High Availability with Redis Sentinel and Spring Lettuce Client



[Naciye Karademir](https://medium.com/@nkaradem?source=post_page-----9da40525fc82--------------------------------)

Follow

[Jun 10, 2020](https://medium.com/trendyol-tech/high-availability-with-redis-sentinel-and-spring-lettuce-client-9da40525fc82?source=post_page-----9da40525fc82--------------------------------) · 7 min read



**RE**mote **DI**ctionary**S**erver

Redis is a very popular, open-source, and very fast in-memory key-value data store. Unlike others, it offers a variety of data structures to store your data. You can use Redis for caching, data storage, or as a message broker. If you want to read more detail about Redis, I highly recommend [Redis documentation](https://redis.io/documentation).

At Trendyol Tech we use Redis very frequently in our applications. In the Delivery team, we use it for session management for third-party companies and cache the most used data by the domain applications.

Since our applications are processing thousands of requests per second, we need to make sure that our data storage is highly available. In this article, I will explain the high availability solution of Redis and give a quick demo about how to configure Sentinel and use it in Redis recommended Java client Lettuce with Spring Data.

Redis Sentinel

What is Redis Sentinel?

Redis Sentinel is the high availability solution of Redis. It provides this by providing 3 things:

* Monitoring
* Automatic failover
* Configuration provider

Redis Sentinel will monitor and automatically detect a master failure and bring all clusters to a stable mode. This makes sure that human interaction will not be needed in case of a fail.

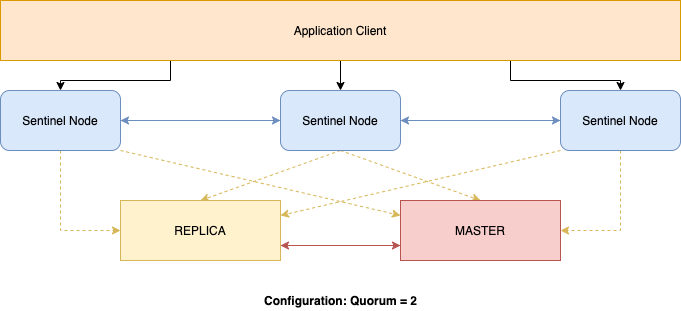
Sentinel decides if the master went down or not. If it is not reachable, Sentinel will choose a replica, which are exact copies of the master, and promote it until the master comes back.

Fundamentals of Sentinel

[Sentinel documentation](https://redis.io/topics/sentinel) states that at least three sentinel instances are required for robust deployment. Although they can run as parallel processes in the same Redis instance, they should be placed into servers or VMs that are believed to fail independently.

Sentinel constantly checks the master for a failure. If enough sentinel agrees that the master is down then it acts as an authority and will start a failover process. As a result, it will promote a replica to be the master, configure other replicas to use new master until the master node is reachable again.

Sentinel agreement depends on the quorum value. Quorum value is the minimum number of the sentinels agree that the master is not reachable now. For 3 sentinel instances, the usual quorum value is 2.



Redis Sentinel

What will this mean for my applications? It means that they will connect to sentinel and sentinel instances will provide the latest master node.

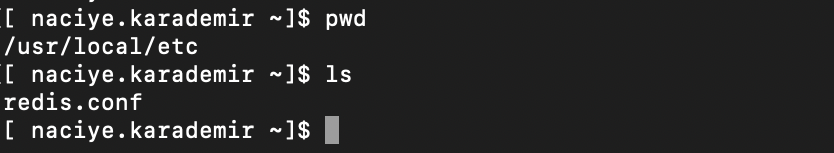
Configuration

To use Sentinel in our local we need to do some configurations. First, we need a replica and three sentinel instances. For this demo, I will have one master node, one replica, and three sentinel instances. In this section, I will explain all of the configuration processes.

Configure Replica

For configuration of Redis Replica open Redis configuration file. We will use the master configuration file, make necessary changes for the replica, and save it as a replica file. In my computer config file path of the master is:

/usr/local/etc/redis.conf



Redis Path

When you open the config file you can see that the default port for Redis master is 6379. To define a replica node change port to be **6380**andlook for**replicaof**which will be commented out.

port 6379  
# replicaof <masterip> <masterport>

In **replicaof,** we need to give our master IP and port which is running on localhost and port 6379. Save this file asredis-replica.conf .

port 6380  
replicaof 127.0.0.1 6379

Configure Sentinel

The default port for Sentinel is 26379, so for sentinel instance to work, port 26379 of your servers must be open to receive connections from the IP addresses of the other sentinel instances. Otherwise, these instances can’t talk and agree about what to do, so failover will never be performed.

You can configure the time required to check the master is down or not. Find down-after-milliseconds command which is the time in milliseconds to an instance should not be reachable for a Sentinel starting to think it is down. The default value is 30000 ms and you can change it as you like.

sentinel down-after-milliseconds mymaster 10000

Sentinel has two concepts of being down. Subjectively Down (SDOWN) condition is the local sentinel instance node thinks that Master is down. Objectively Down (ODOWN) condition is if enough instances (when quorum met) agree that the Master is not reachable.

Configure Sentinel to observe the Master with a specified quorum number.

# sentinel monitor <master-group-name> <ip> <port> <quorum>  
sentinel monitor mymaster 127.0.0.1 6379 2

Configure the second and third sentinel by changing ports and save as different sentinel files.

port 26380  
port 26381

As a result, I have one Master node, one replica, and three Sentinels.



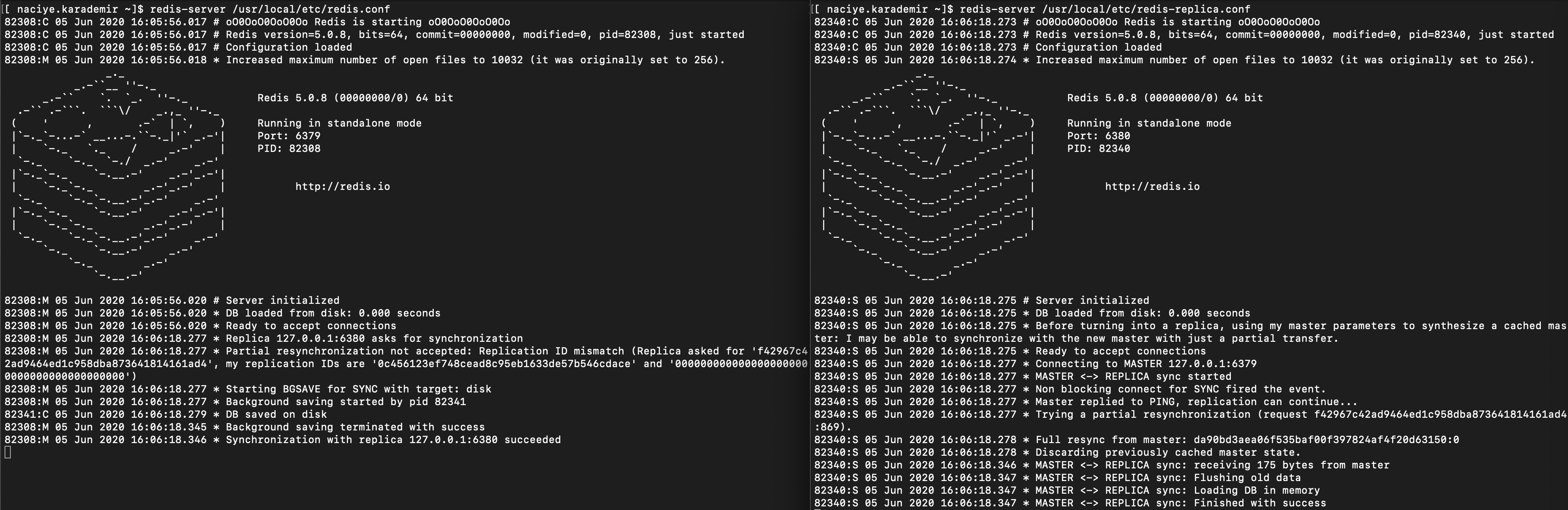
Redis Path After Configuration

Start the nodes and your sentinel configuration is ready to go.

Use Redis Sentinel

Let’s run our configuration files starting with the master node. You can run the configuration files by this command:

redis-server /path/to/redis-\*.conf

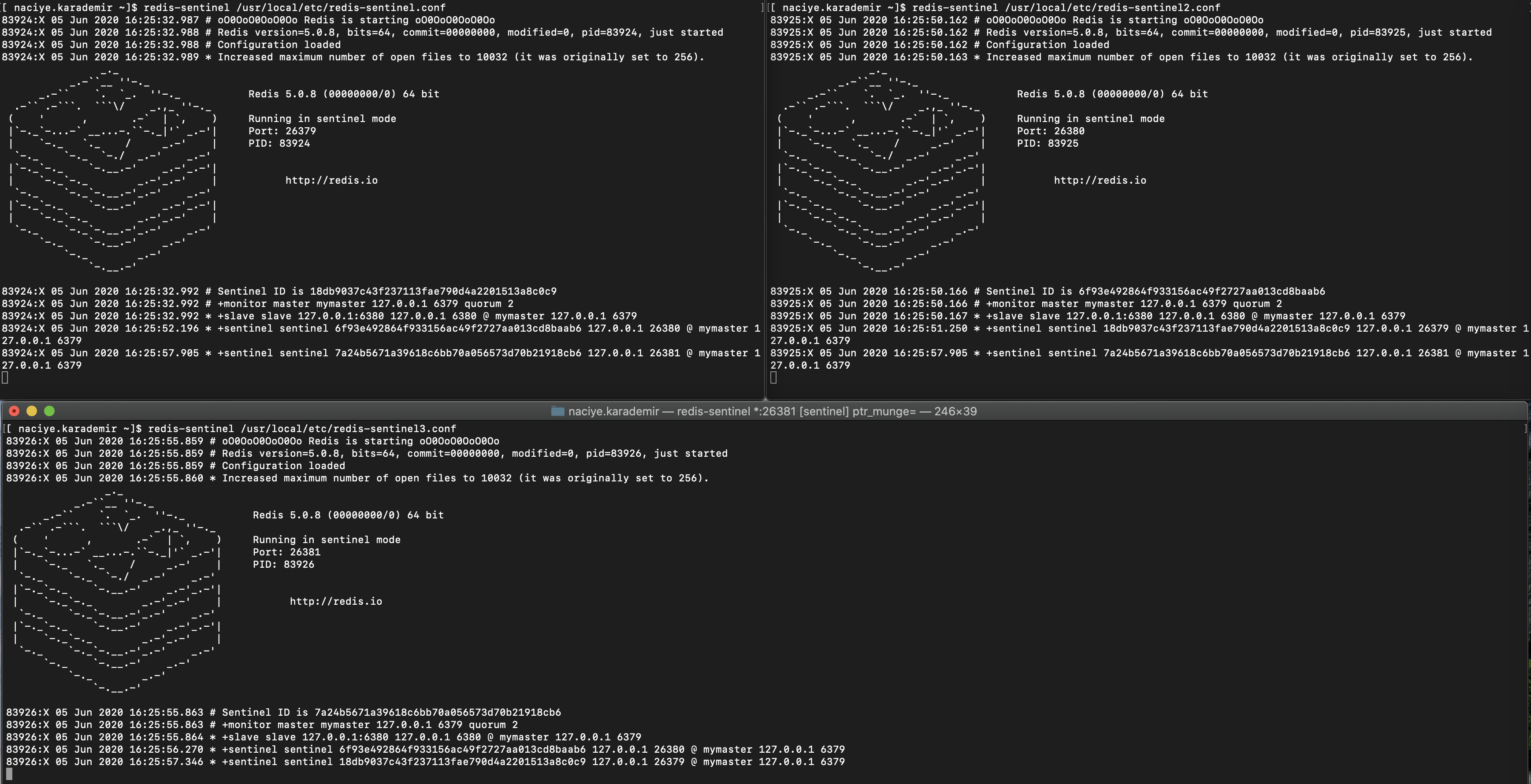


Run Redis and Replica

You can check running ports and if the master and replica are synced or not.

We also need to run our sentinel instances. You can run Sentinel instances by this command:

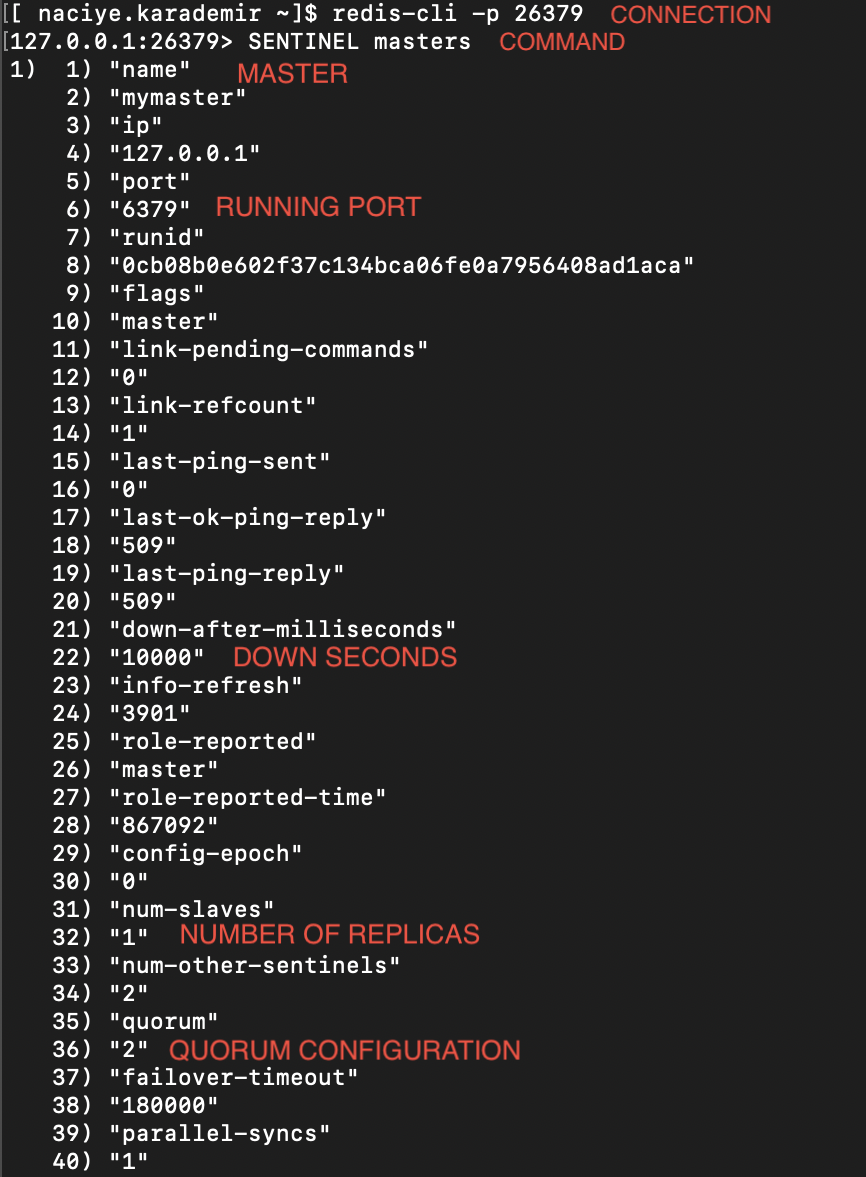
redis-sentinel /path/to/redis-sentinel\*.conf



Run Redis Sentinel Instances

You can check each running instance for Sentinel, you can see the running ports, as well as the master and quorum value.

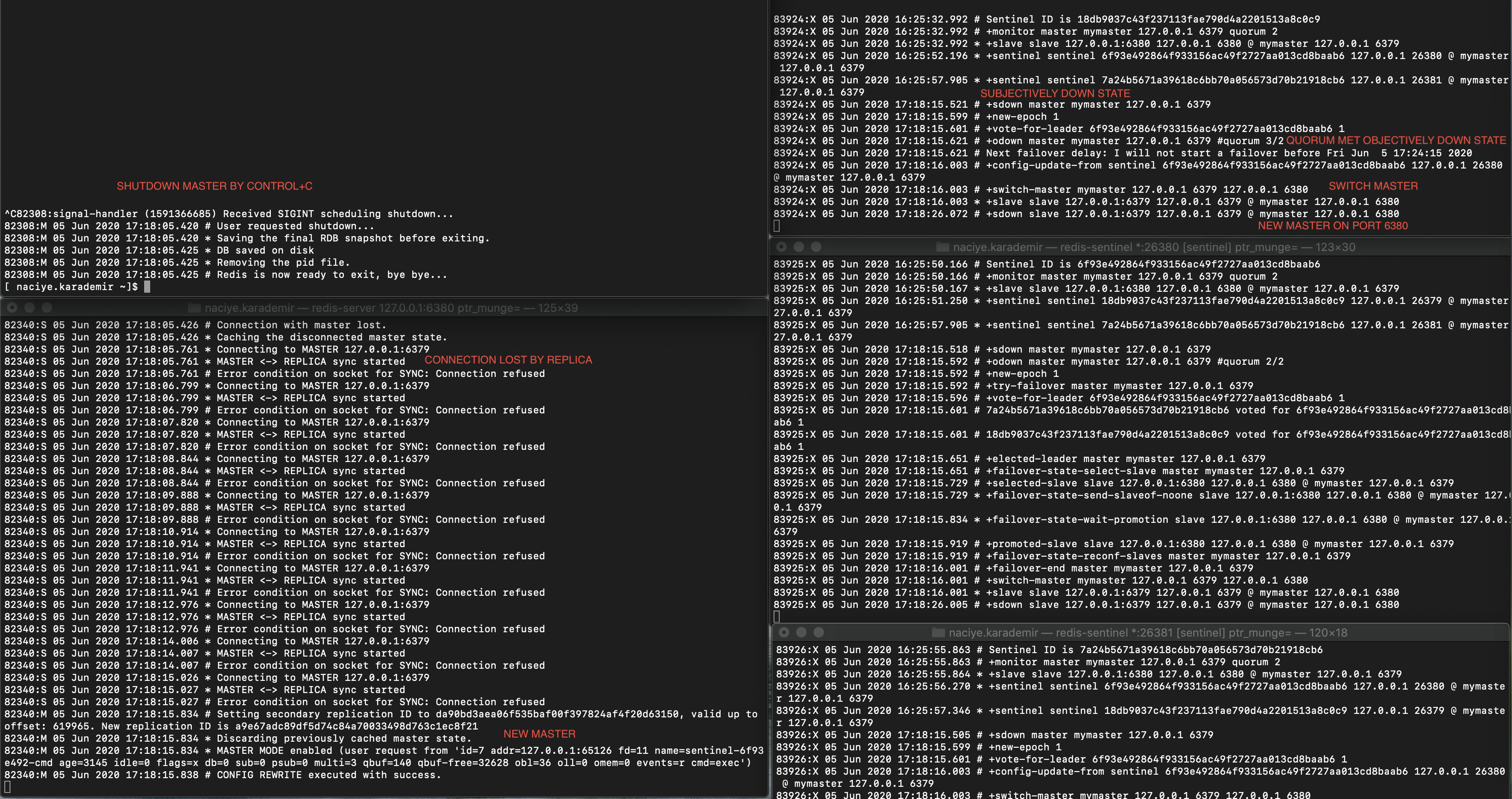
I will use Redis CLI to connect all nodes and use commands to get information or stored data. Let’s connect to a Sentinel and get info about our master and configurations.



Master and Sentinel Configuration

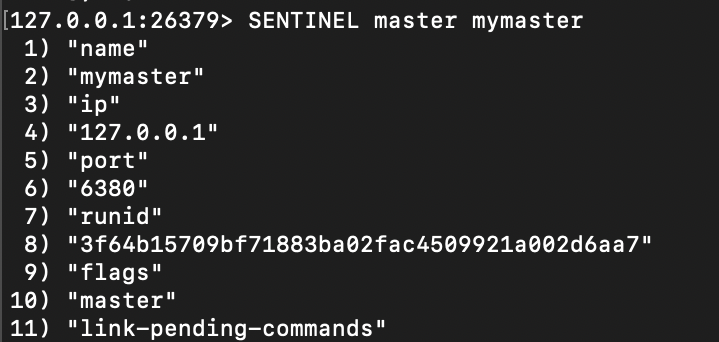
Failure Case

Now let’s see what is the behavior of sentinel instances in case of master failure by killing the master node.



Failure Of Master

After killing the master we can see that replica loses connection with the master. Sentinel instances go to **sdown** condition then quorum is met and **odown** condition is reached. The failover is started after 10 seconds since that is my configuration and after failover, the replica is promoted to be master.



My new master and port

Lettuce Client

To connect sentinel instances we need to use a client that also supports Sentinel itself. You can read about Redis Java clients [here](https://redis.io/clients#java). Stared clients are recommended Java clients for Redis.

In the Delivery Team, we use Lettuce with Spring Data Redis. Spring Boot reduces configurations greatly if we use Spring Data Redis. Also, Lettuce is one of the recommended Java clients for Redis. The documentation explains Lettuce client as:

*Advanced Redis client for thread-safe sync, async, and reactive usage. Supports Cluster, Sentinel, Pipelining, and codecs.*

We prefer Lettuce exactly for the stated reasons. It uses Netty for communication which makes for a heavier API, but also makes it better for sharing a connection with more than one thread. Also, Spring Data Redis uses Lettuce by default. Since our applications are concurrent and use Redis with Sentinel configuration, we prefer to use Lettuce client.

In this section, I will explain how to configure and use Redis with Lettuce client on Spring applications so that you can start to use it for yourself.

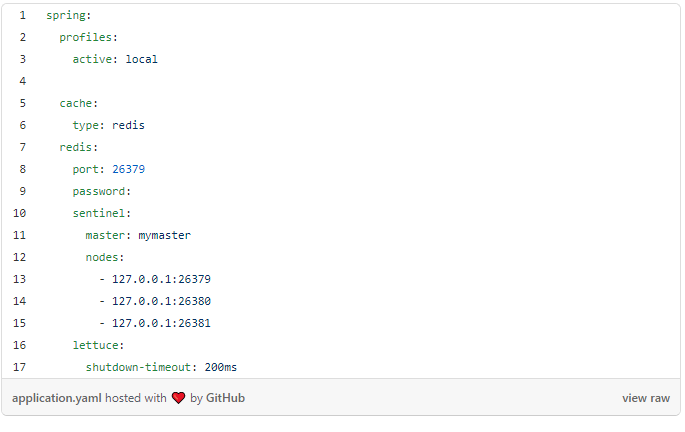
Configuration

I will start by adding the maven dependency of Spring Data Redis to my pom.

<!-- REDIS -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>

Then, continue with the YAML file and I add my connection information that I will use for configuration.

YAML



We also need to establish a connection configuration for Spring so that we can use Redis as our NoSQL database.

Config File

package com.trendyol.redisdemo.configuration;

import lombok.RequiredArgsConstructor;

import org.springframework.boot.autoconfigure.data.redis.RedisProperties;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.redis.connection.RedisPassword;

import org.springframework.data.redis.connection.RedisSentinelConfiguration;

import org.springframework.data.redis.connection.lettuce.LettuceClientConfiguration;

import org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory;

import org.springframework.data.redis.core.RedisTemplate;

import org.springframework.data.redis.serializer.GenericToStringSerializer;

import org.springframework.data.redis.serializer.JdkSerializationRedisSerializer;

import org.springframework.data.redis.serializer.StringRedisSerializer;

@RequiredArgsConstructor

@Configuration

public class RedisConfig {

private final RedisProperties redisProperties;

@Bean

protected LettuceConnectionFactory redisConnectionFactory() {

RedisSentinelConfiguration sentinelConfig = new RedisSentinelConfiguration()

.master(redisProperties.getSentinel().getMaster());

redisProperties.getSentinel().getNodes().forEach(s -> sentinelConfig.sentinel(s, redisProperties.getPort()));

sentinelConfig.setPassword(RedisPassword.of(redisProperties.getPassword()));

return new LettuceConnectionFactory(sentinelConfig, LettuceClientConfiguration.defaultConfiguration());

}

@Bean

public RedisTemplate<String, Object> redisTemplate() {

final RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();

redisTemplate.setKeySerializer(new StringRedisSerializer());

redisTemplate.setHashKeySerializer(new GenericToStringSerializer<>(Object.class));

redisTemplate.setHashValueSerializer(new JdkSerializationRedisSerializer());

redisTemplate.setValueSerializer(new JdkSerializationRedisSerializer());

redisTemplate.setConnectionFactory(redisConnectionFactory());

return redisTemplate;

}

}

In the config file, the information given in the YAML file is used for the connection. Master, sentinel nodes, and sentinel configuration are used for establishing the connection between our application and Redis. Redis Template bean definition is made since it will be used later in our util file for data storage.

Use Redis datatypes

We established our connection and created the necessary beans. Now, let’s use it for data storage. First I will define a Redis utility file. I’ll use the util file to store and get my data in Redis. We can think of this as a repository file for our database. In the util file, I only added the most-used operations but you can extend this for any Redis data type and operation.

Redis Util:

package com.trendyol.redisdemo.utility;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.redis.core.\*;

import org.springframework.stereotype.Component;

import java.util.List;

import java.util.Set;

import java.util.concurrent.TimeUnit;

@Component

public class RedisUtil<T> {

private RedisTemplate<String, T> redisTemplate;

private ValueOperations<String, T> valueOperations;

private ListOperations<String, T> listOperations;

private SetOperations<String, T> setOperations;

private HashOperations<String, Integer, T> hashOperations;

@Autowired

public RedisUtil(RedisTemplate<String, T> redisTemplate) {

this.redisTemplate = redisTemplate;

this.valueOperations = redisTemplate.opsForValue();

this.listOperations = redisTemplate.opsForList();

this.setOperations = redisTemplate.opsForSet();

this.hashOperations = redisTemplate.opsForHash();

}

//region String

public void putValue(String key, T value) {

valueOperations.set(key, value);

}

public T getValue(String key) {

return valueOperations.get(key);

}

public void setExpire(String key, long timeout, TimeUnit unit) {

redisTemplate.expire(key, timeout, unit);

}

//endregion

//region List

public void addList(String key, T value) {

listOperations.leftPush(key, value);

}

public List<T> getListMembers(String key) {

return listOperations.range(key, 0, -1);

}

public Long getListSize(String key) {

return listOperations.size(key);

}

//endregion

//region Set

public void addToSet(String key, T... values) {

setOperations.add(key, values);

}

public Set<T> getSetMembers(String key) {

return setOperations.members(key);

}

//endregion

//region Hash

public void saveHash(String key, Integer id, T value) {

hashOperations.put(key, id, value);

}

public T findInHash(String key, int id) {

return hashOperations.get(key, id);

}

public void deleteHash(String key, int id) {

hashOperations.delete(key, id);

}

//endregion

}

Now let’s see what my services look like.

Before defining my services I created a class called Programmer which will be used as data stored in Redis later.

Programmer Class

package com.trendyol.redisdemo.model;

import lombok.Builder;

import lombok.Data;

import java.io.Serializable;

@Data

@Builder

public class Programmer implements Serializable {

private int id;

private String company;

private String name;

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((company == null) ? 0 : company.hashCode());

result = prime \* result + id;

result = prime \* result + ((name == null) ? 0 : name.hashCode());

return result;

}

}

Redis service will use my object created from this class and the util file for any operation.

Redis Service

package com.trendyol.redisdemo.service.impl;

import com.trendyol.redisdemo.model.Programmer;

import com.trendyol.redisdemo.service.IRedisService;

import com.trendyol.redisdemo.utility.RedisUtil;

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Set;

import java.util.concurrent.TimeUnit;

@RequiredArgsConstructor

@Service

public class RedisService implements IRedisService {

private final RedisUtil<String> redisStringUtil;

private final RedisUtil<Programmer> redisProgrammerUtil;

//region String

@Override

public void setObjectAsString(String key, String value) {

redisStringUtil.putValue(key, value);

redisStringUtil.setExpire(key, 1, TimeUnit.HOURS);

}

@Override

public String getObjectAsString(String key) { return redisStringUtil.getValue(key); }

//endregion

//region List

@Override

public void addToList(String key, Programmer programmer) { redisProgrammerUtil.addList(key, programmer); }

@Override

public List<Programmer> getListMembers(String key) {

return redisProgrammerUtil.getListMembers(key);

}

@Override

public Long getListSize(String key) {

return redisProgrammerUtil.getListSize(key);

}

//endregion

//region Set

@Override

public void addToSet(String key, Programmer... programmers) { redisProgrammerUtil.addToSet(key, programmers); }

@Override

public Set<Programmer> getSetMembers(String key) {

return redisProgrammerUtil.getSetMembers(key);

}

//endregion

//region Hash

@Override

public void saveHash(String key, Programmer programmer) {

redisProgrammerUtil.saveHash(key, programmer.getId(), programmer);

}

@Override

public Programmer findInHash(String key, int id) { return redisProgrammerUtil.findInHash(key, id); }

@Override

public void deleteHash(String key, int id) { redisProgrammerUtil.deleteHash(key, id); }

//endregion

}

Lastly, let’s use the Redis Service in our programmer service.

Programmer Service

package com.trendyol.redisdemo.service.impl;

import com.trendyol.redisdemo.model.Programmer;

import com.trendyol.redisdemo.service.IProgrammerService;

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Set;

@Service

@RequiredArgsConstructor

public class ProgrammerService implements IProgrammerService {

private static final String LIST\_KEY = "LIST\_KEY";

private static final String SET\_KEY = "SET\_KEY";

private static final String HASH\_KEY = "HASH\_KEY";

private final RedisService redisService;

@Override

public void setObjectsAsString(String key, String programmer) {

redisService.setObjectAsString(key, programmer);

}

@Override

public String getObjectAsString(String key) {

return redisService.getObjectAsString(key);

}

@Override

public void AddToProgrammersList(Programmer programmer) {

redisService.addToList(LIST\_KEY, programmer);

}

@Override

public List<Programmer> getProgrammersListMembers() {

return redisService.getListMembers(LIST\_KEY);

}

@Override

public Long getProgrammersListCount() {

return redisService.getListSize(LIST\_KEY);

}

@Override

public void AddToProgrammersSet(Programmer... programmers) {

redisService.addToSet(SET\_KEY, programmers);

}

@Override

public Set<Programmer> getProgrammersSetMembers() {

return redisService.getSetMembers(SET\_KEY);

}

@Override

public void saveHash(Programmer programmer) {

redisService.saveHash(HASH\_KEY, programmer);

}

@Override

public Programmer findInHash(int id) {

return redisService.findInHash(HASH\_KEY, id);

}

@Override

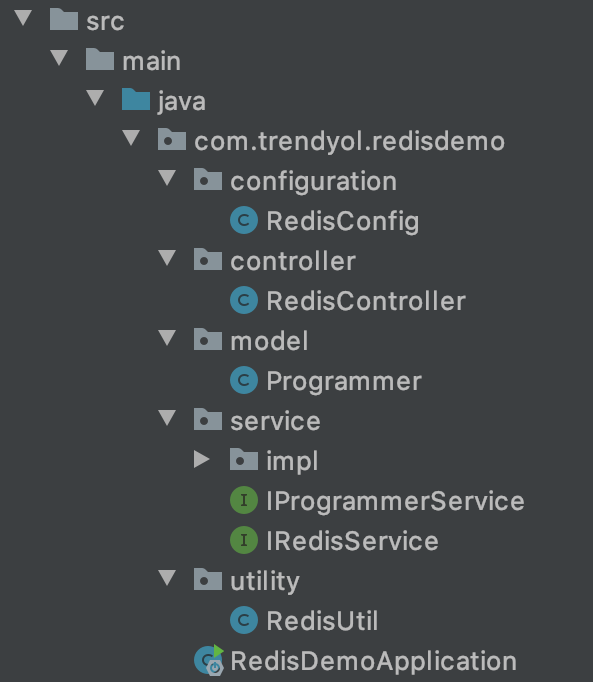
public void deleteHash(int id) {

redisService.deleteHash(HASH\_KEY, id);

}

}

This how my application looks like now!



Overall Application Look

86

* [Redis](https://medium.com/trendyol-tech/tagged/redis)
* [Sentinel](https://medium.com/trendyol-tech/tagged/sentinel)
* [Spring Boot](https://medium.com/trendyol-tech/tagged/spring-boot)
* [Lettuce](https://medium.com/trendyol-tech/tagged/lettuce)
* [Platform](https://medium.com/trendyol-tech/tagged/platform)

86 claps



WRITTEN BY

[Naciye Karademir](https://medium.com/@nkaradem?source=follow_footer-------------------------------------)

Follow

Computer Engineer @ Accenture