THE UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

CEF 440: INTERNET PROGRAMMING AND MOBILE PROGRAMMING

Group 22

TASK 6: Database Design and Implementation

Name	Matricule	Option
1. Enow Myke-Austine Eta	FE21A183	Software
2. Mokfembam Fabrice Kongnyuy	FE21A240	Software
3. Ndangoh Boris Bobga	FE19A072	Network
4. Ndong Henry Ndang	FE21A248	Software
5. Niba Verine Kajock	FE21A267	Network
6. Takem Jim-Rawlings E.	FE21A309	Software

COURSE INSTRUCTOR:

Dr. NKEMENI VALERY

Contents

1. INTRODUCTION:

In today's world, natural disasters and emergencies pose a constant threat to communities. Effective disaster management relies on timely and accurate information to coordinate response efforts, allocate resources, and ensure public safety. This document outlines the design and implementation of a mobile based disaster management system database.

2. DATABASE DESIGN:

This section outlines the detailed design of the database for our disaster management system

Key Considerations

The database was designed with the following priorities in mind:

- **Performance**: Deliver rapid data retrieval and manipulation for time-sensitive decision making.
- **Scalability**: To accommodate the surge in data volume during emergencies.
- **Offline Functionality**: Enable data submission and access even without an internet connection.
- **Accessibility**: To ensure real-time data access for citizens, and responders.
- **Security**: Safeguard sensitive information and maintain data integrity.
- **Real-time Communication**: Facilitate instant information sharing and collaboration among users.

I. IDENTIFYING DATA ENTITIES AND THEIR ATTRIBUTES

Here we outlined the various entities relevant to our disaster management system, such as users, responders, forums, announcements, incidents etc. and the attributes of each entity so as to capture relevant information of each entity.

Entities and their attributes:

1) **User**

- **ID**: Unique identifier for the user.
- **Name**: User's full name.
- **Email**: User's email address.
- **Password**: User's password.
- **Telephone**: User's phone number.
- **Language**: Preferred language of the user.
- **Photo**: reference to the user's photo.
- **Role**: whether or not the user is a volunteer
- **Locations**: An array of user's preferred locations.
- **Forums**: An array of forums the user is part of.

2) **Responders**

- **id**: Unique identifier for the responder.
- **name**: Responder's full name.
- **email**: Responder's email address.
- **password**: Responder's password.
- **telephone**: Responder's phone number.
- **photo**: URL or reference to the responder's photo.
- **assigned_disasters**: An array of disaster types the responder is assigned to handle.

3) **Incident**

- **id**: Unique identifier for the incident.
- **disaster_type**: Type of disaster (e.g., flood, earthquake).
- **location**: Location of the incident.
- **status**: Current status of the disaster e.g. ongoing, potential or occurred.
- **description**: Detailed description of the incident.
- **media evidence** (optional): Optional media files related to the incident.

- **time**: Timestamp of when the incident was reported.
- **Incident-status**: Pending, reported, etc. (whether or not the responders have approved its occurrence).

4) Help Requests

- **id**: Unique identifier for the help request.
- **location**: Location of the help request.
- **content/description**: Description of the help needed.
- **media evidence** (optional): Optional media files related to the help request.
- **disaster type**: Type of disaster the help request is related to.
- **time**: Timestamp of when the help request was made.

5) Forums

- **ID**: Unique identifier for the forum.
- **Name**: Name of the forum.
- **Description**: Description of the forum.
- **CreatedAt**: Timestamp when the forum was created.
- **UpdatedAt**: Timestamp when the forum was last updated.
- **Author**: Reference to the user who created the forum.
- **Members**: An array of members in the forum, each with:
    - **ID**: Unique identifier for the member.
    - **Member name**: Name of the member.
    - **Role**: Boolean indicating the member's role (e.g., admin, regular member).

6) Announcements

- **id**: Unique identifier for the announcement.
- **sender**: Person or entity sending the announcement.
- **content**: Text content of the announcement.
- **timestamp**: Time when the announcement was made.

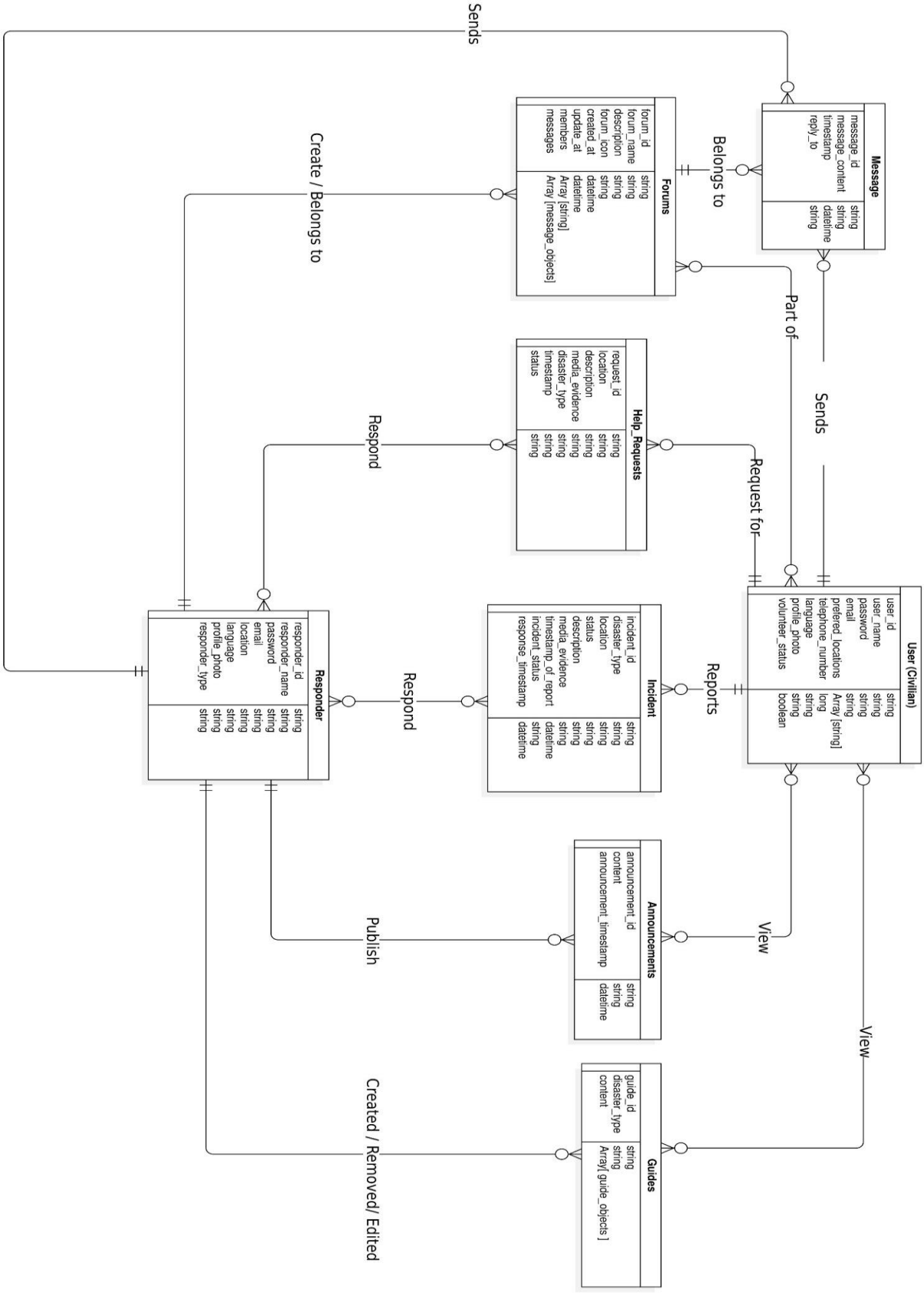
7) Guides

- **id**: Unique identifier for the guide.
- **disaster**: Type of disaster the guide is for.
- **content**:
    - **before**: Instructions or information to follow before the disaster.
        - **image**: Related images.
        - **introductory_text**: First text in the section.
        - **content**: Detailed instructions or information.
    - **during**: Instructions or information to follow during the disaster.
        - **image**: Related images.
        - **introductory_text**: First text in the section.
        - **content**: Detailed instructions or information.
    - **after**: Instructions or information to follow after the disaster.
        - **image**: Related images.
        - **introductory_text**: First text in the section.
        - **content**: Detailed instructions or information.

II. DEFINE RELATIONSHIPS WITH REFERENCES

Since relationships between entities are not explicitly defined using foreign keys, we outlined the relationships between various entities and stored references (document IDs) within documents to link them together by evaluating various criteria.

Here's a visual representation of the system, including the entities, their attributes and their relationships/interactions with each other.


Sends
Belongs to
Part of
Sends
Request for
Create / Belongs to
Respond
Reports
Respond
View
Publish
View
Created / Removed/ Edited
Message
message_id string
message_content string
timestamp datetime
reply_to string
Forums
forum_id string
forum_name string
description string
forum_icon string
created_at datetime
update_at datetime
members Array [string]
messages Array [message_objects]
User (Civilian)
user_id string
user_name string
password string
email string
prefered_locations Array [string]
telephone_number long
language string
profile_photo string
volunteer_status boolean
Help_Requests
request_id string
location string
description string
media_evidence string
disaster_type string
timestamp string
status string
Incident
incident_id string
disaster_type string
location string
status string
description string
media_evidence string
timestamp_of_report datetime
incident_status string
response_timestamp datetime
Responder
responder_id string
responder_name string
password string
email string
location string
language string
profile_photo string
responder_type string
Announcements
announcement_id string
content string
announcement_timestamp datetime
Guides
guide_id string
disaster_type string
content Array[guide_objects]

Since NoSQL databases have no specifically defined approach to producing a schema because the schema structure is solely dependent on the interactions and queries within the project. The multiplicities of the relationship between entities in the project also contribute to schema structure.

The following methods were adopted to create a schema with optimal performance

i. **Optimization structure for reading of frequently read data**

With this approach the NoSQL documents structure was in such a way that frequently read data or queried data were:

- **Denormalized**: With this approach frequently read data were made redundant across documents such that they are easily accessible, saving the overhead time for referencing.
- **Index**: Frequently queried fields were index so as to optimize the reading process.

ii. **Optimization structure for writing of frequently written data**

With this approach the NoSQL documents structure was in such a way that frequently written data were:

- **Normalized:** With this approach frequently written data was made very less redundant such a single change in one document does force multiple updates in other documents containing same data to keep data consistency and integrity.

In cases where the data neither showed a frequent read or frequent write following was adopted

iii. **One-to-one Relationships**

Reading was optimized here through denormalization at either end as this will reduce query complexity, shorten the fetch time and hence increase performance in the sight of the user.

iv. One-to-many Relationships

With the one-to-many relationship denormalization was adopted at the many end such that data in the one end is made redundant at the many end. This will optimize reading.

- **Many-to-many Relationships**

  With many-to-many relationship normalization was chosen as this will in a long run keep performance smooth even as data grows. In other words, too much data will not be stored in one document as the size of data grows.

- **Data size and growth**

  Data with great anticipated growth sizes were either normalized if there was a need for optimizing write or no need for optimizing read. Also, where reading was still to be optimized regardless the size a sub collection embedded in another collection was used.

After following this approach, the following was schema was obtained

1) User (Civilian)

- **user_id**: Unique identifier for the user.
- **user_name**: User's full name.
- **email**: User's email address.
- **password**: User's password.
- **telephone**: User's phone number.
- **language**: Preferred language of the user.
- **profile_photo**: reference to the user's photo.
- **volunteer_status**: whether or not the user is a volunteer
- **prefered_locations**: An array of user’s preferred locations.
- **forums**: An array of forum ids the user is part of referencing forum ids in forum collection.

2) **Responders**

- **responder_id**: Unique identifier for the responder.
- **responder_name**: Responder's full name.
- **email**: Responder's email address.
- **password**: Responder's password.
- **location**: Responder's location.
- **profile_photo**: URL or reference to the responder's photo.
- **responder_type**: responder’s category.

3) Incident

- **id**: Unique identifier for the incident.
- **user**:
    - **user_id**: reporting user’s identifier
    - **user_name**: reporting user’s name
- **disaster_type**: Type of disaster (e.g., flood, earthquake).
- **location**: Location of the incident.
- **status**: Current status of the disaster e.g. ongoing, potential or occurred.
- **description**: Detailed description of the incident.
- **media evidence** (optional): Optional media files related to the incident.
- **timestamp_of_report**: Timestamp of when the incident was reported.
- **incident-status**: Pending, reported, etc. (whether or not the responders have approved its occurrence).
- **responders**: An array of responder id referencing responders responding to a report

4) Help Requests

- **id**: Unique identifier for the help request.

- **location**: Location of the help request.
- **content/description**: Description of the help needed.
- **media evidence** (optional): Optional media files related to the help request.
- **disaster type**: Type of disaster the help request is related to.
- **time**: Timestamp of when the help request was made.
- **user**:
    - **user_id**: identifier of user requesting help
    - **user_name**: name of user requesting help

5) Forums

- **ID**: Unique identifier for the forum.
- **Name**: Name of the forum.
- **Description**: Description of the forum.
- **CreatedAt**: Timestamp when the forum was created.
- **UpdatedAt**: Timestamp when the forum was last updated.
- **Author**: Reference to the user who created the forum.
- **Members**: An array of members in the forum, each with:
    - **ID**: Unique identifier for the member.
    - **Member name**: Name of the member.
    - **Role**: Boolean indicating the member's role (e.g., admin, regular member).

- Sub collection**: messages**

    - **id**: Unique identifier for the message.
    - **content**: Content of the message.
    - **timestamp**: Timestamp of when the message was sent.
    - **sender**:

        - **sender_id**: Id of user or responder who sent message
        - **sender_name**: name of user or responder who sent message
    - **reply_to**: Reference to the message this message is replying to.

6) Announcements

- **id**: Unique identifier for the announcement.
- **content**: Text content of the announcement.
- **timestamp**: Time when the announcement was made.
- **responder id**: Reference to the responder who sent out the announcement.

7) Guides

- **id**: Unique identifier for the guide.
- **disaster**: Type of disaster the guide is for.
- **guide_content**:
    - **before**: Instructions or information to follow before the disaster.
        - **image**: Related images.
        - **introductory_text**: First text in the section.
        - **content**: Detailed instructions or information.
    - **during**: Instructions or information to follow during the disaster.
        - **image**: Related images.
        - **introductory_text**: First text in the section.
        - **content**: Detailed instructions or information.
    - **after**: Instructions or information to follow after the disaster.

- **image**: Related images.
- **introductory_text**: First text in the section.
- **content**: Detailed instructions or information.

- **creator_id**: References the user or responder that uploaded guide

3. DATABASE IMPLEMENTATION:

I. SELECT DATABASE MANAGEMENT SYSTEM (DBMS) AND WHY IT WAS CHOSEN

Why Firebase?

i. Authentication:

- Integration: Firebase provides a ready-to-use authentication system with built-in integrations for popular social media logins (such as Google, Facebook, Twitter) and email/password authentication. This simplifies the implementation of user authentication compared to building a custom solution with MongoDB.

ii. Fast Development:

- Serverless Architecture: Firebase offers a serverless architecture, eliminating the need for server setup and management. This allows developers to focus on writing frontend code and accelerates development speed by reducing infrastructure-related tasks.

iii. **Scalability**:

- Managed Infrastructure: Firebase is a fully managed platform that automatically scales your application as needed. It handles infrastructure provisioning, scaling, and load balancing behind the scenes, allowing your application to handle high traffic and scale seamlessly.

iv. **Realtime Capabilities**:

- Realtime Database: Firebase's Realtime Database offers real-time data synchronization across all clients connected to the database. This makes it easy to build applications that

require instant data updates, such as chat apps or collaborative tools. Changes to the database are propagated to all clients in real-time, ensuring that all users have the most up-to-date information.

- Cloud Firestore: Firestore also provides real-time data synchronization with more advanced querying capabilities and hierarchical data structures. Firestore's real-time listeners ensure that any changes to the database are immediately reflected on all clients, facilitating real-time collaboration and dynamic data updates.

v. Offline Functionality:

- Realtime Database Offline Support: Firebase's Realtime Database provides built-in offline support. It caches data locally on the client device, enabling your application to continue functioning offline and automatically synchronizing the data when the device goes back online. This feature simplifies building offline-capable applications without the need for additional configuration.
- Cloud Firestore Offline Persistence: Firestore, another database option offered by Firebase, also provides offline support with a feature called Firestore Persistence. This allows you to store data locally using SQLite, enabling offline functionality and automatic synchronization when the device goes online. Firestore Persistence ensures that the application remains responsive and functional even when the network connection is unreliable or unavailable.

II. FIREBASE PROJECT SETUP AND IMPLEMENTATION

To set up the Firebase project for our application, our team followed a structured approach on the Firebase console, ensuring comprehensive configuration and implementation. Here's a detailed account of our process:

1. **Setting Up Firebase Project**

To begin, we navigated to the Firebase official website and accessed the Firebase console to initiate our project setup:

2. **Navigate to Firebase Console**:

a. Visit the Firebase Console.
b. Click on "Add project" and provide a project name as per our application requirements.
c. Followed the prompts to configure and create the project.

3. **Configure Firebase SDK**:
   a. Upon project creation, Firebase provided a configuration file (firebaseConfig) containing essential credentials and configuration settings necessary for connecting our application to the Firebase project.
   b. This file includes details such as API keys, authentication domain, project ID, and other identifiers crucial for establishing a secure and functional connection.

Using Firebase Configuration File

The provided firebaseConfig file enables seamless integration of our application with the Firebase project:

```
import { initializeApp } from "firebase/app";
import { getAnalytics } from "firebase/analytics";
import { getAuth, GoogleAuthProvider } from 'firebase/auth'
import {getFirestore} from 'firebase/firestore'


const firebaseConfig = {
  apiKey: "AIzaSyA8Upx8sXo5UV9_YfwRLFrwKgAyXUlDsy0",
  authDomain: "relief-radar.firebaseapp.com",
  projectId: "relief-radar",
  storageBucket: "relief-radar.appspot.com",
  messagingSenderId: "345836559744",
  appId: "1:345836559744:web:f37c8cb83d03e32d71f565",
  measurementId: "G-KM76HFVDN9"
};

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

export const auth = getAuth(app);
export const googleProvider = new GoogleAuthProvider();
export const db = getFirestore(app);
```

Configuring Authentication Methods

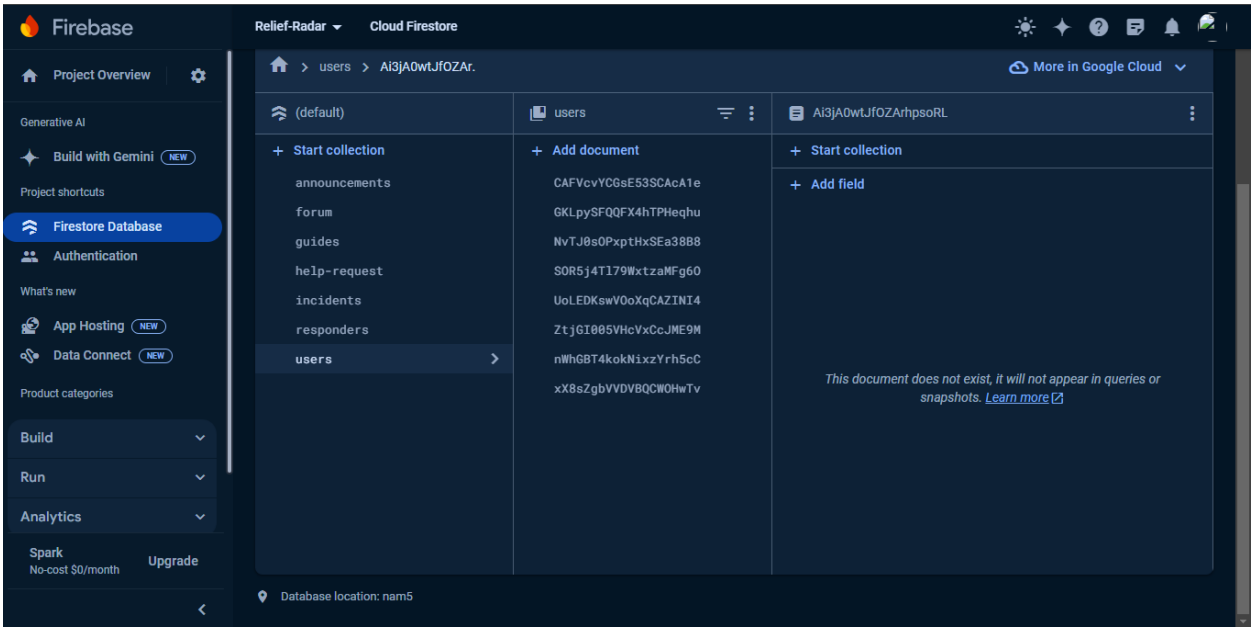
With the Firebase project set up, we proceeded to configure authentication methods suitable for our application's user management needs:

1. **Enable Authentication Providers**:
    - In the Firebase console, under the "Authentication" section, we enabled methods such as Email/Password and Google Sign-In.
    - Configured each authentication provider with necessary credentials and settings to ensure secure and reliable user authentication.

III. CREATE COLLECTIONS

After successful setup, we proceeded to create collections within our Firebase Firestore database. Each collection serves a distinct purpose in storing and managing data for our application:

1. **Users Collection:** stores user profiles and related information.
2. **Responders Collection:** manages information about emergency responders.
3. **Incidents Collection** :records incident reports submitted by users.
4. **Help Requests Collection: m**a nages requests for assistance related to disasters.
5. **Forums Collection** :facilitates communication and collaboration among users.
6. **Announcements Collection**: stores announcements related to disaster management.
7. **Guides Collection:** provides guidelines and instructions for disaster preparedness and response.



Each collection is crucial for managing specific aspects of our disaster management application, ensuring structured data storage, efficient querying, and real-time synchronization. This approach facilitates seamless integration and enhances user experience by enabling robust data management and interaction capabilities.

IV. IMPLEMENT CRUD OPERATIONS

After creating the collections, we proceeded to add documents to represent records in traditional SQL databases and implemented CRUD operations for managing data:

1. **Adding Documents**
    - **Users Collection**: Added documents representing user profiles with their respective fields.
    - **Incidents Collection**: Added documents for incident reports submitted by users.
    - **Guides Collection**: Added documents providing guidelines and instructions for disaster management.
2. **CRUD Operations**

- **Create**: Implemented functions to add new records to the Users, Incidents, and Guides collections;

```
const createUser = async (collectionName: string, userInfo: User): Promise<DocumentReference> ⇒
  try {
    const result = await addDoc(collection(db, collectionName), {
      ...userInfo,
      timestamp: serverTimestamp(),
    });
    return result;
  } catch (error) {
    console.error(error);
    throw error;
  }
};
```

```
const createIncident = async (collectionName: string, incidentInfo: Incident): Promise<DocumentRe
  try {
    const result = await addDoc(collection(db, collectionName), {
      ...incidentInfo,
      timestamp: serverTimestamp(),
    });
    return result;
  } catch (error) {
    console.error(error);
    throw error;
  }
};
```

```
const createGuide = async (collectionName: string, GuideInfo: DisasterGuide): Promise<DocumentRefe
    try {
      const result = await addDoc(collection(db, collectionName), {
        ...GuideInfo,
        timestamp: serverTimestamp()
      })
        return result
    } catch (error) {
        console.error(error)
        throw error
    }
}
```

- **Read**: Developed methods to retrieve data from Firebase Firestore using getDocs() queries for fetching user profiles, incident details, and disaster management guides.

```
// Query to get all guides by disaster type
const qDisaster = query(collection(db, 'guides'), where('disaster', '==', 'earthquake'));

const getGuidesByDisaster = async () => {
  try {
    const { docs } = await getDocs(qDisaster);
    const result = docs.map(doc => ({
      id: doc.id,
      ...doc.data()
    }));
    return result;
  } catch (error) {
    console.error(error);
  }
}
```

```
// Query to get all incidents by status
const qStatus = query(collection(db, 'incidents'), where('status', '==', 'occurred'));

const getIncidentsByStatus = async () => {
  try {
    const { docs } = await getDocs(qStatus);
    const result = docs.map(doc => ({
      id: doc.id,
      ...doc.data()
    }));
    return result;
  } catch (error) {
    console.error(error);
  }
}
```

```
//get all users that are volunteers
const q = query(collection(db,'users'),where('role', '==','volunteer'))

const getAllUserVolunteers = async () => {
  try {
    const { docs } = await getDocs(q)
    const result = docs.map(doc => (
      {
        id: doc.id,
        ...doc.data()
      }
    ))
    return result;
  } catch (error) {
    console.error(error);
  }
}
```

- **Update**: Enabled functionality to update existing records in the Users, Incidents, and Guides collections using updateDoc() operations.

```
// update location
const docRef = doc(db,'users','UoLEDKswVOoXqCAZINI4')

const updateUserLocation = async () => {
  try {
    const updatedDoc = await updateDoc(docRef, {
      locations:['Cameroon','Nigeria','Tchad']
    });

    return updatedDoc;
  } catch (error) {
    console.error(error);
    throw error;
  }
}
```

```
// Update media evidence for a specific incident
const incidentDocRef = doc(db, 'incidents', 'incident-789');

const updateIncidentMediaEvidence = async () => {
  try {
    const updatedDoc = await updateDoc(incidentDocRef, {
      mediaEvidence: ['path/to/newimage1.jpg', 'path/to/newimage2.jpg']
    });
    return updatedDoc;
  } catch (error) {
    console.error(error);
    throw error;
  }
}
```

```
// Update the images during a specific disaster guide
const guideDocRef = doc(db, 'guides', 'guide-001');

const updateGuideImages = async () => {
  try {
    const updatedDoc = await updateDoc(guideDocRef, {
      'content.during.image': ['path/to/newimage1.jpg', 'path/to/newimage2.jpg']
    });
    return updatedDoc;
  } catch (error) {
    console.error(error);
    throw error;
  }
}
```

- **Delete**: Implemented mechanisms to delete records from the database using deleteDoc().

```
// Delete a specific incident
const incidentDocRef2 = doc(db, 'incidents', 'incident-790');

const deleteIncident = async () => {
  try {
    const deletedDoc = await deleteDoc(incidentDocRef2);
    return deletedDoc;
  } catch (error) {
    console.error(error);
    throw error;
  }
}
```

```
// Delete a specific disaster guide
const guideDocRef2 = doc(db, 'guides', 'guide-002');

const deleteGuide = async () => {
  try {
    const deletedDoc = await deleteDoc(guideDocRef2);
    return deletedDoc;
  } catch (error) {
    console.error(error);
    throw error;
  }
}
```

V. DATA VALIDATION

In our Firebase project, ensuring data integrity and consistency was a top priority. Firebase offers robust data validation capabilities that we leveraged to enforce strict data type handling and validation rules directly within our backend. This approach provided a reliable way to validate user input and maintain high-quality data in our Firebase Firestore database.

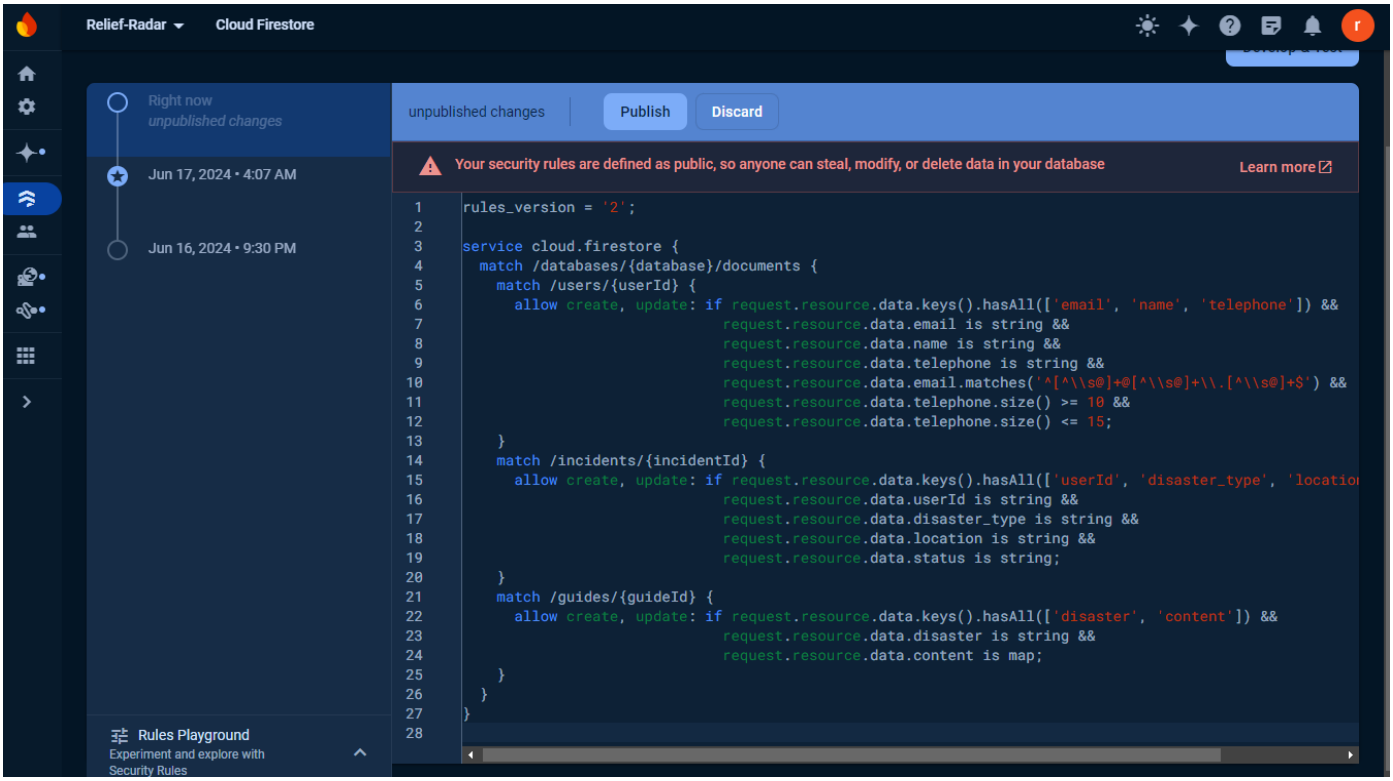
Implementation Approach: We used Firebase's built-in validation features to define and enforce data types for each field in our collections. By specifying data types and validation rules at the database level, we ensured that any data written to our Firestore database adhered to the

expected formats. This prevented incorrect data types from being stored, thereby maintaining the integrity of our data.

Firebase Security Rules: Firebase Security Rules played a crucial role in our data validation strategy. These rules allowed us to define granular validation logic directly within the Firestore database. Here's how we implemented them:

- **Data Type Validation**: For each field in our collections, we specified the expected data type (e.g., string, number, boolean). This ensured that only data of the correct type could be written to the database.
- **Required Fields**: We enforced the presence of required fields by specifying that certain fields must not be null or undefined.
- **Pattern Matching**: For fields like email addresses and phone numbers, we used regular expressions within our security rules to validate the format of the input data.

Example Validation Rules: Below are examples of the Firebase Security Rules we used to enforce data validation:



Integration with CRUD Operations:

- **Create**: When creating new records, Firebase Security Rules ensured that all required fields were present and correctly typed. Invalid data types or missing fields resulted in errors, preventing the creation of invalid records.
- **Read**: Although reading data typically does not involve validation, our rules ensured that sensitive fields were protected and only accessible to authorized users.
- **Update**: During updates, the validation rules were re-applied to ensure that any modifications adhered to the same strict data type requirements as during creation.
- **Delete**: While deletion operations focused more on access control, ensuring the correct records were targeted for removal helped maintain data consistency.

Challenges and Solutions: One challenge was defining comprehensive validation rules that covered all edge cases without being overly restrictive. We addressed this by iteratively refining our rules and testing them against various input scenarios to balance flexibility with strict validation.

By using Firebase's built-in validation features and security rules, we enforced robust data validation directly at the backend. This ensured data integrity and consistency across our collections, significantly reducing the risk of data corruption and improving the overall reliability of our application.

4. SECURITY:

1. Data Protection and Privacy

1.1 Compliance with Regulations

Firebase adheres to privacy regulations such as the EU General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA). Customers act as data controllers

(GDPR) or businesses (CCPA), while Google operates as a data processor (GDPR) or service provider (CCPA). This means that data is under the customer's control. Customers are responsible for obligations like fulfilling an individual's rights with respect to their personal data or information.

1.2 Certifications

Firebase is certified under major privacy and security standards. Its services have successfully completed the ISO 27001, ISO 27017, ISO 27018 and SOC 1, SOC 2, and SOC 3 certification process and evaluation process.

Firebase services are certified under the following standards:

- **ISO 27001:** Information Security Management System (ISMS)
- **SOC 1, SOC 2, SOC 3:** Service Organization Control reports
- **ISO 27017:** Cloud Security Controls
- **ISO 27018:** Personal Data Protection in the Cloud

2. Encryption

2.1 Data in Transit

Firebase encrypts data in transit using HTTPS (SSL/TLS). This ensures that communication between clients and Firebase servers is secure.

2.2 Data at Rest

Several Firebase services also encrypt data at rest. This means that even when data is stored in databases or storage buckets, it remains encrypted.

3. Access Control

3.1 Authorized Access

- Firebase restricts access to authorized employees with valid business purposes.
- Customers retain control over their data and fulfill individual rights regarding personal information.