Baga Martina - Cinquini Sara - Hassany Ariana

Enabling Component Reuse (4th Pillar)

Codice per il progetto di *Architectures for Big Data*.

L'obiettivo richiede la creazione di una architettura riusabile, ovvero che abbia il corretto livello di astrazione/generalizzazione.

```
from abc import ABC, abstractmethod
import requests
import json
import pymongo
import datetime
```

Per permettere l'implementazione del quarto pillar, *Enabling Component Reuse*, è stata creata una classe astratta `Bach_extractor` che espone i metodi per l'estrazione di dati:

- `get_conn` è il metodo per la creazione di una connessione con un server
- `get_token` è il metodo per la richiesta di un token a un provider necessario la successiva domanda di estrazione dei dati
- `get_data` è il metodo che permette lo scambio del token con i dati

```
class Bach_extractor(ABC):

    @abstractmethod
    def get_conn(self):
        pass

    @abstractmethod
    def get_token(self):
        pass

    @abstractmethod
    def get_data(self):
        pass
```

Un'altra classe astratta che è stata creata è la classe `DB` . Questa classe contiene i metodi necessari per il salvataggio dei dati su un database (historical database) che farà da layer intermedio tra il layer operational e il layer analytical, evitanto un numero eccessivo di richieste al server.

- `conn_db` è il metodo che si occupa di creare la connessione con il database
- `insert_data` è il metodo che permette di inserire i dati ottenuti precedentemente nel database

```
class DB(ABC):
  @abstractmethod
  def conn_db(self):
    pass

  @abstractmethod
  def insert_data(self):
    pass
```

Infine è stata creata una classe astratta `Analytics` che espone i metodi necessari per la visualizzazione dei dati desiderati, presenti nell'historical database:

- `create_model` è il metodo che si occupa di creare un model
- `update_model` è il metodo che permette di inserire i dati desiderati per trainare il model
- `view_model` è il metodo che restituisce una visualizzazione (grafica) del modello trainato

```
class Analytics(ABC):
  @abstractmethod
  def create_model(self):
    pass

  @abstractmethod
  def update_model(self):
    pass

  @abstractmethod
  def view_model(self):
    pass
```

Avendo definito le tre classi astratte è possile concretizzarle, per renderle specifiche alle nostre necessità, andando a ridefinire i metodi.

Di seguito sono descritti i metodi della classe `Connection_onestream(Bach_extractor)`.

`get_conn` restituisce un url utilizzabile per la richiesta del token a un provider specifico. Il provider da noi scelto è Okta, utilizzato per chiedere a OneStream i dati e che, a sua volta, necessita di alcune informazioni specifiche tra le quali:

- `client_id` che corrisponde all'ID client della propria applicazione Okta OAuth
- `scope` è un OpenID, cioè l'endpoint */token* restituirà un tokenID
- `redirect_uri` è l'url a cui viene re-indirizzato lo user agent insieme al file code e deve corrispondere a uno degli URI di reindirizzamento che sono stati specificati nell'applicazione Okta alla sua creazione
- `state` è una stringa che il server di autorizzazione riproduce quando re-indirizza l'interprete client e si usa per prevenire la falsificazione di richieste tra siti

`get_token` esegue una richiesta *post* al provider tramite l'url restituito dal metodo `get_conn` con l'aggiunta dei campi necessari. Se le credenziali dell'utente sono corrette allora riceverà una risposta contenente il token necessario per la richiesta dei dati.

`get_data` esegue una richiesta *post* a OneStream per chiedere i dati. In particolare lo scambio necessita del token restituito dalla funzione `get_token` e di un JSON in cui vengono specificati i campi per la richiesta dei dati.

```
class Connection_onestream(Bach_extractor):
  def get_conn(self, client_id, redirect_uri, state, scope):
      okta_url = 'https://${yourOktaDomain}/oauth2/default/v1/authorize?client_id='+cli
      return okta_url

  def get_token(self, okta_url, json):
    result = requests.post(okta_url, json)
    token = result.access_token
    return token

  def get_data(self, json_obj, token):
    data = requests.post(json_obj, token)
    return data
```

Di seguito sono descritti i metodi della classe `DB_mongo(DB)`. Per l'implementazione della classe DB è stato scelto un database NoSQL perchè in questo modo è possibile garantire:

- uno sviluppo flessibile del database, grazie al modello schemaless
- horizontal scalability (5th pillar)
- performance più alte nei tempi di risposta
- la creazione di distribuited systems adatti a elevate quantità di dati

Nello specifico abbiamo scelto di utilizzare MongoDB in quanto i dati sono memorizzati in documenti flessibili simili a JSON, la stessa tipologia di dato restituita da OneStream.

`conn_db` restituisce la collection `dailydata` che conterrà i dati scaricati da OneStream. Abbiamo supposto che questa operazione avvenga giornalmente.

`insert_data` inserisce nella collection creata/selezionata precedentemente i dati in formato JSON restituiti dal metodo `get_data` della classe `Connection_onestream`.

```
class DB_mongo(DB):
  def conn_db(self, myclient):
    dblist = myclient.list_database_names()
    #se non esiste il DB viene creato in automatico
    mongo_db = myclient["historical_db"]
    mycol = mongo_db["dailydata"]
```

```
    return mycol

  def insert_data(self, mycol, data):
    insert = mycol.insert_one(data)
    return('insert ok')
```

Di seguito sono descritti i metodi della classe `Economics_analytics(Analytics)`.

`crete_model` crea un modello a cui successivamente verranno aggiunti i dati. Questo modello viene utilizzato per effettuare previsioni sui possibili andamenti futuri.

`update_model` inserisce nel modello solamente i dati che appartengono a un intervallo temporale compreso tra due timestamp (`ts1` e `ts2`). Se all'interno del JSON restituito da OneStream non fosse presente il timestamp, supponiamo che questo campo venga aggiunto in automatico durante l'inserimento del dato nell'historical database. In questo caso consideriamo che il dato restituito abbia un timestamp nel formato dd/mm/yyyy relativo al giorno del download del dato (download giornaliero). In particolare il metodo riceve in input i seguenti attributi:

- `col` rappresenta la collezione in cui sono contenuti i dati giornalieri
- `ts1` e `ts2` sono i due timestamp scelti dall'utente e che delimitano l'intervallo temporale entro il quale si vogliono considerare i dati (supponiamo che `ts1 > ts2`)
- `model` è il modello creato precedentemente dal metodo `create_model`

`view_data` permette la visualizzazione grafica del modello creato.

```
class Economics_analytics(Analytics):
  def create_model(self):
    ...
    return model

  def update_model(self, col, ts1, ts2, model):
    for obj in col.find(): #obj = file json del db
      #ipotizziamo che in obj ci sia il campo timestamp
      ts = obj['timestamp']
      dt_obj = datetime.fromtimestamp(ts)
      if dt_obj < ts1 and dt_obj > ts2:
        model.add(obj)
    return model

  def view_model(self,model):
    ...
    return graph
```

Connessione a OneStream tramite provider e richiesta del token.

```
client_id = ''
client_secret = ''
redirect_uri = ''
state = ''
scope = 'WebApiMachineToMachineScope'

conn = Connection_onestream()
url_conn = conn.get_conn(client_id, redirect_uri, state, scope) #connessione con il pro
print(url_conn)
```

```
https://${yourOktaDomain}/oauth2/default/v1/authorize?client_id=&response_type=code&s
```

```
# TODO si potrebbe provare a mettere get_data incatenato con get_conn

dict_json = {
  "grant_type": "client_credentials",
  "scope": scope,
  "client_id": client_id,
  "client_secret": client_secret
}
json_obj = json.dumps(dict_json)

token = conn.get_token(url_conn, json_obj) # se credenziali okay, il provider ci restit
```

Richiesta dei dati.

```
base_url = ''
data_stream = '' # sono i dati che richiediamo a OneStream specifici per l'azienda cons

dict_json = {
  "BaseSìWebServerUrl": base_url,
  "ApplicationName": data_stream,
  "SequenceName": 'Export Dara', # nome del pacchetto dove saranno presenti i dati
}
json_obj = json.dumps(dict_json)

data = conn.get_data(json_obj, token)
```

Salvataggio dei dati nel database.

```
# connessione a MongoDB
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
```

```
conn_db_mongo = DB_mongo()
col = conn_db_mongo.conn_db(myclient)
conn_db_mongo.insert_data(col, data)
```

Per visualizzare le performance dell'azienda l'utente deve specificare l'intervallo di tempo che vuole prendere in considerazione, andando a specificare due date. Per esempio per prelevare i dati degli ultimi sei mesi si dovrebbero inserire le seguenti date:

```
ts1 = datetime.date(2022 , 10 , 7)
ts2 = datetime.date(2022 , 4 , 7)
```

Creiamo il model per poi visualizzare i dati in un grafico.

```
new_analytics = Economics_analytics()
new_model = new_analytics.create_model()
updated_model = new_analytics.update_model(col, ts1, ts2, new_model)
updated_graph = new_analytics.view_model(updated_model)
```

Prodotti Colab a pagamento - Annulla i contratti qui