REPUBLIQUE DU CAMEROON
PAIX-Travail-Patrie
MINISTRE DE L'ENSEIGNEMENT SUPERIEUR

FACULTE DE L'ENGINERIE ET TECHGNOLOGIE

REPUBLIC OF CAMEROON
Peace-Work-Fatherland
MINISTER OF HIGHER EDUCATION

FACULTY OF ENGINEERING AND TECHNOLOGY

********** UNIVERSITY OF BUEA **********

DEPARTMENT OF COMPUTER ENGINEERING

COURSE: INTERNET PROGRAMMING AND MOBILE APPLICATIONS

COURSE CODE: CEF 440

TASK 4: Database Design and Implementation

Facilitator: Dr. Valery Nkemeni

Academic year 2023/2024

GROUP MEMBERS

S/N	**Name**	Matricule Number
1	MUYANG ROSHELLA MBAMUZANG	FE21A243
2	TAMBONG KERSTEN MELENGFE	FE21A440
3	EBAI ENOWNKU JANE	FE21A176
4	AGBOR NKONGHO KELLY	FE21A126
5	NICCI NSE NCHAMI	FE21A268

Table of Contents

Introduction

Objective

The primary objective of this project is to design and implement a robust database structure for a disaster management system. This system aims to enhance the coordination and efficiency of disaster response efforts by managing and integrating various data entities such as users, news articles, comments, alerts, and incidents. By providing a centralized and well-structured database, the system will ensure seamless data flow and interaction among users, administrators, and emergency responders.

Scope

This database implementation will cover the following aspects:

- **User Management**: Storing and managing user information, including personal details, authentication credentials, and interaction history.
- **News Management**: Enabling the creation, retrieval, updating, and deletion of news articles, along with managing associated comments and likes.
- **Comment Management**: Allowing users to post and manage comments on news articles.
- **Alert Management**: Storing and retrieving alerts related to various incidents or events, ensuring timely notifications to relevant parties.
- **Incident Management**: Recording and managing incidents reported by users or administrators, providing critical information for emergency response.

The database will serve as the backbone of the disaster management system, ensuring data integrity, security, and scalability to accommodate future growth and additional features.

Background

Disaster management systems are crucial in mitigating the impact of disasters by facilitating efficient communication and coordination among various stakeholders. A well-designed database is essential for such systems to function effectively, as it ensures the reliable storage and quick retrieval of critical data. This project aims to leverage modern database technologies to build a comprehensive solution that supports the needs of disaster management, including real-time data processing, mapping of disaster areas, and automated alert systems.

By implementing a structured and scalable database, the disaster management system will be able to provide accurate and timely information to users, emergency responders, and administrators, ultimately improving the overall efficiency and effectiveness of disaster response efforts.

This report will detail the design and implementation process of the database, including the entity definitions, schema designs, and the various operations supported by the system. Through this project, we aim to create a database that not only meets the current requirements but also adapts to future needs and technological advancements.

Requirements

Functional Requirements

1. **User Management**
    - **User Registration**: The system should allow new users to register by providing their name, email, profile picture, phone number, and other necessary details.
    - **User Authentication**: Users should be able to log in using their email and password.
    - **Profile Management**: Users should be able to update their personal information and profile picture.
2. **News Management**
    - **Create News Articles**: Authorized users should be able to create news articles by providing a title, image, description, and other relevant details.
    - **Edit News Articles**: Users should be able to edit the details of existing news articles.
    - **Delete News Articles**: Users should be able to delete news articles.
    - **View News Articles**: Users should be able to view a list of news articles and detailed views of each article.
    - **Comment on News**: Users should be able to post comments on news articles.
    - **Like News Articles**: Users should be able to like news articles.
3. **Comment Management**
    - **Post Comments**: Users should be able to post comments on news articles.
    - **Edit Comments**: Users should be able to edit their own comments.
    - **Delete Comments**: Users should be able to delete their own comments.
4. **Alert Management**
    - **Create Alerts**: Administrators should be able to create alerts by specifying the alert type, location, description, title, and link.
    - **Edit Alerts**: Administrators should be able to edit the details of existing alerts.
    - **Delete Alerts**: Administrators should be able to delete alerts.
    - **View Alerts**: Users should be able to view a list of alerts and detailed views of each alert.
5. **Incident Management**

- **Report Incidents**: Users should be able to report incidents by providing the incident type, name, title, description, and images.
- **Edit Incidents**: Users should be able to edit the details of their reported incidents.
- **Delete Incidents**: Users should be able to delete their reported incidents.
- **View Incidents**: Users should be able to view a list of incidents and detailed views of each incident.

6. **Emergency Responder Management**
    - **Manage Emergency Responders**: The system should allow the management of emergency responder information, including name, contact, and number.
    - **Notification to Responders**: The system should notify emergency responders of reported incidents and alerts.

Non-Functional Requirements

1. **Performance**
    - The system should be able to handle multiple concurrent users without significant degradation in performance.
    - Data retrieval operations should be optimized to ensure quick access to information.
2. **Security**
    - User data should be stored securely, with proper encryption and access controls.
    - The system should implement robust authentication and authorization mechanisms to prevent unauthorized access.
3. **Scalability**
    - The database should be designed to scale horizontally, allowing for the addition of more servers to handle increased load.
    - The system should support both vertical and horizontal scaling to accommodate future growth.
4. **Reliability**
    - The system should ensure high availability, with minimal downtime.
    - Regular backups should be performed to prevent data loss.
5. **Usability**
    - The user interface should be intuitive and user-friendly, ensuring that users can easily navigate and use the system.
    - The system should provide helpful error messages and guidance to users encountering issues.

6. **Maintainability**
    - The system should be designed with modularity in mind, allowing for easy updates and maintenance.
    - Comprehensive documentation should be provided to support future development and maintenance efforts.

These requirements outline the key functionalities and qualities that the disaster management system's database must support, ensuring it meets the needs of its users while maintaining high standards of performance, security, and usability.

Database Design

Entity-Relationship Diagram (ERD)

The ERD represents the data model for the disaster management system, showing how different entities are related to each other. Below is an overview of the key entities and their relationships:

1. **User**
    - Attributes: `name`, `email`, `profilePicture`, `uid`, `createdAt`, `phone`
    - Relationships: One-to-Many with `News`, `Comments`, `Incidents`
2. **Admin**
    - Attributes: `name`, `password`
    - Relationships: None specified
3. **Emergency Responder**
    - Attributes: `name`, `contact`, `number`
    - Relationships: None specified
4. **News**
    - Attributes: `title`, `image`, `description`, `uid` (foreign key to User), `comments` (array of Comment references), `likes`
    - Relationships: Belongs to `User`, has many `Comments`
5. **Comment**
    - Attributes: `text`, `uid` (foreign key to User), `newsId` (foreign key to News)
    - Relationships: Belongs to `User` and `News`
6. **Alert**
    - Attributes: `alertType`, `location`, `description`, `title`, `link`
    - Relationships: None specified
7. **Incident**
    - Attributes: `type`, `name`, `title`, `description`, `image`, `uid` (foreign key to User)
    - Relationships: Belongs to `User`

The following is a description of each entity and its role within the database:

Entity Definitions

User

The `User` entity represents individuals who interact with the system. Each user has unique attributes and can perform various actions within the system.

- **Attributes**:
  - `name`: The name of the user.
  - `email`: The email address of the user.
  - `profilePicture`: The URL to the user's profile picture.
  - `uid`: A unique identifier for the user.
  - `createdAt`: The date and time when the user account was created.
  - `phone`: The phone number of the user.
- **Relationships**:
  - One user can create multiple news articles (`News`).
  - One user can post multiple comments (`Comment`).
  - One user can report multiple incidents (`Incident`).

Admin

The `Admin` entity represents administrators of the system who have special privileges.

- **Attributes**:
  - `name`: The name of the admin.
  - `password`: The password for the admin account.
- **Relationships**:
  - None specified.

Emergency Responder

The `Emergency Responder` entity represents professionals who respond to emergencies.

- **Attributes**:
  - `name`: The name of the responder.
  - `contact`: Contact details for the responder.
  - `number`: A unique identifier or number for the responder.
- **Relationships**:
  - None specified.

News

The `News` entity represents news articles posted on the platform.

- **Attributes**:

    - `title`: The title of the news article.
    - `image`: The URL to an image associated with the news article.
    - `description`: The description or content of the news article.
    - `uid`: The unique identifier of the user who posted the news.
    - `comments`: An array of references to comments related to the news.
    - `likes`: The number of likes the news article has received.
- **Relationships**:
    - One news article is created by one user (`User`).
    - One news article can have multiple comments (`Comment`).

Comment

The `Comment` entity represents comments posted on news articles.

- **Attributes**:
    - `text`: The text content of the comment.
    - `uid`: The unique identifier of the user who posted the comment.
    - `newsId`: The unique identifier of the news article to which the comment belongs.
- **Relationships**:
    - One comment is created by one user (`User`).
    - One comment belongs to one news article (`News`).

Alert

The `Alert` entity represents alerts generated within the system.

- **Attributes**:
    - `alertType`: The type of alert.
    - `location`: The location related to the alert.
    - `description`: The description of the alert.
    - `title`: The title of the alert.
    - `link`: A link to more information about the alert.
- **Relationships**:
    - None specified.

Incident

The `Incident` entity represents incidents reported within the system.

- **Attributes**:
    - `type`: The type of incident.
    - `name`: The name of the incident.
    - `title`: The title of the incident.
    - `description`: The description of the incident.
    - `image`: The URL to an image related to the incident.
    - `uid`: The unique identifier of the user who reported the incident.

- **Relationships**:
    - One incident is reported by one user (`User`).

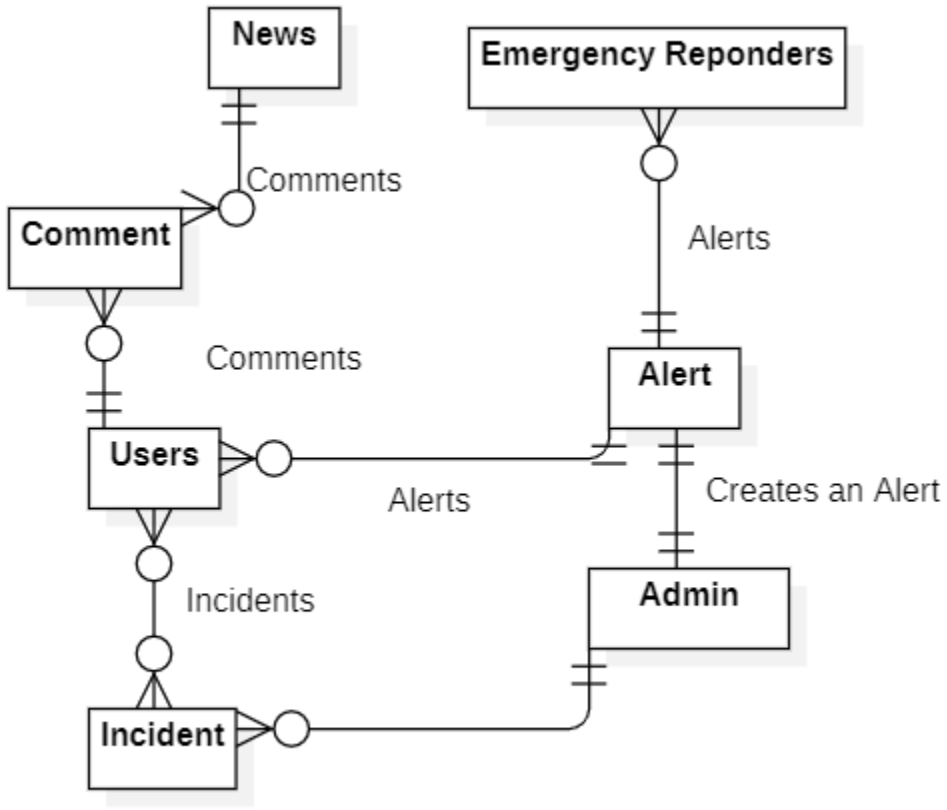
Schema Definitions

Based on the entity definitions, the following schema definitions can be implemented using Mongoose for a Node.js application:

Database Design

Entity-Relationship Diagram (ERD)

The ERD represents the data model for the disaster management system, showing how different entities are related to each other. Below is an overview of the key entities and their relationships:



The following is a description of each entity and its role within the database:

Entity Definitions

User

The `User` entity represents individuals who interact with the system. Each user has unique attributes and can perform various actions within the system.

- **Attributes**:
  - `name`: The name of the user.
  - `email`: The email address of the user.
  - `profilePicture`: The URL to the user's profile picture.
  - `uid`: A unique identifier for the user.
  - `createdAt`: The date and time when the user account was created.
  - `phone`: The phone number of the user.
- **Relationships**:
  - One user can create multiple news articles (`News`).
  - One user can post multiple comments (`Comment`).
  - One user can report multiple incidents (`Incident`).

Admin

The `Admin` entity represents administrators of the system who have special privileges.

- **Attributes**:
  - `name`: The name of the admin.
  - `password`: The password for the admin account.
- **Relationships**:
  - None specified.

Emergency Responder

The `Emergency Responder` entity represents professionals who respond to emergencies.

- **Attributes**:
  - `name`: The name of the responder.
  - `contact`: Contact details for the responder.
  - `number`: A unique identifier or number for the responder.
- **Relationships**:
  - None specified.

News

The `News` entity represents news articles posted on the platform.

- **Attributes**:
  - `title`: The title of the news article.
  - `image`: The URL to an image associated with the news article.
  - `description`: The description or content of the news article.
  - `uid`: The unique identifier of the user who posted the news.
  - `comments`: An array of references to comments related to the news.

    - `likes`: The number of likes the news article has received.
- **Relationships**:
    - One news article is created by one user (`User`).
    - One news article can have multiple comments (`Comment`).

Comment

The `Comment` entity represents comments posted on news articles.

- **Attributes**:
    - `text`: The text content of the comment.
    - `uid`: The unique identifier of the user who posted the comment.
    - `newsId`: The unique identifier of the news article to which the comment belongs.
- **Relationships**:
    - One comment is created by one user (`User`).
    - One comment belongs to one news article (`News`).

Alert

The `Alert` entity represents alerts generated within the system.

- **Attributes**:
    - `alertType`: The type of alert.
    - `location`: The location related to the alert.
    - `description`: The description of the alert.
    - `title`: The title of the alert.
    - `link`: A link to more information about the alert.
- **Relationships**:
    - None specified.

Incident

The `Incident` entity represents incidents reported within the system.

- **Attributes**:
    - `type`: The type of incident.
    - `name`: The name of the incident.
    - `title`: The title of the incident.
    - `description`: The description of the incident.
    - `image`: The URL to an image related to the incident.
    - `uid`: The unique identifier of the user who reported the incident.
- **Relationships**:
    - One incident is reported by one user (`User`).

Schema Definitions

Based on the entity definitions, the following schema definitions can be implemented using Mongoose for a Node.js application:

Admin.js

```
const mongoose = require('mongoose');

const AdminSchema = new mongoose.Schema({
  name: { type: String, required: true },
  password: { type: String, required: true }
});

module.exports = mongoose.model('Admin', AdminSchema);
```

Alert.js

```
const mongoose = require('mongoose');

const AlertSchema = new mongoose.Schema({
  alertType: { type: String, required: true },
  location: { type: String, required: true },
  description: { type: String, required: true },
  title: { type: String, required: true },
  link: { type: String }
});

module.exports = mongoose.model('Alert', AlertSchema);
```

Comment.js

```
const mongoose = require('mongoose');

const CommentSchema = new mongoose.Schema({
  text: { type: String, required: true },
  uid: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  newsId: { type: mongoose.Schema.Types.ObjectId, ref: 'News', required: true }
});

module.exports = mongoose.model('Comment', CommentSchema);
```

Emergency Responders

```
const mongoose = require('mongoose');

const EmergencyResponderSchema = new mongoose.Schema({
  name: { type: String, required: true },
  contact: { type: String, required: true },
  number: { type: String, required: true }
});

module.exports = mongoose.model('EmergencyResponder', EmergencyResponderSchema);
```

Incident.js

```
const mongoose = require('mongoose');

const IncidentSchema = new mongoose.Schema({
  type: { type: String, required: true },
  name: { type: String, required: true },
  title: { type: String, required: true },
  description: { type: String, required: true },
  image: { type: String },
  uid: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true }
});

module.exports = mongoose.model('Incident', IncidentSchema);
```

News.js

```
const mongoose = require('mongoose');

const NewsSchema = new mongoose.Schema({
  title: { type: String, required: true },
  image: { type: String },
  description: { type: String },
  uid: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  comments: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Comment' }],
  likes: { type: Number, default: 0 }
});

module.exports = mongoose.model('News', NewsSchema);
```

User.js

```
const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  profilePicture: { type: String },
  uid: { type: String, required: true, unique: true },
  createdAt: { type: Date, default: Date.now },
  phone: { type: String }
});

module.exports = mongoose.model('User', UserSchema);
```

Implementation

Technology Stack

For this project, the following technology stack has been chosen:

- **Node.js**: A JavaScript runtime used to build the backend server.
- **Express.js**: A web application framework for Node.js, used to create APIs.
- **MongoDB**: A NoSQL database used to store data.
- **Mongoose**: An ODM (Object Data Modeling) library for MongoDB and Node.js.
- **JWT (JSON Web Token)**: Used for user authentication and authorization.
- **Bcrypt**: A library to hash passwords.
- **Postman**: For testing the APIs.

Security Considerations

Authentication

1. **User Authentication**: Implement user authentication using JSON Web Tokens (JWT). JWT allows secure transmission of information between parties as a JSON object. The process involves:
    - **Registration**: When a user registers, their password is hashed using bcrypt and stored in the database.
    - **Login**: Upon login, the user provides their email and password. The system verifies the credentials, and if they are correct, generates a JWT token. This token is sent back to the user and stored on the client-side (typically in local storage or cookies).
    - **Token Validation**: For each subsequent request, the client includes the JWT token in the header. The server verifies the token to authenticate the user.

Authorization

1. **Role-Based Access Control (RBAC)**: Implement RBAC to control access based on the user's role (e.g., admin, user, emergency responder). This involves:
    - Assigning roles to users upon registration or by an admin.
    - Checking the user's role before allowing access to certain routes or functionalities

Data Validation

1. **Input Validation**: Use libraries like Joi or express-validator to validate user inputs, ensuring data integrity and preventing malicious inputs.

Encryption

1. **Password Encryption**: Use bcrypt to hash passwords before storing them in the database, ensuring they are not stored in plaintext.

Data Privacy

1. **Data Encryption**: Encrypt sensitive data stored in the database and ensure encrypted communication between the server and clients using HTTPS.

2. **Access Control**: Implement strict access controls and ensure that only authorized personnel have access to sensitive information.

Conclusion

Throughout this report, we have explored the essential aspects of designing and implementing a database for a disaster management system. The project aims to create a robust, secure, and scalable database to support various functionalities such as user management, news and comment management, alert handling, and incident reporting.

Key Takeaways

1. **Comprehensive Database Design**: We developed a detailed Entity-Relationship Diagram (ERD) that clearly illustrates the relationships between various entities such as users, news articles, comments, alerts, and incidents. This design ensures that the database can handle complex data interactions efficiently.
2. **Schema Implementation**: Using Node.js, Express.js, and MongoDB with Mongoose, we implemented the database schemas for all entities. Each schema is carefully designed to include necessary fields and relationships, ensuring data integrity and consistency.
3. **CRUD Operations**: We demonstrated how to create, read, update, and delete (CRUD) operations for the entities, providing examples of API endpoints for user management. These operations form the backbone of the system, allowing for efficient data manipulation.
4. **Security Considerations**: Security is paramount in our design. We implemented user authentication using JWT, role-based access control (RBAC), password hashing with bcrypt, and input validation to ensure data security and prevent unauthorized access.
5. **Testing and Deployment**: We highlighted the importance of unit and integration testing to ensure system reliability and correctness. The use of Docker for containerization and CI/CD pipelines for automated deployment were also discussed to streamline the deployment process.

Future Work

While the current implementation lays a solid foundation, future work could focus on:

- **Enhancing Security**: Implementing additional security measures such as multi-factor authentication (MFA) and continuous monitoring for suspicious activities.
- **Performance Optimization**: Conducting performance testing and optimizing database queries to handle large volumes of data efficiently.
- **Feature Expansion**: Adding new features such as real-time notifications, geolocation services for incident reporting, and integration with third-party services for enhanced functionality.
- **User Interface Improvements**: Developing a more user-friendly and intuitive interface for both web and mobile applications to improve user experience.

By adhering to best practices in database design, security, and testing, this project provides a robust platform for disaster management. The system is designed to be adaptable and scalable, capable of meeting the evolving needs of its users. Moving forward, continued refinement and expansion of the system will further enhance its effectiveness in managing and responding to disasters.