A new platform to manage our sales.

The principal idea in this PoC is to define two different API thinking in the type of interactions that we can have with our database. The first API (product api) will be used to add information to the database. The teams that process our sales will be responsible for adding this information to the database. The second API (finance api) will recover the data in a human-readable format. This API will be used for the finance department to view our results.

Both APIs have access to different layers in our database. The product API has access to the bronze layer, and the finance API has access to the gold layer.

The necessary elements in this project are a database and an ETL orchestrator.

We use PostgresSQL to define our database. This PostgresSQL is executed in its own container to reduce the possible interactions with the rest of the components.

As an orchestrator, we decided to use Apache Airflow, which will execute in a completely different container. This approach can help us with future updates in which we can decide to use a different database.

Input data

Before to continue explaining and developing our PoC, we must understand which data set we will expect to use. In the documentation shared by BuyBay, we have five input data sets. Four are in CSV files, and the last arise in a table defined in the documentation.

The sets are:

- sold_products.csv
- graded_products.csv
- grading_fees.csv
- transport_cost.csv
- Platform fee

sold_products.csv

In the next table, we represent the structure of the csv files.

Feature	Explanation
license_plate	Unique reference for each product
status	Status of the order, can be shipped or cancelled
platform	Name of the e-commerce platform where the product is sold (bol, ebay, …)
created_at	Timestamp of creation of the order
shipped_at	Timestamp of shipping of the order
sold_price	Selling price of the product
country	Country where the product is shipped to
channel_ref	Platform reference of the order
platform_fee	Platform fee for the sold product. It cloud have no value

The primary key is the "license*plate", we will need to use this field to can have access to each product sold. The fields shipped*at and platform_fee have null values. Would be necessary to evaluate if we need to fix with some value or not.

The shipped*at field is null when the status product is "cancelled". This is a logical and admisible data. The field platform*fee is shared directly by the platform at the beginning of the process. Sometimes, this data would be null, and we must use the table "Platform fee" to fix it. This process will do when we move the data from the bronze tables to the next level.

graded_products.csv

In the next table, we represent the structure of the csv files.

Feature	Explanation
License_plate	Unique reference for each product
grading_cat	Grading category, used to determine the fee
grading_time	Time spend on grading in minutes

This CSV has the final information relative to the grading process. As the sold*products file has the rest of the data relative to the product. We can expect, and we have, a perfect correlation between the license*plate field in both tables. We have the grading data for the whole of the products.

Note

Inside the grading file, the field "Licence_plate" has the first letter in the capital. To reduce future problems, during the ETL process will modify this name to homogeneity with the rest of the fields.

grading_fees.csv

In the next table, we represent the structure of the csv files.

Feature	Explanation
grading_cat	Grading category, used to determine the fee
cost	Fee that is charged for grading

This is a simple table with the structure of a dimension table that we will use in a Star Schema.

We can expect that we will have the whole possible grading categories inside this CSV. And this complete be a perfect match with the data contained in the graded_products file.

More or less, this is the situation that we have here, but this is not precisely the case. In the file graded*products, we have three categories without correlation in the grading*fees file. These are mn1, mn2, mn3. We can consider that these are different categories, but the most logical is to think this is a mistake when typing the categories because we have the categories MN1, MN2, and MN3.

In addition, in the file grading*fees, we have three additional categories not present in the file graded*products. This doesn't generate problems.

transport_cost.csv

In the next table, we represent the structure of the csv files.

Feature	Explanation
country	Country of shipping
transport_cost	Cost that is charged for shipping

This file represents a dimension table, and we would not have many problems merging the data.

The unique problem is that we don't have a value for each possible country. We have a value

for some countries; in other cases, we have a fixed cost. This adds a condition to the ETLs process.

Platform fee

We have a piece of additional information that is not contained in any CSV file. It is directly defined in the request and is in the following table.

Platform	Percentage of teh sold price
Bol	10 %
Amazon	10.5 %
Ebay	9 %
Everything else	11 %

Only in the case in which the platform didn't share a fee with us, we need to use this table to calculate the fee in terms of the sold_price field.

Data Warehouse

As we explained before, our data warehouse will be implemented in a PostgreSQL database.

Our data warehouse will contain the three expected layers, a bronze layer where we will save the data directly without processing. A silver layer where we will define a star schema with the products as a fact table and the fees as dimension tables. In the last layer, the gold, we merge all the data to have in the table the data we need in the reports.

The unique additional data we will inject into the tables are the "created*at" and "updated*at" times to simplify the control about the last data instance.

BRONZE LAYER

We will move the data directly from the CSV, in this case, the tables will be a copy of the CSV files.

Table sold*products*bronze

This is the bronze table in which we will load the whole of updates of each sold product. In this table, we added two additional columns compared with the original CSV; both files will be used to differentiate each product update.

Feature	Data type
id	SERIAL PRIMARY KEY
license_plate	VARCHAR NOT NULL
status	VARCHAR NOT NULL
platform	VARCHAR NOT NULL
created_at	timestamp NOT NULL
shipped_at	timestamp
updated_at	timestamp NOT NULL
sold_price	real NOT NULL
country	VARCHAR NOT NULL
channel_ref	VARCHAR NOT NULL
platform_fee	real

Table graded*products*bronze

This bronze table, as before, is a copy of the "graded*products.csv" file, and in addition, it contains the whole of the updates. And as before, we added two additional columns, "ID" and "updated*at". That would be used with the same idea as before.

Feature	Data type
id	SERIAL PRIMARY KEY
license_plate	VARCHAR NOT NULL
grading_cat	VARCHAR NOT NULL
grading_time	smallint NOT NULL
created_at	timestamp NOT NULL
updated_at	timestamp NOT NULL

grading*fees*bronze

Feature	Data Type
id	SERIAL PRIMARY KEY
grading_cat	VARCHAR NOT NULL
cost	real NOT NULL
created_at	timestamp NOT NULL
updated_at	timestamp NOT NULL

transport*cost*bronze

Feature	Data Type
id	SERIAL PRIMARY KEY
country	VARCHAR NOT NULL
transport_cost	real NOT NULL
created_at	timestamp NOT NULL
updated_at	timestamp NOT NULL

platform*cost*bronze

Feature	Data Type
id	SERIAL PRIMARY KEY
platform	VARCHAR NOT NULL
cost	real NOT NULL
created_at	timestamp NOT NULL
updated_at	timestamp NOT NULL

SILVER LAYER

In this layer, we build the star schema, the fees will be the dimension tables, and we make the fact table with the products sold.

The fee tables is only the last snapshot copy of the data contained in the bronze layer. In the

fact table we merged, by license*plate, the tables sold*products*bronze and graded*products_bronze.

grading*fees*silver

And the last snapshot of the previous data.

Feature	Data Type
id	SERIAL PRIMARY KEY
grading_cat	VARCHAR NOT NULL
cost	real NOT NULL
created_at	timestamp NOT NULL
updated_at	timestamp NOT NULL

transport*cost*silver

And the last snapshot of the previous data.

Feature	Data Type
id	SERIAL PRIMARY KEY
country	VARCHAR NOT NULL
transport_cost	real NOT NULL
created_at	timestamp NOT NULL
updated_at	timestamp NOT NULL

platform*cost*silver

And the last snapshot of the previous data.

Feature	Data Type
id	SERIAL PRIMARY KEY
platform	VARCHAR NOT NULL
cost	real NOT NULL
created_at	timestamp NOT NULL
updated_at	timestamp NOT NULL

Table products_silver

This is the table in which we will load the last snapshot of each product. In addition and from graded*products*bronze we load the 'grading*cat' and 'grading*time' fields. We merge the data using the "license_plate" field.

Feature	Data type
id	SERIAL PRIMARY KEY
license_plate	VARCHAR NOT NULL
status	VARCHAR NOT NULL
platform	VARCHAR NOT NULL
created_at	timestamp NOT NULL
shipped_at	timestamp
updated_at	timestamp NOT NULL
sold_price	real NOT NULL
country	VARCHAR
channel_ref	VARCHAR NOT NULL
platform_fee	real NOT NULL
grading_cat	VARCHAR NOT NULL
grading_time	smallint NOT NULL

GOLD LAYER

Table products_gold

The final table, our finance app will recover the data from this layer, and in particular this table.

Feature	Data type
id	SERIAL PRIMARY KEY
license_plate	VARCHAR NOT NULL
status	VARCHAR NOT NULL
platform	VARCHAR NOT NULL
created_at	timestamp NOT NULL
shipped_at	timestamp
updated_at	timestamp NOT NULL
sold_price	real NOT NULL
country	VARCHAR
transport_cost	real NOT NULL
channel_ref	VARCHAR NOT NULL
platform_fee	real NOT NULL
grading_cat	VARCHAR NOT NULL
grading_fee	real NOT NULL
grading_time	smallint NOT NULL

In addition

In the folder "Notebooks/Loads", I defined four notebooks to help us load the initial data and evaluate whether the system is working correctly or not.

ETL Process

To orchestrate our ETL process, we used an additional docker container in which we executed an Apache Airflow. This docker is the same as that we download from the docker hub; the unique difference is that we stop the examples.

Our process is defined with only two dags, fees and products.

The main idea of the ETL is to use the "created*at" and "updated*at" fields to know when the data was updated in the database and use this information to locate the newest data and move them to the silver area.

In the golden area, we have only the data combined to simplify access to them.

Fees DAG

In this dag, we move the fees contained in the tables grading*fees, platform*cost and transport*cost. The process recovers the last version of the fees, using the updated*at field, and moves from the bronze area to the silver area. Of course, it deletes the old data from the silver area.

This DAG is schedule to execute only one time per day, I don't expect that would be necessary to execute more frequently. But it is necessary to execute before to the Producers DAG, because it needs to load the fees.

Products DAG

This dag moves the product data from the bronze area to the gold area, and it will add the fees. From bronze to silver, we added the graded_products data.

This DAG is schedule to execute each ten minutes.

In the case that somebody updated multiples times some product, between consecutive executions of the DAG, we will sure that we will move the last data into the gold area, because the DAG load the data in the order that is loaded into the database.

Additional config

To be sure that we will have access to the database, we need to modify the IP parameter into the dags/credential/**init**py file. The value that we need to add is the ip of the system in which we will execute the docker that contain the PostgreSQL database.

Product-API

This is one of the two APIs defined in the system. It will access the bronze area, and it will use to load data into the database. Of course, nobody can access the database directly, only across this API.

This API has five endpoints, each one to load each type of data. In all cases, the endpoints are expected to receive a data frame linearised.

The urls are:

- /load_products
- /load_graded
- /load*grading*fees
- /load*transport*cost
- /load*platform*cost

In the folder "Notebooks/Loads" we have five notebooks that would be an example about how the API will work

load_products

The notebook "05 Load Products" load the initial 'sold_products.csv'.

load_graded

The notebook "04 Load graded*products" load the initial 'graded*products.csv'.

load*grading*fees

The notebook "03 Load transport*cost" load the initial 'transport*cost.csv'.

load*transport*cost

The notebook "02 Load grading*fees" load the initial 'grading*fees.csv'.

load*platform*cost

The notebook "01 Load platform_cost" load a fake platform cost, only to demonstrate how it will work

Security

In this API we implemented an OAUTH 2 security, using a "fake" database with the data.

username	full_name	email	password	disabled
tgarsa	Tomas Garcia Saiz	tgarsa@me.com	secret	False
alice	Alice Wonderson	alice@example.com	secret	True

In the Notebooks, it is posible to evaluate how the securioty is working.

Additional config.

As before, to can have access to the database, it is necessary to config the IP of the system. In this app, this is in the line 20 of the file app.py.

Finance API

This is the second, and final API defined. In this case, we have access over the gold layer, and we can download in a readable format the expected data.

We have two endpoints:

- /partner_data
- /finance_report

Both endpoints read the unique table in the gold layer, products_gold, that contain the necessary data.

In the folder "Notebooks/Finance", we have two notebooks to access at each end_point, to have a simple case of use of both end-points.

Security

We duplicate the security used in the previous API.

Additional config.

As before, to can have access to the database, it is necessary to config the IP of the system. In this app, this is in the line 20 of the file app.py.