Architectures for big data - Assignment 1

931271 - Alessandro Di Gioacchino 942428 - Yousef Hammar
08395A - Mattia Paravisi

November 10, 2022

1 Introduzione

Questo assignment ha come obiettivo quello di costruire una architettura usando le classi astratte di Python. L'architettura dovrebbe aiutare un possibile team di developer a raggiungere il seguente business requirement: "I need to show Intercompany impacts on my Company Balance Sheet, without any impact on OneStream performance during the Month End Closing activities".

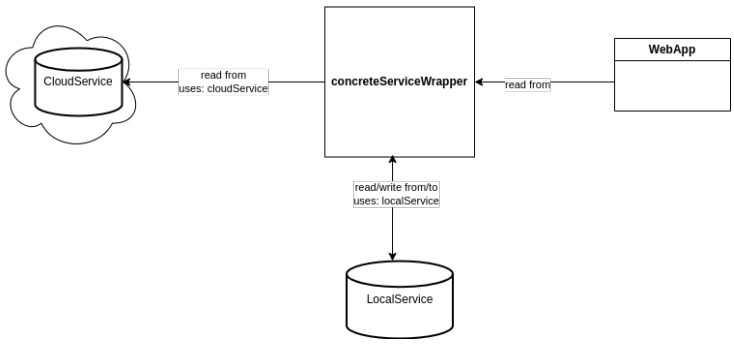
2 Premesse

Durante la risoluzione dell'assignment nel nostro gruppo sono state fatte delle premesse per limitare i possibili scenari che avremmo dovuto affrontare:

1. Abbiamo un servizio online che possiede un database dal quale dobbiamo leggere.
2. Il servizio online ha almeno una tabella di log da cui vogliamo leggere.
3. Dal servizio online ci limitiamo a leggere con lo scopo di creare un mirror locale.
4. Ogni tabella di log ha un timestamp.
5. Abbiamo un database locale.
6. Il database locale ha una tabella che viene usata per fare il mirroring della tabella online.
7. Il database locale permette lettura e scrittura.
8. Non ci interessa restituire alla web application i dati più .aggiornati

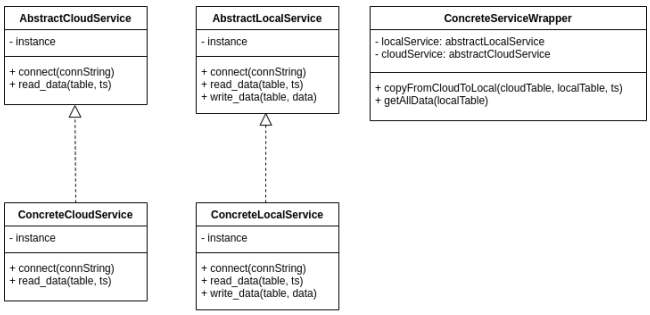
3 Architettura

Abbiamo riassunto la situazione come presentato nel seguente schema:



Consideriamo un servizio online - che deve avere un database associato - da cui vogliamo leggere, un database locale su cui possiamo scrivere e dal quale possiamo leggere e un differente servizio online (ad esempio una web application) che vuole accedere ai dati di cui stiamo facendo il mirroring. Vogliamo fare

il mirroring dei dati per non intaccare le performance delle procedure che il primo servizio esegue a fine mese. La nostra architettura consiste in un software che esegue tutte le operazioni richieste utilizzando come supporto delle classi astratte che verranno adattate ad ogni servizio utilizzato quando rese concrete. In particolare abbiamo immaginato la seguente gerarchia:



Avremo quindi una classe astratta per il servizio online e una classe astratta per il servizio in locale. Le due classi astratte sono molto simili tra loro ma abbiamo deciso di mantenere le due implementazioni separate per permettere di avere per la classe che si occupa del servio cloud solo il metodo read, mentre alla classe che si occupa del servizio locale sia read che write. Così facendo, inoltre, possiamo forzare la classe wrapper a prendere come argomenti i due tipi nel modo corretto; avendo un'unica gerarchia e non le due classi separate avremmo potuto rischiare di invertire `localService` e `cloudService` a causa di un errore. Per quanto riguarda la classe `ServiceWrapper` abbiamo deciso di non fare altro che considerare i due metodi utili al fine di raggiungere i business requirements:

- il primo metodo copia i dati dal servizio cloud al servizio locale (fa il mirroring);
- il secondo metodo restituisce tutti i dati da una certa tabella.

3.1 Esempio di codice

Uno snippet di codice molto ad alto livello che utilizza la gerarchia di cui sopra è il seguente:

```
1  cloudService = ConcreteCloudService()
2  cloudService.connect("mysql://root:password@127.0.0.1:3306/public")
3  print("CloudService connected")
4
5  localService = ConcreteLocalService()
6  localService.connect("postgres://root:password@127.0.0.1:5432/
       public")
7  print("LocalService connected")
8
9  concreteServiceWrapper = ConcreteServiceWrapper(cloudService,
       localService)
10 concreteServiceWrapper.copyFromCloudToLocal("cloudTable", "
       localTable", ("timestamp", ">", "2022-11-10 09:55:45"))
```

Nella chiamata `serviceWrapper.copyFromCloudToLocal(...)`:

```
def copyFromCloudToLocal(self, cloudTable, localTable, timestamp):
    data = self.cloudService.read_data(cloudTable, timestamp)
    for d in data:
        self.localService.write_data(localTable, d)
```

Così facendo le classi concrete per il servizio locale e cloud possono essere qualsiasi; non ci interessa su che DBMS si basano i due servizi, ci basta implementare un metodo `read` e un metodo `write` coerente alla tecnologia utilizzata. Così facendo permetteremmo la lettura delle righe di una specifica tabella che hanno un timestamp consono, e la scrittura nella corrispondente tabella locale.

Consideriamo l'implementazione della seguente classe:

```
select = {
    "exists": "",
  "columns": [],
  "from": "",
  "subQuery": "",
  "fromAlias": "",
  "joins": [],
  "where": "",
  "groupBy": [],
  "orderBy": [],
  "limit": "",
  "offset": "",
    "insertvalue": ""
}

class QueryBuilder():

    def __init__(self):
        self.query = ""
        self.selectdict = select.copy()

    def columns(self, columns):
        self.selectdict["columns"] = columns

    def from_table(self, table):
        self.selectdict["from"] = table

    def where(self, where):
        self.selectdict["where"] = where

    def value(self, value):
        self.selectdict["insertvalue"] = value

    def build_select(self):
        self.query = ""
        if len(self.selectdict["columns"]) > 0:
            self.query = "SELECT " + ', '.join(self.selectdict["
    columns"])
        if self.selectdict["from"]:
            self.query += " FROM " + self.selectdict["from"]
        if self.selectdict["where"]:
            self.query += " WHERE " + " ".join([self.selectdict["
    where"][0],self.selectdict["where"][1],"'" + self.selectdict["
    where"][2] + "'"])
        return self.query

    ...
```

Come è facile intuire la classe descritta permette di creare query arbitrarie: potremmo quindi utilizzarla per costruire quelle che verranno utilizzate nei

metodi `read_data()` e `write_data()` delle due classi descritte sopra, ad esempio:

```
    def read_data(self, table, timestamp):
        qb = QueryBuilder()
        qb.columns(['*'])
        qb.from_table(table)
        if timestamp != "":
            qb.where(timestamp)
        return self.conn.execute(qb.build_select())
```

`engine` è ottenuto utilizzando il metodo `connect` nel seguente modo:

```
    def connect(self, connString):
        self.conn = create_engine(connString)
```

Usando ad esempio `sqlAlchemy`, `url` può essere una stringa del tipo:

- `mysql://root:password@127.0.0.1:3306/public`
- `postgresql://root:password@127.0.0.1:5432/public`

È facile osservare che solo la fase di connessione cambia in base al DBMS scelto, mentre le operazioni che seguono sono indipendenti da esso.

4 Connessione ai pillars

1. Being the framework for satisfying requirements: in questo caso il requirement era uno solo come riportato nell'introduzione; una volta definita l'architettura è stato possibile verificare se il requirement fosse stato raggiunto: la nostra architettura permette di fare il mirroring del servizio durante i giorni in cui non vengono effettuate le operazioni di fine mese, quindi effettivamente non stiamo inficiamo sulle performance del servizio.

2. Being the managerial basis for cost estimation and process management: la definizione di un'architettura come quella presentata permette di eseguire un'analisi dei costi. In questo caso si dovrebbero valutare i costi OPEX come i costi del servizio cloud, degli sviluppatori e dei servizi locali. Inoltre sappiamo quante persone impiegare per sviluppare il numero di classi descritto nell'architettura.

3. Enabling component reuse: la nostra architettura permette di effettuare il mirroring a partire da due servizi generici, quindi è altamanete riutilizzabile.

4. Avoiding handover and people lock-in: l'architettura è semplice da documentare in quanto le classi che vengono utilizzate sono poche, così come i metodi che ogni classe deve implementare.