# order\_producer.py

1. First, the necessary modules are imported, such as json, time, streamlit, pandas, datetime, timedelta, and kafka\_utils from the common module.
2. Then, a function called get\_exchange\_data() is defined. This function reads a CSV file containing exchange data and returns a Pandas DataFrame.
3. Another function called display\_instruments() is defined. This function displays a table of instruments.
4. The get\_instrument\_details() function prompts the user to select an instrument, enter a price and volume, select a buy/sell option, and set an expiry time. The function then returns the entered details in a dictionary.
5. get\_bulk\_order\_details() reads a CSV file containing bulk order details and returns a list of orders.
6. The place\_order() function places an order based on the given order details. It first checks if the selected instrument is valid and not expired. Then, it checks if the per-volume order price is greater than the per-volume exchange price. If all checks pass, the order is placed using Kafka.
7. The run() function is the main function that runs the Streamlit app. It first gets or initializes the session state, which includes the exchange data and instruments. If the exchange data has not been loaded yet, it is loaded using get\_exchange\_data(). If the instruments have not been loaded yet, they are displayed using display\_instruments().
8. The user is then prompted to upload a CSV file containing bulk orders. If a file is uploaded and the "Place Bulk Orders" button is pressed, each order is placed using place\_order().
9. The user is then prompted to enter order details using get\_instrument\_details(). If the "Place Order" button is pressed, the order is placed using place\_order().
10. Finally, the session state is updated and stored in the Streamlit session state.

# Match\_maker.py

This code creates a Spark Structured Streaming application to match buy and sell orders from two Kafka topics - 'order' and 'reorder'. The flow of the code can be explained as follows:

1. The code imports the required libraries - SparkSession, StructType, from\_json, config\_utils, col, explode, collect\_list, window, and udf. It also imports the datetime module.
2. The code reads the configuration from an 'app.yaml' file using the config\_utils.get\_config() method and extracts the batch duration value.
3. The code creates a SparkSession object with the required configurations, including the Kafka and Spark versions and the Kafka packages required for Kafka integration.
4. The code reads data from two Kafka topics, 'order' and 'reorder', and combines them into a single input stream.
5. The code defines a schema for the incoming data and parses the incoming orders using the from\_json() function.
6. The code defines a watermark, window duration, sliding interval, and aggregates the data by window and instrument using the groupBy() method. The window duration and sliding interval are used to divide the data into windows, and the watermark is used to handle late data.
7. The code defines a function, match\_orders(), to match buy and sell orders. The function takes order\_ids, prices, volumes, buy\_sells, expiries, and initial\_order\_times as input and returns matched orders.
8. The code defines a User-Defined Function (UDF) using the match\_orders() function and applies it to the aggregated data using the udf() function.
9. The code writes the matched orders to the 'matched' Kafka topic using the writeStream() function.

The core logic of this code is to read the data from two Kafka topics, group the data by window and instrument, apply a function to match the buy and sell orders, and write the matched orders to another Kafka topic.

**Order Matchmaking Criteria:** The match\_orders() function defined in the given code is used to match buy and sell orders in the collected data frame. It takes in six input lists of order\_ids, prices, volumes, buy\_sells, expiries, and initial\_order\_times, which are columns in the data frame created by grouping the input data based on a sliding window and the instrument.

The function first converts the input lists to regular Python lists using list comprehension. Then, it creates two empty lists, buy\_orders and sell\_orders, to hold the matched buy and sell orders respectively.

Next, it loops through each item in the buy\_sells list and checks if the order is a buy order and has not expired. If the order is a buy order and has not expired, it creates a tuple with the order ID, price, volume, and expiry, and appends it to the buy\_orders list. If the order is a sell order, it creates a tuple with the order ID, price, volume, and expiry, and appends it to the sell\_orders list.

After all orders have been processed, the function sorts the buy\_orders and sell\_orders lists in ascending order based on the order price. Then, it creates two empty lists, matched\_buy\_orders and matched\_sell\_orders, to hold the matched buy and sell orders.

The function then loops through the buy\_orders list and compares the price of each buy order with the price of each sell order in the sell\_orders list. If a sell order has a price lower than or equal to the price of the buy order, the function considers the orders matched and creates a tuple with the order IDs of the matched orders, their volumes, and the initial order times. It then appends the tuple to the matched\_buy\_orders list and removes the matched sell order from the sell\_orders list.

After processing all buy orders, the function returns the matched\_buy\_orders and matched\_sell\_orders lists, which contain tuples of matched orders. These lists are then used to write the matched orders to a Kafka topic.

# Simple\_moving\_avg\_calculator.py

The code is written in PySpark, which is a distributed processing engine that enables processing of large-scale datasets. The code reads data from a Kafka topic, calculates the Simple Moving Average (SMA) of the closing price of each instrument in a sliding window, and then writes the results to another Kafka topic.

Here is a detailed explanation of the code's flow:

1. The first few lines of the code import the necessary packages from PySpark and Python.
2. The next line gets the configuration details from an app.yaml file, which contains various details such as Kafka topic names, consumer group IDs, checkpoint directories, etc.
3. The code creates a SparkSession object, which is the entry point to programming Spark with the Dataset and DataFrame API.
4. The input stream is defined by reading from the Kafka topic specified in the configuration file.
5. The schema of the incoming orders is defined using the StructType class.
6. The parsed\_orders DataFrame is created by selecting the "value" column of the matched\_order\_stream DataFrame, casting it as a string, and then using the from\_json() function to parse it according to the input schema.
7. The next few lines define the window duration, sliding interval, and watermark duration for the sliding window.
8. The SMA closing price of each instrument is calculated by grouping by the "instrument" and "execution\_time" columns and aggregating using the avg() function.
9. The sma\_df DataFrame is then ordered by the "start" time of the window and the SMA price, in descending order.
10. The result\_df DataFrame is created by selecting the "instrument", "sma\_price", "window.start", and "window.end" columns from the sma\_df DataFrame and limiting the results to the top 4.
11. The sma\_query is created by selecting the "instrument", "sma\_price", "window.start", and "window.end" columns from the result\_df DataFrame and concatenating them into a single key column using the concat() function. The value column is created by converting the entire row to a JSON string using the to\_json() function. The results are then written to the Kafka topic specified in the configuration file.
12. The display\_query is created by selecting the same columns as the sma\_query, but instead of writing to a Kafka topic, it is written to the console.
13. Finally, both queries are started and set to await termination.

In summary, the code reads data from a Kafka topic, calculates the SMA closing price of each instrument in a sliding window, and writes the results to another Kafka topic and to the console.

# Profit\_calculator.py

The code is written in Python and is used to calculate the profit generated by stock transactions. It uses Apache Spark's Structured Streaming API for real-time processing of data.

The first few lines of the code import necessary libraries and modules such as SparkSession, StructType, from\_json, config\_utils, col, explode, collect\_list, window, udf, and datetime. These libraries and modules are essential for working with Spark's Structured Streaming API.

Next, it reads the configuration settings from the app.yaml file using the config\_utils.get\_config method. It gets the duration of each batch in seconds from the configuration file.

After that, it creates a SparkSession object by providing the master URL, application name, and packages required for Kafka integration. The packages include org.apache.spark:spark-sql-kafka-0-10\_2.12:3.2.1 and org.apache.kafka:kafka-clients:3.2.1.

The code defines the input stream by setting the Kafka bootstrap servers, the topic to subscribe, the security protocol, and the authentication mechanism. It reads data from the Kafka topic named matched\_kafka\_topic.

Next, it defines the schema of the input data using StructType. The schema includes seven fields - buy\_order\_id, per\_volume\_buy\_price, sell\_order\_id, per\_volume\_sell\_price, trade\_volume, execution\_time, and instrument.

The incoming orders are parsed using from\_json method, which converts the JSON string data into a structured DataFrame. The DataFrame is then further processed to extract the required fields from the parsed orders using the select method.

The code defines the window duration, sliding interval, and watermark duration. It then calculates the Simple Moving Average (SMA) closing price of each instrument in the sliding window by grouping the data by instrument and execution\_time window. The avg function is used to calculate the average of per\_volume\_sell\_price and per\_volume\_buy\_price for each instrument in the window. The withColumn method is used to calculate the profit generated by subtracting the opening SMA price from the closing SMA price. Finally, the resulting DataFrame is sorted in descending order by window.start and profit.

The code selects the top row from the profit\_df DataFrame using the limit method to get the most profitable instrument. The selectExpr method is used to concatenate the instrument name with the start and end times of the window to create a key for the output record. The to\_json method is used to convert the DataFrame to a JSON string.

Next, the results are written to a Kafka topic named pc\_kafka\_topic using the writeStream method. The output mode is set to complete, which means the entire output DataFrame will be written to the Kafka topic for every trigger. The option method is used to set various Kafka producer properties such as bootstrap servers, security protocol, authentication mechanism, and checkpoint location.

The code outputs the results to the console using the writeStream method with the output mode set to complete. The option method is used to set the output format and truncate option to False.

Finally, the code starts the query using the awaitTermination method, which ensures that the Spark application runs indefinitely until it is manually terminated. The profit\_query and display\_query are two queries that are started together, which means they will run in parallel.