Architectures for Big Data - First assignement

Federico Cristiano Bruzzone
&
Andrea Longoni
&
Massimiliano Visconti

Indice

1 Overview

The assignment is focused on implementing a simple Architecture using Python Abstract Classes to build a structure that is lead to retrieve data from generics externals data source for importing them into an internal DB that will be used from the analyst's team for analysis purposes.

2 Project Requirements

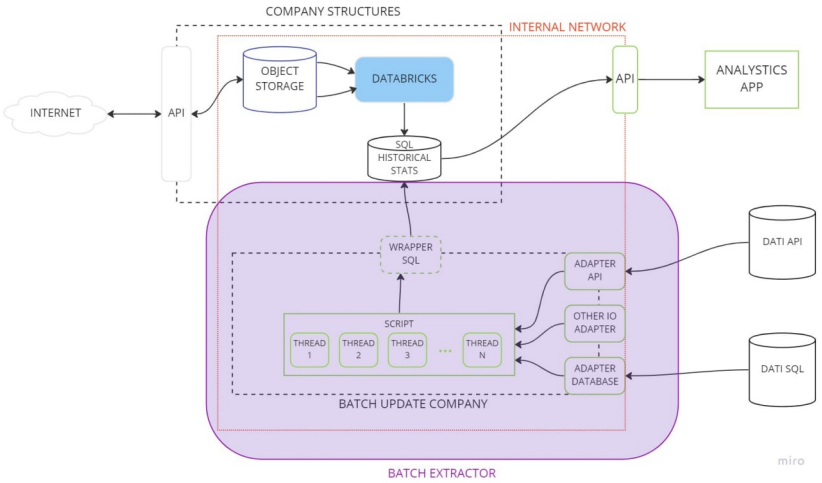
The project requirements are reduced to the ideation and abstract implementation of a software architecture which satisfies to follow the *Software Architecture Pillars* seen during the class lessons.

In specific those pillars are:

1. Being the framework for satisfying requirements
2. Being the technical basis for design
3. Being the managerial basis for cost estimation and process management
4. Enabling component reuse
5. Allowing a tidy scalability
6. Avoiding handover and people lock in

3 Software Architecture Pillars

3.1 Being the framework for satisfying requirements



3.1.1 Functional Requirements

The software will need the ability to read data from generic external sources (like Databases, public API, Data Stream, ...) and to prepare and process them before their insertion inside the local Company Historical DB.

This software, that furthermore we will call 'Batch Extractor', will need to be able to adapt itself for retrieving data from any data source that the company will identify as important to be importable inside the system for allowing analysts to be free of finding whatever data source should fit better for their needs.

The batch system will also need to be adaptable for changes on the internal structure of the Company, in particular the DB, for letting free the company to make the best business and behavioral choices in any moment free of the current implementations of the system.

Important will also be that the structure should be scalable on multiple and concurrent input data source, for allowing the system to inject data at different speeds based on the need of the analyst and the environment.

3.1.2 Technical Requirements

Due to accomplish the necessity of the structure of being adaptable to any external source, is important that the software will presents a layer of external adapters that will be used to connect to the external sources for retrieving properly data based on the external infrastructures and needs.

Important will also be the possibility for the system to be able to add new adapters to interact with new outside data sources. Those will simply need to pass the data to the script unknowingly of the internal DB implementation and this will then take care of them for the inserting inside the Historical DB.

Important will also be the presence of wrappers for the connection with the internal Historical DB. This feature will provide another abstraction layer that will be important to allow the company to apport in any moment whatever changes on the internal technologies used for storing data.

3.1.3 Security Requirements

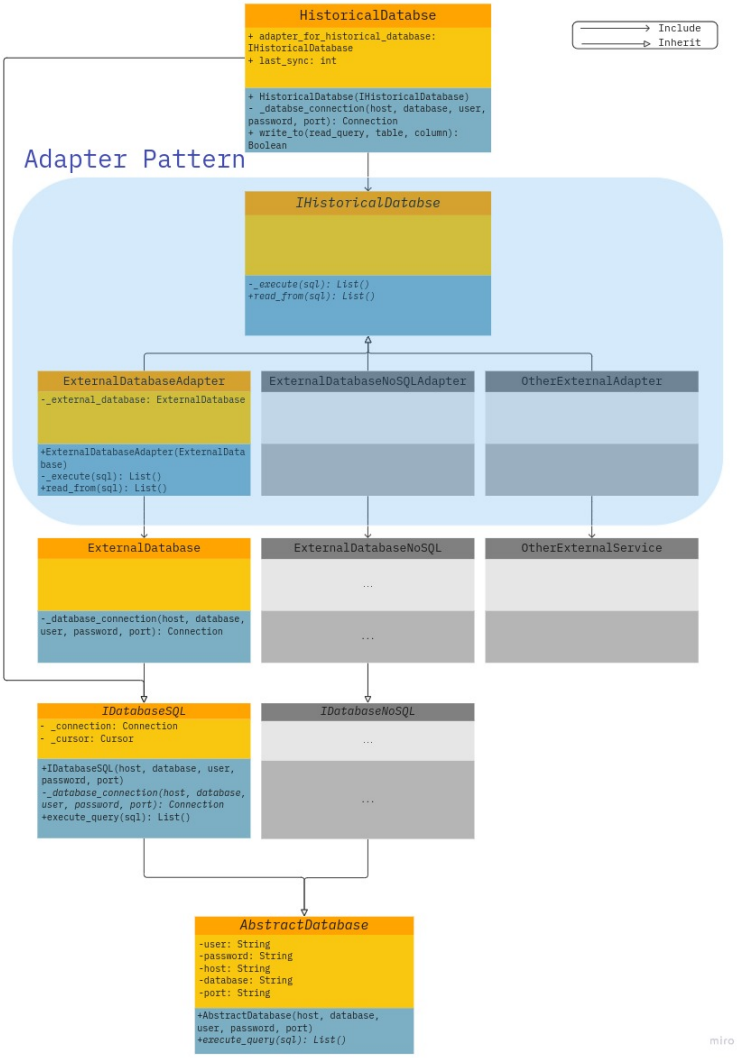
The system will need to be placed inside the company network due to avoid the necessity of opening external ports directly to the Historical DB, for reduce the possibility of penetration of malicious actors inside the system.

Important will also be the process of input sanitization after receiving data from the external source and before inserting them inside the queries, due to avoid the exposure of the system to attack like SQL injection or similar.

3.2 Being the framework for satisfying requirements

The development of the architecture, later illustrated, has been done keeping in mind that we will have a single immutable database (historical database) on which we will write and possible multiple databases or data sources from which we will read.

To interface on these databases (or data sources) we thought to use multiple adapters to transform the inbound data in a writable format for the Historical DB.



Please note: The gray part is not implemented. We will talk about it in the reusability chapter.

We have thought to use Adapter (Wrapper) pattern, because this will allow us to keep the same historical database and make it be able to communicate with the different types of external data source.

This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one.

1. *AbstractDatabase* is an abstract class that contains the information for connecting with any database.

   In the inherited class, you will have to implement execute_query(...) method.

2. *IDatabaseSQL* is an abstract class that define behavior of the SQL database.

   Since IDatabaseSQL inherits from AbstractDatabase it must implements execute_query(...) method.

   In the inherit class, you will have to implement _database_connection(...) method.

   Assuming that in python any database sql library implement .connector.connect(...) and .cursor() methods:

   (a) we use _connection field to store the connection to the specific database;

   (b) we use _cursor field to store the cursor to the specific database.

   Note that _cursor contanis the .execute(...) method that is used to execute the query.

3. *ExternalDatabase* is a concrete class that allow us to establish a connection.

   Since ExternalDatabase inherits from IDatabaseSQL it must implements _database_connection(...).

4. *ExternalDatabaseAdapter* is a concrete class that allow data from ExternalDatadase to be readable and writable for HistoricalDatabase.

   Since ExternalDatabaseAdapter inherits from IHistoricalDatabse it must implements _read_from(...) and _execute(...) methods.

   Since ExternalDatabaseAdapter has an ExternalDatabase object instance inside, it will have the ability to execute query from it and therefore having the ability to read data from it.

5. *IHistoricalDatabse* is an interface that contains the delcarations of methods that must be implemented by each adapters.

   We expect the read_from(...) mothod to return a fitted content for the HistoricalDatabase.

6. *HistoricalDatabse* is a concrete class that contains methods to execute query into the historical database.

Since HistoricalDatabse inherits from IDatabaseSQL it must implements _database_connection(...).

Since HistoricalDatabse has a IHistoricalDatabase object it can get data from it.

To remember the last item we have read, we store its identifier (ordered) into a sync.json file, and when we will have to execute the next query, we'll read from sync.json the identifier.

The .write_to(...) mothod use the .read_from(...) method of the adapter to get a list of tuple and then insert them into the historical database.

After this, it will commit the changes.

3.3 Being the managerial basis for cost estimation and process management

Infrastructure Cost:	Physical Server owning:	4 x 9000$ = 36000$
	Estimated power:	4 x 250$ = 1000$/month
	Estimated internet **access**:	2250$/month
Developer Cost:	3 Agents:	30000$
	Software developing:	35000$
	Network adapt:	20000$
Algorithmical Cost:	Business logics:	10000$
Maintenance Cost:		20000$/Year
TOTAL COSTS:	One Shot:	131000$
	Running:	57K000/Year

Infrastructure Specs

Physical Server: 4 * ProLiant DL385 Gen10 Plus : 9,000.00

Power costs: 4* 1600W (Server alimentation) = 6400W * 13c$/KwH : 1000$

Internet costs: Dedicated Internet Access, 40Gbps IP Transit: 2250$/month

Developer cost: Based on 6 months working

3.4 Enabling component reuse

```
1 class IHistoricalDatabse(ABC):
2     @abstractmethod
3     def _execute(self, sql): pass
4
5     @abstractmethod
6     def read_from(self, sql): pass
7
```

```
 8 class ExternalDatabaseAdapter(IHistoricalDatabse):
 9     external_databse: ExternalDatabase = ''
10
11     def __init__(self, external_databse: ExternalDatabase):
12         self.external_databse = external_databse
13         print('ExternalDatabaseAdapter has been created')
14
15     def _execute(self, sql):
16         query_res = self.external_databse.execute_query(sql)
17         return query_res
18
19     def read_from(self, sql):
20         query_res = self._execute(sql)
21         return query_res
```

Talking about reusability, as you can see in the uml diagram in chapter 1.2, the not implemented gray parts are just examples of external sources that could be implemented if we will need to read from other services.

Doing so we we'll not have to change each time the way we are going to write into the internal DB, because the adapter will achieve the goal to manage the importing of data inside the script. From here then the data will be passed by the wrapper to the internal DB.

Obviously, it you want to create a new adapter, you will have to code a class which will inherit from IHistoricalDatabase. This allow the historical database, which contains an IHistorical database object, to not change the internal code.

3.5 Allowing a tidy scalability

The system has been ideated and structured to be scalable on the quantity of data received in input from the external sources. This choice born from the unknown of the availability of data during time on the remote source.

Allowing this scalability help the system to be ready for situations where the data are given "real time" and are not retrievable in a second moment. For not losing data will be important for the system to be able to adapt his capability of reading data without losing on performance or reliability.

At this purpose the script will have more internal threads that will be capable of increasing in number when there will be high inbound traffic and then reduce themselves when not anymore necessary.

3.6 Avoiding handover and people lock-in

The problem of vendor lock-in is commonly forcing the company to be stuckked to use some internal technologies that cannot be changed because the cost of the replacement would be higher than the benefits. This problem can also raise on

company's employees when the develop of code has not been well documented and the company find herself to be in the position to cannot replace an employee due to his only known about the project code.

To avoid those problem the solution proposal has been to implement an internal wrapper between the script and the internal DB to allow in a second moment to be able to change internal technologies configurations and from the employees prospective we provide a fully code documentation to let any new future worker on the project be able to understand every snip of code what is doing and how to manage them in case of bug or unexpected behavior.

4 Testing

We have tested the code by creating table **user1** and **user2** with *id* (Primary Key, Auto Increment), *name* and *surname* respectively and then we have tried to read from **user1** and write to **user2**.

We have used sync.json file to store the last tuple that we have read from the **user1**.

Reading this file we were able to resume reading **user1** from the last writing in table **user2**, whitout reading the whole database every time.

For simplicity, we have been using the id of the **user1** table to keep track of the last tuple stored in the sync.json file, and we have read and wrote in the same database instance.

Our tests were performed succesfully.

First execution

```
Connection successfull to the:
          127.0.0.1 test_database root welcome123
ExternalDatabaseAdapter has been created
Connection successfull to the:
          127.0.0.1 test_database root welcome123

Query has been executed:
          SELECT name, surname FROM user where id > 0
          ('federico', 'bruzzone')
          ('andrea', 'longoni')
          ('massimiliano', 'visconti')

Query has been executed: INSERT INTO user2 (name, surname)
                         VALUES ('federico', 'bruzzone');

Query has been executed: INSERT INTO user2 (name, surname)
                         VALUES ('andrea', 'longoni');

Query has been executed: INSERT INTO user2 (name, surname)
```

```
20                          VALUES ('massimiliano', 'visconti');
```

Second execution

The second execution did not write data since in the user1 table there are only three tuples and in our sync.json the counter was setted to three after the first execution.

```
1 Connection successfull to the:
2           127.0.0.1 test_database root welcome123
3 ExternalDatabaseAdapter has been created
4 Connection successfull to the:
5           127.0.0.1 test_database root welcome123
6
7 Query has been executed:
8           SELECT name, surname FROM user where id > 3
```