Assignment: RedisCache (Using Redis as a Server-Side Cache)

Avraham Leff

COM3580: Spring 2024

1 Assignment-Specific Packaging

The general packaging is unchanged from the basic "Homework Requirements" (see slides from first lecture and "Homework Policies for COM 3580" on Piazza).

This assignment's "DIR" **must be named** *RedisCache*. You will not create a screen-shot based "writeup file" because you'll be providing a testable implementation ☺.

2 Motivation

The purpose of this assignment is give you experience with:

- Installing and using Redis
- Using Redis to solve a specific business problem

As discussed in lecture, "*Redis provides data structures such as strings, hashes, lists, sets, sorted sets.*". Applications using Redis can associate *keys* with any of the above data-structures (and others as well, see the documentation). Successful use of Redis requires that you pick the right data-structure to model the appropriate piece(s) of your business logic and then assemble the pieces together.

In my opinion, in contrast to e.g., SQL or MONGODB, there is little point in a "pure C/R/U/D" Redis drill. Instead: this assignment specifies *application business logic*, and then asks you to provide an implementation of the requirements **in Java** (for the business logic) and use **only Redis data-structures** to maintain internal state.

> All program state **must be maintained in Redis**! You may not maintain state in any Java primitive or data-structure. As client code retrieves data maintained in Redis, you are allowed to temporarily convert that state into e.g., Java arrays so as to satisfy the method signature. Similarly, the conversion from Redis data-structures to the specified API can use e.g., Java loop structures.
>
> You'll use Jedis (see below) to provide the Java bindings for interacting with the Redis server.

3 Background

Many web-applications are designed to be *stateless*: i.e., the web-server doesn't store information about the clients with which it's interacting. Instead, state is maintained in a database, and the web-application queries the database as needed. The approach has important advantages: it facilitates a simple design and makes it easy to add more front-end servers to the application. One disadvantage: high volumes of C/R/U/D interactions with the database can degrade performance.

Caching is the classic technique for dealing with such issues, and as discussed in lecture, Redis-as-a-cache is well known design pattern. In this assignment, you will use Redis to cache various pieces of state.

3.1 Cookies

See this article for an overview of web-servers use of *cookies*. For this assignment, the key points are:

- The implementation uses a sequence of random bytes as the "cookie data" (the contents won't matter here).

- On the server, the cookie is used as a key to see whether a user is currently associated with the token. The key is used (conceptually) to offload state derived from queries that are currently directed to a (e.g., relational) database to get user state. Your code will not, however, interact with any data-store other than Redis for this assignment.

  In this assignment, the cookie is associated with a user and the **items** that have been viewed by that user. If the number of items viewed by the user exceeds a given threshold, the server will remove the older items from the association.

- Over time, old cookies can be deleted to make room for new ones.

3.2 Shopping Carts

- Users can be associated with *shopping carts* using the same "user-to-cookie" association described above.

- A shopping cart maintains a set of "item, quantity" duples.

- When user state is removed from the cache (see "Cleaner Threads" below), their shopping cart state is removed as well.

3.3 Cleaner Threads

Over time, client interaction with the web-applications can create considerable amounts of cache state. A *cleaner thread* is a process that is periodically invoked to removed "old" state. Cleaner threads can be run as an ongoing daemon process, a cron job, or even embedded in functions directly invoked by the client. For this assignment, the cleaner thread will be invoked explicitly by clients (to facilitate testing). The cleaner removes "older" user state (including all state associated by that user).

Note that Redis has a set of EXPIRE APIs that could be used to automatically age out cached data. This assignment uses the cleaner threads approach because of advantages such as not expiring data automatically (i.e., keeping control of the expiry decision).

4 (Possibly) Useful Tips

The following information may be helpful to you ...

- The "Epoch & Unix Timestamp Conversion Tools" is available here.
- Even though Redis is a "main-memory" database, it periodically saves its current state to disk. If you need to execute your program with a "clean slate", investigate the use of `FLUSHDB` and `FLUSHALL`.

5 Requirements

> Please review the general requirements for a programming assignment (on Piazza)! I've tried to reduce the chances of "mistakes" occurring through the use of a "skeleton interface", but ultimately, **you are responsible** for ensuring that I can compile and test your code without incident.

Many language bindings exists to enable interactions with Redis: because this is a Java assignment, you'll use Jedis. I'm specifying the API as a base class named `RedisCacheBase`: you will implement the API in a class named `RedisCache`.

- Your implementation may **only use** the JDK and *Jedis, **v5.1.1***.

> If you feel that you need additional maven dependencies to do the assignment, speak to me directly, and I'll consider allowing those as well. Without explicit permission, you may **only use** what I've specified above.

- All "time-related" operations (e.g., "maintain the most recent") **must be done** in micro-second accuracy, not e.g., milliseconds!
- I must be able to create a cleaner thread instance thusly:

```
CleanerThreadBase task = new RedisCache.
    CleanerThread(limit);
```

1. Begin by downloading a skeleton `RedisCacheBase.java` class from this git repository.
2. Implement the stubbed methods of *RedisCacheBase* in *RedisCache* which **must extend** `RedisCacheBase`. Implement a *CleanerThread* class that extends `CleanerThreadBase`. The implementation classes must have "public" access.
3. Remember to check the base class into your Git repo.

6 Sample API Usage

```
import redis.clients.jedis.Jedis;
import edu.yu.mdm.RedisCacheBase;
import static edu.yu.mdm.RedisCacheBase.*;
import edu.yu.mdm.RedisCache;

final Jedis conn = new Jedis();
final RedisCacheBase rcb = new RedisCache(conn);

final int limit = 0;
CleanerThreadBase task = new RedisCache.CleanerThread(
    limit);
final ExecutorService executorService = Executors.
    newSingleThreadExecutor();
Future<Integer> future = executorService.submit(task);
```

7 Grading

Your implementation will be graded on both "correctness" and (to a lesser extent) "performance". As usual, it's difficult to define "good performance" in a way that will precisely guide development on your machine. To give you a sense of my expectations, on my *MacBook (M1) Pro*, with 10 cores and 32 GB of memory, individual test method duration (including logging)

is considerably less than ten ms, and the test suite as a whole takes **110** ms. Points will be deducted if your implementation takes significantly more than my expectations.

Note: If your implementation does excessive amounts of printlns, your performance will suffer! I set the log4j2 trace setting at “INFO” when I run your code.