Week VIII

Capstone Project

Intermediate Data Engineer in Azure
Trainer: Balazs Balogh

CUBIX
INSTITUTE OF TECHNOLOGY

Authentication – create and test the function

```python
def authenticate() -> None:
    """Authenticate the SparkSession using the predefined environment variables.

    """
    tenant_id = os.getenv("AZURE_TENANT_ID")
    client_id = os.getenv("AZURE_CLIENT_ID")
    client_secret = os.getenv("AZURE_CLIENT_SECRET")

    if not tenant_id or not client_id or not client_secret:
        raise ValueError("Missing one or more required environment variables: AZURE_TENANT_ID, AZURE_CLIENT_ID, AZURE_CLIENT_SECRET.")  # noqa: E501

    spark = SparkSession.getActiveSession()

    config = _create_authentication_config(tenant_id, client_id, client_secret)

    for key, value in config.items():
        spark.conf.set(key, value)
```

There are multiple ways to handle authentication between Databricks and other services like the Storage Account.

We will store our keys and secret as an Environment Variable in our cluster's configuration.

To test, package your code, and import it in the notebook along with the authentication function, and call **authenticate()**.

Pre-commit

pre-commit is a very useful tool to identify simple issues before pushing or commiting your code. We can choose **predefined** hooks, or can create **custom** ones.

Install it as a dev dependency with “**poetry add --group dev pre-commit”.**

After installed, create a **.pre-commit-config.yaml** in your project’s root directory:

```yaml
files: ^cubix_data_engineer_capstone/

repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: trailing-whitespace
      - id: check-added-large-files
      - id: check-ast
      - id: check-json
      - id: check-merge-conflict
      - id: check-yaml
      - id: debug-statements
      - id: end-of-file-fixer
      - id: mixed-line-ending
        args: [ '--fix=auto' ]

  # Ruff: linting, formatting, import sorting, and more.
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.2.1
    hooks:
      - id: ruff
        args: [ --fix ]
```

We’ll get back to changing the set up pre-commit later in the course.

Medallion Architecture I. – Bronze Layer

Our Bronze layer consists **reading** the file from the source system and **extracting** to our system. Files will be **unmodified**, essentially we are making a copy of the data in our system.

```python
from cubix_data_engineer_capstone.utils.datalake import read_file_from_datalake, write_file_to_datalake


def bronze_ingest(
        source_path: str,
        bronze_path: str,
        file_name: str,
        container_name: str,
        partition_by: list[str]
):
    """Extract files from the source, and load them to the desired container.

    :param source_path:                 Path to source file.
    :param bronze_path:                 Path to the bronze layer.
    :param file_name:                   Name of the file to ingest.
    :param container_name:              Name of the container holding the files.
    :param partition_by:                Column(s) to partition on. "None" by default.
    """
    df = read_file_from_datalake(container_name, f"{source_path}/{file_name}", "csv")

    return write_file_to_datalake(
        df=df,
        container_name=container_name,
        file_path=f"{bronze_path}/{file_name}",
        format="csv",
        mode="overwrite",
        partition_by=partition_by
    )
```

Test it with calling the bronze_ingest() on one the files. Make sure that you packaged your code, installed it in the notebook, and imported the bronze_ingest function.

(Optional) Speeding up package deployment

```
# Configuration
$dbfsPath = "dbfs:/mnt/packages"
$distPath = "C:/Users/balog/python_projects/cubix_data_engineer_capstone/dist"
$profile = "cubix_live"

# Step 1: Change to the project root and build the wheel using Poetry
Write-Output "Changing to the project root directory..."
Set-Location -Path "C:/Users/balog/python_projects/cubix_data_engineer_capstone"

Write-Output "Building wheel using Poetry..."
poetry build -f wheel
if ($LASTEXITCODE -ne 0) {
  Write-Output "Failed to build the wheel. Exiting."
  exit 1
}
Write-Output "Wheel build complete."

# Step 2: Clear the DBFS packages folder
Write-Output "Truncating $dbfsPath..."
databricks fs rm -r $dbfsPath --profile $profile
databricks fs mkdirs $dbfsPath --profile $profile
Write-Output "Truncated $dbfsPath."

# Step 3: Find the latest .whl file in the correct dist folder
Write-Output "Finding the latest .whl file in $distPath..."
$latestWhl = Get-ChildItem -Path $distPath -Filter *.whl | Sort-Object LastWriteTime -Descending | Select-Object -First 1
if (-not $latestWhl) {
  Write-Output "No .whl files found in $distPath. Exiting."
  exit 1
}
Write-Output "Latest .whl file found: $($latestWhl.FullName)"

# Step 4: Copy the latest .whl file to DBFS
Write-Output "Copying $($latestWhl.FullName) to $dbfsPath..."
databricks fs cp $latestWhl.FullName "$dbfsPath/" --profile $profile --overwrite
if ($LASTEXITCODE -ne 0) {
  Write-Output "Failed to copy the .whl file to DBFS. Exiting."
  exit 1
}
Write-Output "Copied $($latestWhl.FullName) to $dbfsPath."

# Completion message
Write-Output "Latest .whl file built and uploaded to $dbfsPath successfully."
```

To semi-automatise the deployment to our cluster we can create a bash script, which can package the code, and copy it to Databricks **dbfs** (Databricks File System).

In the notebook change the pip install line to:

!pip install /dbfs/mnt/packages/*

Medallion Architecture II. – Silver Layer - Calendar

```python
import pyspark.sql.functions as sf
from pyspark.sql import DataFrame


def get_calendar(calendar_raw: DataFrame) -> DataFrame:
    """Clean and transform data type for Calendar data.

    1. Select required columns.
    2. Cast them explicitly.
    3. Drop duplicates.

    :param calendar_raw:    Raw Calendar DataFrame.
    :return:                Transformed Calendar DataFrame.
    """

    return (
        calendar_raw
        .select(
            sf.col("Date").cast("date"),
            sf.col("DayNumberOfWeek").cast("int"),
            sf.col("DayName"),
            sf.col("MonthName"),
            sf.col("MonthNumberOfYear").cast("int"),
            sf.col("DayNumberOfYear").cast("int"),
            sf.col("WeekNumberOfYear").cast("int"),
            sf.col("CalendarQuarter").cast("int"),
            sf.col("CalendarYear").cast("int"),
            sf.col("FiscalYear").cast("int"),
            sf.col("FiscalSemester").cast("int"),
            sf.col("FiscalQuarter").cast("int"),
            sf.col("FinMonthNumberOfYear").cast("int"),
            sf.col("DayNumberOfMonth").cast("int"),
            sf.col("MonthID").cast("int"),
        )
        .dropDuplicates()
    )
```

Most of the magic happens in the Silver layer. Get the data from the Bronze layer’s folder, apply the required transformations, and add unit tests to make sure that the end result is correct.

Setting up pytest for unit testing

Pytest gives the user the ability to **use the common input** in various Python files, this is possible using **Conftest**. “conftest.py” is the file in which various functions are declared which can be accessed across various test files.

In our case creating a SparkSession before each unit test run is essential, and to avoid duplications in the code with multiple session creation, we can outsource this code to the conftest.py. Now we can access it as a fixture.

```python
from pyspark.sql import DataFrame, SparkSession
from pytest import fixture


SPARK = (
    SparkSession
    .builder
    .master("local")
    .appName("localTests")
    .getOrCreate()
)


@fixture
def spark():
    return SPARK.getActiveSession()
```

Pytest **fixtures** are a powerful feature that allows you to set up and tear down resources needed for your tests.