A single-site Key-value store with forwarding

Instructions

This assignment consists of two parts. In the first part, you will build a RESTful single-site key-value store. In the second part, you will create a service that consist of several proxies and instances. They should be able to communicate with the Key-Value store instance.

You will use Docker to create an image which must expose a web server at port 8080 that implements the REST interface below.

As mentioned in the previous assignment, to gain full credit you will need to be part of a team that consists of 3-4 students. You will need to submit a **new repo** for each assignment. **Each repo must contain a "members.txt" file and a "contribution-notes.txt" file, which lists the members and their contributions respectively. The format for these files are in the spec. Please read them carefully and adhere to the instructions exactly for full credit.**

Part 1: Single-site Key-value store

Key-value stores provide a simple storage API in which values (arbitrary data, often semi-structured documents in formats such as JSON) are stored and retrieved by key. The typical KVS API looks like:

Call	**Parameters**	**Returns**	**Example**
put	key, val	Success or failure	put(foo, bar)
get	key	value	X = get(foo) # now X=bar. A GET to a key k should return the **last**value PUT to k, if one exists.
del	key	Success or failure	del(foo) # subsequent gets to foo will fail

You can implement a KVS in whatever way you choose. We will only be testing its adherence to certain functional guarantees, which we enumerate below. As in the last assignment, you will also create a REST API that allows users (as well as our test scripts) to interact with it over HTTP.

Input format restrictions:

- key
- - size: 1 to 200 characters
- val
- - size: 1 MB

Your key-value store does not need persist the data, i.e. it can store data in memory only. For example, if the key-value store goes down and then get started again, then the key-value store does not need contain the data it had before the crash.

Implement your key-value and do not use the existing key value stores like reddis, etc.

Functional guarantees for a single site key-value store:

- service runs on port 8080 and is available as a resource named keyValue-store. i.e. service looks like this : http://server-hostname:8080/keyValue-store
- service listens to port 8080 where requests are sent and responds the requests.
- The keyValue store have four functions:

1. Search function
2. Put function
3. Delete function

- Search function will search for key in the store. If the key exists it returns true, otherwise false.
- Another type of search function is given a key, search for it and return the value.
- Put function adds key-value pair to the keyValue store if the key does not exist and if the key exists updates the value.
- Delete function deletes a key-value pair when given a key.

Part 2: Network of instances with request forwarding

Once you have a single site key-value store working, you will create a network of instances. One instance is the key-value store, which we call the master. All other instances process request by forwarding them to the master.

We are going to create several instances of your container. However, you are just submitting one container, so your service will need to know which either role it needs to play. The role of a container is specified via environment variables.https://docs.docker.com/engine/reference/run/#env-environment-variables

We will set the following ENV variables when running your docker instances:

IP -- the externally-visible ip to which your instance should bind

PORT -- the port to which your instance should bind

MAINIP --is a pair IP:PORT of the main key-value store instance

Then ENV variables for the main instance will look like this:

IP=10.0.0.20

PORT=8080

We will not set the MAINIP variable.

The corresponding ENV variables for a forwarding instance will look like:

IP=10.0.0.21

PORT=8080

MAINIP=10.0.0.20:8080

As an illustration, let our service consist of 3 instance A, B, and C. Instance A is the main key-value store. Instances B and C process requests by forwarding to instance A. When instance B receives request, say get(key1), it queries instance A and returns the value (or error message) it has received from A. If we stop instance B, then it should not affect the service: requests can be successfully processed by instances A or C. However, if we stop instance A, then B and C cannot process put, get and del requests. They should return an error message.

Functional guarantees for a service with forwarding:

- The main instance and the forwarding instances satisfy functional guarantees for a single site key-value store described above.
- If the master or forwarding instance is down, appropriate action should be taken.

Request and Response formats:

0. **Pre-condition -** localhost:port_number forwards to 8080 of the docker container running the keyValue-store service

Note: Here, localhost:49160 is taken only as example, you can use any port number to listen locally. Forwarding should be only on port 8080.

1. **PUT localhost:49160/keyValue-store/subject -d "Distributed System"**

Case: 'subject' does not exist

- status code: 201
- response type: application/json
- response body:

{
'replaced':'False',//'False' since the value did not exist before
'msg':'Added successfully'
}

Case: 'subject' exists

- status code: 200
- response type: application/json
- response body:

{
'replaced':'True'',//'False' if existing value is not updated
'msg':'Updated successfully'
}

If key violates the input restrictions:

{

'msg':'Key not valid'
'result':'Error'
}

If size of object uploaded greater than permitted size:

{
'result':'Error'
'msg':'Object too large. Size limit is 1MB’
}

1. Endpoint 1 : '/' which has function of GET(returns value)
   localhost:49160/keyValue-store/subject

   Case: ‘subject’ exists
   status code: 200
   response type: application/json
   response body:
   {
   ‘result’:‘Success’,
   ‘value’:‘Value’ //Return value for the key
   }
   Endpoint 2: '/search' which has search function(return true or false)
   localhost:49160/keyValue-store/search/subject
   Case: ‘subject’ exists
   status code: 200
   response type: application/json
   response body:
   {
   ‘result’:‘Success’,
   'isExists' : 'true' //Return true if the key exists
   }
2.

1. **DELETE localhost:49160/keyValue-store/subject**

Case: ‘subject’ does not exist

- status code: 404
- response type: application/json
- response body:

{
‘result’:‘Error’
‘msg’:‘Status code 404’
}

Case: ‘subject’ exists

- status code: 200
- response type: application/json
- response body:

{
‘result’:‘Success’,
}

1. **PUT, GET, DELETE**

Case: Main service is down

- status code: 501
- response type: application/json
- response body:

{
‘result’:‘Error’,
‘msg’:‘Server unavailable’
}

Building/Testing your container:

We will formally post test scripts closer to the due date (10/19/2018), ensure your container builds, runs, and responds to the call examples for the methods above during development until then.

It is critical that you run the test scripts before submitting your assignment, as the behavior should be predictable, and the tests we provide are similar to the further tests we will run on our side during grading.

Instructions how to submit your homework:

0. Preconditions - You have a project folder on your local machine.

1. Create a file members.txt. It will contain one member per line, member being the ucsc id. e.g. members.txt:

palvaro
nipashah

2. Create a contribution-notes.txt file. This can be empty until the moment of final submission.
Contribution-notes.txt, for example:
palvaro
Implemented search service
nipashah
Handled integration.

3. Create a local git repo (git init, add, commit)

4. Create a team (Skip if using same team, update members.txt) as specified in last assignment

a. Team ID needs to be unique at bitbucket, **but prefix it with cmps128 e.g. cmps128team1.**

b. Add team members and Cmps128teachingstaff to the repo.

5. Develop as before. **COMMIT FREQUENTLY, we will check contributions against commit history, we would like to see your progress.**

To evaluate a homework submission, I am going to create a docker image using your Dockerfile in your project directory. Then, I will test it by sending GET and PUT request to 8080 port.

Good luck everyone!