Stephen's blog

Follow

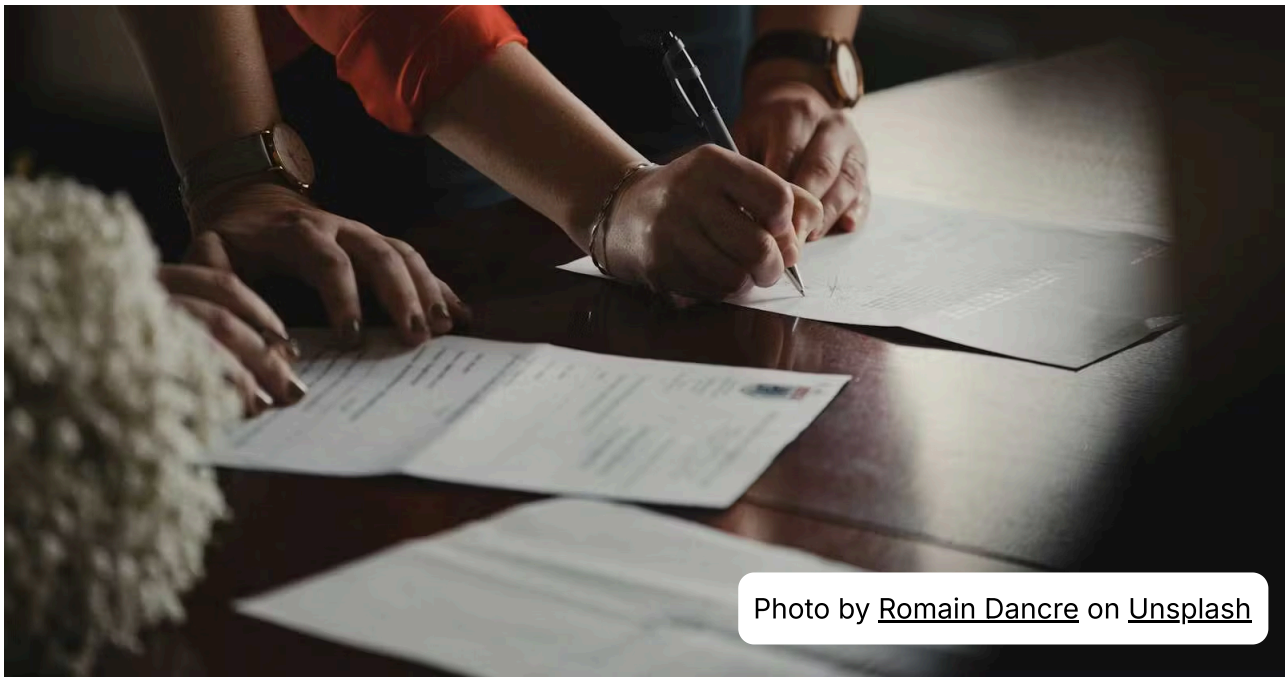


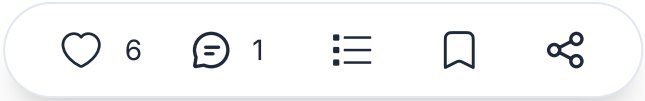
Photo by Romain Dancre on Unsplash

Create a web scraping pipeline with Python using data contracts

Add data quality to each source layer for quality in, quality out

Stephen David-Williams

Feb 13, 2024 · 24 min read



Show more

Preface 💫

This is a practical end-to-end data pipeline demo to show what a data project incorporating data contracts looks like.

We'll be scraping the Premier League table standings for the 2023/24 season, as of 13th February 2024 (the date this article is posted). The scraped data will be uploaded into Postgres database through multiple stages via data contracts, and then saved into AWS S3 programmatically.

Pseudo-code 🧠

Here's a rough brain dump on the steps we want the program to follow:

- Check if we're allowed to scrape data from website

- Scrape the dat[illegible]ise find out if they have an API w[illegible]
- Check the scraped data is what we expect it to look like
- Transform the dataframe
- Check the transformation steps have shaped the data into the expected output
- Write the dataframe to a CSV file
- Check the CSV file is in the expected format
- Upload to AWS S3
- Check if the upload was successful

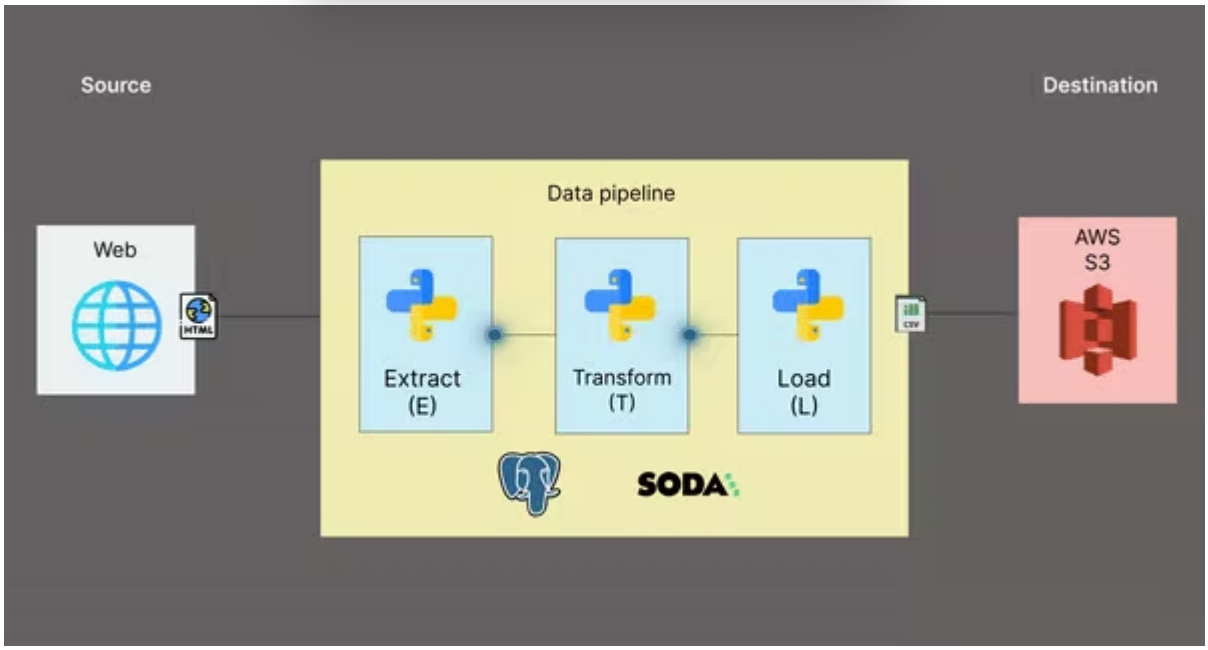
Technologies ⚙️

We've defined the basic steps our pipeline should follow, so now we can map the right modules to support the process:

- os
- boto3
- pandas
- requests
- selenium
- python-dotenv
- soda-core
- soda-core-postgres
- soda-core-contracts

Architectural diagram 📊

This is a visual rep[illegible]e looks like:



Folder structure 📁

COPY

```
|   .env
|   .gitignore
|   check_robots
|   requirements.txt
|
├──config
|       extraction_config.yml
|       transformation_config.yml
|
├──data
|       transformed_data.csv
|
├──drivers
|       chromedriver.exe
|
├──src
|   |   etl_pipeline.py
|   |   __init__.py
```

```
│   │
│   ├──extra
│   │       alter_col_type.py
│   │       main.py
│   │       scraping_bot.py
│   │       __init__.py
│   │
│   ├──loading
│   │       s3_uploader.py
│   │       __init__.py
│   │
│   ├──transformation
│   │       main.py
│   │       transformations.py
│   │       __init__.py
│   │
│   └──web_checker
│           robots_txt_checker.py
│           __init__.py
│
├──tests
│   │   __init__.py
│   │
│   ├──data_contracts
│   │       extraction_data_contract.yml
│   │       transformation_data_contract.yml
│   │
│   └──data_quality_checks
│           scan_extraction_data_contract.py
│           scan_transformation_data_contract.py
│
└──utils
        aws_utils.py
        db_utils.py
        __init__.py
```

Data sour

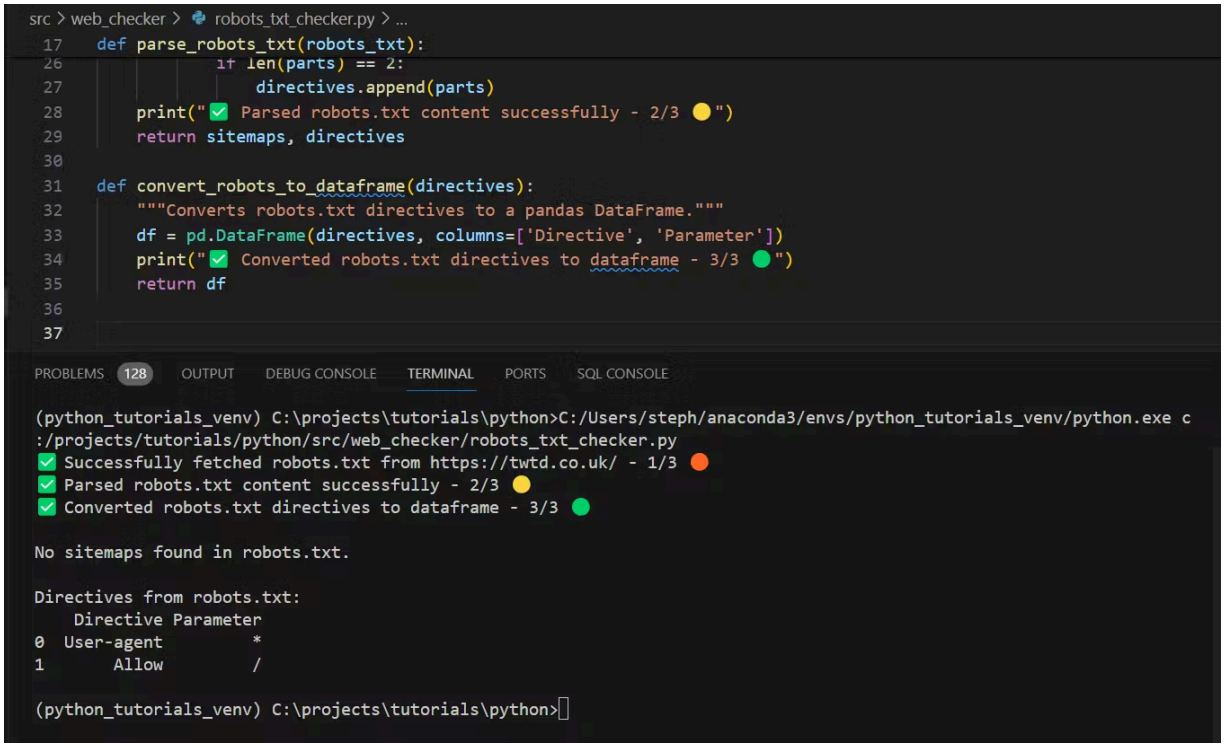
We're going to scrape from this URL: f'https://www.twtd.co.uk/league-tables/competition:premier-league/

Before we scrape the website, we need to verify if we're allowed to do this in the first place. No one likes a bunch of bots overloading their websites with traffic that doesn't benefit them, especially if it's used for commercial purposes.

To do this, we'll need to check the **robots.txt** of the site. Doing things programmatically reduces the chances of humans misinterpreting the response of this check.

So click here to find the Python code to doing just that which can also be found on GitHub.

...and this is the results:



The response contains an **Allow** directive that implies all users are allowed to access any areas o tries.

It's safe to assume [illegible] not allowed to access, but it's st[illegible]dance with their terms of service (ToS) respectfully.

Data contracts 📝

We'll be using data contracts in the blog, so let me provide TLDR context for you on it -

A data contract helps data consumers articulate everything they expect from the developers producing the data (including the ways to meet these expectations, timelines to meet the expectations etc), and these data producers fulfil the expectations using the details in this document (the data contact). A data contract is also referred to as a

- **data sharing agreement**
- **data usage agreement**

One key reason why data contracts are good is that it makes it difficult for changes to occur without consumption users and tools being informed first. So any changes occurring at the source level will be approved first before merging or replacing the existing schema structure.

On that note, consider this blog as a **mini proof of concept** into why coupling data contracts with your data pipelines (or even data products) may be a good approach to improving data governance initiatives with minimal effort.

In the real world, we would need some of the following listed below to implement them in a production environment (among other tools & considerations of course):

- **Data contract** [illegible] data rules and other constra[illegible]
- **Version control system** - to keep and maintain each version of the data contract details
- **Orchestration tool** - to automatically run the data quality tests once changes are detected in the data source
- **CI/CD pipeline** - to merge the changes if they pass the data quality tests, or circuit break the entire operation if they fail

For simplicity's sake, we'll just use the **data contract** and leave the other components mentioned because the current release of Soda data contracts doesn't support Docker. So once it does, I'll write a separate blog for you on this.

Throughout this blog, we'll

- use Soda to design and launch our data contracts
- play the role of data producers and simulate the data consumers for each data contract created

*Disclaimer: I have**no**affiliates or financial links with Soda whatsoever, I'm simply using this objectively for experimentational purposes, so opinions are my own and developed progressively through POCs like these.*

Extract (E) 🚜

To scrape the data we want, we need to first understand what we need to scrape in the first place...from the perspective of the users who will be using it themselves.

We'll use a data contract to achieve this.

To set up a data c[illegible]s:

- **configuration.yml** - to set up the details of the data source
- **contract.yml** - to add the information about the data source, the schema and the data quality checks to run
- **scan_contract.py** - to run the data quality checks

In relation to this extraction stage, the data consumers are the **transformation team**, who happen to have submitted their list of expectations to us (the data producers) for the scraped data they want landed in Postgres:

- 20 unique rows
- 16 fields
- all columns need to be integers (except the 2nd one)
- columns 2 must be varchar type

COPY

```
├──extraction
│       alter_col_type.py
│       main.py
│       scraping_bot.py
│       __init__.py
```

We can document these into our data contract we've named **extraction_data_contract.yml** file, just like so:

COPY

```
dataset: scraped_fb_data

columns:
```

```
  - name: pos
    data_type
    unique: true
  - name: team
    data_type: varchar
    not_null: true
  - name: p
    data_type: integer
    not_null: true
  - name: w1
    data_type: integer
    not_null: true
  - name: d1
    data_type: integer
    not_null: true
  - name: l1
    data_type: integer
    not_null: true
  - name: gf1
    data_type: integer
    not_null: true
  - name: ga1
    data_type: integer
    not_null: true
  - name: w2
    data_type: integer
    not_null: true
  - name: d2
    data_type: integer
    not_null: true
  - name: l2
    data_type: integer
    not_null: true
  - name: gf2
    data_type: integer
    not_null: true
  - name: ga2
```

```
      data_type: integer
      not_null:
    - name: gd
      data_type: integer
      not_null: true
    - name: pts
      data_type: integer
      not_null: true
    - name: date
      data_type: date
      not_null: true

checks:
  - row_count = 20
```

Now let's build on top of these expectations.

Our scraping mechanism needs to:

1. Scrape the content and print a message to signal the job succeeded
2. Extract the required data using Selenium
3. Read the extracts into a dataframe

We need to establish where the content we want to scrape is located on the webpage, so that we know how we want to go about scraping it in the first place. This requires us looking at the DOM the website generates when it first loads.

The **DOM (Document Object Model)** is just a representation of the HTML objects that form the content you see on a website. So each paragraph, heading and button is on a webpage represented as a node within the DOM in a tree-like structure.

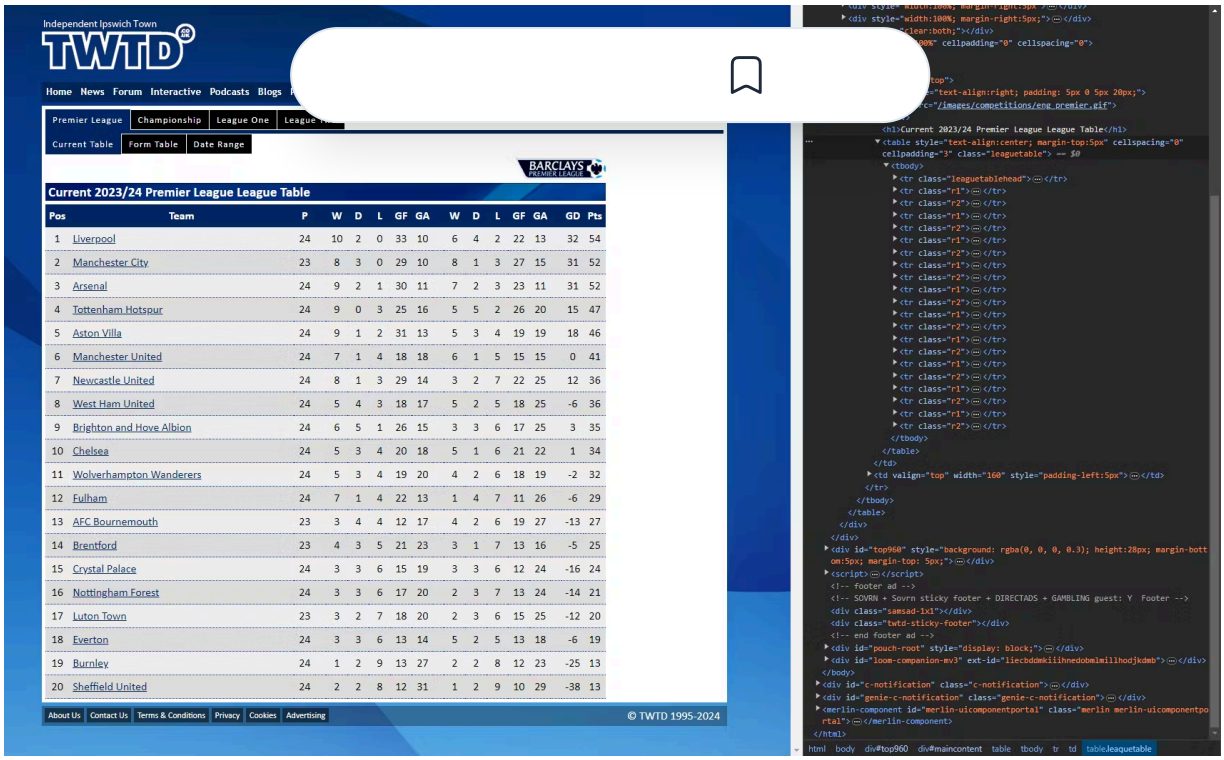
If you're reading th[illegible]his writing), you can check the DO[illegible]our keyboard (or right click on a section of the webpage and click **Inspect**). This should open up the **DevTools** panel - you can find the DOM under the **Elements** tab.

Selenium provides different methods to choose from. We could access the DOM's elements by:

- ID
- Name
- Tag name
- CSS selector
- XPath

This is what we know about the information we see in the DOM:

- the entire Premier League table standings is represented by the <table> tag
- the same <table> tag has a class attribute with the value "leaguetable"
- each row in the Premier League table standings is represented by the <tr> tag



Based on this, we can now formulate a scraping approach for this:

- Find the <table> tag via its class name (leaguetable)
- Iterate through each row within the table (i.e. each <tr> element within the <table> tag)
- For each row, extract from the data from each of its cells

For this we'll create these files:

- db_utils.py
- scraping_bot.py
- main.py

The **db_utils.py** module will hold the commands to interact with the Postgres database:

COPY

```
import os
import psycopg2
```

```
import pandas as pd
from dotenv i

# Load environment variables from .env file
load_dotenv()

def connect_to_db():
        """
        Uses environemnt variables to connect to the Postgres
        """
    HOST=os.getenv('HOST')
    PORT=os.getenv('PORT')
    DATABASE=os.getenv('DATABASE')
    POSTGRES_USERNAME=os.getenv('POSTGRES_USERNAME')
    POSTGRES_PASSWORD=os.getenv('POSTGRES_PASSWORD')

    # Use environment variables directly
    try:
        db_connection = psycopg2.connect(
            host=HOST,
            port=PORT,
            dbname=DATABASE,
            user=POSTGRES_USERNAME,
            password=POSTGRES_PASSWORD,
        )
        db_connection.set_session(autocommit=True)
        print("✅ Connection to the database established succ
        return db_connection
    except Exception as e:
        raise Exception(f"❌[ERROR - DB CONNECTION]: Error cc



def create_extracted_schema_and_table(db_connection, schema_r
        """
        Creates a schema and table in the Postgres database (
        """
```

```
        cursor = db_connection.cursor()
        cursor.execute(f"DROP TABLE IF EXISTS {schema_name
        db_connection.commit()

        create_table_query = f"""
            CREATE TABLE IF NOT EXISTS {schema_name}.{table_name}
                "pos" INTEGER,
                "team" TEXT NOT NULL,
                "p" INTEGER,
                "w1" INTEGER,
                "d1" INTEGER,
                "l1" INTEGER,
                "gf1" INTEGER,
                "ga1" INTEGER,
                "w2" INTEGER,
                "d2" INTEGER,
                "l2" INTEGER,
                "gf2" INTEGER,
                "ga2" INTEGER,
                "gd" INTEGER,
                "pts" INTEGER,
                "date" DATE
            )
        """
        cursor.execute(create_table_query)
        db_connection.commit()
        cursor.close()

    def insert_extracted_data_to_table(db_connection, schema_name
        """
            Inserts data from pandas dataframe into the extractic
            """
            cursor = db_connection.cursor()
        for index, row in dataframe.iterrows():
            data = tuple(row)
            placeholders = ",".join(["%s"] * len(row))
            insert_query = f"""
```

```
                INSERT INTO {schema_name}.{table_name} (
                                                    , "l1", "gf1",
                )
                VALUES ({placeholders})
            """
            try:
                cursor.execute(insert_query, data)
                db_connection.commit()
            except Exception as e:
                print(f"❌Failed to insert data: {data}❌. Error:
        cursor.close()

    # Transformation

    def create_transformed_schema_and_table(db_connection, schema
            """
            Creates a schema and table in the Postgres database (
            """
        cursor = db_connection.cursor()
        cursor.execute(f"CREATE SCHEMA IF NOT EXISTS {schema_name
        db_connection.commit()

        create_table_query = f"""
            CREATE TABLE IF NOT EXISTS {schema_name}.{table_name}
                "position" INTEGER,
                "team_name" VARCHAR,
                "games_played" INTEGER,
                "goals_for" INTEGER,
                "goals_against" INTEGER,
                "goal_difference" INTEGER,
                "points" INTEGER,
                "match_date" DATE
            )
        """
        cursor.execute(create_table_query)
        db_connection.commit()
        cursor.close()
```

```
# Function to                                    ayer
def fetch_extraction_data(db_connection, schema_name, table_n
    """
    Pulls data from the extraction table in Postgres.
    """
    query = f"SELECT * FROM {schema_name}.{table_name};"
    return pd.read_sql(query, db_connection)

def insert_transformed_data_to_table(db_connection, schema_na
    """
    Inserts data from pandas dataframe into the transform
    """
    cursor = db_connection.cursor()

    # Check the dataframe columns before insertion
    # print(f"Dataframe columns: {dataframe.columns.tolist()}

    # Building column names for the INSERT INTO statement
    columns = ', '.join([f'"{c}"' for c in dataframe.columns]

    # Building placeholders for the VALUES part of the INSERT
    placeholders = ', '.join(['%s' for _ in dataframe.columns

    # Construct the INSERT INTO statement
    insert_query = f"INSERT INTO {schema_name}.{table_name} (

    # Execute the INSERT INTO statement for each row in the d
    for index, row in dataframe.iterrows():
        # Print the row to be inserted for debugging purposes
        # print(f"Row data: {tuple(row)}")
        try:
            cursor.execute(insert_query, tuple(row))
            db_connection.commit()
        except Exception as e:
            print(f"❌Failed to insert transformed data: {tup
    cursor.close()
```

```
# Loading
def fetch_transformed_data(db_connection):
        """
        Pulls data from the transformation table in Postgres.
        """
    print("Fetching transformed data from the database...")
    try:
        query = "SELECT * FROM staging.transformed_fb_data;"
        df = pd.read_sql(query, db_connection)
        print("Data fetched successfully.")
        return df
    except Exception as e:
        raise Exception(f"❌ [ERROR - FETCH DATA]: {e}")

def convert_dataframe_to_csv(df, filename):
        """
        Converts a pandas dataframe to a CSV file and saves i
        """
    target_destination = 'data/'
    full_file_path = f"{target_destination}{filename}"
    print(f"Converting dataframe to CSV ('{filename}')...")
    try:
        df.to_csv(full_file_path, index=False)
        print(f"CSV file '{filename}' created and saved to ta
        print(f"Full file path: '{full_file_path}'")
    except Exception as e:
        raise Exception(f"❌ [ERROR - CSV CREATION]: {e}")
```

The **.env** file will hold the database credentials our scraper needs to connect to the Postgres database:

COPY

```
# Postgres
HOST="localhost"
```

```
PORT=5434
DATABASE="tes
POSTGRES_USERNAME=${POSTGRES_USERNAME}
POSTGRES_PASSWORD=${POSTGRES_PASSWORD}

# AWS
...
```

The **scraping_bot.py** file will contain our scraping logic:

COPY

```
import pandas as pd
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as

def scrape_data(dates, executable_path="drivers/chromedriver.
    """
    Scrapes football league data for the dates provided f
    """
    service = Service(executable_path=executable_path)
    driver = webdriver.Chrome(service=service)
    all_data = []

    for match_date in dates:
        formatted_date = pd.to_datetime(match_date).strftime(
        football_url = f'<https://www.twtd.co.uk/league-table
        driver.get(football_url)
        wait = WebDriverWait(driver, 10)
        table_container = wait.until(EC.presence_of_element_l
        rows = table_container.find_elements(By.TAG_NAME, "tr

        for idx, row in enumerate(rows[1:], start=1):
            cols = row.find_elements(By.TAG_NAME, "td")
```

```
                row_data = [col.text.strip() for col in cols if c
                print(f"{' | '.join(row_data)}") if show_outp
                row_data.append(formatted_date)
                all_data.append(row_data)

            if show_output:
                print(f"Premier League Table Standings (as of {fo
                print('-'*60)
                for row_data in all_data:
                    print(' '.join(row_data))
                print('\\n' + '-'*60)

            driver.implicitly_wait(2)

        driver.quit()

            # I've commented out the normal-cased columns because
        # columns = ["Pos", "Team", "P", "W1", "D1", "L1", "GF1",
        columns = ["pos", "team", "p", "w1", "d1", "l1", "gf1", "
        df = pd.DataFrame(all_data, columns=columns)
        return df
```

We can use the **scan_extraction_data_contract.py** file to scan the dataframe and check if its contents meets the requirements laid out by the data consumer:

COPY

```
from soda.contracts.data_contract_translator import DataContr
from soda.scan import Scan
import logging
import os

def run_dq_checks_for_extraction_stage():
        """
        Performs data quality checks for the extraction stage
```

```
        based on the predefined data contract.

        1. Pulls the YAML files for the config + data contrac
        2. Reads the data source, schema and data quality che
        3. Executes the data quality checks
        """

    # Correctly set the path to the project root directory
    project_root_directory = os.path.dirname(os.path.dirname(

    # Construct the full file paths for the YAML files
    extraction_data_contract_path = os.path.join(project_root
    extraction_config_yaml_path = os.path.join(project_root_d

    # Read the data contract file as a Python string
    with open(extraction_data_contract_path) as f:
        data_contract_yaml_str: str = f.read()

    # Translate the data contract standards into SodaCL
    data_contract_parser = DataContractTranslator()
    sodacl_yaml_str = data_contract_parser.translate_data_cor

    # Log or save the SodaCL checks file to help with debuggi
    logging.debug(sodacl_yaml_str)

    # Execute the translated SodaCL checks in a scan
    scan = Scan()
    scan.set_data_source_name("scraped_fb_data")
    scan.add_configuration_yaml_file(file_path=extraction_cor
    scan.add_sodacl_yaml_str(sodacl_yaml_str)
    scan.execute()
    scan.assert_no_checks_fail()

if __name__ == "__main__":
    run_dq_checks_for_extraction_stage()
```

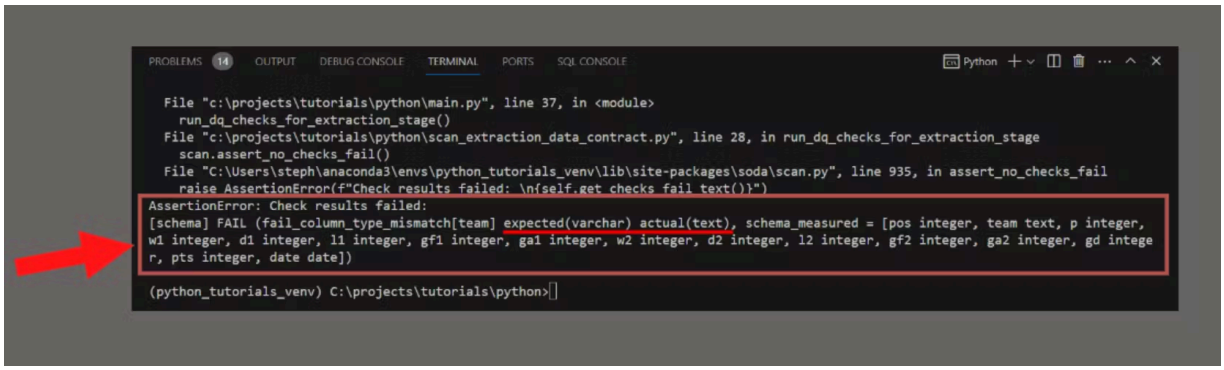
...and the main.py file in the root directory to load the data to Postgres and then run the data quality checks.

COPY

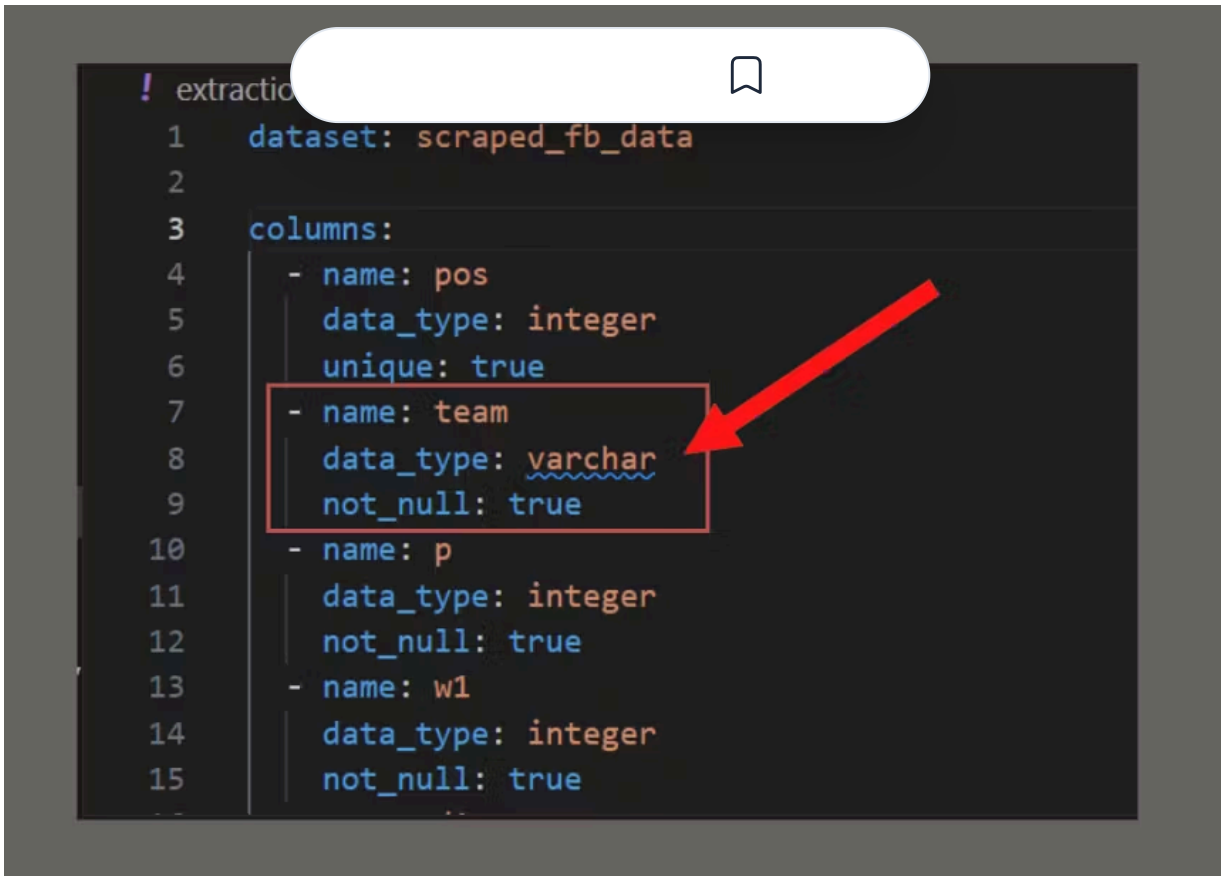
```
try:
    from scraping_bot import scrape_data
    from utils.db_utils import connect_to_db, create_extracte
    from tests.data_quality_checks.scan_extraction_data_contr

def extraction_job():
    # Flag for running the data quality checks only
    RUN_DQ_CHECKS_ONLY = True

    if not RUN_DQ_CHECKS_ONLY:
        # Connect to the database
        connection = connect_to_db()

        # Create schema and table if they don't exist
        schema_name = 'raw'
        table_name = "scraped_fb_data"
        create_extracted_schema_and_table(connection, schema_

        # Scrape the data and store in a DataFrame
        dates = [
            "2024-02-13",
            # "2024-03-31",
            # "2024-04-30",
            # "2024-05-31"
        ]
        df = scrape_data(dates, show_output=False)

        # Insert data into the database from the DataFrame
        insert_extracted_data_to_table(connection, schema_nam

        # Close the database connection
        connection.close()
```

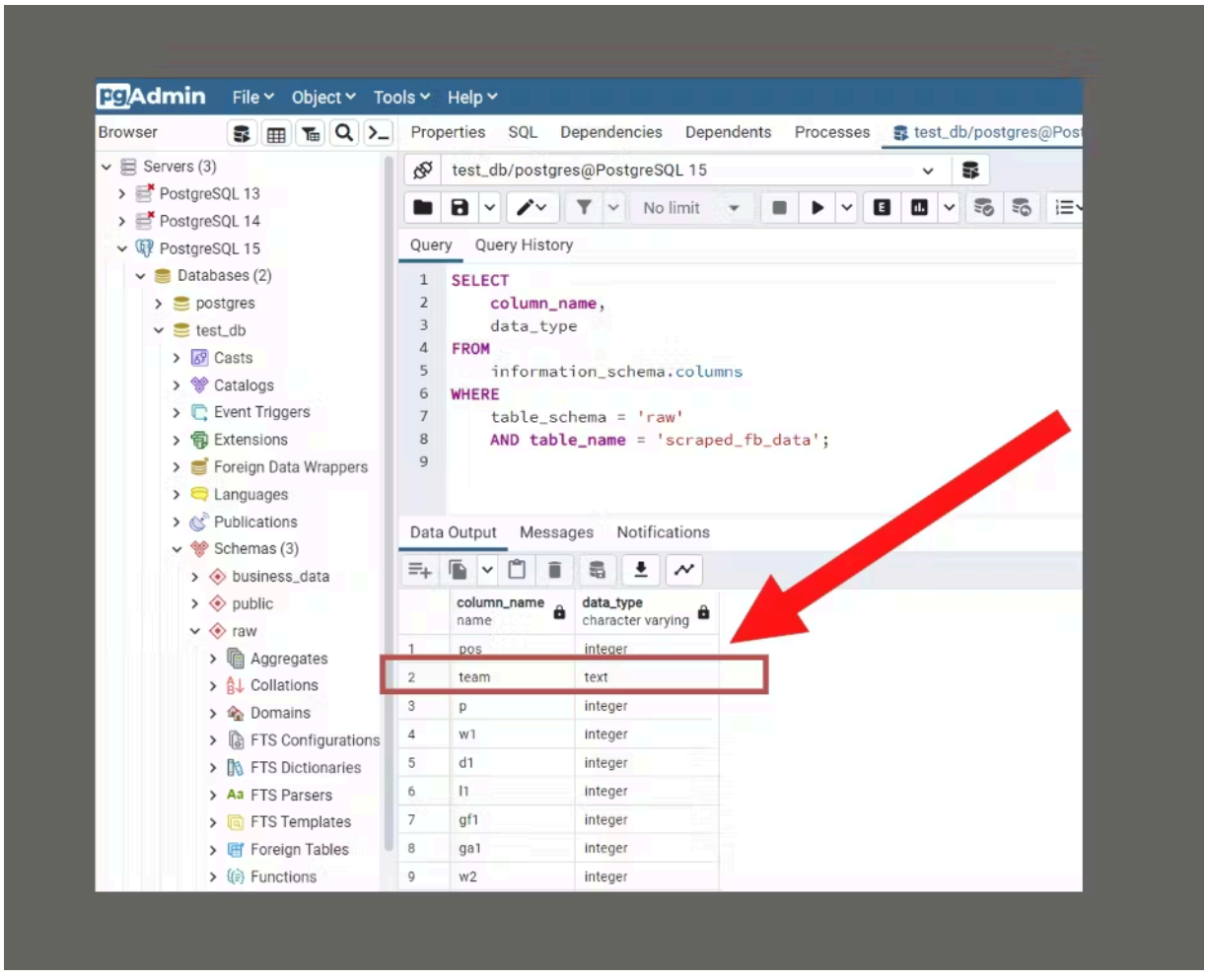
```
        # Run
        run_dq_checks_for_extraction_stage()

    else:
        run_dq_checks_for_extraction_stage()

if __name__=="__main__":
    extraction_job()
```

...and our program detected an issue:



The scan states there was a mismatch between the expected data type of the **team** column (varchar) and the actual data type for the **team** column in the Postgres database (text).


! extractio
dataset: scraped_fb_data
columns:
- name: pos
data_type: integer
unique: true
- name: team
data_type: varchar
not_null: true
- name: p
data_type: integer
not_null: true
- name: w1
data_type: integer
not_null: true


pgAdmin File Object Tools Help
Browser
Properties SQL Dependencies Dependents Processes test_db/postgres@Post
Servers (3)
PostgreSQL 13
PostgreSQL 14
PostgreSQL 15
Databases (2)
postgres
test_db
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (3)
business_data
public
raw
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
test_db/postgres@PostgreSQL 15
No limit
Query Query History
SELECT
column_name,
data_type
FROM
information_schema.columns
WHERE
table_schema = 'raw'
AND table_name = 'scraped_fb_data';
Data Output Messages Notifications
column_name name
data_type character varying
1 pos integer
2 team text
3 p integer
4 w1 integer
5 d1 integer
6 l1 integer
7 gf1 integer
8 ga1 integer
9 w2 integer

This now forces us to [illegible]s (among many potential others):

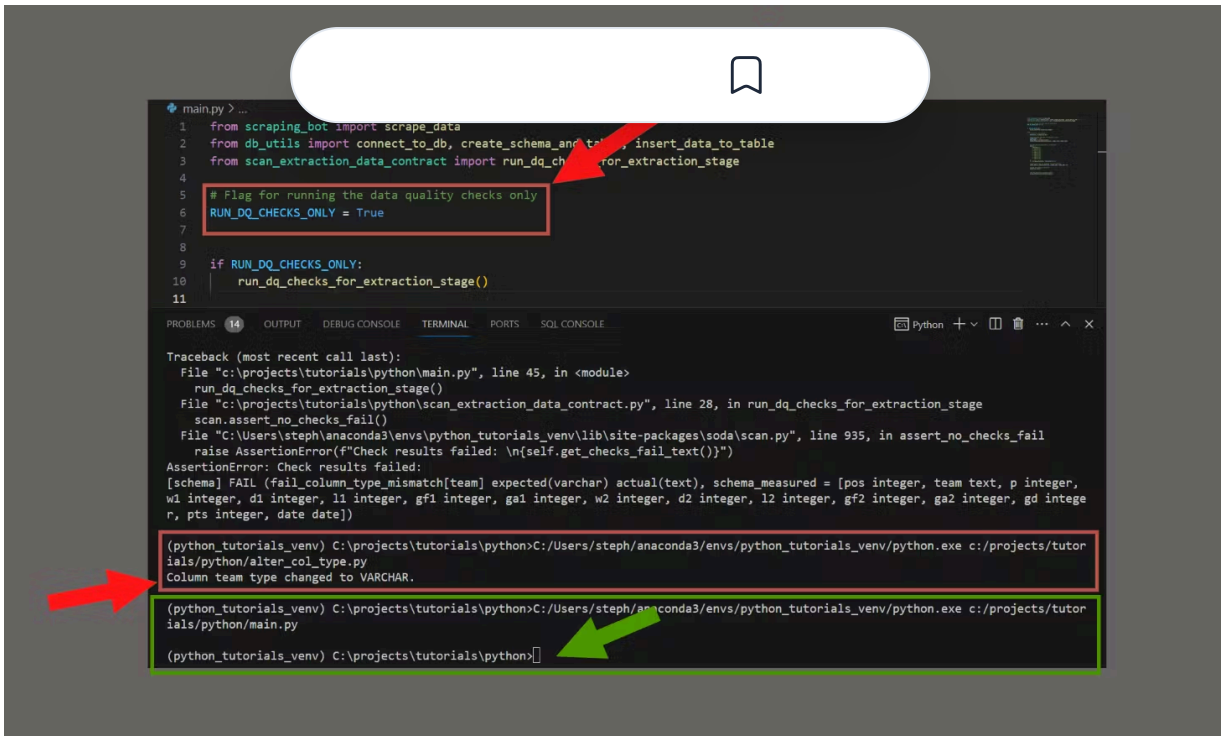
- raise the flagged error message with the 'data consumer' for an adequate response (change it, leave it, or do something else?)
- ignore the previous option, enforce one ourselves and update the consumers on which approach we've adopted (provided we have the official green light from the consumers to do so)

Whichever option we advance with, a good thing about going down the data contract route is that the data quality issue is **immediately exposed,** which gives the data producers the opportunity to immediately address it before it reaches any consumption tools.

In a real world scenario, an issue like this would be more severe because there would be high revenue-generating products that depend on a column's data type needing to be accurate.

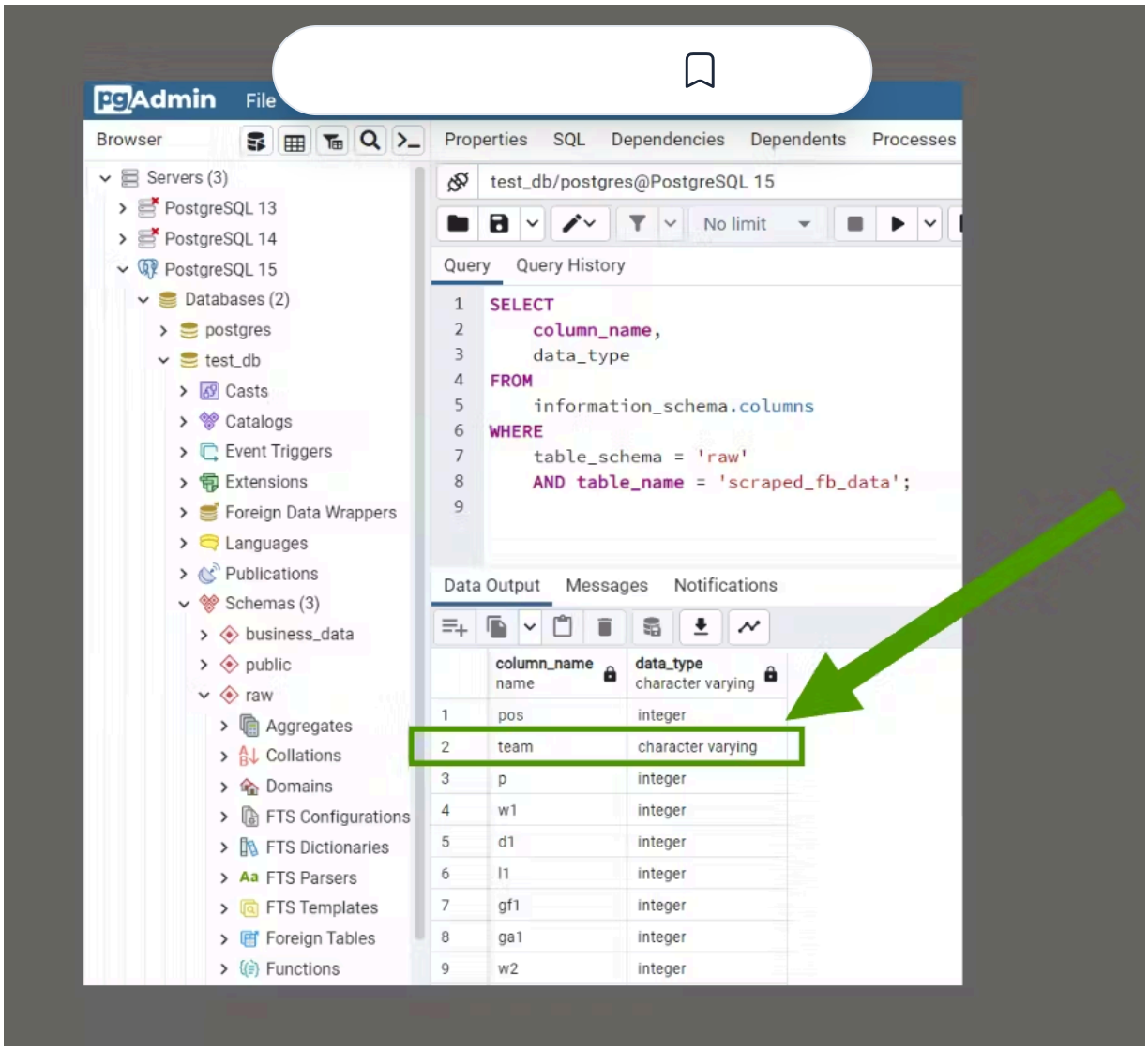
If these issues go unnoticed, they could break downstream tools and make it harder to figure out where the bugs are occurring. This inevitably means more time (and money) spent diagnosing and firefighting to solve the avoidable issue, which could have been solved if the data quality expectations were articulated in an accessible and version-controllable file like a data contract (not too different from this simple YAML example).

Now we can troubleshoot the issue and re-run the checks to confirm if this works well again:



No issues returned from our checks, looks good so far.

But let’s take a quick glance into Postgres just to be extra certain...:



Success! Our data quality checks have passed for the extraction stage✅!

Transform (T) 🔨

This is where we define the steps for curating the data into the format. These steps will be a direct function of the data consumer's expected version of the dataframe.

The data consumers in relation to this stage would be the loading team. Here are their list of expectations:

- renamed field names to longer form versions

- using the hom… …points (but deducting 10 … …olations)
- sort + reset the index ('position' field) once the points have been recalculated

Now that the expectations have been defined by the consumer, we can begin our development.

So what do we need to do?

- Connect to Postgres
- Create the schema and table for the transformed data
- Pull data from the extraction layer in Postgres
- Transform the data based on the consumer requirements
- Insert the data to the **transformed_data** table in Postgres
- Run the data quality tests

COPY

```
├──transformation
│       add_columns.py
│       main.py
│       transformations.py
│       __init__.py
```

Here's the transformations.py file:

COPY

```
import pandas as pd

def rename_fields(df):
    """
```

```
    Rename each field name to a longer form version.
    """
    df_renamed = df.rename(columns={
        'pos': 'position',
        'team': 'team_name',
        'p': 'games_played',
        'w1': 'home_wins',
        'd1': 'home_draws',
        'l1': 'home_losses',
        'gf1': 'home_goals_for',
        'ga1': 'home_goals_against',
        'w2': 'away_wins',
        'd2': 'away_draws',
        'l2': 'away_losses',
        'gf2': 'away_goals_for',
        'ga2': 'away_goals_against',
        'gd': 'goal_difference',
        'pts': 'points',
        'date': 'match_date'
    })
    return df_renamed

def calculate_points(df):
    """
    Use the home and away columns to calculate the points, ar
    deduct 10 points from Everton FC due to PSR violations st
    from November 2023.
    """
    # Calculate points normally for all rows
    df['points'] = (
        df['home_wins'] * 3 + df['away_wins'] * 3 +
        df['home_draws'] + df['away_draws']
    )

    # Calculate total wins, draws, and losses
    df['wins'] = df['home_wins'] + df['away_wins']
    df['draws'] = df['home_draws'] + df['away_draws']
```

```
    df['losses'] = df['home_losses'] + df['away_losses']

    df['goals_for'] = df['home_goals_for'] + df['away_goals_f
    df['goals_against'] = df['home_goals_against'] + df['away

    # Convert the match_date from string to datetime for comp
    df['match_date'] = pd.to_datetime(df['match_date'])


    return df

def deduct_points_from_everton(df):
    """
    Deduct points for Everton FC if the match_date is in or a
    """
    psr_violation_start_date = pd.to_datetime('2023-11-01')
    everton_mask = (df['team_name'] == 'Everton') & (df['matc
    df.loc[everton_mask, 'points'] -= 10

    return df

def sort_and_reset_index(df):
    """
    Sort the dataframe based on the Premier League table star
    and reset the 'position' column to reflect the new rankir


    """
    # Sort by points, then goal difference, then goals for
    df_sorted = df.sort_values(by=['points', 'goal_difference

    # Reset the index to reflect the new ranking
    df_sorted = df_sorted.reset_index(drop=True)

    # Update the 'position' column to match the new index
    df_sorted['position'] = df_sorted.index + 1
```

```
    return df_sorted

def transform_data(df):
    """
    Apply all the transformation intents on the dataframe.
    """
    df_renamed = rename_fields(df)
    df_points_calculated = calculate_points(df_renamed)
    df_points_deducted = deduct_points_from_everton(df_points

    # Create the final dataframe with desired columns only
    df_cleaned = df_points_deducted[['position', 'team_name',

    # Sort the dataframe by points, goal_difference, and goal
    df_final = df_cleaned.sort_values(by=['points', 'goal_dif

    # Reset the position column to reflect the new ranking af
    df_final.reset_index(drop=True, inplace=True)
    df_final['position'] = df_final.index + 1

    return df_final
```

We're

- renaming each field
- creating new calculated columns based on existing numerical columns
- deducting points from Everton FC starting from November 2023 (due to PSR violations)
- dropping the home and away fields once we're done calculating the points

Most of these are f[illegible] straight into the Everton FC situat[illegible]

Just for context, there's a rule known as the PSR (Profit & Sustainability Rule) which states every Premier League club is allowed to lose a maximum of £105 million, but Everton FC lost £124.5 million up to the 2021/22 period, which exceeded the PSR threshold by almost £20 million.

As far as the independent commission reviewing their case was concerned, Everton violated this rule, and therefore penalized with a 10-point deduction. This is no small punishment by any means...this has impacted Everton's position on the Premier League table, which could potentially place them in danger of being relegated from the league entirely for the first time in their history. This naturally forces Everton to appeal this decision, as they also believe the commission have not accurately calculated the losses, so at the time of this writing (February 2024), the point deduction still remains.

So how do we incorporate these point deductions into the data pipeline?

Our function need

- set the **start date** for the penalty (17th November 2023)
- highlight the rows that correspond to every game played by Everton FC after the penalty date
- apply the penalty using a **boolean-based** mask

So here's what we got:

COPY

```
def deduct_points_from_everton(df):
    """
    Deduct points for Everton FC if the match_date is in or a
    """
    psr_violation_start_date = pd.to_datetime('2023-11-17')
    everton_mask = (df['team_name'] == 'Everton') & (df['matc
    df.loc[everton_mask, 'points'] -= 10

    return df
```

This is what the t**ransformation_data_contract.yml** file would look like:

COPY

```
dataset: transformed_fb_data

columns:
  - name: position
    data_type: integer
    unique: true
  - name: team_name
    data_type: varchar
    not_null: true
  - name: games_played
    data_type: integer
```

```
    not_null: true
  - name: win
    data_type: integer
    not_null: true
  - name: draws
    data_type: integer
    not_null: true
  - name: losses
    data_type: integer
    not_null: true
  - name: goals_for
    data_type: integer
    not_null: true
  - name: goals_against
    data_type: integer
    not_null: true
  - name: goal_difference
    data_type: integer
    not_null: true
  - name: points
    data_type: integer
    not_null: true
    valid_min: 0
  - name: match_date
    data_type: date
    not_null: true

checks:
  - row_count = 20  # The table must contain 20 rows
  - min(games_played) >= 0  # Games played must be non-negati
  - max(goal_difference) <= 100  # Replace 100 with your maxi
  - missing_count(team_name) = 0  # Ensure no missing team na
  - failed rows:
      name: No negative points permitted
      fail query: |
        SELECT team_name, points
        FROM transformed_fb_data
```

```
        WHERE points < 0
  - failed ro
      name: Check Everton's points post-PSR penalty
      fail query: |
        WITH PrePointsDeduction AS (
          SELECT SUM(points) as pre_penalty_points
          FROM transformed_fb_data
          WHERE team_name = 'Everton' AND match_date < '2023-
        ), PostPointsDeduction AS (
          SELECT SUM(points) as post_penalty_points
          FROM transformed_fb_data
          WHERE team_name = 'Everton' AND match_date >= '2023
        )
        SELECT
          (SELECT pre_penalty_points FROM PrePointsDeduction)
          (SELECT post_penalty_points FROM PostPointsDeductic
          (SELECT pre_penalty_points FROM PrePointsDeduction)
        WHERE (SELECT pre_penalty_points FROM PrePointsDeduct
```

Now how do we incorporate these point deductions into the data quality checks in the data contract?

This requires us to break down the logical sequence of steps the check needs to take to make this happen. We would need to

- calculate the total number of points before the penalty date (i.e. November 2023)
- calculate the total number of points after the penalty date
- check if the difference is 10 points

As of the time of this writing, creating two CTEs was the best approach I could come up with and incorporate it into the checks using **SodaCL**. There may be a better approach down the line but this seems to be the

most sensible way [illegible] Soda release for YAML-based data

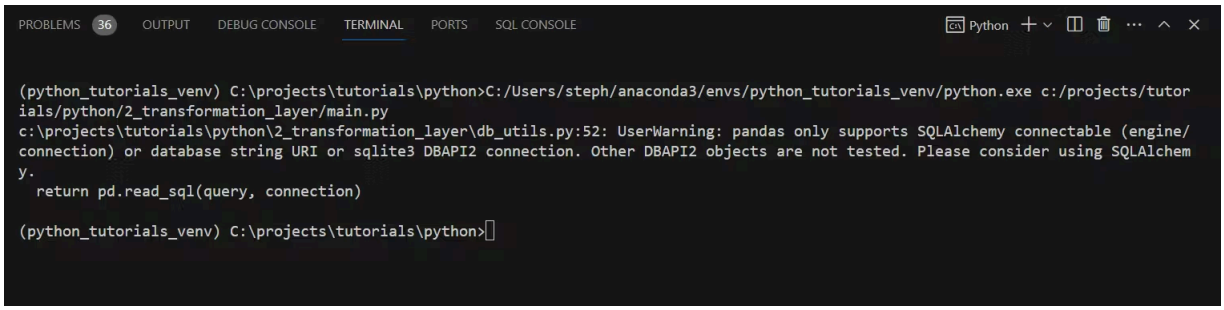
...here's the main.py:

COPY

```
try:
    from utils.db_utils import connect_to_db, create_transfor
    from tests.data_quality_checks.scan_transformation_data_c
    from transformations import transform_data
except:
    from utils.db_utils import connect_to_db, create_transfor
    from tests.data_quality_checks.scan_transformation_data_c
    from .transformations import transform_data

def transformation_job():
    # Establish a connection to the database
    connection = connect_to_db()

    # Define schema and table names for extracted and transfc
    extracted_schema_name = 'raw'
    extracted_table_name = 'scraped_fb_data'
    transformed_schema_name = 'staging'
    transformed_table_name = 'transformed_fb_data'

    # Create schema and table for the transformed data if not
    create_transformed_schema_and_table(connection, transform

    # Fetch data from the extraction layer
    extracted_data = fetch_extraction_data(connection, extrac

    # Perform data transformation
    transformed_data = transform_data(extracted_data)

    # Insert transformed data into the transformed_data table
    insert_transformed_data_to_table(connection, transformed_
```

```
    # Run dat                                    rmation stage
    run_dq_checks_for_transformation_stage()

    # Close the database connection
    connection.close()

if __name__ == "__main__":
    transformation_job()
```

So let's run the main.py file:



There are no data quality errors returned, so we're good to advance to the next stage ✅

Load (L) 🚚

Now we need to write the dataframe to a CSV file to get it ready to upload to our AWS S3 bucket.

At this point, we don't need to apply data quality checks because there are no more transformation processes applied at the data level.

The only transformation we need now is to convert the transformed data into CSV format and upload it to AWS S3.

Here are the steps we need to take for this stage for each table standing we process:

- read the **trans**[illegible] Pandas dataframe
- Convert the Pandas dataframe into a CSV file
- Connect Python to AWS services using boto3
- Set up S3 client
- Create bucket if it doesn't exist to hold premier league table standings data
- Upload CSV file to S3 bucket
- Check the CSV file upload is successful

For this to occur, the load layer directory needs files that

- contain the Postgres configuration details
- AWS S3 configuration details
- Python logic for processing the data from Postgres to AWS S3

So this is the folder structure for this section:

```
├──loading
│       s3_uploader.py
│       __init__.py
```

Target destination: AWS S3 ⚙️

We'll use **boto3** to interact with AWS services using Python.

You'll need an AWS account, so be sure to set one up to follow along this part.

- Create .env file

- Add environm[illegible] file
- Read them into your Python script
- Upload the CSV file into the S3 bucket
- Perform a quick check to confirm the upload was successful

AWS Utilities 💫

The **aws_utils.py** helps interact with the AWS services using Python. It helps us manage th eAWS configurations, set up the bucker if it doesn't exist and error handling with logs that are easy to read (including emojis for visual cues).

Here's the **aws_utils.py**:

COPY

```
import os
import boto3
from dotenv import load_dotenv
from boto3.exceptions import Boto3Error

# Load environment variables from .env file
load_dotenv()

def connect_to_aws_s3():
    print("Connecting to AWS S3...")
    try:
        s3_client = boto3.client(
            's3',
            aws_access_key_id=os.getenv("AWS_ACCESS_KEY_ID"),
            aws_secret_access_key=os.getenv("AWS_SECRET_ACCES
            region_name=os.getenv("REGION_NAME")
        )
        print("Connected to AWS S3 successfully.")
        return s3_client
```

```
        except Bot
            raise                                    ONNECTION]: {e}"

    def create_bucket_if_not_exists(s3_client, bucket_name, regic
        print(f"Checking if the bucket {bucket_name} exists...")
        try:
            if bucket_name not in [bucket['Name'] for bucket in s
                s3_client.create_bucket(Bucket=bucket_name, Creat
                print(f"Bucket {bucket_name} created successfully
            else:
                print(f"Bucket {bucket_name} already exists.")
            return bucket_name
        except Boto3Error as e:
            raise Exception(f"❌[ERROR - BUCKET CREATION]: {e}")

    def upload_file_to_s3(s3_client, local_filename, bucket_name,
        print(f"Uploading {local_filename} to the bucket {bucket_
        try:
            csv_folder = "data/"
            full_csv_file_path = f"{csv_folder}{local_filename}"
            s3_path = f"{s3_folder}/{local_filename}" if s3_folde
            s3_client.upload_file(full_csv_file_path, bucket_name
            print(f"File {local_filename} uploaded to {bucket_nam
        except Exception as e:
            raise Exception(f"❌[ERROR - FILE UPLOAD]: {e}")

    def validate_file_in_s3(s3_client, bucket_name, s3_path):
        print(f"Validating the presence of {s3_path} in the bucke
        try:
            s3_client.head_object(Bucket=bucket_name, Key=s3_path
            print(f"Validation successful: {s3_path} exists in {b
            return True
        except Boto3Error as e:
            raise Exception(f"❌[ERROR - VALIDATE FILE]: {e}")
```

Here's the .env file at the root folder for this:

```
# Postgres
...

# AWS
AWS_ACCESS_KEY_ID="xxxxxxxxx"
AWS_SECRET_ACCESS_KEY="xxxxxxxxx"
REGION_NAME="eu-west-2"
S3_BUCKET="premier-league-standings-2024"
S3_FOLDER="football_data"
```

Loading to S3 📩

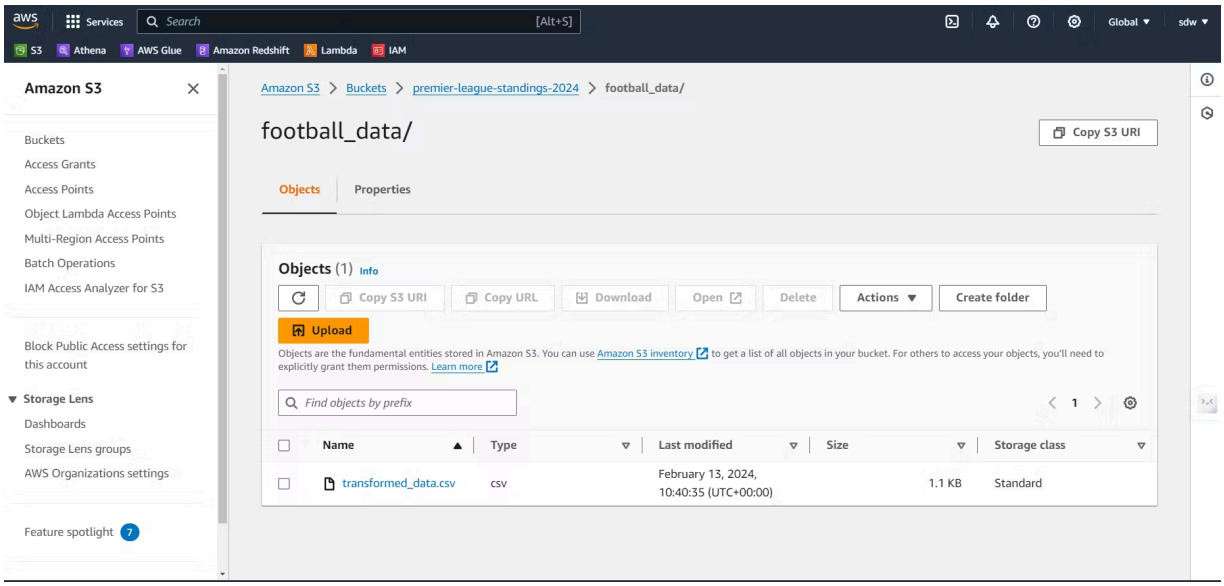
The **s3_uploader.py** file is responsible for uploading the CSV file to the S3 bucket of our choice.

Here's the **s3_uploader.py**:

```
from utils.aws_utils import connect_to_aws_s3, create_bucket_
from utils.db_utils import connect_to_db, fetch_transformed_d
import os

def loading_job():
    print("Starting data transfer process...")
    connection = None
    try:
        connection = connect_to_db()
        df = fetch_transformed_data(connection)
        local_filename = 'transformed_data.csv'
        convert_dataframe_to_csv(df, local_filename)

        s3_client = connect_to_aws_s3()
        bucket_name = create_bucket_if_not_exists(s3_client,
        s3_folder = os.getenv("S3_FOLDER")
```

```
            upload_file_to_s3(s3_client, local_filename, bucket_n

            s3_path = f"{s3_folder}/{local_filename}" if s3_folde
            if validate_file_in_s3(s3_client, bucket_name, s3_pat
                print(f'✅ File {local_filename} successfully upl
            else:
                print(f'❌ File {local_filename} not found in buc

        except Exception as e:
            print(f"❌ An error occurred: {e}")

        finally:
            if connection:
                print("Closing the database connection.")
                connection.close()

    if __name__ == "__main__":
        loading_job()
```

Once we've ran the **s3_uploader.py**, the CSV file is successfully uploaded to the S3 bucket, like so:



ETL Pipeline (Integration zone) 🔄

Instead of running ...ate a single point of entry for all of ...hrough, so something like this:

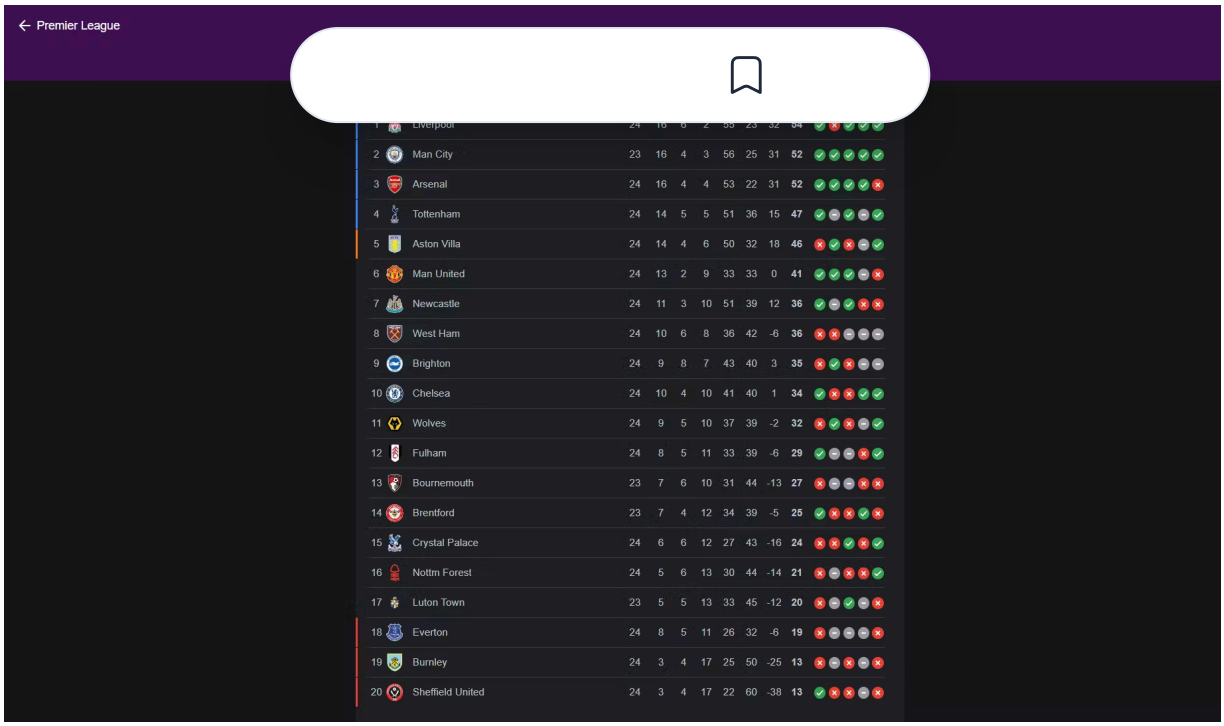
COPY

```
from extraction.main import extraction_job
from transformation.main import transformation_job
from loading.s3_uploader import loading_job

# Create a single point of entry to run the scraping ETL pipe
def run_etl_pipeline():
    extraction_job()
    transformation_job()
    loading_job()

# Execute the pipeline
if __name__=="__main__":
    run_etl_pipeline()
```

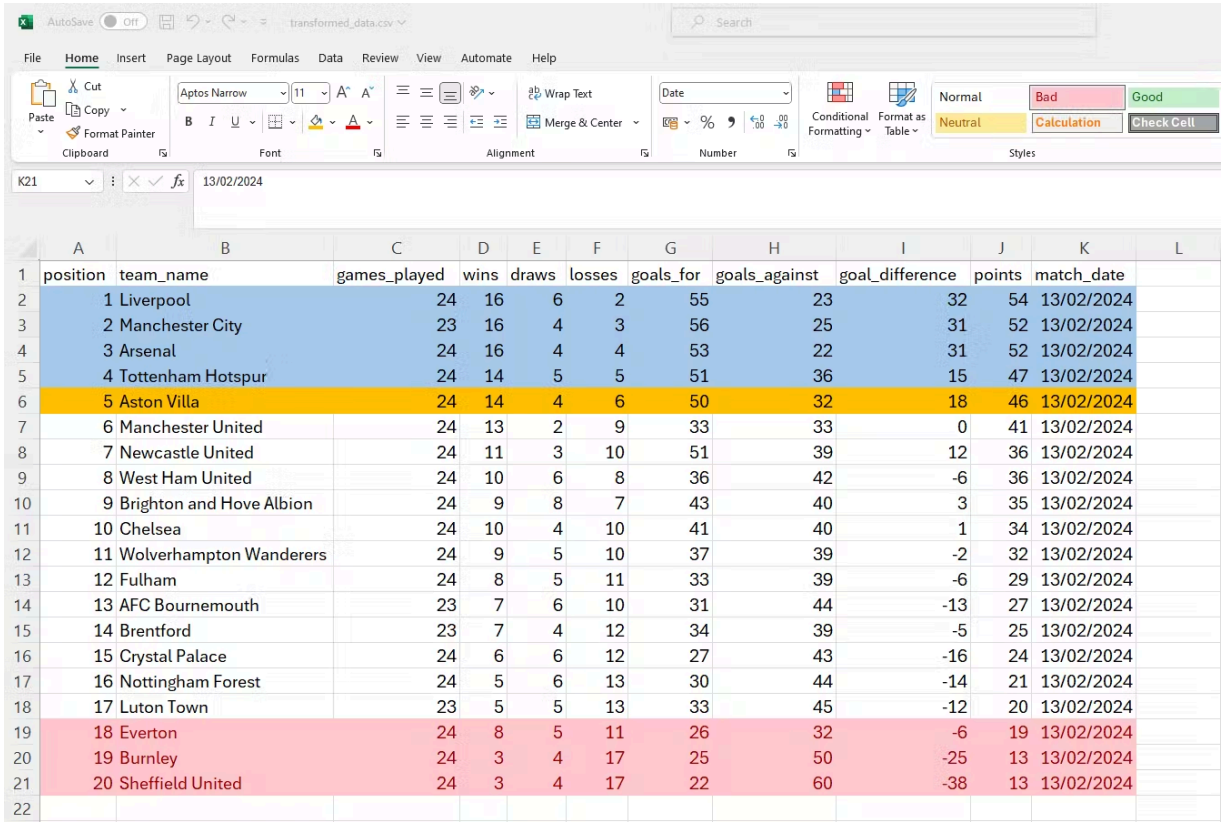
Results 🏆

Now that the scripts have ran, we can compare the actual Premier League table to the outputs we've generated in our CSV file and Postgres table:

Official Premier League Table

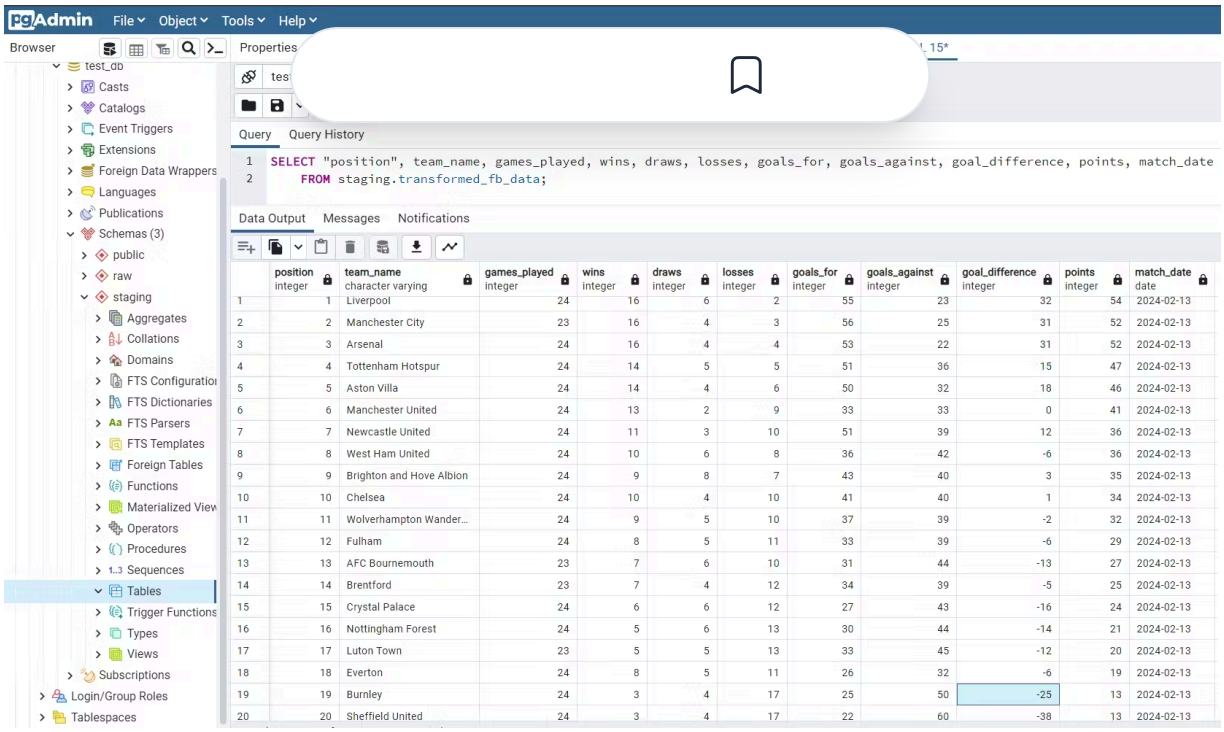


CSV Output (loaded to AWS S3 bucket)



	A	B	C	D	E	F	G	H	I	J	K	L
1	position	team_name	games_played	wins	draws	losses	goals_for	goals_against	goal_difference	points	match_date	
2	1	Liverpool	24	16	6	2	55	23	32	54	13/02/2024	
3	2	Manchester City	23	16	4	3	56	25	31	52	13/02/2024	
4	3	Arsenal	24	16	4	4	53	22	31	52	13/02/2024	
5	4	Tottenham Hotspur	24	14	5	5	51	36	15	47	13/02/2024	
6	5	Aston Villa	24	14	4	6	50	32	18	46	13/02/2024	
7	6	Manchester United	24	13	2	9	33	33	0	41	13/02/2024	
8	7	Newcastle United	24	11	3	10	51	39	12	36	13/02/2024	
9	8	West Ham United	24	10	6	8	36	42	-6	36	13/02/2024	
10	9	Brighton and Hove Albion	24	9	8	7	43	40	3	35	13/02/2024	
11	10	Chelsea	24	10	4	10	41	40	1	34	13/02/2024	
12	11	Wolverhampton Wanderers	24	9	5	10	37	39	-2	32	13/02/2024	
13	12	Fulham	24	8	5	11	33	39	-6	29	13/02/2024	
14	13	AFC Bournemouth	23	7	6	10	31	44	-13	27	13/02/2024	
15	14	Brentford	23	7	4	12	34	39	-5	25	13/02/2024	
16	15	Crystal Palace	24	6	6	12	27	43	-16	24	13/02/2024	
17	16	Nottingham Forest	24	5	6	13	30	44	-14	21	13/02/2024	
18	17	Luton Town	23	5	5	13	33	45	-12	20	13/02/2024	
19	18	Everton	24	8	5	11	26	32	-6	19	13/02/2024	
20	19	Burnley	24	3	4	17	25	50	-25	13	13/02/2024	
21	20	Sheffield United	24	3	4	17	22	60	-38	13	13/02/2024	
22												

Postgres



position integer	team_name character varying	games_played integer	wins integer	draws integer	losses integer	goals_for integer	goals_against integer	goal_difference integer	points integer	match_date date
1	Liverpool	24	16	6	2	55	23	32	54	2024-02-13
2	Manchester City	23	16	4	3	56	25	31	52	2024-02-13
3	Arsenal	24	16	4	4	53	22	31	52	2024-02-13
4	Tottenham Hotspur	24	14	5	5	51	36	15	47	2024-02-13
5	Aston Villa	24	14	4	6	50	32	18	46	2024-02-13
6	Manchester United	24	13	2	9	33	33	0	41	2024-02-13
7	Newcastle United	24	11	3	10	51	39	12	36	2024-02-13
8	West Ham United	24	10	6	8	36	42	-6	36	2024-02-13
9	Brighton and Hove Albion	24	9	8	7	43	40	3	35	2024-02-13
10	Chelsea	24	10	4	10	41	40	1	34	2024-02-13
11	Wolverhampton Wander...	24	9	5	10	37	39	-2	32	2024-02-13
12	Fulham	24	8	5	11	33	39	-6	29	2024-02-13
13	AFC Bournemouth	23	7	6	10	31	44	-13	27	2024-02-13
14	Brentford	23	7	4	12	34	39	-5	25	2024-02-13
15	Crystal Palace	24	6	6	12	27	43	-16	24	2024-02-13
16	Nottingham Forest	24	5	6	13	30	44	-14	21	2024-02-13
17	Luton Town	23	5	5	13	33	45	-12	20	2024-02-13
18	Everton	24	8	5	11	26	32	-6	19	2024-02-13
19	Burnley	24	3	4	17	25	50	-25	13	2024-02-13
20	Sheffield United	24	3	4	17	22	60	-38	13	2024-02-13

Key takeaways 🗝️

- Data contracts are not difficult to incorporate into a data pipeline (especially programmatic ones)
- Using data contracts ensures only valid and accurate data progresses through each stage without any surprises.
- We can increase the quality of data even with Soda's experimental data contracts feature

Resources 🎁

You can find all the code examples used in this article on Github here.

Looking ahead 🔮

The GA version of Soda's data contracts (at the time of this writing) promises to support Docker, which would also support more advanced integrations with Airflow, which means there will be availability to automate the orchestration of each stage easily using DAGs and tasks dependencies.

In a future post I'll [illegible] project using data contracts fo[illegible].

Share your thoughts! 💭

Feel free to share your feedback, questions and comments if you have any!

Subscribe to my newsletter

Read articles from **Stephen's blog** directly inside your inbox. Subscribe to the newsletter, and don't miss out.

Enter your email address SUBSCRIBE

data contracts Scraping data-engineering data-quality Python

Written by

Stephen David-Williams

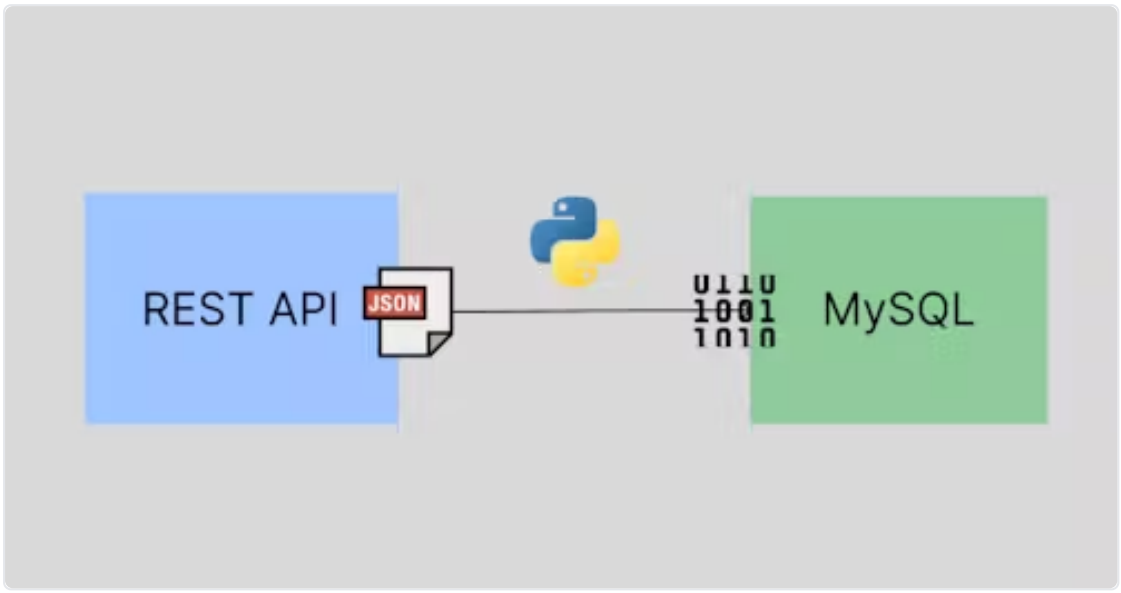
I am a developer with 5+ years of data engineering experience in the financial + professional services sector.

Feel free to drop a message for any questions!

https://medium.com/@sdw-online

Follow

Stephen David-Williams



REST API to MySQL database using Python

Preface💫 In the previous blog post to this, we created a data pipeline that scrapes football data f...

Stephen David-Williams

For 2024...

🤨 What? If your goal is to learn Python and how it applies to data engineering and analytics scenar...

Stephen Da

Set up virtual environment with Anaconda

🌐 What is a virtual environment? A virtual environment is a special directory for holding a specifi...