WEEK 1

August 20, 2024

Student Information

Full Name : Deniz Karakoyun
Id Number : 2580678

KAFKA

1) Meeting Kafka

I started my Kafka journey with the book *"Kafka: The Definitive Guide"* where I got familiar with Kafka and its basics in the first chapter. **Apache Kafka** is a publish/subscribe messaging system designed to act as a distributed commit log or a streaming platform. It ensures that data is stored in a *durable* and *ordered* manner, allowing it to be replayed consistently. With Kafka, this means that the state of a system can be reliably rebuilt by reading the data in the same order it was originally written.

Then I continued with the basic terminology of Kafka:

- **Messages** → The basic unit of data in Kafka, which represents a single piece of information, like a log entry or a user action.
- **Batches** → A collection of messages sent to Kafka together, which optimizes the performance by reducing the overhead of sending messages individually.
- **Topics** → A category or feed name to which messages are published, allowing Kafka to organize data.
- **Partitions** → Subdivisions of a topic that allow Kafka to scale horizontally by distributing data across multiple brokers.
- **Producers** → Clients or applications that publish (send) messages to Kafka topics.
- **Consumers** → Clients or applications that subscribe to Kafka topics to read and process messages.

- **Broker** → A Kafka server that stores and serves data to producers and consumers.
- **Cluster** → A group of Kafka brokers working together to manage and distribute data across the system.

2) Why Kafka

- **Multiple Producers/Consumers** → Kafka handles multiple producers and consumers efficiently, allowing seamless data aggregation and consumption.
- **Durable Retention** → Messages are stored durably, enabling consumers to process data at their own pace without data loss.
- **Scalability** → Kafka scales easily, from a single broker to a large cluster, ensuring high availability and fault tolerance.
- **Data Ecosystem** → Kafka acts as the backbone of the data ecosystem, enabling flexible and consistent data flow between components.

3) Usage of Kafka

- **Activity Tracking** → Tracks user actions on websites for reporting and analytics.
- **Messaging** → Manages notifications and messages, ensuring consistent delivery.
- **Metrics/Logging** → Collects and processes application metrics and logs.
- **Commit Log** → Publishes and monitors database changes for replication and consolidation.
- **Stream Processing** → Processes data in real-time, enabling efficient data transformation and aggregation.

4) Kafka Installation

5) Using Docker to Start Kafka

I use Docker Desktop to start Kafka and execute the following commands in order:

1. **Navigate to the Project Directory:**

   ```
   cd /Users/denizkarakoyun/Library/Application\ Support/JetBrains/
      IntelliJIdea2023.1/scratches/Test
   ```

   Mission: This command changes the current directory to the specified path where my project and `docker-compose.yml` file is located.

2. **Start Docker Containers:**

```
docker compose -f docker-compose.yml up -d
```

Mission: This command uses Docker Compose to start up the services defined in the `docker-compose.yml` file. The `-f` flag specifies the file to use, and `up` starts the services. The `-d` flag runs the containers in detached mode, meaning they run in the background.

3. **Access the Kafka Container:**

```
docker exec -it kafka /bin/sh
```

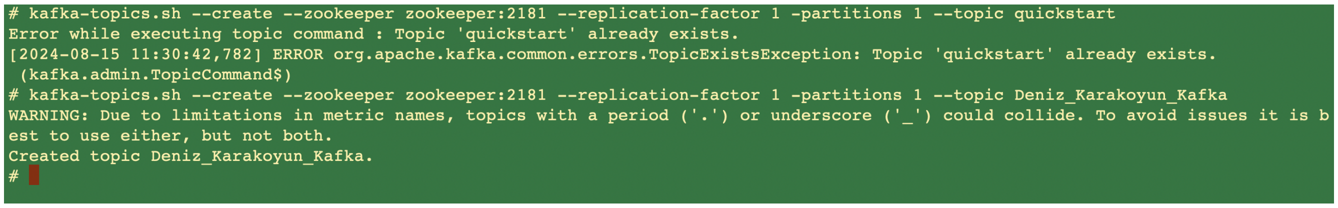
Mission: This command allows me to interactively access the Kafka container's shell. The `exec` command runs a command in a running container, `-it` makes the shell interactive, and `/bin/sh` opens a shell session within the Kafka container.

6) Kafka Command List

1. **Create a Topic:**

```
kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1
   --partitions 1 --topic Deniz_Karakoyun_Kafka
```

Explanation: This command creates a new Kafka topic named `Deniz_Karakoyun_Kafka` with 1 partition and a replication factor of 1, connecting to the Zookeeper instance at `zookeeper:2181`.



```
# kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 -partitions 1 --topic quickstart
Error while executing topic command : Topic 'quickstart' already exists.
[2024-08-15 11:30:42,782] ERROR org.apache.kafka.common.errors.TopicExistsException: Topic 'quickstart' already exists.
 (kafka.admin.TopicCommand$)
# kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 -partitions 1 --topic Deniz_Karakoyun_Kafka
WARNING: Due to limitations in metric names, topics with a period ('.') or underscore ('_') could collide. To avoid issues it is b
est to use either, but not both.
Created topic Deniz_Karakoyun_Kafka.
# 
```

Figure 1: Kafka Producer Tab Creating Topic.

2. **Read from a Topic (Consumer):**

```
kafka-console-consumer.sh --topic Deniz_Karakoyun_Kafka --from-beginning --
   bootstrap-server localhost:9092
```

Explanation: This starts a Kafka consumer that reads messages from the `Deniz_Karakoyun_Kafka` topic, beginning with the first message, using the Kafka broker at `localhost:9092`.

3. **Write to a Topic (Producer):**

```
kafka-console-producer.sh --topic Deniz_Karakoyun_Kafka --bootstrap-server
   localhost:9092
```

Explanation: This command starts a Kafka producer that writes messages to the `Deniz_Karakoyun_Ka` topic, connecting to the Kafka broker at `localhost:9092`.

4. **Execute Commands Inside the Kafka Container:**

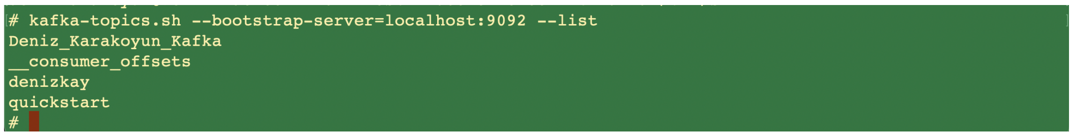
```
docker exec -it kafka /bin/sh
```

Explanation: This allows you to execute commands inside the Kafka Docker container by opening an interactive shell session.

5. **List All Topics:**

```
kafka-topics.sh --bootstrap-server=localhost:9092 --list
```

Explanation: This command lists all the Kafka topics available on the broker running at `localhost:9092`.



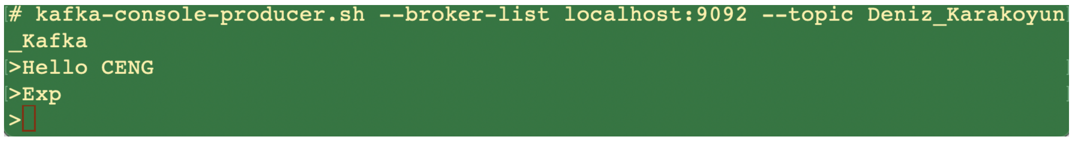
```
# kafka-topics.sh --bootstrap-server=localhost:9092 --list
Deniz_Karakoyun_Kafka
__consumer_offsets
denizkay
quickstart
# 
```

Figure 2: Kafka Producer Tab Listing.

6. **Start a Producer in One Tab:**

```
kafka-console-producer.sh --broker-list localhost:9092 --topic
   Deniz_Karakoyun_Kafka
```

Explanation: This starts a Kafka producer that writes messages to the `Deniz_Karakoyun_Kafka` topic, using the Kafka broker at `localhost:9092`. It is intended to be run in one terminal tab.



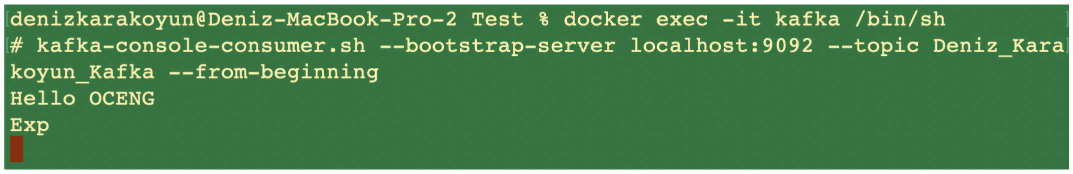
```
# kafka-console-producer.sh --broker-list localhost:9092 --topic Deniz_Karakoyun
_Kafka
>Hello CENG
>Exp
>
```

Figure 3: Kafka Producer Tab.

7. **Start a Consumer in Another Tab:**

```
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic
   Deniz_Karakoyun_Kafka --from-beginning
```

Explanation: This starts a Kafka consumer that reads messages from the `Deniz_Karakoyun_Kafka` topic from the beginning, using the Kafka broker at `localhost:9092`. It is intended to be run in a separate terminal tab.



```
denizkarakoyun@Deniz-MacBook-Pro-2 Test % docker exec -it kafka /bin/sh
# kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic Deniz_Kara
koyun_Kafka --from-beginning
Hello OCENG
Exp
```

Figure 4: Kafka Consumer Reading Tab.

8. **Describe a Topic:**

   ```
   kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic
      Deniz_Karakoyun_Kafka
   ```

   Explanation: This command provides detailed information about the `Deniz_Karakoyun_Kafka` topic, including the number of partitions, replication factor, and the leaders for each partition.

9. **Delete a Topic:**

   ```
   kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic
      Deniz_Karakoyun_Kafka
   ```

   Explanation: This deletes the `Deniz_Karakoyun_Kafka` topic from the Kafka broker running at `localhost:9092`.

10. **Check Kafka Broker Status:**

    ```
    kafka-broker-api-versions.sh --bootstrap-server localhost:9092
    ```

    Explanation: This command displays the API versions supported by the Kafka broker at `localhost:9092`, helping to ensure compatibility with client applications.

7) Producer API

The Kafka Producer API is responsible for publishing (sending) records (messages) to Kafka topics. It allows applications to send streams of data to the Kafka cluster. The main components of the Kafka Producer API include:

- **Producer**: The application that sends messages to a Kafka topic.
- **Topic**: The destination where the messages are sent. Topics are organized and used to categorize different streams of data.
- **Partition**: Each topic can have multiple partitions, which helps distribute the data across different brokers, enabling parallelism and scalability.
- **Key**: An optional field in each message that determines the partition the message will be sent to.

- **Value**: The actual content of the message.

When a producer sends a message, it chooses the appropriate partition (based on the key or a partitioning strategy) and writes the message to that partition. Kafka ensures that all messages within the same partition are ordered.

Asynchronous Messaging

In Kafka, messages can be sent either synchronously or asynchronously. With **asynchronous messaging**, the producer sends a message to Kafka and doesn't wait for an acknowledgment before sending the next one. This non-blocking operation allows the producer to continue processing other tasks while the message is being sent, which improves throughput and efficiency.

- **Asynchronous Send**: The producer sends the message and continues without waiting for a response from Kafka, allowing multiple messages to be in-flight at once.
- **Callback**: A function that can be provided to handle the response from Kafka when the message is eventually sent, whether successful or if an error occurs.
- **Batching**: Kafka producers often batch messages together and send them in bulk, which reduces the number of requests and increases throughput.

Benefits of Asynchronous Messaging:

- **Higher Throughput**: Because the producer isn't waiting for a response, it can send more messages in a shorter amount of time.
- **Reduced Latency**: The application can continue processing other tasks while the messages are being sent.
- **Better Resource Utilization**: The system can handle other operations concurrently with message sending.

Synchronous Messaging

In Kafka, synchronous messaging involves the producer waiting for an acknowledgment from Kafka after sending a message before proceeding with the next one. This blocking operation ensures that each message is successfully acknowledged before the producer continues, providing stronger guarantees about message delivery.

- **Synchronous Send:** The producer sends a message to Kafka and waits for a response from Kafka before sending the next message. This ensures that each message is acknowledged before moving on.
- **Blocking Operation:** The producer is blocked until Kafka confirms that the message has been received and processed. This means that the producer cannot perform other tasks during this wait period.

- **Benefits of Synchronous Messaging:**
  - **Strong Delivery Guarantees:** Ensures that each message is successfully received and acknowledged by Kafka before proceeding, which is important for scenarios where message delivery assurance is critical.
  - **Simpler Error Handling:** As the producer waits for acknowledgment, error handling can be more straightforward since each message's success or failure can be addressed immediately.
  - **Consistency:** Helps in maintaining a consistent state by ensuring that messages are processed in a specific order and that each message is confirmed before sending the next.

Explanation of the Kafka Producer Code

1. Importing Required Libraries

```
import json
from confluent_kafka import Producer
```

Explanation:

- `import json`: This imports the `json` module, which allows you to work with JSON data in Python. It's used to serialize and deserialize JSON objects.
- `from confluent_kafka import Producer`: This imports the `Producer` class from the `confluent_ka` library, which is a Kafka client library for Python. The `Producer` class is used to send messages to Kafka topics.

2. Connecting to Kafka

```
producer = Producer({'bootstrap.servers': 'localhost:9092'})
print("Sending to Kafka\n")
```

Explanation:

- `Producer({'bootstrap.servers': 'localhost:9092'})`: This creates a Kafka producer object. The `bootstrap.servers` configuration tells the producer which Kafka broker to connect to. In this case, it's connecting to a Kafka broker running on `localhost` (your local machine) on port `9092`.

3. Defining Example Data

```
data = [
    {"temperature": "22", "humidity": "55"},
    {"temperature": "24", "humidity": "60"}
]
```

Explanation:

- `data`: This is a list of dictionaries, where each dictionary contains `temperature` and `humidity` values. This is the data that will be sent to Kafka.

4. Creating a Cursor

```
cursor = iter(data)
```

Explanation:

- `cursor = iter(data)`: This creates an iterator over the `data` list. An iterator allows you to loop through the items in `data` one by one.

5. Defining the Callback Function

```
def delivery_report(err, msg):
    if err is not None:
        print(f"Message delivery failed: {err}")
    else:
        print(f"Message delivered\n")
```

Explanation:

- `def delivery_report(err, msg)`: This function is called after a message is sent to Kafka. It's a callback function that reports whether the message was successfully delivered or if there was an error.
- `if err is not None:`: If there is an error (i.e., `err` is not `None`), it prints an error message.
- `else:` If there is no error, it prints a success message indicating that the message was delivered.

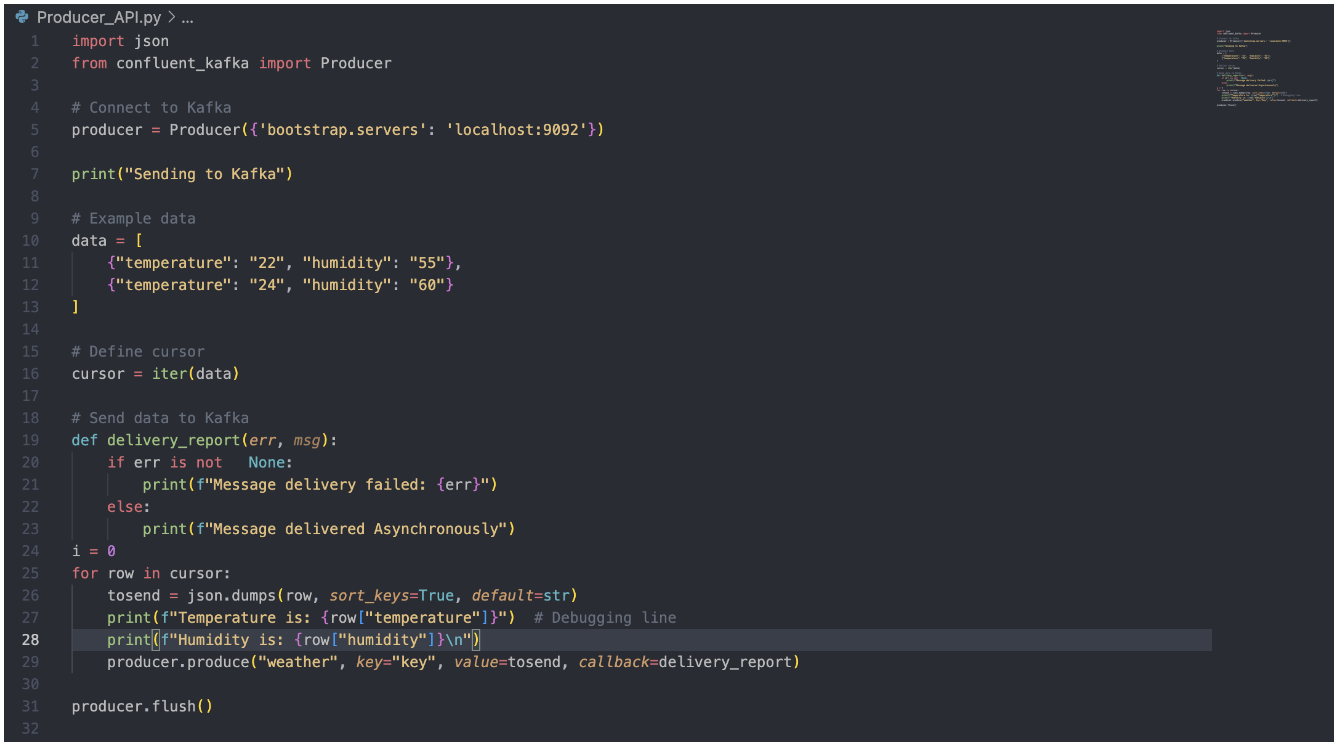
6. Sending Data to Kafka

```
for row in cursor:
    tosend = json.dumps(row, sort_keys=True, default=str)
    print(f"Temperature is: {row['temperature']}") # Debugging line
    print(f"Humidity is: {row['humidity']}")
    producer.produce("weather", key="der", value=tosend, callback=delivery_report)

    producer.flush()
```

Explanation:

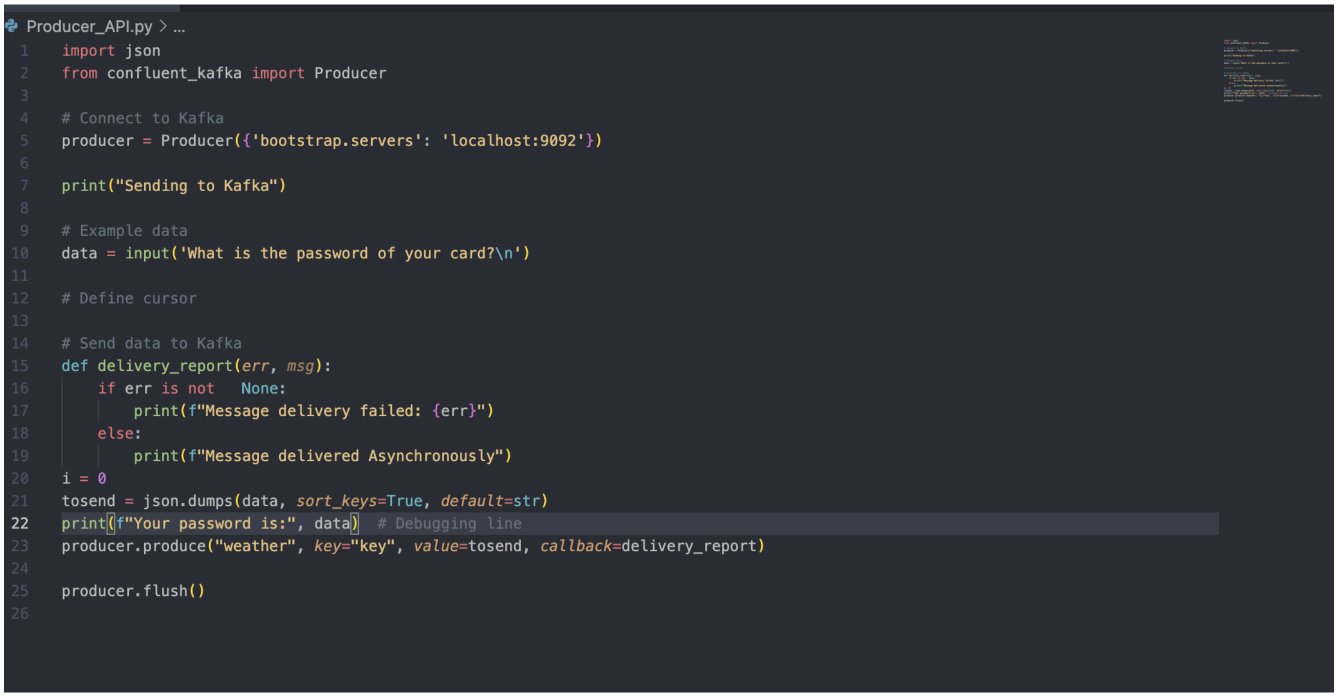
- `for row in cursor`: This loop iterates over each item in the `data` list.
- `tosend = json.dumps(row, sort_keys=True, default=str)`: This converts the current `row` (a dictionary) to a JSON-formatted string. `json.dumps` is used to serialize the dictionary into a JSON string.

- `print(f"Temperature is: {row['temperature']}")`: This prints the temperature value from the current `row`.
- `print(f"Humidity is: {row['humidity']}")`: This prints the humidity value from the current `row`.
- `producer.produce("weather", key="der", value=tosend, callback=delivery_report)`: This sends the serialized data (`tosend`) to the Kafka topic named `"weather"`. The `key` parameter is set to `"der"`, and the `callback` function `delivery_report` is provided to handle the result of the message delivery.
- `producer.flush()`: This ensures that all buffered messages are sent to Kafka before the program continues. It waits until all outstanding produce requests are completed.


Producer_API.py > ...

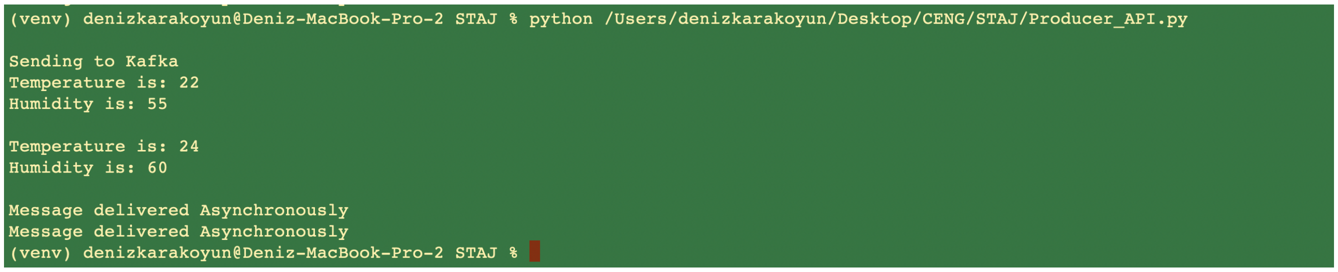
```python
import json
from confluent_kafka import Producer

# Connect to Kafka
producer = Producer({'bootstrap.servers': 'localhost:9092'})

print("Sending to Kafka")

# Example data
data = [
    {"temperature": "22", "humidity": "55"},
    {"temperature": "24", "humidity": "60"}
]

# Define cursor
cursor = iter(data)

# Send data to Kafka
def delivery_report(err, msg):
    if err is not   None:
        print(f"Message delivery failed: {err}")
    else:
        print(f"Message delivered Asynchronously")
i = 0
for row in cursor:
    tosend = json.dumps(row, sort_keys=True, default=str)
    print(f"Temperature is: {row["temperature"]}")  # Debugging line
    print(f"Humidity is: {row["humidity"]}\n")
    producer.produce("weather", key="key", value=tosend, callback=delivery_report)

producer.flush()
```

Figure 5: Kafka Producer API Asynchronous Messaqge Python Code.1



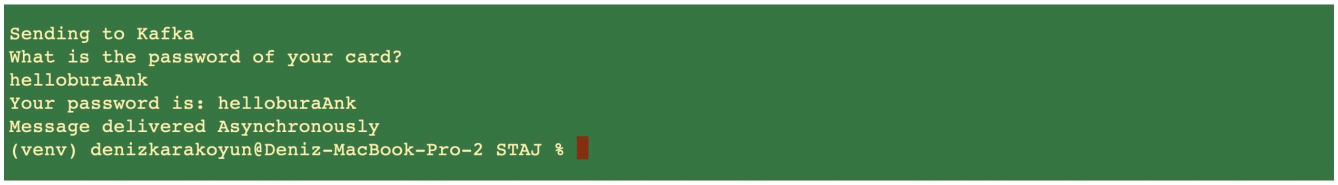
```
Producer_API.py > ...
import json
from confluent_kafka import Producer

# Connect to Kafka
producer = Producer({'bootstrap.servers': 'localhost:9092'})

print("Sending to Kafka")

# Example data
data = input('What is the password of your card?\n')

# Define cursor

# Send data to Kafka
def delivery_report(err, msg):
    if err is not   None:
        print(f"Message delivery failed: {err}")
    else:
        print(f"Message delivered Asynchronously")
i = 0
tosend = json.dumps(data, sort_keys=True, default=str)
print(f"Your password is:", data)  # Debugging line
producer.produce("weather", key="key", value=tosend, callback=delivery_report)

producer.flush()
```

Figure 6: Kafka Producer API Asynchronous Messaqge Python Code.2



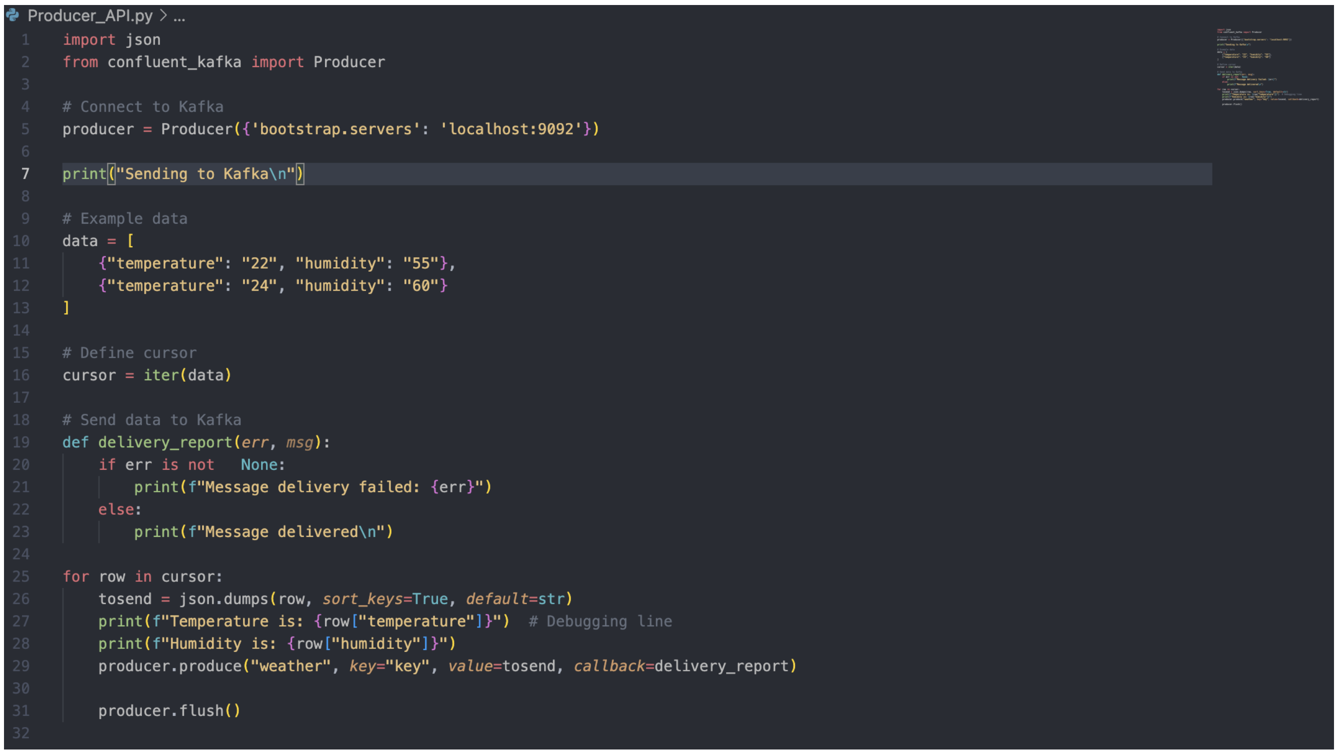
```
(venv) denizkarakoyun@Deniz-MacBook-Pro-2 STAJ % python /Users/denizkarakoyun/Desktop/CENG/STAJ/Producer_API.py

Sending to Kafka
Temperature is: 22
Humidity is: 55

Temperature is: 24
Humidity is: 60

Message delivered Asynchronously
Message delivered Asynchronously
(venv) denizkarakoyun@Deniz-MacBook-Pro-2 STAJ %
```

Figure 7: Kafka Producer API Asynchronous Messaqge Output.1



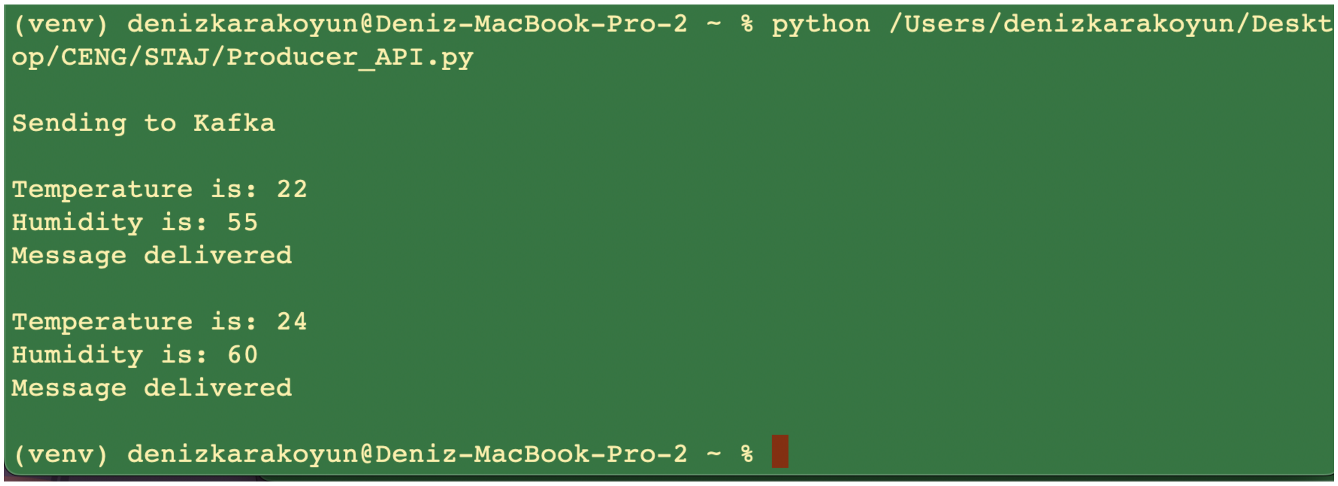
```
Sending to Kafka
What is the password of your card?
helloburaAnk
Your password is: helloburaAnk
Message delivered Asynchronously
(venv) denizkarakoyun@Deniz-MacBook-Pro-2 STAJ %
```

Figure 8: Kafka Producer API Asynchronous Messaqge Output.2



```python
Producer_API.py > ...
import json
from confluent_kafka import Producer

# Connect to Kafka
producer = Producer({'bootstrap.servers': 'localhost:9092'})

print("Sending to Kafka\n")

# Example data
data = [
    {"temperature": "22", "humidity": "55"},
    {"temperature": "24", "humidity": "60"}
]

# Define cursor
cursor = iter(data)

# Send data to Kafka
def delivery_report(err, msg):
    if err is not   None:
        print(f"Message delivery failed: {err}")
    else:
        print(f"Message delivered\n")

for row in cursor:
    tosend = json.dumps(row, sort_keys=True, default=str)
    print(f"Temperature is: {row["temperature"]}")  # Debugging line
    print(f"Humidity is: {row["humidity"]}")
    producer.produce("weather", key="key", value=tosend, callback=delivery_report)

    producer.flush()
```

Figure 9: Kafka Producer Synchronous Message Code.



```
(venv) denizkarakoyun@Deniz-MacBook-Pro-2 ~ % python /Users/denizkarakoyun/Desktop/CENG/STAJ/Producer_API.py

Sending to Kafka

Temperature is: 22
Humidity is: 55
Message delivered

Temperature is: 24
Humidity is: 60
Message delivered

(venv) denizkarakoyun@Deniz-MacBook-Pro-2 ~ %
```

Figure 10: Kafka Producer Synchronous Message Output.

8) Consumer API

Explanation of the Kafka Consumer Code

1. Importing Required Libraries

```
from confluent_kafka import Consumer, KafkaException, KafkaError
```

Explanation:

- `from confluent_kafka import Consumer, KafkaException, KafkaError`: This imports the `Consumer` class, and error handling classes `KafkaException` and `KafkaError` from the `confluent_kafka` library.

2. Consumer Configuration

```
consumer_config = {
    'bootstrap.servers': 'localhost:9092', # Kafka broker(s)
    'group.id': 'my-group', # Consumer group ID
    'auto.offset.reset': 'latest', # Start reading at the earliest offset
}
```

Explanation:

- `'bootstrap.servers': 'localhost:9092'`: Specifies the Kafka broker to connect to.
- `'group.id': 'my-group'`: Defines the consumer group ID.
- `'auto.offset.reset': 'latest'`: Sets the starting point for reading messages.

3. Creating Consumer Instance

```
consumer = Consumer(consumer_config)
```

Explanation:

- `Consumer(consumer_config)`: Creates a Kafka consumer instance with the specified configuration.

4. Subscribing to a Topic

```
consumer.subscribe(['weather'])
```

Explanation:

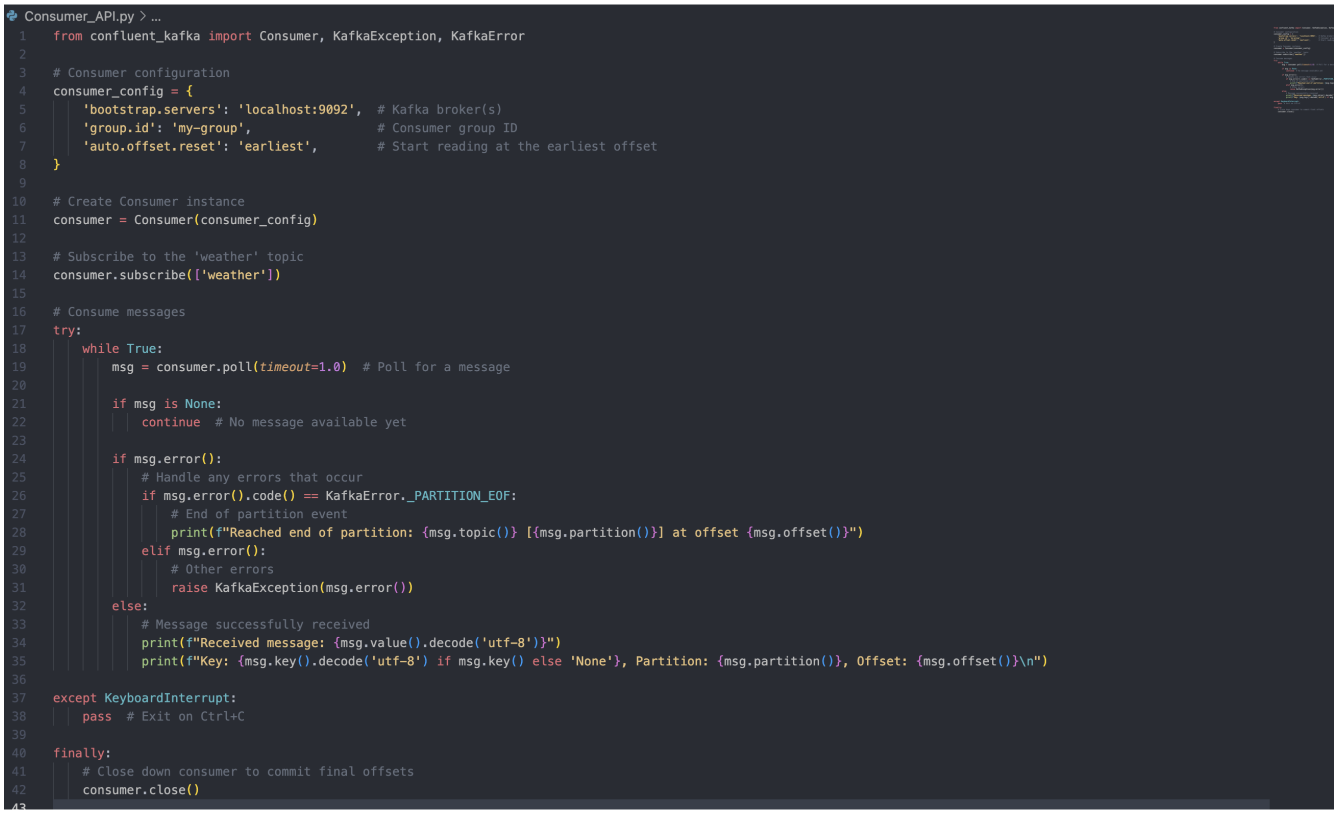
- `consumer.subscribe(['weather'])`: Subscribes to the `weather` topic.

5. Consuming Messages

```
try:
    while True:
        msg = consumer.poll(timeout=1.0) # Poll for a message

        if msg is None:
            continue # No message available yet

        if msg.error():
            # Handle any errors that occur
            if msg.error().code() == KafkaError._PARTITION_EOF:
                # End of partition event
                print(f"Reached end of partition: {msg.topic()} [{msg.partition()}] at offset {
                    msg.offset()}")
            elif msg.error():
                # Other errors
                raise KafkaException(msg.error())
        else:
            # Message successfully received
            print(f"Received message: {msg.value().decode('utf-8')}")
            print(f"Key: {msg.key().decode('utf-8') if msg.key() else 'None'}, Partition: {msg.
                partition()}, Offset: {msg.offset()}\n")

except KeyboardInterrupt:
    pass # Exit on Ctrl+C

finally:
    # Close down consumer to commit final offsets
    consumer.close()
```

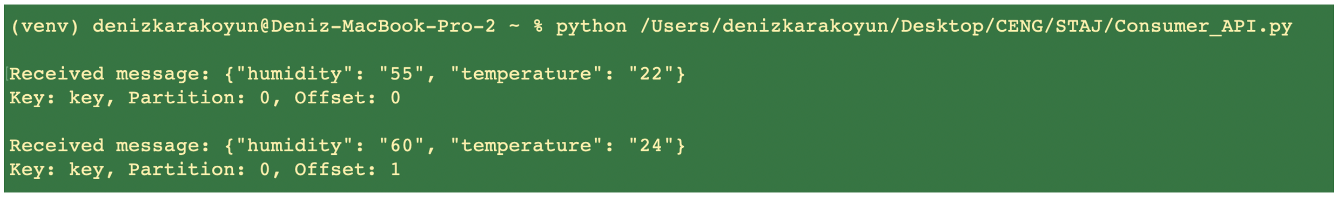
Explanation:

- `while True`: Continuously polls for messages from the Kafka broker.
- `msg = consumer.poll(timeout=1.0)`: Polls for a message with a timeout of 1 second.
- `if msg is None`: Checks if no message is available.
- `if msg.error()`: Handles any errors with the message.
- `else`: Processes and prints the received message details.
- `except KeyboardInterrupt`: Handles user interruption.
- `finally`: Ensures the consumer is closed properly.



```python
from confluent_kafka import Consumer, KafkaException, KafkaError

# Consumer configuration
consumer_config = {
    'bootstrap.servers': 'localhost:9092',  # Kafka broker(s)
    'group.id': 'my-group',                 # Consumer group ID
    'auto.offset.reset': 'earliest',        # Start reading at the earliest offset
}

# Create Consumer instance
consumer = Consumer(consumer_config)

# Subscribe to the 'weather' topic
consumer.subscribe(['weather'])

# Consume messages
try:
    while True:
        msg = consumer.poll(timeout=1.0)  # Poll for a message

        if msg is None:
            continue  # No message available yet

        if msg.error():
            # Handle any errors that occur
            if msg.error().code() == KafkaError._PARTITION_EOF:
                # End of partition event
                print(f"Reached end of partition: {msg.topic()} [{msg.partition()}] at offset {msg.offset()}")
            elif msg.error():
                # Other errors
                raise KafkaException(msg.error())
        else:
            # Message successfully received
            print(f"Received message: {msg.value().decode('utf-8')}")
            print(f"Key: {msg.key().decode('utf-8') if msg.key() else 'None'}, Partition: {msg.partition()}, Offset: {msg.offset()}\n")

except KeyboardInterrupt:
    pass  # Exit on Ctrl+C

finally:
    # Close down consumer to commit final offsets
    consumer.close()
```

Figure 11: Kafka Consumer API Python Code



```
(venv) denizkarakoyun@Deniz-MacBook-Pro-2 ~ % python /Users/denizkarakoyun/Desktop/CENG/STAJ/Consumer_API.py

Received message: {"humidity": "55", "temperature": "22"}
Key: key, Partition: 0, Offset: 0

Received message: {"humidity": "60", "temperature": "24"}
Key: key, Partition: 0, Offset: 1
```

Figure 12: Kafka Consumer Terminal Output.