Republic of Cameroon
******
Peace - Work - Fatherland
*****
Minister of Higher Education
*****
University of Buea

Republique du Cameroun
******
Paix - Travail - Patrie
*****
Ministere de l'enseignement Superieur
*****
Universite de Buea

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

MOBILE APPLICATION DEVELOPMENT

CEF 440

TASK 6 REPORT:
ROAD SIGN AND ROAD STATE NOTIFICATION APP
DATABASE DESIGN AND IMPLEMENTATION

Presented by:

GROUP 8

Nº	NAMES	MATRICULES
1	FOWEDLUNG ATSAFAC AGAFINA	FE21A196
2	KAMDEM KAMGAING GILLES CHRISTIAN	FE21A209
3	NEGUE KWAHAM MAEL GRACE	FE21A252
4	NGUEPI GNETEDEM PATERSON	FE21A264
5	NOUPOUWO DONGMO STEPHANE MERCI	FE21A283

Course Instructor: Dr Valery Nkemeni

June 2024

TABLE OF CONTENTS

I- INTRODUCTION

The database design and implementation for a road sign and road state notification app is a critical phase for creating an effective, reliable, and user-friendly application aimed at enhancing road safety and improving the driving experience by providing real-time data. Considering previous tasks such as Requirement Analysis and System Modelling, which give us a view on the type of data to be stored and how it should be structured, helps us identify and define crucial considerations for this phase.

This report covers the entire process of designing and implementing the database for our road sign and road state notification app. Emphasis is placed on ensuring database reliability, performance, and security considerations.

By meticulously addressing these aspects, the database can efficiently handle large volumes of data, provide quick data retrieval, and maintain the integrity and confidentiality of the information. This ensures that users receive accurate and timely notifications about road signs, traffic conditions, and road statuses, thereby contributing to safer and more informed driving decisions.

II- DESIGN PROCESS

The design process for a database is a methodical and detailed phase that ensures the database robustness, efficiency, and security. This phase builds on the insights gained from the Requirement Analysis and System Modelling phases of our system, ensuring that all necessary data is accurately captured and structured. Below are the detailed steps involved in the design process.

II.1 Data Elements

- **Road Signs:** Describes information about a road signs such as name, description and image.
- **Road States:** Describes information about a road states such as name, description and image.
- **Users:** Describes information about users such as ID, name, contact details and, road signs and states preferences.
- **Saved location:** Describes information about a user saved location such as name, description and image.

II.2 Functional Requirements

a) User Account Management

It is critical for allowing users to register, manage their profiles, and access the application's features securely.

b) Road Sign Notification

This feature ensures that users are informed about road signs and state along their route.

c) Road states Updates

This feature provides users with real-time information about the current conditions of the roads they are traveling on.

d) Location Management

It is crucial for accurately determining and managing the geographic data required for providing notifications and updates.

II.3 Non-Functional Requirements

- The system must ensure high performance for quick data retrieval to provide users with real-time information.
- It should be scalable to handle growing data and user base, ensuring that the app remains responsive and efficient.
- Strong security measures are vital to safeguard user data, ensuring confidentiality and authorized access.
- The system must maintain high availability, ensuring uninterrupted access to the app for users without significant downtime.

III- CONCEPTUAL DESIGN

Conceptual Design creates a high-level representation of the data model, focusing on primary entities and relationships. It also allows stakeholders to grasp the essence of the system without delving into technical implementation details.

III.1 Key Entities and Relationships

a) Users: Following are user attributes stated with data requirement specified;

- **Data Types**
  - User ID: String
  - Name: String
  - Email Address: String
  - Password: String
  - Phone Number: String

- **Constraints**
  - User ID: Primary key constraint to ensure uniqueness.
  - All attributes should not be NULL.

- **Relationship**
  - A user can set zero or many preferences.
  - A user can save zero or many locations.

b) Road Sign: Following are road sign attributes stated with data requirement specified;

- **Data Types**
  - Sign ID: String
  - Name: String
  - Description: String
  - Image: String

- **Constraints**
  - Sign ID: Primary key constraint to ensure uniqueness.
  - All attributes should not be NULL.

- **Relationship**
  - A road sign has zero or many roads sign position.
  - A road sign is defined by o or many preferences.

c) Road State: Following are road state attributes stated with data requirement specified;

- **Data Types**
  - State ID: String
  - Name: String
  - Description: String
  - Image: String

- **Constraints**
  - State ID: Primary key constraint to ensure uniqueness.
  - All attributes should not be NULL.

- **Relationship**
  - A road state has zero or many roads state position.
  - A road state is defined by o or many preferences.

d) Preferences: Following are user notification preferences attributes stated with data requirement specified;

- **Data Types**
  - Preference ID: String
  - User ID: String
  - Sign ID: String
  - State ID: String

- **Constraints**
  - Preference ID: Primary key constraint to ensure uniqueness.
  - User ID, Sign ID and State ID: Foreign Key constraint.

- **Relationship**
  - A preference is set by one and only one user.
  - A preference defines one and only one road sign.
  - A preference defines one and only one road state.

e) Saved Location: Following are user saved location attributes stated with data requirement specified;

- **Data Types**
  - Location ID: String
  - User ID: String
  - Name: String
  - Longitude: String
  - Latitude: String

- **Constraints**
  - Location ID: Primary key constraint to ensure uniqueness.
  - User ID: Foreign Key constraint.
  - All attributes should not be NULL.

- **Relationship**
  - A location can be saved by one or only one user.

f) Road Sign Position: Following are road sign position attributes stated with data requirement specified;

- **Data Types**
    - Sign Position ID: String
    - Sign ID: String
    - Longitude: String
    - Latitude: String

- **Constraints**
    - Sign Position ID: Primary key constraint to ensure uniqueness.
    - Sign ID: Foreign Key constraint.
    - All attributes should not be NULL.

- **Relationship**
    - A road sign position is representing by one and only one road sign.

g) **Road State Position**: Following are road state position attributes stated with data requirement specified;

- **Data Types**
    - State Position ID: String
    - State ID: String
    - Longitude: String
    - Latitude: String

- **Constraints**
    - State Position ID: Primary key constraint to ensure uniqueness.
    - State ID: Foreign Key constraint.
    - All attributes should not be NULL.

- **Relationship**
    - A road sign position is representing by one and only one road sign.

III.2 Entity Relationship (ER) Diagram

The Entity-Relationship (ER) Diagram visually represents data entities, attributes, and relationships, serving as a database structure blueprint.

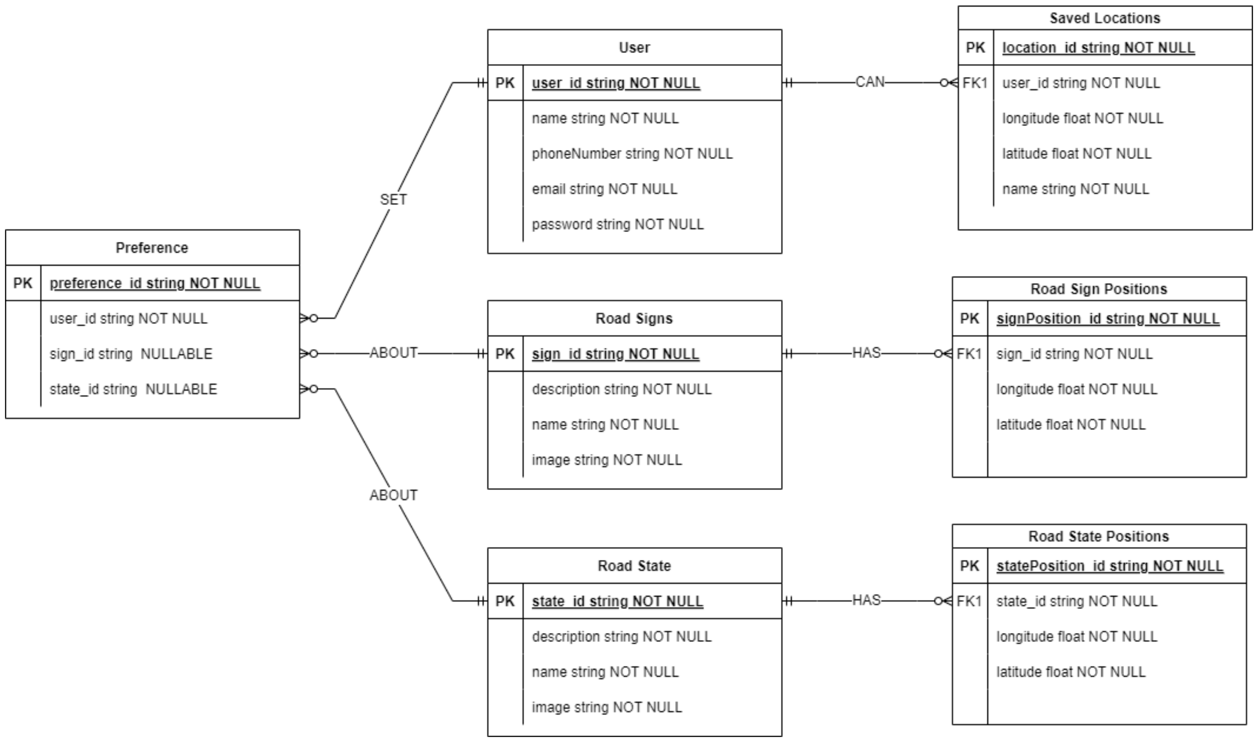
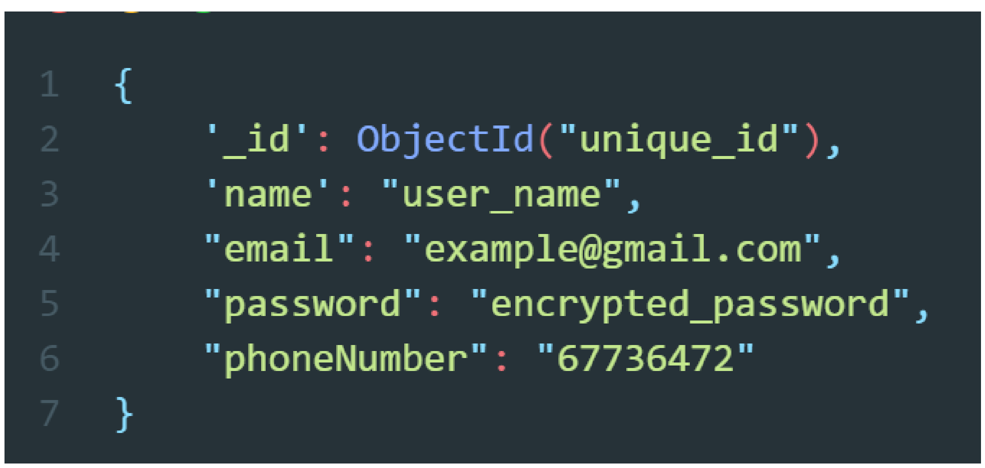


Figure 1: Entity Relationship Diagram

III.3 NoSql Document generation

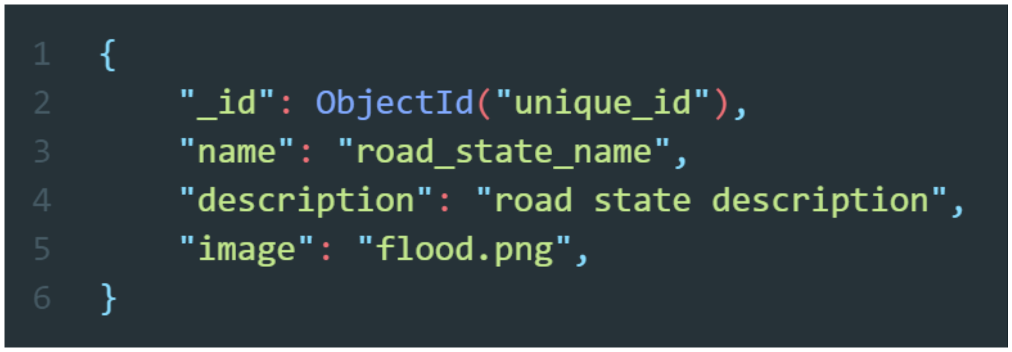
The entities from the ER diagrams are conceptualized into documents which will represent records in each collection. The different documents for the above ER diagram include:

- ***User:***



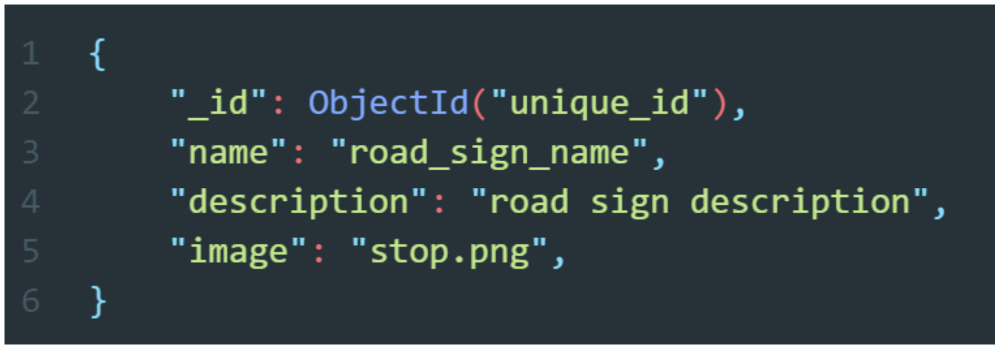
```
{
      '_id': ObjectId("unique_id"),
      'name': "user_name",
      "email": "example@gmail.com",
      "password": "encrypted_password",
      "phoneNumber": "67736472"
}
```

- ***Road State:***



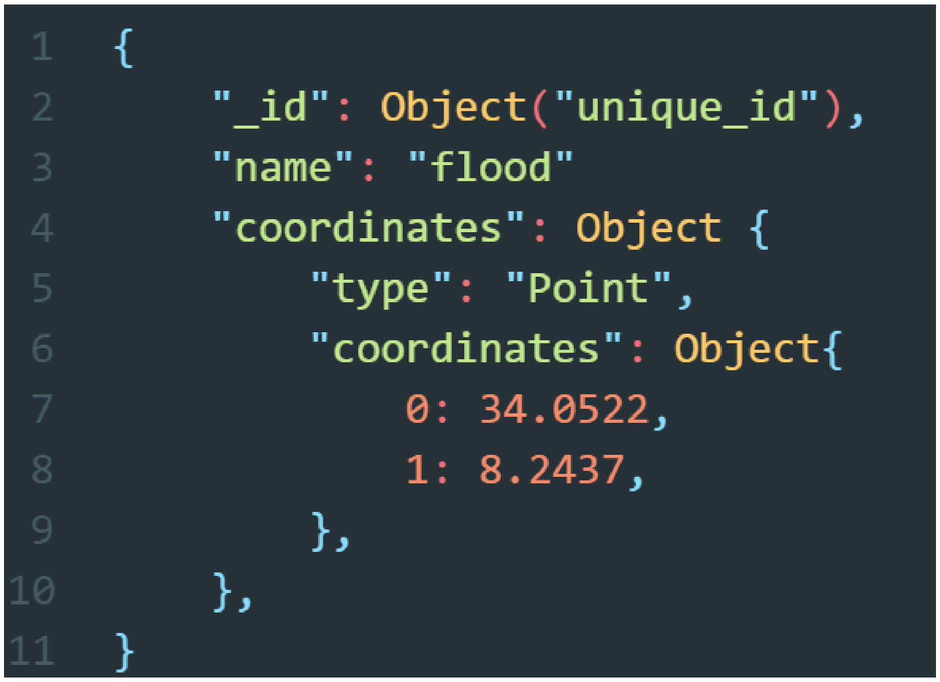
```
{
      "_id": ObjectId("unique_id"),
      "name": "road_state_name",
      "description": "road state description",
      "image": "flood.png",
}
```

- ***Road Sign:***



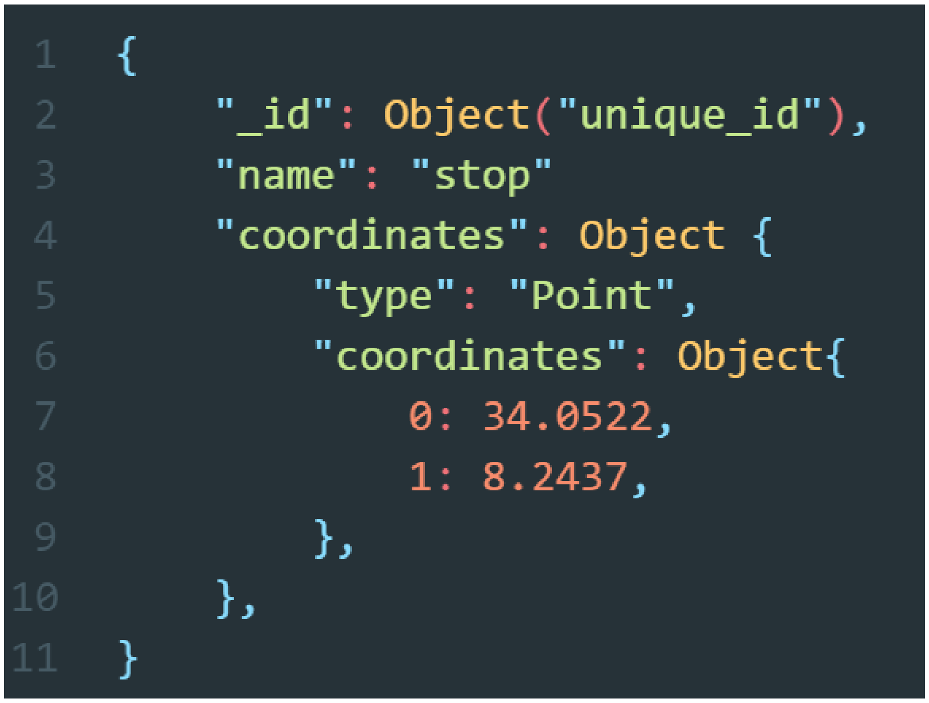
```
{
      "_id": ObjectId("unique_id"),
      "name": "road_sign_name",
      "description": "road sign description",
      "image": "stop.png",
}
```

- ***Road State Marker:***



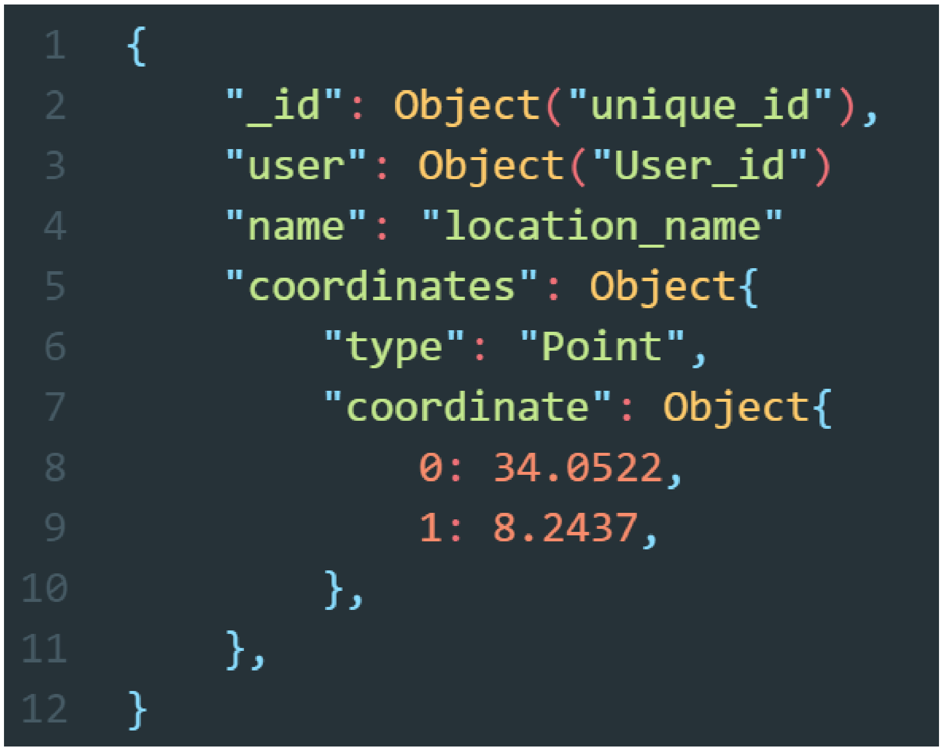
```
{
        "_id": Object("unique_id"),
        "name": "flood"
        "coordinates": Object {
            "type": "Point",
            "coordinates": Object{
                0: 34.0522,
                1: 8.2437,
            },
        },
}
```

- ***Road Sign Marker:***



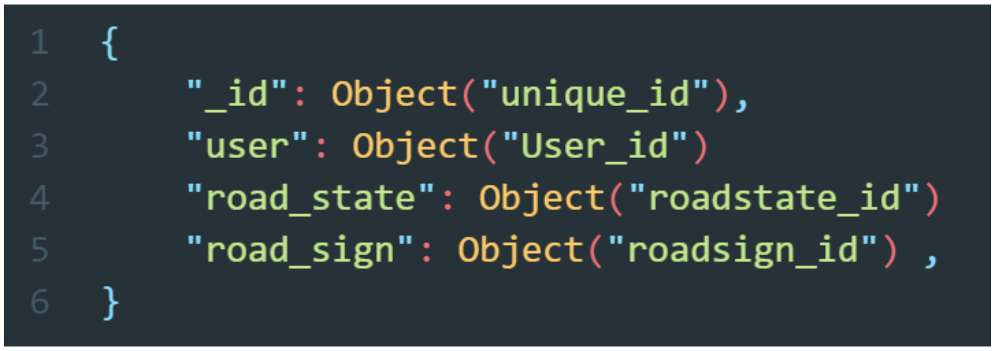
```
{
        "_id": Object("unique_id"),
        "name": "stop"
        "coordinates": Object {
            "type": "Point",
            "coordinates": Object{
                0: 34.0522,
                1: 8.2437,
            },
        },
}
```

- ***Saved Locations:***



```
{
     "_id": Object("unique_id"),
     "user": Object("User_id")
     "name": "location_name"
     "coordinates": Object{
         "type": "Point",
         "coordinate": Object{
             0: 34.0522,
             1: 8.2437,
         },
     },
}
```

- ***Preference:***



```
{
     "_id": Object("unique_id"),
     "user": Object("User_id")
     "road_state": Object("roadstate_id")
     "road_sign": Object("roadsign_id") ,
}
```

IV- IMPLEMENTATION

This section describes the path through the implementation of the NoSql database designed above. This implementation was done following the steps;

IV.1 Stating Libraries and Tools Used

For the purpose of this project, some tools and libraries were used which are described as follows;

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

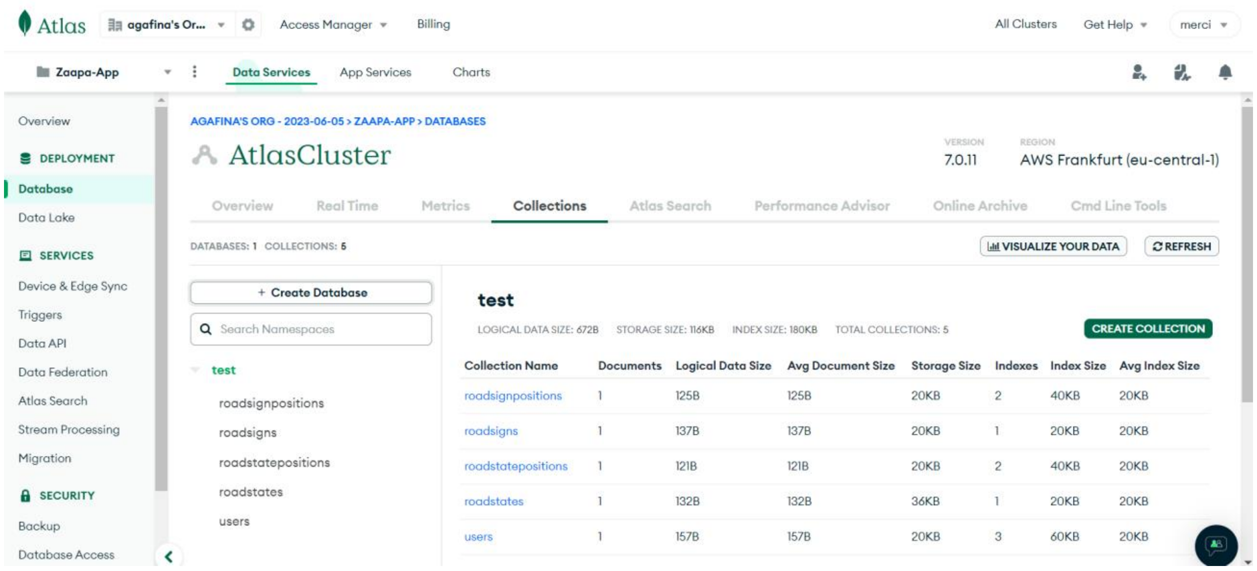
- ***Nodemon:*** A library which helps in automatically running Node server applications and watching over changes that may occur in the project structure.

- ***Mongoose:*** An ORM(Object Relational Mapper) which serves as a powerful tool for developers working with MongoDB in Node.js, providing a structured and organized way to define, manage, and interact with data. Its schema-based approach, combined with features like middleware, validation, and population, makes it a robust choice for building applications that require a consistent and efficient data layer.

- ***.env:*** A simple text file used to store environment variables for a project. These variables can include configuration settings, database credentials, API keys, and other sensitive information that should not be hard-coded in your application's source code. Using a .env file helps keep sensitive data secure and makes it easier to manage different configurations for development, testing, and production environments.

IV.2 Creating Database

The NoSql database was created on MongoDB and named *Zaapa-App* as shown on the figure below



AGAFINA'S ORG - 2023-06-05 > ZAAPA-APP > DATABASES

AtlasCluster

VERSION 7.0.11 REGION AWS Frankfurt (eu-central-1)

DATABASES: 1 COLLECTIONS: 5

test
- roadsignpositions
- roadsigns
- roadstatepositions
- roadstates
- users

test

LOGICAL DATA SIZE: 672B STORAGE SIZE: 116KB INDEX SIZE: 180KB TOTAL COLLECTIONS: 5

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
roadsignpositions	1	125B	125B	20KB	2	40KB	20KB
roadsigns	1	137B	137B	20KB	1	20KB	20KB
roadstatepositions	1	121B	121B	20KB	2	40KB	20KB
roadstates	1	132B	132B	36KB	1	20KB	20KB
users	1	157B	157B	20KB	3	60KB	20KB

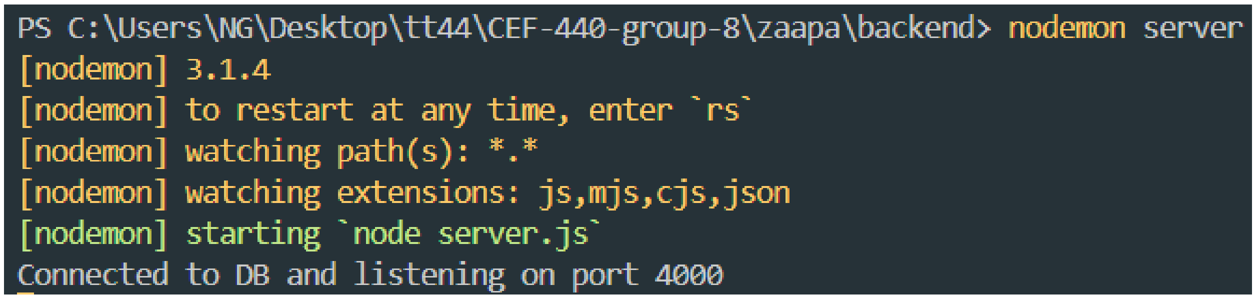
Figure 2: Zaapa-App Database

IV.3 Connecting Database to Backend

The MongoDB database was connected to the backend through Mongoose. The figure below shows how the connection was done. The connection string (URI) and port number were hidden in a .env file for security purposes.

```
mongoose.connect(process.env.URI)
.then(() => {
    app.listen(process.env.PORT, () => {
        console.log("Connected to DB and listening on port", process.env.PORT);
    });
})
.catch((error) => console.log(error));
```

The figure below shows successful connection of the server to the database.



```
PS C:\Users\NG\Desktop\tt44\CEF-440-group-8\zaapa\backend> nodemon server
[nodemon] 3.1.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
Connected to DB and listening on port 4000
```

Figure 3: Server-Database successful connection

IV.4 Creating Models

A fundamental part of the Model-View-Controller (MVC) architectural pattern, where it represents the structure and behavior of data in the application. For the purpose of this project, the following models were created

- **User Model:**

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

const Schema = mongoose.Schema;

const UserSchema = new Schema({
    name: {
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true,
        unique: true
    },
    password: {
        type: String,
        required: true
    },
    phoneNumber: {
        type: String, // Changed to String
        required: true,
        unique: true
    }
}, { timestamps: true }); // Added timestamps

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

- **Road State Model**

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

const Schema = mongoose.Schema;

const roadStateSchema  = new Schema({
    name:{
        type:String,
        required:true
    },
    description:{
        type:String,
        required:true
    }
})
module.exports = mongoose.model('RoadState', roadStateSchema)
```

- **Road Sign Model**

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

const Schema = mongoose.Schema

const roadSignSchema = new Schema({
    name:{
        type:String,
        required:true
    },
    description:{
        type:String,
        required:true
    },
    image:{
        type:String,
        required:true
    }
});
module.exports = mongoose.model('RoadSign', roadSignSchema)
```

- **Road Sate Marker Model**

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

const Schema = mongoose.Schema

const roadStatePosSchema = new Schema({
    name:{
        type:String,
        required:true
    },
    coordinates:{
        type:{
            type:String,
            enum:['Point'],
            required:true
        },
        coordinates:{
            type:[Number],
            required:true
        }
    }
});

roadStatePosSchema.index({coordinates: '2dsphere'})

module.exports = mongoose.model('RoadStatePosition', roadStatePosSchema)
```

- **Road Sign Marker Model**

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

const Schema = mongoose.Schema

const roadSignPosSchema = new Schema({
    name:{
        type:String,
        required:true
    },
    coordinates:{
        type:{
            type:String,
            enum:['Point'],
            required:true
        },
        coordinates:{
            type:[Number],
            required:true
        }
    }
});

roadSignPosSchema.index({coordinates: '2dsphere'})

module.exports = mongoose.model('RoadSignPosition', roadSignPosSchema)
```

IV.5 Creating Controllers:

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

- ***User Controller***

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

// Register user

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

    try {
       const newUser = new User({
            name,
            email,
            password,
            phoneNumber
        });

        await newUser.save();
        res.status(201).json({ message: 'User created successfully' });
    } catch (error) {
        res.status(500).json({ error: 'Internal server error' });
```

- ***Road State Controller***

```
const RoadState = require('../models/roadStateModel')

// Add road State
const addRoadState = async(req, res) => {
    const { name,description  } = req.body;

    try {
       const newRoadState = new RoadState({
            name,
            description
        });

        await newRoadState.save();
        res.status(201).json({ message: 'Road State added  successfully' });
    } catch (error) {
        res.status(500).json({ error: 'Internal server error' });
    }
}
```

```
// get road State

const getRoadState = (req, res) => {
    res.json({mssg: "Get all road States"})
}

module.exports = { addRoadState, getRoadState}
```

- ***Road Sign Controller***

```
const RoadSign = require('../models/roadSignModel')

// Get all road Signs
const getRoadSigns = (req, res) => {
    res.json({mssg:"All Road Signs"})
}

// upload roadSign
const addRoadSign = async(req, res) => {
    const { name,description , image } = req.body;

    try {
       const newRoadSign = new RoadSign({
            name,
            description,
            image
        });

        await newRoadSign.save();
        res.status(201).json({ message: 'Road Sign  added  successfully' });
    } catch (error) {
        res.status(500).json({ error: 'Internal server error' });
    }
}

module.exports = {getRoadSigns, addRoadSign}
-
```

- ***Road Sign Marker Controller***

```
const RoadSignPosition = require('../models/roadSignMarkerModel')


const addRoadSignLocation = async (req, res) => {
    const { name, coordinates } = req.body;
    try {
        const newSignPosition = new RoadSignPosition({
            name,
            coordinates: {
                type: 'Point',
                coordinates: coordinates // assuming coordinates is
[longitude, latitude]
            }
        });

        await newSignPosition.save();
        res.status(201).json({ message: 'Road Sign Location added
successfully' });
    } catch (error) {
        console.error('Error adding road sign location:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
    }


const getRoadSignLocations = (req, res) => {

}

module.exports = {addRoadSignLocation, getRoadSignLocations}
```

- ***Road State Marker Controller***

```
const RoadStatePosition = require('../models/roadStateMarkerModel')


const addRoadStateLocation = async (req, res) => {
    const { name, coordinates } = req.body;

    try {
        const newStatePosition = new RoadStatePosition({
            name,
            coordinates: {
                type: 'Point',
                coordinates: coordinates // assuming coordinates is
[longitude, latitude]
            }
        });

        await newStatePosition.save();
        res.status(201).json({ message: 'Road Sign Location added
successfully' });
    } catch (error) {
        console.error('Error adding road sign location:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
    }


const getRoadStateLocations = (req, res) => {

}

module.exports = {addRoadStateLocation, getRoadStateLocations}
```

IV.6 Creating Routes

These are the files to create the paths and actions to be performed to execute the controllers' functions. Following are the routes created for this project;

- ***User Route***

```
const {loginUser,registerUser} = require('../controllers/userController')
const express = require('express')

const router = express.Router();

router.post('/register', registerUser);
router.post('/login', loginUser);

module.exports = router
```

- ***Road State Route***

```
const {getRoadState, addRoadState}
=require('../controllers/roadStateController')
const express = require('express')

const router = express.Router();

router.get('/get', getRoadState)
router.post('/add', addRoadState)

module.exports = router
```

- ***Road Sign Route***

```
const { addRoadSign, getRoadSigns} =
require('../controllers/roadSignController')
const express = require('express')

const router = express.Router()

router.get('/get', getRoadSigns)
router.post('/add', addRoadSign)

module.exports = router
```

- ***Road State Marker Route***

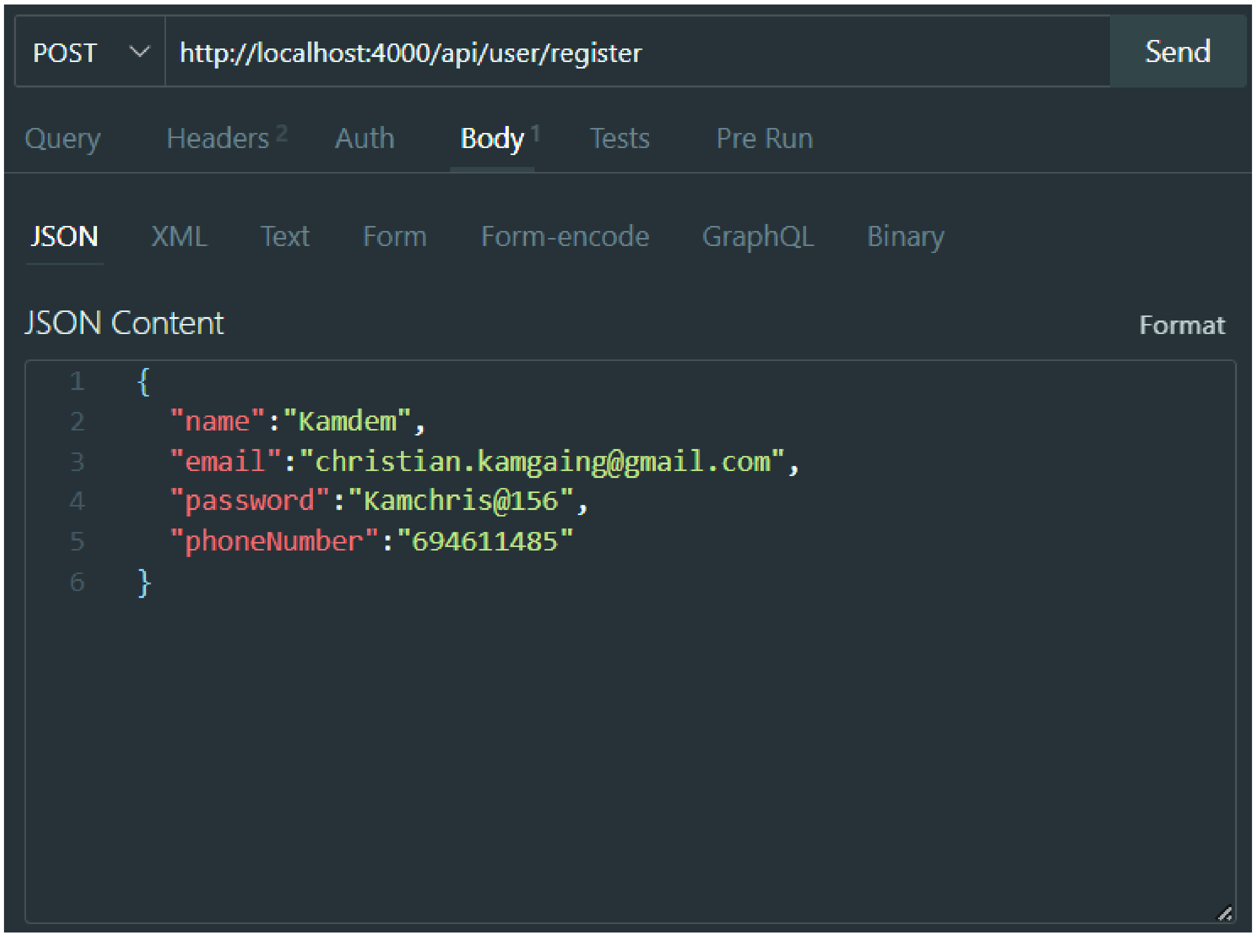
```
const {addRoadStateLocation, getRoadStateLocations} =
require('../controllers/roadStateMarkerController')
const express = require('express')

const router = express.Router();

router.post('/add', addRoadStateLocation)
router.get('/get',getRoadStateLocations)

module.exports = router;
```

- ***Route Sign Marker Route***

```
const {addRoadSignLocation, getRoadSignLocations} =
require('../controllers/roadSignMarkerController')
const express = require('express')

const router = express.Router();

router.post('/add', addRoadSignLocation)
router.get('/get',getRoadSignLocations)

module.exports = router;
```

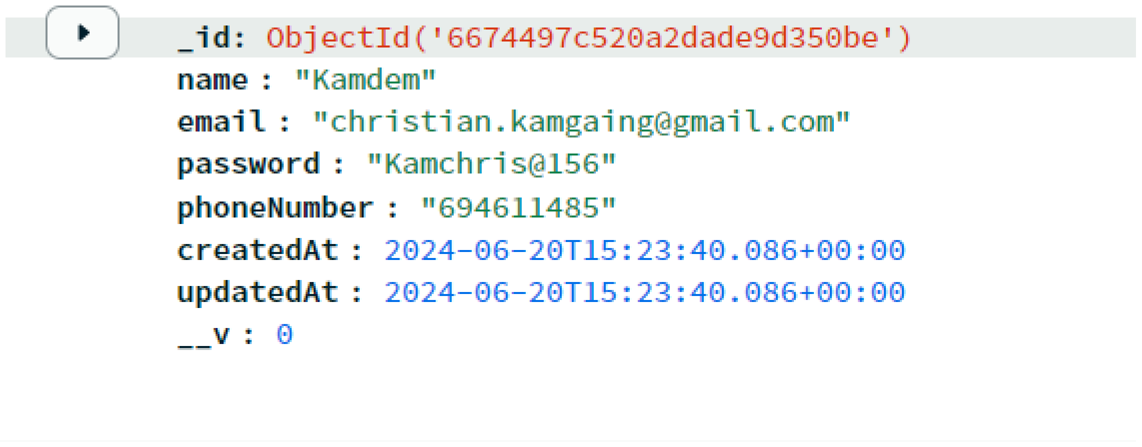
IV.7 Populating Database

Following the creation of routes, the *Thunder Client* tool was used to consume the APIs (routes). It also helps to test routes functionality and insert data into the different collections.



```
{
  "name":"Kamdem",
  "email":"christian.kamgaing@gmail.com",
  "password":"Kamchris@156",
  "phoneNumber":"694611485"
}
```

Figure 4: Testing with Thunder Client



```
_id: ObjectId('6674497c520a2dade9d350be')
name : "Kamdem"
email : "christian.kamgaing@gmail.com"
password : "Kamchris@156"
phoneNumber : "694611485"
createdAt : 2024-06-20T15:23:40.086+00:00
updatedAt : 2024-06-20T15:23:40.086+00:00
__v : 0
```

Figure 5: Result in the database

REFERENCES

[1]https://cloud.mongodb.com/v2/666e908cfcc9a06893541a9b#/metrics/replicaSet/666e9bad2039691cdf2f9b64/explorer/test/users/find (Visited on 18/06/2024)

[2] https://nodejs.org/fr (Visited on 18/06/2024)

[3] https://www.npmjs.com/ (Visited on 19/06/2024)

[4] https://nodemon.io/ (Visited on 20/06/2024)