saveImageToStorage = async (uri: string) => {
        if(uri)
        try {
            const response = await fetch(uri)
            const blob = await response.blob()

            const storageRef = ref(storage, 'itemImages/' + new Date().getTime())
            const uploadTask = uploadBytesResumable(storageRef, blob)

            uploadTask.on("state_changed", (snapshot) => {
                const progress = (snapshot.bytesTransferred / snapshot.totalBytes) * 100 ;
                console.log("you upload is " + progress + "% done")

            },
            (error) => {
                console.log(error.message)
            },
            () => {
                getDownloadURL(uploadTask.snapshot.ref).then(async (donwloadUrl) => {
                    console.log("file availabe at: " + donwloadUrl)
                })
    }
    return (
        <ThemedView style={styles.container}>
            <ThemedText style={styles.textmargin}>choose an image to submit</ThemedText>
            <TouchableOpacity onPress={() => chooseFromGallery()}>
                <Image style={styles.image} source={image ? {uri: image} : defaultImg} />
            </TouchableOpacity>
            <TouchableOpacity style={styles.button} onPress={() => saveImageToStorage(image)}>
                <MaterialCommunityIcons name="file-image-outline" size={24} color="white" />
                <ThemedText>submit image</ThemedText>
            </TouchableOpacity>
        </ThemedView>
    )
}
```

Code for uploading image from device and storing on firebase storage

When the above code was run and executed, we could successfully store the image and display it as seen below.

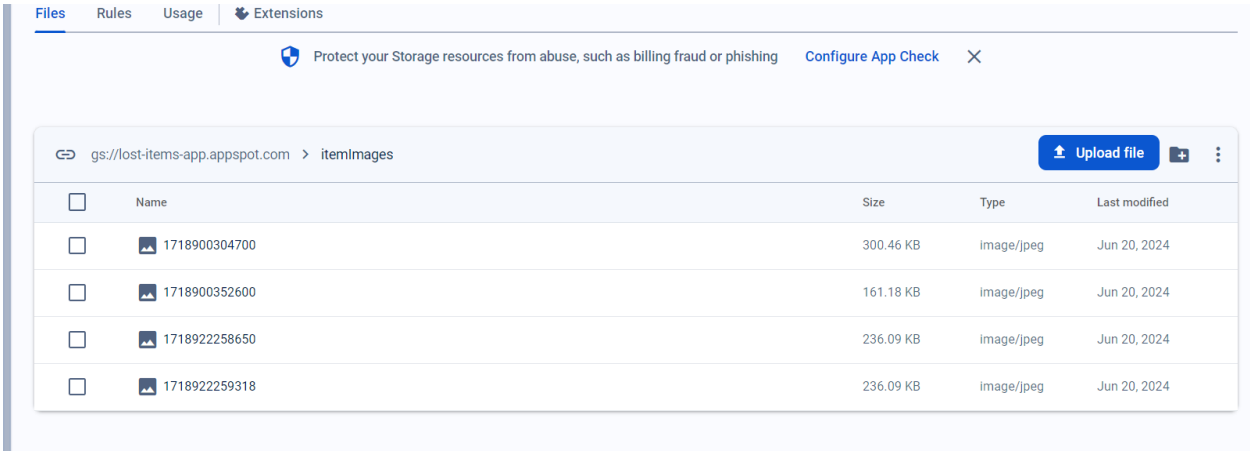


Test interface for orchestrating and testing firebase storage database

Initially the default icon is shown on the image placeholder.

When the image icon is clicked, it opens the gallery for the user to choose and image

When the image is chosen and the submit button pressed , it saves the image to firebase storage successfully as seen below



Firebase storage database showing images saved from test interface

3.5. Security .

This was done in the middleware folder by the require Auth file. In this file interception code is written to intercept any request made to the database to check if the user is an authenticated user and if the user is not, the request is rejected. The code checks for validity in the authorization token sent from the fronted as seen below:

findy_backend > middleware > requireAuth.js > requireAuth

```
1   const jwt = require('jsonwebtoken')
2   const User = require('../models/userModel')
3
4   const requireAuth = async (req, res, next) => {
5     // verify user is authenticated
6     const { authorization } = req.headers
7
8     if (!authorization) {
9       return res.status(401).json({error: 'Authorization token required'})
10    }
11
12    const token = authorization.split(' ')[1]
13
14    try {
15      const { _id } = jwt.verify(token, process.env.SECRET)
16
17      req.user = await User.findOne({ _id }).select('_id')
18      next()
19
20    } catch (error) {
21      console.log(error)
22      res.status(401).json({error: 'Request is not authorized'})
23    }
24  }
25
26  module.exports = requireAuth
```

This interceptor code is then called in the items route file above all routes to make sure all requests to those routes are intercepted and checked for correctness. Hence only authenticated users have access to the database.

4. CONCLUSION:

The database design and implementation phase of the Mobile-Based Application for Archival and Retrieval of Lost and Found Items Using Image Matching was a critical component of the project, ensuring robust and efficient data management. This phase involved careful consideration of the most suitable technologies, resulting in the selection of MongoDB for general data storage and Firebase for image storage.

Throughout this phase, we established a clear and organized file structure, set up the necessary development environments, and meticulously implemented the database schemas and models. Key modules such as user management, item tracking, notifications, transactions, and image matching were designed with modularity and scalability in mind, ensuring that each component could function effectively both independently and as part of the larger system.

The implementation was guided by best practices in database design, ensuring data integrity, consistency, and security. Testing was rigorously conducted to identify and address any potential issues, guaranteeing the reliability and performance of the database system. Security measures were also a priority, with strategies implemented to protect sensitive user data and prevent unauthorized access.

In conclusion, the database design and implementation phase has provided a solid and secure foundation for the application's data management needs. By leveraging the strengths of MongoDB and Firebase, we have ensured efficient data handling and storage, critical for the smooth operation of the application. This phase has set the stage for the successful deployment and operation of the application, demonstrating the importance of thorough planning and execution in database design.

5. REFERENCES:

1) https://docs.expo.dev/guides/using-firebase/ 06/20/2024
2) https://www.mongodb.com/docs/ 06/20/2024
3) https://firebase.google.com/docs?hl=en&authuser=0&_gl=1*lvn02k*_ga*Njc5ODIzNzE5LjE3MTU4MDUwMTM.*_ga_CW55HF8NVT*MTcxODkyMjUzMi4yMC4xLjE3MTg5MjI1MzkuNTMuMC4w /20/2024
4) https://simpledb-java.netlify.app/database-design-and-implementation.pdf Edward Sciore /20/2024