Project#2 Report

Twitter Application

By: Adnane Kesraoui & Yassine Maatougui

Adnane Kesraoui	• Implementation of Flask APIs for integration with MongoDB for user authentication and tweet storage. • Integration with Neo4j Graph DB for followers' recommendations. • Integration with Kafka for tweet live processing. • Implementation of sentiment analysis of tweets using "TextBlob" library and online stream processing with Kafka.
Yassine Maatougui	• **Frontend-Backend Integration**: Bridged the gap between the frontend and backend, allowing for seamless data flow and user experience. • **Troubleshooting and Debugging**: Debugging complex issues during development, significantly reducing downtime and improving system resilience. • **Frontend Development**: Developing front-end interface using HTML, CSS, and JavaScript, focusing on user interaction and visual design. • **Dependency Installation and Configuration in cloud server:** Conducted comprehensive installation and configuration of key dependencies such as Flask for the server backend, MongoDB for the database layer, and Neo4j for graph database functionalities.

Project Definition

Objective:

The goal of this project is to build a twitter-like application which will allow its users to post tweets and get followers recommendations based on the contents of their tweets. At the same time, the application will update the appropriate graph database which holds the user's information and his tweets and then apply sentiment analysis to the tweet to extract its sentiment and store that in the graph database as well.

Functional requirements:

- **Personalized Recommendations:** The user must provide the user with personalized recommendations in the form of other users to follow based on their interactions through tweets.
- **Graph-Based Recommendations:** The system should make use of graph-based data structure to output recommendations.
- **Real-time Sentiment Analysis:** The system should stream tweets for real-time NLP to further improve the recommendations.
- **Graph Database Updates:** The system should update the graph database with the user's inputs accordingly.

Non-Functional requirements:

- **Scalability:** It should dynamically incorporate new content and user interactions without the need for manual recalculations or downtime.
- **Performance:** The system should process and analyze tweets in near real-time, providing timely and relevant recommendations to users.
- **Deployment and Operational Efficiency:** The system should be easy to deploy, with considerations for containerization or virtualization to facilitate consistent environments across development, testing, and production.
- **High Availability and Reliability:** High availability of the recommendation service, ensuring it is operational and capable of generating recommendations without significant downtime.

Project Design

Data Models:

- Mongo DB document database for User authentication and tweet storage.
- Neo4j Graph database for relationship storage and recommendations.

Encoding and Querying:

- Cypher queries to update Neo4j.
- Json format to enable communication between different components of the application (Front-end, Back-end, Databases).

Scalability and Reliability:

- Microservice applications to enable scaling different parts of the application independently.
- Use of Kafka for near-line processing of user data.

Configuration:

- Multiple Kafka brokers integrations for high Availability and reliability insurance.

This comprehensive design approach underscores the project's commitment to robust data management, efficient processing, and scalable architecture, ensuring a responsive and reliable user experience.

Project implementation

Deployment:

- Use of Kafka in a containerized environment to ensure portability and use across different systems ensuring reliability:

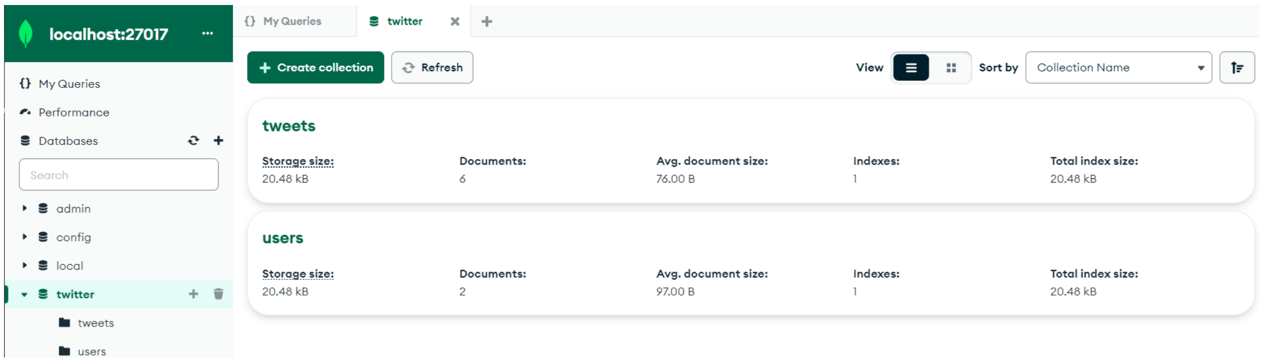
  Get the docker image

  ```
  $ docker pull apache/kafka:3.7.0
  ```

  Start the kafka docker container

  ```
  $ docker run -p 9092:9092 apache/kafka:3.7.0
  ```

- Setting up a mongoDb Local database for development:



tweets

Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
20.48 kB	6	76.00 B	1	20.48 kB

users

Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
20.48 kB	2	97.00 B	1	20.48 kB

- Setting up an already existing neo4j twitter database via a dump file from the following github repo: **https://github.com/neo4j-graph-examples/twitter-v2/tree/main/data**

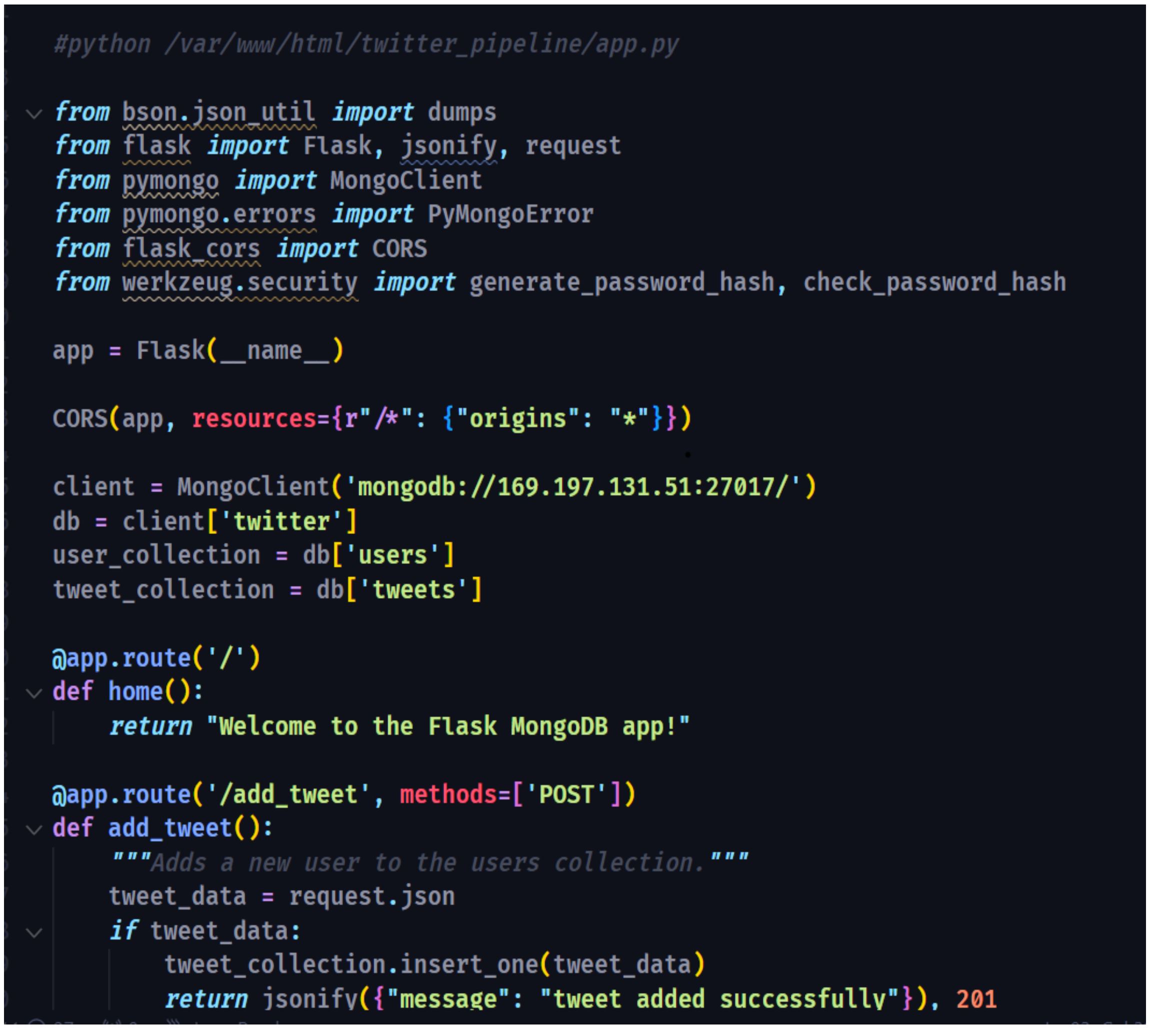
Libraries used:

- **Flask** for the creation of an **API** capable of conversing with the front-end and MongoDB/Neo4j
- **PyMongo** for interaction with MongoDB with python code.
- **Neo4j driver** for neo4j operations.
- **Kafka-python** for messaging.
- **TextBlob** for sentiment analysis

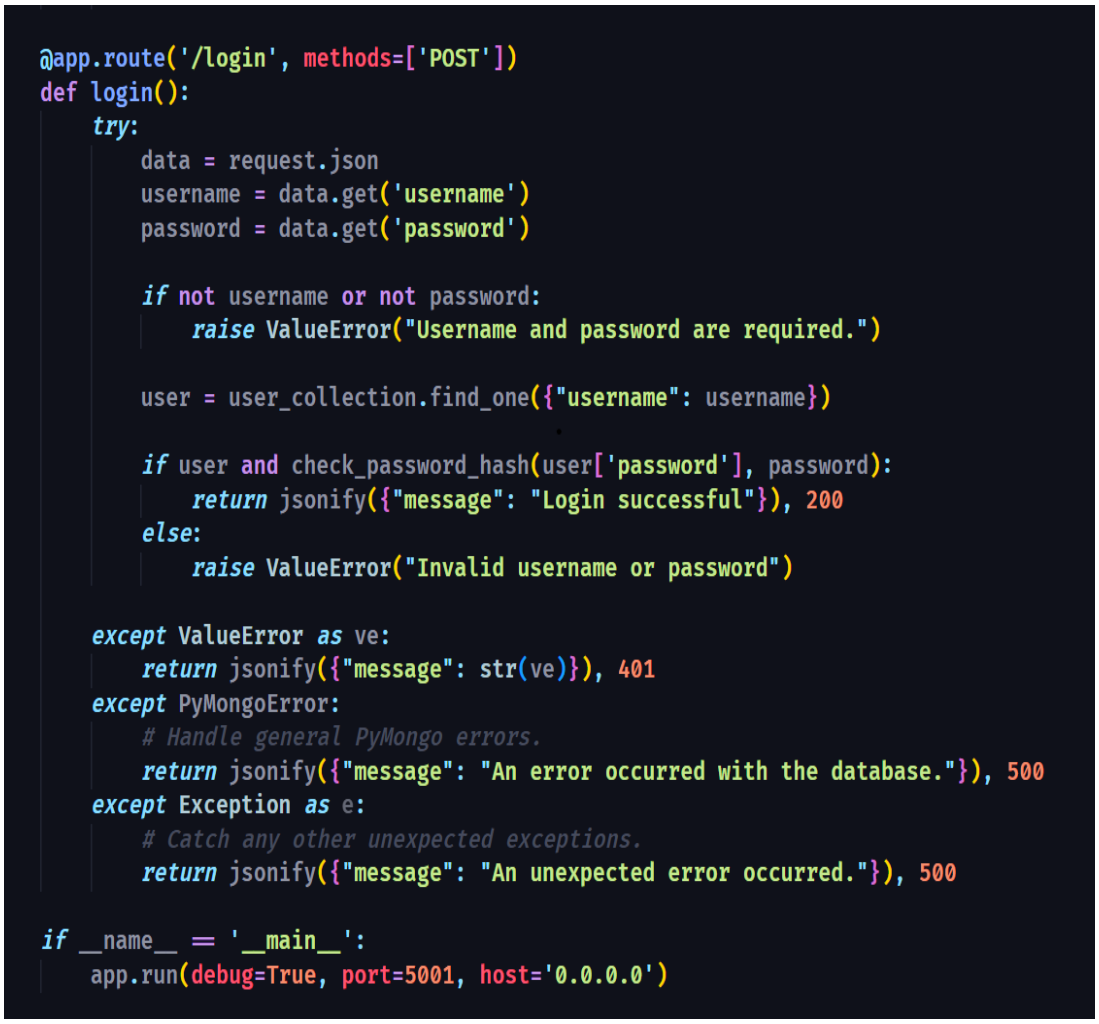
Major code Snippets:

User Authentication and tweets storage:

- This snippet from "**app.py**" showcases the core Flask application setup, emphasizing the API endpoints for user authentication and tweet processing. The code illustrates how Flask routes are defined to handle HTTP requests, integrating seamlessly with MongoDB for data storage.



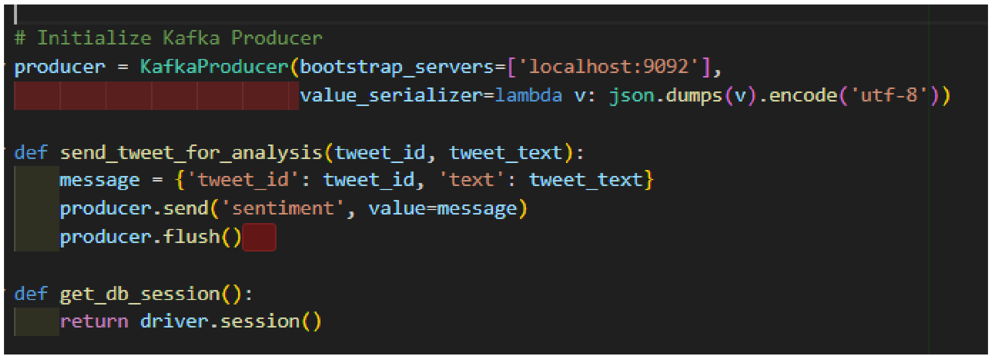
```python
#python /var/www/html/twitter_pipeline/app.py

from bson.json_util import dumps
from flask import Flask, jsonify, request
from pymongo import MongoClient
from pymongo.errors import PyMongoError
from flask_cors import CORS
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__)

CORS(app, resources={r"/*": {"origins": "*"}})

client = MongoClient('mongodb://169.197.131.51:27017/')
db = client['twitter']
user_collection = db['users']
tweet_collection = db['tweets']

@app.route('/')
def home():
    return "Welcome to the Flask MongoDB app!"

@app.route('/add_tweet', methods=['POST'])
def add_tweet():
    """Adds a new user to the users collection."""
    tweet_data = request.json
    if tweet_data:
        tweet_collection.insert_one(tweet_data)
        return jsonify({"message": "tweet added successfully"}), 201
```


```
@app.route('/login', methods=['POST'])
def login():
    try:
        data = request.json
        username = data.get('username')
        password = data.get('password')

        if not username or not password:
            raise ValueError("Username and password are required.")

        user = user_collection.find_one({"username": username})

        if user and check_password_hash(user['password'], password):
            return jsonify({"message": "Login successful"}), 200
        else:
            raise ValueError("Invalid username or password")

    except ValueError as ve:
        return jsonify({"message": str(ve)}), 401
    except PyMongoError:
        # Handle general PyMongo errors.
        return jsonify({"message": "An error occurred with the database."}), 500
    except Exception as e:
        # Catch any other unexpected exceptions.
        return jsonify({"message": "An unexpected error occurred."}), 500

if __name__ == '__main__':
    app.run(debug=True, port=5001, host='0.0.0.0')
```

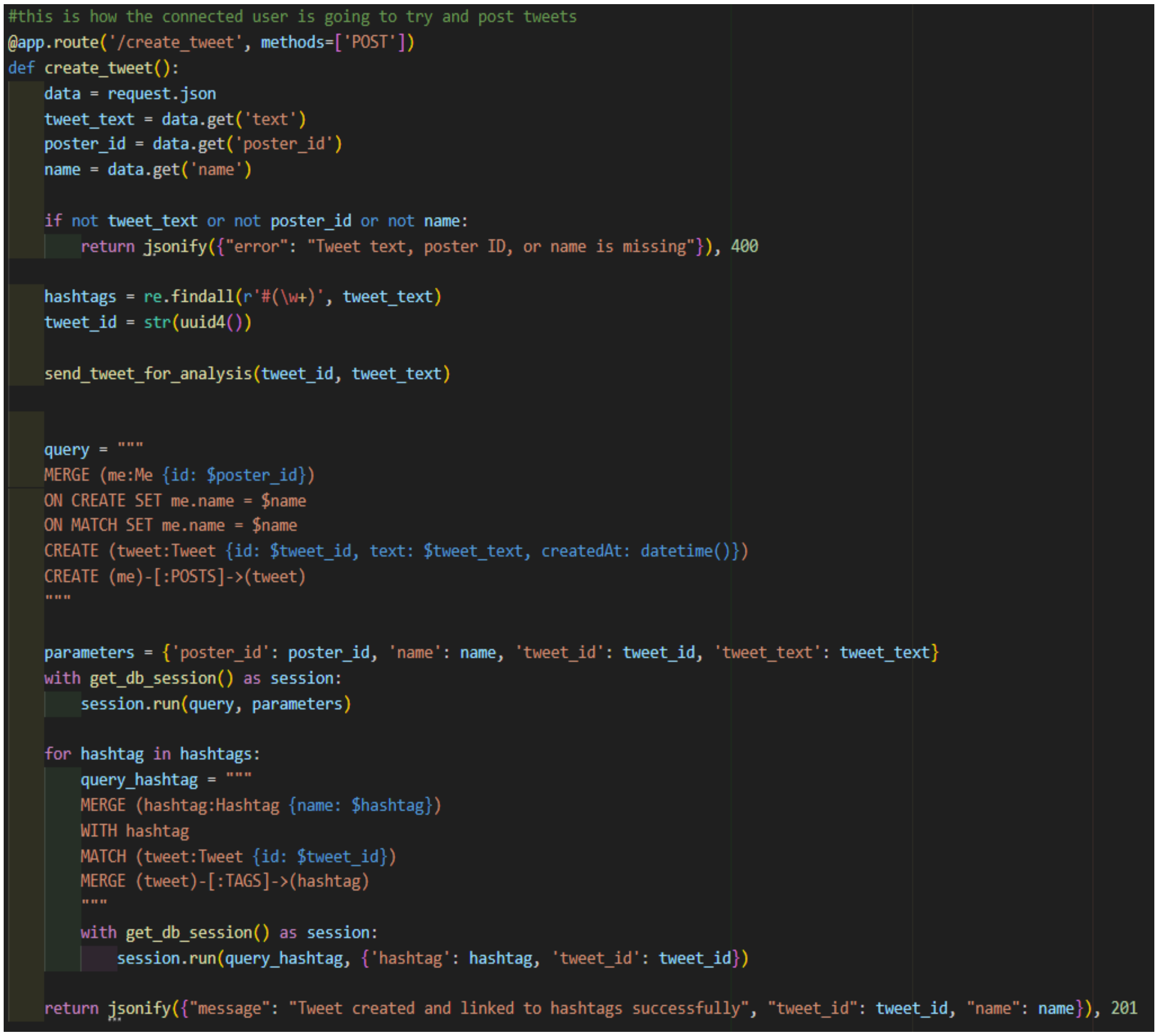
Neo4j Graph update and recommendation response

- In “neo_app.py”, we focus on connecting our front end with neo4j enabling automatic graph updates as well as the automatic stream of tweets directly to Kafka for sentiment processing.

This contains the logic of how the tweet would be submitted to Kafka:

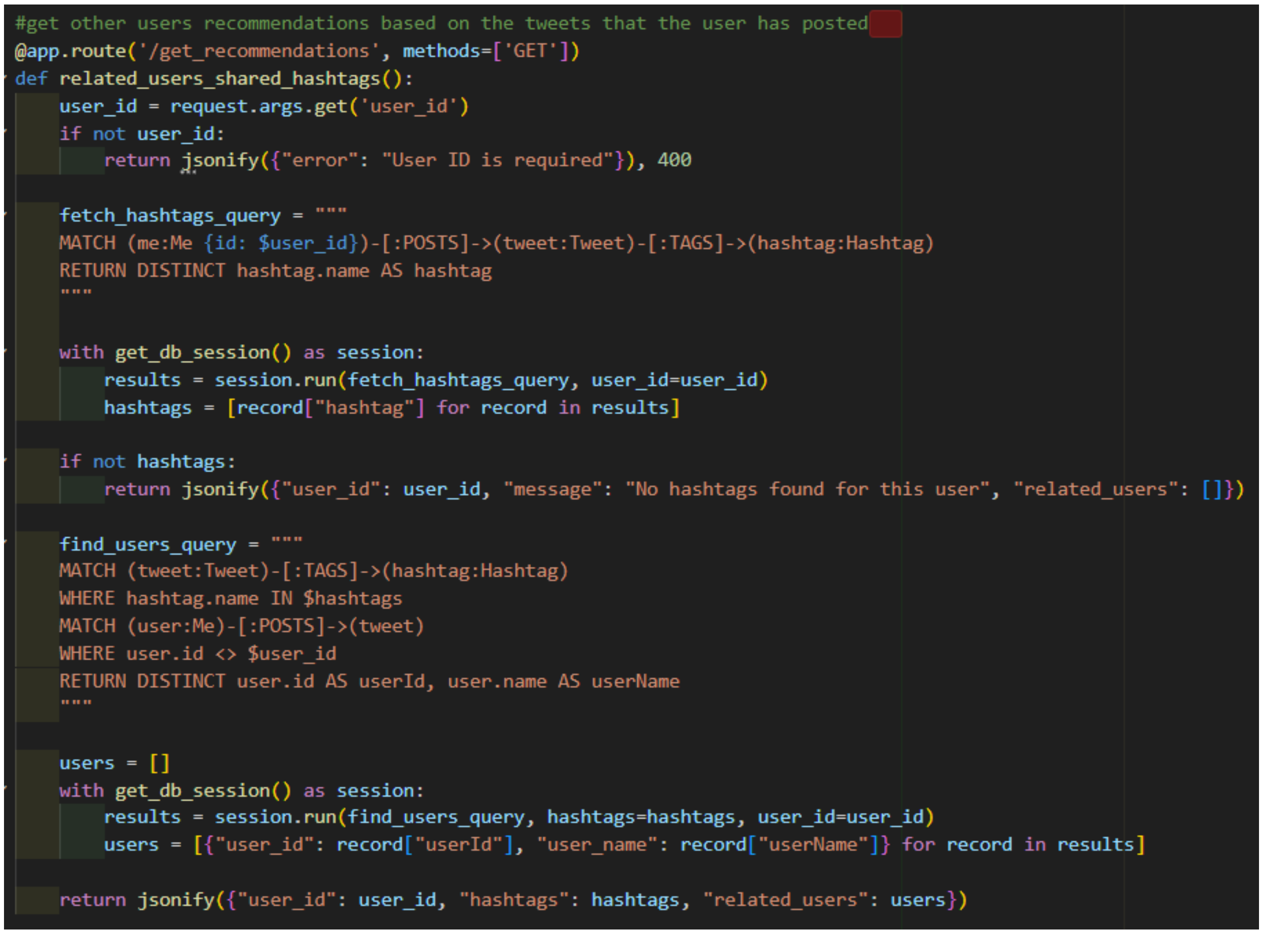

```
# Initialize Kafka Producer
producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
                         value_serializer=lambda v: json.dumps(v).encode('utf-8'))

def send_tweet_for_analysis(tweet_id, tweet_text):
    message = {'tweet_id': tweet_id, 'text': tweet_text}
    producer.send('sentiment', value=message)
    producer.flush()

def get_db_session():
    return driver.session()
```

This contains the logic of how the tweets would be updated in the Neo4j db :



```python
#this is how the connected user is going to try and post tweets
@app.route('/create_tweet', methods=['POST'])
def create_tweet():
    data = request.json
    tweet_text = data.get('text')
    poster_id = data.get('poster_id')
    name = data.get('name')

    if not tweet_text or not poster_id or not name:
        return jsonify({"error": "Tweet text, poster ID, or name is missing"}), 400

    hashtags = re.findall(r'#(\w+)', tweet_text)
    tweet_id = str(uuid4())

    send_tweet_for_analysis(tweet_id, tweet_text)


    query = """
    MERGE (me:Me {id: $poster_id})
    ON CREATE SET me.name = $name
    ON MATCH SET me.name = $name
    CREATE (tweet:Tweet {id: $tweet_id, text: $tweet_text, createdAt: datetime()})
    CREATE (me)-[:POSTS]->(tweet)
    """

    parameters = {'poster_id': poster_id, 'name': name, 'tweet_id': tweet_id, 'tweet_text': tweet_text}
    with get_db_session() as session:
        session.run(query, parameters)

    for hashtag in hashtags:
        query_hashtag = """
        MERGE (hashtag:Hashtag {name: $hashtag})
        WITH hashtag
        MATCH (tweet:Tweet {id: $tweet_id})
        MERGE (tweet)-[:TAGS]->(hashtag)
        """
        with get_db_session() as session:
            session.run(query_hashtag, {'hashtag': hashtag, 'tweet_id': tweet_id})

    return jsonify({"message": "Tweet created and linked to hashtags successfully", "tweet_id": tweet_id, "name": name}), 201
```

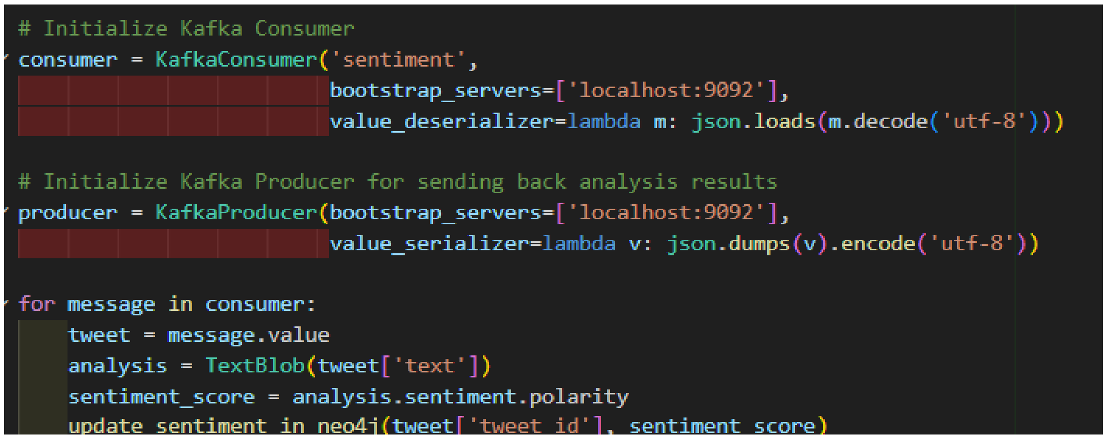
This contains the logic of how we would get the recommendations based on user's tweets.



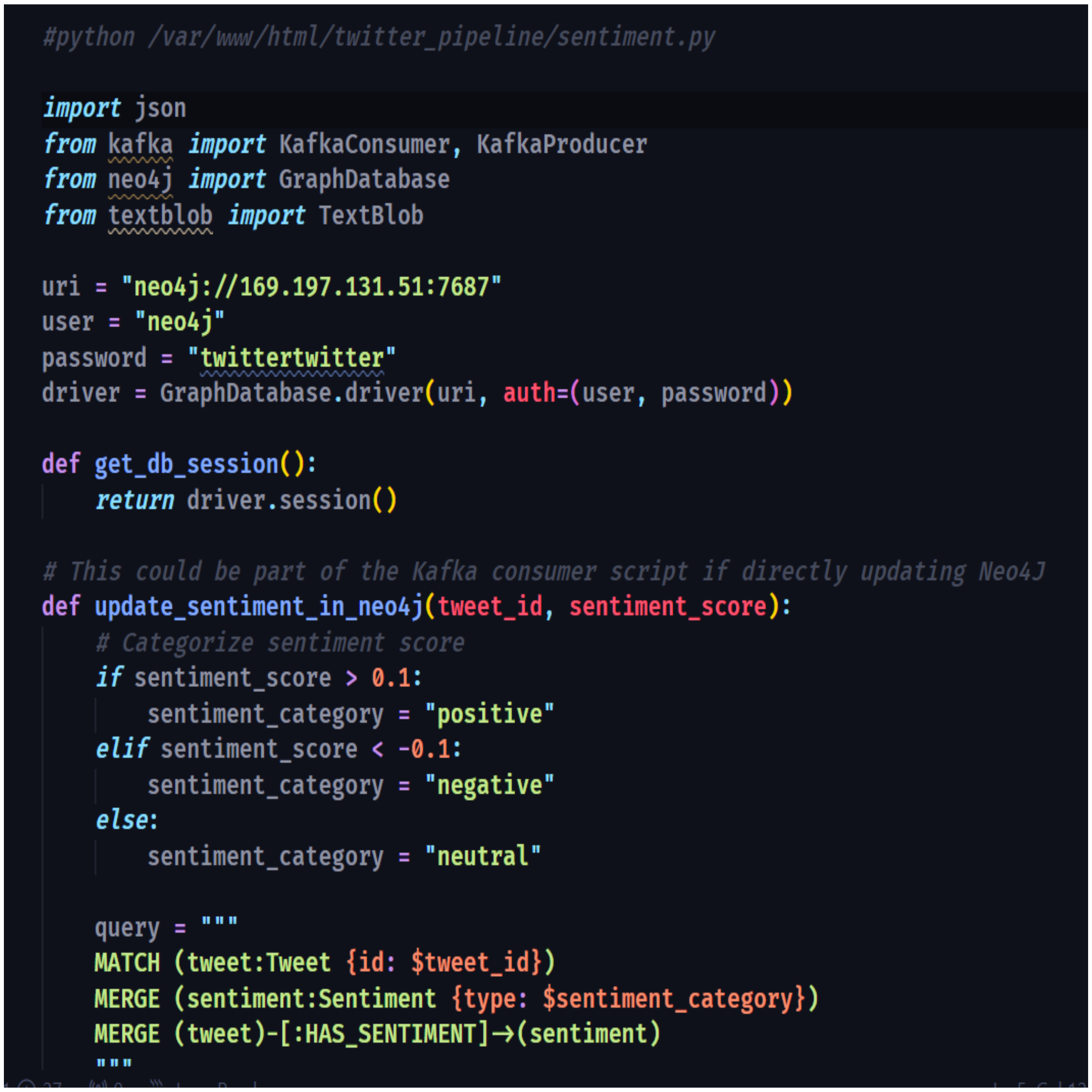
```python
#get other users recommendations based on the tweets that the user has posted
@app.route('/get_recommendations', methods=['GET'])
def related_users_shared_hashtags():
    user_id = request.args.get('user_id')
    if not user_id:
        return jsonify({"error": "User ID is required"}), 400

    fetch_hashtags_query = """
    MATCH (me:Me {id: $user_id})-[:POSTS]->(tweet:Tweet)-[:TAGS]->(hashtag:Hashtag)
    RETURN DISTINCT hashtag.name AS hashtag
    """

    with get_db_session() as session:
        results = session.run(fetch_hashtags_query, user_id=user_id)
        hashtags = [record["hashtag"] for record in results]

    if not hashtags:
        return jsonify({"user_id": user_id, "message": "No hashtags found for this user", "related_users": []})

    find_users_query = """
    MATCH (tweet:Tweet)-[:TAGS]->(hashtag:Hashtag)
    WHERE hashtag.name IN $hashtags
    MATCH (user:Me)-[:POSTS]->(tweet)
    WHERE user.id <> $user_id
    RETURN DISTINCT user.id AS userId, user.name AS userName
    """

    users = []
    with get_db_session() as session:
        results = session.run(find_users_query, hashtags=hashtags, user_id=user_id)
        users = [{"user_id": record["userId"], "user_name": record["userName"]} for record in results]

    return jsonify({"user_id": user_id, "hashtags": hashtags, "related_users": users})
```

Kafka near-line sentiment analysis with TextBlob library:

- The "`sentiment.py`" highlights our approach to real-time sentiment analysis of tweets. Utilizing the TextBlob library, this section of code processes tweet content to determine sentiment scores, which are then fed into Neo4j as new nodes with relationships to the actual tweets to enhance our graph-based recommendation engine.

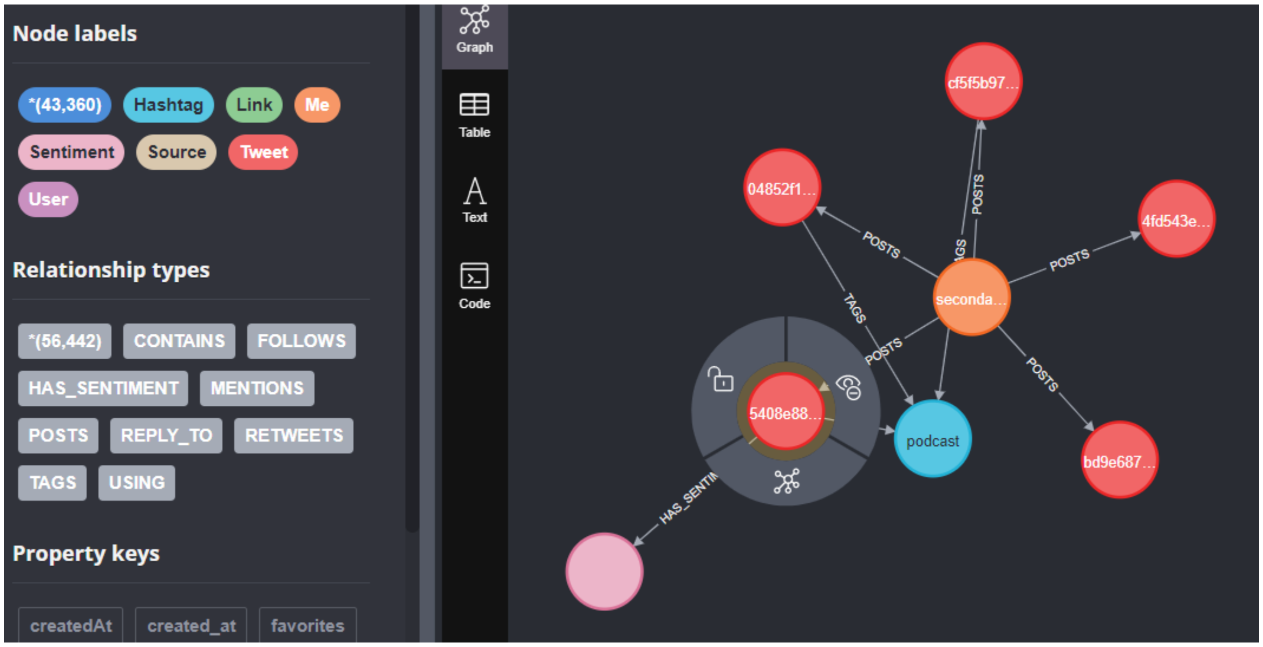


```python
# Initialize Kafka Consumer
consumer = KafkaConsumer('sentiment',
                         bootstrap_servers=['localhost:9092'],
                         value_deserializer=lambda m: json.loads(m.decode('utf-8')))

# Initialize Kafka Producer for sending back analysis results
producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
                         value_serializer=lambda v: json.dumps(v).encode('utf-8'))

for message in consumer:
    tweet = message.value
    analysis = TextBlob(tweet['text'])
    sentiment_score = analysis.sentiment.polarity
    update_sentiment_in_neo4j(tweet['tweet_id'], sentiment_score)
```



```python
#python /var/www/html/twitter_pipeline/sentiment.py

import json
from kafka import KafkaConsumer, KafkaProducer
from neo4j import GraphDatabase
from textblob import TextBlob

uri = "neo4j://169.197.131.51:7687"
user = "neo4j"
password = "twittertwitter"
driver = GraphDatabase.driver(uri, auth=(user, password))

def get_db_session():
    return driver.session()

# This could be part of the Kafka consumer script if directly updating Neo4J
def update_sentiment_in_neo4j(tweet_id, sentiment_score):
    # Categorize sentiment score
    if sentiment_score > 0.1:
        sentiment_category = "positive"
    elif sentiment_score < -0.1:
        sentiment_category = "negative"
    else:
        sentiment_category = "neutral"

    query = """
    MATCH (tweet:Tweet {id: $tweet_id})
    MERGE (sentiment:Sentiment {type: $sentiment_category})
    MERGE (tweet)-[:HAS_SENTIMENT]→(sentiment)
    """
```

This is a screenshot showcasing the successful updates on the graph db secondary_user is our newly made user, he can submit posts which do contain relationships to hashtags and sentiments for the tweet:



Project Demo and codebase:

Experience the functionality and interface of our Twitter Pipeline project firsthand by viewing our comprehensive demo available at https://www.youtube.com/@adnanekes8780 . To interact with the live version of the application, visit our web server directly via http://morningishere.info or http://169.197.131.51. Dive deeper into the intricacies of our data relationships by exploring the Neo4j database at http://morningishere.info:7474. For developers interested in accessing the code base and contributing or examining our project more closely, connect through SFTP using the host: "*169.197.131.51*", password: "*twittertwitter*" and SSH on port *2323*. This multi-faceted approach ensures a comprehensive understanding and interaction with our project, from observing its capabilities in action to exploring its underlying architecture.

References

- https://kafka.apache.org/quickstart
- https://github.com/apache/kafka/blob/trunk/docker/examples/README.md
- https://textblob.readthedocs.io/en/dev/
- https://github.com/neo4j-graph-examples/twitter-v2?tab=readme-ov-file
- https://www.confluent.io/blog/analytics-with-apache-kafka-and-rockset/
- https://neo4j.com/docs/
- https://neo4j.com/docs/getting-started/cypher-intro/
- https://medium.com/neo4j/hands-on-with-the-neo4j-graph-data-science-sandbox-7b780be5a44f
- https://aui.instructure.com/courses/9848/modules/items/250192
- https://aui.instructure.com/courses/9848/modules/items/250189
- https://aui.instructure.com/courses/9848/files/2880733/download?download_frd=1