Introduction

This package contains the Dockerized implementation of a Redis proxy service as detailed in the Segment take home exercise. It implements all basic and one advanced requirement as detailed in the Segment exercise brief.

Description

There are 2 main parts to this project. The first one is an API built using the Java Spark REST framework that exposes an interface over REST to get the value for a key from its in-memory store. In case that value is not found in the local in-memory cache it queries REDIS to get the key if exists and saves a copy of the same in local cache for cached subsequent GETs.

The local cache implemented follows an LRU eviction strategy such that in case the cache is full and we obtained a new key from Redis – in order to store the same the local cache evicts the key value pair that was the least recently used.

The local cache also adds a global expiry to each node (that can be specified during proxy startup) such that once that duration is passed and a GET request comes for the key that maps to the node it will return 404 NOT_FOUND. *One assumption made here is that the expiry only applies to the local in memory cache and not REDIS. So, after an element is expired if there is another fetch request the value will be obtained from REDIS and re-populated in the local cache. Also, the eviction for the expired key happens only once it is queried for instead of continually checking for the same in order to guarantee better performance. However, this design can be adjusted based on how the consumption pattern for expired key is in general once running in production.*

Lastly the following runtime parameters can be specified while starting the proxy server:

- REDIS_PORT – TCP PORT where backing REDIS is running
- GLOBAL_EXPIRY – Duration in seconds post which a node in the local cache will expire
- CACHE_SIZE – Maximum capacity of the in memory local cache (in terms of number of keys)
- PROXY_PORT- TCP/IP PORT where the proxy listens to.

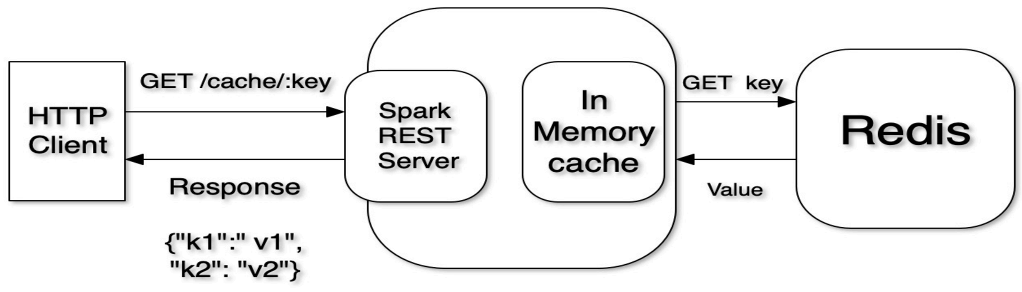
Open Source software Used

- Java Spark REST API Framework for building the REST proxy
- Docker and Docker Compose for deploying and running the app & Redis image
- Apache Maven as the build/run/test tool
- Redis as the backing in-memory key value store
- Jedis as the Java client library to interface with Redis.
- Junit 5 for writing the E2E tests.

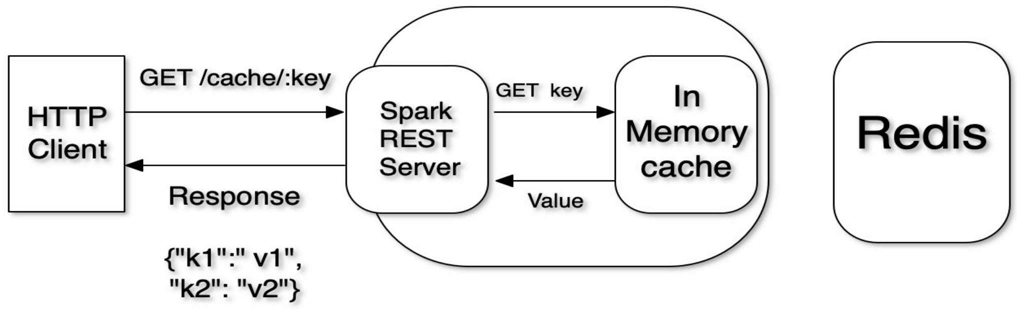
- SLF4J for logging
- wait-for-it.sh for ensuring the different docker containers start in the correct order.

High Level Architecture

- **Query proxy for a key not in local cache and present in REDIS**



- **Query proxy for a key in local cache**



REST APIs exposed

- **API Method and Endpoint**
  GET /cache/:key (where :key is the key for which the proxy server will be queried)

- **Sample Request**
  GET /cache/3f9f245e-4927-11eb-b378-0242ac130002

- **Sample Success Response**
  HTTP Status code – 200 OK
  Response body -- {“name”:”User”, “company” : “Segment”}

- **Sample key Not found response**
  HTTP Status Code – 404 NOT FOUND
  Response body -- {"errorCode":"RESOURCE_NOT_FOUND","details":"Requested resource not found on the server"}

- **Sample Internal Server Error**
  HTTP Status Code – 500 INTERNAL SERVER ERROR
  Response body - {"errorCode":"INTERNAL_SERVICE_ERROR"}

Algorithm

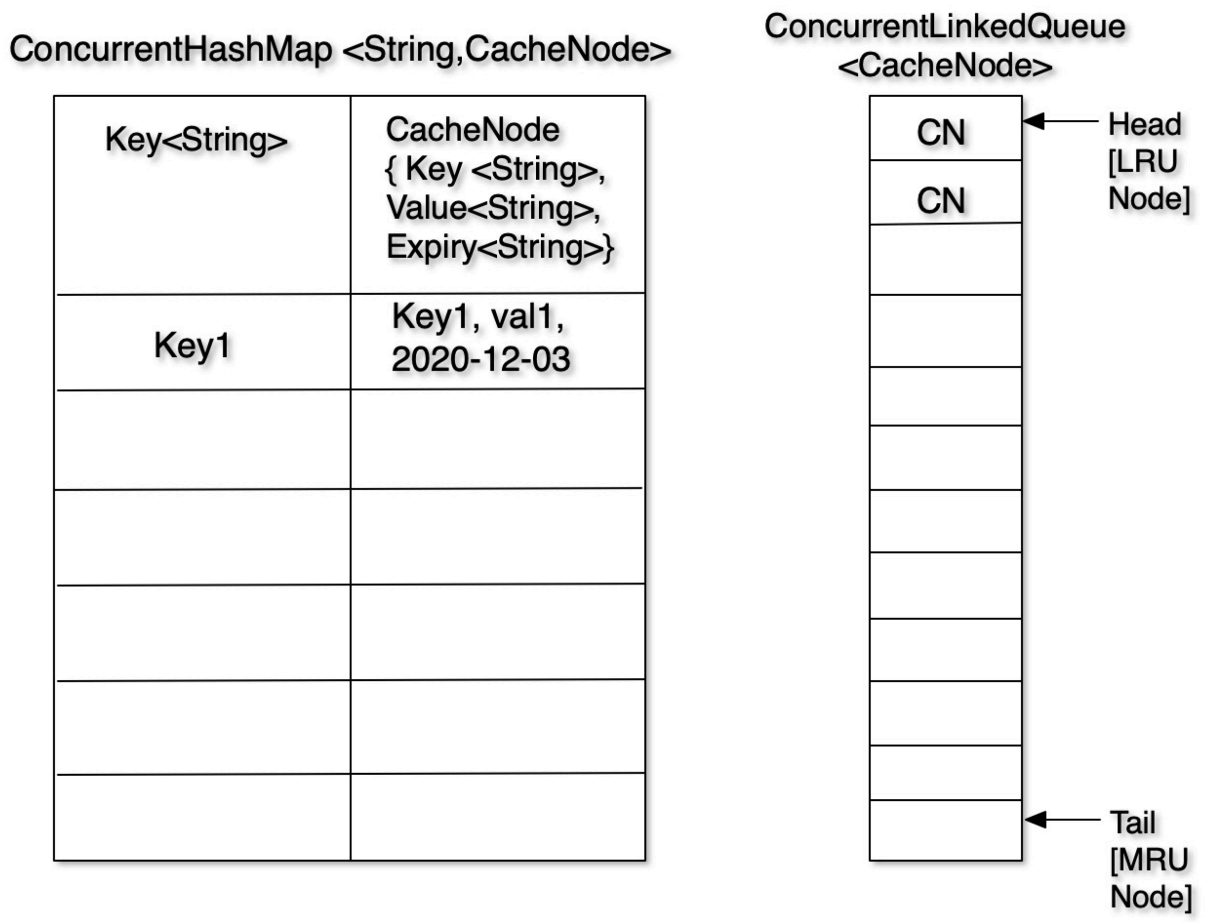
The local in-memory cache has been implemented using the ConcurrentHashMap implementation of the Java HashMap and the ConcurrentLinkedQueue Implementation of the Java Queue.

The map has keys defined as String and are the keys using which the proxy can be queried. The value in the HashMap for these keys are a data structure CacheNode which stores the following bits of information:

- Object **key** of type **String** – this is the key to which the value maps and is needed for the LRU eviction strategy described in detail below.
- Object **value** of type **String** – this stores the value associated with the key and for the purposes of this project is a JSON string.
- Object **expiry** of type **String** – this stores the time when the current CacheNode will expire. This is populated automatically once a cacheNode is inserted based on the Global Expiry configured at service startup.

For the LRU eviction strategy these CacheNodes are also stored in a ConcurrentLinkedQueue which is internally implemented as a doubly linked list. The tail of the Queue tracks the node which was Most Recently Used and the head tracks the Least Recently Used.

The following is a visual representation of the same:

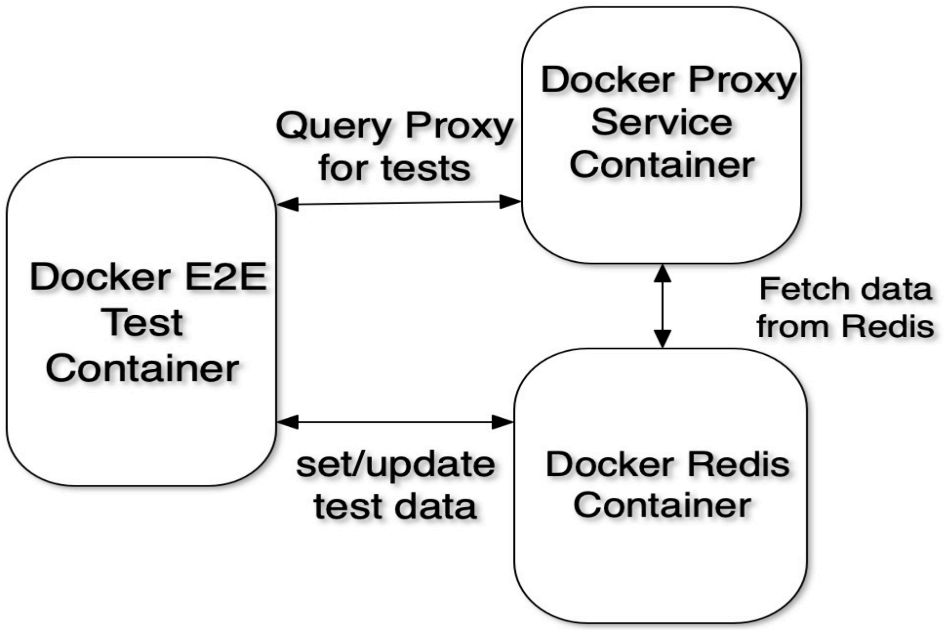

ConcurrentHashMap <String,CacheNode>
Key<String>
CacheNode { Key <String>, Value<String>, Expiry<String>}
Key1
Key1, val1, 2020-12-03
ConcurrentLinkedQueue <CacheNode>
CN
CN
Head [LRU Node]
Tail [MRU Node]

- Every time a new key, value pair is obtained from REDIS we insert it into the Map and also add it to the tail of the Queue since this is the Most Recently Used element.
- Every time a new key that exists in the map is queried for we move that element in the Queue to its tail.
- Once the HashMap is full to its capacity and we want to insert a new node obtained from Redis we remove the CacheNode at the Head of the Queue – find the key it maps to and remove the same from the HashMap as well.
- Currently if the Proxy is queried for a CacheNode after the expiry time stored in the CacheNode we remove that Node both from the Map and the Queue and return a NOT FOUND response back.

- Also using the Concurrent implementations of HashMap and Queue ensure that multiple request threads can be processed concurrently.
- Time Complexity of insert and search – **O(1)**
  Since the hashmap has an algorithmic complexity of O(1) for get and insert and us using a doubly linked list structure also ensures that addition and removal of Nodes can be done in O(1).
- Space Complexity – **O(N)** where N is cache size.

Dockerized Deployment

Once the project directory is entered and ./start.sh script is run it spins up 3 containers using docker-compose – the REDIS server, the proxy server and the e2e test containers.
These 3 containers are on the same network and hence can interface with each other.
The e2e test container runs the Junit tests written for the different use cases and talks to the redis container directly to set and update test data before hitting the proxy container which again talks to the redis container in the backend as needed.



Steps to run the software

- Extract gzip archive **tar -xzvf segment-redis-proxy.tar.gz**
- cd segment-redis-proxy

- Default runtime parameters are specified in the .env file. Please feel free to change there as desired. One important callout is these parameters are common for all 3 containers so a test checking for expiry will always know the duration when a key expires as the same parameter has been passed to both the test container and the proxy service container. Similarly, for others.

- chmod +x start.sh
- sh start.sh

Another thing to note is start.sh will start all 3 containers with the test container running at the end. Once all the tests have been run the test container will exit. However, the proxy server and redis will continue running and you can interact with the proxy server using any API client such as POSTMAN on the port specified directly. For exiting all containers please do Ctrl+C.

Experience And Time Spent

While building the REST API, interfacing with REDIS and tests were something relatively simple as I am used to doing such in my day job – this is the first time I got to learn and use Docker for my own projects directly.
Normally, in my company there is a separate team that maintains the DevOps and deploy infra with us not having to deal with it directly – this project was a really great learning experience for me to get my hands dirty with Docker. And I feel more excited to use it in my future projects directly. ☺
I spent the most time (about 5-6 hrs) understanding how docker and docker compose works and the best way to use it for this assignment. The rest of the development/documentation took around 3-4 hrs of my time.

Requirements Implemented

I implemented all the basic and 1 advanced requirement (the concurrency one using the correct data structures). Since the docker stuff was a new learning for me I wanted to spend more time on that to get it right while also completing the meat of the project -- hence I skipped one advanced requirement. For implementing that I can always modify the current REST API framework to accept TCP messages similar to REDIS and use a similar processing on the backend. Just the interface changes from a conventional REST to RESP.