Contoso ingestion PoC by Claudia Benítez Cuadra

The following is the solution to the problem faced by the company Contoso:

I have taken as a base flow an Azure Data Factory pipeline that calls a Databricks Notebook. Its flow is as follows:

The pipeline parameters would be:

RESTART: parameter that depending on it the flow starts from one state or another:

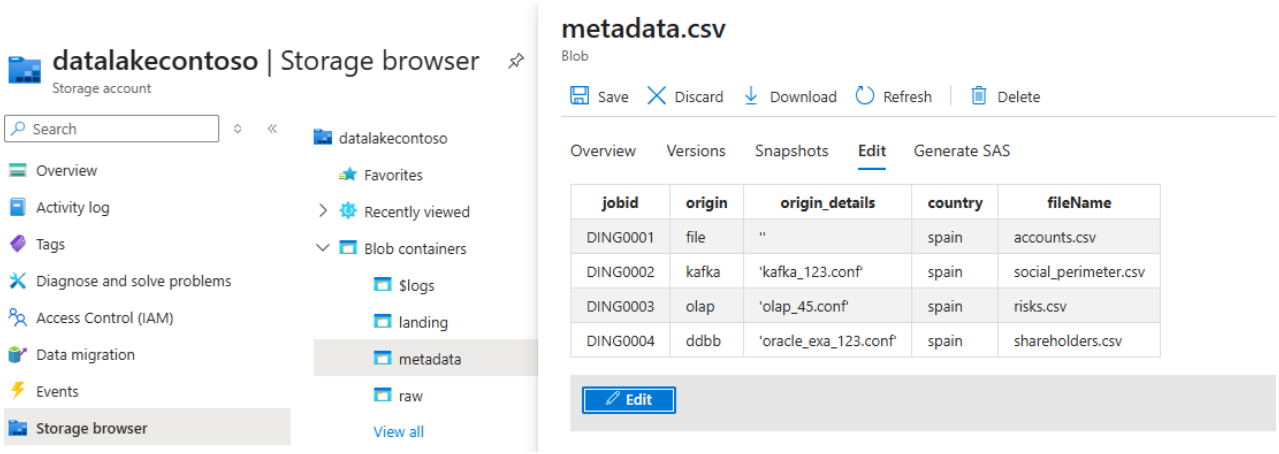
- 0: raw data copy, transformation and upload to Delta Lake.
- 1: the data is already in raw and you want to reprocess it from there, only transform and upload to Delta Lake.
  * It is possible to add more states, it is an example of file reprocessing in the middle of the cycle.

JOBID: the ID of the ingest we want to launch. For example, DING0001

ODATE: Date from which we want to ingest the files (it is partitioned by odate and timestamp of ingestion). For example, 20250622

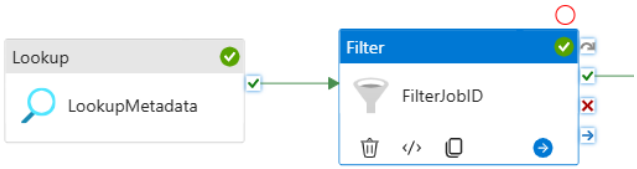
The flow of the pipeline is:

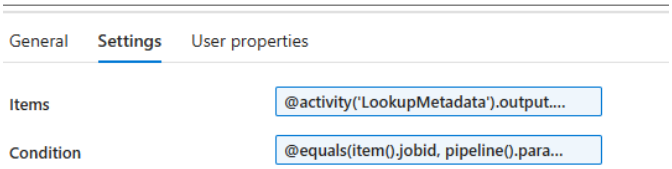
1. The metadata of the ADLS is read (in this case I have created a container called “metadata” with a .csv file with the following data):
   a. Jobid idthe identifier of the ingest with which the pipeline is launched (the third parameter).
   b. Origin is the data origin, in this case I have valued 4 types of origins but it is scalable to more.
   c. Origin_details is a parameter that points to the configuration of the origin, for .csv files there is no configuration as such since they are in landing, but for data extractions from other origins for example from databases we need the data to be able to connect and to extract from there (and may be common to several intakes).
   d. Country is one more parameter to organize data, I have added it for organization.
   e. FileName, is the name of the file in origin (landing) in case of existing.



*A .csv has been used as it is a demo but it can evolve to a more robust database, even dividing this metadata by origins, project, country... and you can also add more parameters such as housekeeping, extraction query in origin if used, data structure file if used...

2. After reading the metadata, the jobid must be filtered in order to extract its associated parameters:

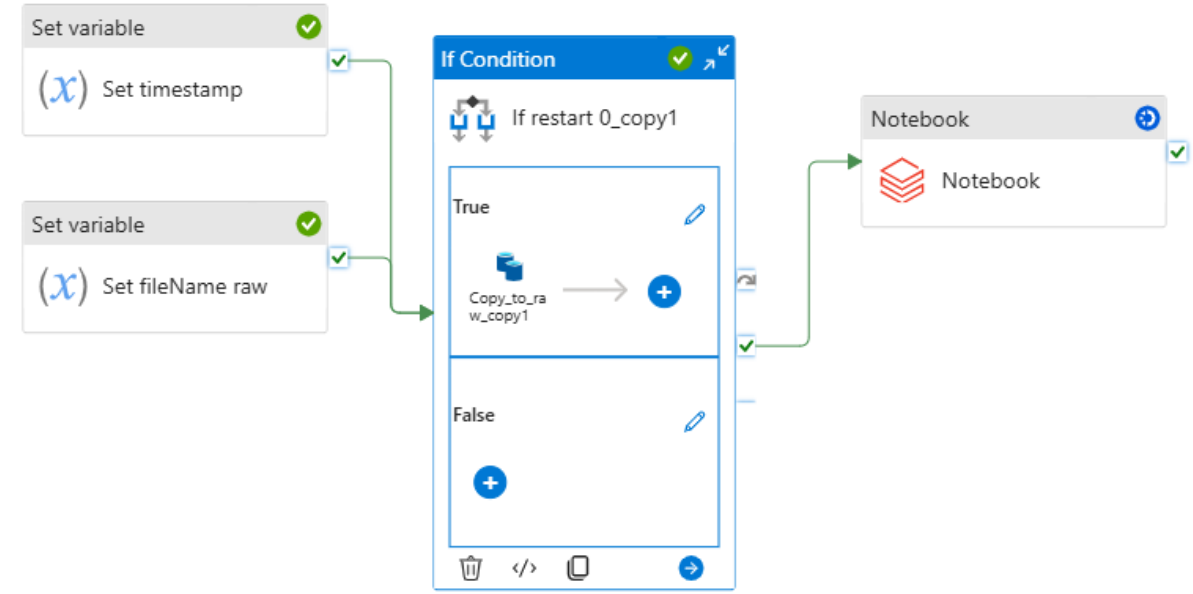


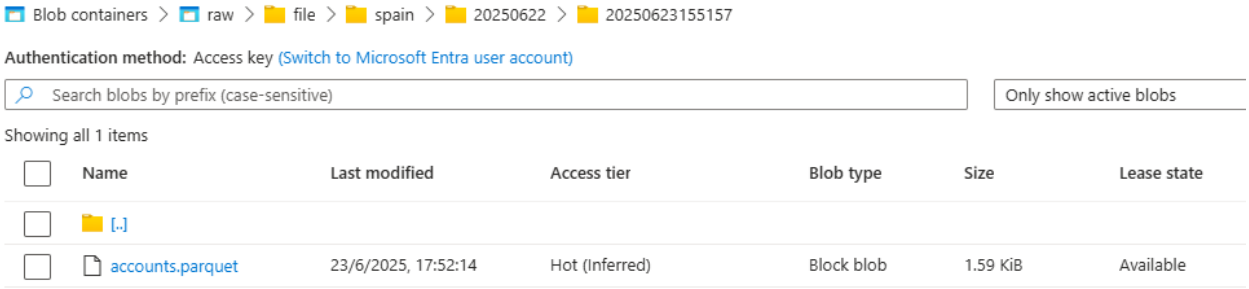


3. When the filter detects the parameters it sets the partition timestamp variable and the fileName raw variable which is basically the file passing it to .parquet. Example:
    a. Landing: /file/spain/20250622/accounts.csv
    b. Raw: /file/spain/20250622/20250623123445/accounts.parquet

4. Then a forEach is called, which, in case the restart variable is 0 that means that it is necessary to do the extraction from origin (in case of an origin like bbdd, Kafka or Olap cube) or simply in case the file is already in landing as the following example copy to the raw container with its partition

and new extension. That is, if the RESTART variable is 0, there are NO files in the raw layer.





5. In case restart is not 0, we don't need that step of copying to raw because it is already there, so the next step is to launch the Databricks notebook, which will take care of getting the necessary pipeline parameters, transform the data (I have applied basic transformations such as duplicate removal and column anonymization, it all depends on the requirements) and upload it to Delta Lake.

This solution meets the requirements because:

- It is an idea embodied in both **code** and an actual **ADF pipeline**.
- A notebook from Databricks orchestrated by an ADF pipeline has been used because ADF offers ease **of integration with other services** and a **graphical interface** ideal for process planning, while Databricks allows to apply **complex logic** and **transformations** with PySpark in an efficient way.
- It is a **Metadata-Driven** solution because the source connections are in an input file that is composed of all the information needed for ingestion, allowing scalability and convenience

when planning jobs (few parameters needed). And if we need to modify a job as such, you only need to configure this Source system.

- The example solution comes with a source of a .csv file but different **sources** are proposed (databases, kafka, OLAP Cube).
- For the extraction of **OLAP cubes**, two alternatives are proposed: the first is to use a **function** within ADF to extract the data from the cube and load it directly into the RAW zone (there is no own connector in ADF so we would have to create an Azure Function to make the MDX query to the cube and convert it into a .csv file in landing for example). This option allows the actual pipeline design, ensures traceability and facilitates reprocessing. And the second option is to perform the **extraction directly from Databricks**, using libraries such as **pyadomd** or **REST** services. This is more flexible and dynamic but does not follow the pipeline flow, which can be a drawback if you need to audit or redo processes. That's why I recommend the option from ADF, as it fits better to the proposed architecture and facilitates the future scalability of the framework.
- A **raw zone** (container in Azure Data Lake Storage) and a **Delta lake** (Data Hub Zone) are used and the data is stored in delta format in the Delta Lake.

Why did I choose this option?

I have chosen this solution because it allows to abstract the extraction and transformation logic based on configurable metadata, which makes it easily scalable to hundreds of different sources without the need to modify the code base. In addition, it clearly **separates** the storage layers (Raw and Data Hub), implements good orchestration practices with ADF **and processing with Databricks**, and leaves room for **future extensions**, such as the inclusion of complex sources like OLAP cubes or streaming from Kafka.

I chose to store all data in the Raw Zone as Parquet to standardize the ingestion process across various sources (databases, files, Kafka, OLAP). Parquet offers **efficient storage** and **fast read** performance, and using a single format simplifies **automation**, improves **reusability**, and ensures **compatibility** with Delta Lake. Although the format changes, the raw content remains untouched, preserving data traceability and auditability.

Orchestration with ADF facilitates the planning and execution of reusable pipelines, while Databricks handles the complex logic. The **RESTART** parameter is very useful for controlling the starting point of the ingest. It simulates a restart or reprocessing behavior, which can be convenient, even more states can be added in the future. I also think that **partitioning** by odate and timestamp is a plus for scalability and efficient querying later.

Future Roadmap

Currently the source system is a .csv, but it is intended to migrate to a table in SQL Server or Cosmos DB containing all metadata centralized by project, country and type of origin. In addition, management APIs could be incorporated for business teams to register new sources without technical intervention.

The pipeline is created for the origin “file”, but within it an” if block” can be added that, depending on the source, executes one copy or another. It is also possible to add in the metadata options for **housekeeping** in the routes, types of **transformations** to apply, queries to extract from origin…etc.

Ideally, each file should have a copy file associated with it in which its structure is set as a data type for future conversions.

Code:

I attach two Databricks notebooks: the original notebook (**Contoso_pipeline_notebook)** that should be launched from the pipeline (with its parameters and connect ADLS paths) and a demo (**Contoso_demo)** without connection to ADF easily executable locally that simulates what the Databricks part would do: take the raw .parquet file that I created in the first part of the pipeline (i declared it in the code instead of reading it from Azure Data Lake as in the pipeline so that it can be run without having a connection to Azure), apply some transformations and upload it to Delta Lake.