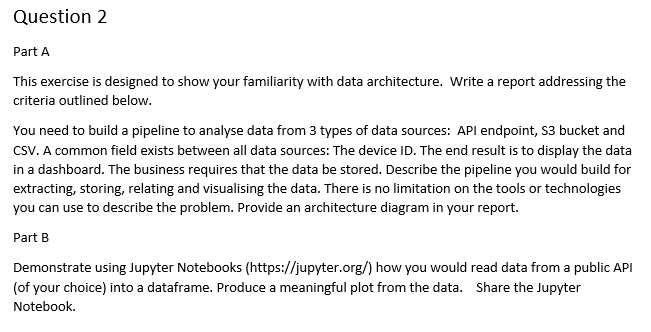
Zachary Taylor

*+358452461454*

*Zachary.mark.taylor@gmail.com*

Lightneer Data Engineer Test:

Question 2 Answer



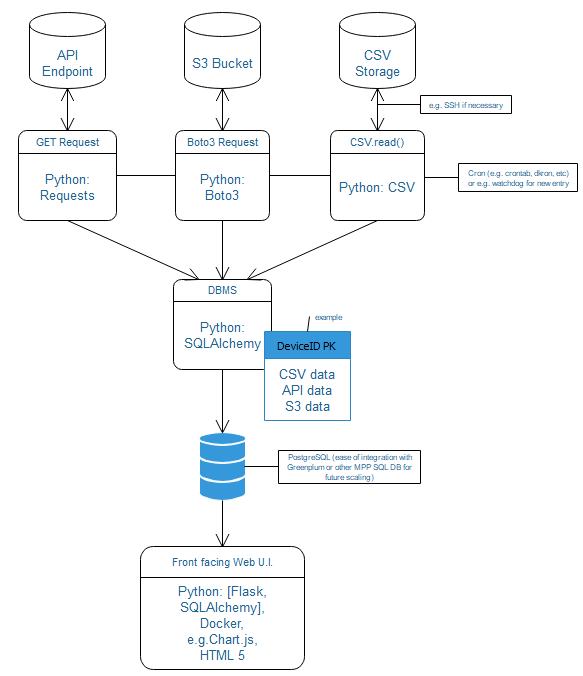
# Part A

This report will be a high-level description of the pipeline designed to gather and visualize data from the three types of storage: API, S3 bucket, and CSV file. Of course, there are a wide variety of options for each and every solution that is presented here so the following will be based on past experience resulting in success. All solutions are subject to adjustment or replacement based on the individual needs of each endpoint and the needs of the client for whom the visualization will be designed.

This solution is also designed with scalability in mind. Each of the endpoint requests will be repeatable with minimal effort and additional storage system queries can be added as the need arises with the integration method allowing for easy additions or modifications.

Lastly, it has also been designed to be cost effective in the long term as it contains no third-party software requiring licensing, although will require a longer deployment stage than a third-party software would require.

The report has been broken down into 3 stages requested by this task: Extraction, Storage/Relation, Visualization, where Storage and Relating the data has been combined into one section due to this system’s data flow.



## Extraction

Each of the three storage systems will require a separate interface in which to request and transfer the data, and each of these systems has a package or combination of packages available in Python. Each will have its own dedicated script for requesting the data and will be managed by a Cron system to provide automated up-to-date data with individual timing based on each storages system. In the beginning, this would most likely be a simple app such as Crontab to minimize deployment time. However, with scaling in mind, third party apps could be added without licensing fees such as Dkron which is a decentralized cron system that creates a network cluster and provides fault tolerant job handling.

In the case of CSVs, a variety of options other than Cron type systems are available depending on storage location, for example Watchdogs is a Python package that allows for event triggering on directory change thus creating “real-time” updates for that data set. If said CSV is not stored locally then it will default to the previously mentioned cron systems. Additional features can be easily added such as SSH functionality to create server crawlers, again though all dependent on the location of the data and needs of the client.

In the case of the S3 Buckets and/or CSV (location dependent) the scripts will also be equipped with authentication functionality to create/access secure connections, each functionality being storage authentication requirement dependent.

### API

For API endpoints the solution will be the Python package Requests, this provides the functionality for Python to create any CRUD API request. In this case, it will submit GET requests on a set schedule. It will then create a dictionary of the data receive storing it as key: value pairs (if the data is not already in said configuration), where the key will be the Device ID and the values will be the data relating to that ID.

This dictionary will then be passed on to the DBMS, which will be discussed further in the Storage section.

Of course, Pandas DataReader would be used to directly import them into DataFrames if that is possible, however it has not been implemented in this situation due to too many unknown with the target data source so generalized solutions must be used to ensure success.

### S3 Bucket

This storage system utilizes AWS APIs, which will allow for easy access and manipulation of the main functions of said Bucket and will be structured similarly to the process descripted in the API section. Meaning the script will send a GET request and store the response as a dictionary with the same key: value pairing as the API section (if not already stored in that configuration) and pass the data to the DBMS.

The only major difference will be in the authentication. The solution to this is Boto3, which is the name for the Python SDK that provides functionality for AWS allowing automated access, file/bucket creation, downloading, copying, etc. of any bucket in which it possesses the credentials.

### CSV

This is the most dependent on location out of the three systems. If stored locally, it is a simple matter of having Python read the CSV and convert it into a dictionary with the same key: value configuration as those in the API and S3 sections. This of course is more or less complex based on the structure of the CSV file, as well as the volume of said files needing to handle.

If it is not stored locally then additional functionality, e.g. SSH tunneling, will be added to access said files.

### Other

This section has been added to highlight that this system of gathering and converting to Python dictionaries can be adapted to a wide variety of storage systems. As long as the storage system is remotely accessible there is no system that could not be added to this structure.

## Storage/ Relation

The storage of this data will utilize the Python package SQLAlchemy as an ORM to allow for the creation/management/etc. of a PostgreSQL Database. The structure of the database itself, of course, is entirely dependent on the data contained within the storage sources and thus not knowing this it cannot be mapped here. PostgreSQL was chosen for both its familiarity and history of success as well as its similarity to familiar MPP SQL Databases (Azure, Greenplum, etc.) allowing for limitless scalability.

The main processing script will receive all the previously configured dictionaries and combined them into individual entries via SQLAlchemy with the Device ID being the primary key of each entry. This can be endlessly modified with additional tables and/or columns ad infinitum. This provides both the storage and relation of the data required by the client, as well as provides an easy data source to be queried by the WebUI for visualization. It also allows for the continual updating or insertion of new data per each Device ID.

## Visualization

Since the desired output is that of an interactive dashboard, the solution that has been the most successful and efficient is that of the Python SDK Flask. This provides a large array of functionality and will be the foundation for the WebUI that will make up the dashboard.

With it, the front end will be coded in HTML 5 and integrate chart and table drawing functionality with add-ons like Chart.js, Seaborn integration, or just about any other so desired. There is no real limitation here and there has been a large amount of personal success in the past integrating these systems and displaying large amounts of data in a variety of graphical representations.

The data will be collected and visualized by querying the previously mentioned database via SQLAlchemy and manipulating the data into the configuration desired. This will also allow for endless updating of new charts or other visualizations and the updating/removal of the old/outdated

Finally, this Flask application with be Dockerized to create persistence within the server it is being hosted on, as well as ease of restart in the case of server failure. This will also allow for the system to be installed on updated or newer systems with minimal effort.

# Part B

As the procedure for sharing the Jupyter Notebook was not specified I have included several options so that one might be picked that works best for the reader.

Github link:

Kyso.io link:

<https://kyso.io/Gurubagus/data-engineer-test-question-2-part-b-zachary-taylor#code=shown>