Chapter 1. Redis Engine

Table of Contents

Redis 1
Maven Setup 1
Dataset Format 1
Getting Started 3

Redis

Redis is an open source, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets and sorted sets.

NoSQLUnit supports *Redis* by using next classes:

Table 1.1. Lifecycle Management Rules

Embedded	`com.lordofthejars.nosqlunit.redis.EmbeddedRedis`
Managed	`com.lordofthejars.nosqlunit.redis.ManagedRedis`

Table 1.2. Manager Rule

NoSQLUnit Management	`com.lordofthejars.nosqlunit.redis.RedisRule`

Maven Setup

To use **NoSQLUnit** with Redis you only need to add next dependency:

Example 1.1. NoSqlUnit Maven Repository

```
<dependency>
 <groupId>com.lordofthejars</groupId>
 <artifactId>nosqlunit-redis</artifactId>
 <version>${version.nosqlunit}</version>
</dependency>
```

Dataset Format

Default dataset file format in *Redis* module is json.

Datasets must have next format :

Example 1.2. Example of Redis Dataset

```
{
"data":[
   {"simple": [
    {
     "key":"key1",
     "value":"value1"
    }
    ]
   },
        {"list": [{
                 "key":"key3",
                 "values":[
                     {"value":"value3"},
                     {"value":"value4"}
                 ]
       }]
        },

       {"sortset": [{
                     "key":"key4",
                     "values":[
                           {"score":2, "value":"value5" },{"score":3, "value":1 },
                 }]
        },
        {"hash": [
           {
            "key":"user",
            "values":[
             {"field":"name", "value":"alex"},
             {"field":"password", "value":"alex"}
            ]
           }
          ]
        },
        {"set":[{
                 "key":"key3",
                 "values":[
                     {"value":"value3"},
                     {"value":"value4"}
                 ]
       }]
        }
]
}
```

Root element must be called *data* , and then depending on kind of structured data we need to store, one or more of next elements should appear. Note that key field is used to set the key of the element, and value field is used to set a value.

- *simple* : In case we want to store simple key/value elements. This element will contain an array of key/value entries.

- *list* : In case we want to store a key with a list of values. This element contain a *key* field for key name and *values* field with an array of values.
- *set* In case we want to store a key within a set (no duplicates allowed). Structure is the same as list element.
- *sortset* : In case we want to store a key within a sorted set. This element contain the key, and an array of values, which each one, apart from value field, also contain *score* field of type Number, to set the order into sorted set.
- *hash* : In case we want to store a key within a map of field/value. In this case *field* element set the field name, and *value* set the value of that field.

Getting Started

Lifecycle Management Strategy

First step is defining which lifecycle management strategy is required for your tests. Depending on kind of test you are implementing (unit test, integration test, deployment test, ...) you will require an embedded approach, managed approach or remote approach.

Embedded Lifecycle

To configure **embedded** approach you should only instantiate next rule :

Example 1.3. Embedded Redis

```
@ClassRule
 public static EmbeddedRedis embeddedRedis = newEmbeddedRedisRule().build();
```

By default managed *Redis* rule uses next default values but can be configured programmatically:

Table 1.3. Default Embedded Values

Target path	This is the directory where *Redis* embedded instance is started and is `target/redis-test-data/impermanent-db` .

Note that target path is only used as a configuration parameter to allow multiple instances of embedded in-memory Redis engine.

For more information about embedded in-memory Redis take a tour to section: Embedded In-Memory Redis

Managed Lifecycle

To configure **managed** approach you should only instantiate next rule :

Example 1.4. Managed Redis

```
@ClassRule
public static ManagedRedis managedRedis = newManagedRedisRule().redisPath("/opt/re
```

By default managed *Redis* rule uses next default values but can be configured programmatically:

Table 1.4. Default Managed Values

Target path	This is the directory where *Redis* server is started and is `target/redis-temp` .
RedisPath	*Cassandra* installation directory which by default is retrieved from `REDIS_HOME` system environment variable.
Port	By default port used is 6379. If port is changed in *Redis* configuration file, this port should be configured too here.
Configuration File	By default *Redis* can work with no configuration file, it uses default values, but if we need to start *Redis* with an specific configuration file located in any directory file path should be set.

Remote Lifecycle

Configuring **remote** approach does not require any special rule because you (or System like Maven) is the responsible of starting and stopping the server. This mode is used in deployment tests where you are testing your application on real environment.

Configuring Redis Connection

Next step is configuring **Redis** rule in charge of maintaining *Redis* store into known state by inserting and deleting defined datasets. You must register `RedisRule` *JUnit* rule class, which requires a configuration parameter with information like host, port, or cluster name.

To make developer's life easier and code more readable, a fluent interface can be used to create these configuration objects. Three different kind of configuration builders exist.

Embedded Connection

The first one is for configuring an embedded connection to managed *Redis* .

Example 1.5. Redis with embedded configuration

```
import static com.lordofthejars.nosqlunit.redis.RedisRule.RedisRuleBuilder.newRedi

@Rule
public RedisRule redisRule = newRedisRule().defaultEmbeddedRedis();
```

Managed Connection

Configuring a connection to managed *Redis* .

Example 1.6. Redis with managed configuration

```
import static com.lordofthejars.nosqlunit.redis.ManagedRedisConfigurationBuilder.n

@Rule
public RedisRule redisRule = new RedisRule(newManagedRedisConfiguration().build())
```

Host and port parameters are already configured with default parameters of managed lifecycle. If port is changed, this class provides a method to set it.

Remote Connection

Configuring a connection to remote *Redis* .

Example 1.7. Redis with remote configuration

```
import static com.lordofthejars.nosqlunit.redis.RemoteRedisConfigurationBuilder.ne

@Rule
public RedisRule redisRule = new RedisRule(newRemoteRedisConfiguration().host("192
```

Port parameter is already configured with default parameter of managed lifecycle. If port is changed, this class provides a method to set it. Note that host parameter must be specified in this case.

Shard Connection

Redis connection can also be configured as shard using ShardedJedis capabilities.

Example 1.8. Redis with remote configuration

```
import static com.lordofthejars.nosqlunit.redis.RemoteRedisConfigurationBuilder.ne

@Rule
public RedisRule redisRule = new RedisRule(newShardedRedisConfiguration()
    .shard(host("127.0.0.1"), port(ManagedRedis.DEFAULT_PORT))
     .password("a")
     .timeout(1000)
     .weight(1000)
    .shard(host("127.0.0.1"), port(ManagedRedis.DEFAULT_PORT + 1))
     .password("b")
     .timeout(3000)
     .weight(3000)
    .build());
```

Note that only
<methodparam>host</methodparam>
and
<methodparam>port</methodparam>
is mandatory, the other ones uses default values.

- *password* : In case repository is protected with password this attribute is used as password. Default values is null.
- *timeout* : Timeout for shard. By default timeout is set to 2 seconds.
- *weight* : The weight of that shard over the other ones. By default is 1.

Verifying Data

`@ShouldMatchDataSet` is also supported for *Redis* engine.

Full Example

To show how to use **NoSQLUnit** with *Redis* , we are going to create a very simple application.

BookManager is the business class responsible of inserting new books and finding books by their title.

Example 1.9. Book manager with Redis

```
public class BookManager {

 private static final String TITLE_FIELD_NAME = "title";
 private static final String NUMBER_OF_PAGES = "numberOfPages";

 private Jedis jedis;

 public BookManager(Jedis jedis) {
  this.jedis = jedis;
 }

 public void insertBook(Book book) {

  Map<String, String> fields = new HashMap<String, String>();

  fields.put(TITLE_FIELD_NAME, book.getTitle());
  fields.put(NUMBER_OF_PAGES, Integer.toString(book.getNumberOfPages()));

  jedis.hmset(book.getTitle(), fields);
 }

 public Book findBookByTitle(String title) {

  Map<String, String> fields = jedis.hgetAll(title);
  return new Book(fields.get(TITLE_FIELD_NAME), Integer.parseInt(fields.get(NUMBER

 }

}
```

And now one integration test is written:

Example 1.10. Redis with managed configuration

```
import static com.lordofthejars.nosqlunit.redis.RedisRule.RedisRuleBuilder.newRedi
import static com.lordofthejars.nosqlunit.redis.ManagedRedis.ManagedRedisRuleBuild

import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.is;

import org.junit.ClassRule;
import org.junit.Rule;
import org.junit.Test;

import redis.clients.jedis.Jedis;

import com.lordofthejars.nosqlunit.annotation.UsingDataSet;
import com.lordofthejars.nosqlunit.core.LoadStrategyEnum;
import com.lordofthejars.nosqlunit.demo.model.Book;
import com.lordofthejars.nosqlunit.redis.ManagedRedis;
import com.lordofthejars.nosqlunit.redis.RedisRule;

public class WhenYouFindABook {

 static {
  System.setProperty("REDIS_HOME", "/opt/redis-2.4.16");
 }

 @ClassRule
 public static ManagedRedis managedRedis = newManagedRedisRule().build();

 @Rule
 public RedisRule redisRule = newRedisRule().defaultManagedRedis();

 @Test
 @UsingDataSet(locations="book.json", loadStrategy=LoadStrategyEnum.CLEAN_INSERT)
 public void book_should_be_returned_if_title_is_in_database() {

  BookManager bookManager = new BookManager(new Jedis("localhost"));
  Book findBook = bookManager.findBookByTitle("The Hobbit");

  assertThat(findBook, is(new Book("The Hobbit", 293)));

 }

}
```

And dataset used is:

Example 1.11. book.json Redis file

```
{
"data":[
        {"hash": [
           {
            "key":"The Hobbit",
            "values":[
             {"field":"title", "value":"The Hobbit"},
             {"field":"numberOfPages", "value":"293"}
            ]
           }
          ]
        }
]
}
```