Database Design And Implementation

PRESENTED BY

GROUP 22

Outline

1. Introduction
2. Database Design
3. Database Implementation
4. Database Security

1. Introduction

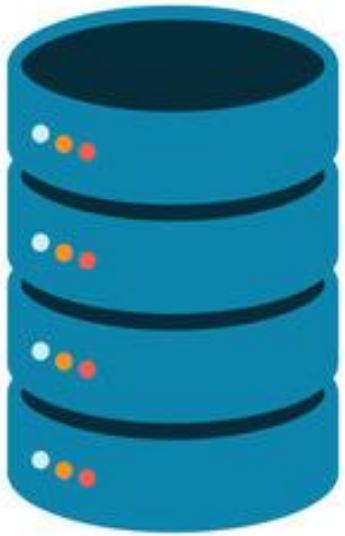
Effective disaster management relies on timely, accurate information for coordinating response efforts, allocating resources, and ensuring public safety.

This presentation outlines the design and implementation of a mobile-based disaster management system database, crucial for enhancing response efficiency and crisis management.

2. Database Design

Key Considerations

- Performance
- Scalability
- Offline Functionality
- Accessibility
- Security
- Real-time Communication



I. Entities and their Attributes

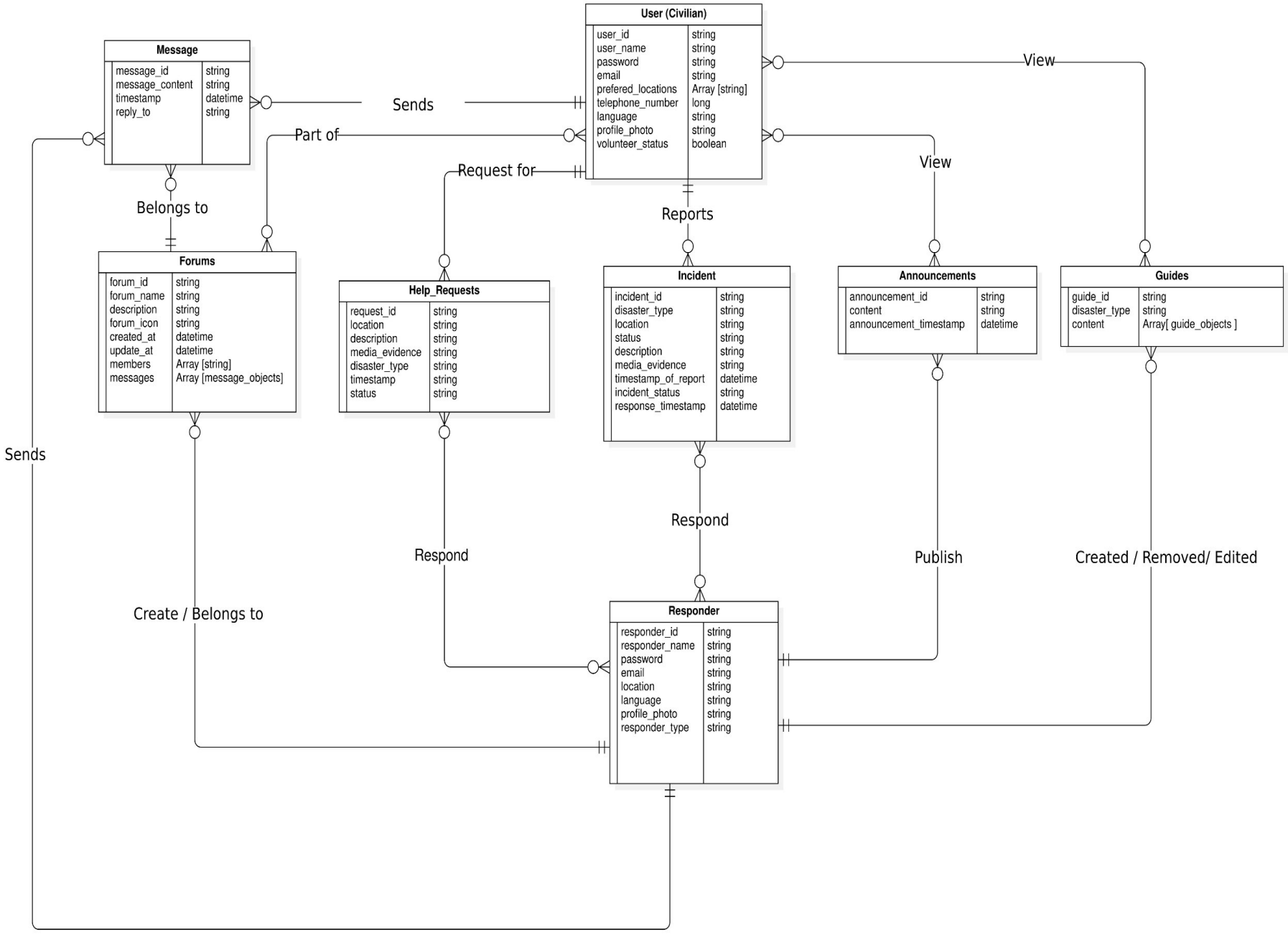
1. User
2. Responders
3. Incidents
4. Help_Requests
5. Forums
6. Announcements
7. Guides

e.g. User

- ID
- Name
- Email:
- Password
- Telephone
- Language
- Photo
- Role
- Locations
- Forums

II. Define Relationships and References

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


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
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]
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
Announcements
announcement_id string
content string
announcement_timestamp datetime
Guides
guide_id string
disaster_type string
content Array[guide_objects]
Responder
responder_id string
responder_name string
password string
email string
location string
language string
profile_photo string
responder_type string
Sends
Part of
Request for
View
View
Belongs to
Reports
Sends
Respond
Respond
Publish
Created / Removed/ Edited
Create / Belongs to

II. Define Relationships and References

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

i. Optimization structure for reading of frequently read data
   - Denormalized
   - Index

ii. Optimization structure for writing of frequently written data
   - Normalized

II. Define Relationships and References

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

iii. One-to-one Relationships
iv. One-to-many Relationships
    - Many-to-many Relationships
    - Data size and growth

Example: User and Forums collection

1) User (Civilian)

- user_id
- user_name
- Email
- Password
- Telephone
- Language
- profile_photo
- volunteer_status
- prefered_locations
- forums

5) Forums

- ID
- Name
- Description
- CreatedAt
- UpdatedAt
- Author
- Members
  - ID
  - Member name
  - Role
- messages
  - Id
  - content:
  - Timestamp
  - sender
    - sender_id:
    - sender_name
  - reply_to

3. Database Implementation

I. Select DBMS and why it was chosen
II. Firebase project setup and Implementation
III. Create collections
IV. Implement CRUD operations
V. Data Validation

I. Select DBMS and why it was chosen

Why Firebase?

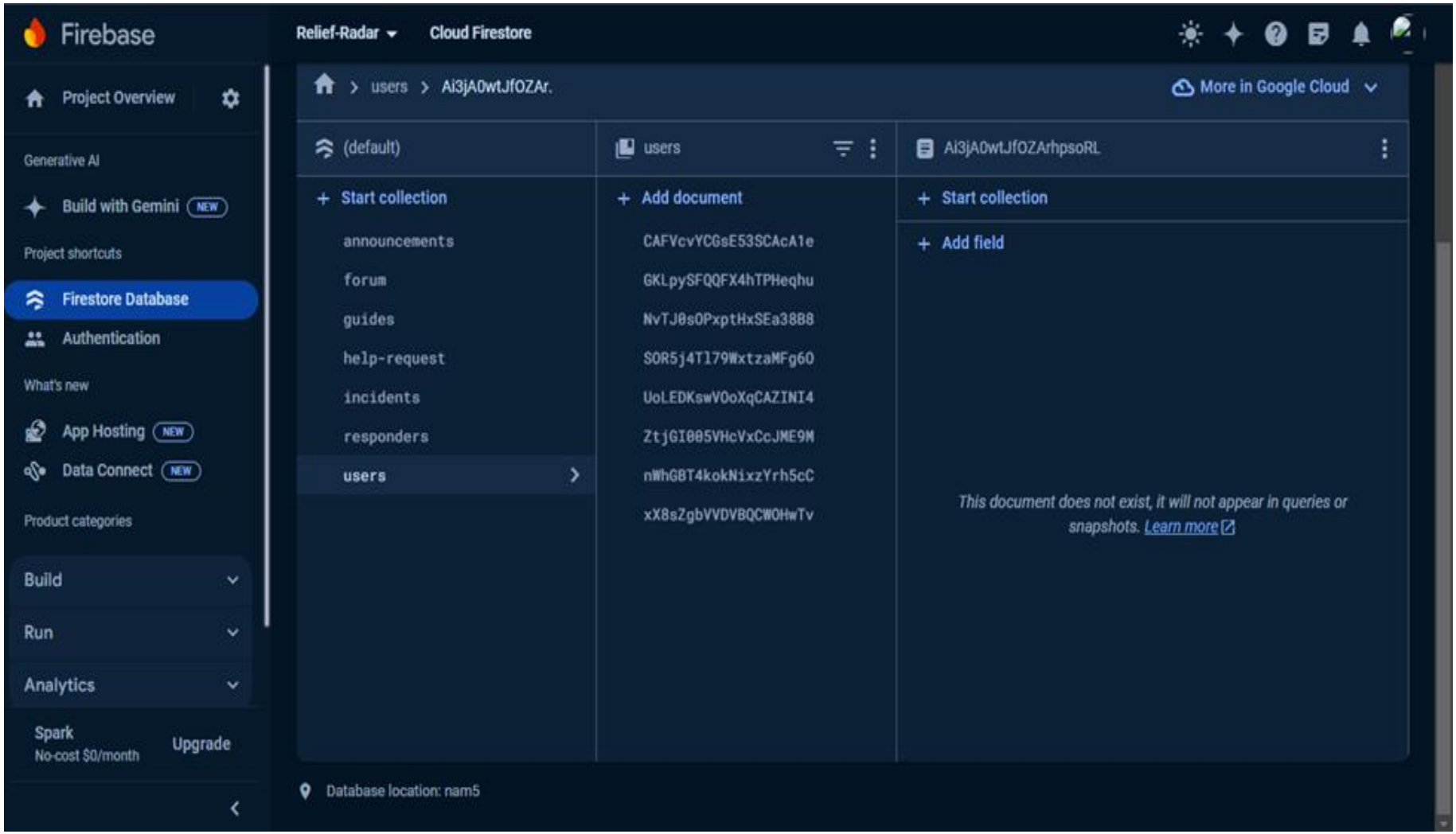
i. Authentication
ii. Fast Development
iii. Scalability
iv. Realtime Capabilities
v. Offline Functionality

II. Firebase project setup and Implementation

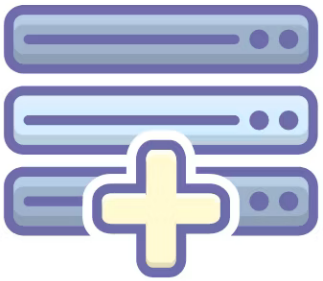
1. Using Firebase Configuration File
2. Enable Authentication Providers

```
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);
```

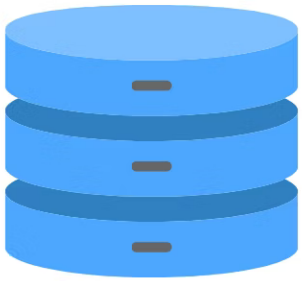
III. Create collections



IV. Implement CRUD operations



CREATE



READ

UPDATE

DELETE

IV. Implement CRUD operations

1. Adding documents
   - Users collection
   - Incidents collection
   - Guides collection
2. CRUD operations

Create

```
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<DocumentRef
  try {
    const result = await addDoc(collection(db, collectionName), {
      ...incidentInfo,
      timestamp: serverTimestamp(),
    });
    return result;
  } catch (error) {
    console.error(error);
    throw error;
  }
};
```

Read

```
// 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

```
// 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 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

```
// 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

V. Data Validation

- Data Type Validation
- Required Fields
- Pattern Matching

Integration with CRUD Operations

- Create
- Read
- Update
- Delete

4. Database Security

1. Data Protection and Privacy
2. Encryption
3. Access Control

THE END!!!