Data Lake

Overview

This project involves the ingestion, transformation, and storage of data in Azure Data Lake Storage (ADLS) Gen2, leveraging Azure Data Factory (ADF), Databricks, and various Azure services for security and performance optimization. The data comes from two sources:

1. **HTTP API** (from https://jsonplaceholder.typicode.com/users in JSON format)
2. **Microsoft SQL Server** (via self-hosted integration runtime)

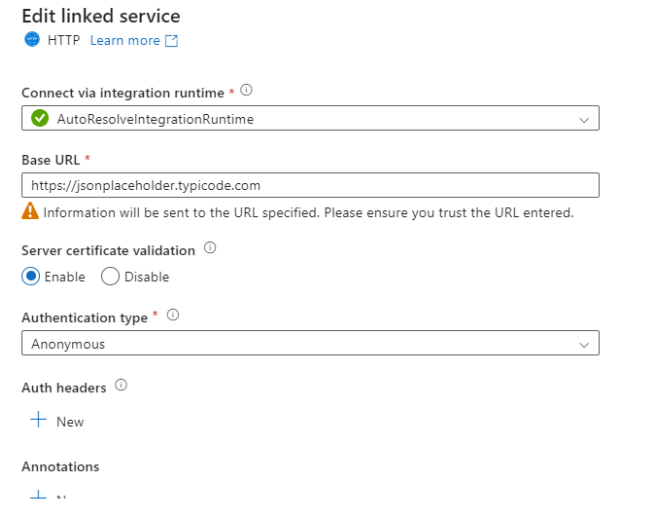
The data is processed using Azure Databricks for cleaning, followed by saving the results in multiple containers in the same Azure Data Lake Storage Gen2 account. Additionally, the project utilizes Azure Key Vault for secure management of credentials and secrets.

Steps Taken

1. Setting Up Azure Data Factory (ADF)

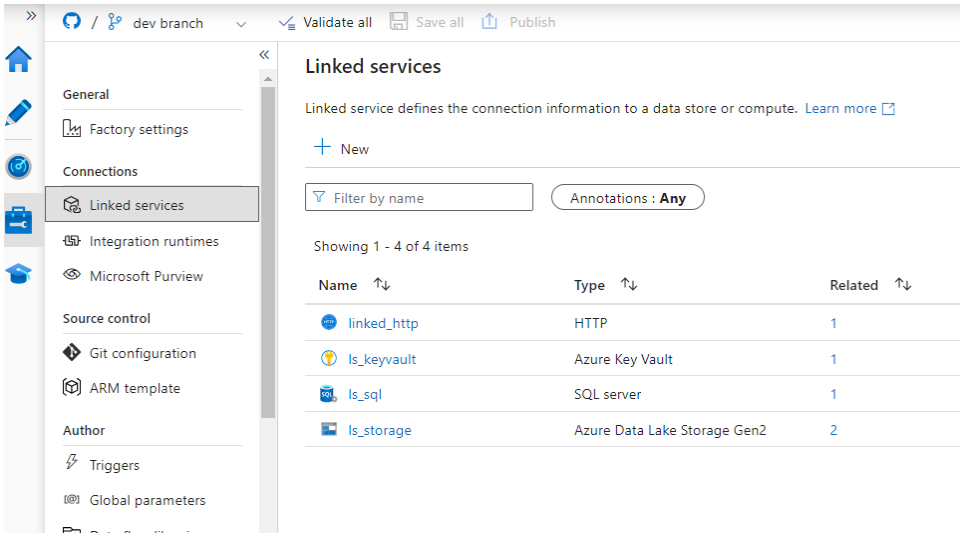
1.1. Linking ADF to GitHub

- I linked Azure Data Factory (ADF) to GitHub by creating a repository named DataLake for version control.
- I made sure the repository was connected to the **Dev** branch.

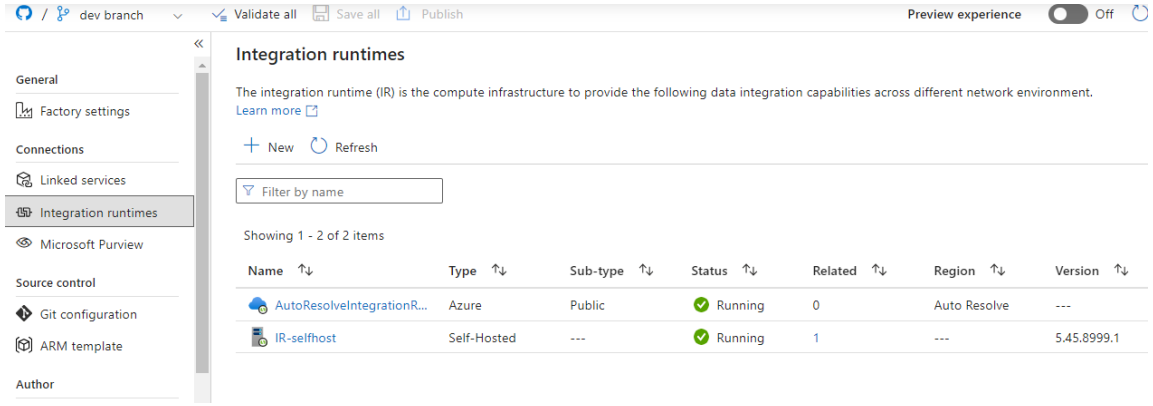


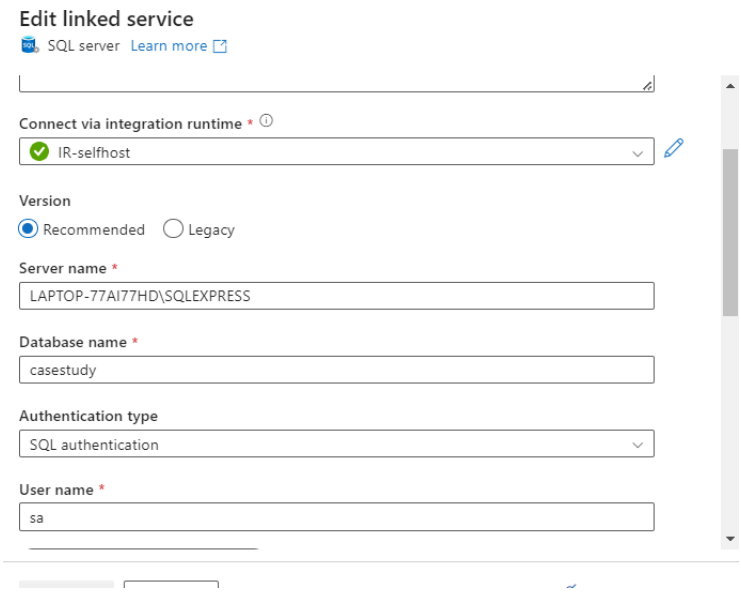
1.2. Creating Linked Services

- **HTTP API to Storage**:
    - I created a linked service to the HTTP API https://jsonplaceholder.typicode.com/users to fetch data in JSON format.
    - The data was stored in the **raw-api** container in Azure Data Lake Storage (ADLS) Gen2 in **Parquet** format.
    - I used Azure Key Vault to store the **storage account key** as a secret for security purposes.



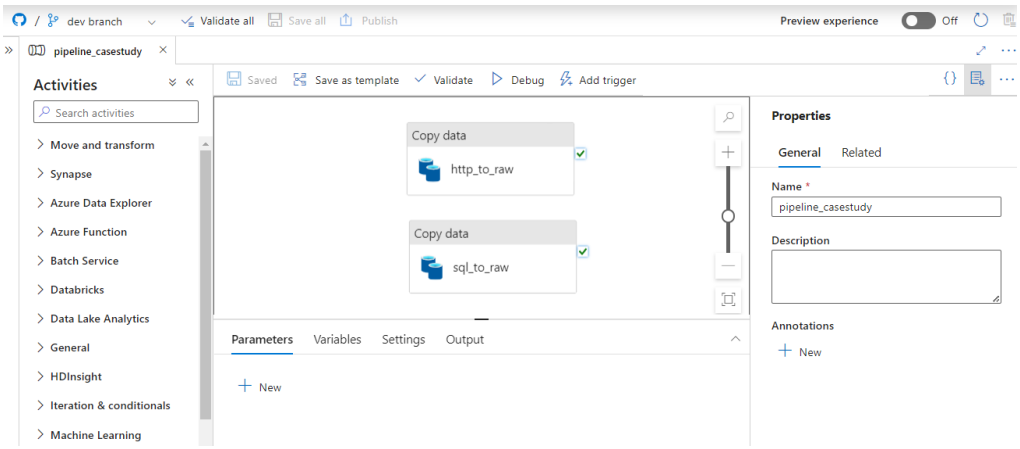
- **Microsoft SQL Server to Storage**:
    - I created a linked service for Microsoft SQL Server using a **self-hosted integration runtime**. This enabled me to connect to the on-premise SQL Server.
    - The data from SQL Server was transferred to the **raw-sql** container in ADLS Gen2 in **CSV** format.





1.3. Creating Pipelines in ADF

- **Pipeline 1** (HTTP API to ADLS Gen2):
    - I used **Copy Data** activity to fetch data from the HTTP API and save it in the **raw-api** container in **Parquet** format.
- **Pipeline 2** (SQL Server to ADLS Gen2):
    - I used the **Copy Data** activity to copy data from Microsoft SQL Server to the **raw-sql** container in **CSV** format.

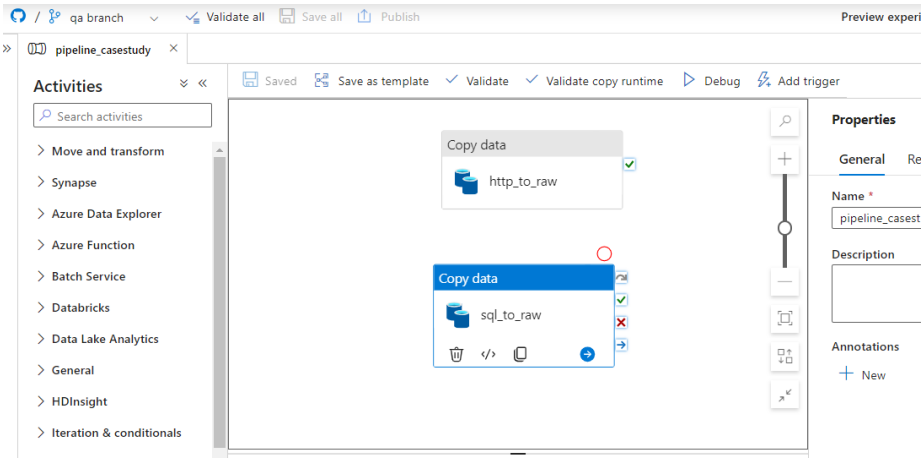


2. Version Control and Deployment

2.1. Creating a QA Branch

- After creating and testing the pipelines in the **Dev** branch, I created a **QA** branch in the GitHub repository.
- I did a pull request (PR) from the **Dev** branch to the **QA** branch.

- After the pull request was approved, I confirmed the pipelines were deployed successfully in the **QA** branch of Data Factory.



3. Data Transformation in Databricks

3.1. Setting Up Databricks

- I created a **Databricks workspace** and mounted the **raw-api** and **raw-sql** containers from ADLS Gen2 to Databricks using **secret scopes** for secure access to the storage account keys.

  - I used the following code to mount the storage account in Databricks:

```
                                                2

dbutils.fs.mount(
    source="wasbs://raw-api@casestudy1new.blob.core.windows.net",
    mount_point = "/mnt/raw-api",
    extra_configs={"fs.azure.account.key.casestudy1new.blob.core.windows.net": dbutils.secrets.get(scope =
    "casestudy", key = "storage")})
```

```
True
```

3.2. Reading Data from Parquet and CSV

- I read the data from the mounted **raw-api** container in **Parquet** format and the **raw-sql** container in **CSV** format using Spark.

  - For the Parquet file:

Read the users.parquet File

```
4

# Read the Parquet file into a DataFrame
file_path = "/mnt/raw-api/users.parquet"
df = spark.read.parquet(file_path)
```

- For the CSV file:

```
# Define the path to the txt file on the mounted container
file_path = "/mnt/raw-sql/dbo.football.txt"

# Read the txt file into a DataFrame
# Assuming the file is comma-delimited (adjust the delimiter as necessary)
df = spark.read.option("delimiter", ",").csv(file_path, header=True, inferSchema=True)
```

3.3. Data Cleaning and Transformation

- I cleaned the data by filtering out unwanted rows and re-indexing the columns.

- **For the Parquet data (raw-api)**:
  - I filtered out rows where the **username** is Samantha and re-indexed the data based on the username.
  - Example code:

```
from pyspark.sql import functions as F
from pyspark.sql.window import Window

# Step 1: Filter out the row where username is 'Samantha'
df_cleaned = df.filter(df.username != 'Samantha')

# Step 2: Re-index the `id` column to make sure it is consecutive
# This assumes you have an existing 'id' column and want to reassign the values.
df_cleaned = df_cleaned.withColumn('id', F.row_number().over(Window.orderBy('username')))

# Step 3: Show the cleaned DataFrame
df_cleaned.show()
```

- **For the CSV data (raw-sql)**:
- I filtered the data to keep only players with **Goals** greater than or equal to 80 and re-assigned the **Rank** column.

- Example code:

```
from pyspark.sql import functions as F
from pyspark.sql.window import Window

# Step 1: Filter the rows where 'Goals' >= 80
df_cleaned = df.filter(df['Goals'] >= 80)  #

# Step 2: Re-index the 'Rank' column to make it consecutive after filtering
# using row_number() over an ordered window to reassign the 'Rank'
window_spec = Window.orderBy(F.col('Goals').desc())  # Rank higher Goals first
df_cleaned = df_cleaned.withColumn('Rank', F.row_number().over(window_spec))

# Step 3: Show the cleaned DataFrame
df_cleaned.show(10)
```

Repartitioning and Coalescing Data

- To optimize performance, I repartitioned and coalesced the data to reduce the number of partitions before writing it back to ADLS.
    - Example code:

""" The coalesce() function in PySpark is often used to reduce the number of partitions in a DataFrame. It can help optimize performance by limiting the number of partitions when performing actions like writing to disk. """

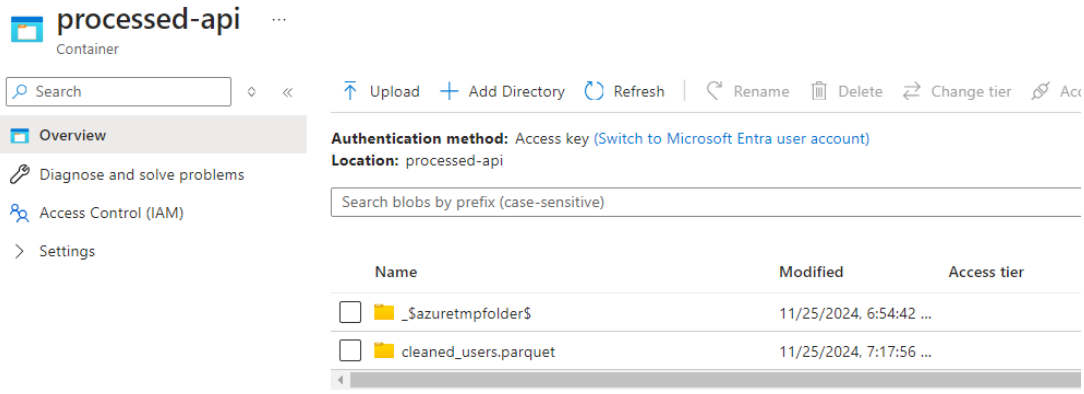

```
# Coalesce the DataFrame to a single partition
df_cleaned_coalesced = df_cleaned.coalesce(1)

# Show the DataFrame after coalescing
df_cleaned_coalesced.show()
```

4. Saving Data to Processed Containers

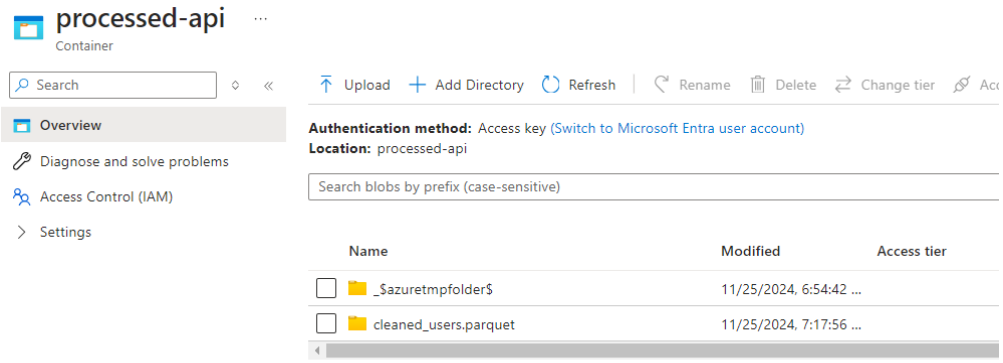
4.1. Saving to Processed Containers

- I saved the cleaned and transformed data into separate processed containers:

- **processed-api** container for the cleaned **raw-api** data (Parquet).



- **processed-sql** container for the cleaned **raw-sql** data (CSV).



- Example code:

For staging

20

```
dbutils.fs.mount(
    source="wasbs://staging-api@casestudy1new.blob.core.windows.net",
    mount_point="/mnt/staging-api",
    extra_configs={"fs.azure.account.key.casestudy1new.blob.core.windows.net": dbutils.secrets.get(scope =
    "casestudy", key = "storage")}
)
```

```
True
```

21

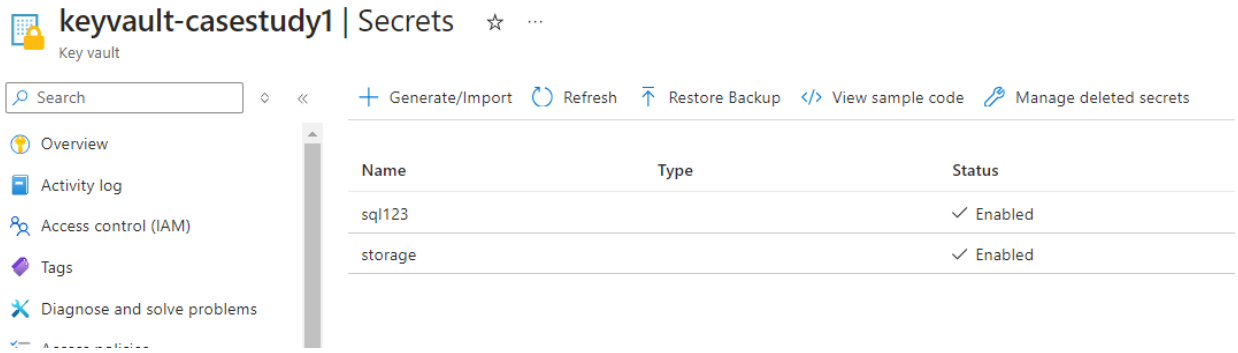
```
# Defining path to the 'staging-api' container
staging_container_path = "/mnt/staging-api/staging_users.parquet"

# Save the cleaned DataFrame to the 'staging-api' container using 'append' mode
df_cleaned_coalesced.write.mode('append').parquet(staging_container_path)
```

4.2. Appending Data to Staging Containers

- After processing, I transferred the data to **staging-api** and **staging-sql** containers for final staging.
    - Example code:
      df_cleaned_coalesced.write.mode('append').parquet("/mnt/staging-api/users_cleaned")

5. Security

- I used **Azure Key Vault** to store sensitive information like storage account keys. Databricks used the secret scope to securely access the storage account keys and mount the containers for data ingestion and saving.



6. Deployment and Monitoring

- All pipelines and notebooks were deployed and monitored in **Azure Data Factory** and **Databricks**.
- I used **Azure Data Factory** to schedule the pipelines, monitor the data transfer, and ensure smooth execution.

Conclusion

This project demonstrates end-to-end data ingestion, transformation, and storage using **Azure Data Factory**, **Databricks**, **Azure Key Vault**, and **Azure Data Lake Storage**. The data flows from an HTTP API and a Microsoft SQL Server, and goes through the cleaning and transformation process before being saved in different containers in the storage account.

This approach ensures secure data processing and optimized storage management with the help of **Azure Databricks**, **Azure Data Factory**, and **Azure Key Vault** for seamless integration and secure access to sensitive data.