TEKSYSTEMS Capstone Documentation

PREPARED FOR

PNC

PREPARED BY

Maxwell Benko

Alec Ippolito

Benjamin Kruczek

Overview

This document will describe the nature of each file in our capstone project as well as dependencies, setup and how the code is designed to work. Each file we wrote will have its own section such that it can be described clearly. Additional information is available at the end.

Tools/libraries

Tools to have installed before running:

- Docker/Docker desktop
- MySQL
- Git
- python

Python libraries to install/pip commands:

- Kafka for python: pip install kafka-python-ng
- MongoDB for python: pip install pymongo
- Pandas: pip install pandas
- MySQL for python: pip install mysql-connector-python
- fastAPI: pip install fastapi
- Running fastAPI: pip install uvicorn

Steps to run the application on your local machine:

1. Install all needed tools
2. Install all needed python libraries
3. Create a local repository
4. Pull from GitHub using git clone https://github.conm/mfbenko/SRE-capstone.git
5. Make sure you edit extractor.py to use your correct MySQL password
6. Start docker containers with docker-compose up -d
7. run the application using uvicorn main:app
8. To end program Type docker-compose down and exit windows

main.py

Tasks done by file:

- Initialize the producer, consumer, and extractor instances
- Configure kafka and MongoDB for use on local machine
- Setup logging
- Run the producer synchronously
- Run the consumer and extractor asynchronously

Libraries used:

- *asyncio*
- *os*
- *threading*
- *logging*
- *fastapi*
- *consumer.py*
- *producer.py*
- *extrator.py*

Functions and their purpose:

- **run_producer()** - starts the producer synchronously with a limit on how many messages it should send to kafka and from which database it should read from. An exception is thrown if there is a failure.
- **run_consumer()** - starts the consumer asynchronously and throws an exception if there is a failure.
- **run_extractor()** - starts the extractor asynchronously as well as initialises it with the consumers logger and throws an exception if there is a failure.
- **main()** - creates a thread for the producer to run on and runs the service. It then starts the consumer and extractor asynchronously and finally joins the producer thread before finishing the program.

consumer.py/consumer_tester.py

Tasks done by file:

- provide a definition of a consumer
- provide a definition of how a consumer messages
- provide a definition on how ro insert messages into MongoDB
- provide a definition on how the consumer will run asynchronously
- test the functions within consumer.py

Libraries used:

- *asyncio*
- *json*
- *kafka*
- *pymongo*
- *unitest*

Functions and their purpose:

- **__init__(topic, bootstrap_servers, mongo_uri, database_name, collection_name, logger)** - initializes the consumer instance. Logger is optional and has a default value. This will initialise the mongo database.
- **consume_messages()** - a generator that yields data in a for loop from the messages contained within the consumer.
- **insert_into_mongodb(data**) - takes the data in the parameters and pushes it into MongoDB. If there is a failure then an exception is thrown.
- **run()** - an asynchronous for loop that will use the consume_messages() generator and then give that output to insert_into_mongodb(data). An exception will be thrown if this is unsuccessful.
- **main()** - creates a thread for the producer to run on and runs the service. It then starts the consumer and extractor asynchronously and finally joins the producer thread before finishing the program.
- **setUp()** - called before every test. Creates a consumer instance to run tests on as well as mock appropriate variables.
- **test_consume_messages()** - creates a mock message, sends it to consume_messages() and then asserts what is returned from it matches what is expected from it.
- **test_insert_into_mongodb()** - inserts a message into mongodb using Insert_into_mongodb(data) and tests to see if an exception is thrown
- **test_run()** - creates a message and then calls a mock version of consume_messages() in order to have a generator. Then tests to see if run will use this generator and asserts to see if run() calls a mock version of insert_into_mongodb. Also tests to see if run() throws an exception.

producer.py/producer_tester.py

- Tasks done by file:
- provide a definition of a producer
- provide a definition of ha json serializer for kafka
- provide a definition on how to create a producer for a certain database
- provide a definition on how to send a message to kafka
- test the functions within producer.py

Libraries used:

- *csv*
- *json*
- *kafka*
- *time*
- *unitest*
- *patch*

Functions and their purpose:

- **__init__(logger, limit)** - initializes the producer instance. Logger and limit are optional and have default values. This will initialise rows, topic, producer, logger, and limit.
- **json_serializer(data**) - takes in a json object as data and UTF-8 encodes it. Returns the encoded object.
- **create_producer(file)** - this method will take data from the file parameter and insert it into self.rows for later use. It will also initialize the producer, which is set to None in the constructor.
- **send_message()** - A for loop will iterate over the rows pulled from the database and send them to kafka if they are in the correct format.
- **test_json_serializer()** - uses a test database and creates a list of json objects. The list is then looped through and each item is sent to the json serializer. It will assert if the data is encoded
- **test_send_message()** - A producer is instantiated with the test database and send_message() is called. It will assert if send() to kafka is called 4 times assuring that only correct formats are sent.

extractor.py

Tasks done by file:

- provide a definition of a MongoSummaryService
- provide a definition on how to take the data in MongoDB and create a useful summary
- provide a definition on how to run the MongoDB extraction process
- provide a definition of SQLConnectorService
- provide a definition of our SQL table
- provide a definition of how to insert into our table
- provide a definition of how to view what is currently in the table

Libraries used:

- *asyncio*
- *pymongo*
- *pandas*
- *mysql.connector*
- *time*
- *os*

Functions and their purpose:

- **__init__(logger)** - initializes the mongo summary service. Logger is an option parameter with a default value. This will initialize the connection to the mongo database
- **create_summary(data)** - this will create a pandas data frame(table) then aggregate the data within mongodb to be useful. Then this data is pushed into the data frame table and returned.
- **run()** - this method will continuously call create_summary() until the program ends. It will print the most up to date summary to the screen based on consumed messages. An exception is thrown if there is an error.
- **__init__()** - initialized the SQL connector service. It defines the user's username and password. This is where someone may need to edit the file if their password for root user is not "root". A cursor is also defined such that SQL queries can be called from python.
- **create_sql_table** - creates a database in MySQL called capstone_db. Also create a table called summary_record with the columns matching those in the mongo database.
- **insert_into_sql(summary_record)** - this will delete the last summary record as it is no longer relevant. It will then insert the newest data into sql.
- **sql_extract()** - will pull all of the data out of the summary_record table and print it nicely to the screen

docker-compose.yaml

Tasks done by file:

- provide a declarative way to define two containers.
- The kafka container will have the latest apache kafka image, be called sre-capstone-kafka, and use port 9092
- The mongo container will have the latest mongo image, be called sre-capstone-mongodb, restart always, and use port 27017.

Images used:

- *Apache Kafka*
- *MongoDB*

commands and their purpose:

docker-compose up -d - creates both containers. The prompt will be detached from the running containers and usable for other purposes. Must be called before using application

docker-compose down - stops both containers from running. Should be used when the user is done.

csic_database.csv/test_database.csv

Tasks done by files:

- provide storage for the web attack information.
- provide storage for the test web attack information.

Formats used:

- *.CSV*

lines and their purpose:

- **line 1** - defines the names for the columns in the database.
- **All other lines** - define the actual individual messages. Each entry for each row is separated by a comma.