We investigated 3 different types of caches for our download scenario

1. Redis
2. Default Cache provided by Spring Boot
3. Megha cache

**1.Redis**

1)It is open source key-value , InMemory cache /store.

2)Keys can be of various data types -> String, hashes, lists , sets etc.

3)Dependencies needed to integrate with application

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

</dependency>

**“spring-boot-starter-data-redis”** is for integrating redis and **“jedis”** is one of various java clients we can use to interact with redis.

4)Various clients we can use to interact with redis are

1. Jedis – This is not thread safe, to make it thread safe and to work in multithread environment , we can create JedisConnectionPool
2. Lettuce – This is thread safe by default

Dependency to be added if you are opting for lettuce

<dependency>

<groupId>biz.paluch.redis</groupId>

<artifactId>lettuce</artifactId>

<version>3.2.Final</version>

</dependency>.

5) Install Redis on Mac and test if its working

<https://medium.com/@petehouston/install-and-config-redis-on-mac-os-x-via-homebrew-eb8df9a4f298>

6) Sample program for using Jedis and Redis .

Below program has 3 options to connect to Redis using

1.JedisPool

2.RedisClient

3.JedisConnectionUsing URI and SSL

**import** java.net.URI;

**import** com.lambdaworks.redis.RedisClient;

**import** com.lambdaworks.redis.RedisConnection;

**import** com.lambdaworks.redis.RedisURI;

**import** redis.clients.jedis.Jedis;

**import** redis.clients.jedis.JedisPool;

**import** redis.clients.jedis.JedisPoolConfig;

**import** redis.clients.jedis.Protocol;

**public** **class** CustomJedisConnectionPool {

**public** **static** **void** main(String[] args) {

//connectUsingJedisPool();

*connectUsingRedisClient*();

}

**public** **static** **void** connectUsingRedisClient() {

RedisClient redisClient = **null**;

RedisConnection<String, String> connection = **null**;

**try** {

redisClient = **new** RedisClient(

RedisURI.*create*(“redis://localhost:6379”));

connection = redisClient.connect();

System.***out***.println(“Pinged using redisClient “+connection.ping());

}**catch**(Exception e) {

e.printStackTrace();

}**finally** {

**if**(connection != **null**)

connection.close();

**if**(redisClient != **null**)

redisClient.shutdown();

}

}

**public** **static** JedisPool getJedisPool() {

JedisPool pool = **new** JedisPool(**new** JedisPoolConfig(), “localhost”, 6379, Protocol.***DEFAULT\_TIMEOUT***);

**return** pool;

}

**public** **static** **void** connectUsingJedisPool() {

JedisPool pool = *getJedisPool*();

Jedis redis = **null**;

**try** {

redis = pool.getResource();

System.***out***.println(redis.ping());

redis.set(“CustomKey”, “CustomValue”);

System.***out***.println(redis.get(“CustomKey”));

//redis.flushAll(); // to flush all data from all DB's in redis

System.***out***.println(redis.dbSize());

}**catch**(Exception e) {

e.printStackTrace();

}**finally** {

**if**(redis != **null**) {

redis.close();

pool.destroy();

}

}

}

**public** **static** **void** connectionUsingSSL() {

//Below both will work,but second one is using SSL

//Jedis jedis = new Jedis(URI.create("http://localhost:6379"));

Jedis jedis = **new** Jedis(URI.*create*("rediss://localhost:6379"));

System.***out***.println(jedis.ping());

jedis.close();

}

}

7) Reference docs for using redis with Spring boot

<https://docs.redislabs.com/latest/rs/references/client_references/client_java/>

<https://redislabs.com/lp/redis-java/>

<https://github.com/ThirstyBrain/spring-boot-redis-cache-demo-project>

And also official redis doc - <https://redis.io/documentation>

8) Pros

1. Redis has flexible data structures, it supports almost all data structures.

2. Since its in-Memory cache, its faster

3.Redis allows storing key and value pairs as large as 512 MB.

4.Open Source and stable.

5. cache data expiry time , TTL other things can be configured

9) Cons

1.Since Data is sharded based on the hash-slots assigned to each Master. If Master holding some slots is down, data written to that slot will be lost.

2. Still have to check on redis cluster setup, handling cache nodes failure etc.

10) We can intergrate redis using **Kubernates cluster environment.** Some teams in Walmart have used it and have integrated using kubernates cluster.

**2. Spring boot default cache**

1) Even this is key-value store.

2) dependency to be added

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>

1. Spring boot with above dependency provides, following annotations for handling cache operations which can be added on top of methods in java classes

* @Cacheable – To add data to cache
* @CachePut – to update cache data
* @CacheEvict – to delete cache data
* @Caching – to add Caching on class level instead of method level
* @EnableCaching – Without this annotation, caching will not be enabled

4) Examples for same can be found in

<https://www.baeldung.com/spring-cache-tutorial>

<https://www.mindbowser.com/spring-boot-with-redis-cache-using-annotation/>

<https://www.youtube.com/watch?v=Pj8Q_9dovgY&ab_channel=GreenLearner>

<https://www.youtube.com/watch?v=Pj8Q_9dovgY&ab_channel=GreenLearner>

1. Yet to check on Pros and Cons

**3) Megha Cache**

1) MeghaCache is the Strati Data Foundation’s caching solution. Even this is key-value store, distributed cache

2) It internally uses Memcache

3) It has Two Primary Components

* Smart Proxy - The smart proxy is McRouter, which is an open source client originally developed by Facebook. Client applications connect to McRouter using the standard Memcache ASCII protocol on port 5000. The smart proxy internally connects to one or more memcached server(s)
* Cache Server - The backend cache servers uses open source memcached, which is a general-purpose distributed memory caching system.

1. Connecting to Meghacache can be done using any one of the 2 Java client libraries

* ForkLift 3 Java Client or
* Spymemcached java client
* It can be connected from even nodeJs environment , Python and .Net clients.

5) More Information about the clients can be found in below link

<https://etools.helpdocs.io/l/en/article/en2btre5qb-megha-cache-client-libraries>

6) Example using ForkLift 3 Java Client can be found in below link

<https://gecgithub01.walmart.com/data-access-frameworks/strati-af-persistence-training/tree/master/fl3-cache-example>

But did not find any examples of using Spymemcached java client for connecting with Megha Cache.

7) Dependencies to be added

<dependency>

<groupId>io.strati</groupId>

<artifactId>strati-af-javase-bom</artifactId>

</dependency>

<!-- Strati Forklift -->

<dependency>

<groupId>io.strati</groupId>

<artifactId>strati-af-forklift-bom</artifactId>

</dependency>

<dependency>

<groupId>com.walmart.strati.forklift</groupId>

<artifactId>forklift-meghacache</artifactId>

</dependency>

<dependency>

<groupId>com.walmart.strati.forklift</groupId>

<artifactId>forklift-cache-impl</artifactId>

</dependency>

7) Performance of Meghacahe has already by calculated by Starti team. Below link has details

<https://etools.helpdocs.io/l/en/article/458ea4rpac-megha-cache-performance>

Note: Among all 3 caches, best suited cache for our application needs to be discussed and finalised.

8) Requesting and Setting up Cache cluster for dev and prod environments and its **cost**

<https://confluence.walmart.com/display/PLDF/How+to+provision+a+MeghaCache+cluster+via+HUB+portal>