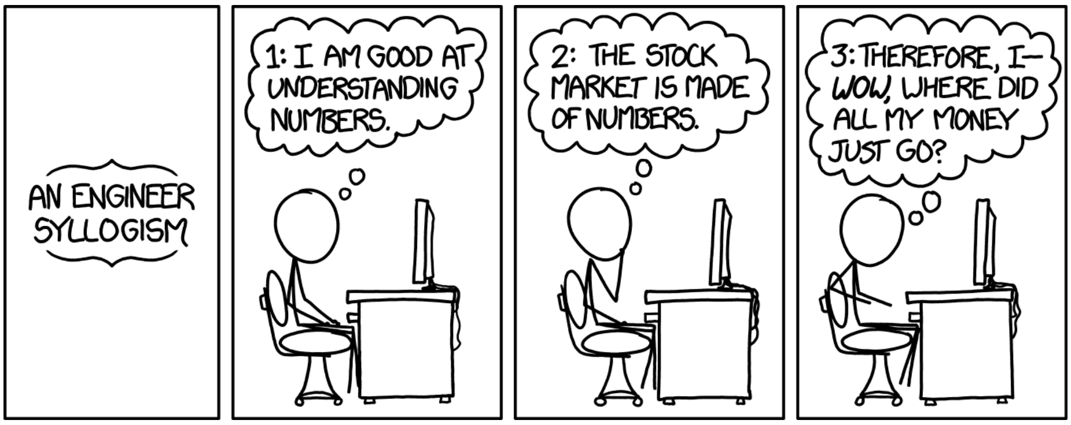
Data Engineering bootcamp (June 2023)

OPA Project technical report

https://xkcd.com/1570/


AN ENGINEER SYLLOGISM
1: I AM GOOD AT UNDERSTANDING NUMBERS.
2: THE STOCK MARKET IS MADE OF NUMBERS.
3: THEREFORE, I— WOW, WHERE DID ALL MY MONEY JUST GO?

Julien Hervé

Table of Contents

Project scope

The expected milestones and goals have been specified in a Google Document written by the Datascientest team[1].

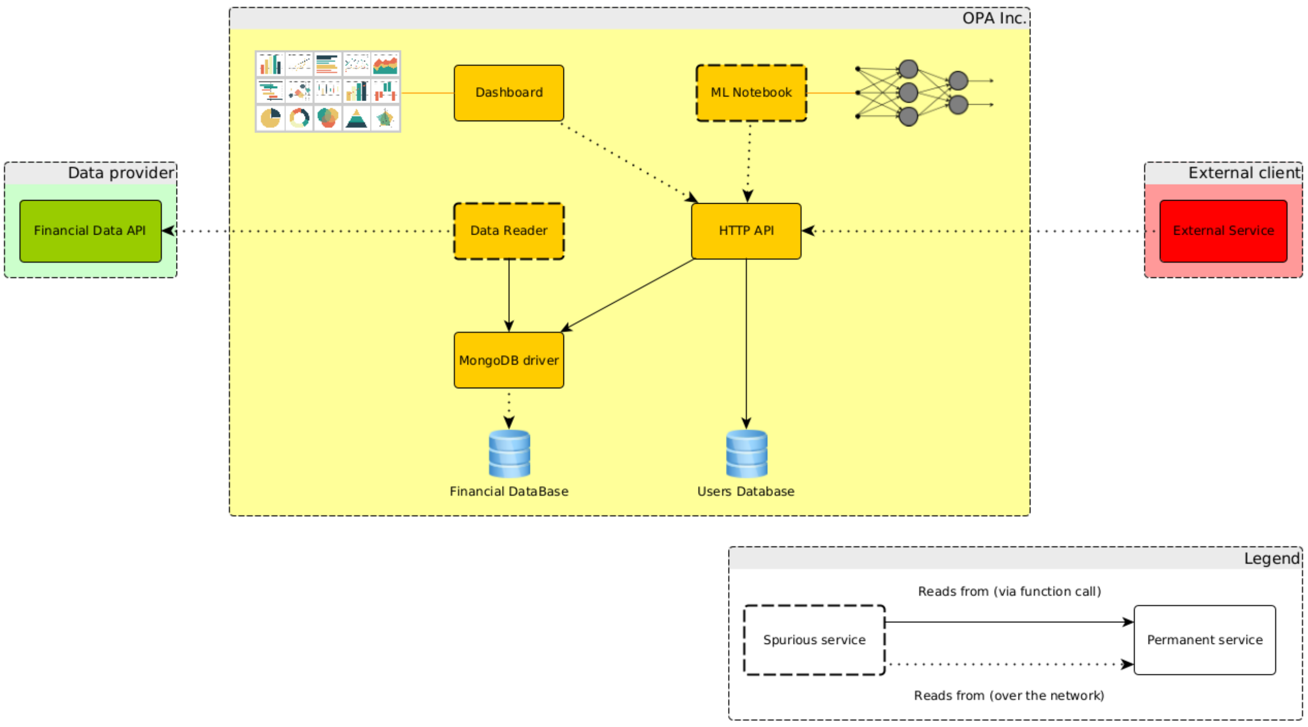
Basically the project aims at setting up a full data pipeline that covers the following aspects :

- retrieving data from an external API
- storing that data locally
- ensure it is accessible to potential internal consumers
- consume that data to train a Machine Learning model or display it in a meaningful way
- package and containerise all the code produced to make it reusable and extensible

The data used will be stock market values from various companies.

The overall architecture is outlined in the diagram below ; the yellow blocks within the "OPA Inc." group are to be built within the course of the project and will be presented in the remaining sections.

Project architectural overview



1. https://docs.google.com/document/d/1UEL9wexhETO2MXpxvIPwVRE4WLfLOOi3YF43Zn8L5sk

Financial data retrieval

Scope

Within this project, we will be looking at the evolution of stock market values of several companies.

In an ideal world, the ultimate goal would be to store as much fine-grained data for as large a period of time as possible ; this would allow for much finer tuning of any Machine Learning model we would train. But this is costly both in terms of data storage and CPU usage and is considered out of scope.

We will therefore restrain our focus on two kinds of data within this project :

"historical" data

- spans long periods of time
- adjusted and corrected afterwards
- quite coarsed-grained (one data point every few hours / every day)

"streaming" data

- very recent (can be retrieved almost in real time)
- usually quite fine-grained (one data point every few minutes)

API provider choice

Dozens of API providers exist for stock market data. Some of them are geared towards profesionnals and provide very fine data over long ranges of time. Access to real-time / slightly delayed data is also of prime value and usually requires proper licensing from the stock exchange authorities[2].

In the relatively basic scope of this project, we have been focusing on some providers that allow a free access to their API. All of them are mainly aimed at US markets.

Some stock market data providers

MarketStack

https://marketstack.com/

Alphavantage

https://www.alphavantage.co/

EODHD

https://eodhistoricaldata.com/

FMP Cloud

https://fmpcloud.io/

Yahoo Finance

2. https://www.alphavantage.co/realtime_data_policy/

The official Yahoo Finance API seems to have been discontinued but it is still working and its API can be retro-engineered from e.g. this unofficial Python client : https://github.com/ranaroussi/yfinance

In terms of the kind of stock market data they give access to, they are virtually identical. The main differences lie in the limits they set on request rate and the information they require for registration and retrieval of an API key. The table below sums up their main differences.

Table 1. Comparison of stock Market data providers

Company	Registration	API limits	API documentation
MarketStack	Email + name + phone	100 requests/month	Yes
Alphavantage	Email	5 requests / minute, 500 requests / day	Yes
Yahoo Finance	No	Unrestricted in practice	No
EODHD	Google OR Github OR email + name	20 requests / day	Yes
FMP Cloud	Google OAuth2 OR email/password	250 requests / day	Yes

Once the project is developed and released in production, a few requests per day (one per company we track) should be enough. This already rules out MarketStack, which is too restricted.

In the development phase though, we'll be making request to the APIs much more often than that, and hitting the API rate should not be a blocking issue. This rules out EODHD and its maximum of 20 requests per day.

Yahoo Finance does not have a proper API documentation and is almost too easy to use with the unofficial Python client `yfinance`.

This leaves us with Alphavantage or FMP Cloud.

Examples of data

Once in possession of an API key, retrieving data from those providers is done via a simple `GET` request on the appropriate endpoint. The ticker[3] is set either in some part of the URL path (e.g. https://fmpcloud.io/api/v3/historical-chart/15min/AAPL?apikey=fmp_secret_key), or as a query string parameter (e.g. https://www.alphavantage.co/query?function=TIME_SERIES_INTRADAY&interval=15min&symbol=AAPL&apikey=alpha_secret_key). The body of the server response contains the data encoded in JSON format.

Below are 4 examples of data mainly retrieved from FMP Cloud, retaining only some records for brievety (a typical response contains hundreds to thousands of records).

3. The ticker is a short string used to uniquely identify a company on stock markets ; e.g. Microsoft is MSFT, Intel is INTC, … In this report, examples will be using the ticker AAPL (for Apple)

- deep historical data that spans about 40 years but with little detail
- detailed historical data that spans only a few years but with great detail
- streaming data
- streaming data from Alphavantage, for comparison

FMPCloud deep historical data[4]

```
{
    "historical": [
        {
            "close": 190.69,
            "date": "2023-07-14"
        },
        {
            "close": 190.54,
            "date": "2023-07-13"
        },
        ...
        {
            "close": 0.12165,
            "date": "1980-12-15"
        },
        {
            "close": 0.12835,
            "date": "1980-12-12"
        }
    ],
    "symbol": "AAPL"
}
```

4. Retrieved from https://fmpcloud.io/api/v3/historical-price-full/AAPL?serietype=line&apikey=fmp_secret_key

FMP Cloud detailed historical data[5]

```
{
    "historical": [
        {
            "adjClose": 190.690002,
            "change": 0.46,
            "changeOverTime": 0.0024181,
            "changePercent": 0.24181,
            "close": 190.69,
            "date": "2023-07-14",
            "high": 191.1799,
            "label": "July 14, 23",
            "low": 189.63,
            "open": 190.23,
            "unadjustedVolume": 40764621,
            "volume": 40835691,
            "vwap": 190.46
        },
        {
            "adjClose": 190.539993,
            "change": 0.04,
            "changeOverTime": 0.0002099738,
            "changePercent": 0.02099738,
            "close": 190.54,
            "date": "2023-07-13",
            "high": 191.19,
            "label": "July 13,23",
            "low": 189.78,
            "open": 190.5,
            "unadjustedVolume": 41342300,
            "volume": 41337338,
            "vwap": 190.57
        }
    ],
    "symbol": "AAPL"
}
```

5. Retrieved from https://fmpcloud.io/api/v3/historical-price-full/AAPL?apikey=fmp_secret_key

FMP Cloud streaming data[6]

```
[
    {
        "close": 190.69,
        "date": "2023-07-14 16:00:00",
        "high": 190.71,
        "low": 190.3978,
        "open": 190.69,
        "volume": 1654688
    },
    {
        "close": 190.72,
        "date": "2023-07-14 15:45:00",
        "high": 190.74,
        "low": 190.26,
        "open": 190.42,
        "volume": 3749214
    },
    {
        "close": 190.415,
        "date": "2023-07-14 15:30:00",
        "high": 190.49,
        "low": 190.16,
        "open": 190.275,
        "volume": 1254758
    },
    {
        "close": 190.275,
        "date": "2023-07-14 15:15:00",
        "high": 190.42,
        "low": 190.04,
        "open": 190.04,
        "volume": 1315560
    }
]
```

6. Retrieved from: https://fmpcloud.io/api/v3/historical-chart/15min/AAPL?apikey=fmp_secret_key

Alphavantage streaming data[7]

```
{
    "Meta Data": {
        "1. Information": "Intraday (15min) open, high, low, close prices and volume",
        "2. Symbol": "AAPL",
        "3. Last Refreshed": "2023-07-14 19:45:00",
        "4. Interval": "15min",
        "5. Output Size": "Compact",
        "6. Time Zone": "US/Eastern"
    },
    "Time Series (15min)": {
        "2023-07-14 19:15:00": {
            "1. open": "190.8000",
            "2. high": "190.8100",
            "3. low": "190.7500",
            "4. close": "190.7600",
            "5. volume": "2896"
        },
        "2023-07-14 19:30:00": {
            "1. open": "190.7600",
            "2. high": "190.7600",
            "3. low": "190.7000",
            "4. close": "190.7100",
            "5. volume": "2211"
        },
        "2023-07-14 19:45:00": {
            "1. open": "190.7300",
            "2. high": "190.7500",
            "3. low": "190.6900",
            "4. close": "190.7200",
            "5. volume": "11717"
        }
    }
}
```

From those examples we can observe that both historical and streaming data from any provider follow the same schema. They define *units of time* over which some values are given.

A given data record will always contain at least the following elements :

Date

The moment in time for which the values are provided

Interval length

The duration over which those values are computed (15 minutes, 24 hours, …)

Close

The value at the end of the unit of time

7. Retrieved from: https://www.alphavantage.co/query?function=TIME_SERIES_INTRADAY&interval=15min&symbol=AAPL&apikey=alpha_secret_key

And they will optionally contain the following additional elements :

Open

The value at the start of the unit of time

Low

The lowest value over the unit of time

High

The highest value over the unit of time

Volume

The number of stocks exchanged over the unit of time

In terms of data format, the *date* is (or can be trivially converted to) a `datetime`. The interval length can be represented as a number of minutes expressed in `int`. The *close*, *open*, *low*, *high* values are prices in dollars, of type `float`. The volume is an `int`.

Table 2. Fields contained in a typical stock value record

Value	Type	Optional?
date	`datetime`	no
interval	`int`	no
close	`float`	no
open	`float`	yes
high	`float`	yes
low	`float`	yes
volume	`int`	yes

Data representation

Now that we have a better understanding of the kind of data we will be handling, we can define a generic class to represent both historical and streaming stock value.

Definition of StockValue class

```
class StockValue(BaseModel):
    ticker: str
    date: datetime
    close: float
    interval: int
    open: float | None = None
    low: float | None = None
    high: float | None = None
    volume: int | None = None
```

Access to providers' data

An abstract class is defined to retrieve data from a provider, with a method that can get either streaming or historical data.

Definition of StockMarketProvider abstract class

```python
class StockMarketProvider(ABC):
    @abstractmethod
    def get_stock_values(
        self,
        ticker: str,
        kind: StockValueKind,
        granularity: StockValueSerieGranularity,
    ) -> list[StockValue]:
        ...

    def get_raw_stock_values(
        self,
        ticker: str,
        kind: StockValueKind,
        granularity: StockValueSerieGranularity,
    ) -> dict:
        raise NotImplementedError()

    @abstractmethod
    def get_company_info(self, tickers: list[str]) -> list[CompanyInfo]:
        ...
```

The implementation of the class that gets data from FMP Cloud provider can be found in fmp_cloud.py.

Data validation

To ensure that the data we get from the APIs is in a format that we expect, we use the `pydantic` library[8] to define some models of the data.

8. https://docs.pydantic.dev/2.0/

```python
class FmpCloudSimpleValue(BaseModel, StockValueMixin):
    date: date
    close: float

class FmpCloudSimpleCoarseData(BaseModel):
    symbol: str
    historical: list[FmpCloudSimpleValue]

class FmpCloudOhlcValue(BaseModel, StockValueMixin):
    date: datetime | date
    open: float
    close: float
    low: float
    high: float
    volume: int

class FmpCloudOhlcCoarseData(BaseModel):
    symbol: str
    historical: list[FmpCloudOhlcValue]
```

Validation of the server's response can now be done by simply instantiating one of those classes with e.g. `FmpCloudHistoricalData(**json)`.

Handling of secret API keys

The API keys are confidential and extra-care has been taken to not leak them into version control or hardcode them.

Those secrets are handled via project configuration, and relies either on the existence of a `app_data/secrets` directory at the root of the projet or a `/secrets` directory (when the project is run via Docker Compose with Docker Secrets).

Data storage

Scope

From Financial data retrieval, we now have a good grasp of the kind of data we will be handling within this project. So far we have only been retrieving it via an external API, and it would be very impractical to keep on doing so.

It is cumbersome indeed to rely on HTTP requests to an external provider. It is extremely slow, and the number of requests we can issue is limited. Chances are we will often be requesting the exact same sets of data. And last but not least, we have no way to organize and query the data to extract the patterns we might be looking for in a machine learning model or simply for data visualisation.

A better approach would be to retrieve all the data we might need by emitting those API requests once and for all and storing the data contained in their response in an organized way ; we could update it when necessary.

Any automated tool or data scientist that may need that data will now use that cache.

Constraints

We have seen that the financial data we aim to retrieve is always shaped basically in the same way, whether it is historical or streaming data. Some fields are not always present, though.

Definition of an abstract StockValue class

```
class StockValue(BaseModel):
    ticker: str
    date: datetime
    close: float
    interval: int
    open: float | None = None
    low: float | None = None
    high: float | None = None
    volume: int | None = None
```

Though they is no numerical ID, short strings named "tickers" or "symbols" in stock market jargon are used to identify a given stock on a stock market[9] ; they are unique by design and can therefore be safely used to identify a given company.

The data does not exhibit any complex relationship patterns either. There is a 1-to-n relationship between companies and stock values. And any other additional information we might need in the future will likely be related to the company as well.

Being the measurement of a stock value at a given moment, the data is intrinsically time-related and exhibit all the characteristics of time-series data.

There is no hard constraint in terms of data consistency or durability. The stored data will be merely a cache of the data available in external APIs, and therefore not critical in any sense. No original data is ever

9. https://en.wikipedia.org/wiki/Ticker_symbol

produced by our applications ; in the worst case the data is still available from the APIs and one can always retrieve from them.

In terms of data access patterns, the values stored in the database will be written to only once (when it is retrieved from the API) and read many times.

Choice of a database

So many database options exist that we will first restrict our choice by choosing among the various types of databases, that is :

1. Relational
2. Graph-oriented
3. Column-oriented
4. Key-value
5. Document-oriented

The choice here is that of a design philosophy, since the lines are quite blurred between the different categories ; e.g. almost all SQL databases have a very good support of JSON, making them very capable as "document-oriented" databases.

As we have seen, the data we'll be handling has very few relational properties. A fully-fledged SQL relational database is therefore not the favored option. It does not exhibit any interesting graph properties either, rendering a graph-oriented database useless.

Column-oriented databases are more of an option for e.g. OnLine Analytical Processing of data with complex relationships, which is not the case here.

Key-value databases trade expressiveness in data structure / query for very high performance. They might be a reasonable option given the simple structure of our data.

Overall, though, we do not really need that very high performance, and we'll be happy to trade a bit of performance for the light data structuring that document-oriented databases offers. We might also need the expressiveness they allow in terms of query / aggregations / …

A document-oriented datastore therefore appears like a good choice. MongoDB[10] is the most popular implementation of such a database. It supports time-series collections[11] of which stock prices are a prime example. It also features schema validation[12] to ensure that the documents we insert are in an expected format.

Implementation

Should we change our mind regarding the database choice, we'll keep some flexibility by defining a `Storage` class with an API that will abstract away the concrete implementation of the database.

10. https://www.mongodb.com/
11. https://www.mongodb.com/docs/v6.0/core/timeseries-collections/
12. https://www.mongodb.com/docs/manual/core/schema-validation/

Definition of an abstract Storage class

```
class Storage(ABC):
    @abstractmethod
    def insert_values(self, values: list[StockValue]):
        ...

    @abstractmethod
    def get_values(
        self, ticker: str, kind: StockValueKind, limit: int = 500
    ) -> list[StockValue]:
        ...

    @abstractmethod
    def get_all_tickers(self) -> list[str]:
        ...

    @abstractmethod
    def insert_company_infos(self, infos: list[CompanyInfo]):
        ...

    @abstractmethod
    def get_company_infos(self, tickers: list[str]) -> dict[str, CompanyInfo]:
        ...

    @abstractmethod
    def get_stats(self, kind: StockValueKind) -> dict[str, StockCollectionStats]:
        ...
```

The layout of the MongoDB database itself will be setup using the approach recommended of the MongoDB Docker image's page[13], via a `mongo-init.js` file executed on the first startup.

We will define as many collections as we have different kinds of data ; one for historical stock values, one for streaming/recent stock values, one for additional company information, ...

Each of these collections has a unique index to ensure that no data is duplicated.

Definition of unique indexes

```
db["stock_values"].create_index({"date": 1, "ticker": 1, "interval": 1}, unique=True)
```

We also leverage MongoDB's schema validation feature to ensure that the inserted data always matches a well-defined set of rules defined from the data format exposed in Fields contained in a typical stock value record. Those rules include ensuring that some fields are ALWAYS present (`ticker`, `date` and `close` values) ; and that fields should have the expected type (mostly `double`, with some `date` or `int`).

13. "Initializing a fresh instance" paragraph on https://hub.docker.com/_/mongo

Definition of constraints

```
db.create_collection(
  "stock_values",
  {
    "validator": {
      "$jsonSchema": {
        "bsonType": "object",
        "title": "Stock values validation",
        "required": ["_id", "date", "close", "ticker", "interval"],
        "properties": {
          "_id": {"bsonType": "objectId", "description": "'_id' must be a objectId"},
          "date": {"bsonType": "date", "description": "'date' must be a date"},
          "ticker": {"bsonType": "string", "description": "'ticker' must be a string"},
          "interval": {"bsonType": "int", "description": "'interval' must be a int"},
          "close": {"bsonType": "double", "description": "'close' must be a double"},
          "open": {"bsonType": "double", "description": "'open' must be a double"},
          "low": {"bsonType": "double", "description": "'low' must be a double"},
          "high": {"bsonType": "double", "description": "'high' must be a double"},
          "volume": {"bsonType": "int","description": "'volume' must be a int"},
        },
        "additionalProperties": False,
      }
    }
  },
)
```

Data report

Scope

This stage aims at the implementation of a dashboard that would display financial data of interest.

One can take inspiration from common financial tools to see what kind of visualization is usually used. Using the indicators labelled as "commonly used" on e.g. Yahoo Finance[14] we can highlight a few characteristic graphs, such as :

- a simple trend line that joins the close values

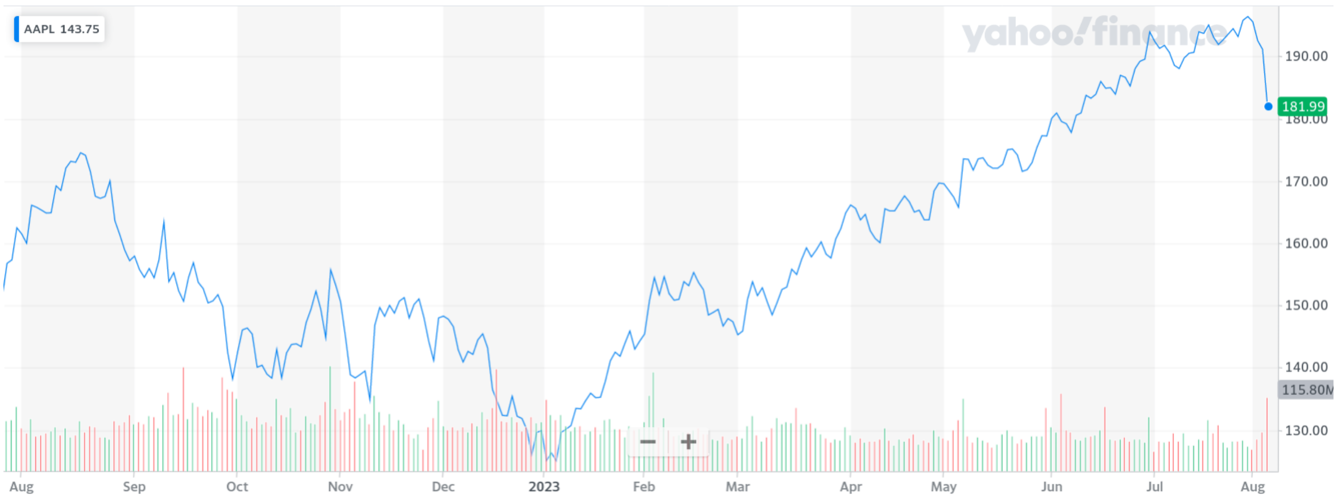


Figure 1. A trend line drawn from "close" value[15]

- a chart that displays open/close/high/low values with candlesticks[16]. Green/red colors indicate whether the stock value has raised over that interval.

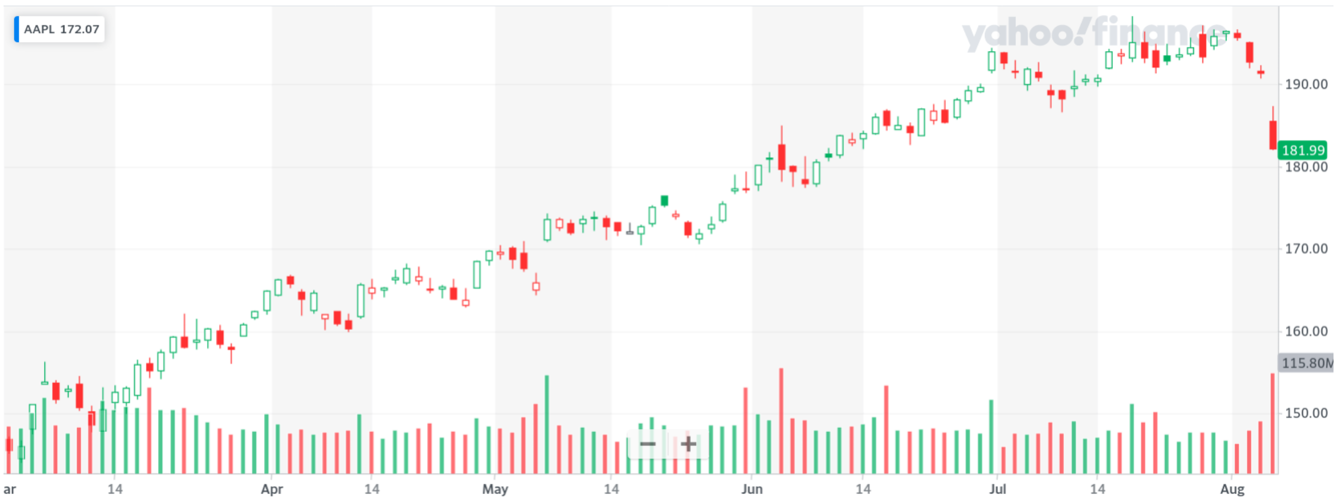


Figure 2. A candlestick graph[17]

14. https://finance.yahoo.com/chart/AAPL?guccounter=1
15. retrieved from https://yhoo.it/3YrLHr2
16. https://datavizcatalogue.com/methods/candlestick_chart.html
17. retrieved from https://yhoo.it/3QuHpwW

- a chart that displays the same information with bars[18]

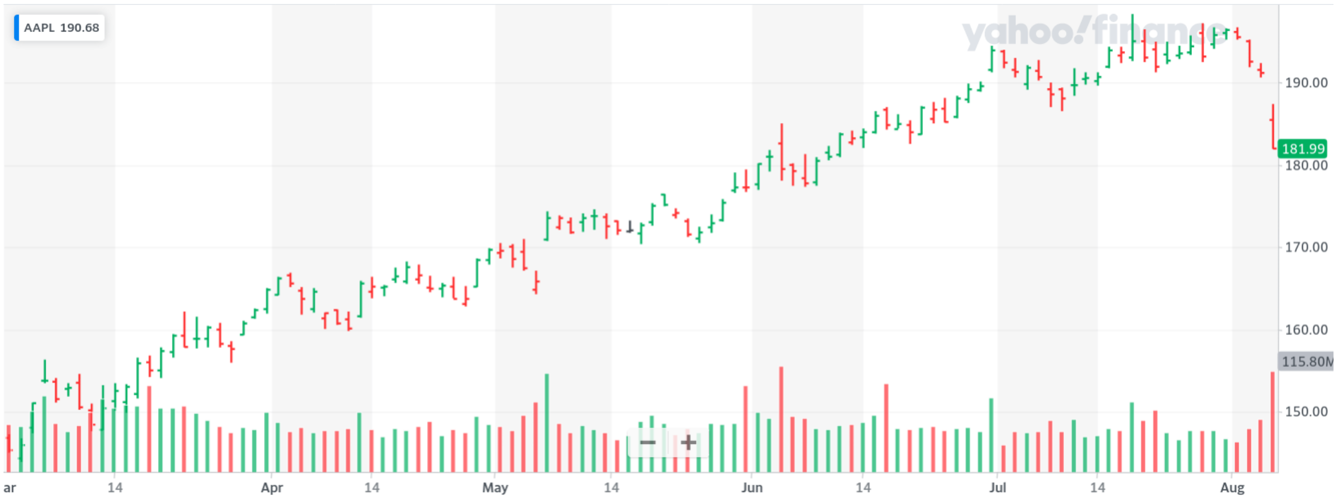


Figure 3. A bar graph[19]

- a chart with values smoothed via a moving average over several days

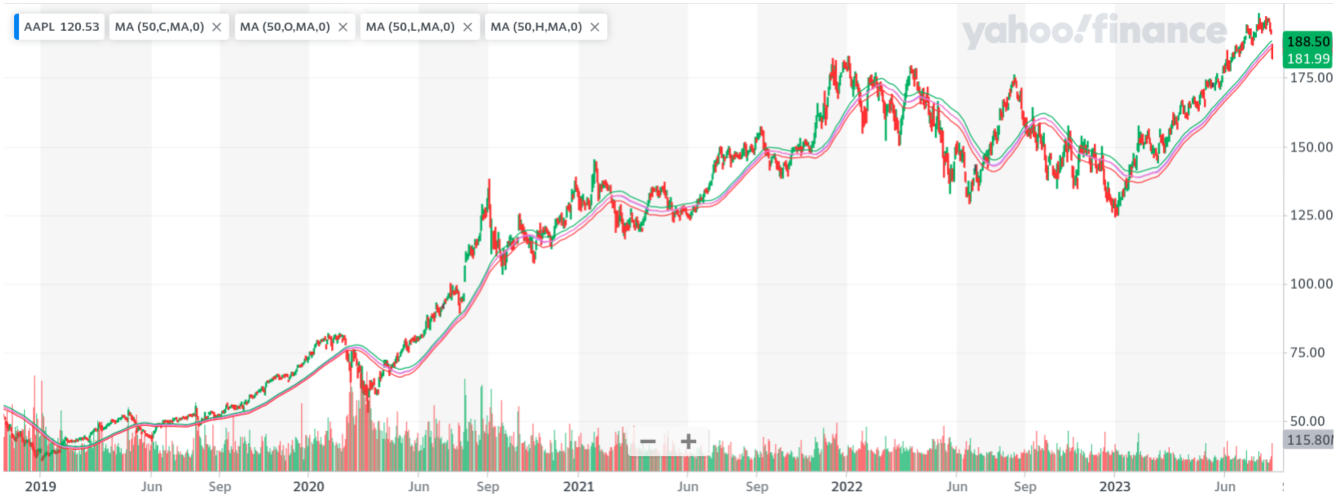


Figure 4. Moving averages[20]

- a chart with high/low bands (indicating the extremas of the stock value)

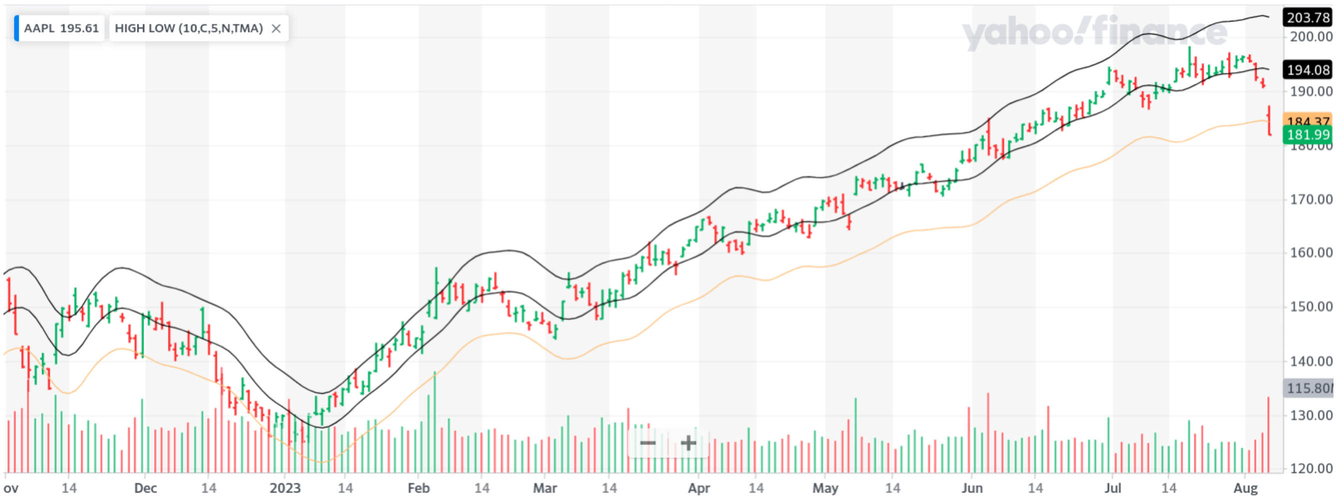


Figure 5. High-low bands[21]

18. https://datavizcatalogue.com/methods/OHLC_chart.html
19. retrieved from https://yhoo.it/3YrreCH
20. retrieved from here

We can also notice that charts by default include a bar chart that indicates the volume of stocks exchanged on the market on a given period.

We will try to reproduce those kinds of charts in the project. More elaborate visualizations are obviously available, but they are out of scope.

Tool

To realize this step we will be using Dash[22], an open-source data-visualization framework that implement React components over the Javascript library Plotly.js[23] (by using Plotly.py[24] as a Python API).

It aims at letting users to write interactive data visualizations by writing nothing else but Python code ; this is achieved by a mechanism of callbacks that can be triggered on mutation of any part of the page layout, and allows to update any part of the page as a response.

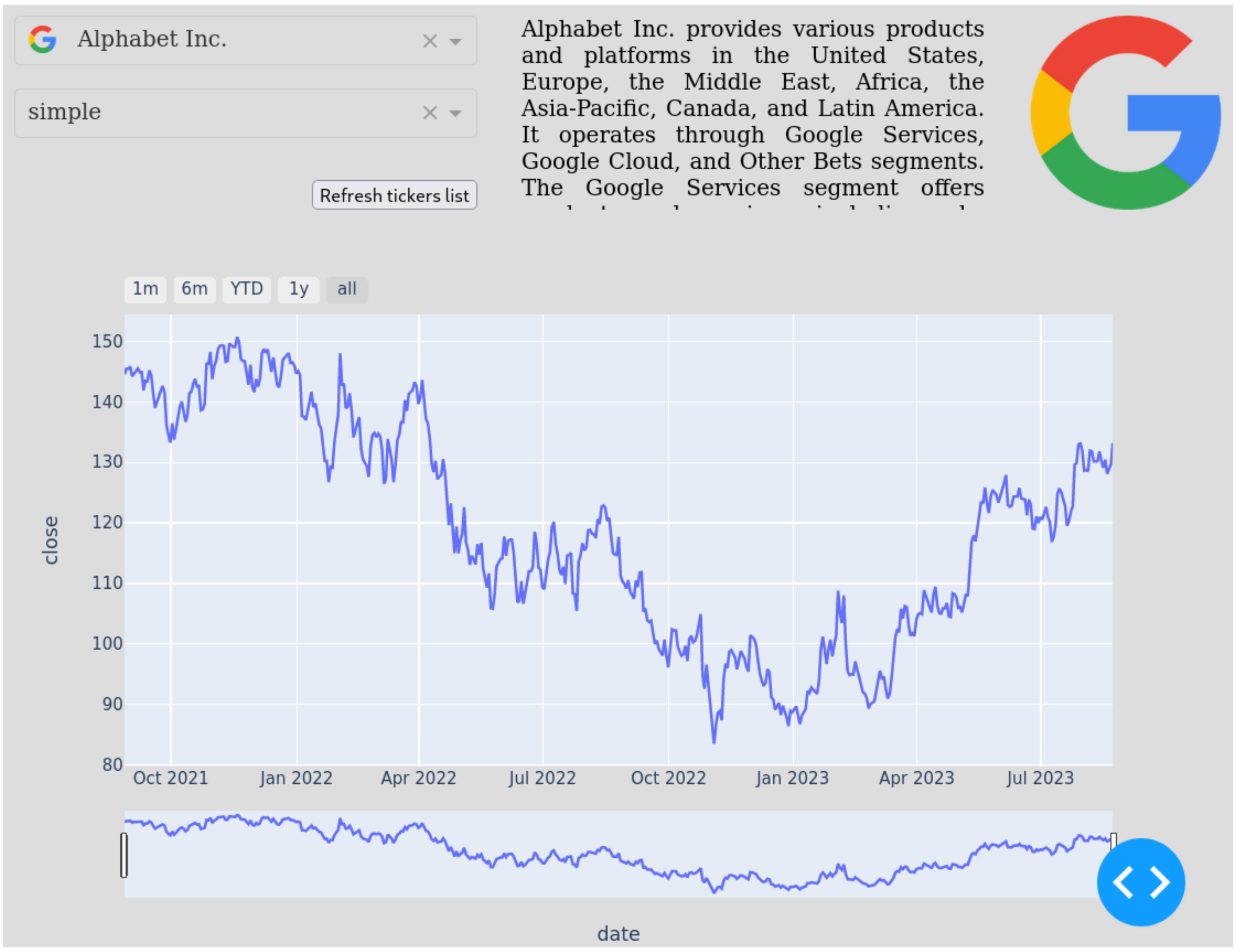
It is widely used, backed up and supported by a dedicated company, and has a huge community.

Implementation

The kind of charts we aim to draw are very well supported by Dash. Line charts are obviously super simple to implement, and candlestick/OHLC graphs are also available out-of-the-box. Charts that imply e.g. moving averages have not been implemented though, because they require a bit more work in terms of data manipulation.

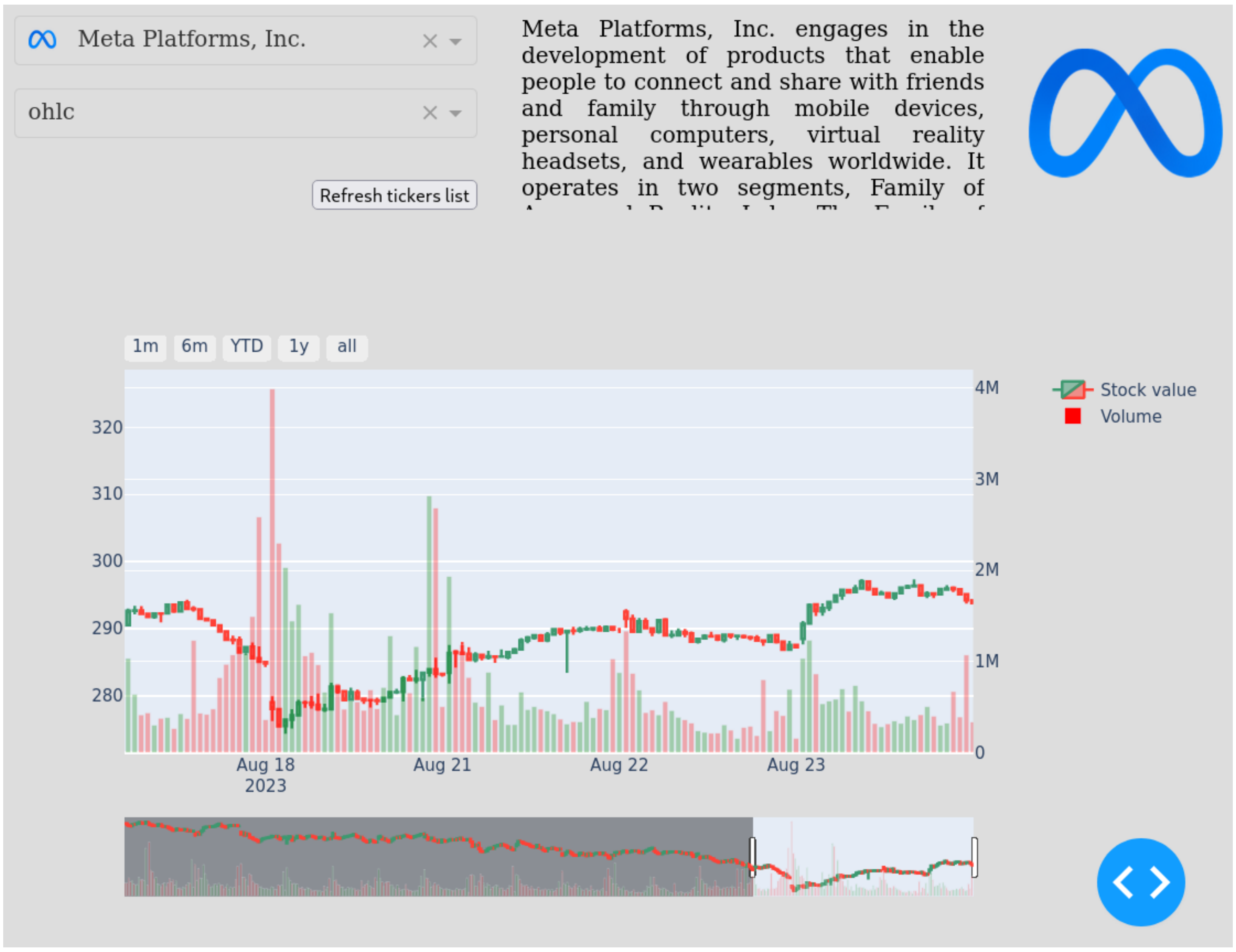
The UI we have chosen is quite simple. The graph displays only one kind of data at a time (e.g. OHLC data for Google), and 2 dropdowns allow to switch company and to switch between a trend line and OHLC chart (which also displays volume). Hovering over the graph displays the values at the current time. Navigation through the graph can be done via a range selector and some buttons that allow to display e.g. 1 month or 1 year of data.

21. retrieved from https://yhoo.it/3OHdArD
22. https://dash.plotly.com/
23. https://github.com/plotly/plotly.js
24. https://plotly.com/python/



Alphabet Inc. provides various products and platforms in the United States, Europe, the Middle East, Africa, the Asia-Pacific, Canada, and Latin America. It operates through Google Services, Google Cloud, and Other Bets segments. The Google Services segment offers

Figure 6. Dashboard displaying historical close value for Alphabet



Meta Platforms, Inc. engages in the development of products that enable people to connect and share with friends and family through mobile devices, personal computers, virtual reality headsets, and wearables worldwide. It operates in two segments, Family of

Figure 7. Dashboard displaying OLHC values for Meta

Care has been taken to display the data in a visually pleasant way, using e.g. rangebreaks that allow to present the timeseries in a continuous way even though they are not (the stock market opens only from 9:00 to 16:30 on weekdays). The layout has been slightly altered using some CSS.

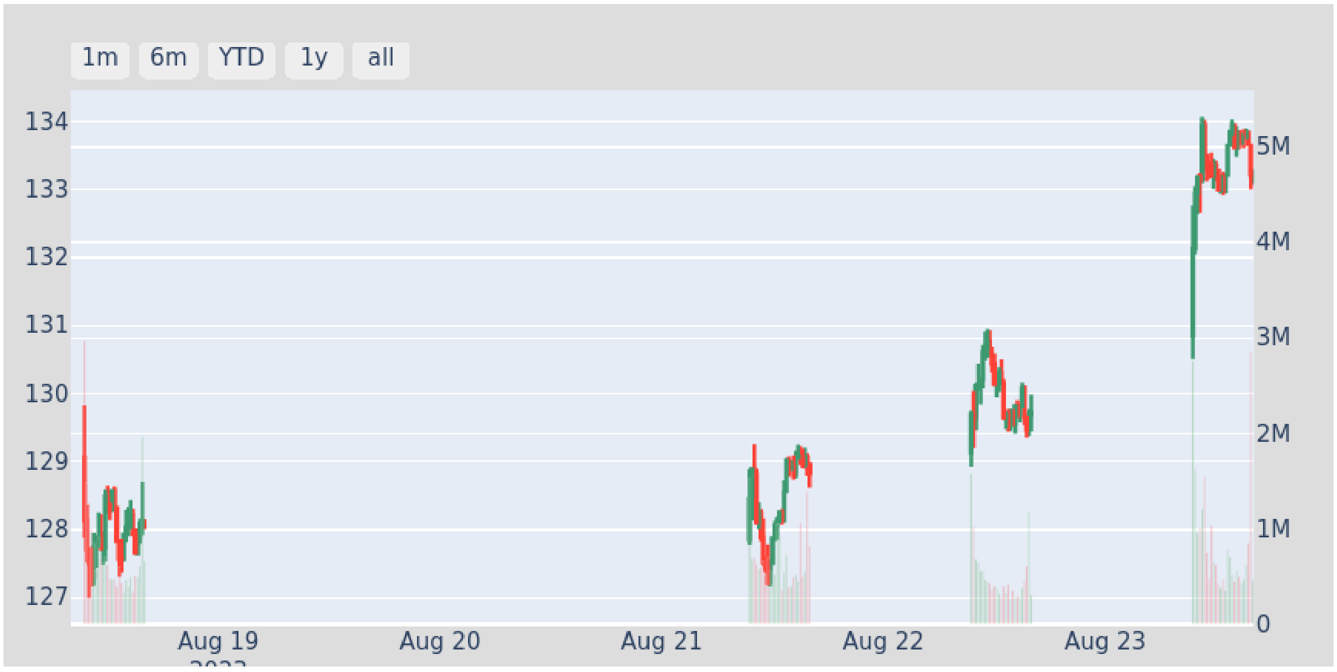


Figure 8. Dashboard without rangebreaks

Limitations and alternative solutions

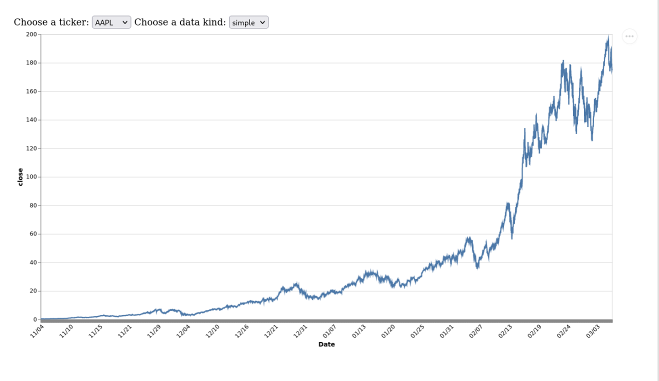
Though quite intuitive and easy to use with its callback system, Dash was found to be extremely slow when it comes to rendering and displaying graphs with a large number of points ; especially OHLC charts that display a long period of time take full seconds to render (of which only about 0.5 second is taken to update the JSON data).

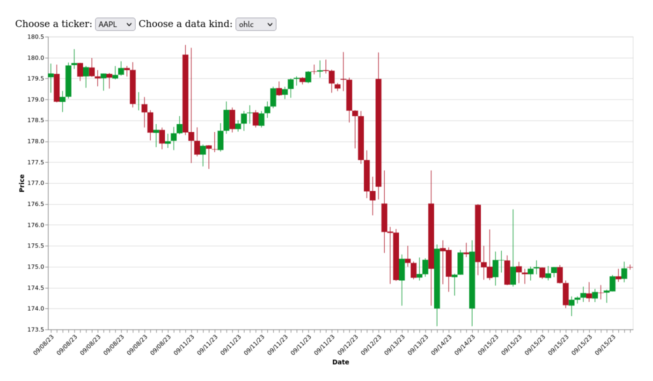
Some solutions to those performance issues are proposed within the documentation, the only relevant one in our case being switching render engine from SVG to WebGL[25], which uses the GPU to render graphics. This proved impractical, unfortunately, as the range breaks are not supported in WebGL mode[26].

An alternative solution was explored using FastAPI and the visualization library Vega-Altair[27], which makes use of the Javascript Vega-Lite library[28]. This library is radically different from Dash / plotly / matplotlib / seaborn and uses a powerful declarative syntax to describe graphs.

Compared to those obtained with Dash, the graphs produced were (subjectively) more visually-pleasing, easy to tweak, and fast to render even with a large values (that Dash could not even render). However, since Vega is dedicated to graph rendering only, the page totally lacked interactivity, e.g. switching from one stock value to another from an option in a dropdown menu. It would have required development of a full (but rather simple) web application in order to achieve the level of interactivity that is available out-of-the-box with Dash. A proof-of-concept using the Javascript frontend library Lit[29] was produced but overall it was a bit too much work within the scope of this project, especially considering that the solution with Dash was already functional. The results of the experiment have been kept on the `vega` development branch[30]

Table 3. Graph rendering using Vega


Choose a ticker: AAPL Choose a data kind: simple


Choose a ticker: AAPL Choose a data kind: ohlc

25. https://dash.plotly.com/performance#graphs
26. See documentation : https://plotly.com/python/webgl-vs-svg/ or this 3-year-old issue : https://github.com/plotly/plotly.js/issues/4630
27. https://altair-viz.github.io/
28. https://vega.github.io/vega-lite/
29. https://lit.dev/
30. See the backend : https://github.com/DataScientest-Studio/juin23_bde_opa/blob/vega/src/opa/vega.py and the JS part : https://github.com/DataScientest-Studio/juin23_bde_opa/blob/vega/src/opa/vega.js

Machine Learning

One way to consume data suggested by the project presentation is by implementing some machine learning models to predict the evolution of the stock market. This is obviously quite a difficult topic, and I have very little know-how in Machine Learning. Therefore I set the lowest priority on this task, but still had time to review it once the other parts of the project were reasonably complete.

Being totally ignorant of the subject, I stumbled upon a very thorough literature review of "Machine learning techniques and data for stock market forecasting"[31] from 2022. This article analyses more than 100 articles published on the subject over 20 years, and is a perfect starting point to get some insights on what can be achieved in this domain, and how :

- the kind of data used as an input,
- the models used
- the predictions that can be made

I only focused on those that take as an input the closing value of a given stock, and aim at predicting its future value without relying on a complex mix of ML models. Even those simpler approaches, though, require techniques and tools that were not studied within the courses ; e.g. the paper "A hybrid ARIMA and support vector machines model in stock price forecasting"[32] makes uses of the ARIMA model[33], a very popular model in this field.

Those models are not available in "classic" libraries like Scikit-learn, and one should turn to e.g. statsmodels[34] to implement them ; Spark also needs an external library[35] to deal with time-series. Choosing the correct parameters is quite a difficult task without much know-how in the field. The model evaluation in itself is quite different from non-time-series data, as well as is the simple split between training and test data (which should not be random).

I played around a bit with those concepts using the superb Orange data mining desktop application[36] but eventually realized it would be way too much work to implement even a simple prediction model in a more reusable form (e.g. a Jupyter notebook) just for the sake of it ; therefore it went out of scope of this project.

31. https://www.sciencedirect.com/science/article/pii/S0957417422001452
32. https://www.sciencedirect.com/science/article/abs/pii/S0305048304001082, freely available on Sci-Hub: https://sci-hub.live/https://doi.org/10.1016/j.omega.2004.07.024
33. https://datascientest.com/arima-series-temporelles (in French)
34. https://www.statsmodels.org/stable/index.html
35. https://github.com/twosigma/flint
36. https://orangedatamining.com/

HTTP API

Scope

The data we gather within this project will be more useful if it is easily accessible to various stakeholders : data scientists that might want to do some machine learning and model training on the data, automated code that would take some investment decisions, data reports, and so on.

This data is obviously already available directly from the MongoDB database and can be accessed in this way by anyone with the proper credentials and in any programming language that has a MongoDB driver. The very code of this project uses the `MongoDbStorage` class, an implementation of the `Storage` abstract class (seen in Data storage section), which abstracts away the kind of database used.

However, we might want to extend this access to other languages and/or keep the option to opt out from MongoDB. Or open the access e.g. to external clients.

In this context, a classic JSON API accessible via HTTP is the ideal solution.

Implementation

There is not much to say about the implementation ; the HTTP API is basically a mirror of the API defined in the `Storage` abstract class ; therefore its implementation is very straightforward, using the FastAPI framework[37]. In the most complex cases it only involves converting some parameters in another format or making them optional, as exemplified below :

Implementation of the stock values endpoint

```python
@app.get("/{ticker}")
async def get_stock_values(
    ticker: str,
    kind: StockValueKind,
    credentials: CredentialsType,
    limit: Optional[int] = None,
) -> list[StockValue]:
    check_user(credentials)

    kwargs = {}
    if limit is not None:
        kwargs |= dict(limit=limit)

    return opa_storage.get_values(ticker, kind, **kwargs)
```

Securisation

Most of the work on this module has been focused on restricting access to our API only to legitimate users. We will use user authentication for that, but in a context where all the API calls are made by well-defined users which operate from an internal network (e.g. the dashboard application, datascientists within the same company, …), the API could simply NOT be exposed on the Internet, which could very well be achieved with a proper setup of the IP tables and the web server on the API hosting machine.

37. https://fastapi.tiangolo.com/

We have opted for a simple username/password scheme using HTTP Basic Authentication[38]. It's important to note that this approach is nowhere near safe in the absence of HTTPS, since the credentials are transmitted in clear text.

On every request to any endpoint of the API, the credentials are checked against a list of valid users/passwords, which (for simplicity) is stored in a plain JSON file. The passwords are obviously never stored in clear text, and the recommandations of the Open Web Application Security Project[39] have been followed for password storage. The recommended hashing algorithm (Argon2id[40]) is resistant to both side-channel timing attacks and GPU attacks ; it has a well-maintained Python library[41].

Example of user credentials file

```
{
    "alice":
"$argon2id$v=19$m=65536,t=3,p=4$GxP3IeCBqV2sMizXXeqzwQ$2I1DOwlzvFKZAV7VYXMlhXrpss2/C79bczf8
0UoT+q0",
    "bob":
"$argon2id$v=19$m=65536,t=3,p=4$StFQXOIkcEyHzo92VoddyA$BxzQZmRTDdwCT9TjLBZh6NU3hkSGASHMFuQo
uVnWguw",
    "charlie":
"$argon2id$v=19$m=65536,t=3,p=4$ekWgEzumCgKfjmuicEJ+GQ$zbWaOxn0U4XiWsXKRbm46ohIKEpLoYK/Uld/
P8Lhe9I",
    "julien":
"$argon2id$v=19$m=65536,t=3,p=4$+6b6zy6RZ3GMClCiH723zg$KOOPOhz3+a9sPP8a1m5sQh5QbK9zk/xU+5wm
nWipsSc"
}
```

38. https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication#basic_authentication_scheme
39. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
40. https://www.password-hashing.net/#argon2
41. https://pypi.org/project/argon2-cffi/

Project development and deployment

The project has been developed following principles that make for easy iterative development and deployment.

It is split into 3 main sub-parts that run independently from each other, following a philosophy akin to micro-services :

- the financial data reader (which reads data from the external API and saves it into the database)
- the HTTP API (which reads data from the database and makes it available to other components)
- the dashboard (which reads from the API and presents it via a graphical UI)

Software development

The code is versioned-controlled using `git`, with a central repository available on GitHub[42]. As of the writing of the report section, the `main` branch has over 200 commits with meaningful messages. GitHub issues[43] have been used.

Commit log of the project (as of 18/09/2023 - showing only the latest 10 entries)

```
f486fbb Write report section on API
fd01200 Add report section on data dashboard
23f52ea Detail project scope in a new file
60f996d Fix typo in schema name in report
d1433c4 Use argon2 for password hashing instead of scrypt
7bc8485 Create some documentation on the demo run
c668c8e [chore] Bump to 0.3.0
43d15ad Add bump_version command
59b0511 Extend project compatibility to Python >= 3.9 instead of 3.11
515d93d Rename pdm.Dockerfile simply to "Dockerfile"
```

The software itself is written using sensible practices : clear and consistent naming, short and readable functions, type annotations, consistent formatting (using `black`),... Following the principle that the best documentation is no documentation (since documentation and code quickly get out-of-sync), code was written with readability as a primary concern. Documentation was only added in places which do not read well enough or feature non-obvious design choices, or to serve as documentation of external features that are not very well-documented.

42. https://github.com/DataScientest-Studio/juin23_bde_opa
43. https://github.com/DataScientest-Studio/juin23_bde_opa/issues

Documenting hard-coded values

```
    if hour_break:
        # The behaviour of this setting is really not well documented
        # (see https://plotly.com/python/time-series/#hiding-nonbusiness-hours
        # or https://plotly.com/python/reference/layout/xaxis/#layout-xaxis-rangebreaks)
        #
        # The natural setting to use seems to be [16, 9.5], but.. Using that setting
        # hides the value produced at 16:00 every day (while the value produced at
        # 9:30 is properly displayed) ; it seems like the first bound is exclusive, and
        # the second is inclusive.
        #
        # Something like `[16.5, 9.5]`` produces a visible gap between the last value of
        # a day, and the first value of the next day. `[16.01, 9.5]`` displays both values
        # but they are almost superposed.
        #
        # Even though completely unlogical, `[16.01, 9.25]` kinda does the trick, and
        # displays both values without a noticeable gap in-between them.
        breaks += [dict(bounds=[16.01, 9.25], pattern="hour")]
```

GitHub actions[44] have been leveraged to provide *Continuous Integration*, e.g. by ensuring that the code is properly formatted or that unit tests keep on passing.

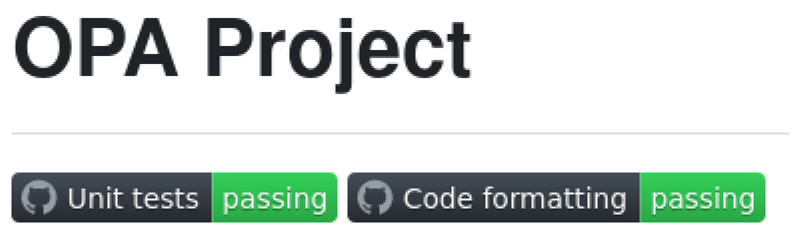


Figure 9. GitHub badges

Packaging the project

Packaging is the process by which a piece of software is prepared for distribution and installation in its runtime environment (which is usually distinct from the environment in which it is developed). In Python, it is a well-known headache[45], with no established ubiquitous tool nor clear, universal instructions.

Of the numerous options available : pip + venv + setuptools, Hatch[46], Poetry[47], pipenv[48], PDM[49],.. I have elected to go for the latter, which works basically like `npm` does in the Node-JS ecosystem. Simple commands

44. https://github.com/DataScientest-Studio/juin23_bde_opa/actions
45. https://chriswarrick.com/blog/2023/01/15/how-to-improve-python-packaging/#does-python-really-need-virtual-environments
46. https://hatch.pypa.io/
47. https://python-poetry.org/
48. https://pipenv.pypa.io/
49. https://pdm.fming.dev/

allow to init a skeleton project, add production or development dependencies, build the project for distribution, install it, and so on.

Runtime configuration (e.g. HTTP ports used, database configuration, ...) is handled by Dynaconf[50], which can read configuration either from a file or from environment variables. The secrets required to run the project (API keys, database credentials, ...) are merged into the configuration at startup.

With a properly packaged project, any machine that can run Python can now install and run it. The helper script `run_local.sh` allows to start any of the 3 parts of the project on the local machine, a feature which has proven extremely useful in the development phase (where running a full Docker image is much heavier and slow). An interactive shell (that uses IPython) was even added to allow "playing" with the internal APIs in the same environment as the running application.

Orchestrating the project

Though it would be possible to manually launch the 3 different parts of the project in different processes of the same machine, it is not necessarily super practical.

For this reason, both Docker and Docker Compose have been leveraged to allow running the whole project in a consistent way.

The `Dockerfile` has been written with inspiration from the Dockerfile suggested on PDM website[51], with some modifications to ensure dependencies are not installed on every source code modification.

The `docker-compose.yml` file is quite straightforward, making use of Docker Compose features such as volumes[52] or secrets[53]. It defines all the services that make up the application :

- external images : `database` which is simply an official `mongo` image[54], properly setup
- internal images : `financial_data_reader`, `data_report`, `internal_api`
- test images : to allow running unit, integration, or functional tests

50. https://www.dynaconf.com/
51. https://pdm.fming.dev/latest/usage/advanced/#use-pdm-in-a-multi-stage-dockerfile
52. https://docs.docker.com/compose/compose-file/07-volumes/
53. https://docs.docker.com/compose/compose-file/09-secrets/
54. https://hub.docker.com/_/mongo