REPUBLIC OF CAMEROON
PEACE -WORK-FATHERLAND
UNIVERSITY OF BUEA
BUEA, SOUTH-WEST REGION
P.O BOX 63.FACULTY OF

RÉPUBLIQUE DU CAMEROUN
PAIX-TRAVAIL-PATRIE
UNIVERSITÉ DE BUEA
BUEA, RÉGION DU SUD-OUEST
ENGINEERING B.P. 63.

FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING
SOFTWARE ENGINEERING

CEF440: Internet and Mobile Programming

SYSTEM MODELING AND DESIGN FOR A DISASTER MANAGEMENT MOBILE APPLICATION (TASK 4)

Course Facilitator: Dr. NKEMENI Valery, PhD

Date Issued: 26ST May 2024

Presented by:

GROUP 2

NAMES	MATRICULE
QUINUEL TABOT NDIP-AGBOR	FE21A300
SIRRI THERESIA ANYE	FE21A306
NGONCHI RAMATOU YOLAND	FE21A260
CHE BLAISE NJI	FE21A157
LIMA CHARLES	FE21A225

Table of Content

1. ABSTRACT

Disaster management is a critical area that requires efficient and effective coordination among various stakeholders and users. This work presents a UML-based modelling approach for disaster management mobile application. The proposed approach provides a structured and systematic way to model and design disaster management mobile applications that meet the specific requirements of different stakeholders and users.

2. INTRODUCTION

In order to effectively manage disasters, it is essential to have a comprehensive plan in place that includes the use of mobile technology. The Unified Modelling Language (UML) is a standard modelling language that can be used to design and design software systems. UML provides a set of graphical notations that can be used to represent the different aspects of a software system, including its architecture, behaviour, and data.

In this work, we present a UML-based modelling approach for a disaster management mobile application. The UML Diagrams that will be used are:

- Context diagram
- Use Case diagram
- Class case diagram
- Sequence diagram
- Deployment diagram

Each of the above diagrams will be defined by its various components, procedure of realisation and tools used and diagram

3. CONTEXT DIAGRAM

A **context diagram** is a high-level visual representation that provides an overview of a system's interactions with external entities. It helps define the scope, boundaries, and relationships of a system. In our case of a disaster management mobile application, we used **draw.io software** to draw the context diagram. In our case, we have a system and external entities.

3.1. System:

The Disaster Management System.

3.2. External Entities (Actors):

User: The primary user of the app.

Emergency Services: Entities like police, fire department, ambulance.

Geospatial Data Service: External service providing real-time geospatial data.

System Administrator: Manages and maintains the system.

Government Agency: Provides official alerts, guidelines, and receives incident reports.

3.3. Data Flows

i. User to System:

Submit incident reports.

Customize alert preferences.

Request assistance.

Access Preparedness resources.

Access emergency contacts.

Use offline capabilities.

Switch language preferences. **ii.**

System to User:

Deliver real-time alerts and notifications.

Display interactive maps.

Provide preparedness resources.

Sync data when back online.

iii. System to Emergency Services:

Forward incident reports and assistance requests. **iv.**

Emergency Services to System:

Provide response status.

v. System to Geospatial Data Service:

Request real-time geospatial data.

vi. Geospatial Data Service to System:

Provide geospatial data.

vii. System Administrator to System:

Manage users.

Update resources.

Perform system maintenance.

viii. Government Agency to System:

Provide official alerts and guidelines.

3.4. Context diagram of a Disaster Management Mobile Application.

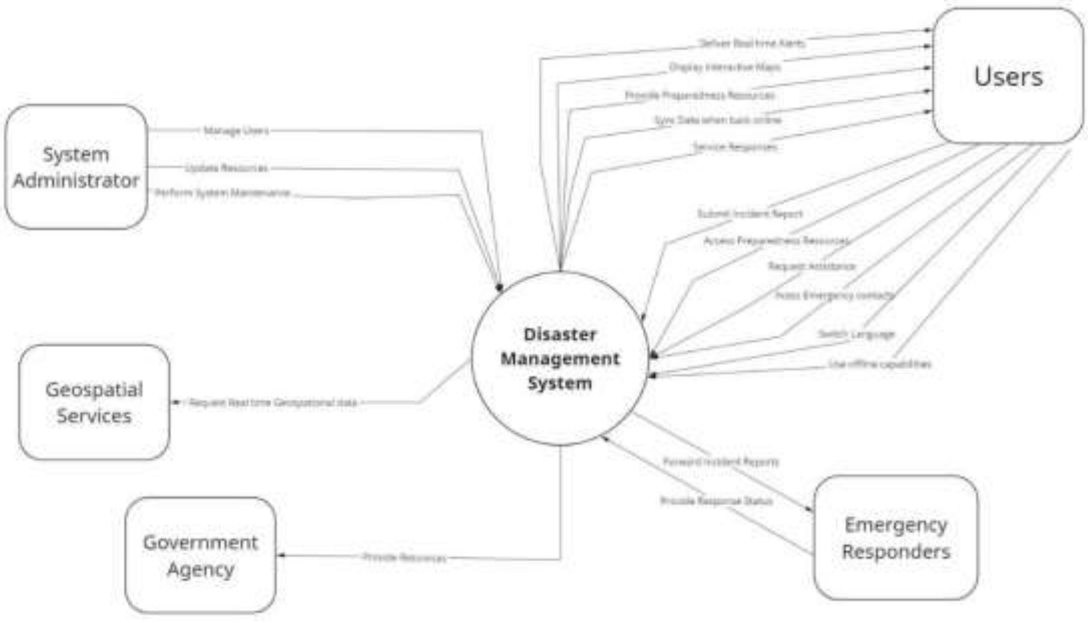

Figure 1 context diagram

4. USE CASE DIAGRAM

A **use case diagram** is a visual representation in the Unified Modeling Language (UML) that illustrates how users (actors) interact with a system. It provides a high-level view of the system's functionality, emphasizing the interactions between users and the system. In our case of a disaster management mobile application, we used **E-Drawmax software** to draw the use case diagram.

4.1. Actors

- **Primary Actors:**

Citizens/Users

Emergency Responders

- **Secondary Actors:**

Government Agencies

- **System Actors:**

System Administrator

4.2. Actors and their Use Cases and the Include and Extend Relationships

i. **User/citizen**

Report Incidents

Include Relationship

Authenticate User

It ensures that only verified and authorized citizens/users can report incidents, preventing potential misuse of the system.

Verify Alert

It validates the authenticity and accuracy of the emergency incident before sending out alerts, ensuring the information is reliable.

Request Emergency Assistance Include

Relationship:

Locate Incident

Receive Alerts and Notifications

Access Preparedness Resources Include Relationship

Authenticate User: It validates the user's login credentials against the user accounts stored in the system's database.

Extend Relationship

Download Resource: downloading is part of accessing resources.

Access Emergency info Include Relationship

Authenticate User: It validates the user's login credentials against the user accounts stored in the system's database.

Access Emergency Contact

Include Relationship

Search Contact: Users Can search contacts of emergency responders

Multilanguage Support Include Relationship

Switch Language: Users can switch between Languages

Login Include Relationship

Enable Single Sign-On: It allows users to log in using their existing credentials from trusted identity providers, such as social media or enterprise accounts. Verify User Credentials: It validates the user's login credentials (username/email and password) against the system's user database.

Authorize User: It determine the user's permissions and access rights based on their role and profile, ensuring they can only perform authorized actions within the system.

ii. **Emergency**

responders

Receive Incident Report Include Relationship

Authenticate User: It validates the user's login credentials against the user accounts stored in the system's database.

Disseminate Emergency Information Provide Medical Assistance

Login

Include Relationship

Enable Single Sign-On: It allows users to log in using their existing credentials from trusted identity providers, such as social media or enterprise accounts. Verify User Credentials: It validates the user's login credentials (username/email and password) against the system's user database.

Authorize User: It determines the user's permissions and access rights based on their role and profile, ensuring they can only perform authorized actions within the system.

iii. **System**

administrator Manage User

Account Maintain the

System

Update Resources iv.

Government agencies

Provide Resources Manage Emergency Resources Disseminate public information

4.3. Use case diagram of a disaster management system.

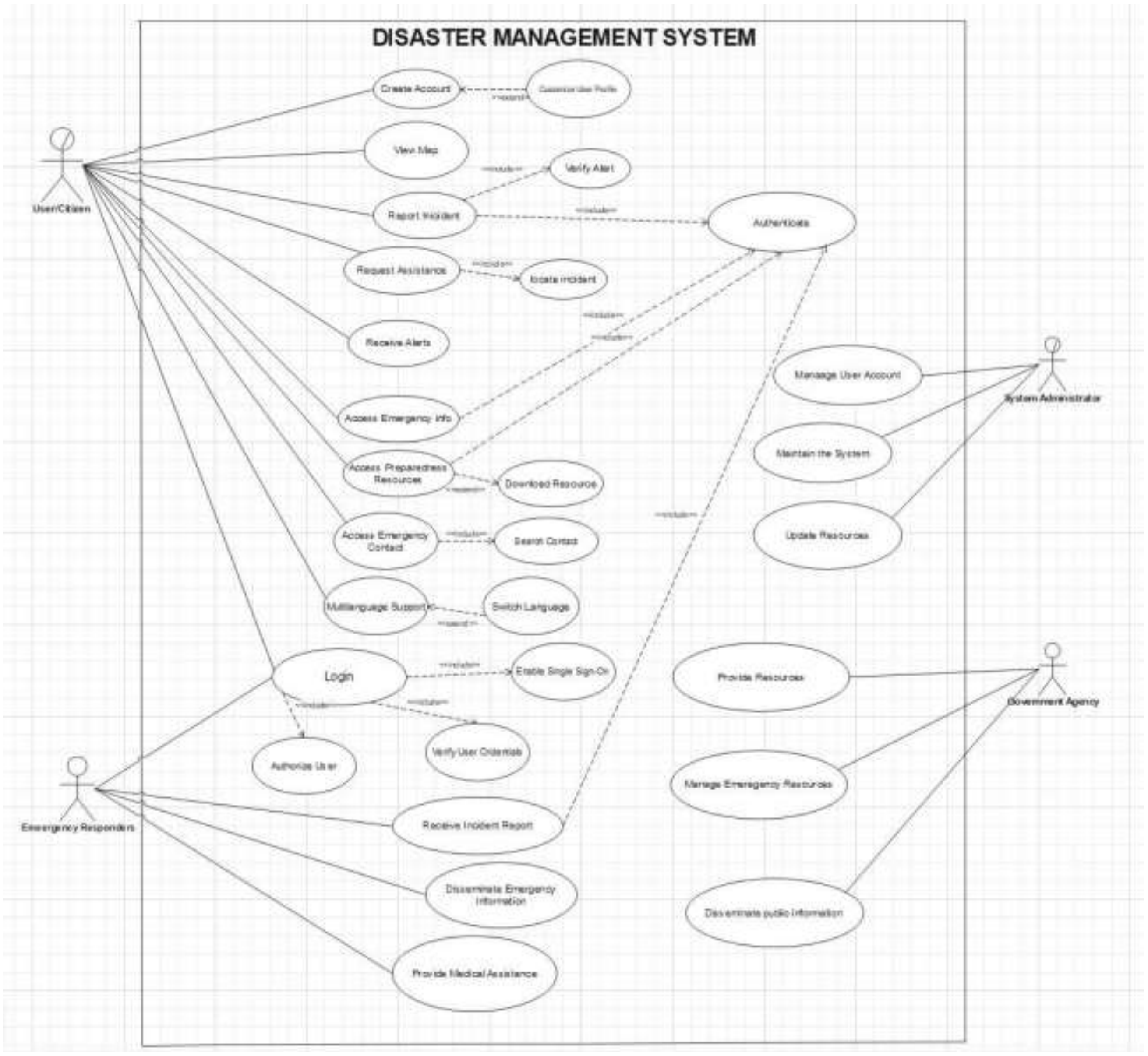

Figure 2: use case diagram

5. CLASS DIAGRAM

A **class diagram** is a type of static structure diagram in the Unified Modelling Language (UML) used in software engineering. It visually represents the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. In our context, we will give the various classes in our disaster management mobile application, define them, give their attributes and methods, followed by the various relations that each class have with another.

5.1. Classes definition, attributes and methods

1. **User:**
   - **Definition:**
     - The **User** class represents individuals who interact with the disaster management mobile application. Users can report incidents, receive alerts, and access relevant information.
   - **Attributes:**
     - userId (string): Unique identifier for each user.
     - username (string): User’s display name.
     - email (string): User’s email address.
     - languagePreference (string): Preferred language for notifications.
     - phoneNumber (string): User’s contact number.
     - emergencyContacts (array of objects): Stores emergency contact information (name, phone number). o

     Methods:
     - registerUser(username: string, email: string): Registers a new user.
     - updateProfile(username: string, email: string): Allows users to update their profile information.
     - setLanguagePreference(lang: string): Sets the preferred language.
     - addEmergencyContact(name: string, phoneNumber: string): Allows users to add emergency contacts. o

     Data Types:
     - string, boolean, datetime, array.

2. **System:**
   - **Definition:**

      The **System** class represents the overall software infrastructure supporting the disaster management application.

o **Attributes:**

 systemId (string): Unique identifier for the system.

 version (string): Current version of the application.

 maintenanceMode (boolean): Indicates if the system is under maintenance.

 lastUpdateTimestamp (datetime): Records the time of the last system update.

o **Methods:**

 checkSystemStatus(): Verifies system availability.

 performSystemUpdate(): Initiates software updates.

o **Data Types:**

 string, boolean, datetime.

3. **Emergency Resources:**

o **Definition:**

 The **Emergency Resources** class manages available resources (e.g., fire trucks, ambulances) for disaster response.

o **Attributes:**

 resourceId (string): Unique identifier for each resource.

 resourceType (string): Type of emergency resource (e.g., medical, firefighting).

 availability (boolean): Indicates if the resource is available.

 location (string): Current location of the resource (GPS coordinates or address).

o **Methods:**

 requestResource(resourceType: string): Requests a specific resource.

 updateResourceAvailability(resourceId: string, available: boolean):

Updates resource availability.

 updateResourceLocation(resourceId: string, location: string): Updates the resource’s location.

o **Data Types:**

 string, boolean.

4. **Alert/Notification:**
   - **Definition:**
     - The **Alert/Notification** class handles communication during emergencies. o **Attributes:**
     - alertId (string): Unique identifier for each alert.
     - message (string): Content of the alert.
     - timestamp (datetime): Time when the alert was generated.
     - severity Level (string): Severity level of the alert (e.g., low, moderate, high).

     o**Methods:**
     - sendAlert(message: string, severity: string): Sends an emergency alert.
     - viewAlert(alertId: string): Displays alert details. o **Data Types:**
     - string, datetime.

5. **Preparedness and Mitigation:**
   - **Definition:**
     - The **Preparedness and Mitigation** class provides tips and guidelines for disaster preparedness. o **Attributes:**
     - tipId (string): Unique identifier for each preparedness tip.
     - tipText (string): Practical advice for disaster preparedness.
     - category (string): Category of preparedness tip (e.g., earthquake, flood). o **Methods:**
     - getRandomTip(): Retrieves a random preparedness tip.
   - **Data Types:**
     - string.
6. **Incident Reporting:**
   - **Definition:**
     - The **Incident Reporting** class allows users to report incidents, providing essential information for emergency responders. o **Attributes:**
     - incidentId (string): Unique identifier for each reported incident.
     - location (string): Incident location description.
     - severity (string): Severity level (e.g., low, moderate, high).

 reportedBy (string): User who reported the incident. ○ **Methods:**

 reportIncident(location: string, severity: string): Allows users to report incidents.

 viewIncidentDetails(incidentId: string): Displays incident details.

 updateIncidentSeverity(incidentId: string, newSeverity: string): Allows authorized users to update incident severity. ○ **Data Types:**

 string.

7. **Location:**
   - **Definition:**

 The **Location** class deals with geospatial information related to incidents. ○ **Attributes:**

 latitude (float): Latitude coordinate of an incident.

 longitude (float): Longitude coordinate of an incident.

 address (string): Human-readable address of the incident location.

 incidentType (string): Type of incident associated with this location
(e.g., fire, flood). ○

Methods:

 getIncidentLocation(incidentId: string): Retrieves the location details of a specific incident.

 convertCoordinatesToAddress(latitude: float, longitude: float):
Converts GPS coordinates to an address. ○

Data Types:

 float, string.

8. **Offline Functionalities:**
   - **Definition:**

 The **Offline Functionalities** class ensures that the app works even when there’s no internet connection. ○ **Attributes:**

 cachedData (object): Stores essential data for offline use (e.g., maps, emergency contacts).

 offlineModeEnabled (boolean): Indicates whether the app is currently in offline mode. o **Methods:**

 syncDataWithServer(): Synchronizes cached data with the server when online.

 enterOfflineMode(): Activates offline functionality.

 exitOfflineMode(): Deactivates offline mode when the device is back online. o **Data Types:**

 object, boolean.

9. **Geospatial Mapping Services:**
   - **Definition:**
      - The **Geospatial Mapping Services** class handles map-related functionalities. o **Attributes:**
      - mapProvider (string): Name of the map service provider (e.g., Google Maps, Leaflet).
      - mapLayers (array of strings): Available map layers (e.g., streets, satellite, topography).
      - defaultZoomLevel (integer): Default zoom level for the map.
   - **Methods:**
      - displayMap(location: string, zoomLevel: integer): Shows incident locations on the map.
      - switchMapLayer(layerName: string): Allows users to switch between different map layers. o **Data Types:**
      - string, array, integer.

10. **Multilanguage Support:**
   - **Definition:**
      - The **Multilanguage Support** class ensures that the app can be used by speakers of different languages. o **Attributes:**
      - supportedLanguages (array of strings): List of languages supported by the app.
      - currentLanguage (string): User’s selected language.

- languageResources (object): Contains translated strings for different languages. ○ **Methods:**
- setLanguage(lang: string): Allows users to switch between supported languages.
- translateText(text: string, targetLanguage: string): Translates text to the specified language.
- getLocalizedResource(resourceKey: string): Retrieves a localized string based on the current language. ○ **Data Types:**
- string, array, object.

11. **Category:**
    - ○ **Definition:**
        - The **Category** class organizes incidents into predefined categories, making it easier for users and responders to understand the nature of each incident. ○ **Attributes:**
        - categoryId (string): Unique identifier for each category (e.g., fire, flood, earthquake).
        - categoryName (string): Descriptive name of the category (e.g., "Fire Incidents," "Flood Alerts").
        - categoryIcon (string): Icon representing the category (e.g., 🔥 for fire, 💧 for flood).
        - incidentCount (integer): Number of incidents associated with this category.
        - colorCode (string): Hex code for the category color (e.g., "#FF5733" for fire).
    - ○ **Methods:**
        - getCategoryName(categoryId: string): Retrieves the name of a specific category.
        - getCategoryIcon(categoryId: string): Retrieves the icon associated with a category.
        - getIncidentCount(categoryId: string): Returns the total number of incidents in this category.

- setCategoryColor(categoryId: string, colorCode: string): Allows customization of category colors. o **Data Types:**
- string, integer.

5.2. Relationships between classes

Describing the relationships between the classes in the context of a disaster management mobile application, considering various types of relationships:

1. **User**:
   - **Relationships**:
     - **Association with Incident Reporting**: Users report incidents (association). Each user can report multiple incidents.
     - **Dependency on Multilanguage Support**: Users rely on the **Multilanguage Support** class for language preferences (dependency).
2. **System**:
   - **Relationships**:
     - **Dependency on Emergency Resources**: The system relies on the **Emergency Resources** class for resource allocation (dependency).
     - **Dependency on Offline Functionalities**: The system depends on the **Offline Functionalities** class for offline functionality (dependency).
3. **Emergency Resources**:
   - **Relationships**:
     - **Association with Incident Reporting**: Emergency resources are dispatched based on incident reports (association). Each resource can be associated with multiple incidents.
     - **Dependency on Geospatial Mapping Services**: Emergency resources use geospatial data for navigation (dependency).
4. **Alert/Notification**:
   - **Relationships**:
     - **Association with User**: Alerts and notifications are sent to users during emergencies (association). Each user can receive multiple alerts.
     - **Dependency on Multilanguage Support**: Alerts are translated into the user’s preferred language (dependency).

5. **Preparedness and Mitigation**:
    - **Relationships**:
        - **Association with User**: Preparedness tips and mitigation strategies are provided to users (association). Each user can access multiple tips.
6. **Incident Reporting**:
    - **Relationships**:
        - **Association with User**: Users report incidents (association). Each user can report multiple incidents.
        - **Dependency on Location**: Incident locations are managed by the **Location** class (dependency).
7. **Location**:
    - **Relationships**:
        - **Association with Incident Reporting**: Stores incident locations reported by users (association). Each incident is associated with a specific location.
        - **Dependency on Geospatial Mapping Services**: Converts GPS coordinates to addresses (dependency).
8. **Offline Functionalities**:
    - **Relationships**:
        - **Dependency on System**: The system relies on the **Offline Functionalities** class for offline functionality (dependency).
        - **Association with User**: Users benefit from offline functionality during emergencies (association).
9. **Geospatial Mapping Services**:
    - **Relationships**:
        - **Dependency on Location**: Converts incident coordinates to addresses (dependency).
        - **Dependency on Emergency Resources**: Provides maps and navigation for emergency responders (dependency).
10. **Multilanguage Support**:
    - **Relationships**:
        - **Dependency on User**: Users set their preferred language using the **Multilanguage Support** class (dependency).

        - **Dependency on Alert/Notification**: Translates alerts into the user’s chosen language (dependency).

11. **Category**:
    - **Relationships**:
        - **Association with Incident Reporting**: Incidents are categorized using the **Category** class (association). Each incident belongs to a specific category.
        - **Dependency on Preparedness and Mitigation**: Categories align with specific preparedness tips (dependency).

5.3. Class Diagram of a Disaster Management Mobile Application

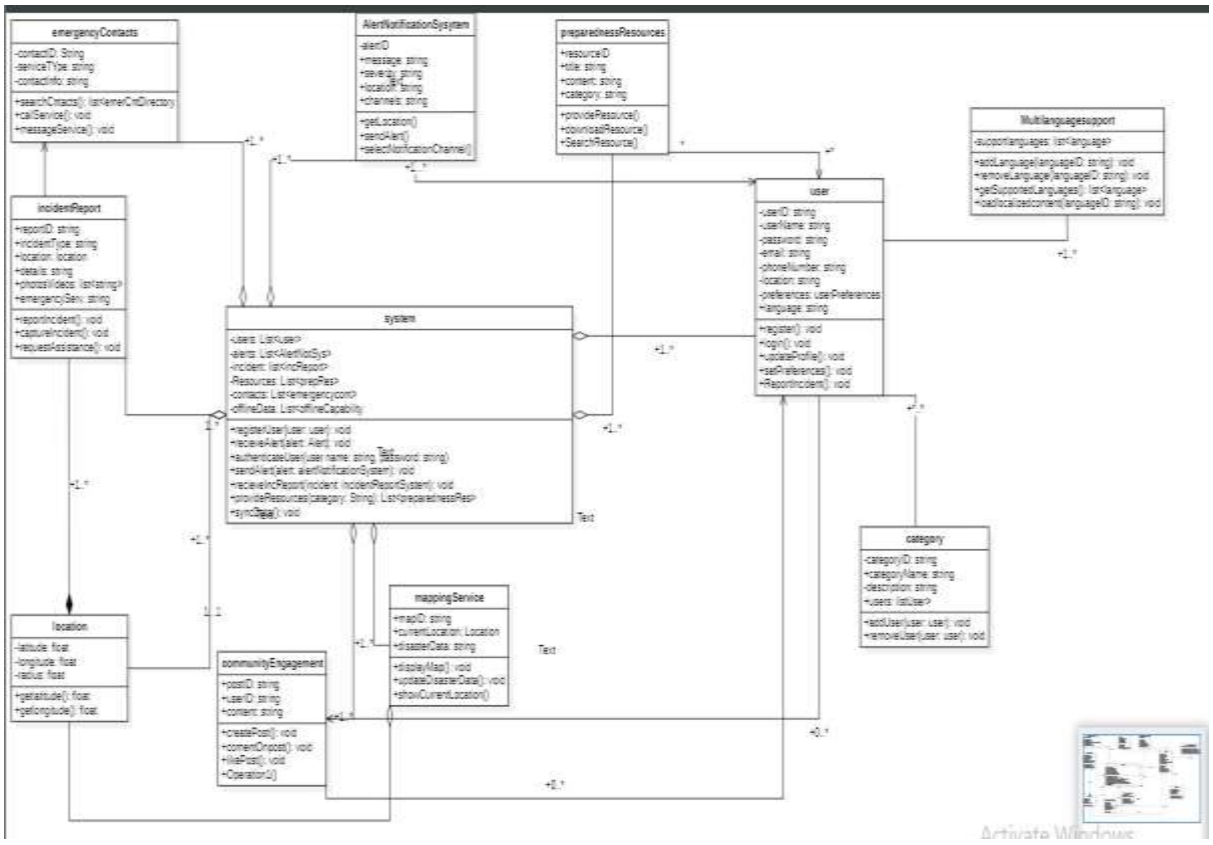

Figure 3: class diagram

These relationships define how classes interact within the system, ensuring effective communication and functionality.

6. SEQUENCE DIAGRAM

A **sequence diagram** for a disaster management mobile app illustrates the interactions and order of events between different components or actors within the system in our context, we will have the following **objects:**

6.1. Objects, functions and Sequential Procedure

i. **User**:

who registers in the application, by proving the required credentials; name, Gmail, location, password and etc. the user can login to the system, and he is capable of receiving alerts/notifications from the system, in addition, the user can view displays options from the system when he/she login. The system in return prompts the user to select and option, while being able to access emergency resources/responders and also able to report incidents based on location and category. ii. **System:**

- o The **system** refers to the entire software infrastructure that supports the disaster management app. It includes the server, databases, APIs, and client-side applications (mobile app and web interfaces).
- o Responsibilities:
  - Handles creation of new user account.
  - Handle user authentication and authorization.
  - Manage data storage and retrieval.
  - Coordinate communication between different components.
  - Ensure overall system reliability and scalability.

iii. **Alert/Notification:**

- o The **alert/notification** component is responsible for informing users and emergency responders about critical events.
- o Responsibilities:
  - Send real-time alerts to users during emergencies (e.g., push notifications).
  - Notify emergency responders about incidents.
  - Provide customizable alert settings for users.

iv. **Emergency Resources/Responders:**

a. The **emergency resources/responders** component manages the availability and deployment of emergency personnel and resources.

b. Responsibilities:

   i. Maintain a database of available responders (firefighters, police, medical personnel).

   ii. Assign responders to incidents based on their proximity and expertise. iii. Track their status and location during an emergency.

v. **Incident Reporting:**

a. The **incident reporting** feature allows users to report emergencies or incidents.

b. Responsibilities:

   i. Capture incident details (type, location, severity).

   ii. Enable users to attach photos or videos.

   iii. Forward incident reports to the server for processing.

vi. **Category:**

a. The **category** component organizes incidents into predefined categories (e.g., fire, flood, earthquake).

b. Responsibilities:

   i. Classify incidents based on their nature.

   ii. Help emergency responders prioritize their actions.

vii. **Location:**

a. The **location** component deals with geospatial information.

b. Responsibilities:

   i. Retrieve the user's location (GPS coordinates).

   ii. Display incident locations on maps.

   iii. Assist responders in navigation.

viii. **Preparedness and Mitigation:**

a. The **preparedness and mitigation** component focuses on proactive measures to reduce disaster impact.

b. Responsibilities:

   i. Provide safety tips and guidelines to users. ii.
   Educate users on disaster preparedness.

   iii. Promote community resilience.

ix. **Multilanguage Support:**

a. The **multilanguage support** feature ensures that the app can be used by speakers of different languages.

b. Responsibilities:

   i. Offer language selection during app setup.

   ii. Translate alerts, instructions, and user interfaces.

   iii. Facilitate communication between responders and users who speak different languages.

x. **Offline Functionalities:**

a. The **offline functionalities** allow the app to work even when there's no internet connection.

b. Responsibilities:

  i. Cache essential data (maps, emergency contacts, safety guidelines).

  ii. Enable users to report incidents offline (data syncs when online).

xi. **Geospatial Data:**

a. The **geospatial data** component handles geographic information.

b. Responsibilities:

  i. Store and retrieve maps, and geospatial layers.

  ii. Overlay incident locations on maps.

  iii. Calculate distances and travel times.

xii. **System Updates:**

a. The **system updates** component ensures that the app remains secure and up-to-date.

b. Responsibilities:

  i. Regularly check for app updates.

  ii. Install security patches.

  iii. Notify users about new features or improvements.

Based on the above objects and their step-to-step relations of each other, **draw.io software** is used to realize the below sequence diagram.

6.2. Sequence Diagram of a disaster management mobile application.

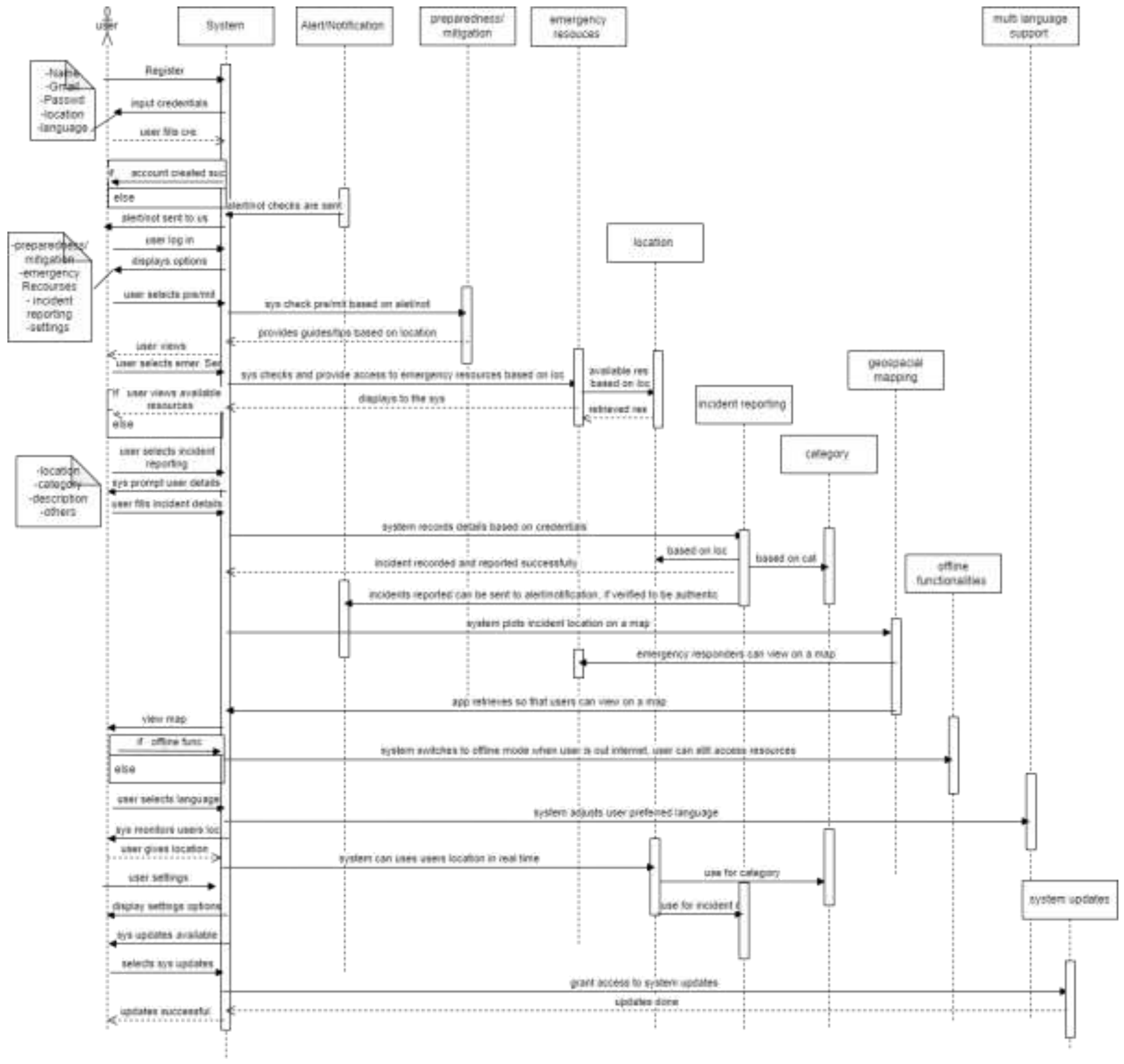
user
System
Alert/Notification
preparedness/ mitigation
emergency resouces
multi language support
location
geospacial mapping
incident reporting
category
offline functionalities
system updates
-Name -Gmail -Passwd -location -language
Register
input credentials
user fills cre
if account created suc
else
alert/not checks are sent
alert/not sent to us
-preparedness/ mitigation -emergency Recourses - incident reporting -settings
user log in
displays options
user selects pre/mit
sys check pre/mit based on alert/not
provides guides/tips based on location
user views
user selects emer. Ser
sys checks and provide access to emergency resources based on loc
available res based on loc
retrieved res
displays to the sys
if user views available resources
else
user selects incident reporting
-location -category -description -others
sys prompt user details
user fills incident details
system records details based on credentials
based on loc
based on cat
incident recorded and reported successfully
incidents reported can be sent to alert/notification, if verified to be authentic
system plots incident location on a map
emergency responders can view on a map
app retrieves so that users can view on a map
view map
if offline func
else
system switches to offline mode when user is out internet, user can still access resources
user selects language
system adjusts user preferred language
sys monitors users loc
user gives location
system can uses users location in real time
use for category
use for incident
user settings
display settings options
sys updates available
selects sys updates
grant access to system updates
updates done
updates successful

Figure 4: sequence diagram

7. DEPLOYMENT DIAGRAM

A **deployment diagram** in the context of a disaster management system mobile application illustrates how software components and hardware nodes interact and are distributed across a network or infrastructure. Let’s break down the key aspects of a deployment diagram for such an application:

7.1. Components:

1. **Users:** These are the individuals who will utilize the disaster management mobile application. They interact with the app's features to receive alerts, access resources, and potentially report information during emergencies.
2. **Mobile Network:** This network provides internet connectivity for users' devices to communicate with the application server.
3. **React Native App (Frontend):**
    - Developed using the React Native framework, this app runs on users' smartphones and tablets.
    - It utilizes various React Native components to build the user interface (UI) for functionalities like:
        - Displaying disaster alerts and updates.
        - Allowing users to access resources like emergency shelters, evacuation routes, and contact information for aid organizations.
        - Potentially enabling features for reporting damage, missing persons, or resource needs (depending on app design).
    - The app communicates with the Node.js API server using well-defined APIs (Application Programming Interfaces) that follow protocols like REST (Representational State Transfer).
4. **Content Delivery Network (CDN) (Optional):**
    - This can be integrated to improve app performance by caching static content (images, Javascript files) geographically closer to users. This reduces latency and improves loading times, especially in remote areas.
5. **Node.js API Server (Backend):**
    - Developed using Node.js and Express.js framework, this server acts as the intermediary between the mobile app and the database.
    - It handles incoming API requests from the mobile app and performs the following actions:
        - Validates user requests and performs authentication checks (if applicable).
        - Interacts with the MongoDB database to retrieve or store data relevant to the request (e.g., disaster alerts, resource locations).

        - Processes data as needed (e.g., filtering, aggregation) before sending responses back to the mobile app.
        - May handle real-time communication functionalities (push notifications) using technologies like websockets or Firebase Cloud Messaging (FCM).

6. **API Gateway (Optional):**
    - This can be introduced as an additional layer of security and management for API access. It acts as a single entry point for API requests, offloading security concerns from the Node.js server.
    - It can handle features like:
        - Rate limiting to prevent abuse of the API.
        - Authentication and authorization for different user roles.
        - API usage analytics.
7. **Cloud Load Balancer:**
    - This service distributes incoming traffic across multiple instances of the Node.js server for scalability and fault tolerance. It ensures that the application remains responsive even during high user traffic situations.
8. **MongoDB Database:**
    - This NoSQL database (MongoDB Atlas in the cloud) stores all the application's data relevant to disaster management. This includes:
        - Disaster information (type, severity, location)
        - Resource information (shelters, evacuation routes, aid organizations)
        - User information (if applicable) for functionalities like reporting or managing user profiles.

7.2. Communication Flow:

1. **User Interaction:** Users interact with the mobile app's UI elements, triggering actions that send API requests to the Node.js server.
2. **API Request:** The mobile app formulates an API request containing relevant data (e.g., user location, disaster type) and sends it over the internet through the mobile network.

3. **Content Delivery Network (Optional):** If a CDN is implemented, it might intercept the request and serve cached static content if available. This reduces load on the server and improves app performance.
4. **API Gateway (Optional):** The API Gateway receives the request, performs any necessary security checks (authentication, authorization), and routes it to the appropriate backend service (Node.js server).
5. **Node.js Server:** The Node.js server receives the request, validates it, and interacts with the MongoDB database.
    - It retrieves or stores data as needed based on the request.
    - It may perform data processing or manipulation before preparing a response.
6. **Response:** The Node.js server formulates a response containing the requested data or relevant messages (e.g., success/failure status) and sends it back to the mobile app.
7. **Mobile App Update:** The mobile app receives the response from the server and updates its UI accordingly. It might display retrieved data (e.g., disaster alerts, resource locations) or handle success/failure messages for user actions.

7.3. Deployment:

- **Mobile App:** The React Native app is typically compiled for Android platforms and deployed to their respective app stores (Apple App Store and Google Play Store).
- **Node.js Server:** The Node.js server application is deployed to a cloud platform that offers hosting services like

Based on the communication flow, **draw.io software** is used to realize the below deployment diagram.

7.4. Deployment diagram of a disaster management system.

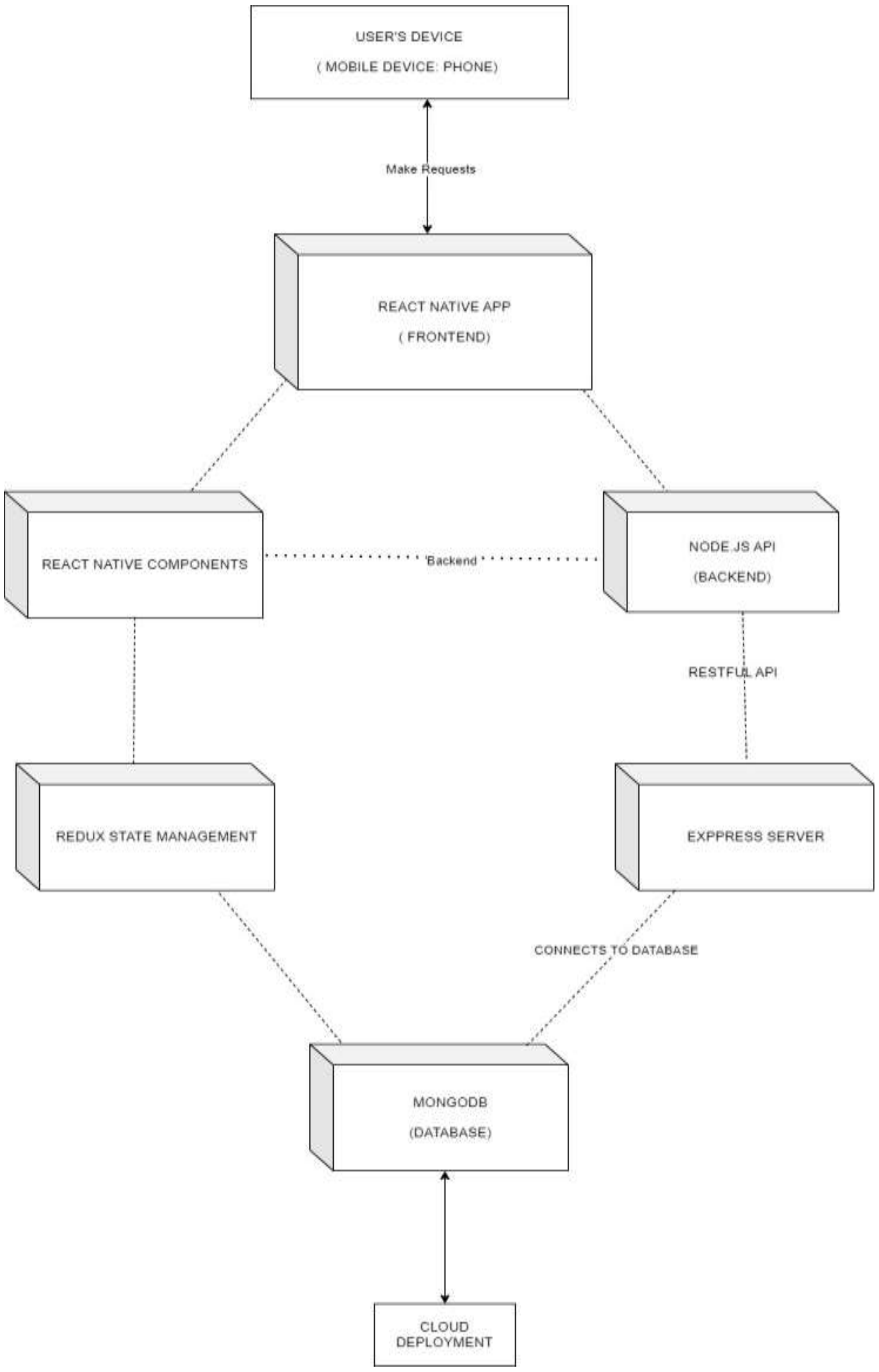

Figure 5: deployment diagram

This detailed deployment diagram provides a comprehensive view of the architecture for a disaster management mobile application built with React Native, Node.js, and MongoDB. By utilizing these technologies and following a well-structured approach, developers can create a robust and scalable application that empowers users with critical information and functionalities during emergencies.

Key benefits of this architecture:

- **Improved User Experience:** React Native allows for a native-like mobile app experience across different platforms.
- **Scalability and Performance:** Cloud deployment with load balancing ensures the application can handle high user traffic and maintain responsiveness.
- **Real-time Communication (Optional):** Integration with technologies like websockets or FCM enables real-time features like push notifications for critical updates.
- **Data Management:** MongoDB provides a flexible NoSQL database solution for storing various disaster management data.

8. CONCLUSION

To conclude, based on the above UML diagrams; context diagram, use case diagram, class diagram, sequence diagram and deployment diagram, the ideal disaster management mobile application task, can be passed across to the next stage, while being open for further corrections when need be.