Documentation

Introduction

This project, developed as part of the Nordeus Data Engineering Challenge 2023, is an API and database management system designed to allow easy and efficient querying of event related data. Utilizing Python, Docker, and PostgreSQL technologies, it offers a robust solution for the tasks described in the problem statement.

The purpose of this document is to provide a detailed overview of the project, including its structure, installation, usage, and troubleshooting guidelines. It is intended for both potential testers who wish to deploy and utilize the application, as well as fellow developers interested in understanding the project.

Due to certain ambiguities in the challenge guidelines, the following assumptions were made to guide the project's development:

- In the original introduction to the problem, 4 event types were described, mainly registration, login, logout and transaction. Later in the dataset description, for the *event_type* field, the constraint given was *registration, login, logout*. I assumed that the transaction event was just forgotten about, so I didn't filter it out;
- In the query description, one of the outputs required for the user level stats query was *Name of user*. As this is not present in the dataset given, I ignored that requirement while doing the task;
- The assumptions after this are not related to the guidelines, but are is vital for the data cleaning process I implemented, and that is that I assume that a successful registration event also logs the user in.
- As this is a game, I assumed that a specific user could only have one active session at a time.

This documentation is structured to provide a comprehensive guide to the project. Following this introduction, the document will delve into the detailed project structure, installation guide and usage instructions.

Project structure

This section outlines the overall structure and organization of the project. The project is structured into several key directories and files, each serving a specific purpose in the application's functionality.

API Directory

Overview:

This directory houses the core logic of the application, including data processing and database interaction.

Sub-directories and Key Files:

- **data_cleaning/**: Contains scripts for processing and cleaning data;
- **data/**: Stores the exchange_rates.jsonl and events.jsonl files, which are the primary data; sources.
- **data_processing.py**: The script used for data cleaning and preparation;
- **db/**: Manages database connections and operations;
    - **db_connection.py**: Handles the connection to the database and executes database operations;
- **requirements.txt**: Lists all the Python dependencies required for the API;
- **app.py**: The main application file that serves as the entry point for the API.

DB Directory

Overview:

Contains SQL scripts related to database setup and schema.

Key File:

- **create_tables.sql**: Script for creating the database tables necessary for the application.

Root Directory

Docker Integration:

- **docker-compose.yaml**: Defines the multi-container Docker setup for the application;
- **Dockerfile**: Specifies the steps to build the Docker image for the API.

Testing:

- **postman_collection.json**: Contains a collection of Postman requests for testing the API endpoints.

API

This section describes the API component of the project, which is responsible for handling data processing, database interactions, and serving the application's core functionalities.

Overview:

The API is built using Python, utilizing the Flask framework, designed to process and manage data efficiently. It serves as the backbone of the application, connecting the front-end with the database layer.

Data Cleaning:

Purpose:

The purpose of this module is to provide the means to process the event data provided in the events.jsonl dataset and prepare it to be loaded into the database.

Structure:

- **data/:**
  - **events.jsonl** - the main dataset used for the challenge. It contains a list of JSON objects, each representing a potential event;
  - **exchange_rates.jsonl** - the supplementary dataset, which contains the exchange rates for the currencies found in the *events.jsonl* dataset
- **data_processing.py**
  - This is the script that actually implements the data cleaning and preparation process. The process takes place in three main steps which will be described below. These steps ensure a robust data cleaning process which filters all invalid and unsensible events, and prepares the data to be loaded into the database:
    - The first step is to read all the JSON objects from the file and store them in-memory, after that, we sort the dataset based on the *event_timestamp* field, as the order of events is important for the validation process;
    - The second step is the validation and grouping process. More precisely:
      - Each event is first validated. For an event to be valid we there are three main conditions:
        - the event_id must be unique;
        - the values of all fields in the JSON must comply with the restraints given in the problem statement;
        - the user has to be in the appropriate state to generate the event;
          - The user state is tracked in a (user_id, int) map, where the states are determined as follows:
            - user_id not in set of keys - user is not registered;
            - value for user_id is True - user is logged in;
            - value for user_id is False - user is logged out;
          - Based on this the following filtering was done:
            - If a user is not registered, all events other than registration are ignored;
            - If a user is registered, all subsequent registration events are ignored;

- While a user is logged in all login events are ignored;
- While a user is logged out, all logout and transaction events are ignored;

- Upon validation, all events are grouped into three maps, all of which use user_ids as keys:
  - one for registration events;
  - one for transaction events;
  - one for both login and logout events;

- The third step is where we transform the events into forms that enable loading into the database, more precisely we create a (user_id, touple) map where the touple contains the following elements:
  - user_record - a dictionary containing all values necessary to create an User record in the database;
  - user_transaction_records - a list of dictionaries containing all values necessary to create a Transaction record in the database;
  - user_session_records - a list of dictionaries containing all values necessary to create a Session record in the database;
    - Here it is important to point out a few things:
      - The first session record uses the *event_timestamp* of the registration event as its *login_timestamp*;
      - We allow for the *logout_timestamp* value to be null.

Database Connection

Purpose:

The purpose of this module is to enable communication with the database. It houses all the necessary methods for creating the connection with the PostgreSQL server, inserting data into the server, and querying the database.

Structure:

- **db_connection.py**
  - This script contains the concrete implementation of all the functions used while communicating with the database, and it also houses some of the information necessary for the application to work properly. Described below are the main functionalities:
    - Upon interpretation of the script, it fetches the URL for connecting to the database, and sets up the SQLAlchemy Base, engine and configures the sessionmaker. It also reflects the concrete entity classes (User, Session, Transaction) from the PostgreSQL server. It also reads the *exchange_rate.jsonl* file and records the EUR to USD conversion rate;
    - The two main functionalities of the class inserting and querying data:
      - Data is inserted into the database by using the *load_data* function. It takes the map of touples generated by the data cleaning process and inserts all records in it into the database;
      - Querying is done using the *query_user_data* and *query_game_data* functions. They are accessed by targeting the appropriate endpoints

provided by the API. They establish a session with the database server and by utilizing multiple queries fetch the required data, and package it into a dictionary so it can be easily delivered back to the client.

Requirements

- **requirements.txt**
    - A basic Python requirements file to be installed before running the service.

Application Endpoint

Overview:

The main orchestrating script of the service. It first processes the data, establishes a connection to the database and loads the data and then runs the service which provides the endpoints used for data querying.

Structure:

- **app.py**
    - The app.py script contains the main method used for starting the script. It also defines the two endpoints provided by the API:
        - *queryUserData*
            - It takes two parameters:
                - a required user_id of type UUID that represents the unique id of the user we are querying for;
                - an optional date in YYYY-MM-DD format to further filter the results;
            - It returns values specified in the problem statement in the form of a JSON;
        - *queryGameData*
            - It takes two optional parameters:
                - country, which is used to filter results based on the country of user;
                - date, which filters the results based on date;
            - It returns values specified in the problem statement in the form of a JSON.
    - Outside of this it also sets up the Flask app, by defining the host and port values and enables cross-origin requests.

DB

This section describes the DB component of the project, which is responsible for creating the necessary tables on the PostgreSQL server.

Overview:

The script used to create the tables is written in the PostgreSQL dialect of SQL. It creates three tables: *users, transactions* and *sessions.* The database schema was designed to allow simple and efficient querying.

Structure:

- **users**
  - This table represents a user of our game. It contains the following columns:
    - user_id of type UUID that represents the primary key;
    - country of type VARCHAR(255);
    - registration_timestamp of type TIMESTAMP;
    - device_os of type VARCHAR(255);
    - marketing_campaign of type VARCHAR(255);
- **sessions**
  - This table represents a single game session of a user. It contains the following columns:
    - session_id of type SERIAL that represents the primary key;
    - user_id of type UUID used as a foreign key for the users table;
    - login_timestamp of type TIMESTAMP;
    - logout_timestamp of type TIMESTAMP;
    - session_length_seconds of type INT;
    - session_valid of type BOOLEAN with default value false;
    - session_ended of type BOOLEAN with default value true.
  - It is important to note that the session_valid and session_ended values are generated by a transaction triggered on every insert. The decision is based on whether the session length is greater than 1 second, and whether the session has been finished (user logged out).
- **transactions**
  - This table represents a transaction executed by a user. It contains the following columns:
    - transaction_id of type SERIAL that represents the primary key;
    - user_id of type UUID used as a foreign key for the users table;
    - transaction_timestamp of type TIMESTAMP;
    - amount of type DECIMAL(10, 2);
    - currency of type VARCHAR(3);

Root Directory

Docker Integration

Overview:

The following files enable our service to be containerized as run using the docker virtualization platform.

Structure:

- **docker-compose.yaml**
  - Defines the multi-container Docker setup for the application. We define two containers, *postgres* and *python*, which represent the two parts of our system. We also assign ports to them, namely 5432 to the *postgres* container and 8086 to the *python* container. Alongside that we specify the order of starting the containers, utilizing the *depends_on* property to make the *python* container wait for the *postgres* server to be ready to accept connections.
- **Dockerfile**
  - Specifies the steps to build the Docker image for the API component of our system.

Testing

- **QUERIES.postman_collection.json**
  - Contains a collection of Postman requests for testing the API endpoints.

Usage guide

As the whole project is dockerized, the only requirements for running are compatible Docker and docker-compose versions. For building the project I used the following versions:

- Docker version 24.0.6, build ed223bc
- Docker Compose version v2.23.0-desktop.1

When the appropriate versions are installed, you should start the Docker deamon (usually done by starting the Docker desktop app), then you should position the terminal into the top level project directory (should be *de_challenge*) and run the following command:

```
docker compose up -d
```

This starts the containers in the background of the terminal and leaves them running. After some time, the app will spin up and be ready for use. You can check this by using:

```
docker logs python
```

There should be a message showing that the Flask app has started.

After that, you can use the HTTP requests given in the QUERIES.postman_collection.json file to test the service by changing the parameter values.