```
FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\run_local\run_local_scraper_amazon.py
import ssl
import string
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Set, Tuple
from urllib.parse import unquote, urlparse

import nltk
import psycopg2
import psycopg2.extras
import requests
from bs4 import BeautifulSoup
from nltk.corpus import stopwords

try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context

nltk.download("punkt")
nltk.download("stopwords")


def amazon_scraper(query) -> Optional[str]:
    SEARCH_PRODUCTS_URL = "https://www.amazon.com.br/s?k="

    proxy = {"http": "http://Eiprice-cc-
any:DQSXomtV7qri@gw.ntnt.io:5959"}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query.replace(' ', '+')}"
    try:
        response = requests.get(url, headers=headers, proxies=proxy)
    except Exception as error:
        print("Erro ao fazer requisição:", error)
        return None

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all(
        "div", class_="s-main-slot s-result-list s-search-results sg-row"
    )

    products = []

    for product_section in product_list:
        try:
            product = product_section.find(
                "div", class_="a-section aok-relative s-image-square-
aspect"
            )
            descricao = product.find("img", class_="s-image")["alt"]
            link_section = product_section.find(
                "a", class_="a-link-normal s-no-outline"
            )
            link = f"https://www.amazon.com.br{link_section['href']}"
            # img_link = product.find("img", class_="s-image")["src"]
            # sku = link.split("/dp/")[1].split("/")[0]

            products.append({"url": link, "description": descricao})

        except Exception as error:
            print("Erro ao buscar produto:", error)
            continue

    if most_similar_product := find_most_similar(query, products):
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")
        return None


def tokenize_and_lower(text: str) -> Set[str]:
    cleaned_text = remove_punctuation(text)
    tokens = nltk.word_tokenize(cleaned_text.lower())
    return set(tokens)


def remove_punctuation(text: str) -> str:
    """
    Remove punctuation from the given text.

    Args:
        text (str): The input text.

    Returns:
        str: The text without punctuation.
    """
    translator = str.maketrans("", "", string.punctuation)
    return text.translate(translator)


def find_most_similar(
    query: str, products: List[Dict[str, str]]
) -> Optional[Dict[str, str]]:
    query_tokens = tokenize_and_lower(query)
    max_intersection = 0
    most_similar_product = None
    min_intersection = calculate_min_intersection(query)

    for product in products:
        try:
            product_tokens = tokenize_and_lower(product["description"])
            intersection = len(query_tokens.intersection(product_tokens))

            if intersection >= min_intersection and intersection >
max_intersection:
                max_intersection = intersection
                most_similar_product = {
                    "url": product["url"],
                    "description": product["description"],
                }

        except Exception as error:
            print(f"Error processing product: {error}")
            continue

    return most_similar_product


def calculate_min_intersection(query: str, percentage: float = 50.0) ->
int:
    """
    Calculate the minimum intersection value based on the input query and
percentage.

    Args:
        query (str): The input query.
        percentage (float, optional): The percentage to calculate the
minimum intersection. Defaults to 50.0.

    Returns:
        int: The minimum intersection value.
    """
    tokens = nltk.word_tokenize(query)
    return int(len(tokens) * (percentage / 100))


def create_connection():
    return psycopg2.connect(
        dbname="eiprice-hml",
        user="postgres",
        password="GNrx+6bh)So<mU",
        host="35.184.21.249",
    )


def fetch_schemas(connection) -> List[Tuple[str]]:
    cursor = connection.cursor(cursor_factory=psycopg2.extras.DictCursor)
    cursor.execute(
        "SELECT schema_name FROM information_schema.schemata WHERE
schema_name NOT IN ('public', 'information_schema', 'pg_catalog',
'pg_toast', 'sku');"
    )
    return cursor.fetchall()


def fetch_products(connection, schema: str, id_concorrente) ->
List[Tuple]:
    cursor = connection.cursor(cursor_factory=psycopg2.extras.DictCursor)
    cursor.execute(
        f"SELECT * FROM {schema}.produto_novo WHERE id_concorrente =
{id_concorrente};"
    )
    return cursor.fetchall()


def insert_sku(connection, schema: str, data: Tuple):
    cursor = connection.cursor(cursor_factory=psycopg2.extras.DictCursor)
    cursor.execute(
        f"INSERT INTO {schema}.sku (ean, cod_ref, id_loja, slug) VALUES
(%s, %s, %s, %s)",
        data,
    )
    connection.commit()


def insert_product_not_found(produto: dict, schema: str):
    connection = create_connection()
    cursor = connection.cursor(cursor_factory=psycopg2.extras.DictCursor)
    cursor.execute(
        f"INSERT INTO {schema}.produto_nao_encontrado (ean, sku, produto,
departamento, categoria, marca, id_concorrente, nome_concorrente) VALUES
(%s, %s, %s, %s, %s, %s, %s, %s)",
        (
            produto["ean"],
            produto["sku"],
            produto["produto"],
            produto["departamento"],
            produto["categoria"],
            produto["marca"],
            produto["id_concorrente"],
            produto["nome_concorrente"],
        ),
    )
    connection.commit()
    close_connection(connection)


def delete_sku(schema: str, id: str):
    connection = create_connection()
    cursor = connection.cursor(cursor_factory=psycopg2.extras.DictCursor)
    cursor.execute(f"DELETE FROM {schema}.produto_novo WHERE id = %s",
(id,))
```

```
    connection.commit()


def close_connection(connection):
    connection.close()


def extract_product_sku(url: str) -> str:
    parsed_url = urlparse(unquote(url))
    path_components = parsed_url.path.strip("/").split("/")
    if "dp" in path_components:
        dp_index = path_components.index("dp")
        if dp_index + 1 < len(path_components):
            return path_components[dp_index + 1]


def executar_consulta():
    connection = create_connection()

    try:
        schemas = fetch_schemas(connection)
        print(f"Schemas: {schemas}")

        for schema in schemas:
            produtos = fetch_products(connection, schema[0], 156)
            if not produtos:
                print("Nenhum resultado encontrado.")
                continue

            for produto in produtos:
                print("=====================================")
                print(produto[3])
                slug = amazon_scraper(produto[3])
                print(slug)
                print("=====================================")
                if slug is None:
                    insert_product_not_found(produto, schema[0])
                    delete_sku(schema[0], produto[0])
                else:
                    try:
                        sku = extract_product_sku(slug)
                        insert_sku(
                            connection,
                            schema[0],
                            (
                                produto[1],
                                sku,
                                produto[7],
                                slug,
                            ),
                        )
                        delete_sku(schema[0], produto[0])
                        print("SKU inserido com sucesso:", sku)
                        print("=====================================")
                    except Exception as error:
                        print("Erro ao inserir sku:", error)
                        print(error)
                        print("=====================================")
                        continue

    except Exception as error:
        print("Erro ao executar consulta:", error)

    finally:
        close_connection(connection)


executar_consulta()


FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\scrapers_template.py
import string
from datetime import datetime, timedelta
from typing import Dict, List, Optional

import nltk
import requests
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from bs4 import BeautifulSoup
from nltk.corpus import stopwords
from unidecode import unidecode

nltk.download("punkt")
nltk.download("stopwords")

from airflow.models import Variable

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 2),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "scraper_X",
    default_args=default_args,
    description="Classificador de produtos X",
    schedule_interval="0 21 * * *",
)
```

```
def remove_punctuation(text: str) -> str:
    """
    Remove punctuation from the given text.

    Args:
        text (str): The input text.

    Returns:
        str: The text without punctuation.
    """
    translator = str.maketrans("", "", string.punctuation)
    return text.translate(translator)


def tokenize_and_lower(text: str) -> Set[str]:
    """
    Tokenize the input text and convert it to lowercase.

    Args:
        text (str): The input text.

    Returns:
        Set[str]: A set of unique lowercase tokens.
    """
    cleaned_text = remove_punctuation(text)
    tokens = nltk.word_tokenize(cleaned_text.lower())
    return set(tokens)


def calculate_min_intersection(query: str, percentage: float = 50.0) ->
int:
    """
    Calculate the minimum intersection value based on the input query and
percentage.

    Args:
        query (str): The input query.
        percentage (float, optional): The percentage to calculate the
minimum intersection. Defaults to 50.0.

    Returns:
        int: The minimum intersection value.
    """
    tokens = nltk.word_tokenize(query)
    return int(len(tokens) * (percentage / 100))


def find_most_similar(
    query: str, products: List[Dict[str, str]]
) -> Optional[Dict[str, str]]:
    query_tokens = tokenize_and_lower(query)
    max_intersection = 0
    most_similar_product = None
    min_intersection = calculate_min_intersection(query)

    for product in products:
        try:
            product_tokens = tokenize_and_lower(product["description"])
            intersection = len(query_tokens.intersection(product_tokens))

            if intersection >= min_intersection and intersection >
max_intersection:
                max_intersection = intersection
                most_similar_product = {
                    "url": product["url"],
                    "description": product["description"],
                }

        except Exception as error:
            print(f"Error processing product: {error}")
            continue

    return most_similar_product


def amazon_scraper(query: str) -> Optional[str]:
    query_tokens = tokenize_and_lower(query)
    SEARCH_PRODUCTS_URL = "https://SUAURL/SEUSPARAMENTROSDEBUSCA"

    # proxy = {"http": Variable.get('PROXY')} # usar ao subir para
airflow
    proxy = {
        "http": "http://Eiprice-cc-any:DQSXomtV7qri@gw.ntnt.io:5959"
    }  # usar local para testar

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query.replace(' ', '+')}"
    try:
        response = requests.get(url, headers=headers, proxies=proxy)
    except Exception as error:
        print("Erro ao fazer requisição:", error)
        return None

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all(
        "div", class_="SUA CLASSE ONDE ESTA A GRADE DE PRODUTOS"
    )

    products = []

    for product_section in product_list:
        try:
            product = product_section.find(
                "div", class_="CLASSE DE CADA PRODUTO DA LISTA"
            )
            descricao = product.find("img", class_="CLASSE DA
IMAGEM")["alt"]
            link_section = product_section.find("a", class_="CLASSE DE
SEU LINK")
            link = f"https://www.amazon.com.br{link_section['href']}"
            img_link = product.find("img", class_="CLASSE DO LINK DA
IMAGEM")["src"]
            sku = link.split("/dp/")[1].split("/")[
                0
            ]  # GERALMENTE ENCONTRAMOS SKU NAS URLS, MAS EM ALGUNS SITES
PRECISAMOS ANALISAR CODIGO FONTE/CLASSES...

            products.append(
                {
                    "url": link,
                    "description": descricao,
                    "sku": sku,
                    "img_link": img_link,
                }
            )

        except Exception as error:
            print("Erro ao buscar produto:", error)
            continue

    if most_similar_product := find_most_similar(query_tokens, products):
        print("Produto mais similar encontrado:")
        print(most_similar_product["description"],
most_similar_product["url"])
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")
        return None


def executar_consulta():
    # connection_hook =
PostgresHook(postgres_conn_id=Variable.get('POSTGRES_DB')) # usar ao
subir para airflow
    connection_hook = PostgresHook(postgres_conn_id="eiprice-dev")  #
usar local
    connection = connection_hook.get_conn()
    cursor = connection.cursor()

    try:
        cursor.execute(
            "SELECT ean, sku, id_loja, atributo, status, descricao FROM
kabum.produto_carga;"
        )
        resultados = cursor.fetchall()
        for resultado in resultados:
            ean, sku, id_loja, atributo, status, descricao = resultado
            print("=====================================")
            print(resultado[5])
            slug = amazon_scraper(unidecode(resultado[5]))
            print(slug)
            print("=====================================")

            try:
                cursor.execute(
                    "INSERT INTO kabum.sku (ean, cod_ref, id_loja,
atributo, status, slug) VALUES (%s, %s, %s, %s, %s, %s)",
                    (ean, sku, id_loja, atributo, status, slug),
                )
            except Exception as error:
                print("Erro ao inserir sku:", sku)
                print(error)
                print("=====================================")
                continue

    except Exception as error:
        print("Erro ao executar consulta:", error)

    finally:
        connection.commit()
        connection.close()


run_query = PythonOperator(
    task_id="run_query",
    python_callable=executar_consulta,
    dag=dag,
)

run_query


def run_X_scraper_local(query: str) -> None:
    """
    Run the X scraper locally with the given query.

    Args:
        query (str): The input query.
    """
    print("Running X scraper locally...")
    print("Query:", query)

    SEARCH_PRODUCTS_URL = "https://www.X.com.br/"
    PROXY = "http://Eiprice-cc-any:DQSXomtV7qri@gw.ntnt.io:5959"
    HEADERS = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query.replace(' ', '+')}"
    response = requests.get(url, headers=HEADERS, proxies={PROXY})

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all(
        "div", class_="s-main-slot s-result-list s-search-results sg-row"
    )

    products = []

    for product_section in product_list:
        try:
            product = product_section.find(
                "div", class_="a-section aok-relative s-image-square-
aspect"
            )
            descricao = product.find("img", class_="s-image")["alt"]
            link_section = product_section.find(
                "a", class_="a-link-normal s-no-outline"
            )
            link = f"{SEARCH_PRODUCTS_URL}{link_section['href']}"

            products.append({"url": link, "description": descricao})

        except Exception as error:
            print("Erro ao buscar produto:", error)
            continue

    if most_similar_product := find_most_similar(query, products):
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
    else:
        print("Nenhum produto similar encontrado.")


if __name__ == "__main__":
    sample_query = "COMPUTADOR GAMER 128GB RAM SSD HD 1TB NOVO"
    run_X_scraper_local(sample_query)


FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\scraper_amazon.py
import ssl
import string
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Set, Tuple

import nltk
import requests
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from bs4 import BeautifulSoup
from nltk.corpus import stopwords

from urllib.parse import urlparse, unquote


nltk.download("punkt")
nltk.download("stopwords")

from airflow.models import Variable

try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 2),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "scraper_amazon",
    default_args=default_args,
    description="Classificador de produtos da Amazon",
    schedule_interval="0 21 * * *",
)


def remove_punctuation(text: str) -> str:
    """
    Remove punctuation from the given text.

    Args:
        text (str): The input text.

    Returns:
        str: The text without punctuation.
    """
    translator = str.maketrans("", "", string.punctuation)
    return text.translate(translator)



def tokenize_and_lower(text: str) -> Set[str]:
    """
    Tokenize the input text and convert it to lowercase.

    Args:
        text (str): The input text.

    Returns:
        Set[str]: A set of unique lowercase tokens.
    """
    cleaned_text = remove_punctuation(text)
    tokens = nltk.word_tokenize(cleaned_text.lower())
    return set(tokens)


def calculate_min_intersection(query: str, percentage: float = 50.0) ->
int:
    """
    Calculate the minimum intersection value based on the input query and
percentage.

    Args:
        query (str): The input query.
        percentage (float, optional): The percentage to calculate the
minimum intersection. Defaults to 50.0.

    Returns:
        int: The minimum intersection value.
    """
    tokens = nltk.word_tokenize(query)
    return int(len(tokens) * (percentage / 100))


def find_most_similar(
    query: str, products: List[Dict[str, str]]
) -> Optional[Dict[str, str]]:
    query_tokens = tokenize_and_lower(query)
    max_intersection = 0
    most_similar_product = None
    min_intersection = calculate_min_intersection(query)

    for product in products:
        try:
            product_tokens = tokenize_and_lower(product["description"])
            intersection = len(query_tokens.intersection(product_tokens))

            if intersection >= min_intersection and intersection >
max_intersection:
                max_intersection = intersection
                most_similar_product = {
                    "url": product["url"],
                    "description": product["description"],
                }

        except Exception as error:
            print(f"Error processing product: {error}")
            continue

    return most_similar_product


def amazon_scraper(query: str) -> Optional[str]:
    SEARCH_PRODUCTS_URL = "https://www.amazon.com.br/s?k="

    proxy = {"http": Variable.get("PROXY")}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query.replace(' ', '+')}"
    try:
        response = requests.get(url, headers=headers, proxies=proxy)
    except Exception as error:
        print("Erro ao fazer requisição:", error)
        return None

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all(
        "div", class_="s-main-slot s-result-list s-search-results sg-row"
    )

    products = []

    for product_section in product_list:
        try:
            product = product_section.find(
                "div", class_="a-section aok-relative s-image-square-
aspect"
            )
            descricao = product.find("img", class_="s-image")["alt"]
            link_section = product_section.find(
                "a", class_="a-link-normal s-no-outline"
            )
            link = f"https://www.amazon.com.br{link_section['href']}"
            # img_link = product.find("img", class_="s-image")["src"]
            # sku = link.split("/dp/")[1].split("/")[0]

            products.append({"url": link, "description": descricao})

        except Exception as error:
            print("Erro ao buscar produto:", error)
            continue

    if most_similar_product := find_most_similar(query, products):
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")
        return None


####
def get_connection():
    """
    Get the PostgresHook connection object.

    Returns:
        Connection object.
    """
    return
PostgresHook(postgres_conn_id=Variable.get("POSTGRES_DB")).get_conn()


def fetch_schemas() -> List[str]:
    connection = get_connection()
    cursor = connection.cursor()
    cursor.execute(
        "SELECT schema_name FROM information_schema.schemata WHERE
schema_name NOT IN ('public', 'information_schema', 'pg_catalog',
'pg_toast', 'sku');"
    )
    # Transforma a lista de tuplas em uma lista de strings
    return [result[0] for result in cursor.fetchall()]



def fetch_products(schema: str, id_concorrente) -> List[Tuple]:
    connection = get_connection()
    cursor = connection.cursor()
    cursor.execute(
        f"SELECT * FROM {schema}.produto_novo WHERE id_concorrente =
{id_concorrente};"
    )
    return cursor.fetchall()


def insert_sku(schema: str, data: Tuple):
    connection = get_connection()
    cursor = connection.cursor()
    cursor.execute(
        f"INSERT INTO {schema}.sku (ean, cod_ref, id_loja, slug) VALUES
(%s, %s, %s, %s)",
        data,
    )
    connection.commit()


def insert_product_not_found(produto: dict, schema: str):
    connection = get_connection()
    cursor = connection.cursor()
    cursor.execute(
        f"INSERT INTO {schema}.produto_nao_encontrado (ean, sku, produto,
departamento, categoria, marca, id_concorrente, nome_concorrente) VALUES
(%s, %s, %s, %s, %s, %s, %s, %s)",
        (
            produto["ean"],
            produto["sku"],
            produto["produto"],
            produto["departamento"],
            produto["categoria"],
            produto["marca"],
            produto["id_concorrente"],
            produto["nome_concorrente"],
        ),
    )
    connection.commit()
    close_connection(connection)


def delete_sku(schema: str, id: str):
    connection = get_connection()
    cursor = connection.cursor()
    cursor.execute(f"DELETE FROM {schema}.produto_novo WHERE id = %s",
(id,))
    connection.commit()


def close_connection():
    connection = get_connection()
    connection.close()


def extract_product_sku(url: str) -> str:
    parsed_url = urlparse(unquote(url))
    path_components = parsed_url.path.strip("/").split("/")
    if "dp" in path_components:
        dp_index = path_components.index("dp")
        if dp_index + 1 < len(path_components):
            return path_components[dp_index + 1]


def executar_consulta():
    try:
        schemas = fetch_schemas()
        print(f"Schemas: {schemas}")

        for schema_name in schemas:
            try:
                produtos = fetch_products(schema_name, 156)
                if not produtos:
                    print("Nenhum resultado encontrado.")
                    continue
                for produto in produtos:
                    print("======================================")
                    print(produto[3])
                    slug = amazon_scraper(produto[3])
                    print(slug)
                    print("======================================")
                    if slug is None:
                        insert_product_not_found(produto, schema_name)
                        delete_sku(schema_name, produto[0])
                    try:
                        sku = extract_product_sku(slug)
                        insert_sku(
                            schema_name,
                            (
                                produto[1],
                                sku,
                                produto[7],
                                slug,
                            ),
                        )
                        delete_sku(schema_name, produto[0])
                        print("SKU inserido com sucesso:", sku)
                        print("======================================")
                    except Exception as error:
                        print("Erro ao inserir sku:", error)
                        print(error)
                        print("======================================")
            except Exception as schema_error:
                print(f"Erro ao processar o schema {schema_name}:",
schema_error)
                print("======================================")
    except Exception as error:
        print("Erro ao executar consulta:", error)


run_query = PythonOperator(
    task_id="run_query",
    python_callable=executar_consulta,
    dag=dag,
)

run_query


def run_amazon_scraper_local(query: str) -> None:
    """
    Run the Amazon scraper locally with the given query.

    Args:
        query (str): The input query.
    """
    print("Running Amazon scraper locally...")
    print("Query:", query)
    SEARCH_PRODUCTS_URL = "https://www.amazon.com.br/s?k="
    PROXY = "http://Eiprice-cc-any:DQSXomtV7qri@gw.ntnt.io:5959"
    HEADERS = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query.replace(' ', '+')}"
    response = requests.get(url, headers=HEADERS, proxies={PROXY})

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all(
        "div", class_="s-main-slot s-result-list s-search-results sg-row"
    )

    products = []

    for product_section in product_list:
        try:
            product = product_section.find(
                "div", class_="a-section aok-relative s-image-square-
aspect"
            )
            descricao = product.find("img", class_="s-image")["alt"]
            link_section = product_section.find(
                "a", class_="a-link-normal s-no-outline"
            )
            link = f"https://www.amazon.com.br{link_section['href']}"

            products.append({"url": link, "description": descricao})

        except Exception as error:
            print("Erro ao buscar produto:", error)
            continue

    if most_similar_product := find_most_similar(query, products):
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
    else:
        print("Nenhum produto similar encontrado.")


if __name__ == "__main__":
    sample_query = "COMPUTADOR GAMER 128GB RAM SSD HD 1TB NOVO"
    run_amazon_scraper_local(sample_query)


FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\scraper_americanas.py
from datetime import datetime, timedelta
```

```
import nltk
import requests
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from bs4 import BeautifulSoup
from unidecode import unidecode

nltk.download("punkt")

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 2),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "scraper_americanas",
    default_args=default_args,
    description="Classificador de produtos Americanas",
    schedule_interval="0 21 * * *",
)

nltk.download("punkt")


def americanas_scraper(query):
    query_tokens = set(nltk.word_tokenize(query.lower()))
    max_intersection = 0
    most_similar_product = None
    SEARCH_PRODUCTS_URL = "https://www.americanas.com.br/busca/"
    MIN_INTERSECTION = 2

    proxy = {"http": "http://Eiprice-cc-
any:DQSXomtV7qri@gw.ntnt.io:5959"}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query.replace(' ', '-')}"
    response = requests.get(url, headers=headers, proxies=proxy)

    base_url = "https://www.americanas.com.br"

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all("div", class_="grid__StyledGrid-sc-
1man2hx-0")

    for produto in product_list:
        try:
            for a in soup.find_all("a", class_="inStockCard__Link-sc-
1ngt5zo-1 JOEpk"):
                link = a["href"].split(" ")[0]
                url = base_url + link
                descricao = a.find(
                    "h3", class_="product-name__Name-sc-1shovj0-0 gUjFDF"
                ).text

                product_tokens =
set(nltk.word_tokenize(descricao.lower()))
                intersection =
len(query_tokens.intersection(product_tokens))

                if intersection > max_intersection:
                    max_intersection = intersection
                    most_similar_product = {"url": url, "alt_text":
descricao}
        except Exception as error:
            print("=====================================")
            print(f"Erro ao buscar produto: {produto}")
            print(error)
            print("=====================================")
            continue

    if most_similar_product:
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")


def executar_consulta():
    connection_hook = PostgresHook(postgres_conn_id="eiprice-dev")
    connection = connection_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(
        "SELECT ean, sku, id_loja, atributo, status, descricao FROM
kabum.produto_carga;"
    )
    resultados = cursor.fetchall()
    for resultado in resultados:
        ean, sku, id_loja, atributo, status, descricao = resultado
        print("=====================================")
        print(resultado[5])
        slug = americanas_scraper(unidecode(resultado[5]))
        print(slug)
        print("=====================================")
        try:
            cursor.execute(
                "INSERT INTO kabum.sku (ean, cod_ref, id_loja, atributo,
status, slug) VALUES (%s, %s, %s, %s, %s, %s)",
                (ean, sku, id_loja, atributo, status, slug),
            )
        except Exception as error:
            print("======================================")
            print(f"Erro ao inserir sku: {sku}")
            print(error)
            print("======================================")
            continue
    connection.commit()
    connection.close()


run_query = PythonOperator(
    task_id="run_query",
    python_callable=executar_consulta,
    dag=dag,
)

run_query


FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\scraper_carrefour.py
import ssl
from datetime import datetime, timedelta
from urllib.parse import quote

import nltk
import requests
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from bs4 import BeautifulSoup
from unidecode import unidecode

nltk.download("punkt")

try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 2),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "scraper_carrefour",
    default_args=default_args,
    description="Classificador de produtos do Carrefour",
    schedule_interval="0 21 * * *",
)


def carrefour_scraper(query):
    query_encoded = quote(query, safe="")
    query_tokens = set(nltk.word_tokenize(query.lower()))
    max_intersection = 0
    most_similar_product = None
    SEARCH_PRODUCTS_URL = "https://www.carrefour.com.br/busca/"
    MIN_SIMILARITY = 0.6
    MIN_INTERSECTION = 2

    proxy = {"http": "http://Eiprice-cc-
any:DQSXomtV7qri@gw.ntnt.io:5959"}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query_encoded}"
    response = requests.get(url, headers=headers, proxies=proxy)

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all(
        "article",
        class_="vtex-product-summary-2-x-element pointer pt3 pb4 flex
flex-column h-100",
    )
    href_list = soup.find_all(
        "a", class_="vtex-product-summary-2-x-clearLink h-100 flex flex-
column"
    )

    for product_index, product in enumerate(product_list):
        descricao = product.find(
            "img",
            class_="vtex-product-summary-2-x-imageNormal vtex-product-
summary-2-x-image",
        )["alt"]
        product_tokens = set(nltk.word_tokenize(descricao.lower()))
        intersection = len(query_tokens.intersection(product_tokens))
        if intersection > max_intersection:
            max_intersection = intersection
            most_similar_product = {
                "url": href_list[product_index]["href"],
                "alt_text": descricao,
            }

    if most_similar_product:
        print("Produto mais similar encontrado:")
        print(most_similar_product)
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")


def executar_consulta():
    connection_hook = PostgresHook(postgres_conn_id="eiprice-dev")
    connection = connection_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(
        "SELECT ean, sku, id_loja, atributo, status, descricao FROM
kabum.produto_carga;"
    )
    resultados = cursor.fetchall()
    for resultado in resultados:
        ean, sku, id_loja, atributo, status, descricao = resultado
        print("======================================")
        print(resultado[5])
        slug = carrefour_scraper(unidecode(resultado[5]))
        print(slug)
        print("======================================")
        try:
            cursor.execute(
                "INSERT INTO kabum.sku (ean, cod_ref, id_loja, atributo,
status, slug) VALUES (%s, %s, %s, %s, %s, %s)",
                (ean, sku, id_loja, atributo, status, slug),
            )
        except Exception as error:
            print("======================================")
            print(f"Erro ao inserir sku: {sku}")
            print(error)
            print("======================================")
            continue
    connection.commit()
    connection.close()


run_query = PythonOperator(
    task_id="run_query",
    python_callable=executar_consulta,
    dag=dag,
)

run_query
```

```
FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\scraper_casasbahia.py
from datetime import datetime, timedelta

import nltk
import requests
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from bs4 import BeautifulSoup
from unidecode import unidecode

nltk.download("punkt")

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 2),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "scraper_casasbahia",
    default_args=default_args,
    description="Classificador de produtos da Casas Bahia",
    schedule_interval="0 21 * * *",
)

BASE_URL = "https://www.casasbahia.com.br"
MIN_INTERSECTION = 2


def casas_bahia_scraper(query):
    query_tokens = set(nltk.word_tokenize(query.lower()))
    max_intersection = 0
    most_similar_product = None

    proxy = {"http": "http://Eiprice-cc-
any:DQSXomtV7qri@gw.ntnt.io:5959"}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{BASE_URL}/{query.replace(' ', '-')}/b"
    response = requests.get(url, headers=headers, proxies=proxy)

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all("div", class_="sc-18eb4054-0 dPlWZd")

    for product in product_list:
        try:
            descricao = product.find("img", class_="sc-d2913f46-0
htwjrw")["alt"]
            link = product.find("a", class_="sc-2b5b888e-1
cflebu")["href"]

            product_tokens = set(nltk.word_tokenize(descricao.lower()))
            intersection = len(query_tokens.intersection(product_tokens))

            if intersection > max_intersection:
                max_intersection = intersection
                most_similar_product = {"url": link, "alt_text":
descricao}
        except Exception as error:
            print("=====================================")
            print(f"Erro ao buscar produto: {product}")
            print(error)
            print("Nenhum produto similar encontrado.")
            print("=====================================")
            continue

    if most_similar_product:
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")


def executar_consulta():
    connection_hook = PostgresHook(postgres_conn_id="eiprice-dev")
    connection = connection_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(
        "SELECT ean, sku, id_loja, atributo, status, descricao FROM
kabum.produto_carga;"
    )
    resultados = cursor.fetchall()
    for resultado in resultados:
        ean, sku, id_loja, atributo, status, descricao = resultado
        print("=====================================")
        print(resultado[5])
        slug = casas_bahia_scraper(unidecode(resultado[5]))
        print(slug)
        print("=====================================")
        try:
            cursor.execute(
                "INSERT INTO kabum.sku (ean, cod_ref, id_loja, atributo, 
status, slug) VALUES (%s, %s, %s, %s, %s, %s)",
                (ean, sku, id_loja, atributo, status, slug),
            )
        except Exception as error:
            print("=======================================")
            print(f"Erro ao inserir sku: {sku}")
            print(error)
            print("=======================================")
            continue
    connection.commit()
    connection.close()


run_query = PythonOperator(
    task_id="run_query",
    python_callable=executar_consulta,
    dag=dag,
)

run_query


FILEPATH: C:\Users\andwe\Downloads\classification-scrapers\scraper_cec.py
from datetime import datetime, timedelta

import nltk
import requests
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from bs4 import BeautifulSoup
from unidecode import unidecode

nltk.download("punkt")

# DADOS DE QUANDO E COMO AIRFLOW VAI SER EXECUTADO
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 2),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "scraper_cec",
    default_args=default_args,
    description="Classificador de produtos do CEC",
    schedule_interval="0 21 * * *",
)
```

```
# SCRAPER
def cec_scraper(query):
    query_tokens = set(nltk.word_tokenize(query.lower()))
    max_intersection = 0
    most_similar_product = None
    SEARCH_PRODUCTS_URL = "https://www.cec.com.br"
    MIN_SIMILARITY = 0.6
    MIN_INTERSECTION = 2

    proxy = {"http": "http://Eiprice-cc-
any:onQoK56mACxA@gw.ntnt.io:5959"}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{'/busca?q='}{query.replace(' ',
'%20')}{'&ranking=2&topsearch=1'}"

    response = requests.get(url, headers=headers, proxies=proxy)

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all(
        "div",
        class_="products",
    )
    for product in product_list:
        try:
            link =
f"{SEARCH_PRODUCTS_URL}{product.find('a',class_='photo',)['href']}"
            descricao = product.find(
                "a",
                class_="photo",
            )["title"]
            product_tokens = set(nltk.word_tokenize(descricao.lower()))
            intersection = len(query_tokens.intersection(product_tokens))

            if intersection > max_intersection:
                max_intersection = intersection
                most_similar_product = {"url": link, "alt_text":
descricao}
        except Exception as error:
            print("====================================")
            print(f"Erro ao buscar produto: {product}")
            print(error)
            print("====================================")
            continue

    if most_similar_product:
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")


def executar_consulta():
    connection_hook = PostgresHook(postgres_conn_id="eiprice-dev")
    connection = connection_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(
        "SELECT ean, sku, id_loja, atributo, status, descricao FROM
kabum.produto_carga;"
    )
    resultados = cursor.fetchall()
    for resultado in resultados:
        ean, sku, id_loja, atributo, status, descricao = resultado
        print("=======================================")
        print(resultado[5])
        slug = cec_scraper(unidecode(resultado[5]))
        print(slug)
        print("=======================================")
        try:
            cursor.execute(
                "INSERT INTO kabum.sku (ean, cod_ref, id_loja, atributo,
status, slug) VALUES (%s, %s, %s, %s, %s, %s)",
                (ean, sku, id_loja, atributo, status, slug),
            )
        except Exception as error:
            print("=======================================")
            print(f"Erro ao inserir sku: {sku}")
            print(error)
            print("=======================================")
            continue
    connection.commit()
    connection.close()


run_query = PythonOperator(
    task_id="run_query",
    python_callable=executar_consulta,
    dag=dag,
)

run_query


FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\scraper_leroy_merlin.py
from datetime import datetime, timedelta

import nltk
import requests
```

```
from airflow import DAG
from bs4 import BeautifulSoup

# from airflow.operators.python_operator import PythonOperator
# from airflow.providers.postgres.hooks.postgres import PostgresHook
from unidecode import unidecode

nltk.download("punkt")

# DADOS DE QUANDO E COMO AIRFLOW VAI SER EXECUTADO
# default_args = {
#     "owner": "airflow",
#     "depends_on_past": False,
#     "start_date": datetime(2023, 5, 2),
#     "email": ["airflow@example.com"],
#     "email_on_failure": False,
#     "email_on_retry": False,
#     "retries": 1,
#     "retry_delay": timedelta(minutes=5),
# }

# dag = DAG(
#     "scraper_cec",
#     default_args=default_args,
#     description="Classificador de produtos do CEC",
#     schedule_interval="0 21 * * *",
# )


# SCRAPER
def leroy_scraper(query):
    query_tokens = set(nltk.word_tokenize(query.lower()))
    max_intersection = 0
    most_similar_product = None
    SEARCH_PRODUCTS_URL = "https://www.leroymerlin.com.br/search?term="
    MIN_SIMILARITY = 0.6
    MIN_INTERSECTION = 2

    proxy = {"http": "http://Eiprice-cc-
any:onQoK56mACxA@gw.ntnt.io:5959"}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{'/busca?q='}{query.replace(' ',
'%20')}{'&searchType=default'}"
    print(url)
    response = requests.get(url, headers=headers, proxies=proxy)
    print(response.status_code)
    soup = BeautifulSoup(response.content, "html.parser")
    product = soup.find_all("script", {"type": "application/ld+json"})
    print(product)

    # product_list = soup.find_all(
    #     "div",
    #     class_="products",
    # )
    # for product in product_list:
    #     try:
    #         link =
f"{SEARCH_PRODUCTS_URL}{product.find('a',class_='photo',)['href']}"
    #         descricao = product.find(
    #             "a",
    #             class_="photo",
    #         )["title"]
    #         sku = link.split("produto=")[1]
    #         product_tokens = set(nltk.word_tokenize(descricao.lower()))
    #         intersection =
len(query_tokens.intersection(product_tokens))

    #         if intersection > max_intersection:
    #             max_intersection = intersection
    #             most_similar_product = {"url": link, "alt_text":
descricao}
    #     except Exception as error:
    #         print("====================================")
    #         print(f"Erro ao buscar produto: {product}")
    #         print(error)
    #         print("====================================")
    #         continue

    # if most_similar_product:
    #     print("Produto mais similar encontrado:")
    #     print(most_similar_product["url"])
    #     return most_similar_product["url"]
    # else:
    #     print("Nenhum produto similar encontrado.")


def executar_consulta():
    connection_hook = PostgresHook(postgres_conn_id="eiprice-dev")
    connection = connection_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(
        "SELECT ean, sku, id_loja, atributo, status, descricao FROM
kabum.produto_carga;"
    )
    resultados = cursor.fetchall()
    for resultado in resultados:
        ean, sku, id_loja, atributo, status, descricao = resultado
        print("====================================")
        print(resultado[5])
        slug = cec_scraper(unidecode(resultado[5]))
        print(slug)
        print("=====================================")
        try:
            cursor.execute(
                "INSERT INTO kabum.sku (ean, cod_ref, id_loja, atributo,
status, slug) VALUES (%s, %s, %s, %s, %s, %s)",
                (ean, sku, id_loja, atributo, status, slug),
            )
        except Exception as error:
            print("=====================================")
            print(f"Erro ao inserir sku: {sku}")
            print(error)
            print("=====================================")
            continue
    connection.commit()
    connection.close()


leroy_scraper("PLACA DE VIDEO")
# run_query = PythonOperator(
#     task_id="run_query",
#     python_callable=executar_consulta,
#     dag=dag,
# )

# run_query


FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\scraper_magalu.py.py
from datetime import datetime, timedelta

import nltk
import requests
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from bs4 import BeautifulSoup
from unidecode import unidecode

nltk.download("punkt")

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 2),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "scraper_magalu",
    default_args=default_args,
    description="Classificador de produtos Magalu",
    schedule_interval="0 21 * * *",
)


def magalu_scraper(query):
    query_tokens = set(nltk.word_tokenize(query.lower()))
    max_intersection = 0
    most_similar_product = None
    SEARCH_PRODUCTS_URL = "https://www.magazineluiza.com.br/busca/"

    proxy = {"http": "http://Eiprice-cc-
any:DQSXomtV7qri@gw.ntnt.io:5959"}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query.replace(' ', '+')}"

    try:
        response = requests.get(url, headers=headers, proxies=proxy)
        soup = BeautifulSoup(response.content, "html.parser")
        product_list = soup.find_all("div", class_="sc-eDvSVe koHJnT")

        for produto in product_list:
            for a in soup.find_all("li", class_="sc-ibdxON fwviCj"):
                link = a.find("a")["href"]
                descricao = a.find("a")["title"]

                product_tokens =
set(nltk.word_tokenize(descricao.lower()))
                intersection =
len(query_tokens.intersection(product_tokens))

                if intersection > max_intersection:
                    max_intersection = intersection
                    most_similar_product = {"url": link, "alt_text":
descricao}
    except Exception as error:
        print("Erro ao buscar produto:", error)

    if most_similar_product:
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")
```

```
def executar_consulta():
    connection_hook = PostgresHook(postgres_conn_id="eiprice-dev")
    connection = connection_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(
        "SELECT ean, sku, id_loja, atributo, status, descricao FROM
kabum.produto_carga;"
    )
    resultados = cursor.fetchall()
    for resultado in resultados:
        ean, sku, id_loja, atributo, status, descricao = resultado
        print("=====================================")
        print(resultado[5])
        try:
            slug = magalu_scraper(unidecode(resultado[5]))
            print(slug)
            print("=====================================")
            cursor.execute(
                "INSERT INTO kabum.sku (ean, cod_ref, id_loja, atributo,
status, slug) VALUES (%s, %s, %s, %s, %s, %s)",
                (ean, sku, id_loja, atributo, status, slug),
            )
        except Exception as error:
            print("=====================================")
            print(f"Erro ao inserir sku: {sku}")
            print(error)
            print("=====================================")
            continue
    connection.commit()
    connection.close()


run_query = PythonOperator(
    task_id="run_query",
    python_callable=executar_consulta,
    dag=dag,
)

run_query


FILEPATH: C:\Users\andwe\Downloads\classification-
scrapers\scraper_mercadolivre.py
from datetime import datetime, timedelta

import nltk
import requests
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from bs4 import BeautifulSoup
from unidecode import unidecode

nltk.download("punkt")
```

```
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 5, 2),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "scraper_mercadolivre",
    default_args=default_args,
    description="Classificador de produtos do Mercado Livre",
    schedule_interval="0 21 * * *",
)


def mercado_livre_scraper(query):
    query_tokens = set(nltk.word_tokenize(query.lower()))
    max_intersection = 0
    most_similar_product = None
    SEARCH_PRODUCTS_URL = "https://lista.mercadolivre.com.br/"
    MIN_SIMILARITY = 0.6
    MIN_INTERSECTION = 2

    proxy = {"http": "http://Eiprice-cc-
any:DQSXomtV7qri@gw.ntnt.io:5959"}

    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36",
        "Accept-Language": "en-US,en;q=0.5",
    }

    url = f"{SEARCH_PRODUCTS_URL}{query.replace(' ', '-')}"
    response = requests.get(url, headers=headers, proxies=proxy)

    soup = BeautifulSoup(response.content, "html.parser")
    product_list = soup.find_all(
        "section",
        class_="ui-search-results ui-search-results--without-disclaimer
shops__search-results",
    )

    for product in product_list:
        try:
            link = product.find(
                "a",
                class_="ui-search-item__group__element shops__items-
group-details ui-search-link",
            )["href"]
            descricao = product.find(
                "a",
                class_="ui-search-item__group__element shops__items-
group-details ui-search-link",
            )["title"]

            product_tokens = set(nltk.word_tokenize(descricao.lower()))
            intersection = len(query_tokens.intersection(product_tokens))

            if intersection > max_intersection:
                max_intersection = intersection
                most_similar_product = {"url": link, "alt_text":
descricao}
        except Exception as error:
            print("=====================================")
            print(f"Erro ao buscar produto: {product}")
            print(error)
            print("=====================================")
            continue

    if most_similar_product:
        print("Produto mais similar encontrado:")
        print(most_similar_product["url"])
        return most_similar_product["url"]
    else:
        print("Nenhum produto similar encontrado.")


def executar_consulta():
    connection_hook = PostgresHook(postgres_conn_id="eiprice-dev")
    connection = connection_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(
        "SELECT ean, sku, id_loja, atributo, status, descricao FROM
kabum.produto_carga;"
    )
    resultados = cursor.fetchall()
    for resultado in resultados:
        ean, sku, id_loja, atributo, status, descricao = resultado
        print("=====================================")
        print(resultado[5])
        slug = mercado_livre_scraper(unidecode(resultado[5]))
        print(slug)
        print("=====================================")
        try:
            cursor.execute(
                "INSERT INTO kabum.sku (ean, cod_ref, id_loja, atributo,
status, slug) VALUES (%s, %s, %s, %s, %s, %s)",
                (ean, sku, id_loja, atributo, status, slug),
            )
        except Exception as error:
            print("=====================================")
            print(f"Erro ao inserir sku: {sku}")
            print(error)
            print("====================================")
            continue
    connection.commit()
    connection.close()


run_query = PythonOperator(
    task_id="run_query",
    python_callable=executar_consulta,
    dag=dag,
)

run_query


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\database\config.py
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

from app.settings import settings

SQLALCHEMY_DATABASE_URL = f"postgresql+asyncpg://{settings.POSTGRES_URL}"

engine = create_async_engine(
    SQLALCHEMY_DATABASE_URL,
    pool_size=20,
    max_overflow=30,
    pool_timeout=86400,   # aumentado para 1 dia
    pool_recycle=86400,   # aumentado para 1 dia
)

async_session = sessionmaker(engine, class_=AsyncSession)

Base = declarative_base()


async def get_db():
    db = async_session()
    try:
        yield db
    finally:
        await db.close()


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\database\redis.py
import aioredis
from app.settings import settings

# Cria uma conexão com o Redis
async def get_redis_client():
    redis_client = aioredis.from_url(settings.REDIS_URL)
    return redis_client
```

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\database\__init__.py

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\logger.py

```
import logging


def get_logger():
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)

    # create console handler
    ch = logging.StreamHandler()
    ch.setLevel(logging.INFO)

    # create formatter
    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )

    # add formatter to ch
    ch.setFormatter(formatter)

    # add ch to logger
    logger.addHandler(ch)

    return logger
```

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\main.py

```
import asyncio
from fastapi import FastAPI
from starlette.middleware.cors import CORSMiddleware

from app.router.get_clients_router import router as get_clients_router
from app.router.process_file_router import router as process_file_router


def get_app() -> FastAPI:
    app = FastAPI()
    app.include_router(get_clients_router)
    app.include_router(process_file_router)

    app.add_middleware(
        CORSMiddleware,
        allow_origins=["*"],
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )
    return app

app = get_app()


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\models\product.py
from sqlalchemy import (
    BigInteger,
    Column,
    Date,
    Float,
    Integer,
    Numeric,
    String,
)
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()


class Product(Base):
    __tablename__ = 'produto_loja'

    id_loja = Column(Integer, primary_key=True)
    ean = Column(BigInteger, unique=True)
    sku = Column(String(20), unique=True)
    descricao = Column(String(255))
    status = Column(String(1))
    id_departamento = Column(Integer)
    id_categoria = Column(Integer)
    id_marca = Column(Integer)
    id_modelo = Column(Integer)
    id_atributo = Column(Integer)
    preco_min = Column(Float)
    preco_max = Column(Float)
    custo = Column(Float)
    data_cadastro = Column(Date)
    lista = Column(String(2))
    frete = Column(String(1))
    curva = Column(String(10))
    gerentes = Column(String)
    sensibilidade = Column(String)
    estoque = Column(Integer)
    sec_id = Column(String(255))


class ProdutoCarga(Base):
    __tablename__ = 'produto_carga'

    uri = Column(String, primary_key=True)
    status = Column(Integer)
    sku = Column(String)
    secid = Column(Integer)
    preco_min = Column(Numeric)
    preco_max = Column(Numeric)
    modelo = Column(String)
    marca = Column(String)
    lista = Column(String)
    joja = Column(Integer)
    ean = Column(String)
    descricao = Column(String)
    departamento = Column(String)
    custo = Column(Numeric)
    curva = Column(String)
    categoria = Column(String)
    atributo = Column(String)


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\models\__init__.py


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\repository\get_clients_repository.py
from sqlalchemy import text
from sqlalchemy.orm import Session


class GetClientsRepository:
    async def get_schemas(self, db: Session):
        query = text(
            "SELECT schema_name FROM information_schema.schemata WHERE
schema_name NOT IN ('public', 'information_schema', 'pg_catalog',
'pg_toast', 'sku');"
        )
        result = await db.execute(query)
        return result.fetchall()


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\repository\process_file_repository.py
import io

from app.repository.validade_products_repository import
ConcorrenteHandler
from fastapi import HTTPException
from sqlalchemy import text

from app.logger import get_logger
from app.database.config import async_session
from app.service.s3_handler_service import S3Handler
from app.service.process_file_service import FileHandler


class DataValidator:
    def __init__(self):
        self.file_handler = FileHandler()
        self.s3_handler = S3Handler()
        self.concorrente_handler = ConcorrenteHandler()
        self.logger = get_logger()

    # apenas retorna o dataframe
    async def get_dataframe(self, client_name, file_name,
delete_divergences):
        try:
            dataframe = await self.file_handler.process_file(
                client_name, file_name
            )
            await self.duplicate_ean(
                dataframe, delete_divergences, client_name
            )
            self.logger.info("Successfully read the Excel file.")
        except Exception as e:
            self.logger.exception(f"Error reading the Excel file: {e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error reading the Excel file: {e}",
            )

    async def duplicate_ean(self, dataframe, delete_divergences,
client_name):
        try:
            df_ean_sku = dataframe[['ean', 'id_loja']]
            repeat_ean = []
            for ean, id_loja in df_ean_sku.itertuples(index=False,
name=None):
                async with async_session() as db:
                    query = text(
                        f"SELECT * FROM {client_name}.produto_loja WHERE
ean = :ean AND id_loja = :id_loja"
                    )
                    result = await db.execute(query, {"ean": ean,
"id_loja": id_loja})
                    if result.rowcount > 0:
                        repeat_ean.append((ean, id_loja))

            self.logger.info("Successfully dropped duplicates by ean.")
            await self.get_divergences_ean(repeat_ean, dataframe,
client_name)
            await self.delete_divergences_ean(
                repeat_ean, dataframe, delete_divergences, client_name
            )
        except Exception as e:
            self.logger.exception(f"Error dropping duplicates by ean:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error dropping duplicates by ean: {e}",
            )
    async def get_divergences_ean(self, repeat_ean, dataframe,
client_name):
        try:
            divergences_ean = dataframe[
                dataframe[['ean', 'id_loja']]
                .apply(tuple, axis=1)
                .isin(repeat_ean)
            ]
            await self.save_divergences_ean_to_s3(divergences_ean,
client_name)
            self.logger.info("Saved divergences by ean to S3.")
        except Exception as e:
            self.logger.exception(
                f"Error saving divergences by ean to S3: {e}"
            )
            raise HTTPException(
                status_code=400,
                detail=f"Error saving divergences by ean to S3: {e}",
            )

    async def delete_divergences_ean(
        self, repeat_ean, dataframe, delete_divergences, client_name
    ):
        try:
            if delete_divergences is True:
                df_ean_sku = dataframe[['ean', 'id_loja']]

                async with async_session() as db:
                    for _, row in df_ean_sku.iterrows():
                        ean, id_loja = row['ean'], row['id_loja']

                        # Primeira consulta DELETE
                        query1 = text(
                            f"DELETE FROM {client_name}.produto_loja
WHERE ean = :ean AND id_loja = :id_loja"
                        )
                        await db.execute(
                            query1, {"ean": ean, "id_loja": id_loja}
                        )

                        # Segunda consulta DELETE
                        query2 = text(
                            f"DELETE FROM {client_name}.sku WHERE ean =
:ean AND id_loja = :id_loja"
                        )
                        await db.execute(
                            query2, {"ean": ean, "id_loja": id_loja}
                        )

                    await db.commit()
                    await self.duplicate_sku(
                        dataframe, delete_divergences, client_name
                    )
                self.logger.info("Deleted divergences by ean.")
            else:
                async with async_session() as db:
                    for ean in repeat_ean:
                        dataframe.drop(
                            dataframe[dataframe['ean'] == ean].index,
                            inplace=True,
                        )
                    await self.duplicate_sku(
                        dataframe, delete_divergences, client_name
                    )
        except Exception as e:
            self.logger.exception(f"Error deleting divergences by ean:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error deleting divergences by ean: {e}",
            )

    async def duplicate_sku(self, dataframe, delete_divergences,
client_name):
        try:
            df_sku = dataframe[['sku', 'id_loja']]
            # para cada sku verifica se tem no banco de dados
            repeat_sku = []
            for sku, id_loja in df_sku.itertuples(index=False):
                async with async_session() as db:
                    query = text(
                        f"SELECT * FROM {client_name}.produto_loja WHERE
sku = :sku AND id_loja = :id_loja"
                    )
                    result = await db.execute(query, {"sku": str(sku),
"id_loja": id_loja})
                    if result.rowcount > 0:
                        repeat_sku.append((sku, id_loja))
            self.logger.info("Successfully dropped duplicates by sku.")
            await self.get_divergences_sku(repeat_sku, dataframe,
client_name)
            await self.delete_divergences_sku(
                repeat_sku, dataframe, delete_divergences, client_name
            )
        except Exception as e:
            self.logger.exception(f"Error dropping duplicates by sku:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error dropping duplicates by sku: {e}",
            )

    async def get_divergences_sku(self, repeat_sku, dataframe,
client_name):
        try:
            diverngences_sku =
dataframe[dataframe['sku'].isin(repeat_sku)]
            await self.save_divergences_sku_to_s3(
                diverngences_sku, client_name
            )
            self.logger.info("Saved divergences by sku to S3.")
        except Exception as e:
            self.logger.exception(
                f"Error saving divergences by sku to S3: {e}"
            )
            raise HTTPException(
                status_code=400,
                detail=f"Error saving divergences by sku to S3: {e}",
            )

    async def delete_divergences_sku(
        self, repeat_sku, dataframe, delete_divergences, client_name
    ):
        try:
            if delete_divergences is True:
                df_sku = dataframe[['sku', 'id_loja']]

                async with async_session() as db:
                    for _, row in df_sku.iterrows():
                        sku, id_loja = row['sku'], row['id_loja']

                        # Primeira consulta DELETE
                        query1 = text(
                            f"DELETE FROM {client_name}.produto_loja
WHERE sku = :sku AND id_loja = :id_loja"
                        )
                        await db.execute(
                            query1, {"sku": str(sku), "id_loja": id_loja}
                        )

                        # Segunda consulta DELETE
                        query2 = text(
                            f"DELETE FROM {client_name}.sku WHERE cod_ref
= :sku AND id_loja = :id_loja"
                        )
                        await db.execute(
                            query2, {"sku": str(sku), "id_loja": id_loja}
                        )

                    await db.commit()
                self.logger.info("Deleted divergences by sku.")
                return await self.insert_data(dataframe, client_name)
            else:
                # apaga os dados en no repeat_sku do dataframe
                for sku in repeat_sku:
                    dataframe.drop(
                        dataframe[dataframe['sku'] == sku].index,
inplace=True
                    )
                self.logger.info("Deleted divergences by sku.")
                return await self.insert_data(dataframe, client_name)
        except Exception as e:
            self.logger.exception(f"Error deleting divergences by sku:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error deleting divergences by sku: {e}",
            )

    async def save_divergences_ean_to_s3(
        self, divergences_ean, client_name: str
    ) -> str:
        try:
            file = io.BytesIO()
            ean_divergences = divergences_ean
            ean_divergences.to_excel(file, index=False)
            file.seek(0)
            self.logger.info("Saved divergences by ean to S3.")
            return await self.s3_handler.upload_to_s3(
                'divergences_ean.xlsx',
                file,
                client_name,
                file_bucket="divergencias",
            )
        except Exception as e:
            self.logger.exception(
                f"Error saving divergences by ean to S3: {e}"
            )
            raise HTTPException(
                status_code=400,
                detail=f"Error saving divergences by ean to S3: {e}",
            )

    async def save_divergences_sku_to_s3(
        self, divergences_sku, client_name: str
    ) -> str:
        try:
            file = io.BytesIO()
            sku_divergences = divergences_sku
            sku_divergences.to_excel(file, index=False)
            file.seek(0)
            self.logger.info("Saved divergences by sku to S3.")
            return await self.s3_handler.upload_to_s3(
                'divergences_sku.xlsx',
                file,
                client_name,
                file_bucket="divergencias",
            )
        except Exception as e:
            self.logger.exception(
                f"Error saving divergences by sku to S3: {e}"
            )
            raise HTTPException(
                status_code=400,
                detail=f"Error saving divergences by sku to S3: {e}",
            )
    async def insert_data(self, dataframe, client_name):
        try:
            async with async_session() as db:
                dataframe = dataframe.fillna(value="")
                for index, row in dataframe.iterrows():
                    values = {
                        'ean': row['ean'],
                        'id_loja': row['id_loja'],
                        'sku': str(row['sku']),
                        'descricao': row['descricao'],
                        'departamento': row['departamento'],
                        'categoria': row['categoria'],
                        'marca': row['marca'],
                        'modelo': str(row['modelo']),
                        'atributo': row['atributo'],
                        'preco_min': float(row['preco_min'])
                        if row['preco_min']
                        else None,
                        'preco_max': float(row['preco_max'])
                        if row['preco_max']
                        else None,
                        'status': row['status'],
                        'custo': float(row['custo']) if row['custo'] else
None,
                        'lista': row['lista'],
                        'url': row['url'],
                        'curva': row['curva'],
                        'sec_id': row['sec_id'],
                    }
                    query = text(
                        f"""INSERT INTO {client_name}.produto_carga (ean,
id_loja, sku, descricao, departamento, categoria, marca, modelo,
atributo, preco_min, preco_max, status, custo, lista, url, curva, sec_id)
                        VALUES (:ean, :id_loja, :sku, :descricao,
:departamento, :categoria, :marca, :modelo, :atributo, :preco_min,
:preco_max, :status, :custo, :lista, :url, :curva, :sec_id)"""
                    )
                    await db.execute(query, values)
                    await db.commit()
                    # Adicione o comando adicional aqui
                    cadastra_produtos = text(
                        f"SELECT
{client_name}.cadastro_produtos({row['id_loja']})"
                    )
                    await db.execute(cadastra_produtos)
                    await db.commit()
            self.logger.info("Inserted data into database.")
            return await self.concorrente_handler.clean_dataframe(
                dataframe, client_name
            )
        except Exception as e:
            self.logger.exception(f"Error inserting data into database:
{e}")
```

```
            raise HTTPException(
                status_code=400,
                detail=f"Error inserting data into database: {e}",
            )


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\repository\validade_products_repository.py
import io

import pandas as pd
from fastapi import HTTPException

from app.logger import get_logger
from app.database.config import async_session
from app.service.s3_handler_service import S3Handler

class ConcorrenteHandler:
    def __init__(self):
        self.s3_handler = S3Handler()
        self.logger = get_logger()

    async def _get_concorrentes_from_db(self, client_name):
        try:
            async with async_session() as db:
                query = f"""
                    SELECT c.id_concorrente, l.descricao AS
nome_concorrente
                    FROM {client_name}.concorrente c
                    JOIN public.loja l ON c.id_concorrente = l.id
                """
                result = await db.execute(query)
            self.logger.info("Got concorrentes from database.")
            return result.fetchall()
        except Exception as e:
            self.logger.exception(
                f"Error getting concorrentes from database: {e}"
            )
            raise HTTPException(
                status_code=400,
                detail=f"Error getting concorrentes from database: {e}",
            )

    async def clean_dataframe(self, dataframe, client_name):
        try:
            df = dataframe
            df = df.fillna(value="")
            df = df.loc[
                :,
                [
                    'ean',
                    'sku',
                    'descricao',
                    'departamento',
                    'categoria',
                    'marca',
                ],
            ]
            df = df.rename(columns={'descricao': 'produto'})
            concorrentes = await 
self._get_concorrentes_from_db(client_name)
            self.logger.info("Cleaned dataframe.")
            return await self._add_concorrentes_to_dataframe(
                df, concorrentes, client_name
            )
        except Exception as e:
            self.logger.exception(f"Error cleaning dataframe: {e}")
            raise HTTPException(
                status_code=400, detail=f"Error cleaning dataframe: {e}"
            )

    async def _add_concorrentes_to_dataframe(
        self, df, concorrentes, client_name
    ):
        try:
            dfs = []
            for concorrente in concorrentes:
                id_concorrente = concorrente[0]
                nome_concorrente = concorrente[1]
                df_concorrente = df.assign(
                    id_concorrente=id_concorrente,
                    nome_concorrente=nome_concorrente,
                    sku_concorrente="",
                    url_concorrente="",
                    atributo_concorrente="",
                )
                dfs.append(df_concorrente)
            df_final = pd.concat(dfs)
            self.logger.info("Added concorrentes to dataframe.")
            return await self._validate_ean_exists(df_final, client_name)
        except Exception as e:
            self.logger.exception(
                f"Error adding concorrentes to dataframe: {e}"
            )
            raise HTTPException(
                status_code=400,
                detail=f"Error adding concorrentes to dataframe: {e}",
            )

    async def get_schemas_by_ids(self, client_name):
        try:
            async with async_session() as db:
                query = f"""SELECT l."schema"
                            FROM public.loja l
                            INNER JOIN {client_name}.concorrente c ON 
l.id = c.id_concorrente
                            WHERE l."schema" IS NOT NULL AND l."schema" 
!= '';"""
```

```
                result = await db.execute(query)
            self.logger.info("Got schemas.")
            return result.scalars().all()
        except Exception as e:
            self.logger.exception(f"Error getting schemas by ids: {e}")
            raise HTTPException(
                status_code=400, detail=f"Error getting schemas by ids:
{e}"
            )

    async def validat_exists_schema(self, schema):
        try:
            async with async_session() as db:
                query = f"""SELECT EXISTS(SELECT schema_name FROM
information_schema.schemata WHERE schema_name = '{schema}')"""
                result = await db.execute(query)
            self.logger.info("Got schemas.")
            return result.fetchone()[0]
        except Exception as e:
            self.logger.exception(f"Error getting schemas by ids: {e}")
            raise HTTPException(
                status_code=400, detail=f"Error getting schemas by ids:
{e}"
            )

    async def _validate_ean_exists(self, df_final, client_name):
        try:
            schemas = await self.get_schemas_by_ids(client_name)
            data_to_extract = []
            async with async_session() as db:
                for schema in schemas:
                # Iterar sobre as linhas do DataFrame
                    valid_schema = await
self.validat_exists_schema(schema)
                    if valid_schema == False:
                        continue
                    for index, row in df_final.iterrows():
                        # Executar a consulta SQL
                        query = f"SELECT cod_ref, slug, atributo  FROM
{schema}.sku s WHERE id_loja = {row['id_concorrente']} and ean =
{row['ean']}"
                        cursor = await db.execute(query)
                        results = cursor.fetchall()
                        if len(results) > 0:
                            data_to_extract.append(
                                (
                                    row['ean'],
                                    row['sku'],
                                    row['produto'],
                                    row['departamento'],
                                    row['categoria'],
                                    row['marca'],
                                    row['id_concorrente'],
                                    row['nome_concorrente'],
                                    results[0][0],
                                    results[0][1],
                                    results[0][2],
                                )
                            )

            await self._create_dataframe_from_list(
                data_to_extract, client_name
            )
            await self._extract_data_from_list(
                data_to_extract, df_final, client_name
            )
        except Exception as e:
            self.logger.exception(f"Error validating EAN: {e}")
            raise HTTPException(
                status_code=400, detail=f"Error validating EAN: {e}"
            )

    async def _create_dataframe_from_list(self, data_to_extract,
client_name):
        try:
            df = pd.DataFrame(
                data_to_extract,
                columns=[
                    'ean',
                    'sku',
                    'produto',
                    'departamento',
                    'categoria',
                    'marca',
                    'id_concorrente',
                    'nome_concorrente',
                    'sku_concorrente',
                    'url_concorrente',
                    'atributo_concorrente',
                ],
            )
            await self.insert_classicated_into_db(df, client_name)
            return await self.upload_to_s3_classificated(df, client_name)
        except Exception as e:
            self.logger.exception(f"Error creating dataframe from list:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error creating dataframe from list: {e}",
            )

    async def insert_classicated_into_db(self, df, client_name):
        try:
            async with async_session() as db:
                for _, row in df.iterrows():
                    query = f"""
                        INSERT INTO {client_name}.sku (
                            ean,
                            cod_ref,
                            id_loja,
                            slug,
                            atributo
                        )
                        VALUES (
                            '{row['ean']}',
                            '{row['sku_concorrente']}',
                            '{row['id_concorrente']}',
                            '{row['url_concorrente']}',
                            '{row['atributo_concorrente']}'
                        )
                        ON CONFLICT DO NOTHING;
                    """
                    await db.execute(query)
                    await db.commit()
            self.logger.info("Inserted classificated into database.")
        except Exception as e:
            self.logger.exception(f"Error inserting classificated into
database: {e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error inserting classificated into database:
{e}",
            )

    async def upload_to_s3_classificated(self, df, client_name: str):
        try:
            file = io.BytesIO()
            df.to_excel(file, index=False)
            file.seek(0)
            self.logger.info("Uploaded to s3 classificated.")
            return await self.s3_handler.upload_to_s3(
                'classificated.xlsx',
                file,
                client_name,
                file_bucket="classificados",
            )
        except Exception as e:
            self.logger.exception(f"Error uploading to s3 classificated:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error uploading to s3 classificated: {e}",
            )

    async def _extract_data_from_list(
        self, data_to_extract, df_final, client_name
    ):
        try:
            eans_to_remove = [d[0] for d in data_to_extract]
            mask = df_final['ean'].isin(eans_to_remove)
            df_final = df_final.drop(df_final[mask].index)
            self.logger.info("Extracted data from list.")
            await self.insert_new_products_into_db(df_final, client_name)
            return await self.upload_to_s3(df_final, client_name)
        except Exception as e:
            self.logger.exception(f"Error extracting data from list:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error extracting data from list: {e}",
            )

    async def insert_new_products_into_db(self, df_final, client_name):
        try:
            async with async_session() as db:
                for _, row in df_final.iterrows():
                    query = f"""
                        INSERT INTO {client_name}.sku (
                        ean,
                        sku,
                        produto,
                        departamento,
                        categoria,
                        marca,
                        id_concorrente,
                        nome_concorrente,
                        sku_concorrente,
                        url_concorrente,
                        atributo_concorrente,

                        )
                        VALUES (
                            '{row['ean']}',
                            '{row['sku']}',
                            '{row['produto']}',
                            '{row['departamento']}',
                            '{row['categoria']}',
                            '{row['marca']}',
                            '{row['id_concorrente']}',
                            '{row['nome_concorrente']}',
                            '{row['sku_concorrente']}',
                            '{row['url_concorrente']}',
                            '{row['atributo_concorrente']}',
                        )
                    """
                    await db.execute(query)
                    await db.commit()
            self.logger.info("Inserted new products into database.")
        except Exception as e:
            self.logger.exception(f"Error inserting new products into
database: {e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error inserting new products into database:
{e}",
            )
    async def upload_to_s3(self, df_final, client_name: str):
        try:
            file = io.BytesIO()
            df_final.to_excel(file, index=False)
            file.seek(0)
            self.logger.info("Uploaded to s3.")
            return await self.s3_handler.upload_to_s3(
                'ready_to_classification.xlsx',
                file,
                client_name,
                file_bucket="novos_produtos",
            )
        except Exception as e:
            self.logger.exception(f"Error uploading to s3: {e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error uploading to s3: {e}",
            )
```

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\repository\__init__.py

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\router\get_clients_router.py

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session

from app.database.config import get_db
from app.repository.get_clients_repository import GetClientsRepository

router = APIRouter()
repository = GetClientsRepository()


@router.get("/clients")
async def get_clients(db: Session = Depends(get_db)):
    return await repository.get_schemas(db)
```

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\router\process_file_router.py

```
from fastapi import APIRouter, BackgroundTasks
from pydantic import BaseModel

from app.service.background_service import ProcessFileService

router = APIRouter()

process_file_service = ProcessFileService()
```

```
class ProcessFileSchema(BaseModel):
    client_name: str
    file_name: str
    delete_divergence: bool
    email: str


@router.post("/process/file")
async def process_file(
    process_file_schema: ProcessFileSchema,
    background_tasks: BackgroundTasks = BackgroundTasks(),
):
    client_name = process_file_schema.client_name
    file_name = process_file_schema.file_name
    delete_divergence = process_file_schema.delete_divergence
    email = process_file_schema.email
    background_tasks.add_task(
        process_file_service.process_file,
        client_name,
        file_name,
        delete_divergence,
        email,
    )
    return {"message": "File processed successfully"}


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\router\__init__.py


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\service\background_service.py
from app.repository.process_file_repository import DataValidator
from app.service.send_email_service import SendEmail


class ProcessFileService:
    def __init__(self):
        self.data_validator = DataValidator()
        self.send_email = SendEmail()

    async def process_file(
        self,
        client_name: str,
        file_name: str,
        delete_divergence: bool,
        email: str,
    ):
        try:
            await self.data_validator.get_dataframe(
                client_name, file_name, delete_divergence
            )
            await self.send_email.send_email_success(email, client_name)
        except Exception as e:
            await self.send_email.send_email_error(email, client_name,
str(e))
            raise e


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\service\process_file_service.py
import io

import pandas as pd
from fastapi import HTTPException

from app.logger import get_logger
from app.service.s3_handler_service import S3Handler


class FileHandler:
    def __init__(self):
        self.s3_handler = S3Handler()
        self.dataframe = None
        self.logger = get_logger()

    async def read_xlsx(
        self, client_name: str, file_name: str
    ) -> pd.DataFrame:
        try:
            file_name, file_bytes = await self.s3_handler.download_file(
                client_name, file_name
            )
            self.dataframe = pd.read_excel(file_bytes)
            self.logger.info("Successfully read the Excel file.")
        except Exception as e:
            self.logger.exception(f"Error reading the Excel file: {e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error reading the Excel file: {e}",
            )

    async def drop_duplicates(self):
        try:
            await self.drop_duplicates_by_ean_and_sku()
            await self.drop_duplicates_by_ean()
            await self.drop_duplicates_by_sku()
            self.logger.info("Dropped duplicates from the dataframe.")
        except Exception as e:
            self.logger.exception(f"Error dropping duplicates: {e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error dropping duplicates: {e}",
            )

    async def drop_duplicates_by_ean_and_sku(self):
        try:
            self.duplicated_by_ean_and_sku = self.dataframe[
                self.dataframe.duplicated(subset=['ean', 'sku'],
keep='first')
            ]
            self.dataframe.drop_duplicates(subset=['ean', 'sku'],
inplace=True)
            self.logger.info("Dropped duplicates by ean and sku.")
        except Exception as e:
            self.logger.exception(
                f"Error dropping duplicates by ean and sku: {e}"
            )
            raise HTTPException(
                status_code=400,
                detail=f"Error dropping duplicates by ean and sku: {e}",
            )

    async def drop_duplicates_by_ean(self):
        try:
            self.duplicated_by_ean = self.dataframe[
                self.dataframe.duplicated(subset=['ean'], keep=False)
                & ~self.dataframe.duplicated(subset=['sku'])
            ]
            self.dataframe.drop_duplicates(
                subset=['ean'], keep=False, inplace=True
            )
            self.logger.info("Dropped duplicates by ean.")
        except Exception as e:
            self.logger.exception(f"Error dropping duplicates by ean:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error dropping duplicates by ean: {e}",
            )

    async def drop_duplicates_by_sku(self):
        try:
            self.duplicated_by_sku = self.dataframe[
                self.dataframe.duplicated(subset=['sku'], keep=False)
                & ~self.dataframe.duplicated(subset=['ean'])
            ]
            self.dataframe.drop_duplicates(
                subset=['sku'], keep=False, inplace=True
            )
            self.logger.info("Dropped duplicates by sku.")
        except Exception as e:
            self.logger.exception(f"Error dropping duplicates by sku:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error dropping duplicates by sku: {e}",
            )

    async def save_duplicates_by_ean_and_sku_to_s3(
        self, client_name: str
    ) -> str:
        try:
            file = io.BytesIO()
            self.duplicated_by_ean_and_sku.to_excel(file, index=False)
            file.seek(0)
            self.logger.info("Saved duplicates by ean and sku to S3.")
            return await self.s3_handler.upload_to_s3(
                'duplicated_by_ean_and_sku.xlsx',
                file,
                client_name,
                file_bucket="duplicadas",
            )
        except Exception as e:
            self.logger.exception(
                f"Error saving duplicates by ean and sku to S3: {e}"
            )
            raise HTTPException(
                status_code=400,
                detail=f"Error saving duplicates by ean and sku to S3:
{e}",
            )

    async def save_duplicates_by_ean_to_s3(self, client_name: str) ->
str:
        try:
            file = io.BytesIO()
            self.duplicated_by_ean.to_excel(file, index=False)
            file.seek(0)
            self.logger.info("Saved duplicates by ean to S3.")
            return await self.s3_handler.upload_to_s3(
                'duplicated_by_ean.xlsx',
                file,
                client_name,
                file_bucket="duplicadas",
            )
        except Exception as e:
            self.logger.exception(f"Error saving duplicates by ean to S3:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error saving duplicates by ean to S3: {e}",
            )

    async def save_duplicates_by_sku_to_s3(self, client_name: str) ->
str:
        try:
            file = io.BytesIO()
            self.duplicated_by_sku.to_excel(file, index=False)
            file.seek(0)
            self.logger.info("Saved duplicates by sku to S3.")
            return await self.s3_handler.upload_to_s3(
                'duplicated_by_sku.xlsx',
                file,
                client_name,
                file_bucket="duplicadas",
            )
        except Exception as e:
            self.logger.exception(f"Error saving duplicates by sku to S3:
{e}")
            raise HTTPException(
                status_code=400,
                detail=f"Error saving duplicates by sku to S3: {e}",
            )

    async def process_file(self, client_name, file_name):
        await self.read_xlsx(client_name, file_name)
        await self.drop_duplicates()
        await self.save_duplicates_by_ean_and_sku_to_s3(client_name)
        await self.save_duplicates_by_ean_to_s3(client_name)
        await self.save_duplicates_by_sku_to_s3(client_name)
        return self.dataframe


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\service\s3_handler_service.py
import io
from typing import Tuple

import boto3

from app.settings import settings


class S3Handler:
    def __init__(self):
        self.access_key = settings.ACCESS_KEY
        self.secret_key = settings.SECRET_KEY
        self.bucket_name = settings.BUCKET_NAME

    async def download_file(
        self, client_name: str, file_name: str
    ) -> Tuple[str, bytes]:
        s3 = boto3.client(
            's3',
            aws_access_key_id=self.access_key,
            aws_secret_access_key=self.secret_key,
        )
        bucket_name = self.bucket_name
        file = io.BytesIO()
        s3.download_fileobj(bucket_name, f"{client_name}/{file_name}",
file)
        file.seek(0)
        return file_name, file.read()

    async def download_file_new_product(
        self, client_name: str, file_name: str
    ) -> Tuple[str, bytes]:

        s3 = boto3.client(
            's3',
            aws_access_key_id=self.access_key,
            aws_secret_access_key=self.secret_key,
        )
        bucket_name = self.bucket_name
        file = io.BytesIO()
        s3.download_fileobj(
            bucket_name, f"{client_name}/novos_produtos/{file_name}",
file
        )
        file.seek(0)
        return file_name, file.read()

    async def upload_to_s3(
        self,
        file_name: str,
        file: io.BytesIO,
        client_name: str,
        file_bucket: str,
    ) -> str:
        s3 = boto3.client(
            's3',
            aws_access_key_id=self.access_key,
            aws_secret_access_key=self.secret_key,
        )
        s3.put_object(
            Body=file,
            Bucket=self.bucket_name,
            Key=f"{client_name}/{file_bucket}/{file_name}",
        )
        return
f"https://{self.bucket_name}.s3.amazonaws.com/{client_name}/{file_bucket}
/{file_name}"


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\service\send_email_service.py
import requests
from fastapi import HTTPException


class SendEmail(object):
    def __init__(self):
        self.BASE_URL = "https://retail.eiprice.com.br/email/report"

    async def send_email_success(self, email, client_name):
        post = {
            "to": [{"email": email, "name": email.split(".")[0]}],
            "title": "List Manager - Arquivo processado com sucesso!",
            "content": f"""
            <html>
                <body>
                    <h2 style="color:#00305B;">Arquivo do cliente
{client_name} foi processado com sucesso!</h2>
```

```
                    <p style="color:#00305B;">O List Manager finalizou o
processamento do arquivo. Para visualizar os links do S3, acesse: <a
href="https://listmanager.eiprice.com.br/list-files"
target="_blank">https://listmanager.eiprice.com.br/list-files</a></p>
                </body>
            </html>
        """
        }

        payload = {
            "template": "emails.default",
            "title": post['title'],
            "to": post['to'],
            "data": {"content": post['content']},
        }

        response = requests.post(self.BASE_URL, json=payload)
        if not response.ok:
            raise HTTPException(
                status_code=response.status_code, detail=response.text
            )
        return response

    async def send_email_error(self, email, client_name, e):
        post = {
            "to": [{"email": email, "name": email.split("@")[0]}],
            "title": "List Manager - Erro ao processar arquivo",
            "content": f"""
            <html>
                <body>
                    <h2 style="color:#00305B;">Erro ao processar arquivo
do cliente {client_name}</h2>
                    <p style="color:#00305B;">O List Manager encontrou um
erro ao processar o arquivo. Entre em contato com o suporte.</p>
                    <p style="color:#00305B;">Erro: {e}</p>
                </body>
            </html>
        """
        }

        payload = {
            "template": "emails.default",
            "title": post['title'],
            "to": post['to'],
            "data": {"content": post['content']},
        }

        response = requests.post(self.BASE_URL, json=payload)
        if not response.ok:
            raise HTTPException(
                status_code=response.status_code, detail=response.text
            )
        return response
```

```
FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\service\__init__.py


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\settings.py
from pydantic import BaseSettings


class Settings(BaseSettings):
    POSTGRES_URL: str
    ACCESS_KEY: str
    SECRET_KEY: str
    BUCKET_NAME: str
    REDIS_URL: str

    class Config:
        env_file = ".env"


settings = Settings()


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\tests\conftest.py


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\tests\repository\test_process_file_repository.py
import io
import unittest
from unittest.mock import ANY, MagicMock, patch

import pandas as pd
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from app.models.product import Product
from app.repository.process_file_repository import DataValidator
from app.service.process_file_service import FileHandler, S3Handler


class TestDataValidator(unittest.TestCase):
    def setUp(self):
        self.data_validator = DataValidator()
        self.engine = create_engine("sqlite:///:memory:")
        self.session = sessionmaker(bind=self.engine)()
        self.session.execute = MagicMock()

    @patch("app.service.process_file_service.FileHandler.process_file")
    @patch(

"app.repository.process_file_repository.DataValidator.duplicate_ean"
    )
    def test_get_dataframe(self, mock_duplicate_ean, mock_process_file):
        mock_process_file.return_value = pd.DataFrame()
        self.data_validator.get_dataframe(
            "client_name", "file_name", "delete_divergences",
self.session
        )
        mock_duplicate_ean.assert_called_once()

    @patch(

"app.repository.process_file_repository.DataValidator.get_divergences_ean
"
    )
    @patch(

"app.repository.process_file_repository.DataValidator.delete_divergences_
ean"
    )
    def test_duplicate_ean(
        self, mock_delete_divergences_ean, mock_get_divergences_ean
    ):
        mock_result = MagicMock()
        mock_result.rowcount = 1
        self.session.execute.return_value = mock_result

        dataframe = pd.DataFrame({"ean": [123, 456]})
        self.data_validator.duplicate_ean(
            dataframe, "delete_divergences", "client_name", self.session
        )
        mock_get_divergences_ean.assert_called_once()
        mock_delete_divergences_ean.assert_called_once()

    @patch(

"app.repository.process_file_repository.DataValidator.duplicate_sku"
    )
    def test_delete_divergences_ean(self, mock_duplicate_sku):
        mock_result = MagicMock()
        mock_result.rowcount = 1
        self.session.execute.return_value = mock_result

        dataframe = pd.DataFrame({"ean": [123, 456]})
        self.data_validator.delete_divergences_ean(
            "repeat_ean",
            dataframe,
            "delete_divergences",
            "client_name",
            self.session,
        )
        mock_duplicate_sku.assert_called_once()

    @patch(
"app.repository.process_file_repository.DataValidator.get_divergences_sku
"
    )
    @patch(

"app.repository.process_file_repository.DataValidator.delete_divergences_
sku"
    )
    def test_duplicate_sku(
        self, mock_delete_divergences_sku, mock_get_divergences_sku
    ):
        mock_result = MagicMock()
        mock_result.rowcount = 1
        self.session.execute.return_value = mock_result

        dataframe = pd.DataFrame({"sku": [123, 456]})
        self.data_validator.duplicate_sku(
            dataframe, "delete_divergences", "client_name", self.session
        )
        mock_get_divergences_sku.assert_called_once()
        mock_delete_divergences_sku.assert_called_once()

    @patch("app.service.process_file_service.S3Handler.upload_to_s3")
    def test_save_divergences_ean_to_s3(self, mock_upload_to_s3):
        dataframe = pd.DataFrame({"ean": [123, 456]})
        divergences_ean = dataframe
        client_name = "test_client"
        self.data_validator.save_divergences_ean_to_s3(
            divergences_ean, client_name
        )
        mock_upload_to_s3.assert_called_once_with(
            'divergences_ean.xlsx',
            ANY,
            'test_client',
            file_bucket="divergencias",
        )

    @patch("app.service.process_file_service.S3Handler.upload_to_s3")
    def test_save_divergences_sku_to_s3(self, mock_upload_to_s3):
        dataframe = pd.DataFrame({"sku": [123, 456]})
        divergences_sku = dataframe
        client_name = "test_client"
        self.data_validator.save_divergences_sku_to_s3(
            divergences_sku, client_name
        )
        mock_upload_to_s3.assert_called_once_with(
            'divergences_sku.xlsx',
            ANY,
            'test_client',
            file_bucket="divergencias",
        )
```

```
FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\tests\router\test_health_check.py
from fastapi.testclient import TestClient

from app.main import app

client = TestClient(app)


def test_health_check_success():
    response = client.get("/health")
    assert response.status_code == 200


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\tests\router\test_process_file_router.py
from unittest.mock import patch

from fastapi.testclient import TestClient

from app.main import app

client = TestClient(app)


@patch("app.repository.process_file_repository.DataValidator.get_datafram
e")
def test_process_file_success(mock_process_file):
    client_name = "test_client"
    file_name = "test_file.xlsx"
    mock_process_file.return_value = True
    response = client.post(
        "/process/file",
        params={
            "client_name": client_name,
            "file_name": file_name,
            "delete_divergence": True,
        },
    )
    assert response.status_code == 200
    assert response.json() == {"message": "File processed successfully"}


FILEPATH: C:\Users\andwe\Downloads\eiprice-list-
manager\app\tests\service\test_process_file_service.py
import io
import unittest
from unittest.mock import ANY, MagicMock, patch

import pandas as pd

from app.service.process_file_service import FileHandler, S3Handler
```

```
class TestS3Handler(unittest.TestCase):
    def setUp(self):
        self.s3_handler = S3Handler()

    @patch("boto3.client")
    def test_download_file(self, mock_boto3_client):
        # Arrange
        client_name = "test_client"
        file_name = "test_file.xlsx"

        # Act
        result = self.s3_handler.download_file(client_name, file_name)

        # Assert
        self.assertEqual(result[0], file_name)
        self.assertIsInstance(result[1], bytes)
        mock_boto3_client.assert_called_once()

    @patch("boto3.client")
    def test_upload_to_s3(self, mock_boto3_put_object):
        # Arrange
        file_name = "test_file.xlsx"
        file = io.BytesIO()
        client_name = "test_client"
        file_bucket = "test_bucket"

        # Act
        result = self.s3_handler.upload_to_s3(
            file_name, file, client_name, file_bucket
        )

        # Assert
        self.assertEqual(
            result,

f"https://{self.s3_handler.bucket_name}.s3.amazonaws.com/{client_name}/{f
ile_bucket}/{file_name}",
        )
        mock_boto3_put_object.assert_called_once()


class TestFileHandler(unittest.TestCase):
    def setUp(self):
        self.file_handler = FileHandler()
        self.file_handler.s3_handler = MagicMock()
        self.client_name = 'client_name'
        self.file_name = 'file_name'

    def test_read_xlsx_error(self):
        self.file_handler.s3_handler.download_file.side_effect =
ValueError(
            "Error"
        )
        with self.assertRaises(ValueError):
            self.file_handler.read_xlsx(self.client_name, self.file_name)

    def test_save_duplicates_by_ean_and_sku_to_s3(self):
        df = pd.DataFrame({'ean': [1, 2, 2, 3, 4], 'sku': [5, 6, 6, 7,
8]})
        self.file_handler.duplicated_by_ean_and_sku = df[
            df.duplicated(subset=['ean', 'sku'])
        ]
        self.file_handler.s3_handler.upload_to_s3.return_value = (
            'duplicated_by_ean_and_sku.xlsx'
        )
        result = self.file_handler.save_duplicates_by_ean_and_sku_to_s3(
            self.client_name
        )
        self.assertEqual(result, 'duplicated_by_ean_and_sku.xlsx')
        self.file_handler.s3_handler.upload_to_s3.assert_called_with(
            'duplicated_by_ean_and_sku.xlsx',
            ANY,
            self.client_name,
            file_bucket='duplicadas',
        )

    def test_save_duplicates_by_ean_to_s3(self):
        df = pd.DataFrame({'ean': [1, 2, 2, 3, 4], 'sku': [5, 6, 6, 7,
8]})
        self.file_handler.duplicated_by_ean =
df[df.duplicated(subset=['ean'])]
        self.file_handler.s3_handler.upload_to_s3.return_value = (
            'duplicated_by_ean.xlsx'
        )
        result = self.file_handler.save_duplicates_by_ean_to_s3(
            self.client_name
        )
        self.assertEqual(result, 'duplicated_by_ean.xlsx')
        self.file_handler.s3_handler.upload_to_s3.assert_called_with(
            'duplicated_by_ean.xlsx',
            ANY,
            self.client_name,
            file_bucket='duplicadas',
        )

    def test_save_duplicates_by_sku_to_s3(self):
        df = pd.DataFrame({'ean': [1, 2, 2, 3, 4], 'sku': [5, 6, 6, 7,
8]})
        self.file_handler.duplicated_by_sku =
df[df.duplicated(subset=['sku'])]
        self.file_handler.s3_handler.upload_to_s3.return_value = (
            'duplicated_by_sku.xlsx'
        )
        result = self.file_handler.save_duplicates_by_sku_to_s3(
            self.client_name
        )
        self.assertEqual(result, 'duplicated_by_sku.xlsx')
        self.file_handler.s3_handler.upload_to_s3.assert_called_with(
            'duplicated_by_sku.xlsx',
            ANY,
            self.client_name,
            file_bucket='duplicadas',
        )

    def test_drop_duplicates_by_ean(self):
        data = {'ean': [1, 2, 3, 4, 4], 'sku': ['a', 'b', 'c', 'd', 'e']}
        df = pd.DataFrame(data)
        obj = FileHandler()
        obj.dataframe = MagicMock(return_value=df)
        obj.drop_duplicates_by_ean()
        obj.dataframe.drop_duplicates.assert_called_once_with(
            subset=['ean'], keep=False, inplace=True
        )
        self.assertTrue(obj.duplicated_by_ean.equals(df.loc[df['ean'] ==
4]))

    def test_drop_duplicates_by_sku(self):
        data = {'ean': [1, 2, 3, 4, 4], 'sku': ['a', 'b', 'c', 'd', 'e']}
        df = pd.DataFrame(data)
        obj = FileHandler()
        obj.dataframe = MagicMock(return_value=df)
        obj.drop_duplicates_by_sku()
        obj.dataframe.drop_duplicates.assert_called_once_with(
            subset=['sku'], keep=False, inplace=True
        )
        self.assertTrue(obj.duplicated_by_sku.equals(df.loc[df['sku'] ==
'e']))

    def test_drop_duplicates_by_ean_and_sku(self):
        data = {'ean': [1, 2, 3, 4, 4], 'sku': ['a', 'b', 'c', 'd', 'e']}
        df = pd.DataFrame(data)
        obj = FileHandler()
        obj.dataframe = MagicMock(return_value=df)
        obj.drop_duplicates_by_ean_and_sku()
        obj.dataframe.drop_duplicates.assert_called_once_with(
            subset=['ean', 'sku'], inplace=True
        )
        self.assertTrue(
            obj.duplicated_by_ean_and_sku.equals(
                df.loc[(df['ean'] == 4) & (df['sku'] == 'e')]
            )
        )
```

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\tests\__init__.py

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\app\__init__.py

FILEPATH: C:\Users\andwe\Downloads\eiprice-list-manager\docker-compose.yaml

```yaml
version: '3'

services:
  eiprice_manager:
    build:
      context: .
      dockerfile: Dockerfile
    command: uvicorn app.main:app --host 0.0.0.0 --port 8082 --reload
    working_dir: /app
    volumes:
      - .:/app
    restart: always
    env_file:
      - .env
    ports:
      - '8082:8082'
    dns:
      - 8.8.8.8
      - 9.9.9.9
```