Workflow managers

Nombre del alumno: José Ulises Vallejo Sierra

Código: 219747905

Sección: D06

Curso: Computación Tolerante a Fallas

Nombre del profesor: Michel Emanuel López Franco

Realizar un programa que sea capaz de revisar el estatus de tu aplicación.
Técnica a utilizar: prefect

En esta ocasión se nos plantea un tema actual y novedoso que promete ser solución a diversos problemas o fallas en la computación y es una rama de la ingeniería que apoya a la tolerancia de estas fallas para mitigar los problemas presentados en las aplicaciones y ser sutil y comprensivo con el usuario y el desarrollador, se trata de la gestión de el flujo de trabajo, y en esta ocasión específicamente con la utilización del módulo prefect que nos ayudará precisamente con este rubro en el lenguaje Python, cuando sea necesario y como sistema de administración de flujos de trabajo, Prefect facilita agregar registros, reintentos, mapeo dinámico, almacenamiento en caché, notificaciones de fallas y más a sus canalizaciones de datos, por lo que es idóneo para trabajar con subtareas dentro de un proceso.

Programa: Creación de una canalización

Usaremos Prefect para completar una tarea relativamente simple hoy: ejecutar una canalización ETL. Esta canalización descargará los datos de una API ficticia, los transformará y los cargará a una base de datos nueva. Usaremos un sitio web para alojar dicha API ficticia que contiene archivos de bases de datos.

Primero importamos las librerías json para la transformación del flujo de datos, request para importar la dirección de la API, collections, contextlib y sqlite3 son módulos que nos ayudarán a proporcionar funciones para el manejo de datos tipo data base, para enriquecer y hacer mas complejas nuestras tareas a administrar, y del módulo prefect importamos task para la creación de las tareas y Flow para el manejo del flujo de dichas tareas

```
1   import requests
2   import json
3   from collections import namedtuple
4   from contextlib import closing
5   import sqlite3
6
7   from prefect import task, Flow
```

Figura 1. Importación de librerías

Creamos nuestra tarea para la extracción de datos de la API definida con request y alojada en la variable r que será la que aloja la solicitud (GET) requerida y posteriormente para su visualización transformamos la solicitud a formato diciionarios o JSON regresando como parametro el key ‘hits’ que contiene la lista de diccionarios de los registros de la base de datos, por lo que solo vamos a extraer algunas keys de los diccionarios de la API y permitir un mejor flujo

```
9   ## extract
10  @task
11  def get_complaint_data():
12      r = requests.get("https://www.consumerfinance.gov/data-research/consumer-complaints/search/api/
13      response_json = json.loads(r.text)
14      return response_json['hits']['hits']
```

Figura 2. Primera tarea para la extracción de datos

Para transformar los datos utilizamos tuplas para acceder a los diccionarios de la lista de diccionarios anterior y mediante el for iteramos sobre cada fila o registro para cambiar el alojamiento de los datos en su estructura por una estructura más básica y así solo conservar los atributos más básicos de los registros y guardarlos en la lista complaints ya transformados y retornar dicho resultado

```
16  ## transform
17  @task
18  def parse_complaint_data(raw):
19      complaints = []
20      Complaint = namedtuple('Complaint', ['data_received', 'state', 'product', 'company',
21                                           'complaint_what_happened'])
22      for row in raw:
23          source = row.get('_source')
24          this_complaint = Complaint(
25              data_received=source.get('date_recieved'),
26              state=source.get('state'),
27              product=source.get('product'),
28              company=source.get('company'),
29              complaint_what_happened=source.get('complaint_what_happened')
30          )
31          complaints.append(this_complaint)
32      return complaints
```

Figura 3. Segunda tarea para la transformación de los datos

Para cargar los datos, la tarea creará una base de datos a la que le cargaremos los registros con el comando INSERT sql y mediante la función sqlite3 conectamos la información a la base de datos real por lo que se insertarán a dicha tabla todos los datos ya analizados y de este modo guardamos el transformado anterior en una nueva base de datos

```
34  ## load
35  @task
36  def store_complaints(parsed):
37      create_script = 'CREATE TABLE IF NOT EXISTS complaint (timestamp TEXT, state TEXT, product TEXT, company TEXT, complaint_what_happened TEXT)'
38      insert_cmd = "INSERT INTO complaint VALUES (?, ?, ?, ?, ?)"
39
40      with closing(sqlite3.connect("cfpbcomplaints.db")) as conn:
41          with closing(conn.cursor()) as cursor:
42              cursor.executescript(create_script)
43              cursor.executemany(insert_cmd, parsed)
44              conn.commit()
```

Figura 4. Tercera tarea para la carga de los datos transformados

Por último, para que el flujo funcione debemos relacionar cada una de las tareas entre sí, por lo que la sintaxis de la función with y la instancia del objeto Flow nos permitirá manejar este flujo. Dentro de este bloque de código llamamos las funciones realizadas en las tareas, lo que realmente permitirá agregarlas al flujo y relacionarlas entre sí compartiendo los datos, así que primero ejecuto la función de la tarea que extraerá los datos y el resultado de la extracción la guardo en la variable raw para después llamar la función de la tarea que transforma los datos pasando como parámetro los datos extraídos y guardando esta transformación en la variable parsed, la cual finalmente se le manda como parámetro a la función de la tarea que carga los datos transformados en nuestra nueva base de datos.

Finalmente, corremos el flujo con la función run() y para visualizarlo gráficamente utilizamos la función visualize() que requiere de el software graphicviz y que se instaló previamente.

```
45
46  with Flow("my etl flow") as f:
47      raw = get_complaint_data()
48      parsed = parse_complaint_data(raw)
49      store_complaints(parsed)
50
51  f.run()
52  f.visualize()
```

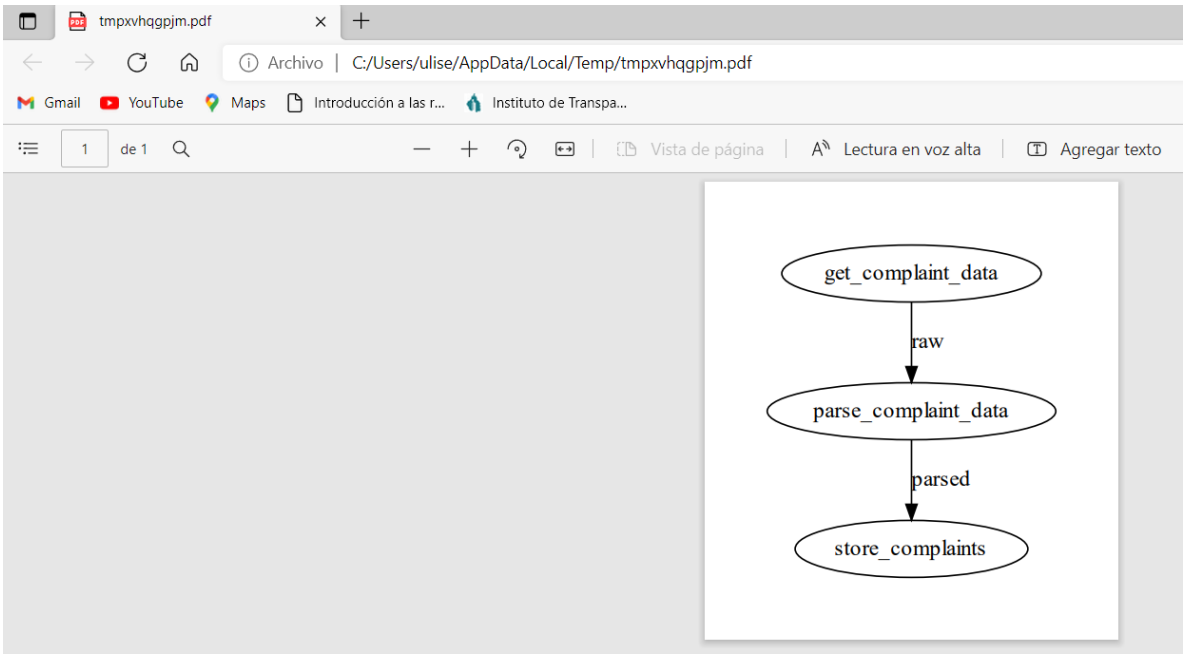
Figura 5. Establecemos el flujo y la relación de este para cada una de las tareas

Ejecución del programa

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

[2022-03-07 13:55:11-0600] INFO - prefect.FlowRunner | Beginning Flow run for 'my etl flow'
[2022-03-07 13:55:11-0600] INFO - prefect.TaskRunner | Task 'get_complaint_data': Starting task run...
[2022-03-07 13:55:15-0600] INFO - prefect.TaskRunner | Task 'get_complaint_data': Finished task run for task with final state: 'Success'
[2022-03-07 13:55:15-0600] INFO - prefect.TaskRunner | Task 'parse_complaint_data': Starting task run...
[2022-03-07 13:55:15-0600] INFO - prefect.TaskRunner | Task 'parse_complaint_data': Finished task run for task with final state: 'Success'
[2022-03-07 13:55:15-0600] INFO - prefect.TaskRunner | Task 'store_complaints': Starting task run...
[2022-03-07 13:55:15-0600] INFO - prefect.TaskRunner | Task 'store_complaints': Finished task run for task with final state: 'Success'
[2022-03-07 13:55:15-0600] INFO - prefect.FlowRunner | Flow run SUCCESS: all reference tasks succeeded
```

Ejecutamos la aplicación en la terminal y observamos el estatus de cada una de las tareas, el momento en que inició y el momento en el que finalizó con éxito; notamos que la primera tarea se ejecuta correctamente y la segunda hace lo propio con el resultado que obtuvo de la tarea anterior y así sucesivamente creando el flujo secuencial de tareas ejecutando exitosamente la dependencia de los datos



Además, gracias a la función visualize() obtenemos una visualización clara de los nodos del flujo y el parámetro requerido para su resultado, así que podemos decir que la extracción producirá ‘raw’ para su uso en la transformación y el resultado ‘parsed’ a su vez se usará para la carga, mostrando otra perspectiva de la relación del flujo y los datos de los que depende.

No solo se ejecuta la canalización de ETL (extract, transform, load), sino que también obtenemos información detallada sobre cuándo comenzó y finalizó cada tarea.

Conclusión

Prefect es un módulo de administración de flujo de trabajo que permite y simplifica la canalización y dependencia de extracción transformación y carga de datos compartidos a través de dependencias, para ello se utilizan los decoradores task que le dirán al sistema cuales son las tareas que se ejecutarán y que se deben vigilar o administrar sus dependencias con la creación de flujos de trabajo, lo que permite decirle al programa cual es la pauta de flujo que se debe ejecutar en el futuro.

En pocas palabras, este nuevo módulo Prefect permite la administración de las tareas visualizando su dependencia, flujo y estado, y también permitiendo fallas exitosas de cada tarea en dicho flujo, lo cual es muy útil en la tolerancia de fallas, ya que una canalización de datos implementada con Prefect ignorará el estado de la tarea y solamente se dedicará a transmitir los datos, esto con la finalidad de permitir usar el estado para tomar decisiones y permitir que tareas fallidas continúen con un flujo funcional

Bibliografías

- Prefect [PrefectIO]. (2020, abril 17). Getting started with Prefect (PyData Denver). Youtube. https://www.youtube.com/watch?v=FETN0iivZps
- PyData [PyDataTV]. (2019, enero 3). Task failed successfully - Jeremiah lowin. Youtube. https://www.youtube.com/watch?v=TlawR_gi8-Y&list=PLMGWGsnelbxcHmA5cVRq8a39S9s_gxAMq
- Radečić, D. (2021, julio 22). Prefect: How to write and schedule Your first ETL pipeline with Python. Towards Data Science. https://towardsdatascience.com/prefect-how-to-write-and-schedule-your-first-etl-pipeline-with-python-54005a34f10b

LINK AL REPOSITORIO:

https://github.com/UlisesVallejo/workflow-managers.git