Omnixell Sync Documentation - 2018

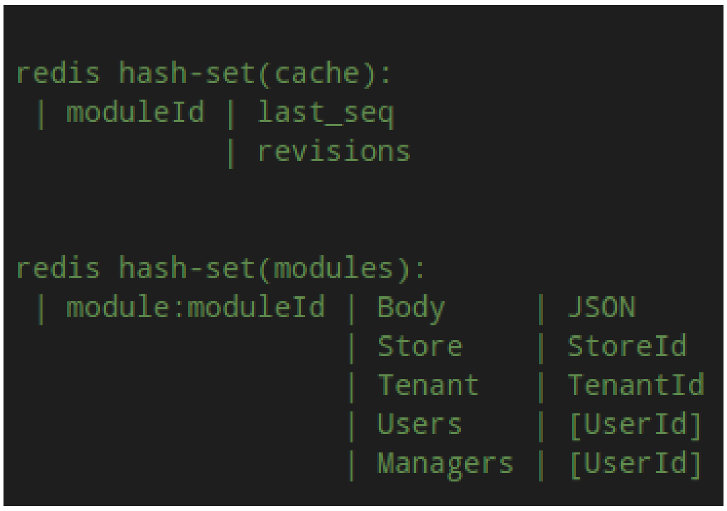
Introduction

This report illustrates the synchronization protocol that was implemented at Omnixell. The project implements a system for the live synchronization of data between the Omnixell server and its clients. The back-end infrastructure is composed of several databases schemes that are not compatible with each other or with the front-end clients. This synchronization feature allows the user to function in a manner which is agnostic to the underlying operations happening on its data/entities. Moreover, by using this feature all the users are guaranteed to have access to their up-to-date data which are kept live through unobtrusive means.

To make it easier for the reader, lets consider two use cases. A user wants to browse his personal calendar through his phone to request for an appointment on a specific day, or perhaps, browse a store to buy an item. Unfortunately, before he finishes the ordeal he has to change to another device, e.g., a tablet. In our system, user's digital data follow him on every device! Despite the change, he can instantly continue on his tablet. In another example, the user adds a product into his cart, but before paying, the store manager decided that this particular product should have it's price reduced. In this case, the client will get the new product and his cart will be updated automatically (The user should be notified about the change from the front-end client, but this is out of the scope of this feature). So, two different users of Omnixell's platform are able to access the same data from different devices with almost real time synchronization and perform some action on it, depending on their privileges.

Underlying components

The redis DB lies at the center of this feature. Redis is used as the cache of the server and holds, except from the documents used, also the metadata needed to implement the sync protocol. To facilitate a faster updating of the documents between the different users, an extra instance is used. This extra instance is necessary in order to propagate the changes that happen on the same type of objects. The image below gives an overview of the way these instances are used. The *cache* instance is responsible for holding, for each object, only the last sequence number used by the replication process and the the history of the revisions in order to handle conflicts. That way, the replication process can easily identify documents/objects that need to be updated because of changes made by a client application (client, manager etc.).The second instance, called *modules*, is responsible for storing the actual object (person, item, manager etc.) along with some frequent-searched identifiers in order to make the retrieval process, and the processing, faster. Any change that happens on an object needs only to propagate in this instance thus eliminating the need for repeating data. One can notice, also, the fields *Users* and *Managers;* these acts as watchers and are responsible for enabling the fast updating of the users' data.



```
redis hash-set(cache):
 | moduleId | last_seq
            | revisions


redis hash-set(modules):
 | module:moduleId | Body     | JSON
                   | Store    | StoreId
                   | Tenant   | TenantId
                   | Users    | [UserId]
                   | Managers | [UserId]
```

Illustration 1: Two redis instances required by the replication protocol

The replication protocol used is taken by CouchDB 2.0 but our implementation does not require this particular database scheme and is reverse engineered (also implemented in JS) to function in an agnostic function. This way, the client application can communicate without any interference even if it expects a couchDB or another database scheme in the back-end.

Setup

Environment

The current implementation is usable with the following node and npm versions:

- Backend Node: 8.9.4
- Backend npm: 5.6.0

Redis steps

1. Download redis from https://redis.io/download and follows instructions there.
2. In the redis root folder
   a. create a folder: instances
   b. create two folders instances/cache and instances/modules
3. copy the following configurations to their corresponding folder:
   a. Cache conf: https://drive.google.com/open?id=1YQLXLGc_zpL6yu9bQbTkkm0Vhyb4MK3P
   b. Modules conf: https://drive.google.com/open?id=1m3AaHdAN6hpyUIAN_YI7KW9OB-1zZ_3t
4. Create the following script in the *instances* folder. The script is responsible for instantiating the redis instances.

```
#!/bin/bashif [ "$1" == "stop" ]
then
   ps all | grep redis-server | grep -v grep | awk '{print $3}' | xargs kill
else
   ./../src/redis-server cache/redis-cache.conf&
   ./../src/redis-server modules/redis-modules.conf&
fi
```

Redis-commander

Redis-commander is a tool that enables the management of redis instances.

1.Download and install it: *$ npm install -g redis-commander*
2.Create a file *.redis-commander* in the home directory of the user with this content:

```
{
  "sidebarWidth": 250,
  "locked": false,
  "CLIHeight": 50,
  "CLIOpen": false,
  "default_connections": [{
    "label": "cache",
    "host": "localhost",
    "port": "5555",
    "password": "",
    "dbIndex": 0
  }, {
    "label": "modules",
    "host": "localhost",
    "port": "5556",
    "password": "",
    "dbIndex": 0
  }]
}
```

Drawing 1: The configuration needed to connect to the two redis instances

Running the project:

Backend side:

1. Run init_redis.sh
2. Run redis-commander -p 8888
3. In the backend folder:
    a. nvm use 8.9.4
    b. npm run start

Frontend side:

1. nvm use 8.9.4
2. ionic serve
3. Ionic serve -p #port

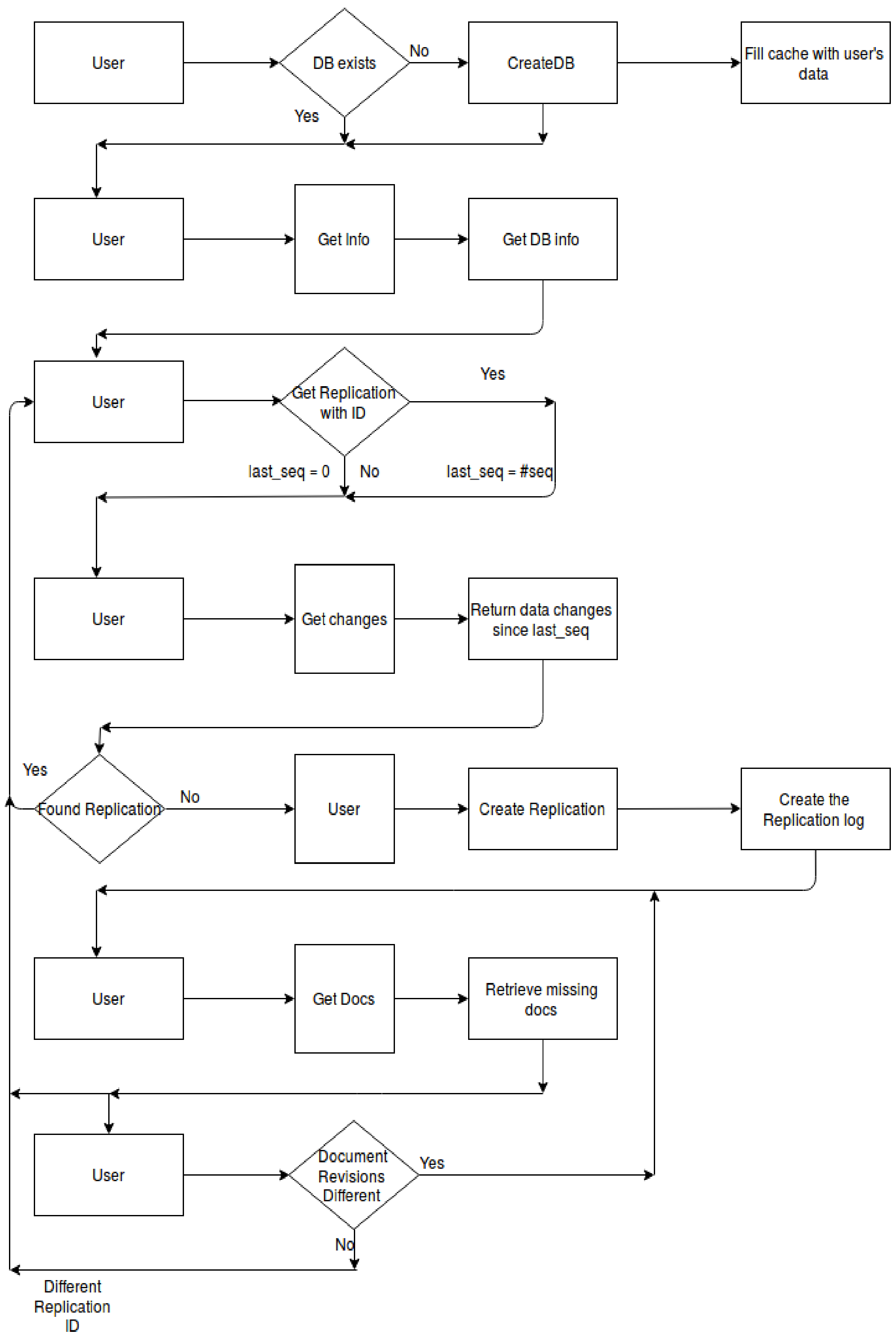
Real-World deployment – Redis Sentinel

In a production environment, for every redis instance there should be another one acting as a “slave” who will replace the “master” instance in a case of unavailability. This instance can be on a different VM/machine to single out cases of machine availability. Redis sentinel can/should be used for this high availability configuration with minimum of three Sentinel instances to ensure no false positives in regards to instance drops.

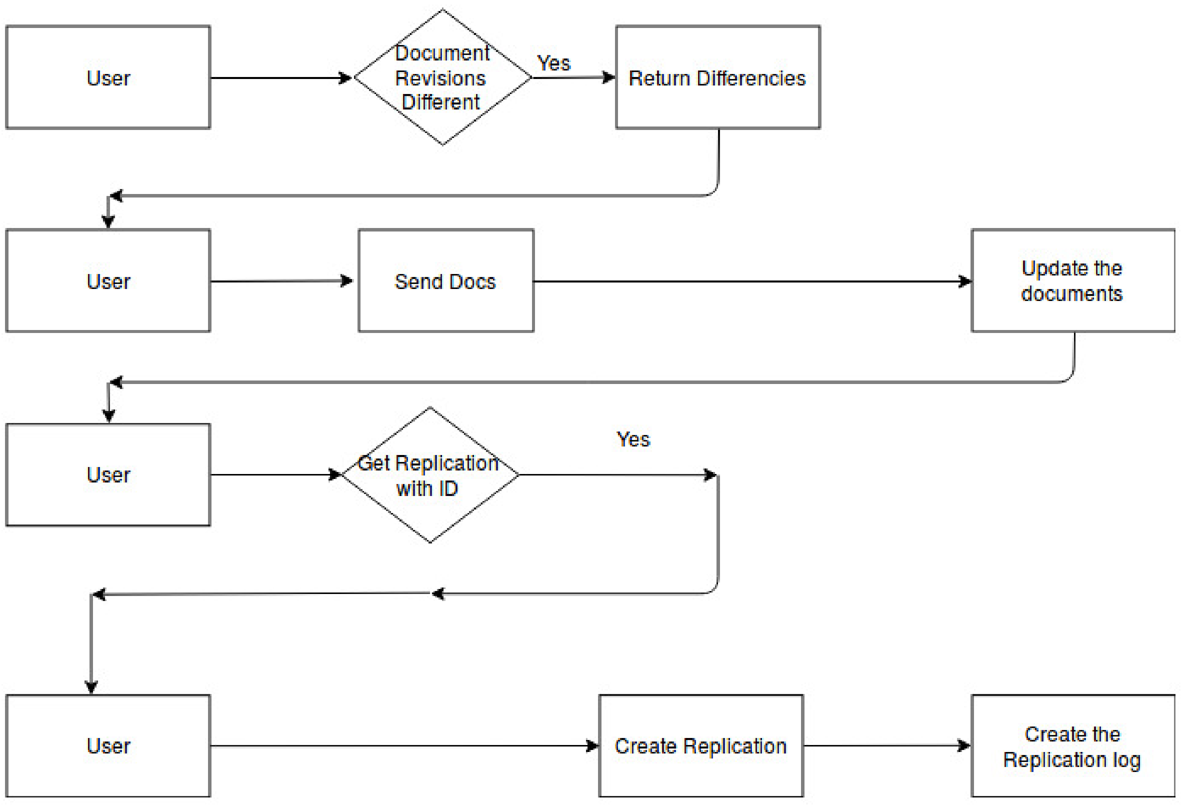
Flow-Charts

We present here two flowcharts in regards to the way the feature functions when a user logs in the system and when he creates/updates an object (add an item to a cart etc.).

Login



Create/Update


User
Document Revisions Different
Yes
Return Differencies
User
Send Docs
Update the documents
User
Get Replication with ID
Yes
User
Create Replication
Create the Replication log