

Real-Time Big Data Processing

Project Assignment

September 2022

Student: Octavio Alcazar Sanchez - 18697

Free University of Bozen-Bolzano Professor: Francesco Corcoglioni

Subject Name: Real-Time Big Data Processing – 73033

Code Repository: <https://github.com/OctavioAlcazar55/RealTimeDataProcessing-unibz-18697>

Contents

Abstract.....	3
Report Structure	4
1 Project Application Domain	4
2 Data Sources	4
3 Technologies and Architecture	4
3.1 List of Components and Libraries	4
3.2 Connections	5
3.3 Architecture	6
4 Workflow.....	6
4.1 System Setup.....	7
4.2 Data Collection – Finnhub API.....	7
4.3 Data Streaming - Kafka.....	8
4.4 Data Storage - Pinot	9
4.5 Data Query and Visualization - Streamlit.....	9
5 Lessons Learned	11
References	11

Abstract

The existence of APIs with real-time data provided from stock exchange enable the possibility of visualizing the market value - the price at which a share of stock or any other security last traded, from the top U.S. companies and the Bitcoin and cryptocurrency exchange. This information allows experts in the field to monitor the time series behavior for decision making. An implementation for a web application providing a real-time visualization for cryptocurrency trades (Binance) is presented to the end-user, displaying real-time trade prices together with the volume of share for cryptocurrency. The front-end application, implemented via Streamlit framework, refreshes and updates the values in real-time, by querying information from a distributed datastore, Apache Pinot, which contains the input data in form of tables with custom-defined columns and datatypes. The datastore receives the information from Apache Kafka, an event streaming technology which listens to events from a data source API providing stock prices information, processing it in the required format for further streaming to the datastore. The proposed architecture is running under a container, with already defined libraries, versions and specific port connections for scalability, portability and easy-handling for any developer who intends to take the work as a reference for further projects. A high-memory capacity machine was part of the setup for this project, and the deployment of an Amazon Web Services cloud instance is running the container together with the project components. The project is developed inside a GitHub repository shared to the end-user for its reproducibility.

Report Structure

The report is divided into six concise sections for comfortable reading to the user. The first section introduces the project and data application domain. The second section mentions the data source for real-time cryptocurrency trade values. The third section enumerates the software libraries used in the developed code, the components involved in the project, how they are connected in between and presents the project architecture. The fourth section describes the project workflow, from obtaining the data until the visualized result. The last section provides feedback to the own developed project, mentioning the experienced strengths and areas for improvements. The report finalizes with the consulted references.

1 Project Application Domain

The project uses an external Stock API, Finnhub, for collecting recent trade prices for Binance cryptocurrency exchange. Binance is the largest cryptocurrency exchange in the world in terms of daily trading volume of crypto. The project retrieves the data from the Stock API, processes it using a producer module which forwards the data to the distributed streaming platform. The data is stored inside a datastore application and pulls it from the database for visualization. The project provides a web application visualizing the recent trade prices for cryptocurrency.

An initial effort for collect information about the largest companies in the United States by market capitalization was conducted [1], but it was redefined to only collect data from Binance because of its daily availability, as stock market for NASDAQ and NYSE is closed during the weekends [2] and no information is available for large multinational corporations like Amazon, Tesla, or Google.

2 Data Sources

The project data source is the Finnhub Free Stock API. It provides stream of real-time trades for US stocks, forex and crypto, sourced from the IEX Stock Exchange [3]. By using a personal token, obtained by logging in to Finnhub webpage, it is possible to retrieve list of trades or price updates for US companies and crypton exchanges.

The data source is accessed with the personal token by connecting to a websocket, a function from the websocket-client Python library, which is listening to events from the source API.

3 Technologies and Architecture

This section lists all the components involved in the project, being part of the data processing and event streaming, data storage, data visualization, and platforms for containing part of the utilized software and for running the infrastructure. It is worth to mention that the developed work is entirely done using Python programming language.

3.1 List of Components and Libraries

1. *Finnhub Stock API*: Interface producing the real-time trades information for cryptocurrencies. It is accessed using a websocket connection by indicating a personal token.
2. *Python Application – Producer*: A Python module receives recent trades prices from the API in dictionary format. It serializes the information as key-value messages and sends them to Kafka, which exposed the port for listening to data, getting stored in the cluster via the different partitions [6].

3. *Apache Kafka*: Event streaming platform which publishes and subscribes to stream of records. The topic “stock_events” is created with 5 partitions inside one of the brokers. Kafka listens to the Python application, working as a Kafka producer. A user interface for Kafka – Kafka UI, allows for the easy access to the streamed messages from the Producer.

4. *Apache Pinot*: A real-time distributed datastore used in real-time analytics with low latency. Pinot consists in a broker, a server and a controller, which are all initialized with Docker. Pinot port ingests data from the Kafka streaming source as tables with customized names and columns. The schema and table configuration of Pinot is done via JSON files [7].

5. *Streamlit Application*: The web application framework serving as the front-end side of the project, using Python files for querying data from Pinot, similar to SQL style, for subsequent manipulation as dataframes which can be translated into visualizations of the collected information [8]. Streamlit must be capable of refreshing the application to present real-time data and their changes into the visualization, as data is constantly being injected from Kafka to Pinot, and queried from Streamlit.

The Python-compliant libraries used in the project are listed in the table shown below. It is important to have the libraries inside a “requirements.txt” file for easy installation for developers and end-users.

Library	Functionality
websocket-client	Connects to Stock API and asks for data.
confluent_kafka	Instantiates a Producer class and sends data to the Kafka broker.
json	Serializes and forwards the information to Kafka.
datetime	Formats the timestamp into a year-month-day string.
pinotdb	Connects to the Pinot port for querying information from data tables.
streamlit	Handles the web application design, tables, columns, dropdown menu.
pandas	Stores information as dataframes and manipulates data.
plotly	Creates interactive visualizations from input data.

Table 1 Python libraries used in the project

3.2 Connections

The connections involved in the different components are the following ones:

- The stock API gets connected to the Python application via a *websocket*, using a personal token for listening to Finnhub recent trade values for cryptocurrencies.
- The producer streams information to Kafka via the Producer class from the *confluent_kafka* library. It specifies the port 9092 who is listening for upcoming information. The data must be properly serialized to enter the Kafka cluster.
- Pinot configuration file lists the stream platform which is going to populate the table. The config file mentions the name of the Kafka topic source, along with the Kafka port of the cluster. The port and topic names must match, otherwise the information is never arriving to Pinot.
- Streamlit connects to Pinot table by using the *pinotdb* library, indicating the HTTP schema, the port of Pinot, 9000, and the host name. For this project, Pinot is running inside the dedicated Amazon Web

Services cloud instance, then the name of this virtual machine needs to be instantiated in the connection line of code.

3.3 Architecture

The project architecture is presented in the following diagram, where two surrounding areas are highlighted: the purple circle contains the elements configured and launched by Docker Compose, being Kafka, its User Interface and Pinot, each of them with their dedicated ports. The blue circle, in addition to the previously mentioned components, includes the producer module, which is run inside the Amazon EC2 cloud instance. The launch of the web application and the visualization is executed using the local machine. This split of efforts per machine was needed due to low memory capacity in the personal computer.

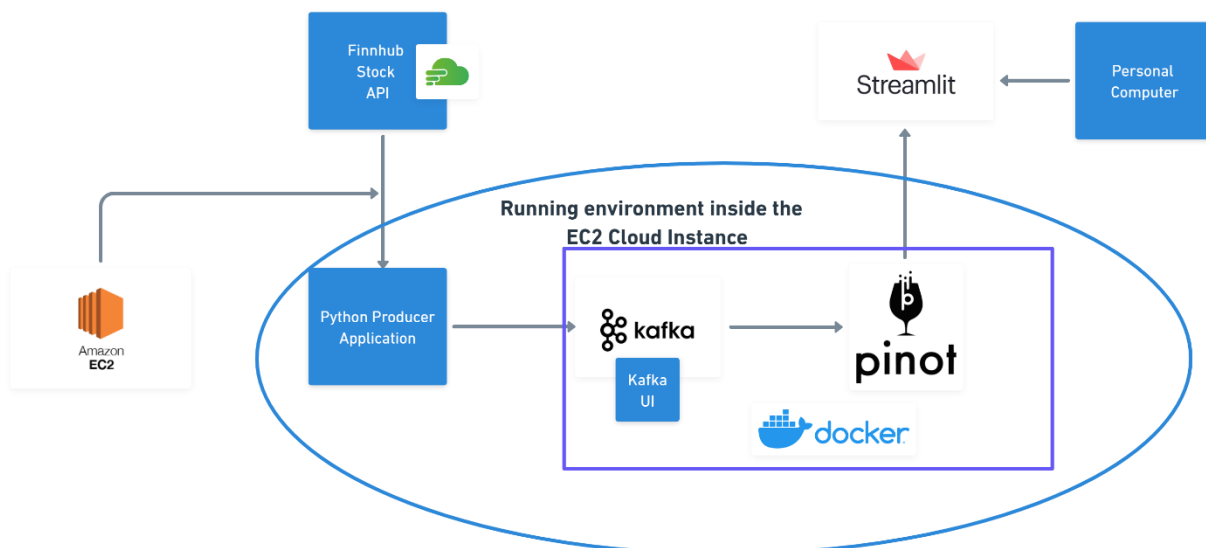


Figure 1 Project Architecture

4 Workflow

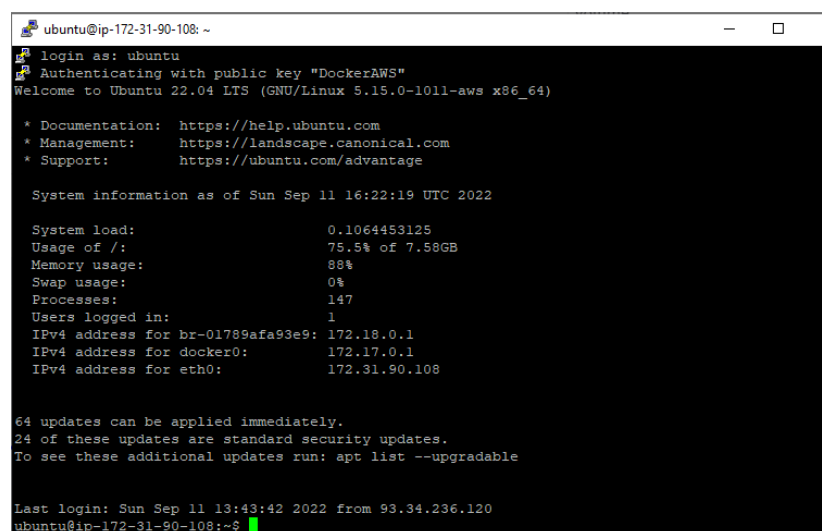
This section covers the backbone of the project, summarizing each of the conducted steps for collecting, processing, storing and visualizing the data. It is important to mention that:

> Apache Kafka and Apache Pinot, together with the Zookeeper - the centralized service for system synchronization, and the user interface for Kafka (Kafka UI) are all part of a Docker Container. Docker configures the services by keeping stable versions of each component, the input ports, and the dependencies within its boundaries. Docker purpose is to deliver software into packages that could be flexibly shared among users, avoiding package dependencies and version mismatches.

> As Docker acts as an internal machine running the listed components, the current laptop is not capable of providing the enough memory for running and deploying Docker. In addition to the memory, the student's personal laptop is not allowed to virtualization, the run of different CPUs for maintaining the process. Hence, the use of cloud services was needed. Amazon Web Services (AWS) provides EC2 instances for low on-demand prices. The install and configuration of Docker, Python, the required environment, as well as running the application producing messages to Kafka was done in the cloud instance.

4.1 System Setup

The cloud instance is accessed via a SSH client like Putty, using a defined username and an authentication key provided by Amazon Web Services. The machine, running under Ubuntu, runs the Docker container, initializing the ports for Kafka and Pinot, ready to be listened by any host name that connects to them. Without running the Python application that will retrieve the trade value data from the API, the host ports are open, and the local computer can access them, for finding an empty table in Pinot, and a configured broker in Kafka with no messages. From the project run, the porty from the cloud instance **"ec2-44-204-8-116.compute-1.amazonaws.com:28080"** for Kafka and **"ec2-44-204-8-116.compute-1.amazonaws.com:9000"** for Pinot are accessed for confirming that the cloud instance is running Docker properly. The IP of the cloud instance could change if the machine is restarted. The image below displays the terminal from the EC2 instance, with Docker and the proper environment already setup.



```
ubuntu@ip-172-31-90-108: ~  
login as: ubuntu  
Authenticating with public key "DockerAWS"  
Welcome to Ubuntu 22.04 LTS (GNU/Linux 5.15.0-1011-aws x86_64)  
  
* Documentation:  https://help.ubuntu.com  
* Management:    https://landscape.canonical.com  
* Support:       https://ubuntu.com/advantage  
  
System information as of Sun Sep 11 16:22:19 UTC 2022  
  
System load:          0.1064453125  
Usage of /:           75.5% of 7.58GB  
Memory usage:        88%  
Swap usage:           0%  
Processes:            147  
Users logged in:      1  
IPv4 address for br-01789afa93e9: 172.18.0.1  
IPv4 address for docker0: 172.17.0.1  
IPv4 address for eth0: 172.31.90.108  
  
64 updates can be applied immediately.  
24 of these updates are standard security updates.  
To see these additional updates run: apt list --upgradable  
  
Last login: Sun Sep 11 13:43:42 2022 from 93.34.236.120  
ubuntu@ip-172-31-90-108:~$
```

Figure 2 Ubuntu based cloud instance for running Kafka and Pinot

4.2 Data Collection – Finnhub API

Retrieving data about cryptocurrency recent trade values is performed by connecting to a websocket, part of the developed Python producer application, which connects to the API, asks for bitcoin data and stores the collected information as JSON strings, which is a default serialized format which Kafka receives via its dedicated brokers. The file **"stocks_to_kafka.py"** is run on the cloud instance terminal, and it collects the data and forwards it to the listening Kafka broker which is subscribed to the input stream. The message from the API source consists of a dictionary with the following information:

Variable	Description
s	Symbol, cryptocurrency or corporation
p	Last traded price
t	Timestamp in milliseconds
v	Volume amount
c	List of trade conditions (not used)

Table 2 Data variables coming from Finnhub API

The variables of interest are the traded price, the symbol or corporation and the timestamp.

4.3 Data Streaming - Kafka

Using the Kafka User Interface provides a great visibility for knowing that is happening inside the Kafka cluster. As the Python application is run, real-time data from cryptocurrency trades is processed and streamed to Kafka, which stores the information inside the defined topic across its partitions. The figure below displays the Kafka UI with some of the collected messages.

Partitions	Replication Factor	URP	In Sync Replicas	Type	Segment Size	Segment Count	Clean Up Policy	Message Count
5	1	0	5 of 5	External	15MB	5	DELETE	5581

Partition ID	Broker Leader	First Offset	Next Offset	Message Count
0	0	24061	25124	1063
1	0	22788	23872	1084
2	0	23756	24942	1186
3	0	23913	25003	1090
4	0	23102	24260	1158

Figure 3 Overview of Kafka UI with the created topic and the list of partitions

Offset	Partition	Timestamp	Key	Content
22788	1	09.11.2022 18:27:21	BINANCE:BTCUSDT1662913640611	{"c": null, "p": 21651.35, "s": "BINANCE:BTCUSDT", "t": 1662913640611, "v": ...}
22789	1	09.11.2022 18:27:21	BINANCE:BTCUSDT1662913640611	{"c": null, "p": 21651.35, "s": "BINANCE:BTCUSDT", "t": 1662913640611, "v": ...}
22790	1	09.11.2022 18:27:21	BINANCE:BTCUSDT1662913640611	{"c": null, "p": 21651.34, "s": "BINANCE:BTCUSDT", "t": 1662913640611, "v": ...}
22791	1	09.11.2022 18:27:21	BINANCE:BTCUSDT1662913640651	{"c": null, "p": 21651.34, "s": "BINANCE:BTCUSDT", "t": 1662913640651, "v": ...}
22792	1	09.11.2022 18:27:21	BINANCE:BTCUSDT1662913640655	{"c": null, "p": 21651.34, "s": "BINANCE:BTCUSDT", "t": 1662913640655, "v": ...}
22793	1	09.11.2022 18:27:21	BINANCE:BTCUSDT1662913640662	{"c": null, "p": 21651.3, "s": "BINANCE:BTCUSDT", "t": 1662913640662, "v": ...}
22794	1	09.11.2022 18:27:21	BINANCE:BTCUSDT1662913640662	{"c": null, "p": 21651, "s": "BINANCE:BTCUSDT", "t": 1662913640662, "v": ...}

Figure 4 Inflow stream of data from the Finnhub API about cryptocurrency trade values

4.4 Data Storage - Pinot

Apache Pinot is also configured inside the Docker container, and launched by the high-memory cloud instance. Pinot gets configured by defining a table and a connection schema. The connection schema defines the table name and the input stream source. In this case, the name of the Kafka topic is specified, along with the Kafka port to listen. The table schema defines the column names and datatypes for the Pinot data tables. Pinot is configured with the table being called “*stockevents*”, and the columns are “*symbol*”, “*price*”, “*volume*”, and “*ts*”, with string-double-double-timestamp as datatypes.

The Python producer application is left running in a separate terminal on the cloud instance, so by accessing the Pinot port of the EC2 machine, the created table can be visualized, together with a query table for accessing the desired entries.

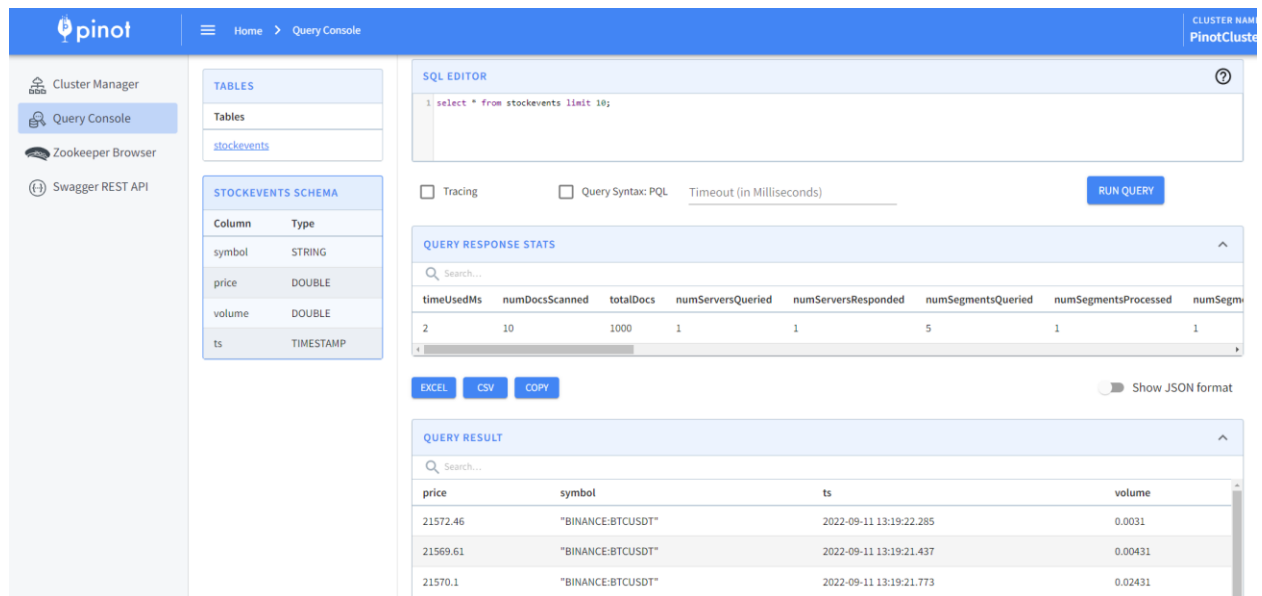


Figure 4 Pinot Interface displaying queried information

Examples of queries are the following ones. It is important to mention that the queries are going to be copied for the Streamlit application.

- **select * from stockevents WHERE symbol LIKE '%BTC%' limit 20;**
- **select * from stockevents limit 20;**

4.5 Data Query and Visualization - Streamlit

The retrieved real-time data has been streamed to Kafka and stored in Pinot, now it is time for presenting it to the user. Streamlit is a web application framework compliant with Python for configuring in fast way a front-end platform for visualizing information [10]. Streamlit is launched by creating a Python file which connects to the Pinot port of the cloud instance, and performs queries on the data tables. The information is stored as Pandas dataframe format and plotted using the *plotly* library. Every step involving streamlit is run inside the local machine, so it gets a total independence from the code related to Kafka and Pinot, which resides on the cloud instance.

The Python file “stock_prices_app.py” is launched inside the terminal of the local machine, and by accessing the dedicated port 8501, the front-end visualization is provided to the user. The development of the front-end is a simple design containing a dropdown menu, an auto-refresh module and the visualization.

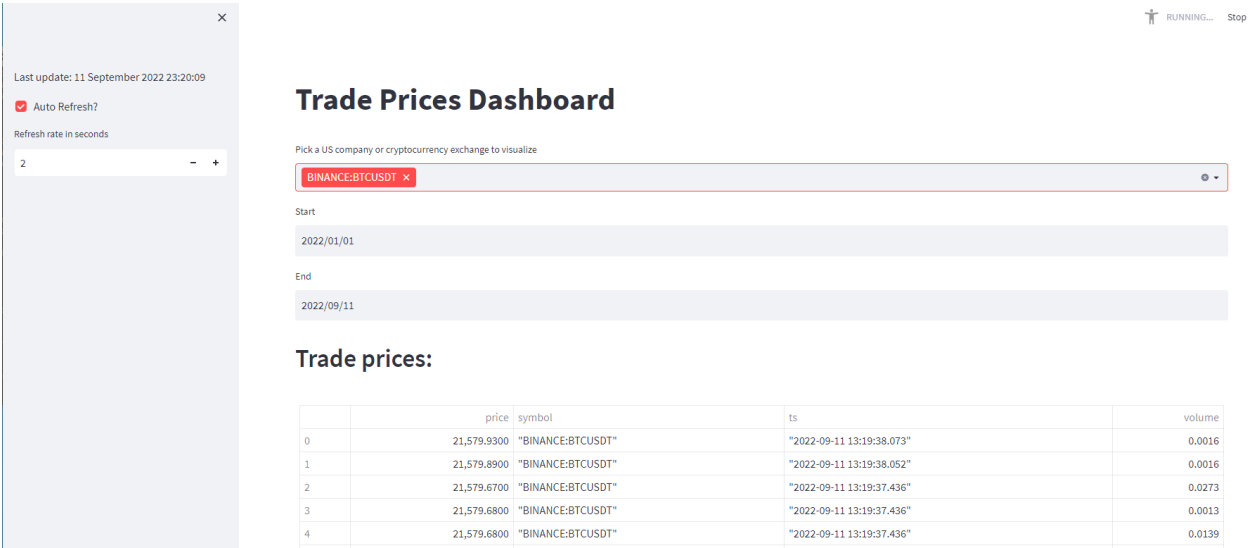


Figure 5 Streamlit Design

The next two figures describe the real-time values for volume and price of the Binance cryptocurrency exchange. One positive feature of the front-end application is the refresh and immediate update of the visualization by modifying the query or the Streamlit variables inside the “stock_prices_app.py” file.



Figure 6 Streamlit visualization of volume for Binance trade values

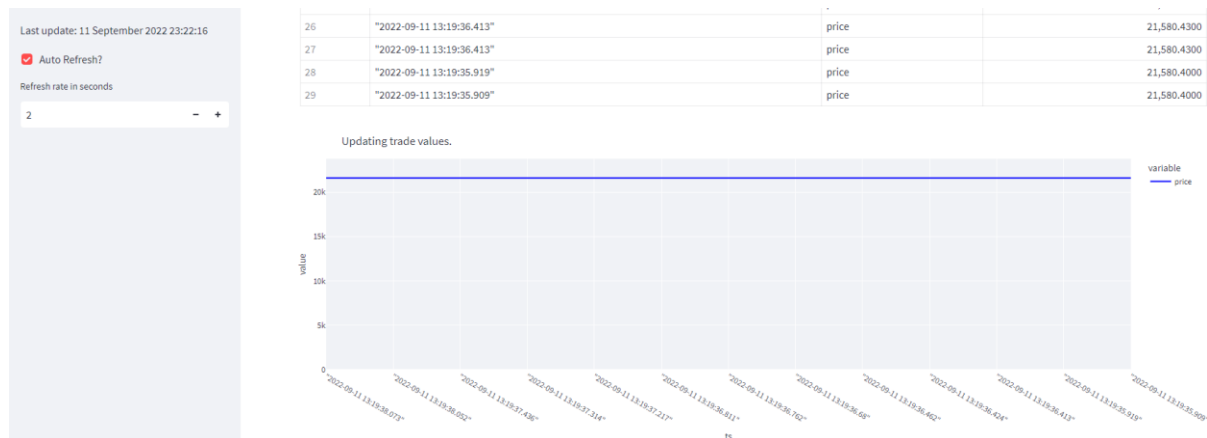


Figure 7 Streamlit visualization of price for Binance trade values

5 Lessons Learned

An interesting workflow has been developed for retrieving, preprocessing, storing and visualizing real-time data about cryptocurrency stock exchange. In the project development there are some lessons learned that are significant to be highlighted:

- The consideration of memory consumption, for having in advance the proper machine to deploy the project.
- The use of Docker for having most of the project containerized, with dependencies and components living in the same environment.
- The project makes a good user of an event streaming service, like Kafka, and a datastore service, like Pinot. Streamlit adds an easy interface platform, as the use of another front-end application like Angular, NodeJS or Flask would have required more developing time.
- The project can be easily extended to add more corporation trade values, for plot comparisons between different organizations.

References

- [1] <https://companiesmarketcap.com/usa/largest-companies-in-the-usa-by-market-cap/>
- [2] <https://www.marketbeat.com/stock-market-holidays/>
- [3] <https://finnhub.io/docs/api/websocket-trades>
- [4] <https://github.com/Finnhub-Stock-API/>
- [5] <https://aws.amazon.com/free/>
- [6] <https://analyticshut.com/kafka-producer-and-consumer-in-python/>
- [7] <https://docs.pinot.apache.org/>
- [8] <https://blog.streamlit.io/how-to-build-a-real-time-live-dashboard-with-streamlit/>

- [9] <https://aiven.io/blog/teach-yourself-apache-kafka-and-python-with-a-jupyter-notebook/>
- [10] <https://markhneedham.medium.com/analysing-github-events-with-apache-pinot-and-streamlit-2ed555e9fb78/>
- [11] <https://medium.com/elca-it/streamlit-vs-dash-and-beyond-an-introduction-to-web-dashboard-development-frameworks-for-python-fec59d835784/>