OpenPaaS Database API Specification

REST VERSION

Contributors:

Rami Sellami, CETIC, Charleroi, Belgium

Bruno Defude, Telecom SudParis, Evry, France

Table of content

1 OpenPaaS DataBase API : ODBAPI

Nowadays, either relational DBMSs or NoSQL DBMSs are on demand and the application developer must be familiar with the proprietary API of each type of DBMS. However, these APIs are heterogeneous on different levels :

- API level: There are two categories of API. In the first category, the operations are not explicit in the API since the access queries to a database are described in strings (i.e. JDBC). In the second category, the operations are explained in the API with a method for each operation (i.e. MangoDB driver).
- Typing level: There are APIs that are closely related to a programming language and manipulate typed objects (i.e. Java JDBC) and others who handle more neutral data structures (i.e. JSON).

In order to satisfy different storage requirements, cloud applications usually need to access and interact with different relational and NoSQL data stores having heterogeneous APIs. This APIs heterogeneity induces two main problems. First it ties cloud applications to specific data stores hampering therefore their migration. Second, it requires developers to be familiar with different APIs.

For this sake, we propose OpenPaaS DataBase API (ODBAPI) a streamlined and a unified REST API enabling to execute CRUD operations on different NoSQL and relational databases. ODBAPI decouples cloud applications from data stores alleviating therefore their migration. Moreover it relieves developers task by removing the burden of managing different APIs. In the upcoming sections, we provide a specification and an implementation of ODBAPI based on a relational DBMS, a document data store called CouchDB, and a key/value data store called Riak.

1.1 State of the art

In some cases, applications want to store and manipulate explicitly their data in multiple data stores. Applications know in advance the set of data stores to use and how to distribute their data on these stores. However, in order to simplify the development process, application developers do not want to manipulate different proprietary APIs especially when interacting with multiple NoSQL data stores. In fact, Stonebraker [1] exposes the problems and the limits that a user may encounter while using NoSQL data stores. These problems derive from the lack of standardization and the absence of a unique query language and API, not only between NoSQL data stores but also between relational data stores and NoSQL data stores.

To rule out these problems, there are nowadays some research works proposing solutions to provide transparent access to heterogeneous data stores. In this context, some of them are based on the definition of a neutral API; others

are based on a framework capable to support access to different data stores.

Developers used to use JDBC [2] for java application in order to interact with different types of relational DBMSs (i.e. Oracle, MySQL, etc.). However, interacting with different types of DBMSs is more complex in the cloud's context because there is a large number of possible data stores, which are quite heterogeneous in all dimensions: data model, query language, transaction model, etc. In particular, NoSQL DBMSs [3] are widely used in cloud environment and are not standardized at all: their data model are different (key/value, document, graph or column-oriented), their query languages are proprietary and they implement consistency models based on eventual consistency.

In this direction, the Spring Data Framework [4] provides some generic abstractions to handle different types of NoSQL DBMSs and relational DBMSs. These abstractions are refined for each DBMS. In addition, they are based on a consistent programming model using a set of patterns and abstractions defined by the Spring Framework. Nevertheless, the addition of a new data store is not so easy and the solution is strongly linked to the Java programming model.

Atzeni et al. [5] propose a common programming interface to seamlessly access to NoSQL and relational data stores referred to as Save Our Systems (SOS). SOS is a database access layer between an application and the different data stores that it uses. To do so, authors define a common interface to access different NoSQL DBMSs and a common data model to map application requests to the target data store. They argue that SOS can be extended to integrate relational data store; meanwhile, there is no proof of the efficiency and the extensibility of their system.

Object-NoSQL Datastore Mapper (ONDM) [6] is a framework aiming at facilitating persistent objects storage and retrieval in NoSQL data stores. In fact, it offers to NoSQL-based applications developers an Object Relational Mapping-like (ORM-like) API (e.g. JPA API). However, ONDM does not take into account relational data stores.

In [7], Haselmann et al. present a universal REST-based API concept. This API allows to interact with different Database-as-a-Services (DaaSs) whether they are based on relational DBMSs or NoSQL DBMSs. They propose a new terminology of different concepts in either types of DBMSs. In fact, they introduce the terms entity, container, and attribute to represent respectively (i) an information object similar to a tuple in a relational DBMS, (ii) set of information objects equivalent to a table, and (iii) the content of an information object. These terms represent the resources targeted by their API. This API enable either CRUD operations or complex queries execution. However, authors remain only to the conceptual model and do not give any details about the implementation level of their API. In addition, their resource model is not generic to each category of DBMS. Haselmann et al. do not deny that proposing such an API is an awkward task. Indeed, they point out in their paper a list of problems

that encounter their API.

Nowadays, NoSQL data stores enter the stage of providing more scalability and flexibility in term of databases in cloud environment. Nevertheless, there is a gap to fill in terms of developer support. Indeed, each type of NoSQL data stores exposes different traits, drivers, APIs, and data models. In most of the time, application developers are lost in this plethora of data stores and they have to manage that by hook or by crook. All that will degrade the developer productivity.

Against this background, ODBAPI can be considered as an innovative and more universal API. First, it takes into account the relational DBMSs and the NoSQL DBMSs. Second, it eases the task of a developers which just need to fully control one API to interact with multiple data stores. Thus, ODBAPI will alleviate the burden of managing several API at the same time. In addition, it separates applications from data stores while migrating an application from a data store to another.

1.2 Concepts comparison

In Table 1, we present an analogy between the concepts terminology in a relational DBMS, a key/value DBMS Riak, and a document DBMS CouchDB. Based on this, we propose our own concepts terminology in order to define the ODBAPI. For instance, a table in a relational database is equivalent to a database in a CouchDB and Riak databases. Hence, to eliminate this heterogeneity, we propose to rename this data structure Entity Set

Table1 : Concepts comparison chart

Relational concepts	CouchDB concepts	Riak concepts	ODBAPI concepts
Database	Environment	Environment	Database
Table	Database	Database	Entity Set
Row	Document	Key/value	Entity
Column	Field	Value	Attribute

1.3 Resources model of ODBAPI

Based on Table 1, we propose the resources model defining the different target resources by ODBAPI (see Figure 1). The main entity in this model is the resource *environment* that is characterized by a name *envName*. An *environment* may contain one or multiple resources *database*. This latter is identified by a unique name *dbName* and contains one or more resources of type *entitySet*. An *entitySet* is determined by a unique name *esName* and has one or more resources of type *entity*. An *entity* is characterized by a unique *entityId* and has

a set of resources of type *attribute*. This latter is identified by a content *attContent* that represents the value of an attribute.

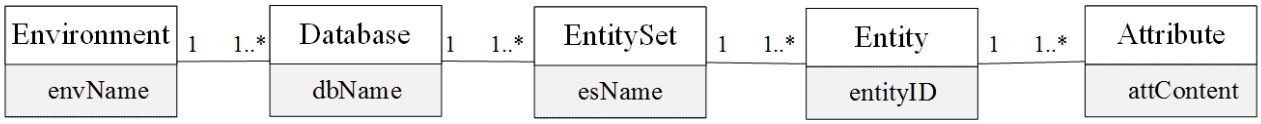

Figure 1 ODBAPI resources model

1.4 Panorama of ODBAPI

In Figure 2, we present a panorama of ODBAPI. This API is designed to provide an abstraction layer and seamless interaction with data stores deployed in a cloud environment. Developers can execute CRUD operations in an uniform way regardless of whether the type of a data store is either relational or NoSQL. In Figure 2, we show mainly four parts that we introduce starting from the right side to the left side. In fact, we have first of all the deployed data stores (e.g. relational DBMS, Couch DB, etc.) that a developer may interact with. Second, we find the proprietary API and driver of each data store implemented by ODBAPI. For instance, we use in our API implementation the JDBC API and MySQL drive to interact with a relational DBMS. The third part of Figure 2 represents the ODBAPI implementation. In fact, this part represents the shared part between all the integrated data stores. In addition, it contains specific implementation of each data store. As we said previously, we propose in this paper a version implementing three data stores: (1) relational DBMS, (2) Couch DB, and (3) Riak by using their drivers and their appropriate APIs. However, to integrate a new data store and define interactions with it, one has simply to add the specific implementation of that data store. Finally, we show the different operations that ODBAPI offers to the user.

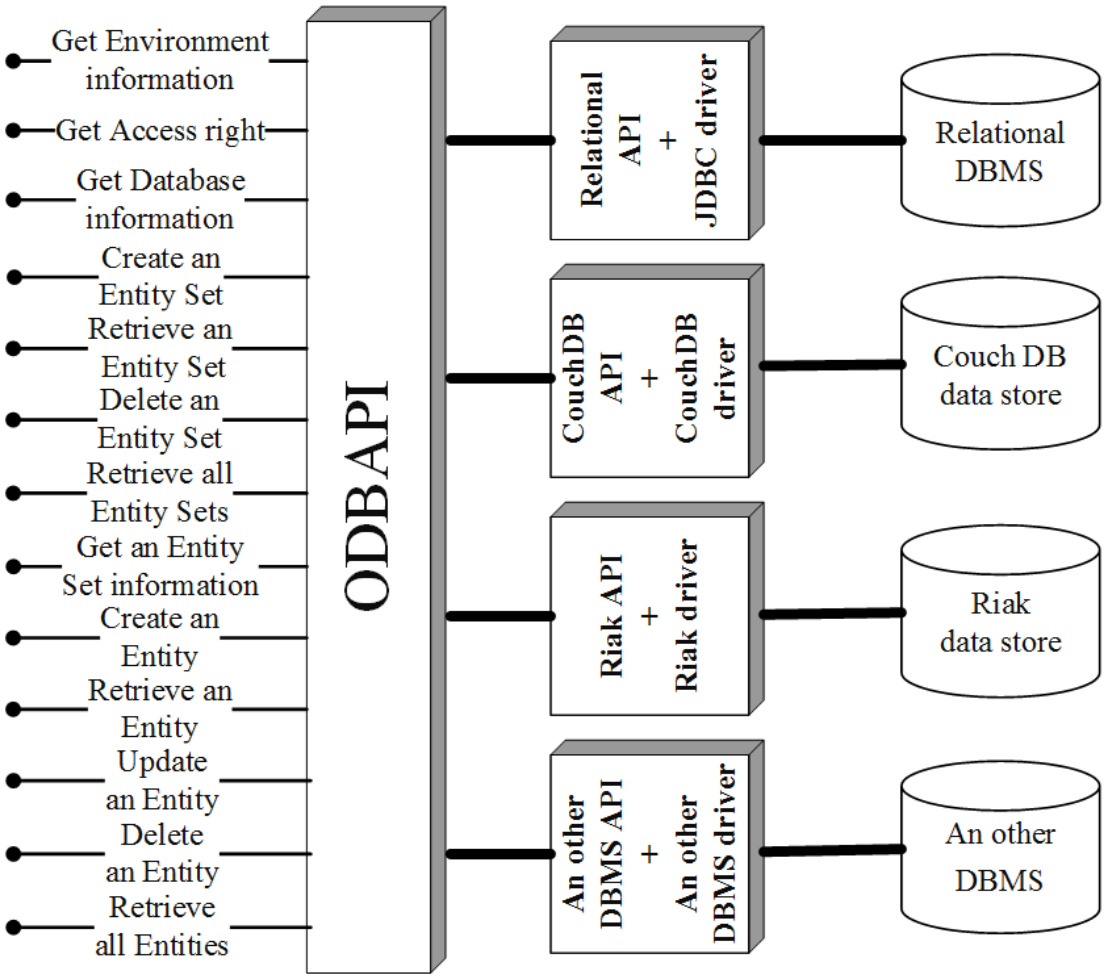

Figure 2 Overview of ODBAPI

In Figure3, we introduce the the protocol architecture of our API. In fact, the client interacts with cloud data stores through the ODBAPI based on REST protocol. Then, according to the target data store ODBAPI interacts with it based on the appropriate API. It is worthly noting that these proprietary APIs are not restful APIs.

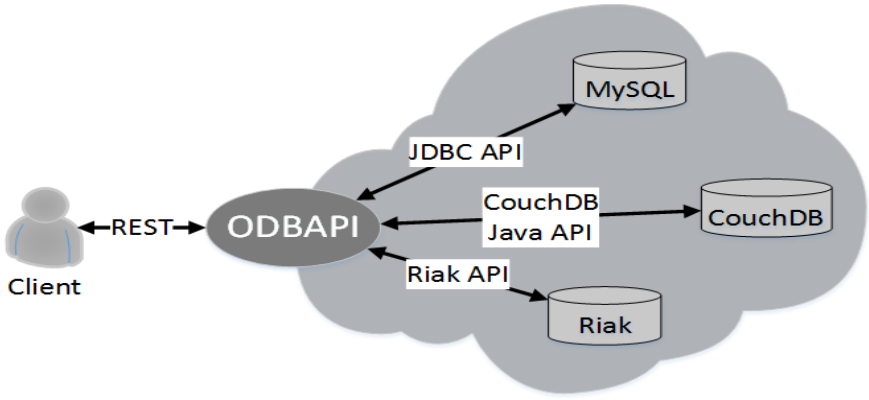

Figure 3 Protocol architecture of ODBAPI

1.5 Operations of ODBAPI

In Figure 4, we propose a box based representation of the different operations ensured by ODBAPI. Each box contains the name of a resource (e.g. /odbapi/{esName}, /odbapi/{esName}/{entityID}, etc) and the different operations that are intended to this resource. Each operation is ensured by a REST method (e.g. GET, PUT, etc).

In our specification, we consider two kinds of operations. The first operations family is dedicated to get meta-information about the resources using the REST method \textit{GET}. In fact, ODBAPI offers four operations:

- Get information about the user's access right: This operation is provided by *getAccessRight* and allows a user to discover his access rights concerning the deployed data stores in a cloud environment. To do so, the user must append to his request the keyword *accessright*. This operation will help the user in choosing the appropriate data store according to his application requirements by listing his access right to databases. The *getAccessRight* operation is linked to the couple user/data store and an user should run it when he use data stores of an *environment* for the first time. In fact, once an user run it, its output is stored in order to avoid its rerun whenever the user interacts with the data stores. However, an user must re-run this operation whether there is some modifications in the data data stores.

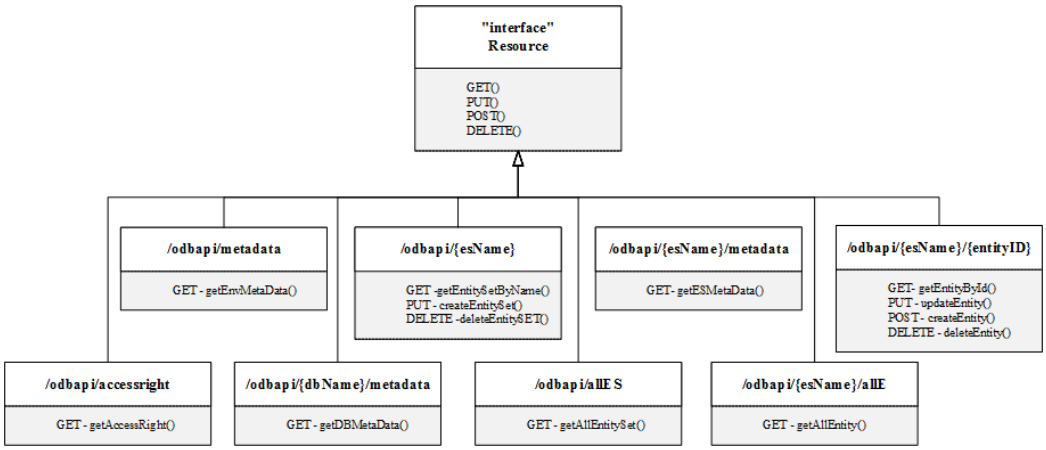

Figure 4 ODBAPI operations

- Get information about an *environment*: This operation is ensured by *getEnvMetaData* and lists the information about an *environment*. To execute this kind of operation, user must provide the keyword *metadata* in his request. This keyword should be also present in the following two operations. Following the execution of this operation, user discovers the deployed data stores in an *environment*. Thus, he will be able to choose the suitable data store. The *getEnvMetaData*

operation is linked only to the different data stores in an *environment*.

- Get information about a *database*: An user can retrieve meta-information about a *database* by executing the operation *getDBMetaData* and providing the name of the target *database dbName*. This operation outputs information about a *database* (e.g. duplication, replication, etc) and the *entity sets* that contains.
- Get information about an *entity set*: This operation is provided by *getESMetaData* and enables to discover information about an *entity set* by giving its name *esName*. For instance, it helps the user to know the number of *entities* that an *entity set* contains and the *attributes* that constitute these *entities*.

Whereas the second operations family represents the CRUD operations executed on resources of type either *entitySet* or *Entity*. In this context, ODBAPI provides nine operations:

- Get an *entity set* by its *esName*: By executing the operation *getEntitySetByName*, an user can retrieve an *entity set* by giving its name *esName*. It is ensured by the *GET* method.
- Create an *entity set*: The operation *createEntitySet* allows an user to create an *entity set* by giving its name *esName*. This operation is provided by the REST method *PUT*.
- Delete an *entity set*: An *entity set* can be deleted by using the operation *deleteEntitySet* and giving as input its name *esName*. It is ensured by the *DELETE* method.
- Get list of all *entity sets*: User can retrieve the list of all *entity sets* by executing the operation *getAllEntitySet* and using the keyword *allES*. It outputs the names of the entity sets and several infomation (e.g. number of entities in each entity set, the type of database containing it, etc.).
- Get an *entity* by is *entityID*: By executing the operation *getEntityById*, an user can retrieve an *entity* by giving its identifier *entityID*. It is ensured by the *GET* method.
- Update an *entity*: An *entity* can be updated by using the operation *updateEntity* and its identifier *entityID*. It is ensured by the *PUT* method.
- Create an *entity*: The operation *createEntitySet* allows an user to create an *entity* by giving its identifier *entityID*. This operation is provided by the REST method *POST*.

- Delete an *entity*: An *entity* can be deleted by using the operation *deleteEntity* and giving as input its identifier *entityID*. It is ensured by the *DELETE* method.
- Get list of all *entities*: User can retrieve the list of all *entities* of an *entity set* by executing the operation *getAllEntity* and using the key-word *allE*. It outputs the identifiers of the *entities* and their contents.

1.6 Detailed specification of ODBAPI

1.6.1 Get information about the user's access right

Identifiant de la ressource	/odbapi/accessright/
HTTP method	GET
Input parameter	Database-Type: Content-Type : Accept:
Response	Information about a user access right in JSON
Status code	200 if OK the error code otherwise

1.6.2 Get information about an environment

Identifiant de la ressource	/odbapi/metadata/
HTTP method	GET
Input parameter	Database-Type: Content-Type : Accept:
Response	Metadata about an environment in JSON
Status code	200 if OK the error code otherwise

1.6.3 Get information about a database

Identifiant de la ressource	/odbapi/{db_name}/metadata
HTTP method	GET
Input parameter	Database-Type: Content-Type : Accept:

Response	Metadata about a database in JSON
Status code	200 if OK the error code otherwise

1.6.4 Get information about an entity set

Identifiant de la ressource	/odbapi/entityset/{es_name}/metadata
HTTP method	GET
Input parameter	Database-Type: Content-Type : Accept:
Response	Metadata about an entity set in JSON
Status code	200 if OK the error code otherwise

1.6.5 Create an entity set

Identifiant de la ressource	/odbapi/ entityset/{es_name}/
HTTP method	PUT
Input parameter	Database-Type: Content-Type : Accept:
Status code	200 if OK the error code otherwise

```
Request
PUT /odbapi/entityset/person/

Database-Type: database/couchDB
Accept : application/json

Response
http/1.1 200 OK
Accept : application/json

The person entity set has been created successfully...
```

1.6.6 Get an entity set by its esName

Identifiant de la ressource	/odbapi/ entityset/{es_name}/
HTTP method	GET
Input parameter	Database-Type: Content-Type : Accept:

Response	Information about an entity set in JSON
Status code	200 if OK the error code otherwise

```
Request
GET /odbapi/entityset/person/

Database-Type: database/couchDB
Accept : application/json

Response
http/1.1 200 OK
Content-Type : application/json

{
    documentCount: 1,
    allDocuments: {
       total_rows: 1,
       offset: 0,
       rows: [1],
       0: {
          id: "1",
          key: "1",
          value: {
             rev: "1-205b04ab704f7d5ff5e0fcc5f302fc41"
                 }
          }
    }
    updateSeq: 12,
    name: "person"
}
```

1.6.7 Delete an entity set

Identifiant de la ressource	/odbapi/ entityset/{es_name}/
HTTP method	DELETE
Input parameter	Database-Type: Content-Type : Accept:
Status code	200 if OK the error code otherwise

```
Request
DELETE /odbapi/entityset/person/

Database-Type: database/couchDB
Accept : application/json

Response
http/1.1 204 No Content
Accept : application/json

The person entity set has been deleted successfully...
```

1.6.8 Get list of all entity sets

Identifiant de la ressource	/odbapi/ entityset/allES
HTTP method	GET

Input parameter	Database-Type: Content-Type : Accept:
Response	List of all entity sets in JSON
Status code	200 if OK the error code otherwise

1.6.9 Create an entity

Identifiant de la ressource	/odbapi/ entityset/{es_name}/ entity/{entityID}
HTTP method	POST
Input parameter	Database-Type: Content-Type : Accept: The content to insert in JSON
Status code	200 if OK the error code otherwise

```
Request
POST /odbapi/entityset/person/entity/1

Database-Type: database/couchDB
Content-Type: application/json
Accept: application/json

{
    "name": "john",
    "surname": "Doe"
}

Response
http/1.1 200 OK
Accept : application/json

Document 1 has been created successfully...
```

1.6.10 Get an entity by its entityID

Identifiant de la ressource	/odbapi/ entityset/{es_name}/ entity/{entityID}
HTTP method	GET
Input parameter	Database-Type: Accept:
Response	The content of an entity in JSON
Status code	200 if OK the error code otherwise

```
Request
GET /odbapi/entityset/person/entity/1

Database-Type: database/couchDB
Content-Type: application/json
Accept: application/json

Response
http/1.1 200 OK
Accept : application/json

{
    _id: "1",
    _rev: "1-205b04ab704f7d5ff5e0fcc5f302fc41",
    name: "john",
    surname: "Doe"
}
```

1.6.11 Update an entity

Identifiant de la ressource	/odbapi/ entityset/{es_name}/ entity/{entityID}
HTTP method	PUT
Input parameter	Database-Type: Content-Type : Accept:
Status code	200 if OK the error code otherwise

```
Request
PUT /odbapi/entityset/person/entity/1

Database-Type: database/couchDB
Content-Type: application/json
Accept: application/json

{
    "name": "Peter",
    "surname": "Doe"
}

Response
http/1.1 200 OK
Accept : application/json

Document 1 has been updated successfully...
```

1.6.12 Delete an entity

Identifiant de la ressource	/odbapi/ entityset/{es_name}/ entity/{entityID}
HTTP method	DELETE
Input parameter	Database-Type: Accept:

Status code	200 if OK the error code otherwise

```
Request
DELETE /odbapi/entityset/person/entity/1

Database-Type: database/couchDB
Accept: application/json

Response
http/1.1 200 OK
Accept : application/json

Document 1 has been deleted successfully...
```

1.6.13 Get list of all entities

Identifiant de la ressource	/odbapi/ entityset/{es_name}/ entity/allE
HTTP method	GET
Input parameter	Database-Type: Content-Type : Accept:
Response	List of all entities in JSON
Status code	200 if OK the error code otherwise

1.7 ODBAPI use cases

1.7.1 First scenario: Application migration from one data store to another

In Figure 5, we exemplify a migration scenario where the Application A needs to migrate from one cloud environment where it interacts with the document data store CouchDB to another in order to meet new data requirements. In the new cloud environment, the application connects to another document data store Mongo DB.

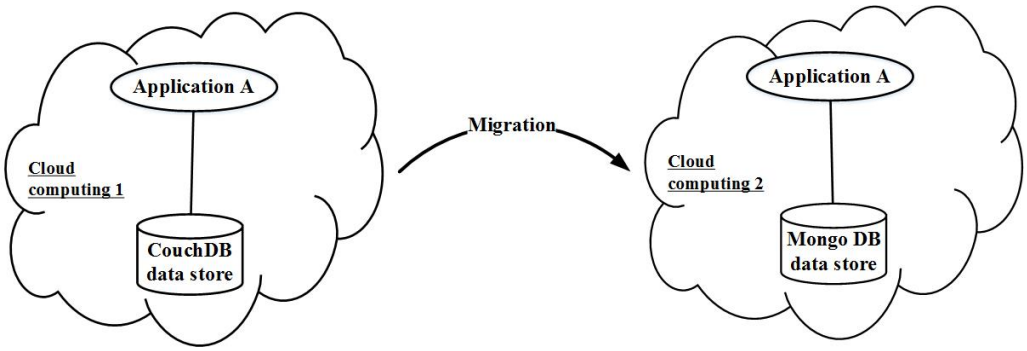

Figure 5 Application migration from on cloud environment to another scenario

In the source code below, we present an example of using ODBAPI in the case of migration from couchDB data store to MongoDB data store. Indeed, the developer creates two entities in a couchDB data store. Then after the migration, developer still using ODBAPI and he has just to replace the old type of the target data store (i.e. *database/couchDB*) by the new target data store (i.e. *database/mangoDB*). Hence, ODBAPI simplifies application migration. In fact, it decouples cloud applications from data stores alleviating therefore their migration.

```
/*****************************************
  Create 2 documents in a couchDB database
*****************************************/
CreateEntitySetImpl ces = new CreateEntitySetImpl();

/*********Insert document 1  in couchDB**********/
InputStream is = new FileInputStream("file/EntitySet1.json");
InputStreamReader isr =new InputStreamReader(is);
JSONTokener tokener = new JSONTokener(isr);
JSONObject jsonEntity = new JSONObject(tokener);
ces.createEntitySet("http://localhost:8182/odbapi/person/1","database/couchDB", jsonEntity);

/*********Insert document 2  in couchDB**********/
InputStream is1 = new FileInputStream("file/EntitySet2.json");
InputStreamReader isr1 =new InputStreamReader(is1);
JSONTokener tokener1 = new JSONTokener(isr1);
JSONObject jsonEntity1 = new JSONObject(tokener1);
ces.createEntitySet("http://localhost:8182/odbapi/person/2","database/couchDB", jsonEntity1);

/*****************************************
 -----<Migration from Couchdb to MangoDB>-----
  Retrive 2 documents in a MangoDB database

*****************************************/
RetrieveEntityImpl re = new RetrieveEntityImpl();

/*********Retrieve document 1  in couchDB**********/
re.retrieveEntity("http://localhost:8182/odbapi/person/1","database/mangoDB");

/*********Retrieve document 1  in couchDB**********/
re.retrieveEntity("http://localhost:8182/odbapi/person/2","database/mangoDB");
```

1.7.2 Second scenario: Polyglot persistence

In a cloud environment, an application can use multiple data stores that correspond to what is popularly referred to as the polyglot persistence. In Figure 6, we show an example of this situation. Application A interacts with three heterogeneous data stores: a relational data store, a document data store that is CouchDB, and a key value data store which is Riak.

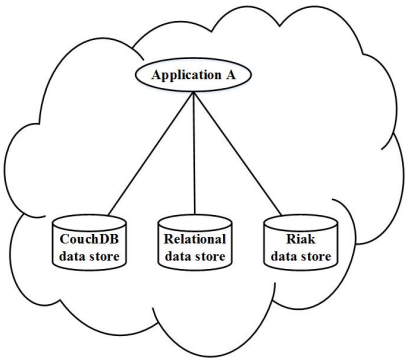

Figure 6 Multiple data store use in one cloud environment

In the source code below, we present an example of using ODBAPI in the case of interacting of two data stores at the same time. These data stores are CouchDB and MySQL.

```
/****************************************
  Polyglot persistence
****************************************/
CreateEntitySetImpl ces = new CreateEntitySetImpl();

/*********Insert document 1 in couchDB database**********/
InputStream is = new FileInputStream("file/EntitySet1.json");
InputStreamReader isr =new InputStreamReader(is);
JSONTokener tokener = new JSONTokener(isr);
JSONObject jsonEntity = new JSONObject(tokener);
ces.createEntitySet("http://localhost:8182/odbapi/person/1","database/couchDB", jsonEntity);

/*********Retrieve document 1 in couchDB database**********/
RetrieveEntityImpl re = new RetrieveEntityImpl();
re.retrieveEntity("http://localhost:8182/odbapi/person/1", "database/ couchDB");

/*********Insert a tuple in MySQL database**********/
InputStream is1 = new FileInputStream("file/EntitySet1.json");
InputStreamReader isr1 =new InputStreamReader(is1);
JSONTokener tokener1 = new JSONTokener(isr1);
JSONObject jsonEntity1 = new JSONObject(tokener1);
ces.createEntitySet("http://localhost:8182/odbapi/city/2 ","database/MySQL","world",
jsonEntity1);

/*********Retrieve a tuple in MySQL database **********/
re.retrieveEntity("http://localhost:8182/odbapi/city/2 ","database/MySQL", "world");
```

References

[1] M. Stonebraker, “Stonebraker on nosql and enterprises,” Commun. ACM, vol. 54, no. 8, pp. 10–11, 2011.

[2] M. Fisher, J. Ellis, and J. C. Bruce, JDBC API Tutorial and Reference, 3rd ed. Pearson Education, 2003.

[3] A. B. M. Moniruzzaman and S. A. Hossain, “Nosql database: New era of databases for big data analytics - classification, characteristics and comparison,” CoRR, vol. abs/1307.0191, 2013.

[4] M. Pollack, O. Gierke, T. Risberg, J. Brisbin, and M. Hunger, Eds., Spring Data. O’Reilly Media, October 2012.

[5] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to nonrelational database systems: The sos platform," in Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings, 2012, pp. 160–174.
[6] L. Cabibbo, "Ondm: an object-nosql datastore mapper," Faculty of Engineering, Roma Tre University. Retrieved June 15th, 2013.
[7] T. Haselmann, G. Thies, and G. Vossen, "Looking into a restbased universal api for database-as-a-service systems," in 12th IEEE Conference on Commerce and Enterprise Computing, CEC 2010, Shanghai, China, November 10-12, 2010, 2010, pp. 17–24.