2014 IEEE International Congress on Big Data

ODBAPI: a unified REST API for relational and NoSQL data stores

Rami Sellami, Sami Bhiri, and Bruno Defude
Computer Science Departement
Institut Mines-Telecom, Telecom SudParis, CNRS UMR Samovar
Evry, France
Email: Firstname.Lastname@telecom-sudparis.eu

***Abstract*—Cloud computing has recently emerged as a new computing paradigm enabling on-demand and scalable provision of resources, platforms and software as services. 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. In this paper, we propose a generic resources model defining the different concept used in each type of data store. These resources are managed by 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.**

***Keywords*-REST-based API, NoSQL data stores, relational data stores, CRUD operations.**

I. Introduction

Over the past years a new paradigm has emerged which is referred to as cloud computing. This latter is defined as a large scale distributed computing paradigm based on virtualized computing and storage resources, and modern Web technologies. Over the internet network, cloud computing provides scalable and abstract resources as services. These services are on demand and offered on a pay-per-use model. Cloud computing is often presented at three levels [1]: the Infrastructure as a Service (IaaS) gives the users an abstracted view on the hardware, the Platform-as-a-Service (PaaS) provides to the developers programming environments and execution environment where proprietary software written in a specific programming language can be executed, and the Sofwtare as a Service (SaaS) enables the end users to run cloud software applications.

Spurred by the cloud computing popularity, researches are increasingly abundant and are focusing on various axes in this area. In this context, data management in the cloud is an inherent research topic that received particular attention recently. A plethora of modern applications and researches such as Bigtable [2], PNUTS [3], Dynamo [4], etc. take into account the data management in the cloud. Nevertheless, they are not sufficient to address all the goals of data management in a cloud environment which are [5]: (i) designing scalable database management architectures, (ii) enabling elasticity and less complexity during databases migration, and (iii) designing intelligent and autonomic Data Base Management System (DBMS). Indeed, an application uses in most cases a single DBMS (or data store) to manage its data. This DBMS is supposed to support the whole needs of an application. Subsequently, it seems illusory to find a single DBMS that efficiently supports various applications with different requirements in terms of data management.

To circumvent this disadvantage, we can imagine a cloud environment with multiple data stores where applications choose one or multiple data stores and interact with it or them simultaneously according to their requirements. The fact that an application uses multiple data stores at the same time is called polyglot persistence [6]. For instance, an application can use a relational data store and a NoSQL data store at the same time or partition its data into multiple data stores.

In a previous work [7], we presented and discussed the requirements of such environment and analyzed current state of the art. We pointed out six requirements among which we tackle two in this paper. The first requirement consists in migrating an application from a data store to another data store either in the same cloud environment or to another one. This requirement necessitates (i) the migration of data from the old data store to the new one and (ii) the adaptation of the application source code to the new proprietary API. Concerning the latter task, the application developer must be familiar with the source code and the new API in order to adjust the application to the new data store. Whereas the second requirement is to enable an easy access to multiple data stores in a cloud environment. At first glance, ensuring this requirement seems to be a simple task; however it is not evident. In fact, each type of data store exposes its own data model, its own query language, etc.; So its own proprietary API. Whence a high level of heterogeneity that is not easy to support and developers are central to it. They must be accustomed with the proprietary API of each data store.

As we can note, these two requirements are closely tied to the data stores APIs. Indeed, the application developer must manage more than one API at the same time either while migrating an application or while interacting with

978-1-4799-5057-7/14 $31.00 © 2014 IEEE
DOI 10.1109/BigData.Congress.2014.98

multiple data stores. It is worth noting that these data stores may be relational or NoSQL. Regardless the scenario, the application developer productivity will degrade due to the APIs high heterogeneity.

To alleviate this burden on the developer, we propose in this paper OPEN-PaaS-DataBase API (ODBAPI) a streamlined and a unified REST-based API. This API enables to execute CRUD operations on relational and NoSQL data stores. The highlights of ODBAPI are twofold: (i) decoupling cloud applications from data stores in order to facilitate the migration process, and (ii) easing the developers task by lightening the burden of managing different APIs. ODBAPI will manage the different resources of our generic resources model. This latter represents the different components that we target with our API.

The upcoming sections are organized as follows. In Section II we introduce the OPEN-PaaS Project in which we propose ODBAPI and two possible scenarios. In Section III, we discuss the related work to our API. In Section IV, we introduce ODBAPI. In Section V, we present three examples of ODBAPI use while in Section VI we summarize the conclusions and outline directions of future work.

II. Use cases and motivation

To show the utility of the ODBAPI API in realistic situations, we present a real use case (see Section II-A). Then, we introduce two possible scenarios of the use of our API: application migration from one data store to another (see Section II-B), and multiple data store use in one cloud environment (see Section II-C).

A. The OpenPaaS project

The OpenPaaS project [1] aims at developing a PaaS technology dedicated to enterprise collaborative applications deployed on hybrid clouds (private / public). OpenPaaS is a platform that allows to design and deploy applications based on proven technologies provided by partners such as collaborative messaging system, integration and workflow technologies that will be extended in order to address Cloud Computing requirements (see Figure 1).

One of the objective of this projet is to specify a unified REST-based API that allows an application deployed in the OpenPaaS platform to interact with relational data stores and NoSQL data stores. The OpenPaaS project focuses in particular on relational, key/value and document data stores. In the following sections, we present two scenarios showing the need to an API as ODBAPI.

B. First scenario: Application migration from one data store to another

Cloud environment usually provides one data store for the deployed applications. However, in some situations, this data store model do not support the whole applications requirements. Subsequently, an application needs to migrate from one data store to another in order to find a more convenient data store to its requirements. It is worth noting that an application may migrate to another cloud environment to find the most suitable data store according to new requirements.

[1]https://research.linagora.com/display/openpaas/Open+PAAS+Overview

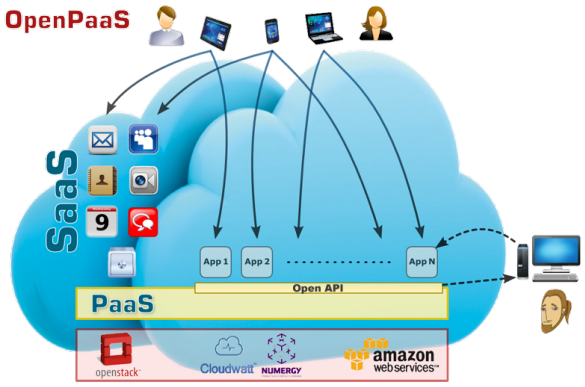


Figure 1. OpenPaaS overview

In Figure 2, 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 an other document data store *Mongo DB*.

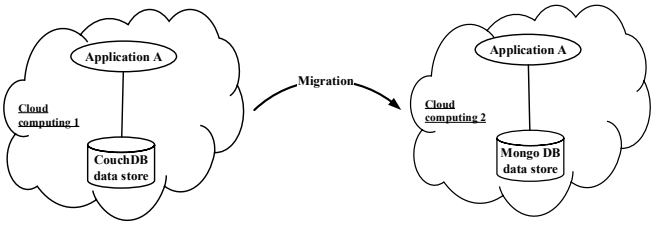


Figure 2. Application migration from on cloud environment to another scenario

At first glance, application migration seems simple and automatic; but behind this scenario there is a tedious and meticulous work to ensure. In fact, the application developer has to re-adapt the application source code in order to interact with the new data store. To do so, he should discover its proprietary API and be familiar with it. In addition, he must migrate data from the old data store to the new one.

C. Second scenario: Polyglot persistence

In a cloud environment, an application can use multiple data stores that corresponds to what is popularly referred to as the polyglot persistence. In Figure 3, 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*.

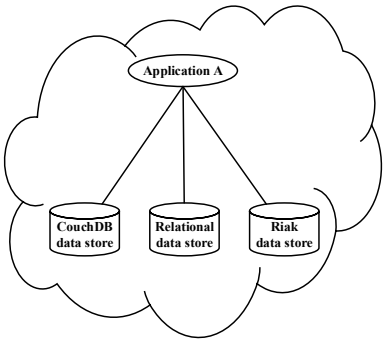


Figure 3. Multiple data store use in one cloud environment

Nevertheless, this scenario presents some limits. Linking an application with multiple data stores is very complex due to the different APIs, data models, query languages and consistency models. If the application needs to query data coming from different data sources (e.g joining data, aggregating data, etc.), it can not do it declaratively unless some kinds of mediation have been done before. Finally, the different data stores may use different transaction and consistency models (for example classical ACID and eventual consistency). It is not easy for programmers to understand these models and to properly code their application to ensure desired properties.

In either scenarios, we see how the developer's task is cumbersome in order to develop and manage the source code of an application. Since, this application interacts with heterogeneous data stores with different API's. Hence, the developer has to be familiar with a plethora of APIs. To deal with this problem, we propose in this paper a streamlined and unified REST API enabling to execute CRUD operations on different NoSQL and relational databases. This API is called ODBAPI and its goal is twofold: First, it simplifies the developer's task during the application migration from one data store to another. By using a unique API, the adaptation of the source code of an application becomes easier. Second, it alleviates the burden of interacting with various data stores at the same time by using just ODBAPI.

III. Related work

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 [8] 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 [9] 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 [10] 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 [11] 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. [12] 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) [13] 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 [14], 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.

IV. OpenPaaS Database API: ODBAPI

In this section, we introduce a generic resource model defining the different concepts used in each category of data store. These resources are managed by ODBAPI. This API enables the execution of CRUD operations on different types of data stores. Doing so, we introduce the ODBAPI resource model (see Section IV-A). Then, we give an overview of ODBAPI and its different operations (see Section IV-B).

A. ODBAPI Resource Model

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

Table I
Comparison chart of concepts used in different data stores

In Table I, we present a comparison between the different concepts used in each DBMS that are used in the context of the OpenPaaS project. We take into account three categories of data stores: *MySQL* a relational data store, *Riak* a key/value data store and *CouchDB* a document data store. For instance, a table in a relational DBMS is equivalent to a database in couchDB and Riak data stores. In ODBAPI, we refer to this concept by *Entity Set*. In addition, the *Entity* concept in ODBAPI represents a row in a relational DBMS, a document in CouchDB and a key/value pair in Riak. In order to organize elements of type *databases* belonging to one cloud environment, we define a new concept that we call *Environment* which includes all resources. For instance, this concept may be a cloud environment.

Based on Table I, we propose the resource model defining the different target resources by ODBAPI (see Figure 4). In the following, we present five resources:

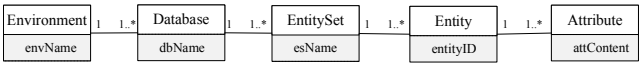
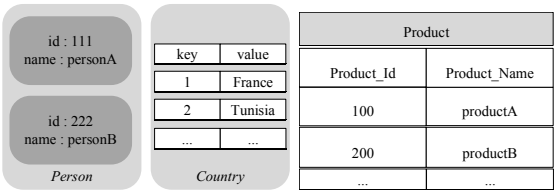


Figure 4. ODBAPI resources model

- The *Environment* resource: The main entity in our model is the resource *Environment* that is characterized by a name *envName*. This resource represents a pool of data stores and an application can choose some of them to interact with. For instance, in Figure 3 the cloud computing containing the three data stores is an *Environment*.
- The *Database* resource: An *Environment* may contain one or multiple resources of type *Database*. A resource *Database* is identified by a unique name *dbName* and contains one or more resources of type *entitySet*. For example, in Figure 3, we exemplify a *relational*, a *CouchDB*, and a *Riak* data stores which are resources of type *database*.
- The *EntitySet* resource: An *EntitySet* is determined by a unique name *esName* and has one or more resources of type *Entity*. For instance, a table in a relational data store is the counterpart of an *EntitySet*. In Figure 5(c), we give an example of an *EntitySet* of type relational table having the *esName Product*.



(a) Example of a *EntitySet* of type couchBD database
(b) Example of a *EntitySet* of type Riak database
(c) Example of a *EntitySet* of type relational table

Figure 5. Exemples of *EntitySet* resources

- The *Entity* resource: An *Entity* is characterized by a unique *entityId* and has a set of resources of type *Attribute*. In Figure 5(a), we give the example of two *Entities* of type *document* identified by their *id*. In Figure 5(b), we exemplify two *Entities* of type *value* identified by a *key* (see Table I). Finally, in Figure 5(c),

we give an example of two *Entities* of type table. They are identified by the *entityId Product_Id*.

- The *Attribute* resource: This latter is identified by a content *attContent* that represents the value of an attribute. In Figures 5(a), 5(b), 5(c), we have respectively the *name*, *value*, and *Product_Name* as *attContents*.

In Figure 6, we present the mapping of the resources depicted in Figures 5(a), 5(b), and 5(c) to our resource model. The *Environment* resource is called *env_1* and contains the three resources: *Person*, *Country*, and *Product*.

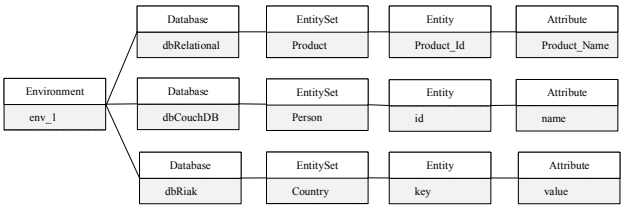


Figure 6. Mapping of the resources depicted in Figures 5(a), 5(b), and 5(c) to our resource model.

B. Panorama of ODBAPI

In Figure 7, 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 a uniform way regardless of the type of the data store whether it is relational or NoSQL. An overview of the API is given in Figure 7. The figure is divided in four parts that we introduce in the following starting from the right to the left side. First, 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 driver to interact with a relational DBMS. The third part of Figure 7 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, Figure 7 shows the different operations that ODBAPI offers to the user.

In Figure 8, 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).

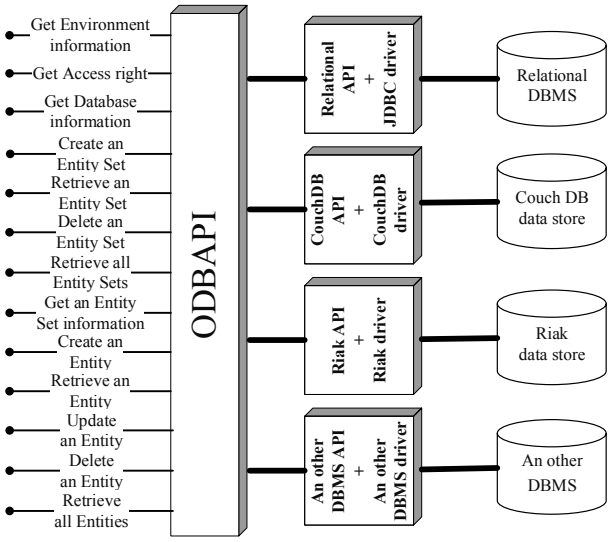


Figure 7. An overview of ODBAPI

In our specification, we consider two kinds of operations. The first operations family is dedicated to get meta-information about the resources using the *GET* REST method. Indeed, 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 a 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 a user should run it when he use data stores of an *environment* for the first time. In fact, once a user run this operation, its output is stored in order to avoid its rerun whenever the user interacts with the data stores. However, a user must re-run this operation whether there is some modifications in the data stores.
- 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*: A user can retrieve information about a *Database* by executing the operation *getDBMetaData* and providing the name of the

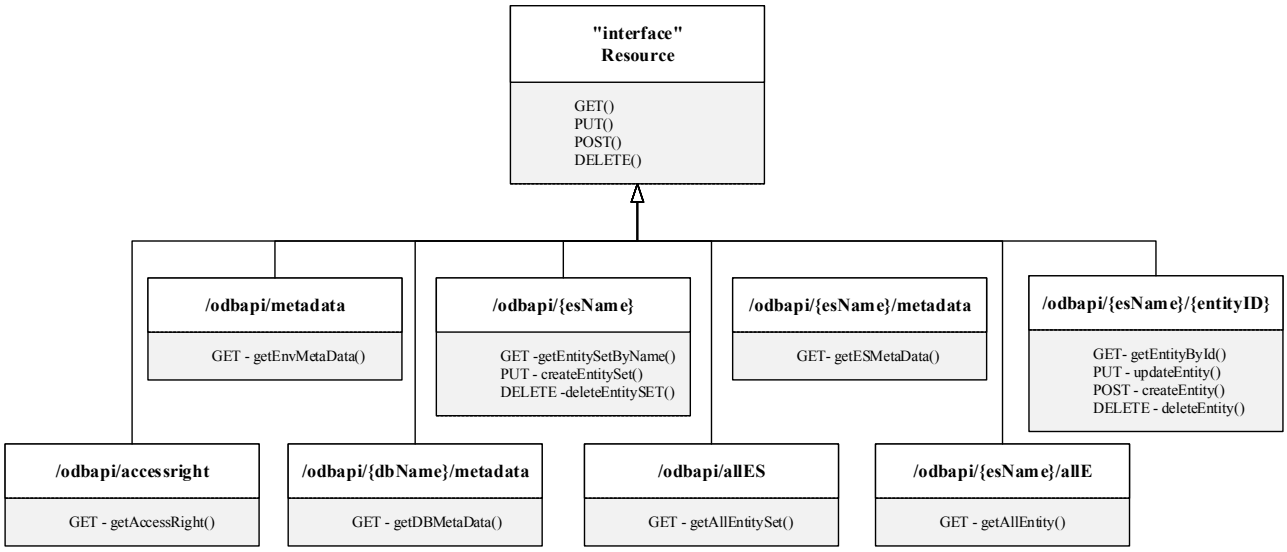


Figure 8. ODBAPI operations

target *Database dbName*. This operation outputs information about a *Database* (e.g. duplication, replication, etc) and the *entity sets* that it contains.

- Get information about an *EntitySet*: This operation is provided by *getESMetaData* and enables to discover information about an *EntitySet* by giving its name *esName*. For instance, it helps the user to know the number of *entities* that an *EntitySet* contains and the *Attributes* that constitute these *Entities*.

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

However, these operations do not work correctly without specifying the target data store. For this sake, we propose to add a new header parameter to the already defined HTTP header parameters. Indeed, we define the parameter *database-type* that allows to precise the type of the target data store by a REST query. For instance, this parameter can have the value *Database/couchDB* to root a query to a CouchDB data store. In addition, the query answering will be based on the implementation provided for this document DBMS.

V. IMPLEMENTATION

According to the requirements of the OpenPaaS project, we provide a version of ODBAPI including three types of data stores: MySQL data store, Riak data store and CouchDB data store. The current version is developed in Java and is provided as a runnable RESTful web application (e.g. jar file). Now we are working diligently on testing ODBAPI using various use cases in the OpenPaaS project so that we identify possible discrepancies and make this

version more stable to use. A description of the realized work is available at http://www-inf.int-evry.fr/~sellam_r/Tools/ODBAPI/index.html. The page contains two links: (i) the first allows to access the ODBAPI specification and (ii) the second allows to download a jar file of ODBAPI.

We are also working on providing a web client for executing CRUD operations based on ODBAPI. This web client will be an access point to the three data stores that we implement. In the two implementations, we use the Restlet framework[2]. Restlet is an easy framework that allows to add REST mechanisms. After adding the needed libraries, one needs just to add the Restlet annotations to implement the different REST actions (i.e., POST, GET, PUT and DELETE). Then to set up the server, one has to create one or more Restlet Components and attach the resources to them. We used the proposed annotations to define the needed actions (see Figure 8) to manage the different resources of our generic resources model (see Figure 4).

We present in the remainder of this section three examples of the same operation targeting the three data stores that we use in our implementation. This operation consists in retrieving an *Entity* by its *entityID*. Hence, we show how ODBAPI unifies the access to heterogeneous data stores to execute CRUD operations.

We present the first example which is a HTTP request and response of retrieving an *Entity* of type document illustrated below. User should specify the type of the HTTP method that is `GET` followed by the target resource `/odbapi/person/111`. Added to that, he should precise the type of the target data store `database/couchDB` and the content type `application/json` that is an acceptable for the response. In our case, it is the JSON format. As a response to this request, we have the status code `200 OK` accompanied by the asked resource written in the JSON format.

```
> GET /odbapi/person/111
> Database-Type: database/couchDB
> Accept: application/json

< HTTP/1.1 200 OK
< Content-Type: application/json
< {
< entityID: "111",
< name: "personA"
< }
```

We illustrate bellow the example example that is a HTTP request and response of retrieving an *Entity* of type value. In this request, the user should specify the key `/odbapi/country/1` of the target value and the type of the target data store `database/Riak` that is the key value DBMS Riak. This request should return the status code `200 OK` and the asked resource written in the JSON format.

```
> GET /odbapi/country/1
> Database-Type: database/Riak
> Accept: application/json

< HTTP/1.1 200 OK
< Content-Type: application/json
< {
< entityID: "1",
< value: "France"
< }
```

The last example is presented below and consists in a HTTP request and response of retrieving an *Entity* of type relational tuple. In fact, user should precise the HTTP method `GET` followed by the resource `/odbapi/product/100` that he wants to retrieve. In addition, he should specify the type of the target data store `Database-Type: database/MySQL` that is the relational DBMS MySQL and the content type `application/json`. In this example, the response of this request is the status code `200 OK` and the requested resource written in the JSON format.

```
> GET /odbapi/product/100
> Database-Type: database/MySQL
> Accept: application/json

< HTTP/1.1 200 OK
< Content-Type: application/json
< {
< entityID: "100",
< productName: "productA"
< }
```

VI. Conclusions and future work

In this paper, we proposed a generic resources model to represent the different elements of heterogeneous data stores in a Cloud environment. We also proposed a unique REST API that enables the management of the described resources in a uniform manner. This API is called ODBAPI and allows the execution of CRUD operations on relational and NoSQL data stores. To do so, we defined a resource model representing the different resources that we target within our API. It is noteworthy that in this version we took into account only three data stores. These latter are a relational DBMS, a key/value data store that is Riak, and a document data store which is CouchDB. In addition, we gave an overview of ODBAPI and its different operations.

[2]http://restlet.org/

This API is designed to provide utmost control for the developer against heterogeneous data stores. ODBAPI eases the interaction with data stores at the same time by replacing an abundance of APIs. Moreover, it decouples cloud applications from data stores alleviating therefore their migration.

The proposed API is not meant as the silver bullet to solve all heterogeneity problems between data stores. However, we proved that we are aware of this problem and we are trying to alleviate this burden by unifying the execution of CRUD operations. We do not deny that it still remains the problem of executing complex queries (e.g. *GROUP BY*, *LIKE*, *JOIN*, etc.). We did not take into account other categories of data stores such as graph data stores and column data stores that we intend to include in our future work. In addition, we did not ensure data transactions with REST architecutre.

We are working on finalizing the web client implementation. Besides, we target applying our API to other qualitatively and quantitatively various scenarios in the OpenPaaS project. This allows us to identify possible discrepancies and make ODBAPI more reliable for public use. Our second perspective consists in providing new implementation for document data stores like Mongo DB and key value data store such as Redis and Berkeley DB. In the long term, we aim to take an interest about complex queries.

ACKNOWLEDGMENT

This work has been partly funded by the French FSN OpenPaaS grant(https://www.openpaas.org/display/openpaas/Open+PAAS+Overview).

REFERENCES

[1] C. Baun, M. Kunze, J. Nimis, and S. Tai, *Cloud Computing - Web-Based Dynamic IT Services*. Springer, 2011.

[2] C. Fay and et al., "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[3] B. F. Cooper and et al., "Pnuts: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[4] G. DeCandia and et al., "Dynamo: amazon's highly available key-value store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles, SOSP 2007, Stevenson, Washington, USA, October 14-17*, 2007, pp. 205–220.

[5] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore, "Database scalability, elasticity, and autonomy in the cloud - (extended abstract)," in *Database Systems for Advanced Applications - 16th International Conference, DASFAA 2011, Hong Kong, China, April 22-25, 2011, Proceedings, Part I*, 2011, pp. 2–15.

[6] P. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, ser. Always learning. Addison Wesley Professional, 2012.

[7] R. Sellami and B. Defude, "Using multiple data stores in the cloud: Challenges and solutions," in *Data Management in Cloud, Grid and P2P Systems - 6th International Conference, Globe 2013, Prague, Czech Republic, August 28-29, 2013. Proceedings*, 2013, pp. 87–98.

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

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

[10] 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.

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

[12] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to non-relational 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.

[13] L. Cabibbo, "Ondm: an object-nosql datastore mapper," *Faculty of Engineering, Roma Tre University. Retrieved June 15th*, 2013.

[14] T. Haselmann, G. Thies, and G. Vossen, "Looking into a rest-based 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.