FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

Trackify: Database Design And Implemenation

COURSE MASTER: Dr. NKEMENI VALERY

COURSE TITLE: Internet Programming and Mobile Programming

COURSE CODE: CEF 440

GROUP 23

S/N	NAME	MATRICULE
1	KINENUI STEPH	FE22A244
2	TIAYA FOTSEU JOSUE	FE22A315
3	NGANYU BRANDON YUNIWO	FE22A258
4	NGULEFAC FOLEFAC FOBELLA	FE19081
5	JAMISON LII	FE22A284

Table of Content

1. Introduction

In the rapidly evolving landscape of mobile and web applications, robust and efficient database design is paramount to the success and sustainability of any digital product. A well-structured database serves as the backbone, ensuring data integrity, fast retrieval, and scalability to accommodate growing user bases and data volumes. For applications like Trackify, a mobile network monitoring tool, the ability to store, process, and analyze vast amounts of real-time network data, user interactions, and performance metrics is critical. Traditional relational databases, while excellent for structured data with predefined schemas, often present challenges when dealing with the dynamic, unstructured, or semi-structured nature of data generated by modern applications.

This document details the database design and implementation for Trackify, with a specific focus on leveraging MongoDB as the backend database solution. MongoDB, a leading NoSQL document database, offers distinct advantages that make it particularly suitable for Trackify's requirements. Its schema-less design provides unparalleled flexibility in handling diverse data types, from simple user profiles to complex nested speed test results and signal logs, without requiring rigid schema definitions upfront. This flexibility significantly accelerates development cycles and allows for agile adaptation to evolving application features.

Furthermore, MongoDB's inherent scalability, achieved through horizontal scaling (sharding), enables Trackify to effortlessly manage an increasing load of concurrent users and data points. Its document-oriented nature aligns seamlessly with the object-oriented programming paradigms prevalent in modern JavaScript-based tech stacks, such as Node.js and Express. This compatibility streamlines the development process by minimizing the impedance mismatch between application code and database storage, leading to more intuitive data modeling and efficient data manipulation. This report will elaborate on these aspects, outlining the conceptual design, implementation details, and the integration of MongoDB with Trackify's backend.

2. Data Elements

The core functionality of Trackify revolves around capturing, storing, and analyzing various types of data generated by user interactions and network monitoring activities. These data elements represent real-world entities and events, each with distinct characteristics and relationships. Understanding these elements is crucial for designing an effective database schema.

The **User Info** collection serves as the central point of reference and stores essential data about each registered user, including authentication tokens and customizable settings such as privacy preferences. It forms the foundation of the system's relational logic, with all other collections referencing the unique user identifier (_id) through MongoDB's ObjectId mechanism.

The **Network Feedback** collection is designed to store subjective user ratings and feedback. It captures qualitative and quantitative data including the feedback category (e.g., signal, speed, call quality), a 1–5 rating, optional user comments, and the timestamp of submission. This collection supports the app's mission of capturing user-perceived Quality of Experience (QoE).

The **Network Metrics** collection holds objective performance data gathered during speed tests or background network checks. It includes download and upload speeds, latency, packet loss, and the time the test was performed. This collection enables real-time analytics and supports visual summaries on the user dashboard.

The **Location Data** collection records the user's geographical context during signal monitoring or network testing. It includes GPS coordinates, network type (3G/4G/5G), signal strength, and the timestamp. This information helps identify network blind spots and allows users to correlate their experience with location-specific data.

Lastly, the **History Logs** collection tracks user interactions within the app, such as performed tests, submitted reports, or switched networks. Each log entry includes the activity type, a short description, and a timestamp. This collection provides transparency and a chronological overview of user activity, aiding both users and administrators..

Each of these data elements is designed to capture specific aspects of network monitoring and user interaction, collectively enabling Trackify to deliver its core functionalities effectively and efficiently.

3. Conceptual Design

The conceptual design for Trackify is based on five core entities: **User**, **Network Metric**, **Feedback**, **Administrator**, and **Mobile Network Provider**. These entities are implemented using **MongoDB** collections, with clearly defined attributes and relationships that reflect how the system captures, stores, and processes network-related data.

The **User** entity stores authentication details, user preferences, and metadata such as language and role (either “user” or “admin”). By embedding the role directly into the user document, Trackify simplifies access control and reduces complexity in the data model.

json

```
{
  "_id": ObjectId("..."),
  "name": "Jane Doe",
  "email": "jane@example.com",
  "preferred_language": "English",
  "role": "user"}
```

The **Network Metric** collection stores quantitative data related to network performance. Each metric entry is linked to a specific user and includes values such as signal strength, latency, network type, geographic coordinates, and a timestamp. These documents are referenced using the user’s ObjectId to maintain scalability and avoid bloating the user document.

json

```
{
  "_id": ObjectId("..."),
  "userId": ObjectId("..."),
  "timestamp": ISODate("2023-10-26T11:00:00Z"),
  "signal_strength": -75,
  "latency": 35,
  "network_type": "4G",
  "cell_id": "310-260",
  "latitude": 6.5244,
  "longitude": 3.3792}
```

The **Feedback** collection stores user-submitted ratings and comments about network experience. Each entry is linked to a user, with fields for rating (e.g., 1–5), written comments, the language of the feedback, and the network type experienced. This data helps the backend analyze subjective Quality of Experience (QoE) reports alongside objective metrics.

json

{

"_id": ObjectId("..."),

"userId": ObjectId("..."),

"timestamp": ISODate("2023-10-26T12:00:00Z"),

"rating": 4,

"comment": "Good speed but inconsistent in evenings",

"language": "English",

"network_type": "4G"}

The **Administrator** role is embedded within the User entity via the role field, which allows admin-specific access rights. Admins can view all submitted feedback and metrics for oversight and analytics but do not generate these entries themselves.

The **Mobile Network Provider** is a passive entity in the data model. It does not directly interact with the system but receives periodic **aggregated reports** via the backend for analysis and decision-making. As such, it does not have a physical collection but is modeled conceptually as a recipient of processed data.

This model ensures that:

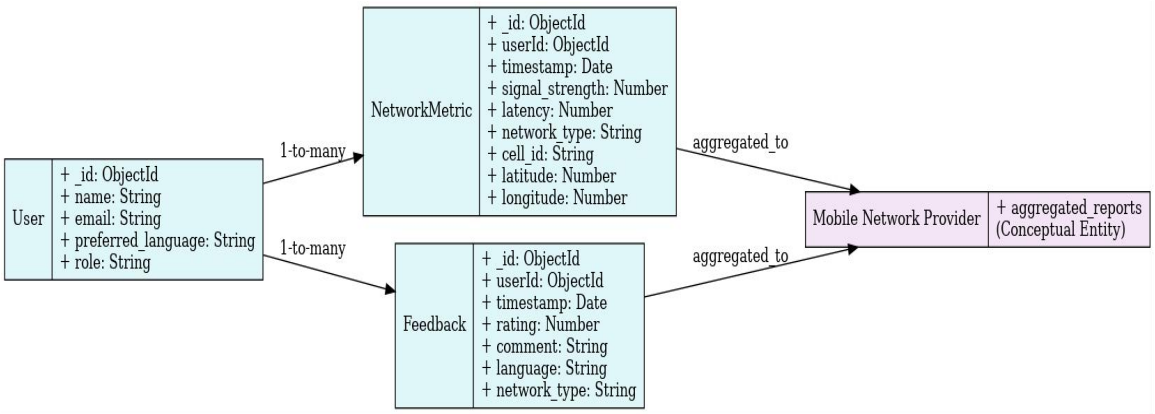
Each **User** can generate multiple **Metrics** and **Feedback** entries.

Admins can access all metrics and feedback but do not submit them.

Mobile Network Providers receive summarized data.

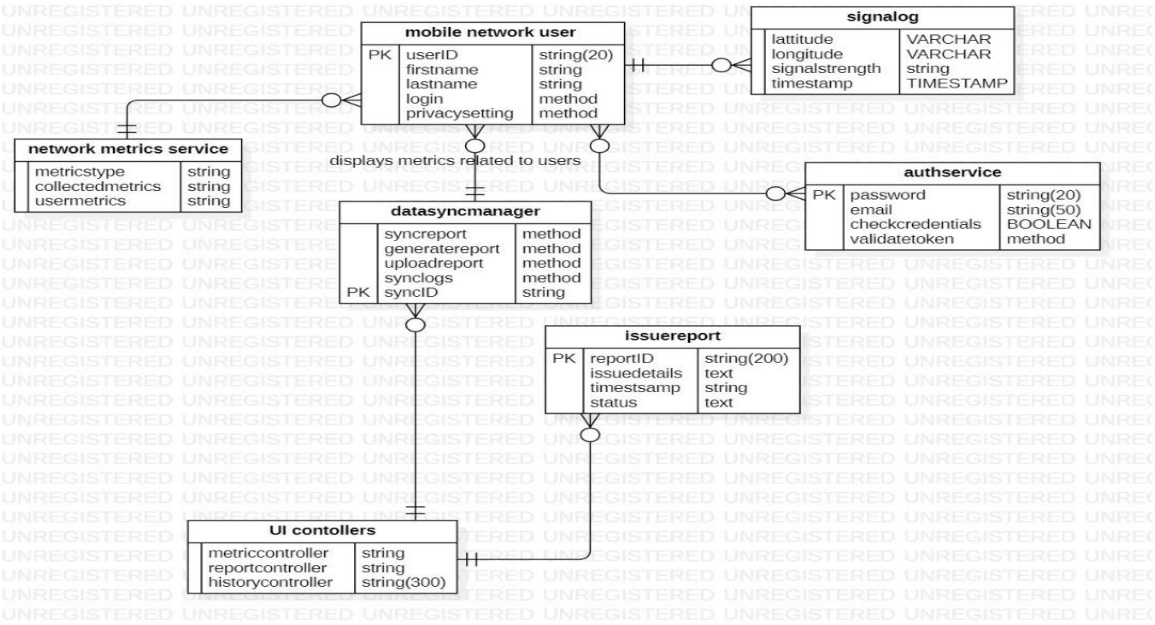
MongoDB's referencing strategy keeps documents clean and scalable.

This conceptual structure reflects Trackify's mission to combine user-reported QoE and real-time technical data in a flexible, secure, and analytics-ready design.



4. ER Diagram

This report provides an overview of the entities present in the Mobile Network User System as depicted in the ER diagram. Each entity is defined along with its attributes and relationships to other entities.



4.1 Entities and Attributes

1. Mobile Network User

'userID': string (Primary Key) - Unique identifier for each user.

'firstName': string - User's first name.

'lastName': string - User's last name.

'login()': method - Facilitates user login.

'privacySettings()': method - Allows users to manage privacy options.

Description: Represents individuals using the mobile network service. Each user can log in and manage their profile settings.

2. Signal Log

'latitude': VARCHAR - Geographic latitude of the user's location.

'longitude': VARCHAR - Geographic longitude of the user's location.

'signalStrength': string - Strength of the mobile signal at the user's location.

‘timestamp‘: TIMESTAMP - Time when the signal was recorded.

Description: Stores data related to the signal strength experienced by users at specific locations and times.

3. Network Metrics Service

‘metricType‘: string - Type of metrics collected (e.g., signal strength, user metrics).

‘collectedMetrics‘: string - Details about the metrics collected from users.

‘userMetrics‘: string - Metrics specific to individual users.

Description: Provides functionalities to collect and analyze metrics related to network performance and user experience.

4. DataManager

‘syncReport()‘: method - Synchronizes reports from different sources.

‘generateReport()‘: method - Creates reports based on collected data.

Description: Handles data management tasks, including synchronizing and generating reports related to user metrics and network performance.

5. AuthService

‘password‘: string(200) - User’s account password.

‘email‘: string(200) - User’s email address for authentication.

‘checkCredentials()‘: method - Validates user login credentials.

‘validateToken()‘: method - Validates authentication tokens for user sessions.

Description: Manages user authentication, ensuring secure access to the system by validating user credentials and session tokens.

6. Issue Report

‘reportID‘: string(200) (Primary Key)

Unique identifier for each issue report.

‘issueDetails‘: text

Description of the reported issue.

‘timestamp‘: string Time when the issue was reported.

‘status‘: text Current status of the reported issue (e.g., open, resolved).

Description: Represents issues reported by users, allowing the system to track and manage user-reported problems.

7. UI Controllers

‘metricController‘: string - Manages user interface for metrics display.

‘reportController‘: string - Manages user interface for issue reports.

‘historyController‘: string - Manages user interface for historical data.

• **Description:** Controls the user interface components, facilitating user interactions with the system for displaying metrics, reports, and historical data.

Relationships

Network User has a one-to-many relationship with signal Log (each user can have multiple signal logs).

Mobile Network User has a one-to-many relationship with Issue Report(each user can report multiple issues).

DatasyncManager interacts with Network Metrics Service to display metrics related to users. This visual representation helps in understanding the data flow and how different parts of the application's data model interact within MongoDB's document-oriented paradigm.

5. Database Implementation

The implementation of the Trackify database utilized MongoDB Atlas, a cloud-hosted database service, for its ease of setup, scalability, and managed services. This allowed the development team to focus on application logic rather than database infrastructure. Once the MongoDB cluster was provisioned, the next step involved structuring the collections and defining the document schemas, albeit with MongoDB's inherent flexibility.

5.1 Collection Structure and Document Examples

For **Trackify**, the MongoDB-based NoSQL backend is structured around three primary collections: users, networkMetrics, and feedback. Each collection holds documents tailored to specific types of information that support both user experience monitoring and system analytics. A conceptual data recipient—MobileNetworkProvider—does not exist as a physical collection but is supported through backend aggregation pipelines.

Users Collection

Stores account information and access roles. Each user has preferences embedded directly in their document, and the role field distinguishes between regular users and administrators.

json

```
{
  "_id": ObjectId("653a7b1c0a1b2c3d4e5f6789"),
  "name": "network_enthusiast",
  "email": "user@trackify.com",
  "preferred_language": "English",
  "role": "user",
  "registrationDate": ISODate("2023-01-15T10:00:00Z"),
  "lastLogin": ISODate("2023-10-26T14:30:00Z"),
  "settings": {
    "theme": "dark",
    "notifications": {
      "email": true,
      "inApp": true
    }
```

},

"isActive": true}

Networkmetrics Collection

Captures objective performance data such as signal strength, network type, and location at the time of recording. This data is referenced back to the user who generated it via userId.

Json

{

"_id": ObjectId("653a7b1c0a1b2c3d4e5f6790"),

"userId": ObjectId("653a7b1c0a1b2c3d4e5f6789"),

"timestamp": ISODate("2023-10-26T14:05:30Z"),

"signal_strength": -95,

"latency": 22,

"network_type": "5G",

"cell_id": "ABC123DEF456",

"latitude": 40.7128,

"longitude": -74.0060}

Feedback Collection

Holds user-submitted qualitative feedback such as ratings and comments. Each feedback record includes the user reference and metadata like language and network type.

json

{

"_id": ObjectId("653a7b1c0a1b2c3d4e5f6791"),

"userId": ObjectId("653a7b1c0a1b2c3d4e5f6789"),

"timestamp": ISODate("2023-10-26T15:00:00Z"),

"rating": 3,

"comment": "Network drops frequently in the evening.",

"language": "English",

"network_type": "LTE"}

MobileNetworkProvider (Conceptual Entity)

Although not a stored collection in MongoDB, the **Mobile Network Provider** is treated as a data consumer at the backend level. Aggregated results from both the networkMetrics and feedback collections are processed and formatted for provider dashboards or reports. This abstraction allows the backend to deliver summarized insights without compromising raw user-level data integrity.

This document structure ensures clear separation of concerns, optimal storage, and efficient querying, while also supporting the app's goals of network performance monitoring, feedback collection, and administrative insight.

5.2 Indexing for Search Optimization

To ensure fast query performance, especially for frequently accessed fields, indexes were strategically applied:

- `users` collection: An index on `email` (unique) and `username` for quick user lookups during login.
- `speedTests` collection: Indexes on `userId` and `timestamp` for efficient retrieval of a user's test history and for time-based data analysis. A geospatial index on `location.coordinates` (2dsphere) was also added to support location-based queries for network performance mapping.
- `signalLogs` collection: Similar to `speedTests`, indexes on `userId`, `timestamp`, and `location.coordinates` were crucial for analyzing signal quality over time and across geographical areas.
- `feedbackReports` collection: An index on `userId` and `timestamp` to quickly retrieve a user's submitted reports or to sort reports by submission date.

5.3 Schema Validation

While MongoDB is schema-less, schema validation was considered and partially implemented to enforce data consistency for critical fields. Using JSON Schema, rules were defined at the collection level to ensure that documents inserted into `users` or `speedTests` adhered to expected data types and required fields. This helps prevent malformed data from entering the database, enhancing data quality and application stability.

5.4 Test and Seed Data Insertion

Initial test data and seed data were inserted programmatically using scripts and MongoDB's `mongoimport` utility. This allowed for populating the database with realistic scenarios for development and testing purposes, ensuring that the application logic could be thoroughly tested against various data conditions.

6. Backend Implementation

The backend for Trackify was developed using Node.js, a powerful JavaScript runtime, combined with the Express.js framework, a fast, unopinionated, minimalist web framework for Node.js. This choice provided a robust and scalable environment for building RESTful APIs that interact with the MongoDB database. A critical component in facilitating this interaction was Mongoose, an Object Document Mapper (ODM) for MongoDB and Node.js.

6.1 Node.js with Express for API Development

Express.js was utilized to create a series of RESTful API endpoints that handle all client-server communication. These endpoints are responsible for receiving requests from the mobile application, processing them, interacting with the database, and sending back appropriate responses. Examples of such routes include:

- POST /api/auth/register: Handles new user registration.
- POST /api/auth/login: Authenticates users and issues JWTs for subsequent requests.
- POST /api/speedtests: Receives and saves new speed test results.
- GET /api/speedtests/:userId: Retrieves a user's historical speed test data.
- POST /api/feedback: Submits user feedback or bug reports.
- PUT /api/users/:userId/preferences: Updates a user's application preferences.

Middleware functions were extensively used for tasks like authentication (verifying JWTs), authorization, input validation, and error handling, ensuring the API is secure and robust.

6.2 Mongoose for Schema Modeling and CRUD Operations

Mongoose plays a pivotal role in bridging the gap between Node.js application code and MongoDB's document-oriented database. While MongoDB is schema-less, Mongoose allows for defining schemas at the application level, providing structure, validation, and a rich set of features for interacting with the database. For each core data element, a corresponding Mongoose schema and model were defined:

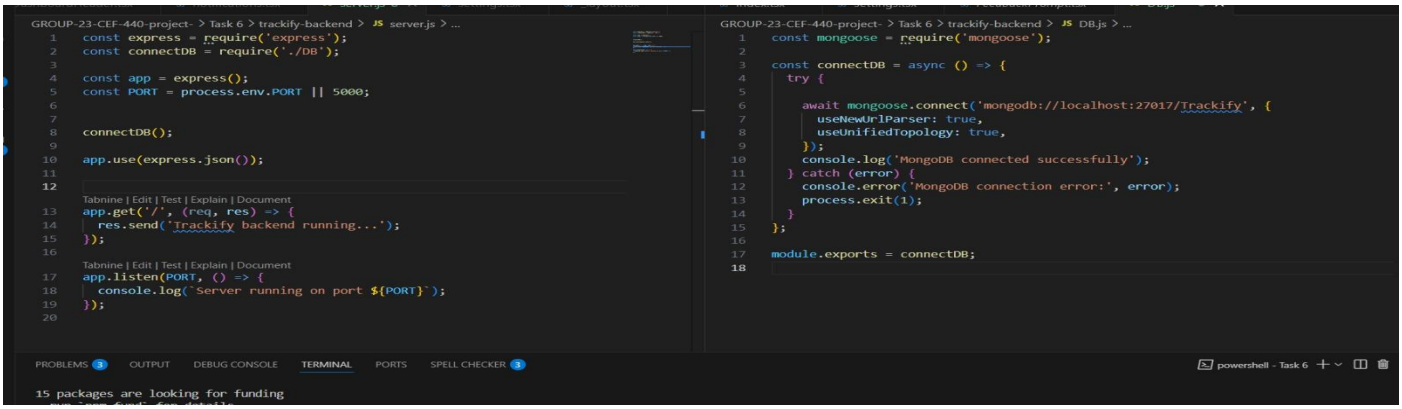
```
// Example Mongoose Schema for SpeedTest
const speedTestSchema = new mongoose.Schema({
  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  timestamp: { type: Date, default: Date.now },
  downloadSpeedMbps: { type: Number, required: true },
  uploadSpeedMbps: { type: Number, required: true },
```

```
  pingMs: { type: Number, required: true },
  networkType: { type: String, enum: ['Wi-Fi', '2G', '3G', '4G', '5G'], required: true },
  deviceInfo: {
    model: String,
    os: String
  },
  location: {
    type: { type: String, enum: ['Point'], default: 'Point' },
    coordinates: { type: [Number], required: true } // [longitude, latitude]
  },
  isp: String
});

speedTestSchema.index({ userId: 1, timestamp: -1 }); // Compound index for queries
speedTestSchema.index({ "location.coordinates": "2dsphere" }); // Geospatial index

const SpeedTest = mongoose.model('SpeedTest', speedTestSchema);
```

This schema enforces data types, defines required fields, and can include custom validation logic. Mongoose models then provide an intuitive API for performing CRUD (Create, Read, Update, Delete) operations. For example:

- **Create:** const newTest = new SpeedTest(req.body); await newTest.save();
- **Read:** const userTests = await SpeedTest.find({ userId: req.user.id }).sort({ timestamp: -1 });
- **Update:** await User.findByIdAndUpdate(req.params.userId, { $set: req.body });
- **Delete:** await FeedbackReport.findByIdAndDelete(req.params.reportId);

Mongoose also facilitates operations like population (to retrieve referenced documents), aggregation (for complex data analytics), and pre/post hooks for middleware-like logic on schema events. This comprehensive approach ensured a well-structured, maintainable, and efficient backend for Trackify.



7. Connecting Database to Backend

Establishing a secure and reliable connection between the Node.js/Express backend and the MongoDB database is a foundational step in the Trackify application. This integration was primarily achieved using Mongoose, which internally leverages the official MongoDB driver. Ensuring a robust connection involves proper configuration, secure handling of credentials, and effective error management.

7.1 Integration with Mongoose

Mongoose simplifies the connection process considerably. The connection string, which includes the database host, port, and authentication details, is passed to the `mongoose.connect()` method. This method returns a promise, allowing for asynchronous handling of the connection status.

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

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
      // useFindAndModify: false, // Deprecated in newer Mongoose versions
      // useCreateIndex: true     // Deprecated in newer Mongoose versions
    });
    console.log(`MongoDB Connected: ${conn.connection.host}`);
  } catch (err) {
    console.error(`Error: ${err.message}`);
    process.exit(1); // Exit process with failure
  }
};

module.exports = connectDB;
```

This connection function is typically called once when the backend application starts, ensuring that the database connection pool is initialized and ready to handle incoming requests.

7.2 Secure Environment Variables

Crucially, sensitive database connection URIs, including credentials, are never hardcoded directly into the application's source code. Instead, they are stored as environment variables. The `dotenv` package is used in

development environments to load these variables from a `.env` file. In production, these variables are configured directly on the hosting platform (e.g., Heroku, AWS, DigitalOcean) where the Node.js application is deployed. An example `.env` entry would look like:

MONGO_URI=mongodb+srv://<username>:<password>@cluster0.abcde.mongodb.net/trackify_db?retryWrites=true&w=majority

This practice significantly enhances security by preventing sensitive information from being exposed in version control systems or publicly accessible codebases.

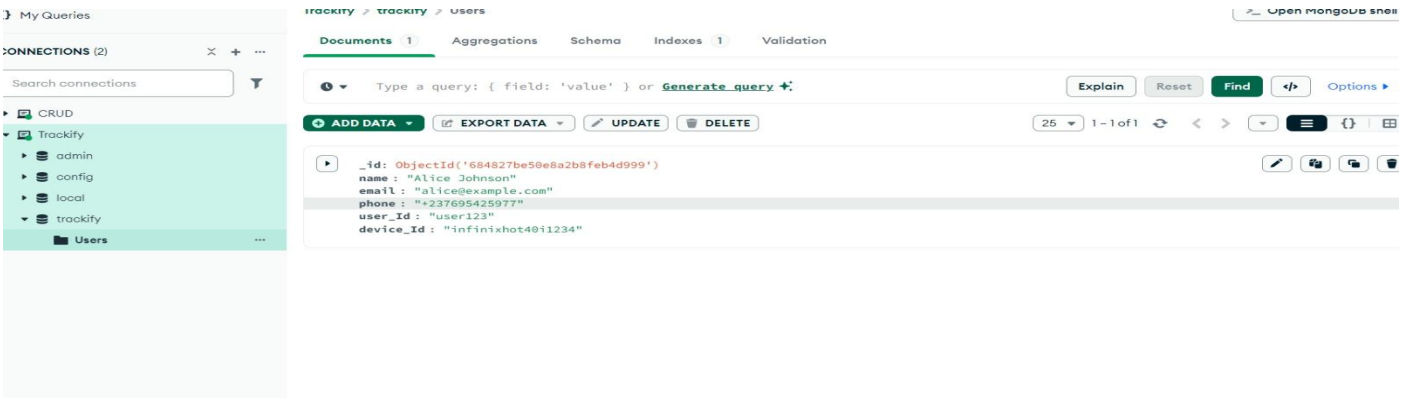
7.3 Error Handling During Connection

Robust error handling for database connection failures is critical to prevent application crashes and provide meaningful diagnostics. As shown in the `connectDB` function above, a `try-catch` block is used to catch any errors that occur during the `mongoose.connect()` call. If a connection error occurs, the error message is logged, and the application exits. This "fail-fast" approach prevents the application from running in a state where it cannot perform its core functions due to database unavailability.

7.4 Middleware for Endpoint Protection and Input Sanitization

Beyond connection, security is maintained at the API endpoint level through middleware. Authentication middleware ensures that only authenticated users can access protected routes by validating JSON Web Tokens (JWTs). Authorization middleware further restricts access based on user roles (e.g., only administrators can view all feedback reports).

Input sanitization middleware (e.g., using libraries like `express-validator` or `express-mongo-sanitize`) is also implemented to prevent common security vulnerabilities such as NoSQL injection and Cross-Site Scripting (XSS). This ensures that data received from clients is clean and safe before it interacts with the database or is sent back in responses.



8. Conclusion

The strategic decision to utilize MongoDB as the backend database solution for Trackify has proven instrumental in building a highly scalable, flexible, and performant mobile network monitoring application. MongoDB's document-based structure provided a natural fit for the diverse and often semi-structured data generated by user interactions and network performance tests, allowing for agile schema evolution without the rigidity of traditional relational databases.

The core functionalities of Trackify, including comprehensive user interaction tracking (user profiles, preferences), detailed performance analytics (speed test results, signal logs), and efficient real-time data retrieval, are robustly supported by MongoDB's architecture. The ability to model complex relationships through a combination of document embedding for frequently accessed, contained data and referencing via ObjectIDs for large, independently manageable datasets has optimized data access patterns and ensured efficient storage.

Furthermore, the seamless integration with a modern JavaScript-based backend, specifically Node.js with Express and Mongoose, significantly streamlined the development process. Mongoose's ODM capabilities provided a powerful abstraction layer, simplifying schema definition, validation, and CRUD operations, thus accelerating development and enhancing code maintainability. The focus on secure database connection practices, including the use of environment variables for sensitive URIs and robust error handling, has also contributed to the application's overall stability and security posture.

In summary, MongoDB's inherent flexibility, scalability, and developer-friendliness have enabled Trackify to effectively manage large volumes of dynamic network data, support a growing user base, and deliver a responsive and reliable user experience. This database design and implementation not only meets the current requirements of the application but also provides a solid foundation for future enhancements and scaling initiatives.

9. References

- MongoDB Documentation: https://docs.mongodb.com/
- Mongoose Documentation: https://mongoosejs.com/docs/
- Express.js Guide: https://expressjs.com/
- "Designing Data-Intensive Applications" by Martin Kleppmann.
- "MongoDB: The Definitive Guide" by Kristina Chodorow.
- OWASP Top 10 Web Application Security Risks.