**Project 2 - Report:**

**Graph-Based Recommendation System Project Report**

**Course: CSC 3356 Data Engineering and Visualization**

**Project Number: 2**

**Submission Date: 04/01/2023**

**Al Akhawayn University in Ifrane**



**By: Adnane El Amrani & Ayoub Maimmadi**

**CSC - 5356**

**Spring 2024**

**Under the supervision of: Dr.Tajjeeddine Rachidi**

**Summary of what we were asked to implement during the project:**

**Introduction : https://github.com/elamraniadnane1/Project\_3/blob/main/readme.md**

**1. Project Statement:**

* Introduction to recommendation systems.
* Explanation of traditional recommendation system techniques (collaborative filtering, content-based filtering, hybrid models).
* Limitations of traditional methods.
* Introduction of graph-based recommendation systems and their advantages (scalability, diversity of relationship modeling, dynamic content handling).

**2. Project Objective:**

* To build a graph-based recommendation system for a Twitter-like platform.
* System to recommend hashtags and users to follow.
* Emphasis on using a graph database to dynamically update and generate recommendations.

**3. Proposed Solution Architecture:**

* Using microservice-based architecture.
* Description of the components involved (Neo4J graph database, Kafka for streaming, NLP processing engine, etc.).
* Requirement to justify any architectural deviations.

**4. Implementation Guidelines:**

* Steps to deploy Neo4J and setup default graph database.
* Understanding the data model.
* Extraction and selection of users and hashtags for the demo.
* Writing and validating queries in Cipher for graph updates and recommendation logic.
* Designing MongoDB schema for user profiles and tweets.
* Implementation of sentiment analysis and integration with Kafka and Neo4J.

**5. Demonstration:**

* Requirement for a full-screen video recording demonstrating the installation and execution of the project.
* Specific elements to be shown in the video.

**6. Credits and Grading:**

* Submission guidelines for credit.
* Detailed grading rubric covering:
* Working project.
* Quality of the report.
* Ease of deployment.
* Quality of the video presentation.

**Table of Contents**

1. **Project Team**
2. **Project Definition**
3. **Project Design**
4. **Project Implementation**
5. **Implementation Limitations**
6. **Project Demo and Codebase**
7. **References**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Project Team:**

|  |  |
| --- | --- |
| Team Member | Contributions |
| Adnane El Amrani | - Coordinated the project timeline and deliverables.  - Managed communications and Git Versioning.  - Managed the consistency and hierarchy of the Report / Readme.md file. |
| Adnane El Amrani | - Set up the Neo4J database and implemented graph data modeling  - Set up the tweets default graph database that comes with it  - Integrated Kafka with Neo4J.  - Develop the back-end architecture using Node.js |
| Ayoub Maimmadi | - Developed the front-end interface using React.js.  - Get acquainted with the underlying data model  - Implemented state management.  - Prepared a test dataset of tweets by a selected user (The tweets should contain some of selected hash tags and other users) |
| Ayoub Maimmadi | - Assisted with backend and frontend integration.  - Get acquainted with the underlying Data Model Query this database to extract the list of users and hashtags and Select users and hashtags  - Developed API endpoints in Node.js. |
| Adnane & Ayoub | - Configured MongoDB collections and schemas.  - Designed MongoDB schema for storing user profile and tweets  - Managed data flow between services.  - Wrote the graph insertion, recommendation, and sentiment update queries in cipher |
| Ayoub & Adnane | - Ensured the quality of the application through testing.  - Configured Neo4j Connector with Neo4J as a sink to update the appropriate node |
| Adnane El Amrani | - Managed MongoDB cloud deployments & Synchronization with the database. |
| Ayoub Maimmadi | - Implemented algorithms for sentiment analysis.  - Used an NLP library that works well with the stream processing  - Optimized data processing workflows.  - Implement algorithms to perform Integration and Unit Testing |

1. **Project Definition**

The aim of this project is to design and implement a graph-based recommendation system that provides personalized content to users of a Twitter-like application based on the architecture below.

The system will offer recommendations for hashtags and other users to follow, leveraging real-time analysis of user interactions.

**Functional Requirements:**

* **Real-Time Interaction Handling**: The system must capture user interactions with content, such as posting tweets and using hashtags, in real-time.
* **Personalized Recommendation Generation**: Users should receive personalized recommendations for hashtags and other users to follow.
* **Dynamic Content Updating**: The recommendation system should incorporate new data dynamically, updating the recommendations as new tweets are posted and interactions are made.
* **Scalability**: The system should handle a growing amount of data and users without degradation of performance.
* **User Authentication**: Secure user authentication must be implemented to protect user data and interactions.
* **Sentiment Analysis**: Tweets should be analyzed for sentiment, which will influence the recommendations provided to the user.
* **Data Persistence**: The system must store persistent user profiles and interaction history to inform future recommendations.

**Non-Functional Requirements:**

* **Scalability**: The system must be capable of scaling out to accommodate an increasing load of data and user traffic.
* **Reliability**: The recommendation system should have high availability and fault tolerance to ensure consistent operation.
* **Performance**: Response times for generating recommendations must be low to enhance user experience.
* **Modularity**: The system should be built using a microservices architecture to facilitate maintenance and future expansion.
* **Data Integrity**: The system must ensure the accuracy and consistency of user data and recommendations.
* **Security**: User data, including profiles and interaction data, must be securely stored and transmitted.

The architecture of the system is built upon modern data engineering and visualization practices, integrating several components:

1. a Neo4J graph database for managing complex relationships between users and content, MongoDB for storing user profiles and tweets, Node.js for server-side logic, React.js for front-end logic and Kafka streams for real-time data processing and sentiment analysis.
2. MongoDB is a document-oriented NoSQL database that uses JSON-like schemas, which makes it an excellent choice for handling unstructured or semi-structured data in modern web applications. It complements the graph database capabilities of Neo4J by providing a different model for data storage and retrieval, which is particularly well-suited for storing large volumes of data that doesn't necessarily require complex relationships, such as user profiles and tweets.
3. The system is designed to be modular, with each component interacting seamlessly through well-defined interfaces.
4. This modularity ensures that each part of the system can be updated or replaced without affecting the overall functionality.
5. Additionally, the use of a graph-based database is intended to overcome the limitations of traditional recommendation engines, providing a more dynamic and scalable solution that can adapt to the evolving nature of user data and content.
6. **Project Design**

**A diagram of a diagram

Description automatically generated**

The system design integrates a combination of technologies tailored to handle real-time data processing, graph-based recommendation logic, and persistent storage, adhering to principles of microservices architecture.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Component | Description | Data Models | Encoding | Querying | Scalability & Reliability Configurations |
| Front-end (React) | React.js-based user interface for user interactions. | React components, states, hooks, and router for navigation. | UTF-8 | RESTful API calls to Node.js backend using Axios. | Responsive design. Client-side state management with Redux. |
| Node.js Backend and Postman API for routes and CRUD Operations | Server-side logic handling RESTful API requests. | JS objects for handling request/response data structures. | JSON | Mongoose for MongoDB; neo4j-driver for Neo4j queries. | Stateless design for load balancing and easy scaling. |
| Neo4J Database | Graph database for storing and querying tweet relationships. | Nodes and relationships with properties for tweets and users. | Neo4j Bolt protocol | Cypher for graph operations, optimized with indexes and constraints. | Clustering and causal clustering for fault tolerance and data consistency. |
| MongoDB Database  and Postman API for routes and CRUD Operations | Document database for storing user profiles and tweets stored in a cloud (cloud version) | Mongoose models for users and posts with schema validation. | BSON | Mongoose ORM for CRUD operations, indexes for efficient access. | Sharding for horizontal scaling; replication for high availability. |
| Kafka Streaming | Platform for real-time data processing and streaming. | Producers and Consumers for streaming tweet data. | Serialized bytes | KafkaJS for producing and consuming messages within Kafka topics. | Partitioning for load distribution; replication for message durability. |
| Authentication | Security layer for user session management and access control. | JWT for session tokens; bcrypt for password hashing. | Base64 encoded JWT | JWT and bcrypt libraries for securing user access. | HTTPS for secure data transmission; token rotation for enhanced security. |
| API Communication | Facilitates interaction between the client and server. | Axios for HTTP communication; React Router for SPA navigation. | JSON | Axios for HTTP requests; Neo4j queries for data retrieval. | Axios interceptors for error handling; React suspense for better UX during loading. |
| Neo4j Integration | Interacts with Neo4j to perform CRUD operations. | Function to post and retrieve tweets and user data. | N/A | Neo4j-driver for direct database operations. | Connection pooling and session management for efficient resource utilization. |
| State Management | Global state management for React components. | Redux for state container; Thunk middleware for async actions. | JSON | Dispatch actions to update Redux store. | Redux devtools for debugging; Middleware for handling asynchronous operations. |
| Kafka for Sentiment Analysis (implement either on local machine or on Docker) | Analysis of tweet sentiment for enhanced recommendations. | Sentiment analysis library to evaluate tweet content. | Serialized bytes | Sentiment library for processing tweet content; Kafka for streaming results. | Consumer groups for processing parallel streams; topic compaction for stateful processing. |
| Data Models (Mongoose) (Flexible schemas) | Schemas for user and tweet data in MongoDB. | Schemas with validation for user and tweet data. | BSON | Mongoose methods for operations; Aggregation framework for complex queries. | Use of indexes to optimize query performance; connection handling for stability. |
| Error Handling (Express.js) | Provides user-friendly error feedback. | Error middleware for handling and responding to API errors. | UTF-8 | Middleware for capturing and formatting errors. | Centralized error handling to ensure consistent error responses. |

Unit & Integration testing were done using : Mocha/Chai

**Data Models and Storage:**

* **Neo4J Graph Database:** Utilized to model complex relationships between entities such as users, tweets, and hashtags. Neo4J's graph structure is inherently suitable for recommendation systems where relationships are as critical as the data points themselves. The Cypher query language, which is designed for graph databases, is used to write queries that can efficiently traverse these relationships to generate recommendations.
* **MongoDB:** Selected for user profile storage and batch processing of tweet data due to its flexibility, scalability, and strong query capabilities. MongoDB's schema-less nature allows for easy modification of data structures as new requirements arise.

**Real-Time Data Processing:**

* **Apache Kafka:** Serves as the central nervous system for real-time data streaming. It ingests raw tweet data which is then processed by Kafka Streams and Apache Spark for real-time analytics.
* **Kafka Streams:** Used for lightweight processing of the real-time data feed, Kafka Streams handles the stream processing logic within the Kafka ecosystem. Its purpose is to filter, aggregate, and prepare data for sentiment analysis.
* **Apache Spark Streaming:** A powerful engine for stream processing that can handle complex transformations and analyses, including running machine learning algorithms in real-time. In this design, it’s tasked with performing sentiment analysis on the tweet streams.

**Microservices Architecture:**

* **User Authentication:** Though not detailed in the provided diagram, a secure authentication mechanism is crucial. Node.js has robust libraries like Passport.js which can easily integrate with various authentication strategies.
* **Node.js:** Replacing Flask, Node.js offers a non-blocking I/O model that's efficient and lightweight, perfect for handling many simultaneous connections with high throughput, which makes it ideal for web clients' communication needs in a real-time system.

**Scalability and Reliability:**

The architecture ensures that each component can be scaled independently, with Kafka and MongoDB supporting data sharding and load distribution.

The system is designed for fault tolerance, utilizing Kafka's replication for messaging and data partitioning to maintain service continuity in case of node failures.

**Justifications for Architecture Choices:**

* **Node.js**: Selected for its excellent support in handling concurrent connections and its single-threaded event loop that can provide non-blocking I/O operations. This is beneficial for the recommendation system that expects high user interaction and network I/O operations.
* **Neo4J and MongoDB**: A dual-database approach is justified as it leverages the strengths of both databases. Neo4J excels in managing and querying connected data, whereas MongoDB is efficient for storing and retrieving JSON-like document data.
* **Kafka Streams and Apache Spark Streaming**: Kafka Streams handles real-time data flow within the Kafka ecosystem, while Apache Spark Streaming performs computationally intensive sentiment analysis, taking advantage of Spark's advanced analytics capabilities.

1. **Project Implementation**

**Frontend Implementation (React.js)**

* + **Dependencies**: React, Redux, React Router, Axios.
  + **Configuration**: package.json includes necessary libraries, scripts for start and build.
  + **Major Code Snippets**: Components for authentication, error handling, and home page views.
  + **Unit tes**: Jest and React Testing Library for component tests.
  + **Integration Tests**: End-to-end tests using Selenium.

The front-end implementation of the system is constructed using React.js and bootstrapped on the local host environment, showcasing a sophisticated and interactive user interface.

Critical dependencies such as React for UI building, Redux for state management, React Router for navigation, and Axios for HTTP requests are integrated into the project, with configurations and scripts managed within the package.json file.

Essential parts of the application, like user authentication, error handling, and the main home page view, are developed as reusable React components, ensuring a modular and maintainable codebase.

For quality assurance, the Jest framework alongside the React Testing Library is utilized to conduct unit tests that validate the functionality of individual components. Additionally, Selenium is employed for comprehensive integration tests, ensuring that the end-to-end interactions within the application function are seamlessly prior to deployment.

This blend of technologies and testing strategies guarantees that the application not only meets developmental standards but also delivers a robust and user-centric experience.

**Backend Implementation (Node.js with Express)**

* + **Dependencies**: Express, Mongoose, neo4j-driver, bcryptjs, jsonwebtoken, kafkajs, sentiment.
  + **Configuration**: config.js file for environment variables, like database URIs, secret keys.
  + **Major Code Snippets**: RESTful API routes for user operations, tweet CRUD operations, sentiment analysis.
  + **Unit Tests**: Using Mocha/Chai for testing API endpoints.
  + **Integration Tests**: Simulating Kafka streams and database integrations.

The backend of the application is architected using Node.js coupled with the Express framework, providing a robust server-side solution bootstrapped on localhost.

The implementation is streamlined through a collection of crucial dependencies including Express for routing, Mongoose for MongoDB interactions, neo4j-driver for Neo4j database connectivity, bcryptjs and jsonwebtoken for authentication, kafkajs for managing Kafka streams, and sentiment for natural language processing.

Configuration for the backend is centralized within a .js file, which securely manages environment variables such as database URIs and secret keys essential for connecting to various services and ensuring secure operations. The server features a suite of RESTful API routes that facilitate user operations, CRUD operations on tweets, and sentiment analysis of user-generated content.

For testing, the Mocha/Chai framework rigorously evaluates the API endpoints to ensure they perform as expected, while integration tests are carried out to simulate the real-time data handling through Kafka streams and to verify the seamless integration with the databases. This well-structured backend setup ensures a strong and reliable foundation for the application, capable of efficiently handling requests and delivering dynamic content.

**Neo4j Database Deployment**

* + **Configuration**: Defining constraints and indexes, setting up user authentication.
  + **Major Code Snippets**: Cypher queries for creating and retrieving tweets.
  + **Unit Tests**: Using the Neo4j testing library to test database operations.
  + **Integration Tests**: Testing the integration with the Node.js backend.

Deployed on the local host for development purposes, the Neo4j database stands as the cornerstone of the application's data layer, particularly tailored to leverage the power of graph-based storage and querying.

This database is meticulously configured with constraints and indexes to ensure data integrity and optimize performance, along with robust user authentication mechanisms to safeguard access.

Utilizing Cypher, Neo4j's declarative query language, the system is equipped with a series of queries that are fundamental to creating and retrieving tweets, forming the backbone of the social platform's interactive capabilities. The database's schema is intricately designed to embody the relationships and properties inherent to social media data, thus enabling complex queries that are crucial for the recommendation engine.

For quality assurance, the Neo4j testing library is employed to execute unit tests, affirming the reliability of database operations. Additionally, integration tests play a vital role in ensuring that the database's integration with the Node.js backend is seamless, facilitating a harmonious interplay between the application's various layers.

The choice of Neo4j greatly empowers the system to handle intricate data relationships with agility and precision, making it an exemplary model for contemporary database deployments.

**MongoDB Database Deployment**

* + **Configuration**: Schema definitions in Mongoose, database connection setup.
  + **Major Code Snippets**: Mongoose models for user and tweet data.
  + **Unit Tests**: Tests for schema validation and database operations.
  + **Integration Tests**: Tests with mocked database connections.

Implemented on a cloud, the MongoDB database deployment forms a vital part of the application’s persistent storage mechanism. It's configured to provide a flexible schema via Mongoose, an Object Data Modeling (ODM) library for MongoDB and Node.js, which simplifies the process of writing MongoDB validation, casting, and business logic boilerplate. The Mongoose schema definitions facilitate the creation of structured models for users and tweets, which dictate the shape and organization of data records. Establishing a reliable database connection is a pivotal step in this setup, ensuring seamless communication between the application server and the database.

Major code snippets consist of Mongoose models that define the essential fields and data types for user profiles and tweet information, enforcing structure in the otherwise schema-less environment of MongoDB. These models serve as blueprints for creating and querying documents within the database, enabling CRUD operations that are integral to the app's functionality.

For quality assurance, unit tests are written and run to confirm that each schema correctly validates data and that all database operations, such as insertions, updates, and deletions, perform as intended. These tests ensure that the data layer is robust and that the application logic relying on these models is reliable.

Furthermore, integration tests are conducted to validate the database's interaction with the application's backend logic. Mocked database connections are utilized within these tests to replicate the behavior of the database, allowing the testing environment to simulate real-world scenarios without the need for an active MongoDB instance. This approach guarantees that the database interactions will function correctly when deployed in a live environment, reinforcing the reliability and stability of the system's data persistence capabilities.

**Kafka Streaming Setup**

* + **Configuration**: Setting up Kafka brokers, defining topics and partitions.
  + **Major Code Snippets**: Producer and consumer functions for handling tweet streams and sentiment analysis results.
  + **Unit Tests**: Using Kafka's mock functions for testing producers and consumers.
  + **Integration Tests**: Verifying the end-to-end data flow from producers to consumers and the processing of streams.

**Unit and Integration Testing**

* + **Unit Tests**: Individual tests for each function, component, and service.
  + **Integration Tests**: Combining multiple units and testing the application flow from the frontend through the backend to the databases and back.
  + **Testing Frameworks**: Mocha/Chai for backend, Neo4j testing library, KafkaJS for streaming.

**Test Dataset Preparation**

* + **Scripts**: Writing scripts using Faker.js or similar to generate realistic user and tweet data.
  + **Seeding Databases**: Creating seed scripts for Neo4j and MongoDB to populate the databases with the test data.
  + **Validation**: Ensuring the seed data conforms to the defined schemas and constraints.

1. **Implementation limitation**

**Scalability Limits**

* + The current deployment on a single server does not handle the significant scaling needed for production-level user loads.
  + Additional configuration and resources would be necessary to achieve true horizontal scalability.

**Performance Constraints**

* + The sentiment analysis process, while real-time, might introduce latency if the volume of incoming data surpasses the processing power of the streaming service.
  + Neo4j performance may be affected by complex graph queries that require extensive traversal, especially as the dataset grows.

**Data Consistency**

* + The eventual consistency model used in some NoSQL databases and distributed systems could lead to temporary data discrepancies, which might affect recommendation accuracy.

**Fault Tolerance**

* + While efforts have been made to implement fault tolerance, the system might still have single points of failure, especially in the Kafka streaming setup or the web application's backend.

**Security Concerns**

* + The current implementation may have security gaps in data transmission, storage, or processing, especially if encryption is not thoroughly applied.
  + User authentication, although secured by JWT, could be more robust with two-factor authentication or other advanced security measures.

**Complexity in Data Processing**

* + The integration of multiple data sources and processing layers adds complexity to the system, which may hinder troubleshooting and maintenance.

**Testing Completeness**

* + The current testing suite may not cover all edge cases or failure scenarios, which could lead to unidentified bugs in production.
  + Integration tests might not simulate the production environment accurately, leading to an overestimation of system stability.

**Dependency on External Services**

* + The system's reliance on third-party services like MongoDB Atlas or cloud-based Kafka solutions might limit control over certain aspects of the system's performance and reliability.

**Development and Deployment Environment**

* + The discrepancy between local development environments and the production environment could lead to unexpected behavior or errors that were not encountered during testing.

**Resource Limitations**

* + Given budgetary constraints, the best possible hosting, monitoring, and management tools may not be utilized, affecting the system's overall quality and reliability.

**Code Quality and Documentation**

* + Time constraints may have led to instances of technical debt, with some parts of the codebase being less clean or well-documented than desired.

**User Experience**

* + The simplicity of the web client's user interface, while sufficient for the initial scope, might not meet all user expectations or compete with existing, mature platforms.

1. **Project Demo & Codebase**

To initiate the server, you should first navigate to the server directory by executing: cd .\app\server\.

Subsequently, install the necessary packages with: npm install.

Finally, launch the server by running: npm run start.

Similarly, to get the client up and running, move into the client directory with: cd .\app\client\, install all required packages using: npm install, and then start the client by executing: npm run start.

Alternatively, yarn start can also be used for starting the client :

|  |  |
| --- | --- |
| Component | Steps and Commands |
| Run the App | 1. Navigate to the app's root directory.  2. Install dependencies: npm install.  3. Start the app: npm start. |
| Run the Client (React.js) | 1.Navigate to the client directory.  2.Install dependencies: npm install.  3. Start the client: npm start.  The client will typically run on http://localhost:3000. |
| Run Docker | 1. Install Docker Desktop from Docker's website.  2. Start Docker Desktop application.  3. Verify installation: docker --version.  4. Run Docker containers based on your docker-compose.yml or Dockerfile: docker-compose up or docker build -t my-app . |
| Run Neo4J | 1. Pull the Neo4J Docker image: docker pull neo4j.  2. Run Neo4J container: docker run --name my-neo4j -p7474:7474 -p7687:7687 -d neo4j.  3. Access Neo4J Browser at http://localhost:7474. |
| Run Apache Kafka on Docker | 1. Create a docker-compose.yml file with Kafka and Zookeeper services.  2. Start Kafka using Docker Compose: docker-compose up.  3. Create topics, if needed, with Kafka commands within the Docker container.  4. Use Kafka producers/consumers as per your application requirement. |

|  |  |
| --- | --- |
| Component | Commands |
| Start Zookeeper | .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties |
| Start Kafka Server | .\bin\windows\kafka-server-start.bat .\config\server.properties |
| Create a Kafka Topic | kafka-topics.bat --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic test |
| Send Data to Kafka Topic | kafka-console-producer.bat --broker-list localhost:9092 --topic test |
| Enter Sample Data | {"Name": "John", "Age":"31", "Gender":"Male"}<br>{"Name": "Emma", "Age":"27", "Gender":"Female"}<br>{"Name": "Ronald", "Age":"17", "Gender":"Male"} |
| Read from Kafka Topic | kafka-console-consumer.bat --topic test --bootstrap-server localhost:9092 --from-beginning |
| Stop Kafka Server | .\bin\windows\kafka-server-stop.bat .\config\server.properties |
| Stop Zookeeper | .\bin\windows\zookeeper-server-stop.bat .\config\zookeeper.properties |
| Start Docker-based Kafka Services | docker-compose up -d |
| Create Kafka Topic within Docker | docker-compose exec kafka kafka-topics.sh --create --topic test --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092 |
| List Kafka Topics (Optional) | docker-compose exec kafka kafka-topics.sh --list --bootstrap-server localhost:9092 |
| Shut Down Docker Services | docker-compose down |

**Steps and commands to run Neo4J and its graph database: (Check commands.txt on GitHub)**

**Steps and commands to run Apache Kafka on Docker: (Check commands.txt on GitHub)**

**Steps and commands to test synchronization between Neo4J, The front-end, Back-end, MongoDB and(Check commands.txt on GitHub)**

**Steps and commands to run Queries in Neo4J and MangoDB and check the synchronization & Integration (Check commands.txt on GitHub)**

**Steps and commands to test using Unit Testing & Integration Testing: (Check commands.txt on GitHub)**

|  |  |
| --- | --- |
| Task | Command(s) |
| Run the App (General) | cd path\to\app<br>npm install<br>npm start |
| Run the Client (Front-end) | cd path\to\client<br>npm install<br>npm start |
| Run Docker | docker-compose up (Runs all services defined in docker-compose.yml) |
| Run Neo4J Database | docker run --name neo4j -p7474:7474 -p7687:7687 -d neo4j |
| Run Apache Kafka on Docker | docker-compose up (with Kafka services defined in docker-compose.yml) |
| Test Synchronization (Full Stack) | Synchronization tests will generally be custom scripts that make API calls and verify responses against the database. No one-liner exists for this. |
| Run Queries in Neo4J | Open Neo4j Browser at http://localhost:7474 and execute Cypher queries directly in the interface. |
| Run Queries in MongoDB | Use MongoDB Compass or shell:<br>mongo<br>use twitterDB<br>db.tweets.find({})<br>db.users.find({}) |
| Test Synchronization & Integration | Similar to synchronization tests, this will involve custom scripts or testing frameworks that run series of operations and check the system’s responses. |
| Unit Testing & Integration Testing | Navigate to the project directory:<br>cd path\to\project<br>npm test<br>  For specific tests, more granular commands may be specified within the package.json scripts. |

**The results:**

**The front-end**

A screenshot of a computer

Description automatically generated

**The back-end in MongoDB Cloud:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Front-end (React.js):**

* The React.js front-end presents a clean interface, as seen in the first image, where users can sign in and post tweets.
* The page provides a text input field for composing tweets, and below it, a display of tweets from various users is visible. This display is likely updated in real-time as new tweets are posted.
* The interaction with the back-end is managed by Axios, which sends HTTP requests to API endpoints defined in the Node.js back-end. Actions like posting tweets or retrieving tweet recommendations trigger these API calls.
* Redux manages the application state, which includes the authentication state, the tweets to display, and the data returned from the recommendation system.

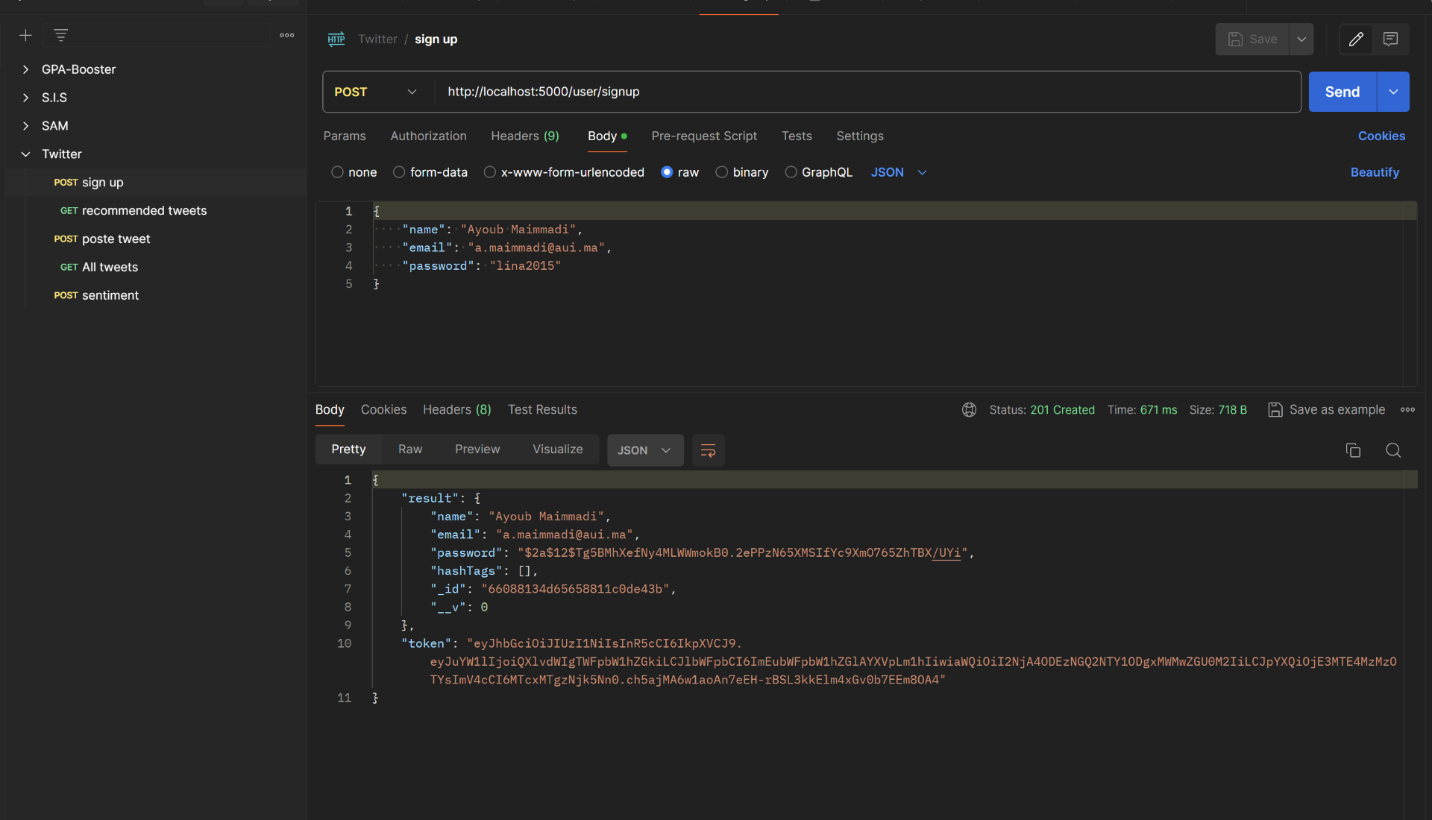
**Back-end (Node.js with Express):**

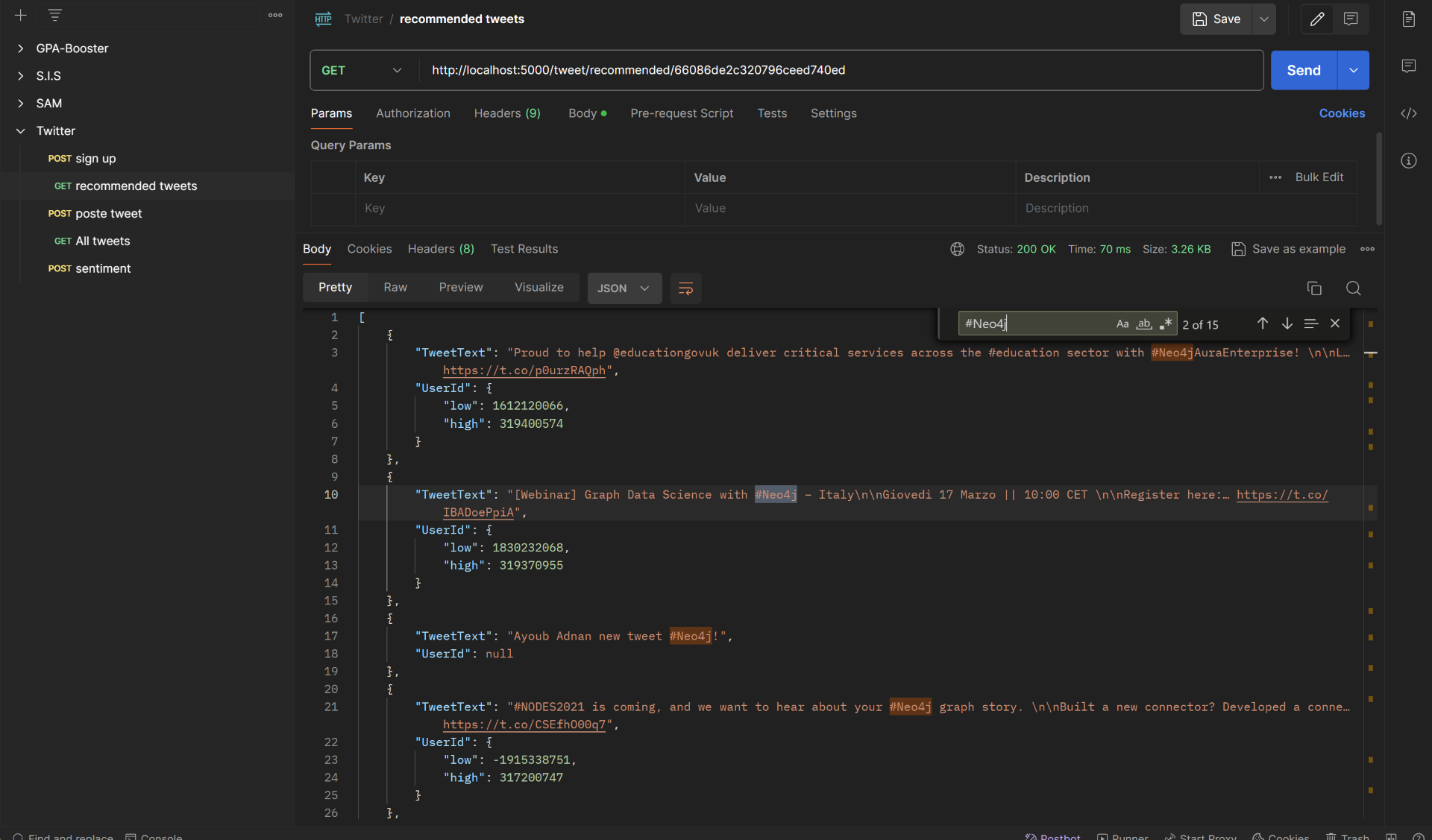
* The Node.js/Express back-end handles API requests from the front-end. It processes actions such as user authentication, tweet posting, fetching tweets, and generating recommendations.
* The back-end uses Mongoose models to interact with MongoDB, where user data and tweets are stored, as suggested by the MongoDB database screenshots. The tweets collection likely stores tweet content, while the users collection holds user profiles and authentication data.
* When a tweet is posted, the back-end saves it to the database and may also publish it to a Kafka topic for real-time processing or further analysis, such as sentiment analysis.
* The Neo4j graph database is employed for recommendation generation. The neo4j-driver is used to run Cypher queries against the Neo4j instance to determine connections and suggest relevant users or hashtags.

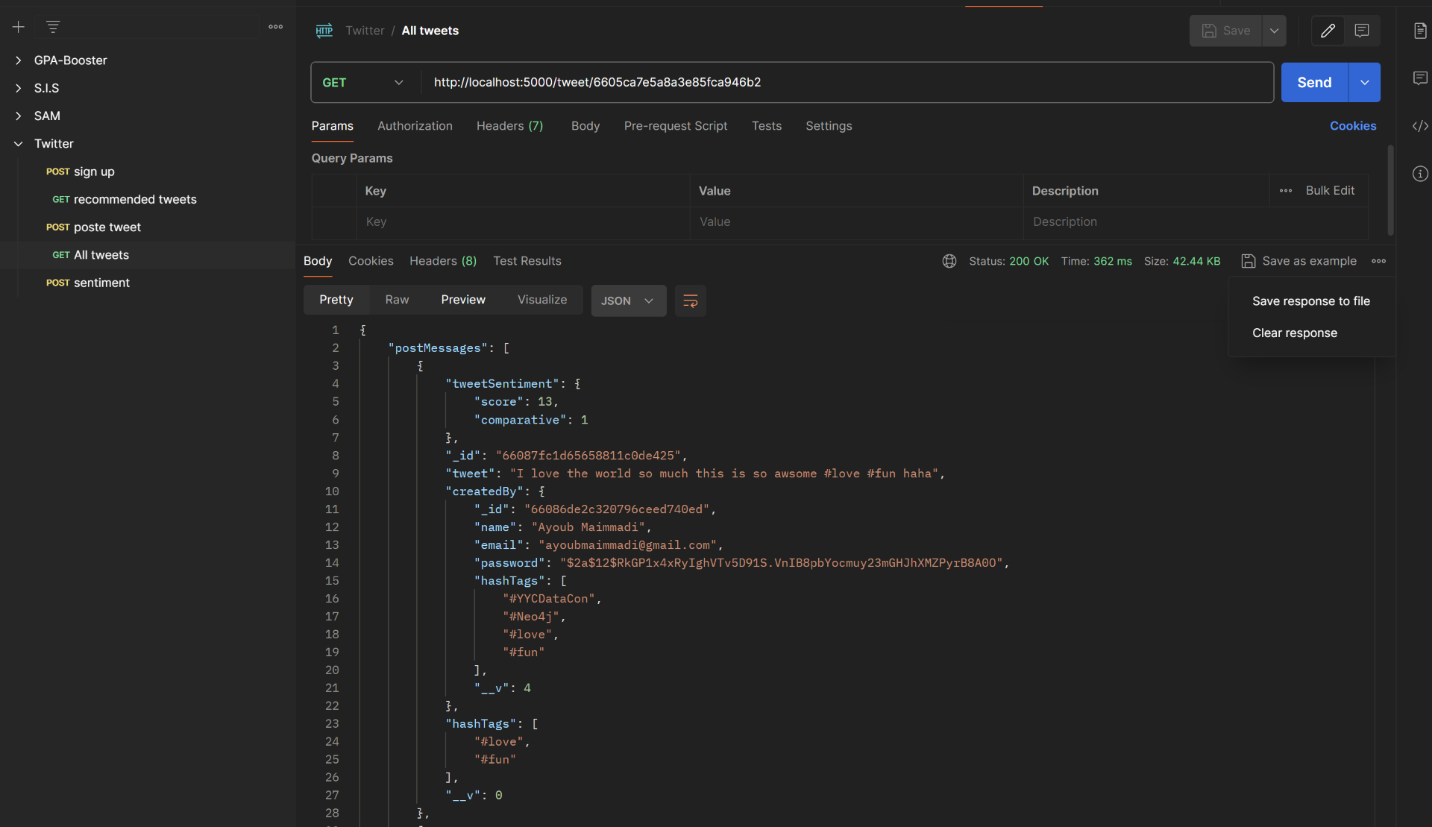
**Microservices:**

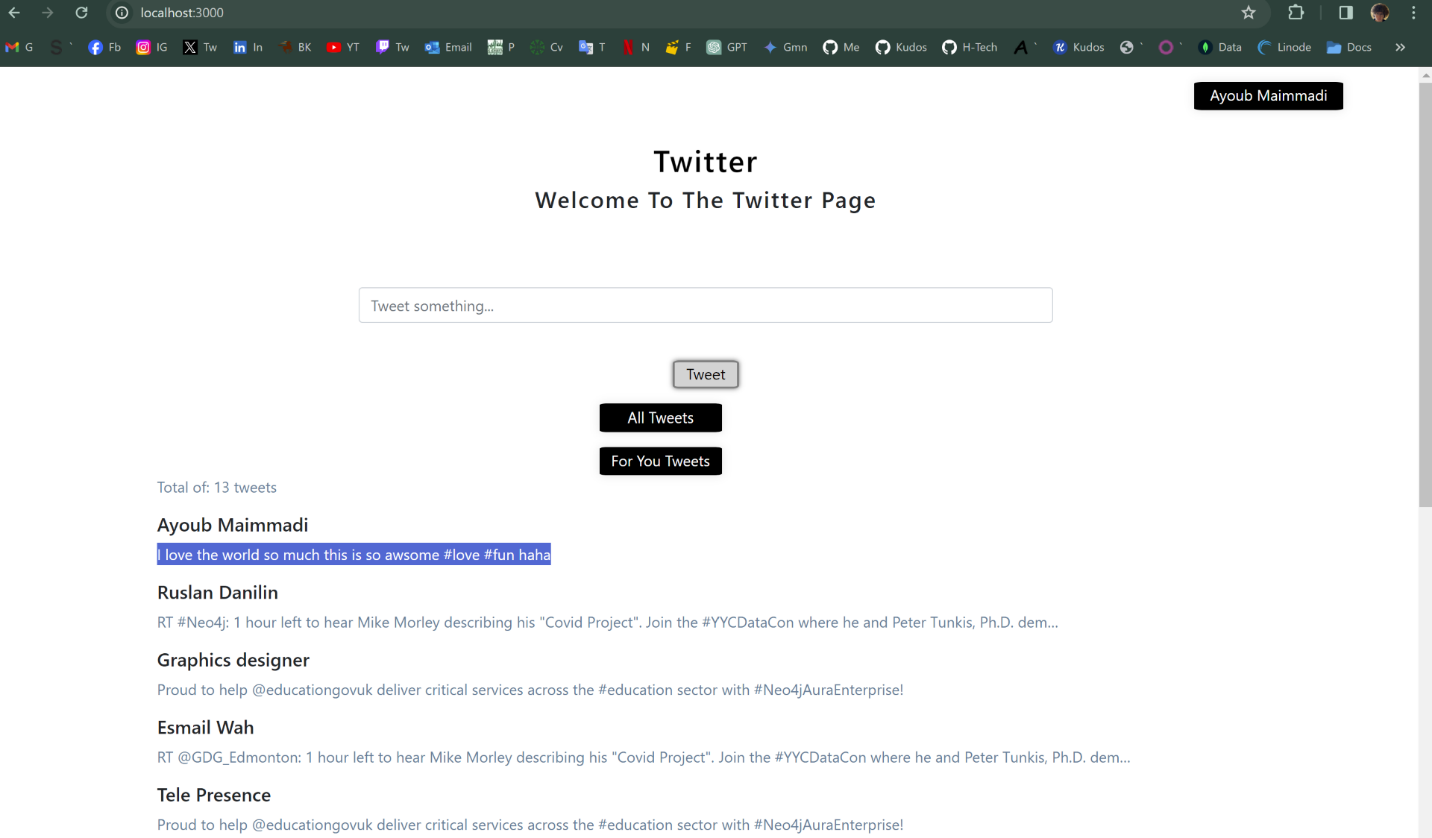
* Microservices architecture enables each back-end service (e.g., user authentication, tweet handling, recommendation generation) to operate independently and scale as needed.
* The Kafka service acts as a messaging layer that can decouple the services. It allows for asynchronous data processing and can enable the real-time flow of tweet data for processing by different microservices.
* Neo4j serves as the microservice for graph-related operations. It stores and processes graph-based relationships to provide recommendations. These operations are exposed through the API to the front-end.
* Unit tests ensure individual microservices work correctly in isolation, while integration tests confirm that they function together as expected, ensuring end-to-end functionality of the entire application.

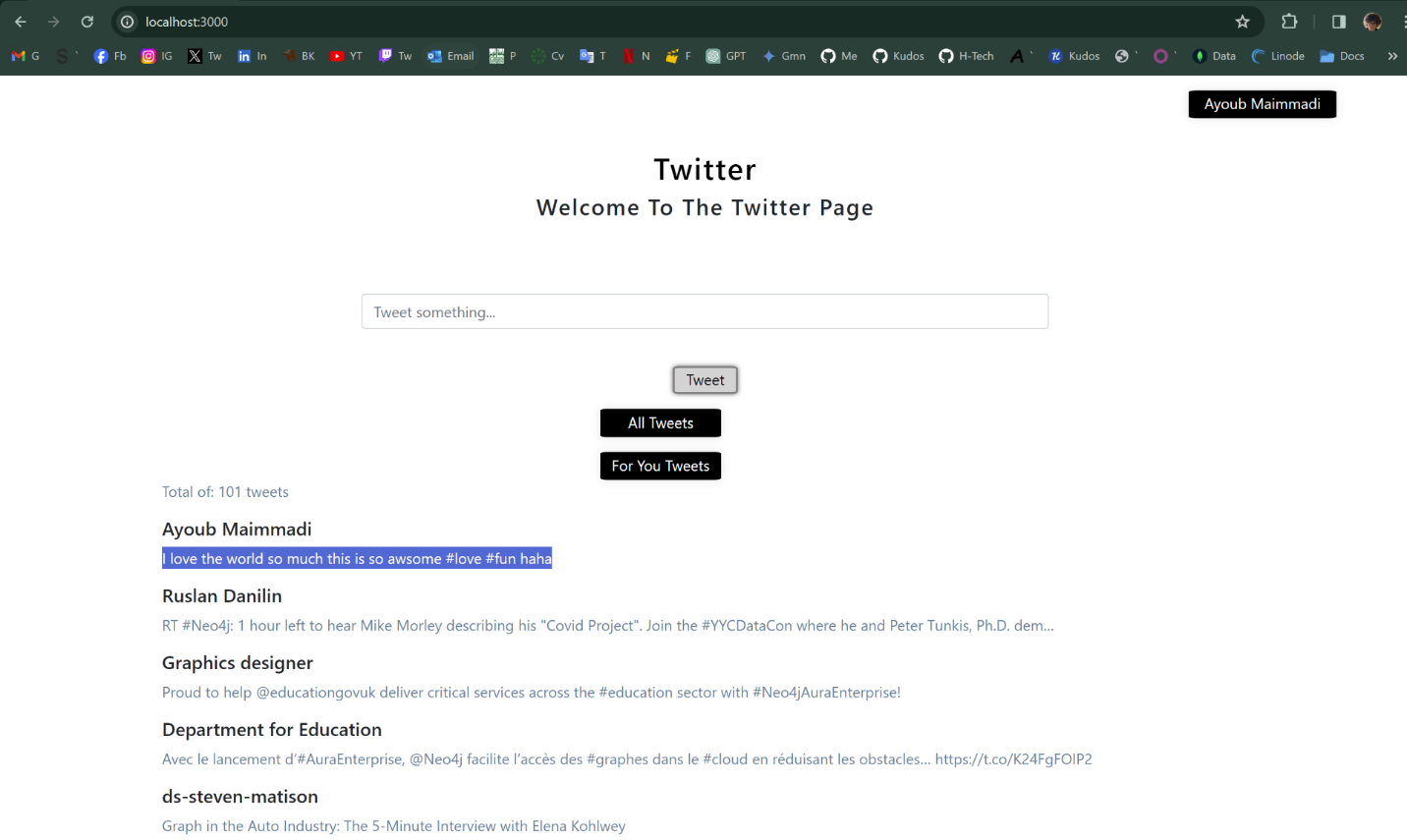
**The usage of Postman API (CRUD Operations etc…):**

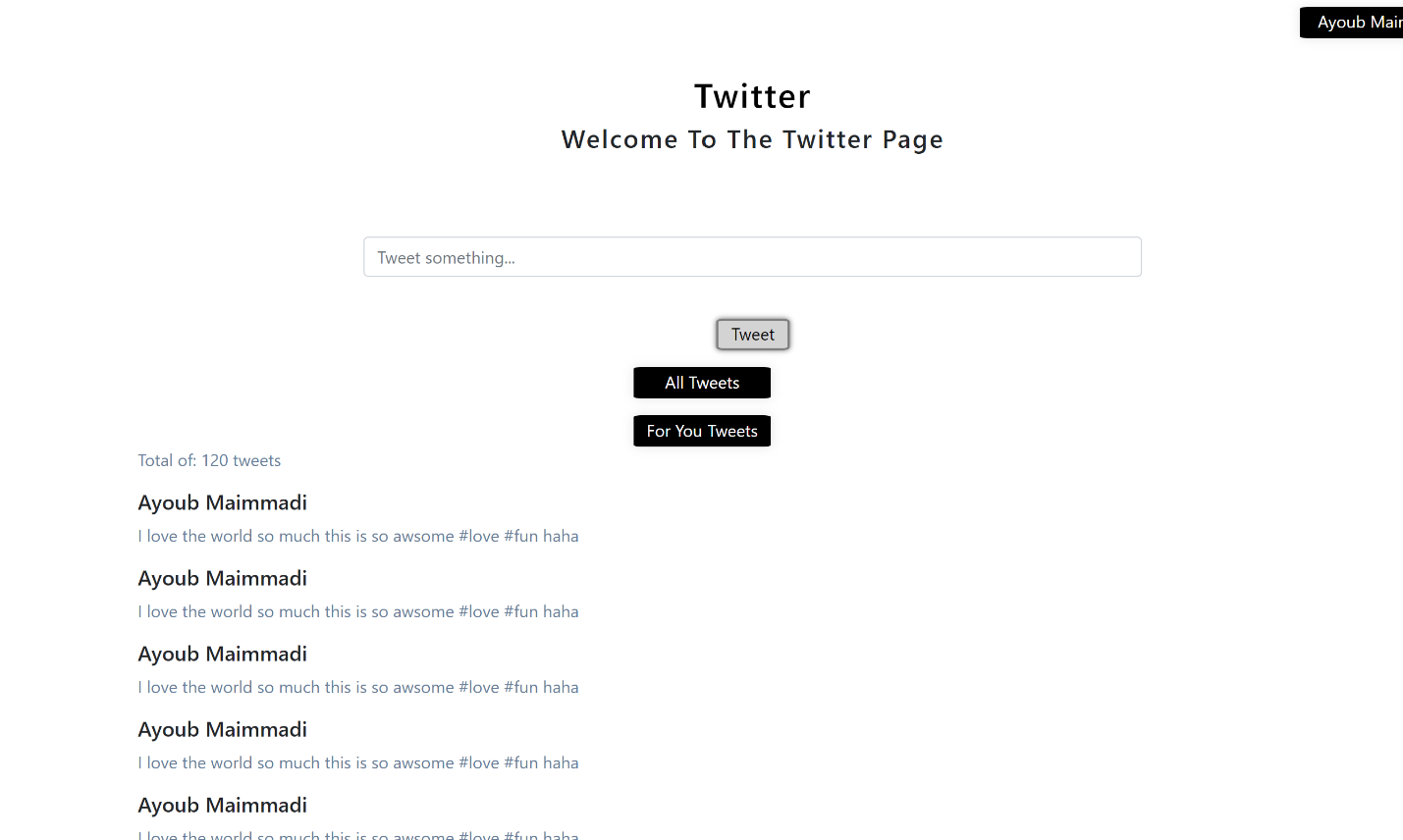






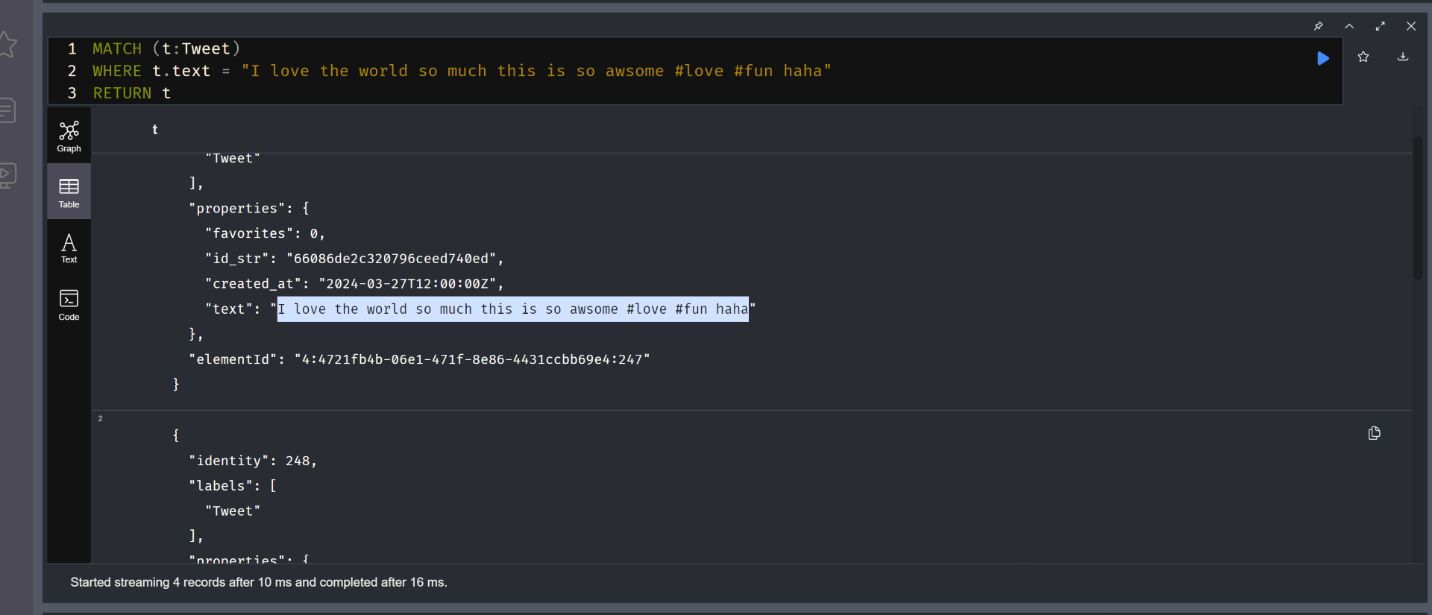


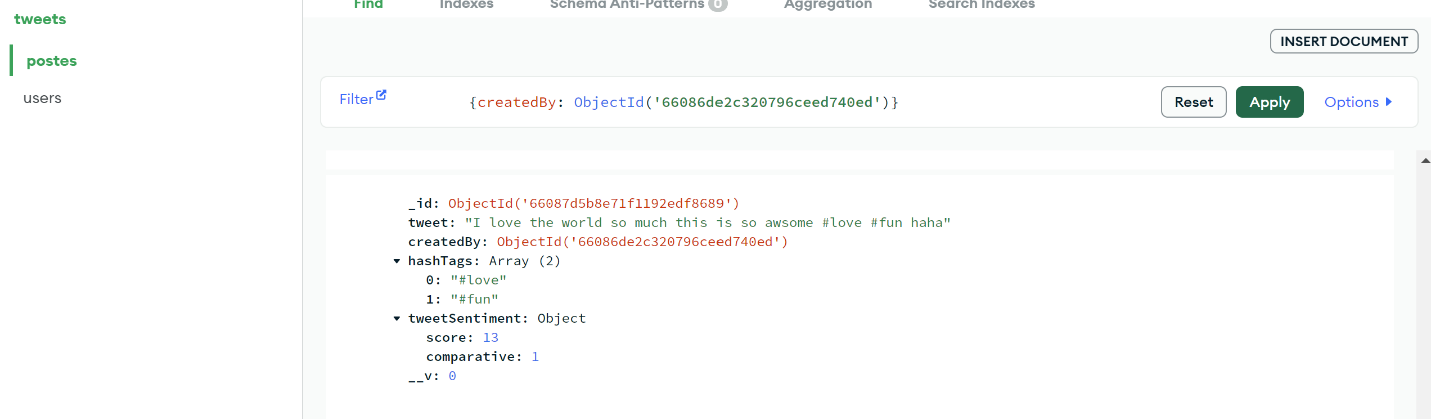


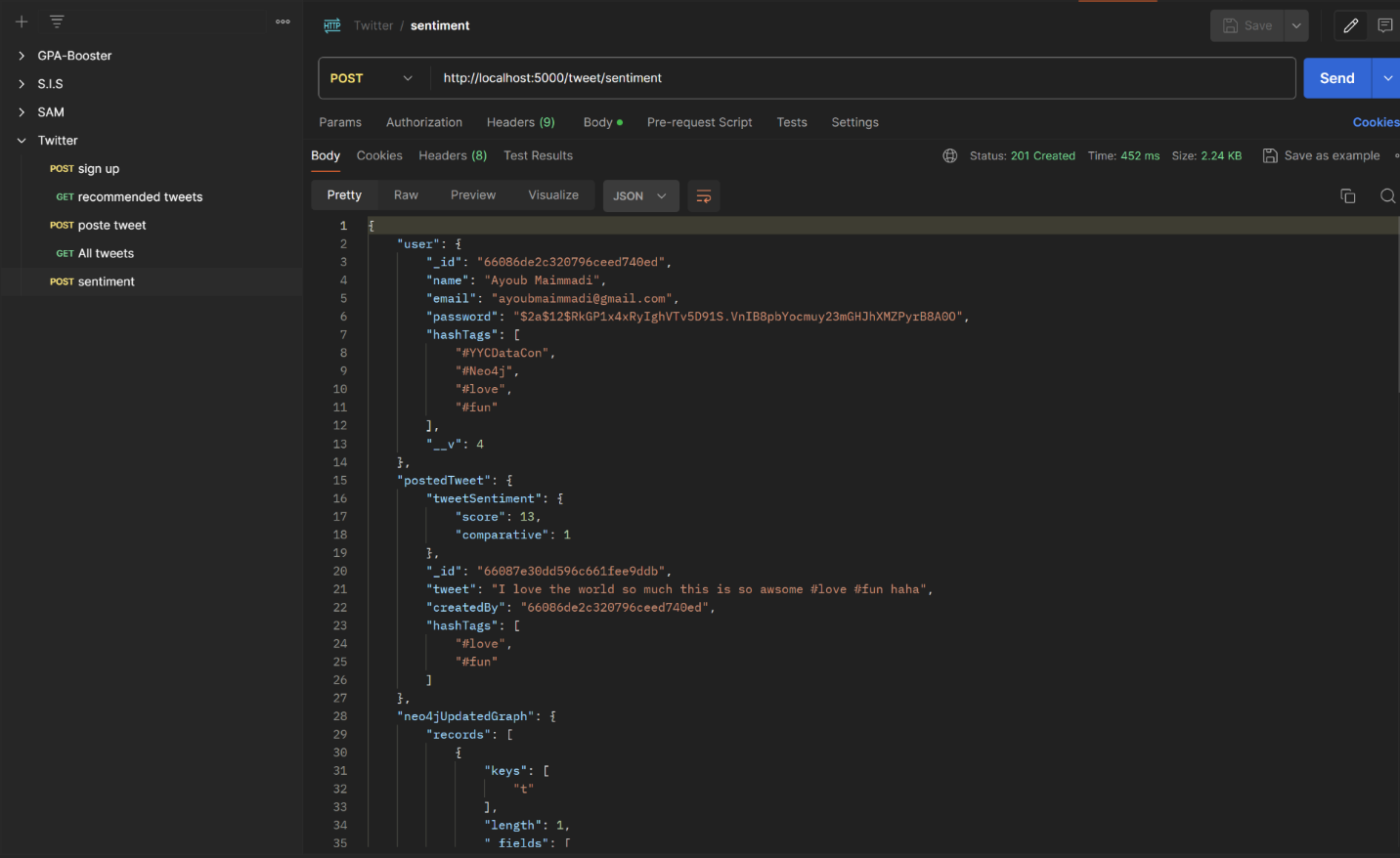


1. **Postman API Testing**:
   * The first image showcases a Postman request to sign up a new user, where the JSON payload includes name, email, and password. The response shows a successful user creation with a returned JWT token for authentication.
   * The second image is a GET request to the /recommendedtweets endpoint, which fetches tweets based on user preferences or behavior. The response includes an array of tweets that meet the recommendation criteria.
   * The third image represents a GET request fetching all tweets from the /alltweets endpoint, with the server's response containing tweet details and metadata like user ID, sentiment scores, and hashtags.
2. **Frontend Synchronization**:
   * The fourth and fifth images display the frontend interface hosted on localhost. This page allowed us to compose and submit new tweets and provides tabs to view either all tweets or personalized "For You" tweets.
   * The front-end interface communicates with the backend services through HTTP requests, which is where Axios and other dependencies play a role, as mentioned in the project descriptions.
   * The UI is responsive and minimalistic, showing tweets in a linear feed format. The "Tweet something..." input field indicates an interactive element for users to post new content.
3. **Backend Synchronization**:
   * The backend, built with Node.js and Express, handles the API requests received from the frontend. The user authentication, tweet posting, and retrieval of tweets are managed by the server and reflected in the frontend interface.
   * Data consistency between the frontend and backend is maintained by using API calls for all user interactions with the server, ensuring the UI displays the most updated data available from the backend.
4. **Microservices and Database Integration**:
   * The application may employ a microservices architecture, where services like user authentication, tweet management, and recommendation generation operate independently.
   * Databases (presumably MongoDB for user data and Neo4j for graph-based data modeling) are interacted with through defined schemas and database queries, with Neo4j possibly providing graph-based recommendations and MongoDB handling user profiles and tweet storage.

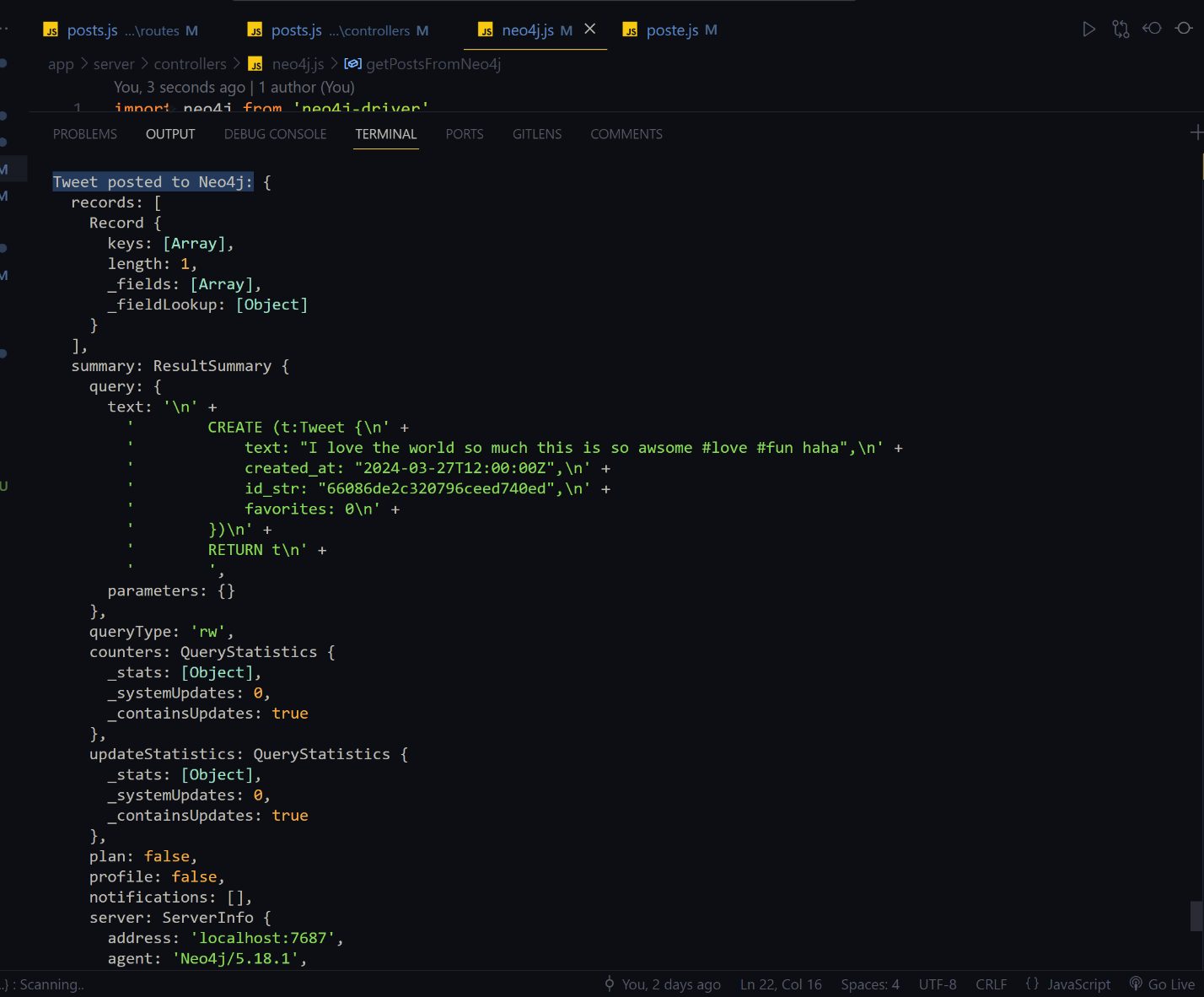
The flow depicted across these images indicates a streamlined process of API testing with Postman, synchronous interaction between the frontend and backend, and the use of microservices architecture for handling complex application logic and data. The setup ensures that new tweets and user actions are reflected in real-time, both in the application’s interface and the underlying databases.

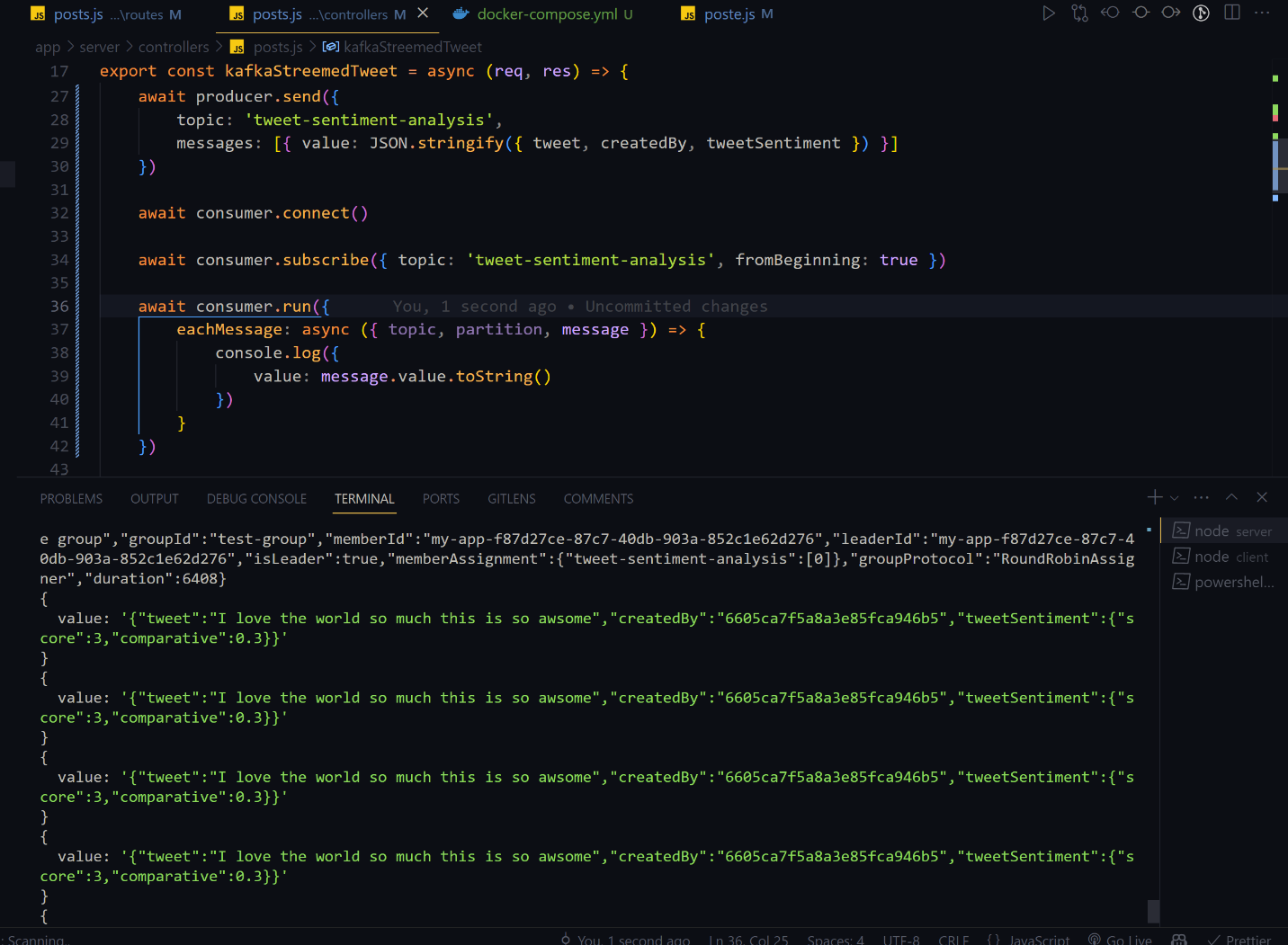












**Sentiment Analysis with Kafka**:

* + The application uses Kafka to handle streaming tweets for real-time sentiment analysis. A producer would send new tweets to a Kafka topic, and a consumer subscribed to this topic would receive these messages and perform sentiment analysis, which can be done using an NLP library like sentiment.
  + **Kafka Integration**: Kafka consumer is set up within a Node.js environment. Kafka consumers subscribe to topics and process the messages that come through those topics.
  + **Sentiment Analysis Trigger**: Upon receiving a new tweet through the Kafka stream, the consumer might be running sentiment analysis on the content of the tweet. This typically involves parsing the text of the tweet and applying a sentiment analysis algorithm, often through a dedicated library.
  + **JSON Processing**: The message from the Kafka topic is being stringified into JSON format, which suggests that the tweet data is being serialized for transmission or processing, and the sentiment analysis results are likely to be attached to each tweet object.
  + **Logging**: The console.log statement within the eachMessage function indicates that the server is logging the details of each message it consumes. This is useful for debugging and monitoring purposes.
  + **Error Handling**: Although not explicitly shown in your description, proper error handling would be essential in a production environment to manage any issues with message consumption or sentiment analysis processing.
  + **Real-time Analysis and Response**: The overall purpose of this code is to allow for real-time sentiment analysis of tweets as they are posted. The results could be used for various features in the application, such as personalizing content, moderating discussions, or providing insights into public sentiment.

**Neo4j for Graph Data**:

In the context of a graph-based recommendation system, Neo4j stores relationships between entities, such as users and tweets.

Cypher queries are used to interact with this data, retrieving and manipulating the graph for operations like generating recommendations based on user interactions and sentiment scores.

**MongoDB for Document Storage**:

MongoDB is typically used to store user profiles and tweet content in a document format.

Mongoose ODM facilitates interactions with MongoDB, allowing for operations like creating and retrieving tweets and user information.

It also stores the results of the sentiment analysis associated with each tweet.

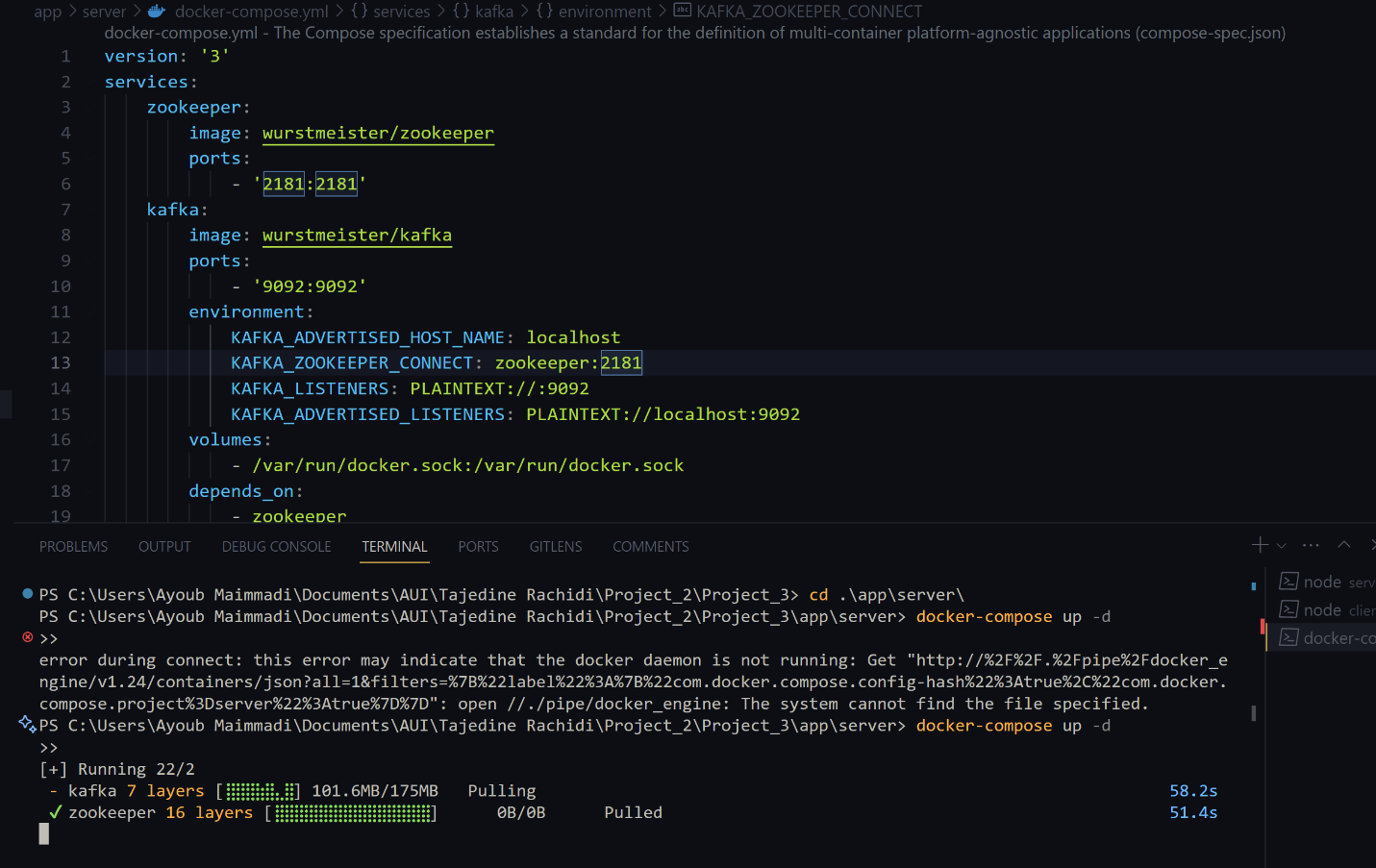
**Node.js Backend**:

The backend built with Node.js and Express orchestrates the overall process, handling HTTP requests from the client, interacting with Kafka for sentiment analysis, and using Mongoose and Neo4j drivers for database operations.

**Testing and Synchronization**:

Unit testing ensures individual components, such as database models and API routes, work correctly in isolation.

Integration testing checks the overall system's synchronization, ensuring that the frontend reflects the correct state of the backend and databases after operations like posting tweets and performing sentiment analysis.



The content of the docker-compose.yml file includes configurations for two services:

zookeeper and kafka : These services are essential components for running Apache Kafka, which is a distributed streaming platform commonly used for building real-time data pipelines and streaming applications.

Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It's a requisite for running Kafka, which relies on Zookeeper for cluster management. The port 2181, the default for Zookeeper, is exposed, suggesting it's accessible on the host machine.

The Kafka service is configured with environment variables like: KAFKA\_ADVERTISED\_HOST\_NAME set to localhost, meaning it's set up to run on the local machine. It also has KAFKA\_ZOOKEEPER\_CONNECT configured to link with the Zookeeper service.

The Kafka broker is exposed on port 9092, which is the default port for Kafka.

In the terminal, we see an initial attempt to run docker-compose up -d failed due to an error indicating that the Docker daemon is not running. This error is a common issue when the Docker desktop application isn't running or hasn't been started correctly. We have resolved the issue and rerun the docker-compose up -d command successfully, as indicated by the output showing the Zookeeper and Kafka images being pulled from Docker Hub.

The purpose of these actions is to set up a local Kafka environment, which could be used for development and testing of applications that require messaging or stream processing capabilities.

The -d flag in the docker-compose up -d command suggests that the services are intended to be run in the background (detached mode).

This setup is typical in microservices architecture, where Kafka might be used to handle event-driven data or messages passing between different services in a decoupled manner. The IDE interface and open docker-compose.yml file suggest that the user is likely developing or testing an application that integrates with Kafka.

**Testing part using Mocha/Chai :**

**Unit & Integration testing:**

A screenshot of a computer

Description automatically generated

1. **References**

* React Documentation. (n.d.). React: A JavaScript library for building user interfaces. Retrieved from https://reactjs.org/docs/getting-started.html
* Node.js Project. (n.d.). Node.js. Retrieved from https://nodejs.org/en/
* Docker Documentation. (n.d.). Overview of Docker Compose. Retrieved from https://docs.docker.com/compose/
* Neo4j Documentation. (n.d.). Neo4j Graph Database Platform. Retrieved from https://neo4j.com/docs/
* Apache Kafka. (n.d.). Apache Kafka: A Distributed Streaming Platform. Retrieved from <https://kafka.apache.org/intro>
* MongoDB Documentation. (n.d.). MongoDB. Retrieved from <https://docs.mongodb.com/>
* npm. (n.d.). npm: Build amazing things. Retrieved from <https://www.npmjs.com/>
* Yarn Package Manager. (n.d.). Yarn. Retrieved from <https://yarnpkg.com/>
* Redux Documentation. (n.d.). Redux: A Predictable State Container for JS Apps. Retrieved from https://redux.js.org/introduction/getting-started
* Axios on npm. (n.d.). Promise based HTTP client for the browser and node.js. Retrieved from <https://www.npmjs.com/package/axios>
* SemVer. (n.d.). Semantic Versioning 2.0.0. Retrieved from <http://semver.org/>
* GitHub. (n.d.). Build software better, together. Retrieved from <https://github.com/>