REPUBLIC OF CAMEROON

PEACE-WORK-FATHERLAND

MINISTER OF HIGHER EDUCATION

UNIVERSITY OF BUEA

REPUBLIQUE DU CAMEROUN

PAIX-TRAVAIL-PATRIE

MINISTERE DE L'ENSEIGNEMENT SUPERIEURE

UNIVERSITE DE BUEA

CEF 440: Internet Program and Mobile Programming

TASK 6

Group 18

Course Instructor: ***Dr. NKEMENI Valery***

June 2024

Table of content

Database Design and Implementation

Introduction

Databases are foundational to most modern applications, enabling efficient storage, retrieval, and management of data. A well-designed database not only ensures data integrity and performance but also simplifies application development and maintenance.

A database is an organized collection of structured information, typically stored electronically in a computer system. A database management system (DBMS) is software that interacts with users, applications, and the database itself to capture and analyze data.

For our disaster management mobile application, we used:

- NoSQL database is suitable for use in disaster management mobile applications due to several of its characteristics that aligns well with the unique requirements and challenges of our application.

  These characteristics include;

1. Flexibility in Handling Diverse Data Types:

Disaster management scenarios often involve a wide variety of data: text, images, videos, sensor readings, and geospatial data. NoSQL databases, especially document stores like MongoDB, offer a flexible schema that allows the storage of diverse data types without predefined structures.

Adjusting to changing data requirements: As new types of data or unexpected data formats arise during a disaster, NoSQL databases can adapt without requiring a complex and time-consuming schema migration.

2. High Availability and Reliability

Essential for continuous operation during crises: Disaster management systems must remain operational even in adverse conditions. NoSQL databases often include built-in replication and automatic failover features, ensuring that the system remains available even if some nodes fail.

Low-latency access to data: In disaster situations, timely access to information is crucial. NoSQL databases are designed to provide quick, efficient access to data across distributed systems, reducing the latency that could otherwise hinder response efforts.

3. Geospatial Capabilities

Critical for mapping and location-based services: Many NoSQL databases, like MongoDB, have robust support for geospatial data. This capability is vital for applications that need to track disaster impacts, coordinate resources, and provide location-based alerts to users.

4. Handling Unstructured and Semi-Structured Data

Integration of various data sources: During a disaster, information can come from various unstructured sources such as social media, IoT devices, and user reports. NoSQL databases are adept at storing and processing unstructured or semi-structured data, making them ideal for integrating diverse data streams.

Facilitating data analytics and insights: By efficiently managing different data formats, NoSQL databases help in performing real-time analytics and generating insights that are crucial for decision-making during emergencies.

Database Design process

Database design is a structured process that involves several stages, from understanding the requirements to creating a functional database schema. A well-designed database ensures data integrity, supports efficient query execution, and can scale as the application grows.

Data element

User Information

- **User ID**: Unique identifier for each user.
- **Name**: Full name of the user.
- **Contact Information**: Phone number, email address, and emergency contacts.
- **Role/Type**: Classification of the user (e.g., citizen, volunteer, emergency responder).
- **Location**: Current location or residence of the user (GPS coordinates or address).

2. Disaster Events

- **Event ID**: Unique identifier for each disaster event.
- **Type**: Type of disaster (e.g., earthquake, flood, hurricane, wildfire).
- **Location**: Affected area's geographic details (coordinates, region names).
- **Start Time**: When the disaster began.
- **Status**: Current status (e.g., active, resolved, ongoing).
- **Description**: Detailed description of the disaster event.

3. Alerts and Notifications

- **Alert ID**: Unique identifier for each alert.
- **Type**: Type of alert (e.g., evacuation, shelter-in-place, weather warning).
- **Message**: Content of the alert message.
- **Region**: Specific region or area the alert applies to.
- **Recipients**: List of users or groups receiving the alert.

4. Resource Information

- **Resource ID**: Unique identifier for each resource.
- **Type**: Type of resource (e.g., medical supplies, food, water, personnel).
- **Quantity**: Amount of the resource available.
- **Location**: Where the resource is stored or available.
- **Provider**: Organization or individual providing the resource.

5. Shelters and Safe Zones

- **Shelter ID**: Unique identifier for each shelter.
- **Name**: Name of the shelter or safe zone.
- **Location**: Geographic location (address, GPS coordinates).
- **Facilities Available**: List of facilities and services available (e.g., food, medical care, sleeping arrangements).

6. Emergency Contacts

- **Contact ID**: Unique identifier for each emergency contact.
- **Name**: Name of the contact person or organization.
- **Role/Function**: Function in the context of disaster management (e.g., police, fire department, medical emergency).
- **Contact Information**: Phone numbers, email addresses, and other contact details.
- **Location**: Geographic location of the contact or their office.

8. Geospatial Data

- **Geospatial ID**: Unique identifier for each geospatial data element.
- **Type**: Type of geospatial data (e.g., maps, GPS coordinates, regions).
- **Coordinates**: Latitude and longitude of the geospatial point or area.
- **Boundary Data**: Details about the boundaries of an affected area or safe zone.
- **Layer Information**: Additional map layers (e.g., flood zones, evacuation routes).

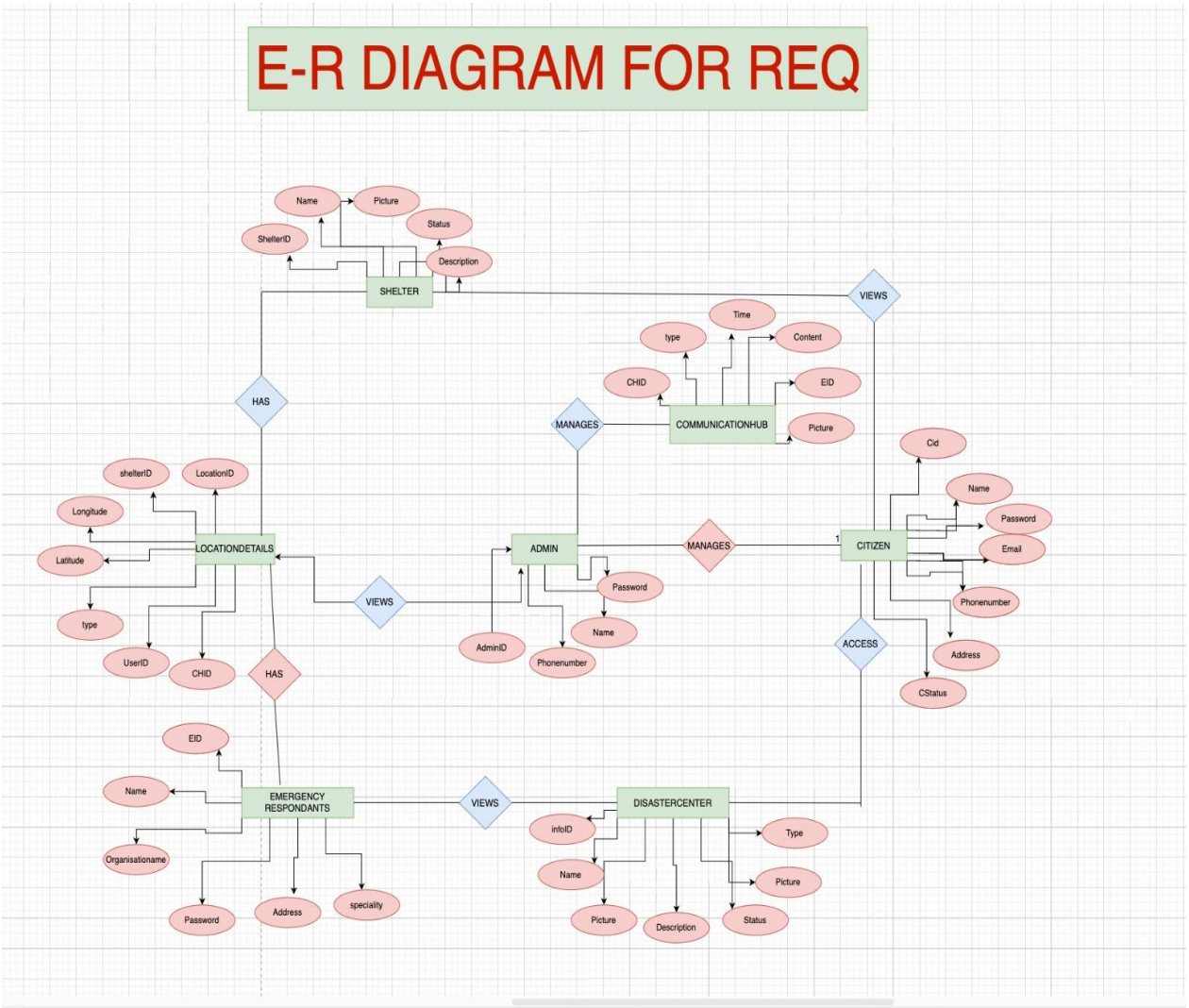
12. Communication and Collaboration

- **Message ID**: Unique identifier for each message or communication.
- **Sender ID**: Identifier for the sender of the message.
- **Receiver ID**: Identifier for the receiver or group of receivers.

- **Message Content**: Text or multimedia content of the message.
- **Timestamp**: When the message was sent.
- **Status**: Delivery status of the message (e.g., sent, delivered, read).

ER Diagram

An ER diagram, or Entity-Relationship diagram, is a visual representation of the data model that shows the relationships between entities (objects or concepts) in a system. It is widely used in database design and serves as a blueprint for designing databases in relational database management systems (RDBMS). ER diagrams are essential for visualizing and organizing the structure of data within an application or system.



Key Elements:

1. **Entities (Green Rectangles)**:
    - These represent the main objects or tables in the database.
2. **Attributes (Red Ovals)**:
    - These represent the data fields or columns within each entity.
3. **Relationships (Diamonds)**:
    - These describe how entities are related to each other.
4. **Foreign Keys**:
    - Attributes used to link entities together in a relational manner.

Detailed Breakdown:

1. Entities and Attributes:

SHELTER:

- ShelterID: Primary Key, uniquely identifies each shelter.
- Name: The name of the shelter.
- Picture: An image associated with the shelter.
- Status: The current status of the shelter (e.g., open, closed).
- Description: Additional details about the shelter.

LOCATIONDETAILS:

- LocationID: Primary Key, uniquely identifies each location.
- Longitude & Latitude: Geographic coordinates.
- Type: Type of location (could relate to emergency services, shelters, etc.).
- UserID: Reference to a user associated with the location.
- ShelterID: Foreign Key referencing SHELTER.
- CHID: Foreign Key referencing COMMUNICATIONHUB.

COMMUNICATIONHUB:

- CHID: Primary Key, uniquely identifies each communication hub.
- Type: Type of communication (e.g., message, alert).
- Time: The time the communication was recorded.

- Content: The content or message being communicated.
- Picture: An image associated with the communication.
- EID: Foreign Key referencing EMERGENCYRESPONDANTS.

ADMIN:

- AdminID: Primary Key, uniquely identifies each admin.
- Name: Admin's name.
- Password: Admin's password for authentication.
- Phonenumber: Admin's contact number.

CITIZEN:

- Cid: Primary Key, uniquely identifies each citizen.
- Name: Citizen's name.
- Password: Citizen's password for authentication.
- Email: Citizen's email address.
- Phonenumber: Citizen's contact number.
- Address: Citizen's home address.
- CStatus: Status of the citizen (possibly related to safety or response status).

DISASTERCENTER:

- Type: Type of disaster center (e.g., emergency response, medical).
- Picture: Image related to the disaster center.
- Description: Description of the center.
- Status: Current operational status of the disaster center.

EMERGENCYRESPONDANTS:

- EID: Primary Key, uniquely identifies each respondent.
- Name: Name of the respondent.
- Age: Age of the respondent.
- Phonenumber: Contact number.
- Address: Home or operational address.

- Specialty: The area of expertise or role in emergencies.

2. Relationships:

HAS:

- **Between SHELTER and LOCATIONDETAILS**: Indicates that a shelter can have multiple location details.
- **Between EMERGENCY RESPONDENTS and LOCATIONDETAILS**: Suggests that emergency respondents can be associated with multiple locations.

MANAGES:

- **ADMIN and COMMUNICATIONHUB**: Shows that an admin manages the communication hub.

VIEWS:

- **LOCATIONDETAILS and CITIZEN**: Citizens can view location details.
- **CITIZEN and DISASTERCENTER**: Citizens can view disaster center information.
- **ADMIN and LOCATIONDETAILS**: Admins can view location details.

ACCESS:

- **CITIZEN and DISASTERCENTER**: Citizens have access to disaster center data.

Foreign Key Constraints:

- ShelterID in LOCATIONDETAILS refers to ShelterID in SHELTER.
- CHID in LOCATIONDETAILS refers to CHID in COMMUNICATIONHUB.
- EID in COMMUNICATIONHUB refers to EID in EMERGENCYRESPONDANTS.

Implementing a database

Database implementation refers to the process of constructing, deploying, and making operational a database management system (DBMS) based on the database design. It involves translating the logical database design into a physical database schema that can be executed on a specific DBMS platform.

Libraries and tools used

Express: A framework that uses Node.Js to create a running server application. It was used due to its simplicity in creating APIs (Application Programming Interface) and ease of communication with the database. Also, it interacts with all other backend directories comprising of models, routes and controllers.

Nodemon: It monitors changes in the source code files and automatically restart the server.

Mongoose: It is an Object Data Modeling (ODM) powerful Node.js library that simplifies the interaction with MongoDB.

Database Creation

Creating a database in a NoSQL environment typically involves defining a schema or structure for our data collections. Unlike traditional SQL databases where tables with predefined schemas are used, NoSQL databases are schema-less or schema-flexible, allowing more flexibility in storing and querying data.

The NoSql database was created on MongoDB and named Resq-app as shown on the figure below;

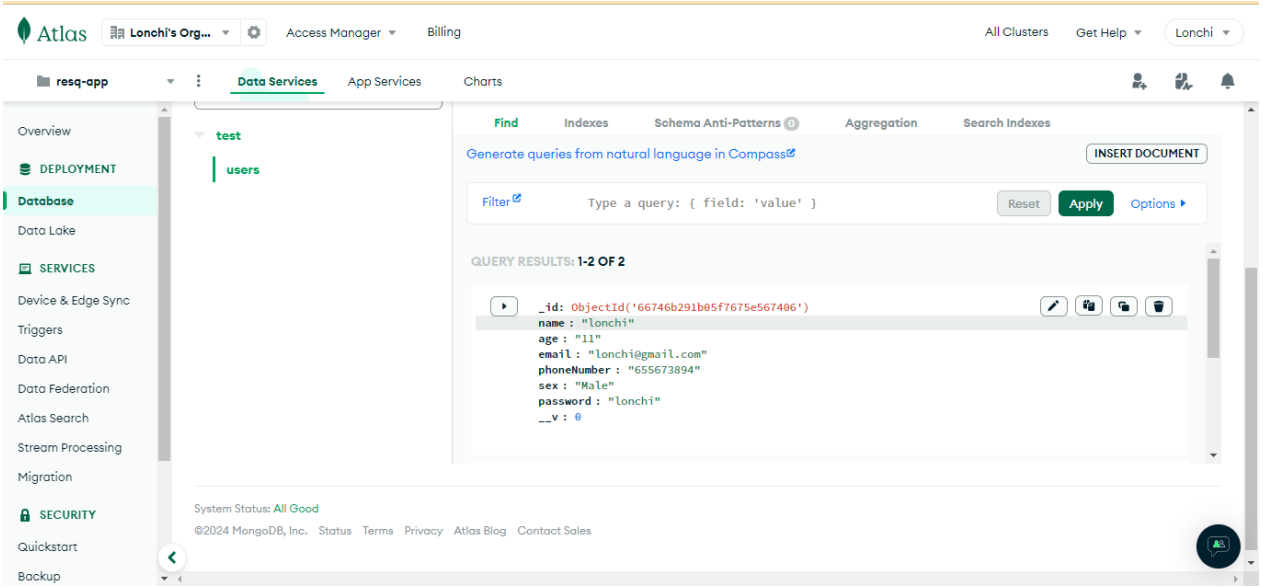


Figure 1 NoSQL database

Connecting our database to the backend

The connection of MongoDB database using Mongoose, a popular Object Data Modeling (ODM) library for MongoDB and Node.js providing a higher-level API for interacting with MongoDB.

The figure below shows how the connection was done.

```
const express = require("express")
const mongoose = require("mongoose")
const userRouter = require("./Router/userRoute")

const app = express();

app.use(express.json())


app.use("/api/user", userRouter)

mongoose.connect("mongodb+srv://resq:resq123@resq-app.htlsrfh.mongodb.net/?retryWrites=true&w=ma
.then(() => {
    app.listen(4000, () => {
        console.log(" Connected to DB & listening on port 4000")
    })
})
.catch(error => {
    console.log('error connecting to database')
})
```

Here is the successful connection of the server to the database

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
 Connected to DB & listening on port 4000
```

Creating models

A model typically refers to the logical structure or representation of data within a database system.

Creating models in MongoDB using Mongoose involves defining schemas and then creating models based on those schemas.

- User Model:

```
const mongoose = require("mongoose")

const Schema = mongoose.Schema

const userSchema = new Schema({
    name:{
        type:String,
        required:true
    },
    age:{
        type:String,
        required:true
    },
    email:{
        type:String,
        required:true,
        unique:true
    },
    phoneNumber:{
        type:String,
        required:true,
        unique:true
    },
```

```
const userSchema = new Schema({
    },
    email:{
        type:String,
        required:true,
        unique:true
    },
    phoneNumber:{
        type:String,
        required:true,
        unique:true
    },
    sex:{
        type:String,
        required:true
    },
    password:{
        type:String,
        required:true
    }
})
module.exports = mongoose.model('user', userSchema)
```

Creating controllers

Controllers are functions which govern the logic of the application. In this case, the controllers were created for each model in order to populate data into the database.

- User Controller

Backend > controller > userController.js > <unknown>

```
const User = require('../Model/userModel')

const registerUser = async(req, res) => {
    const { name,age, email,phoneNumber,sex, password } = req.body;

    try {
       const newUser = new User({
            name,
            age,
            email,
            phoneNumber,
            sex,
            password,
        });
        await newUser.save();
        res.status(201).json({ message: 'User created successfully' });
    } catch (error) {
        res.status(500).json({ error: 'Internal server error'});
    }
}
module.exports = {registerUser}
```

Creating Routers

Routers help in defining how an application responds to client requests to various endpoints, typically corresponding to different database operations. A router is a component of a web application that determines how to handle incoming HTTP requests (like GET, POST, PUT, DELETE) and direct them to the appropriate controllers.

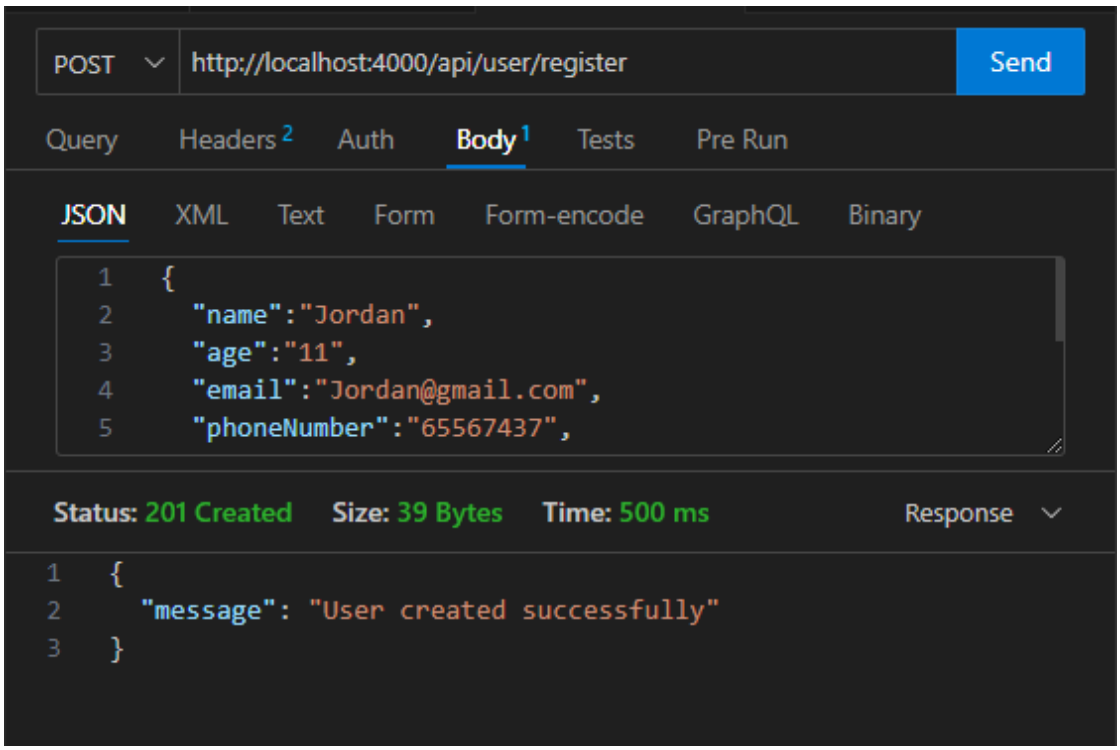
- User Route

Backend > Router > userRoute.js > ...

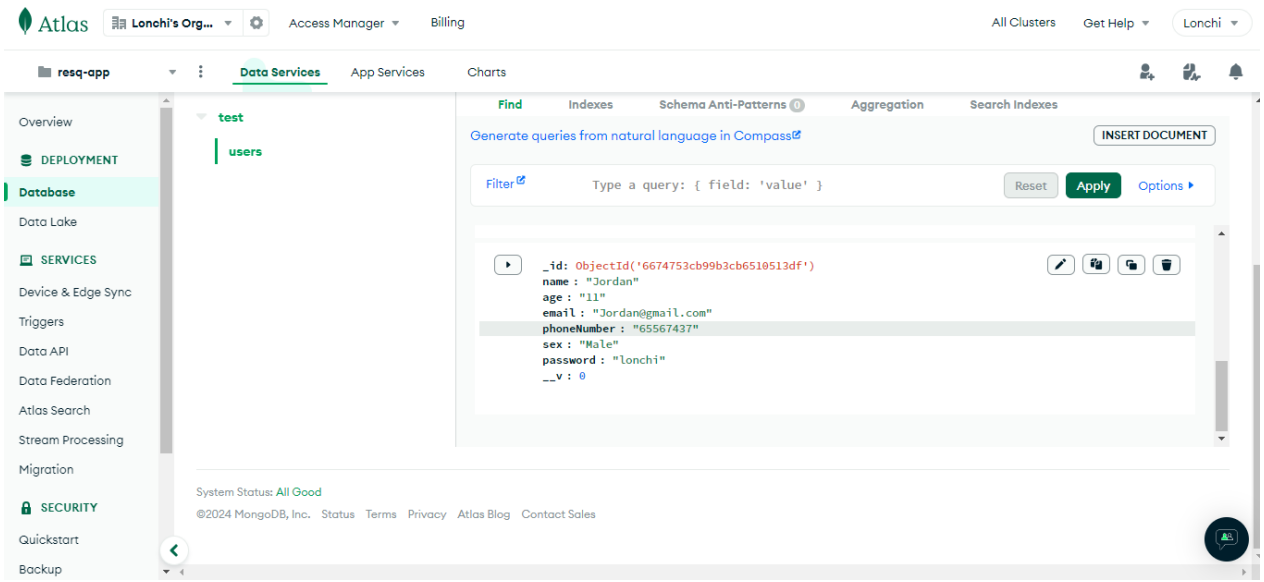
```
const express = require('express')
const { registerUser } = require('../controller/userController')

const router = express.Router()

router.post('/register', registerUser)

module.exports = router;
```

Populating database

Using the Thunder Client tool, which is a REST API client extension for Visual Studio Code, we populated the router by testing and interacting with our API endpoints. Thunder Client allows us to send HTTP requests to our Express.js server to ensure that the routes and controllers are functioning correctly.



Result



,

Conclusion

In conclusion, the database design and implementation for a disaster management mobile application form the backbone of a resilient and effective system. By prioritizing data integrity, scalability, security, and user accessibility, the database supports the critical mission of saving lives and mitigating the impact of disasters. Continuous evolution and enhancement of the database system will further strengthen disaster management efforts, making communities more prepared and resilient in the face of adversity.