THE UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

CEF 440: INTERNET PROGRAMMING AND MOBILE PROGRAMMING

Group 22

System Design and Modelling

Task 4

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

I. INTRODUCTION

Our system design and modeling process involved grouping functionalities based on similarities in implementation and breaking them down into modules, each providing a specific set of functions. This approach was used to come up with the system design and the system model using UML diagrams. This ensured the system design adhered to the requirements gathered from users and the various stakeholders.

II. SYSTEM DESIGN OF DISASTER MANAGEMENT SYSTEM

System design in the context of our disaster management system refers to the process of creating a detailed plan or blueprint for a software system that effectively addresses the requirements and specifications. It involves defining the architecture, components, modules, interfaces, and behaviors of the system to ensure its successful implementation.

In realizing our system design we went through the following steps:

A. **Grouping Requirements** whose implementations patterned were closely related or similar as this help easily come up system modules
B. **Identify System Modules and Their Functionalities**: Here we identified the modules from each grouped requirement and functionalities each module is to provide to accomplish a given task
C. **Choose Architectural Style**: Where we choose an architectural style or pattern that suits the system's needs and considers the non-functional requirements of the system, such as performance, scalability, security, and maintainability. These requirements often have a significant impact on the architecture of the system.
D. **Technology Selection**: Choose the technologies that will be used to implement the system. This includes programming languages, databases, frameworks, and other tools.

A. Grouping Requirements with similar implementation pattern

In this stage we group our functionalities using 'ands' where necessary as follows:

1. Request help and Incident reporting
2. User registration and authentication
3. Communication with authorities and Community engagement
4. Real time alerts and notifications
5. Geospatial data, mapping services and Evacuation routes

6. Emergency Resource Access
7. Data privacy and security
8. Offline functionality
9. Multilanguage support
10. User feedback and support

B. Identifying System Modules and their Functionalities

From the grouping of requirements based on similarities in implementation as done above the following modules were produced for each requirement:

1. **Request help and Incident reporting**
   - **Validation of user's statement.**
     - Validate existence of location provided by user
     - Ensure contact information is correctly provided
     - Check disaster type actually exists
     - Ensure video is in appropriate format (mp4) and size limit is not exceeded
     - Ensure description of disaster meet minimum character length and does not contain any malicious characters
   - **Verification of statement's authenticity.**
     - Verify information stated in the user's statement is coherent with geospatial data about that same location provided
     - Disseminate information to closest responders (local authorities) of category police for onsite assessment
     - Evaluate Historical data of reported area using NASA disaster API
     - Advice for social media monitor
   - **Feedback based on authenticity.**
     - User is updated with status of the report or help request made
     - Timely response is provided to user incase the report or help request is validated
     - Disaster info is recorded in database if disaster is valid.
2. **User registration and authentication**

   i. User registration:
   - **Validation**
     - Ensure users credentials entered such as name, email, password, telephone number and preferred locations are of correct format, lengths and also do not contain malicious or unwanted characters
   - **Database storage**
     - This module is in charge of storing user record in database of the system

   ii. **User authentication**:
   - **Input Validation**
     - Ensure users email, password are of correct format, lengths and also do not contain malicious or unwanted characters
   - **Verification**

- Verifies that user's credentials entered exist or is compatible with what exists in the systems data. Access if denied if there is any incorrect credential or record doesn't exist at all

- **Session management**
  - Handles and manages user session granting him/her access to functionalities of the user management system.
  - Creates session's ID and also determines when a session has expired or is invalid.
  - Json Web Token authentications will be used for session management.

3. **Communication with authorities and Community engagement**
   - **Messaging module**
     - Accounts for sending and receiving messages functionality
     - Also include functionality to delete messages
   - **Forum management module.**
     - Manages Forums including it settings and restrictions placed by admins of the Forums
   - **Chatroom management module**
     - Manages Chatroom session and messages
     - Manages chatroom participants
4. **Real time alerts and notifications**
   - **Alerts module**
     - Utilize socket.io provide real-time alerting functionalities for ongoing disasters and potential disasters.
     - Incorporates functionalities for predicting disasters using analytics tools for early warnings.
     - Utilize push notifications for delivery of alerts to users
   - **Notification module**
     - Inform users of mainly new messages in the communities (chatrooms and forums)
     - Provide users with tips and advise on what to and what not to do
5. **Geospatial data, mapping services and evacuation routes**
   - **Detection module**
     - Leverage geospatial data analytics API to make accurate predictions of potential disasters (ArcGIS API and tool)
   - **Display module**
     - Responsible for marking and indication disaster affected areas on map with relevant signs and information such as the kind of disaster, the affected areas. This will utilize mapping services (Google's map be precise)
   - **Modification module**
     - In cases of a confirmed disasters reported by user which was not detected by the system, the disaster confirmed area will receive adjustments or modification on the map by the system
   - **Routing module**

- Responsible for providing safest routes away from the disaster utilizing APIs

6. **Emergency Resource Access**
   - **Civilian emergency Resource module**
     - Updating emergency resources
     - Adding emergency resources
     - Retrieving emergency resources
7. **Data privacy and security**
   - Encryption module
     - Encryption of communication sessions to ensure secure communication channel
     - Encryption of stakeholder data using bcrypt algorithms before transmission and or storage
8. **Offline functionality**
   - **Service worker controller module**
     - This module will mainly be concerned with the control and manipulation of our service workers to ensure offline functionalities
   - **Local SQLite database control module**
     - This module will provide functionalities such as fetching and storing user data in SQLite database while user is offline such user still access to some functionalities of the application.
   - **Synchronization module**
     - This module ensures that user's modifications and actions on data stored in SQLite database and other local storage are implement to match on the actual database when user returns online.
9. **Multilanguage support**
   - **Language support Module**
     - This module will be in charge of holding all supported languages by the system
   - **Language dictionary Module**
     - This module with provide translation functionalities between supported languages using international Libraries like React international translation library such that all translations are done without internet.
10. **User feedback and support**
    - **Q/A AI module**
      - This module will permit uses as disaster related questions in-app and gain insights about the application and disaster management as a whole by leveraging the power of AI.
    - **App feedback**
      - This module will be responsible for permitting user give feedback on their experience in using the application as this will better facilitate updates.

C. Architecture Choice for our Disaster Management System

For the disaster management system, the architecture selection was driven by non-functional requirements, particularly focusing on performance, scalability, and offline functionality. The chosen architecture is a combination of Client-Server and Offline-First. Here's a detailed explanation of the rationale and benefits of this approach:

1. Client-Server Architecture

The client-server architecture divides the system into two main components:

- **Client:** The front-end application (Ionic with React TypeScript) running on users' mobile devices.
- **Server:** The back-end system (Node.js) handling the business logic, data processing, and database interactions (MongoDB).

Reason:

- **Performance:**
  - **Server-Side Processing:** Offloading the computationally intensive tasks to the server ensures that mobile devices (clients) remain responsive. The server, being more powerful, handles complex operations like data encryption, route optimization, and real-time notification management.
  - **Efficient Data Handling:** By centralizing data processing on the server, the system can efficiently manage large datasets and perform operations like data aggregation and analysis.
- **Scalability:**
  - **Centralized Management:** Centralizing the core functionalities on the server simplifies updates, maintenance, and scaling, as changes can be made in one place and reflected across all clients.
  - **Horizontal Scaling:** The server can be scaled horizontally by adding more instances behind a load balancer, ensuring the system can handle a large number of concurrent users, especially during disaster scenarios when user activity spikes.

2. Offline-First Architecture

Offline-first architecture ensures the application remains functional even without a stable internet connection. This approach prioritizes local data storage and synchronization, allowing users to interact with the system and later sync with the server when connectivity is restored.

Reason:

- **Reliability in Disasters:**
  - **Offline Functionality:** During disasters, network connectivity can be unreliable or unavailable. An offline-first approach ensures users can still report incidents, access critical information, and use core features of the application without needing an internet connection.

    - **Data Synchronization:** Once the connection is re-established, the application can synchronize local data with the server, ensuring that all information is up-to-date and consistent across the system.
- **Performance:**
    - **Local Storage:** Storing data enables faster access to information compared to fetching data from the server, improving user experience.
    - **Reduced Latency:** Local operations reduce latency and provide instant feedback to users, essential for emergency scenarios where timely information is crucial.

Benefits:

- **User Independence:** Users can rely on the app even in areas with poor connectivity, increasing the app's reliability and usability during disasters.
- **Continuous Operation:** Ensures the app remains operational and users can continue to perform critical tasks without interruption.
- **Data Integrity:** Synchronization mechanisms handle conflict resolution and data merging, maintaining data integrity across the system.

D. Technologies

1. **Client: Ionic with React TypeScript**

Ionic Framework:

- **Cross-Platform Development:**
    - **Advantage:** Ionic allows for the development of a single codebase that can be deployed across multiple platforms (iOS, Android, and web). This significantly reduces development time and cost.
    - **Relevance to Disaster Management:** Ensures that the app can reach the maximum number of users quickly, which is crucial during disaster situations.
- **Native-like Performance:**
    - **Advantage:** Ionic provides components that mimic native behaviors, giving users a smooth and familiar experience.
    - **Relevance to Disaster Management:** A seamless and responsive UI is essential for users who might need to act quickly in emergency situations.

React with TypeScript:

- **Component-Based Architecture:**
    - **Advantage:** React's component-based architecture promotes reusability and maintainability of the code.
    - **Relevance to Disaster Management:** Components like forms for reporting incidents, maps for evacuation routes, and real-time notifications can be reused and maintained easily.
- **Type Safety with TypeScript:**
    - **Advantage:** TypeScript adds static typing to JavaScript, which helps in catching errors during development, improving code quality and reliability.

- **Relevance to Disaster Management:** Ensures the application is robust and less prone to runtime errors, which is critical in emergency applications where reliability is paramount.

2. **Server: Node.js**

- **Event-Driven, Non-Blocking I/O:**
  - **Advantage:** Node.js is designed for handling asynchronous operations, which makes it ideal for real-time applications.
  - **Relevance to Disaster Management:** Enables real-time notifications and updates, such as alerts, messages, and incident reports, ensuring that users receive timely information.
- **Scalability:**
  - **Advantage:** Node.js can handle a large number of simultaneous connections with high throughput, making it highly scalable.
  - **Relevance to Disaster Management:** Can manage a large influx of users and data during a disaster when the usage spikes unexpectedly.
- **Rich Ecosystem:**
  - **Advantage:** Node.js has a vast ecosystem of modules and libraries (npm), which can accelerate development.
  - **Relevance to Disaster Management:** Libraries for real-time communication (e.g., Socket.io), geospatial data handling, and encryption can be readily integrated into the application.

3. **Database: MongoDB**

- **Flexible Schema Design:**
  - **Advantage:** MongoDB is a NoSQL database that uses a flexible schema, allowing for the storage of diverse data types and structures.
  - **Relevance to Disaster Management:** Can efficiently store varied data such as user profiles, incident reports, real-time messages, and resource information without the need for complex migrations.
- **High Performance and Scalability:**
  - **Advantage:** MongoDB is designed for high performance, handling large volumes of read and write operations efficiently.
  - **Relevance to Disaster Management:** Supports the quick retrieval and storage of large amounts of data during disasters, such as real-time incident updates and user communications.
- **Geospatial Capabilities:**
  - **Advantage:** MongoDB has strong support for geospatial queries, which is essential for location-based services.
  - **Relevance to Disaster Management:** Can store and query location data for incidents, resources, and evacuation routes, providing critical information based on users' locations.

III. SYSTEM MODELLING

System modeling is the process of creating abstract representations (models) of a system to understand, analyze, and communicate its structure and behavior. These models help stakeholders visualize the system, identify requirements, detect issues early, and guide the development process. System modeling involves various techniques and diagrams to represent different aspects of the system which include

1. Context Diagram
2. Use Case Diagram
3. Sequence Diagram
4. Class Diagram
5. Deployment Diagram

1. CONTEXT DIAGRAM

A context diagram (Level 0 Data Flow Diagram) is a high-level visual representation of a system and its interactions with external entities. It focuses on the entire system as a single process, outlining how data flows into and out of the system. It is used in systems analysis to show the boundaries of a system and the external factors that interact with it.

In a context diagram, external entities are anything outside the system boundary that interacts with the system by exchanging data. They represent the actors or elements that influence the system or rely on the system's functionality.

Types of External Entities:

- **Users:** these are individuals who directly interact with the system
- **Other Systems:** these are external software systems that exchange data with our system.
- **Databases:** External databases that store and provide data to our system.
- **Sensors and Devices:** In some systems, external entities might be physical devices that provide data or receive instructions from the system.
- **External Services:** external services that our system interacts with to obtain specific functionalities.

Identifying External Entities:

When creating a context diagram, consider these questions to identify relevant external entities:

- Who are the users of the system?
- What other systems does this system interact with?
- What data sources (databases) does the system rely on?
- Are there any external devices or sensors involved?
- Does the system utilize any external services?

Components of a Context Diagram:

- **System:** Represented as a circle in the center. It represents the entire system.
- **External Entities:** Shown as squares or rectangles outside the system boundary.
- **Data Flows:** Indicated by arrows connecting the system to external entities.

Context Diagram for Our Disaster Management System

A. External Entities:

1. **User**: Individuals using the system to register, receive alerts, report incidents, request for help, access resources, communicate, and provide feedback.
2. **Authorities**: Emergency response teams, government agencies, NGO's etc. They can respond to users in distress and also do resource allocation.
3. **External Services**: External services providing real-time alerts and notifications, and real-time geospatial and mapping data (Google maps).

B. Data Flows:

1) User to System:
- Registration Data (email, password, name, etc.)
- Incident Reports (text, photos, videos)
- Requests for Help
- User Preferences (alert preferences, language settings)
- Feedback and Support Requests
- Communication Messages to authorities

2) System to User:
- Authentication Confirmation
- Real-Time Alerts and Notifications
- Communication Messages from authorities

3) Authorities to System:
- Incident Response
- Validate incident occurrence
- Evacuation Orders

4) System to Authorities:
- Incident Reports
- Requests for Help
- User Communications

5) External Services to System:
- Real-time Alerts and Notifications
- Real-Time Geospatial Data (maps, disaster areas, evacuation routes) e.g. Google Maps API.

6) System to External Services:
- Alert Requests (based on user preferences and geographic locations)
- Requests for Geospatial Data

C. Diagram:

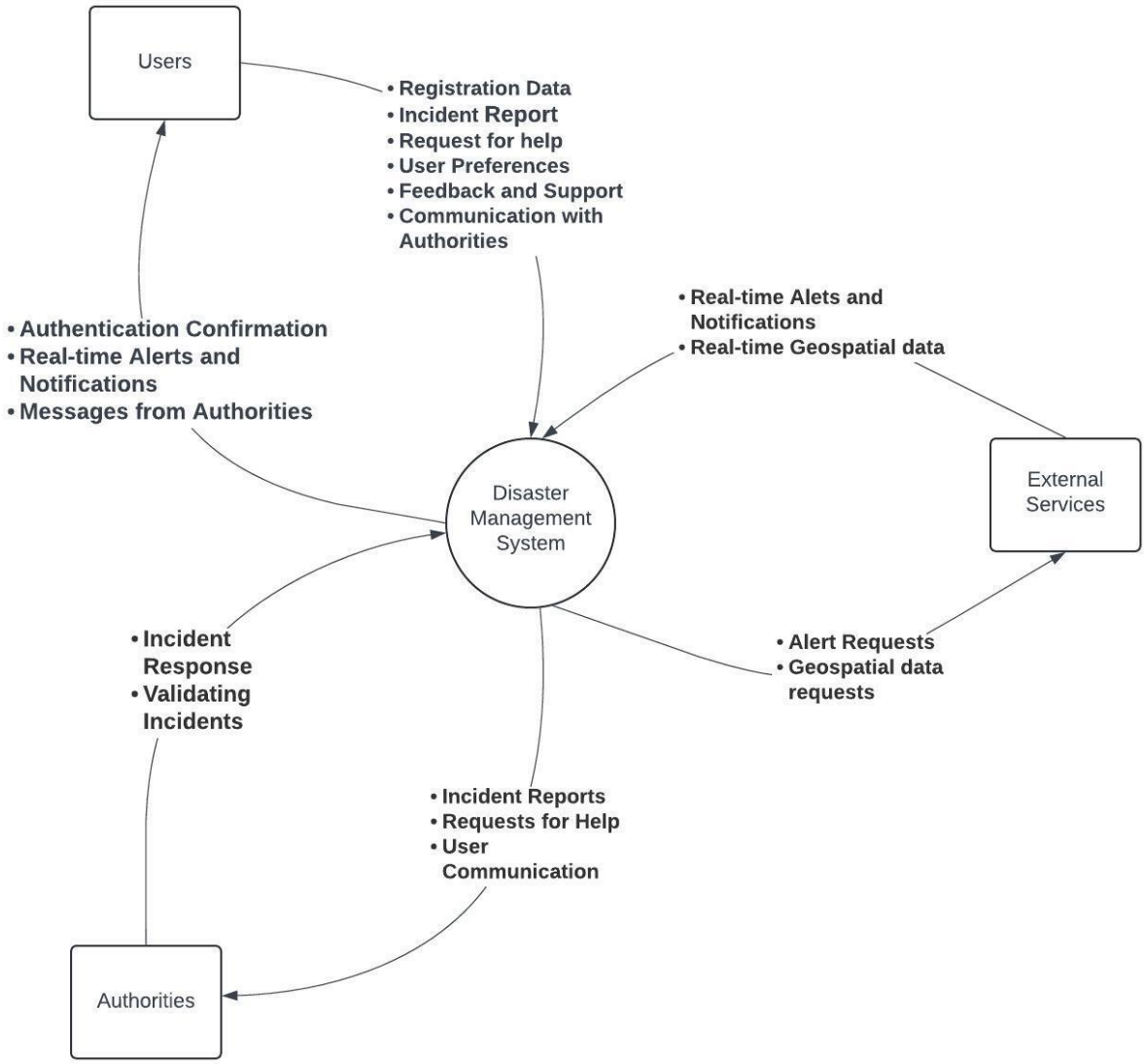


Fig: Context Diagram

D. Explanation:

i. **User**: interacts with the Disaster Management System by providing registration data, incident reports, preferences, feedback, and support requests. They receive authentication confirmations, real-time alerts and communication messages.

ii. **Authorities:** receive incident reports, help requests and user communications from the system and provide incident response updates (incident validation) and resource availability information.

iii. **External Services**: provide real-time alerts and geospatial data, which the system uses to update users and authorities. The system sends requests to external services for generating alerts and obtaining geospatial data.

This context diagram provides a high-level overview of the disaster management system's interactions with external entities, highlighting the flow of information essential for effective disaster response and management. By integrating these components and data flows, the Disaster Management System ensures effective communication, timely alerts, and accurate information dissemination during emergencies, enhancing overall disaster response and management.

2. USE CASE DIAGRAM

A use case diagram is a graphical representation that depicts how users interact with a system and its functionalities. It illustrates the various use cases or actions that users (known as actors) can perform within the system and the relationships between these use cases.
In the context of disaster management, a use case diagram would depict the various interactions between users, such as individuals affected by disasters, emergency responders, and external systems, and the functionalities provided by the disaster management system.

Actors:

1. **User**: Represents individuals using the mobile-based disaster management system to access functionalities and receive assistance during emergencies.
2. **Emergency Responders**: Represents personnel responsible for managing and responding to disaster incidents.
3. **External APIs (Geospatial Data Provider)**: Represents external services providing geospatial data, evacuation routes, and disaster validation.

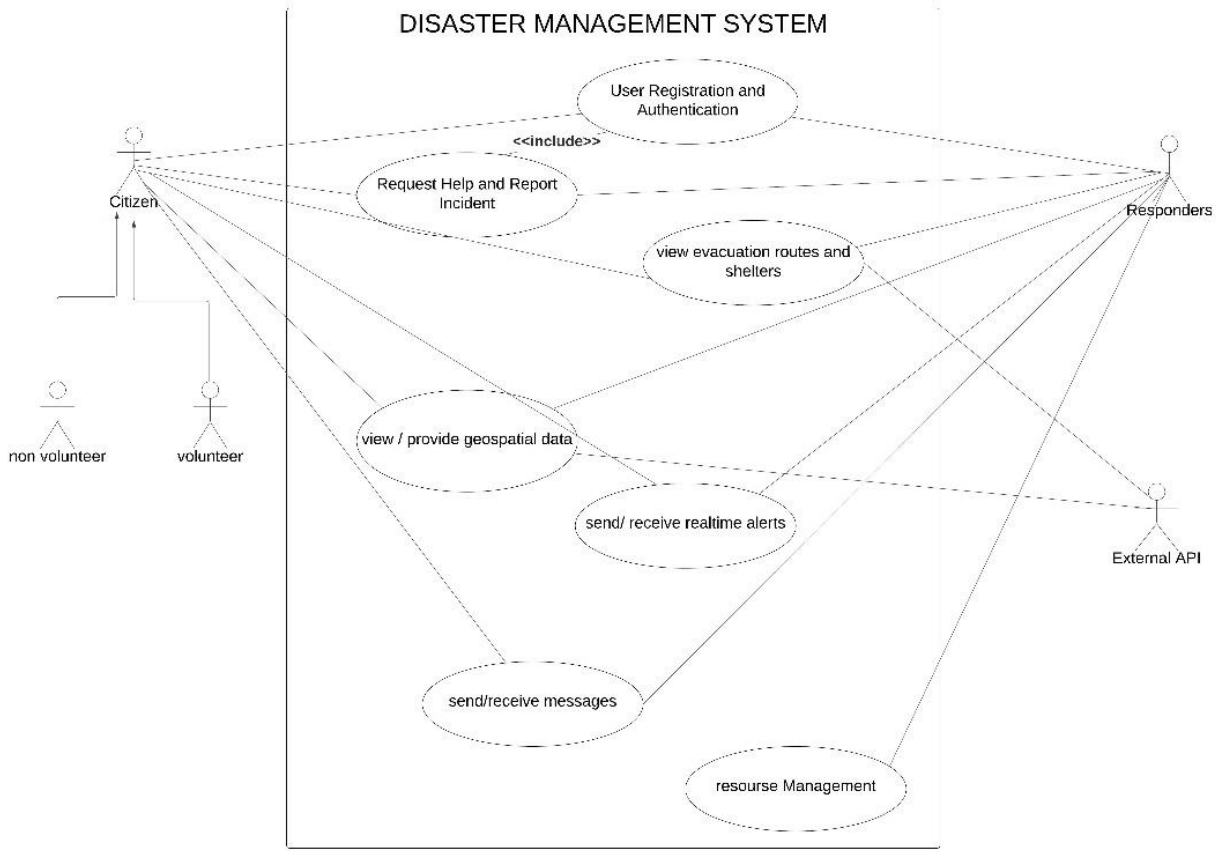


Fig: Use Case Diagram

Description of Use Case

1. **User Registration and Authentication**:
    - User: Registers for an account and authenticates identity to access system functionalities.
    - Emergency Responders: Register and authenticate their identities to access specialized features and resources.
2. **Report Disaster and Request for Help**:
    - User:
        - Reports disaster incidents or requests help during emergencies.
    - Emergency Responders:
        - Validate reported disasters and provide assistance or resources as needed.
    - External APIs:
        - Validate disaster occurrences and provide additional data for incident verification and also disaster data based on user location.
3. **Geospatial Data and Evacuation Routes**:
    - User:
        - Accesses geospatial data and evacuation routes related to ongoing or potential disasters for navigation and safety.
    - Emergency Responders:

        - Utilize geospatial data analytics and evacuation routes to coordinate response efforts and assist affected populations.
    - External APIs:
        - Provide geospatial data analytics and evacuation routes to enhance situational awareness and decision-making.
4. **Resource Management**:
    - Emergency Responders:
        - Manage and allocate resources efficiently across all disaster phases to meet evolving needs and priorities.
5. **Messaging**:
    - User:
        - Sends and receives messages to communicate with emergency responders, authorities, and other users.
    - Emergency Responders:
        - Communicate with users and coordinate response efforts through messaging functionalities.
6. **Real-time Alerts and Notifications**:
    - User:
        - Receives real-time alerts and notifications about ongoing or potential disasters for immediate action.
    - Emergency Responders:
        - Provide data about ongoing disasters to the system for generating real-time alerts and notifications.
    - External APIs:
        - Contribute to the data pool for real-time alerts and notifications by validating disaster occurrences and providing relevant information.

3. SEQUENCE DIAGRAM

A sequence diagram is a visual representation of the message flow between objects in a system, arranged in chronological order. It shows process interactions arranged in time sequence.

In the context of a disaster management system, these diagrams will illustrate how users, system components, and external sources interact to facilitate various actions during a disaster.

Here's a breakdown of the functionalities explored through these sequence diagrams, with corresponding figures illustrating the message flow:

1) **User Registration and Authentication:** This diagram illustrates the process of a user registering for an account and subsequently logging in to access the system's functionalities. We'll see how the user interacts with the system's registration component and the authentication service to gain authorized access.

- **User:** Represents the individual attempting to register and access the disaster management system.
- **Disaster Management System (DMS):** This encompasses the functionalities for user registration, login, and user interface for interaction.
- **Database Server:** This server stores user information securely.

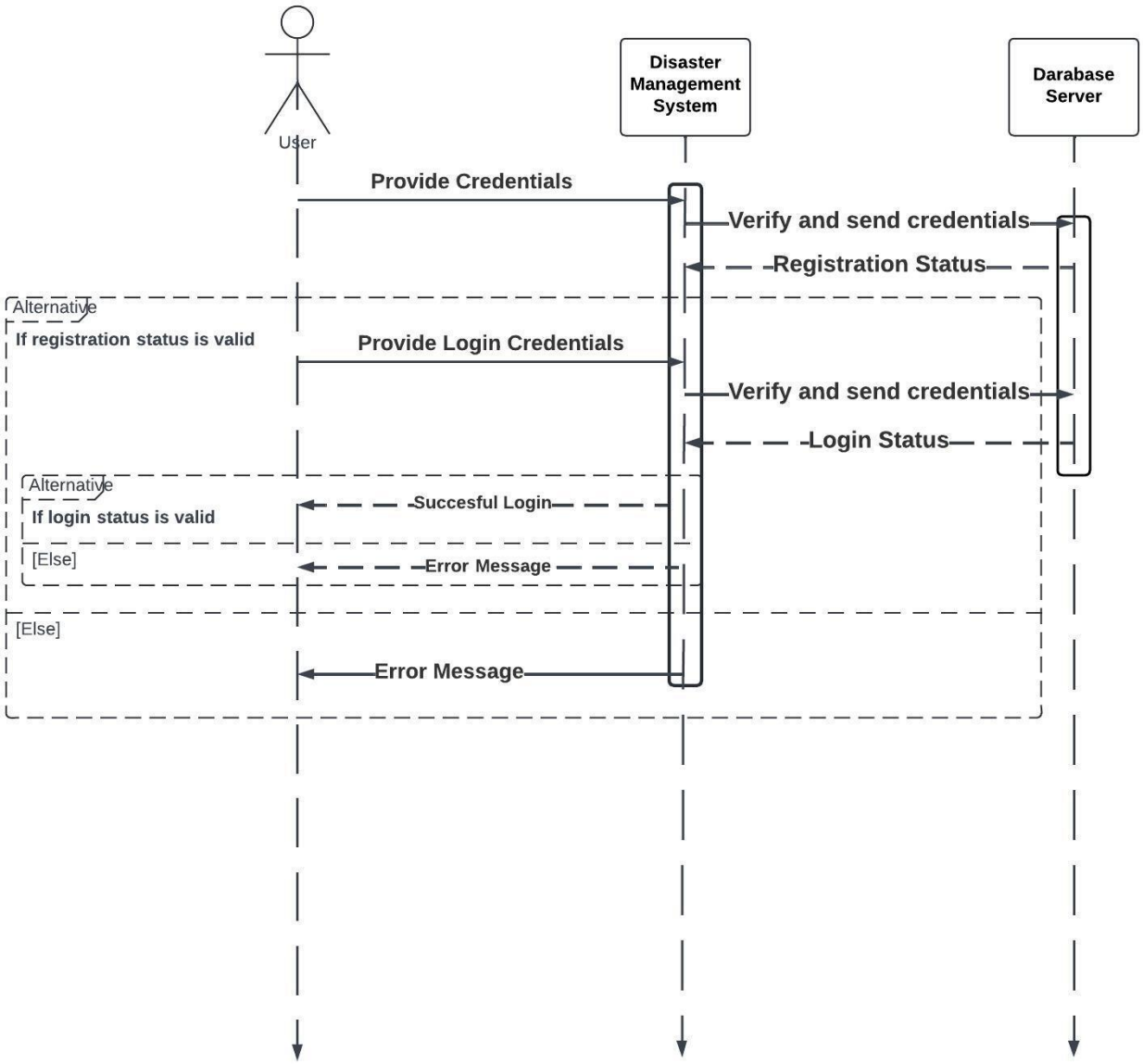


Fig: Sequence Diagram for User Registration and Authentication

Message Flow:

i. **User -> DMS:** The user initiates the registration process by providing their credentials (username, password, etc.) to the DMS.

ii. **DMS -> Database Server:** The DMS validates the information and sends it to the Database Server for secure storage.
iii. **Database Server -> DMS:** The Database Server confirms successful user registration or returns an error message if registration fails (e.g., username already exists).
iv. **DMS -> User:** The DMS informs the user of the registration outcome (success or failure).

- **Success:** If registration is successful, the user can proceed to login else it displays an error message.

v. **User -> DMS:** The user attempts to login by providing their username and password to the DMS.
vi. **DMS -> Database Server:** The DMS retrieves the user's information from the Database Server based on the provided username.
vii. **Database Server -> DMS:** The Database Server sends the user's information (including hashed password) to the DMS.
viii. **DMS -> User:** (**Internal Process** not shown) The DMS compares the entered password with the hashed password stored in the database.
    a. **Match:** If the passwords match, the DMS generates a session token for the user.
ix. **DMS -> User:** The DMS sends a successful login response along with a session token to the user.
x. **User -> System:** The user gains authorized access to the disaster management system functionalities using the received session token.

Note: This is a simplified representation, and additional steps like password hashing and secure communication channels might be implemented for enhanced security.

2. **Report Disaster and Request Help:** This sequence diagram depicts the steps involved in a user reporting a disaster event and requesting assistance during a disaster. We'll see how the user specifies their needs (e.g., medical aid, evacuation) and location, and how the system facilitates this request for help.

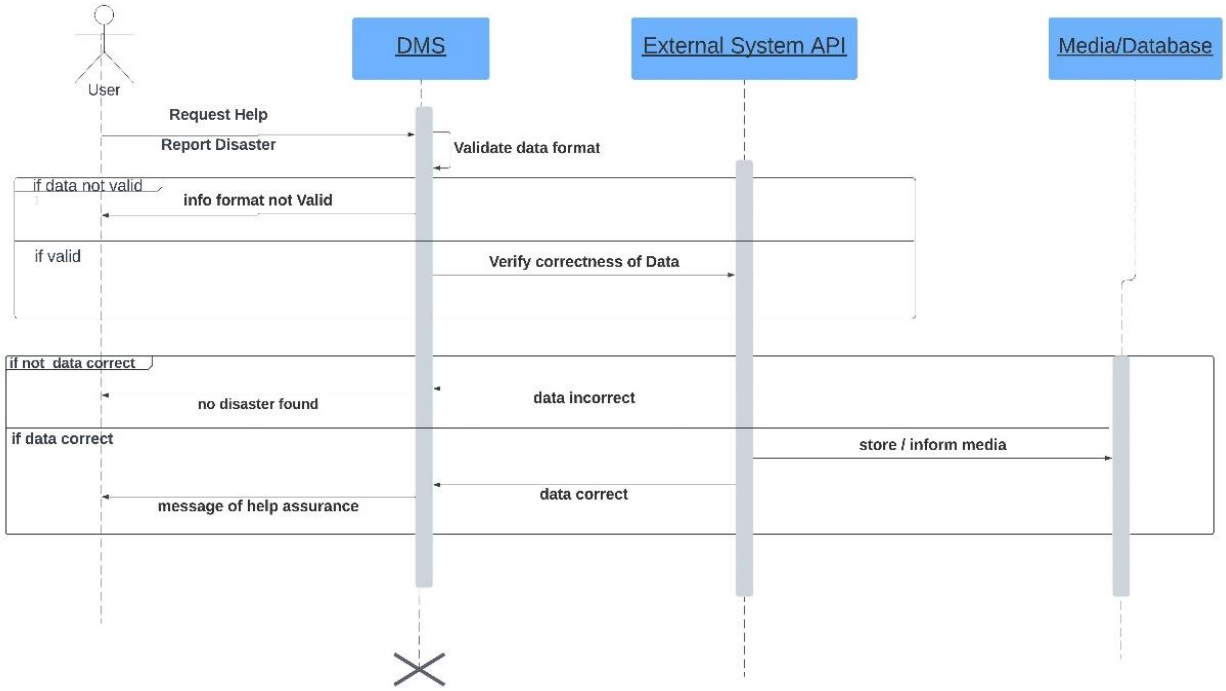


Fig: Sequence Diagram for Report Disaster and Request for Help

Initiating the Request

The sequence begins with the user, who can be anyone needing help or reporting an incident. The user triggers the process by submitting a request for help or reporting a disaster through the DMS application.

Data Validation

The DMS application receives the user's request and validates the data format.

- If the data format is invalid (such as incorrect picture formats, video sizes), the application sends a message back to the user informing them of the issue.
- If the data format is valid, the process proceeds.

Verification via External API

A key step in your DMS is the location verification. The DMS application extracts the reported location from the request data.

The DMS then interacts with the external system API, specifically designed to verify disaster occurrences based on location data.

This external API which is connected to real-time disaster monitoring systems, government etc.

The DMS application sends the reported location to the external API and waits for a response.

Verification Response and Data Storage

6. The external system API verifies the user's reported location against its disaster data.
- If a disaster is confirmed at the reported location, the external API sends a confirmation message back to the DMS application.
- If no disaster is detected, the external API sends a corresponding message indicating no disaster at that location.

7. Based on the response from the external API:
- If a disaster is confirmed, the DMS application stores the validated and verified data (including the confirmed location) for further action. This data might include details like the type of disaster, severity, and potential impact zone.
- If no disaster is confirmed, the DMS application might still store the user's report for further investigation or as a potential early warning sign.

Sending Help Assurance

8. Finally, the DMS application sends a message back to the user acknowledging their request. The message content will vary depending on the verification outcome:
- If a disaster is confirmed, the message might assure the user that help is being directed their way and might include estimated response time or additional instructions based on the disaster type.
- A message thanking the user for reporting and letting them know no disaster was found.

3. Geospatial Data Integration: This sequence diagram illustrates how the Disaster Management System (DMS) retrieves and integrates geospatial data to enhance situational awareness during a disaster.

Objects

- **Disaster Management System (DMS)**
- **Geospatial Data Provider API**: This represents an external service that provides real-time or historical geospatial data based on location.
- **Geospatial Data Analytics Provider API and Evacuation Route Service Provider API (Optional)**: This represents an external service that analyzes geospatial data to identify potential disasters or assess their impact (if triggered by the DMS) and an external service that provides optimal evacuation routes based on real-time conditions (if triggered by the DMS).
- **Mapping Service Provider API**: This represents an external service that displays geospatial data on a map interface.

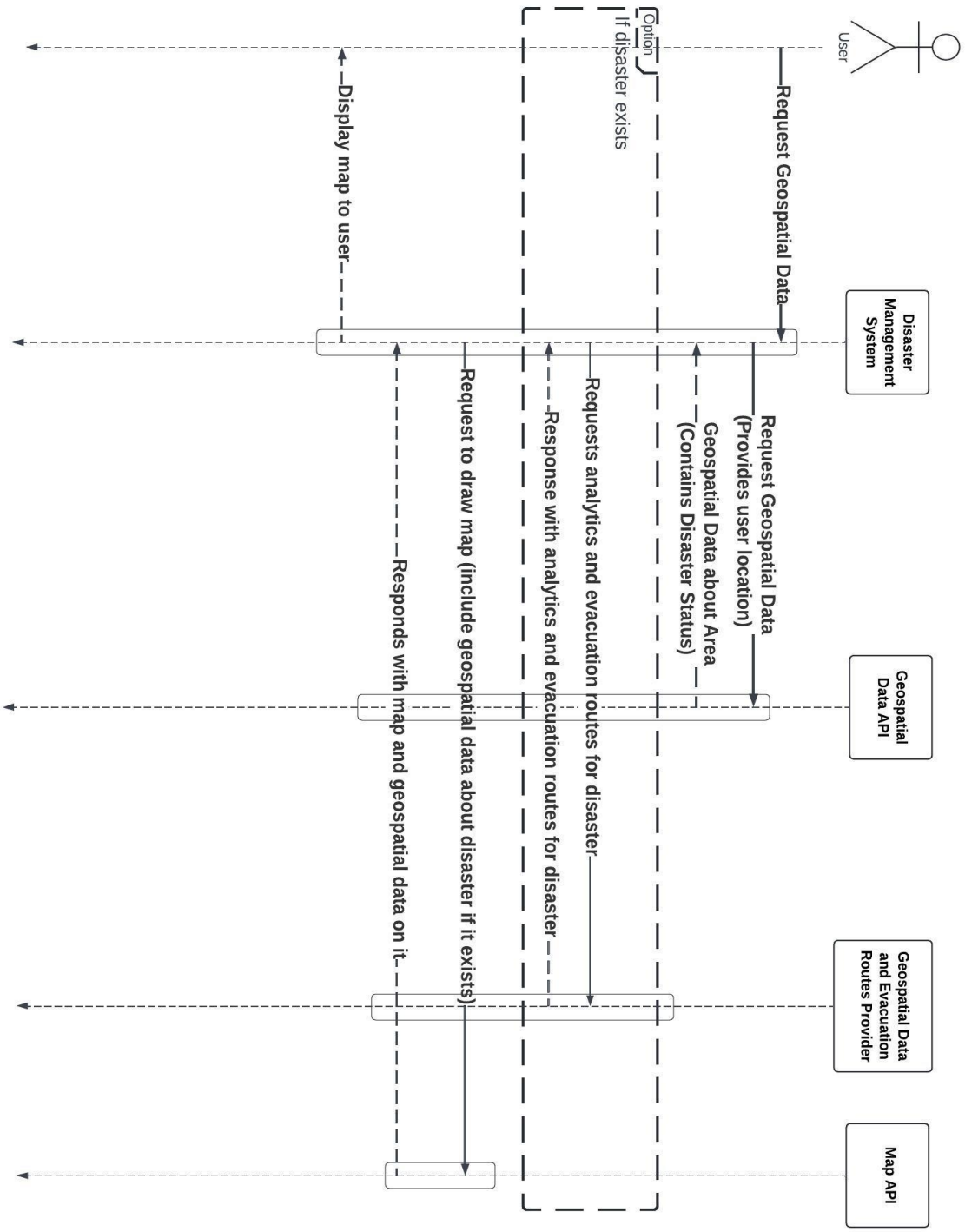


Fig: Sequence Diagram for Geospatial Data Integration

Explanation:

1. **Request Geospatial Data for Current Location**: The user requests geospatial data for their current location from the disaster management system.
2. **Send Request for Geospatial Data**: The disaster management system sends a request to the geospatial data provider API to obtain geospatial data.
3. **Response with Geospatial Data**: The geospatial data provider API responds to the system with geospatial data about the location.
4. **Check for Disaster in Geospatial Data**: The system checks if the received geospatial data indicates the presence of a disaster.
5. **Request Analytics and Evacuation Routes for Disaster**: If a disaster is detected, the system sends a request (the request is sent alongside the disaster coordinates gotten from the geospatial data provider) to the geospatial data analytics provider API for analytics related to the disaster and the system also requests evacuation routes from the evacuation route service provider API.
6. **Response with Disaster Analytics and Evacuation Routes**: The geospatial data analytics provider API responds with analytics information related to the disaster and the evacuation route service provider API responds with data on evacuation routes.
7. **Draw Map with Geospatial Data**: The system uses the mapping service provider API to draw a map displaying the geospatial data, including the location point of the disaster, type of disaster, evacuation routes, and shelter points.
8. **Response with Map**: The mapping service provider API responds with the map drawn with the requested data.
9. **Display Map with Disaster Data and Evacuation Routes**: The system displays the map with disaster data and evacuation routes to the user.

5. **Real-time Alerts and Notifications**: This sequence diagram illustrates the real-time communication flow between the Disaster Management System (DMS), geospatial data providers, and users for disaster alerts and notifications.

Objects:

- Disaster Management System (DMS)
- Geospatial Data Provider: provides geospatial data about ongoing disasters
- Geospatial Data Analytics API: analyses data based on users current coordinates and provides info concerning potential disasters.

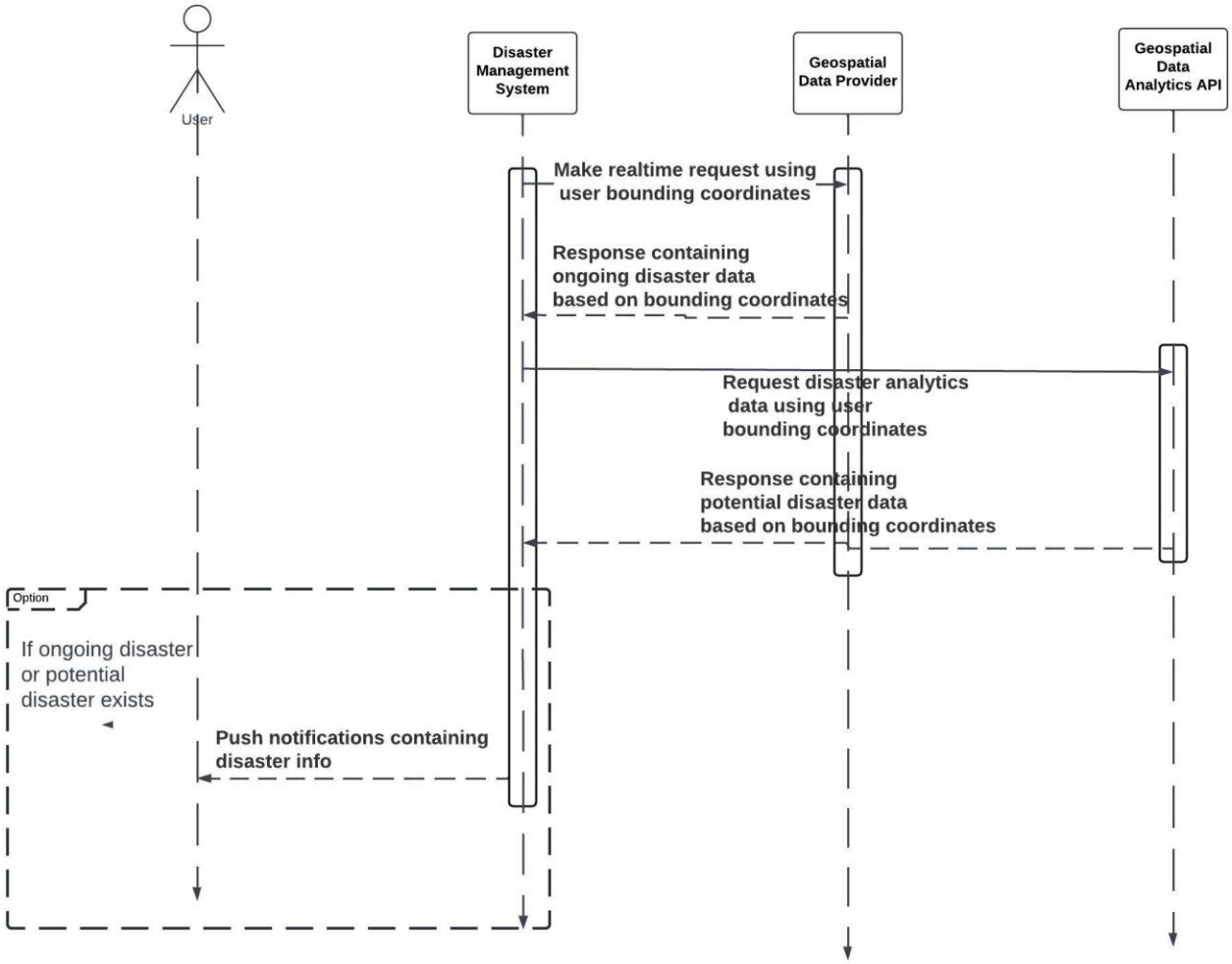


Fig: Sequence Diagram for Real-time Alerts and Notifications

Explanation:

1. **Make Real-Time Request (Bounding User Point Coordinates)**: The Disaster Management System (DMS) initiates a real-time request using the bounding user point coordinates to specify the area of interest.
2. **Request Disaster Data (Geospatial Data Provider)**: the DMS sends a request using socket.IO to the Geospatial Data Provider for disaster data within the specified coordinates.
3. **Response with Disaster Data (Geospatial Data Provider)**: The Geospatial Data Provider responds to the DMS via Socket.IO with real-time disaster data concerning ongoing based on the provided coordinates.
4. **Request Disaster Data from Geospatial Data Analytics API**: the DMS via Socket.IO also sends a request containing user bounding coordinates from geospatial data provider to the Analytics Tools for disaster data within the specified coordinates.

5. **Response with Disaster Data from Geospatial Data Analytics API**: The Analytics Tools respond to the DMS using Socket.IO with real-time disaster data and potential disaster based on the provided coordinates.
6. **Combine and Process Disaster Data**: the DMS combines and processes the received disaster data from both the Geospatial Data Provider and Geospatial Data Analytics Tools.
7. **Send Instant Push Notification (Disaster Information)**: The DMS sends an instant push notification to the Push Notification Service, including information about the disaster.
8. **Deliver Push Notification (Disaster Information)**: The Push Notification Service delivers the push notification containing disaster information to the user's device in real-time.

   This sequence diagram illustrates the flow of interactions between the Disaster Management System, Socket.IO, Geospatial Data Provider, Analytics Tools, Push Notification Service, and the user, enabling real-time alerts and notifications for disaster management.

4. CLASS DIAGRAM

The class diagram of our disaster management system shows the various classes their attributes, methods and relationship to other classes.

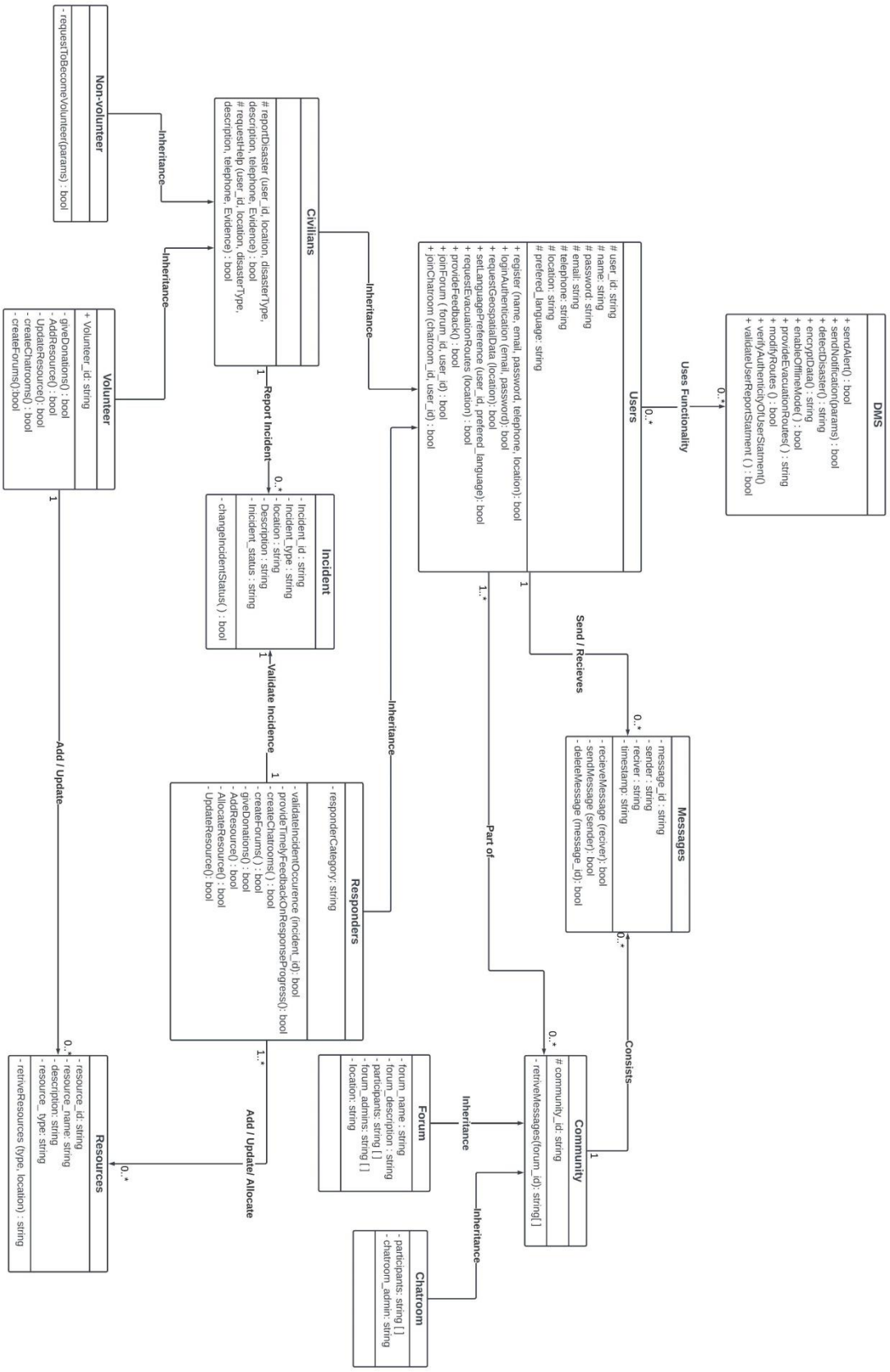


Fig: Class Diagram

Class Diagram Documentation

This class diagram represents a Disaster Management System (DMS) designed to facilitate various functionalities such as alerting, messaging, community management, incident reporting, and resource allocation. Below is a detailed documentation of each class, its attributes, methods, and relationships:

1. DMS (Disaster Management System)

- **Methods:**
  - sendAlert() : bool
  - sendNotification(params) : bool
  - detectDisaster() : string
  - encryptData() : string
  - enableEvacMode() : bool
  - provideEvacuationRoutes() : string
  - modifyRoutes() : bool
  - verifyAuthenticityOfUserStatement() : bool
  - validateUserReportStatement() : bool
- **Relationships:**
  - Provides functionalities to Users (many-to-many relationship).

2. Users

- **Attributes:**
  - user_id : string
  - name : string
  - password : string
  - email : string
  - telephone : string
  - location : string
  - preferred_language : string
- **Methods:**
  - register(name, email, password, telephone, location) : bool
  - loginAuthentication(email, password) : bool
  - requestGeospatialData(location) : bool
  - setLanguagePreference(user_id, preferred_language) : bool
  - requestEvacuationRoutes(location) : bool
  - provideFeedback() : bool
  - joinForum(forum_id, user_id) : bool
  - joinChatroom(chatroom_id, user_id) : bool
- **Relationships:**
  - Inherits to Civilians and Responders.
  - Sends and receives Messages.
  - Part of Community.

3. Messages

- **Attributes:**
  - message_id : string
  - sender : string
  - receiver : string
  - timestamp : string
- **Methods:**
  - receiveMessage(receiver) : bool
  - sendMessage(sender) : bool
  - deleteMessage(message_id) : bool
- **Relationships:**
  - Constitutes a Community (many-to-one relationship).

4. Community

- **Attributes:**
  - community_id : string
- **Methods:**
  - retrieveMessages(forum_id) : string[]
- **Relationships:**
  - Parent class to Forum and Chatroom.

5. Forum

- **Attributes:**
  - forum_name : string
  - forum_description : string
  - participants : string[]
  - forum_admins : string[]
  - location : string
- **Relationships:**
  - Inherits methods and attributes of the community class.

6. Chatroom

- **Attributes:**
  - participants : string[]
  - chatroom_admin : string
- **Relationships:**
  - Inherits methods and attributes of the community class.

7. Civilians (inherits from Users)

- **Methods:**

    - reportDisaster(user_id, location, disasterType, description, telephone, evidence) : bool
    - requestHelp(user_id, location, disasterType, description, telephone, evidence) : bool
- **Relationships:**
    - Can report Incident / request help.
    - Inherits methods and attributes of the users class

8. Responders (inherits from Users)

- **Attributes:**
    - responderCategory : string
- **Methods:**
    - validateIncidentOccurrence(incident_id) : bool
    - provideTimelyFeedbackOnResponseProgress() : bool
    - createChatrooms() : bool
    - createForums() : bool
    - giveDonations() : bool
    - addResource() : bool
    - allocateResource() : bool
    - updateResource() : bool
- **Relationships:**
    - Can validate Incident.
    - Inherits properties, attributes and methods of the user class

9. Incident

- **Attributes:**
    - incident_id : string
    - type : string
    - location : string
    - description : string
    - incident_status : string
- **Methods:**
    - changeIncidentStatus() : bool
- **Relationships:**
    - Many-to-one relationship with Civilians and Responders for reporting and validation respectively.

10. Volunteer (inherits from Civilians)

- **Attributes:**
    - volunteer_id : string
- **Methods:**
    - giveDonations() : bool
    - addResource() : bool

    - updateResource() : bool
    - createChatrooms() : bool
    - createForums() : bool
- **Relationships:**
    - Can request to become a volunteer from responders.

11. Non-volunteer (inherits from Civilians)

- **Methods:**
    - requestToBecomeVolunteer(params) : bool
    - inherits functions and attributes from civilians

12. Resources

- **Attributes:**
    - resource_id : string
    - resource_name : string
    - description : string
    - resource_type : string
    - location : string
- **Methods:**
    - retrieveResources(type, location) : string
- **Relationships:**
    - Can be added, updated, and allocated by Responders and Volunteers.

5. DEPLOYMENT DIAGRAM

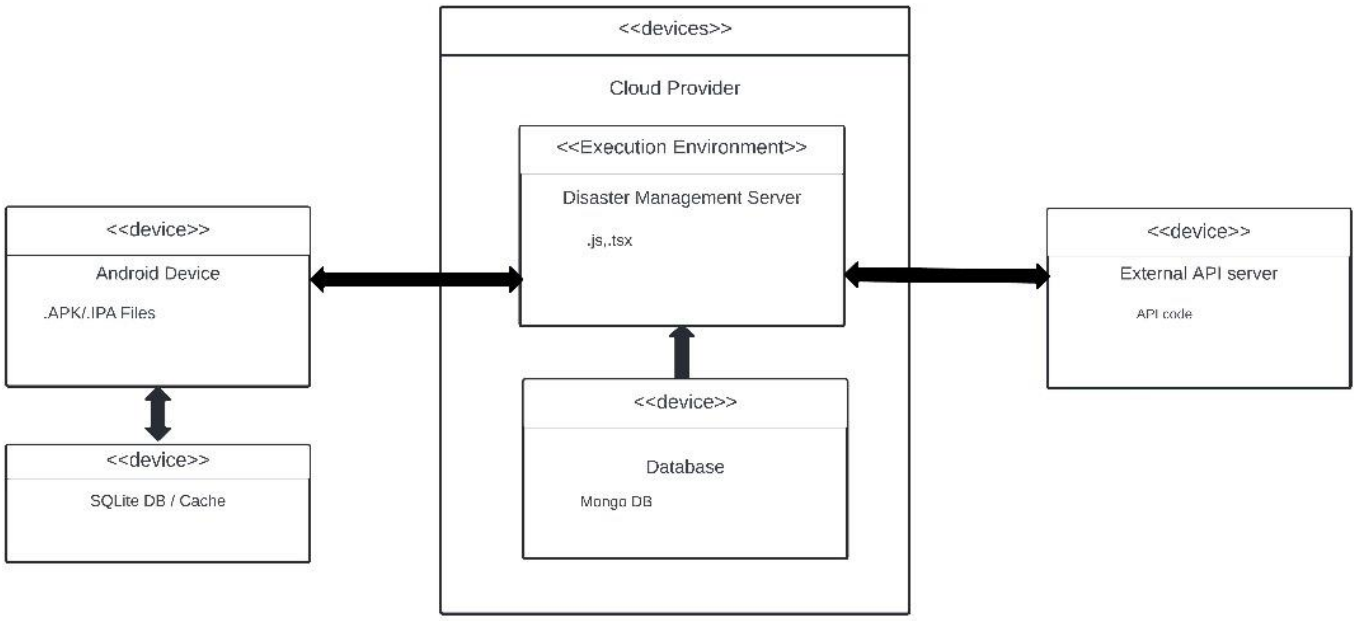
A deployment diagram is a visual representation that showcases how software components are allocated to physical resources within a system. It essentially depicts the physical architecture of your disaster management system, highlighting how the software interacts with the underlying hardware infrastructure.

Elements of a Deployment Diagram:

- **Nodes:** These represent physical elements like servers, workstations, databases, network devices, or any hardware component involved in running the system. They are typically depicted as rectangles.
- **Artifacts:** These represent deployable files or packages that make up the components. Think of them as the actual program code or database schema. They are often depicted as small files within components.

- **Dependencies:** These illustrate the communication channels between components or nodes. They are represented as lines with arrows showing the direction of data flow.

The deployment diagram below illustrates the system's architecture, showcasing how software components interact with the underlying infrastructure.



Deployment diagram for the Disaster Management System

Explanations

Cloud Provider

Our system leverages a cloud provider, denoted by "<<Cloud Provider>>" in the diagram. This cloud provider offers on-demand and scalable infrastructure, eliminating the need for the disaster management system to maintain its own physical servers. This translates to cost-efficiencies and avoids the burden of managing physical infrastructure.

Execution Environment

The execution environment, represented by the blue box labeled "<<Execution Environment>>", which is, in our case **Disaster Management Server**,a core component, the Disaster Management Server, with the artifacts of "js.tsx" in the diagram, houses the disaster management system's backend code. This server executes on the cloud provider's infrastructure and is responsible for the critical functionalities such as:

- Managing user authentication and authorization
- Coordinating communication between various components of the system
- Processing data and responding to disaster events

Database

The system interacts with a database, which resides on the cloud provider's infrastructure and stores critical information for the system's operation, such as:

- Hazard maps
- User information
- Resource inventories
- Emergency response plans

Device and External systems

The diagram depicts various devices that can interact with the disaster management system, categorized as follows:

- Android Device: This represents a smartphone or tablet used by first responders or the public to access the system and receive critical updates during a disaster. The APK/IPA files represent the mobile application that would be installed on these devices.
  This device will also contain an SQLite DB or Cache for fast data retrieval.
- External API server: This represents an interface providing data from external sources, such as weather monitoring stations, geospatial systems etc, which could be valuable for situational awareness during a disaster. The API code likely refers to the code used to develop this external API server.

Communication flow

The diagram also shows clearly the communication between various components and devices. While specific communication protocols are not depicted in the diagram, we can infer general interactions:

- The Android Device communicates with the Disaster Management Server to retrieve critical updates, potentially using a secure web service protocol like HTTPS. It interacts with the SQLite database or Cache for bi-directional flow of data that is instantly and requested often such as the user's Location.
- The Disaster Management Server also depicts the bi-directional communication with the External API server indicating data fetching from external sources and sending data to the systems to receive particular feedbacks.
- The Disaster Management Server interacts with the Database to store and retrieve disaster-related information.

IV. CONCLUSION

In conclusion, our system design and modelling has been instrumental in creating a robust, scalable, and user-centered solution. By grouping functionalities into modules based on similarities in implementation and breaking them down into manageable components, we have enhanced maintainability, scalability, and reusability. Through the use of various diagrams and models, we gained valuable insights into the system's structure, interactions, and behavior, guiding the implementation process effectively. Our approach ensures alignment with stakeholders' needs and facilitates efficient development, seamless integration, and successful deployment of the system. Moving forward, continual refinement and adaptation will be essential to meet evolving requirements and maintain the system's effectiveness in addressing real-world challenges.