# Caas - Conversion as a Service

## “Data Layer architecture”

# Authors


- **Yu Kaiwen** - yu.kaiwen.amelie@gmail.com
- **Hamza Sahli -** sh.hamza.90@gmail.com
- **Abdelmoughit Afailal -** Abdou.afailal.1994@gmail.com
- **Hamza Lahbabi -** hamza.lahbabi5@gmail.com


## 1. Introduction

This report will introduce the data layer architecture of our application : what data will this application collect, what is the data structure, what Google tools will be used to store these data. First, here’s a brief introduction about Google tools for storage. Google provides us with several storage options according to our needs, such as:

- Google Cloud Datastore : Schemaless (NoSQL)
- Google Cloud SQL : Relational (MySQL compatible)
- Google Cloud Storage : Files and their associated metadata (Cloud file storage)

For our conversion application we will use two storage options : **DataStore** and **Cloud Storage**.

## 2. Data usage Scenario

At the beginning, the user needs to give his personal google email to be contacted and choose one quality of service. The data collected in this step are:

- The user’s Google email.
- The QoS (quality of service) chosen.

Before the conversion process, each time the user wants to convert a video, he needs to enter the title and duration of it. The data collected in this step are:

- the title of the video
- the duration of the video
- the date (and time) when the video was uploaded (the moment the user confirms his inputs)

During the conversion process, one random file of 1 MByte/second will be created and stored to simulate one video. The data collected in this step are:

- file of 1MB/s, EX: for a video with 1min duration, we will get a video file of 60MB.

## 3. The Data Layer Architecture

According to the data needed to be retrieved mentioned in the previous section, in our data layer architecture, we are using google Datastore. Datastore uses different terminologies than the ones used in relation databases for corresponding concepts:

- **Kind** is the category of the object or the entry (table in RDBMS).
- **Entity** is an object or an entry (row in RDBMS).

- **Key** is a unique identifier for the data entry (primary key in RDBMS).
- **Property** represents individual data (field in RDBMS).

We propose two different kinds of entities: ***User*** and ***Video***.

For the entities of the kind ***User*** have 3 properties:

- ***id***, the identifier of entity generated automatically by google datastore
- ***email***, the email of the user
- ***QoS***, the quality of service chosen by the user

For the entities of kind ***Video***, we define 4 properties and 1 reference:

Entities :

- ***id***, the identifier of entity generated automatically by google datastore
- ***title,*** the title of the video
- ***duration,*** the duration of the video
- ***date,*** the date (and time) in which the video was uploaded to this datastore

reference:

- ***parent***, reference to ***User*** entity to indicate the ***User*** owner of this video

The random files which will simulate the real videos are stored in the google **Cloud Storage**. The file is identified by its filename which should be indicated by us. The filename will be the identifier of the video it simulates with the identifier of the corresponding user. We use the filename to identify uniquely the file and find the information of corresponding video and user stored in the datastore.

- ***filename***, id_entityOfVideo (id of the simulated video) + id_entityOfUser (id of the corresponding user)

The data layer architecture based on datastore that we propose for the CaaS project is given in the Figure below.

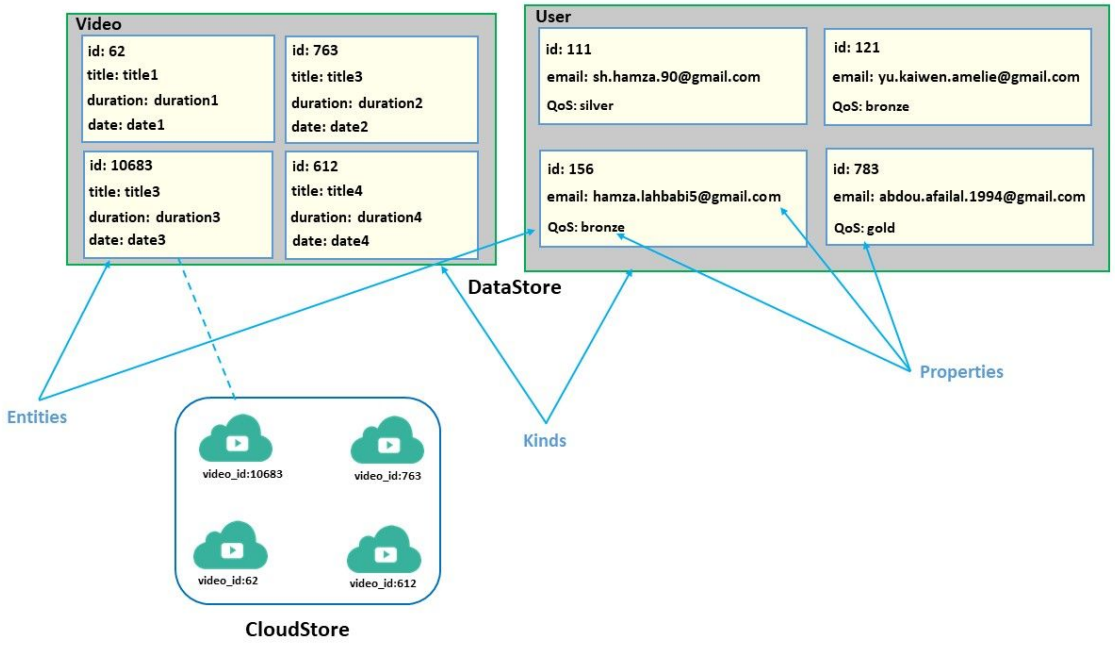

**Figure1. Data Layer Architecture**

In our project, we use the Java data access API "objectify" to persist, retrieve, delete, and query our different objects. So, basically this API can be used to translate Java objects to entities stored in the datastore. Objectify is easier yet more convenient than the low-level datastore API.

The code below shows how we can declare the two classes ***User*** and ***Video*** representing our entities. The annotation ***@Entity*** is used to declare entities***,*** while the annotation ***@Id*** is used to set a primary key which will be generated automatically by the datastore, since we are using the type Long (not String) for the Id. ***@Index*** annotation is used to explicitly define the field that we need to index or to query in our case the user's email. In the objectify API the properties are not indexed by default because *indexes are very expensive to create and to maintain.* Hence, in order to **reduce expenses in our project** we do not index the whole entity or object, we just index the property that we will need. Finally, the annotation ***@parent*** is used to declare a parent relationship between two entities. Hence, we add in our code the ***@parent*** annotation in Video class to the inform it about the User that owes it (parent).

```
import com.googlecode.objectify.annotation.*;

@Entity
public class User {
    @Id Long id;
    @Index String email;
    int QoS;

    private User() {}

    public User (String email, int QoS) {
        this.id = id;
        this.email = email;
        this.QoS = QoS;  } }
```

```
import com.googlecode.objectify.Key;
import com.googlecode.objectify.annotation.*;

@Entity
public class Video {
    @Id Long id;
    @Parent Key<User> owner;
    String title;
    String duration;
    Date date;

    private Video() {}

    public Video(String title, String duration, Key<User> owner) {
        this.title = title;
        this.duration = duration;
        this.owner = owner;
        this.date = new Date(); }}
```

# 4. Justifications

## 4.1. Why we design entities like this?

Asmentioned the section 2, after the user gives his personal email and chooses one quality of service, he can convert as many videos as he wants, so in the data layer, there'll be a batch of videos associated with one user. That's why we store the information of user and the information of video separately.

In this case, one kind of entity can scales out without influent other kinds. For example, when one user continues to convert more videos, we don't need to store the information of user each time. That wastes spaces and spends a lot. We just need to store the information of user one time and associate all of his videos of him.

In this case, it's easier to scale down. For example, when one video has been converted and the time allowed to download has expired, this video needs to be deleted. Our data layer architecture allows us to be easier to find the right video and delete it without worrying about making any influence to the data of user.

## 4.2. Why we use NoSQL instead of RDMS?

RDMS products scale-up. It's expensive to scale for large installations. It hits a ceiling when storage reaches 100s of terabytes. Actually, we can store the list of users, the information of each user, the list of videos of each user and the information of each video into four tables of RDMS. But why we choose NoSQL instead of RDMS? Because we anticipate to offer the conversion service of our application to the users all over the world. In this case, this native cloud application will be a distributed application who will store different parts of data in different places. Each part of data will be stored close to the users of that part in order to have a good performance. This can reduce the time of retrieving data and users' waiting time. RDMS is expensive to scale for large installations. It's hard to duplicate tables and maintain the tables of different places same. NoSQL is better in this case to satisfy the demands. We can store the information of each user and the information of each video

in entities. We don't need to complicate relationship between entities which may create duplication of data to waste storage space. We can easily store entities of users and entities of videos in different places without worrying about the relationship of different kinds of data.

### 4.3. Why we use Datastore and CloudStore?

Basically, Google Datastore is a persistent storage for App Engine data. As mentioned before, our application needs a highly scalable and reliable database that match its scalability, which is very limited in relational databases (very hard to scale and maintain performance and strong consistency at the same time).

**Cloud Datastore** is a NoSQL database that offers great scalability, distribution, highly availability and structured storage technology for Web applications. This technology ensures not only scalability, but also reliability and strong consistency on a single row or eventual consistency on multiple row. This tool automatically manages partitioning and data replication, Cloud Datastore offers a multitude of features, such as ACID transactions, SQL queries, indexes.

It can be used for :

- Semi-structured application data
- Hierarchical data
- Sustainable key-values

**Google Cloud Storage** allows world-wide storage and retrieval of any amount of data at any time. You can use Google Cloud Storage for a range of scenarios including serving website content, storing data for archival and disaster recovery, or distributing large data objects to users via direct download.

It can be used for :

- Images, photos and videos
- Objects
- Unstructured data

We use Google Cloud Storage to store the files which are the simulation of real videos.

### 4.4 How do we ensure lower cost

The first thing we did to reduce cost is to exclude any property that we don't need to query from indexes. As we mentioned in section 3, indexes are very expensive to create and to maintain. If we index properties that we don't need it would increase latency to achieve consistency which will increase also the total storage cost.

We also avoid in our data architecture any composite indexes. The only property we index is the email which we can query to get all the information that we need. Other practices could be avoided or done in order to reduce the expenses such as :

- Avoiding property indexes that could impact datastore latency such indexing properties with increasing values (exp. NOW() timestamp).
- Using key-only query when we need to access only the key from the query results which could lower the latency and also the cost

- Using projection query to access specific properties in a given entity (lower latency, hence, lower cost).
- Avoiding rolling back and doing dual writes operations and parallel operation that could impact datastore cost.

We believe that we have designed our data-layer architecture in a way where we can reduce the datastore cost to a minimum. We also learned the best practices to do and to avoid while writing the code to interact with the datastore in order to reduce expenses.