FORECAST CHALLENGE

REST API - Python

Jun - 2022

Ramiro Cugat

Index.

Objective

With the objective of providing a sample of my professional skills, I have been tasked with a challenge. This documentation serves as a technical and functional guide to help you understand how the application work.

Limitations

As this project was done during my time off work, time itself has proven to be the only limiting factor influencing in the quality of the app. And even in these conditions, the result is solid.

Components

Job

- Once the variables are set, the program first calculates all the coordinates of the planets.
- With the XY Coordinates, defines in which situation are we in, therefore, calculating the forecast. This happens one day at a time, looping it for as long as necessary.
- Once we get the results, we insert them in a NoSQL database to be used later by the REST API.
- If enabled, it executes the answers to the challenge

REST API

- If an async function is called (Ex, an incoming get request), reads the database, parses the request and returns the result.
- Returns an error if the request is not valid.

Functions

area()

Parameters : (x1, y1, x2, y2, x3, y3)

Calculates the perimeter of a triangle given the corresponding XY of each vertex.

Returns an INT

is_inside()

Parameters : (x1, y1, x2, y2, x3, y3, x, y):

Calculates if the given point (x,y) is inside the coordinates provided to form a triangle.

Returns a **Boolean**

is_straight()

Parameters : (coordinates_bundle)

Calculates if the coordinates provided inside a dict form a straight line.

Returns a **Boolean**

perimeter_counter()

Parameters : (max_perimeter Type=**dict**, new_perimeter Type=**dict**)

The objective of this function is to provide a solution that finds the max perimeter of the triangles inside a loop. Then, store the max perimeter including how many times it happened and the days it did.

Returns a **dict**.

forecast ()

Parameters : (planet1_x, planet1_y, planet2_x, planet2_y, planet3_x, planet3_y, sun_location_x, sun_location_y)

Finds the forecast of the day, includes most of the previous functions.

Returns a **String**

Classes and Methods

Class planet

- Includes: name, distance, clockwise, rotation, cords_x, cords_y
- This class includes all the planets to be defined inside the scope of this application.

Planet Methods:

calc_coordinates_xy()

- Parameters : (self, day)
- Given the day parameter, this method calculates the real X Y coordinates of the planet at that time.

Class weather

- Includes: name, count, max_streak
- This class includes all the weathers to be calculated inside the scope of this application.

weather Methods:

counter()

- Parameter : (self, actual_streak)
- Calculates the repetitions of each weather to later be used to determine the contents of the database.

REST API

The REST API provided in this project was built in Flask.

It is a small application that takes from a NoSQL database the information required and returns it to the client.

Endpoints

GET ALL [GET]

- https://planet-forecast.rj.r.appspot.com/
- Returns Vulcano Forecast results.
- This Api was built as a response to a Vulcano need, so it makes sense that it returns only the Vulcano Forecast (Not that it matters but they all share the forecast as of today).
- No query parameter is accepted, it will ignore it.

GET ONE [GET]

- https://planet-forecast.rj.r.appspot.com/clima
- Returns one Vulcano Forecast Result
- **Query Parameters: day (Must be an Int)**
- Example: https://planet-forecast.rj.r.appspot.com/clima?day=233

Error Handling

In case of a query that results in an error, the server will respond with a custom error message with the cause.

Examples:

- User sends a REST request with a letter inside the query
- /clima?day=23A3
- User sends a REST request with a day that it doesn't exist / it isn't calculated
- /clima?day=9999999999999

Data Models

There are 2 different data models designed to this project. The first solution was to implement a SQL data model, but as time was of the essence, it was much easier to configure a NoSQL database.

The data model was built around the fact that there are 3 different planets, and even though they share a similarity on how they calculate the forecast and its results, it makes sense that each planet will have its own forecast log for they may need it in the future (Global warming is a real thing in Vulcano!).

SQL Data Model Code:

```
CREATE TABLE PLANETS
(PLANET_ID                  NUMBER NOT NULL
,PLANET_NAME                VARCHAR2(250)
,PLANET_DESCRIPTION         VARCHAR2(500)
,SUN_DISTANCE               VARCHAR2(50)
,ROTATION                   VARCHAR2(100)
,PRIMARY KEY (PLANET_ID)
)
```

```
CREATE TABLE PLANET_FORECAST_DAY
(PLANET_ID                  NUMBER NOT NULL
,PLANET_DAY_ID              VARCHAR2(250)
,COORDENADA_X               VARCHAR2(250)
,COORDENADA_Y               VARCHAR2(250)
,DAY_DESC                   VARCHAR2(50)
,PERIMETER                  VARCHAR2(250)
,FORECAST                   VARCHAR2(250)
,PRIMARY KEY (PLANET_DAY_ID)
,FOREIGN KEY (PLANET_ID) REFERENCES PLANETS(PLANET_ID)
)
```

NoSQL Data Model: (Example with 2 planets)


```
[
  {
    "planet_id":1,
    "planet_name":"Ferengui",
    "planet_description":"Economy is blooming",
    "sun_distance":500,
    "rotation":1,
    "forecast":[
      {
        "day_id":1,
        "day":1,
        "cord_x":270.1511529340699,
        "cord_y":420.73549240394823,
        "day_desc":"Nice day",
        "perimeter":604563.4077602724,
        "weather":"normal"
      }
    ]
  },
  {
    "planet_id":2,
    "planet_name":"Betasoide",
    "planet_description":"Economy is blooming",
    "sun_distance":2000,
    "rotation":3,
    "forecast":[
      {
        "day_id":1,
        "day":1,
        "cord_x":-1979.9849932008908,
        "cord_y":282.24001611973443,
        "day_desc":"Nice day",
        "perimeter":604563.4077602724,
        "weather":"normal"
      }
    ]
  }
]
```