Chapter 1. Advanced Usage

Table of Contents

Embedded In-Memory Redis

When you are writing unit tests you should keep in mind that they must run fast, which implies, among other things, no interaction with IO subsystem (disk, network, ...). To avoid this interaction in database unit tests, there are embedded in-memory databases like *H2* , *HSQLDB* , *Derby* or in case of *NoSQL* , engines like *Neo4j* or *Cassandra* have their own implementation. But *Redis* does not have any way to create an embedded in-memory instance in Java. For this reason I have written an embedded in-memory *Redis* implementation based on *Jedis* project.

If you are using **NoSQLUnit** you only have to register embedded *Redis* rule as described here , and internally **NoSQLUnit** will create instance for you, and you will be able to inject the instance into your code.

But also can be used outside umbrella of **NoSQLUnit** , by instantiating manually, as described in next example:

Example 1.1. Embedded In-Memory Redis.

```
EmbeddedRedisBuilder embeddedRedisBuilder = new EmbeddedRedisBuilder();
Jedis jedis = embeddedRedisBuilder.createEmbeddedJedis();
```

Notice that Jedis class is the main class defined by Jedis project but proxied to use in-memory data instead of sending requests to remote server.

Almost all *Redis* operations have been implemented but it has some limitations:

- Connection commands do nothing, they do not throw any exception but neither do any action. In fact would not have sense that they do something.
- Scripting commands are not supported, and an `UnsupportedOperationException` will be thrown if they are called.
- Transaction commands are not supported, but they do not throw any exception, simply returns a null value and in cases of List return type, an empty list is returned.
- Pub/Sub commands do nothing.
- Server commands are implemented, but there are some commands that have no sense and returns a constant result:

  move always return 1.

  debug commands throws an UnsupportedOperationException.

bgrewriteaof, save, bgsave, configSet, configResetStat, salveOf, slaveOfNone and slowLogReset returns an OK.

configGet, slowLogGet and slowLogGetBinary returns an empty list.

- From Key commands, only sort by pattern is not supported.

All the other operations, including flushing, expiration control, and each operation of every datatype is supported in the same way Jedis support it. Note that expiration management is also implemented as described in Redis manual.

Warning

This implementation of Redis is provided for testing purposes not as a substitution of Redis. Feel free to notify any issue of this implementation so can be fixed or implemented.

Managing lifecycle of multiple instances

Sometimes your test will require that more than one instance of same database server (running in different ports) was started. For example for testing database sharding. In next example we see how to configure **NoSQLUnit** to manage lifecycle of multiple instances.

Example 1.2. Multiple Instances of Redis.

```
@ClassRule
public static ManagedRedis managedRedis79 = newManagedRedisRule().redisPath("/opt/
        .targetPath("target/redis1")
        .configurationPath(getAbsoluteFilePath("src/test/resources/redis_6379.conf
        .port(6379)
        .build();

@ClassRule
public static ManagedRedis managedRedis80 = newManagedRedisRule().redisPath("/opt/
        .targetPath("target/redis2")
        .configurationPath(getAbsoluteFilePath("src/test/resources/redis_6380.conf
        .port(6380)
        .build();
```

Warning

Note that target path should be set to different values for each instance, if not some started processes could not be shutdown.

Fast Way

When you instantiate a Rule for maintaining database into known state (`MongoDbRule` , `Neo4jRule` , ...) **NoSQLUnit** requires you set a configuration object with properties like host, port, database name, ... but although most of the time default values are enough, we still need to create the configuration object, which means our code becomes harder to read.

We can avoid this by using an inner builder inside each rule, which creates for us a Rule with default parameters set. For example for `Neo4jRule` :

Example 1.3. Embedded Neo4jRule with defaults.

```
import static com.lordofthejars.nosqlunit.neo4j.Neo4jRule.Neo4jRuleBuilder.newNeo4
@Rule
public Neo4jRule neo4jRule = newNeo4jRule().defaultEmbeddedNeo4j();
```

In previous example `Neo4jRule` is configured to be used as embedded approach with default parameters.

Another example using `CassandraRule` in managed way.

Example 1.4. Managed Cassandra with defaults.

```
import static com.lordofthejars.nosqlunit.cassandra.CassandraRule.CassandraRuleBui
@Rule
public CassandraRule cassandraRule = newCassandraRule().defaultManagedCassandra("T
```

And each Rule contains their builder class to create default values.

Simultaneous engines

Sometimes applications will contain more than one *NoSQL* engine, for example some parts of your model will be expressed better as a graph (Neo4J for example), but other parts will be more natural in a column way (for example using Cassandra). **NoSQLUnit** supports this kind of scenarios by providing in integration tests a way to not load all datasets into one system, but choosing which datasets are stored in each backend.

For declaring more than one engine, you must give a name to each database *Rule* using `connectionIdentifier()` method in configuration instance.

Example 1.5. Given a name database rule

```
@Rule
public MongoDbRule remoteMongoDbRule1 = new MongoDbRule(mongoDb()
                                        .databaseName("test").connectionIdentifier
```

And also you need to provide an identified dataset for each engine, by using `withSelectiveLocations` attribute of `@UsingDataSet` annotation. You must set up the pair "named connection" / datasets.

Example 1.6. Selective dataset example

```
@UsingDataSet(withSelectiveLocations =
    { @Selective(identifier = "one", locations = "test3") },
   loadStrategy = LoadStrategyEnum.REFRESH)
```

In example we are refreshing database declared on previous example with data located at *test3* file.

Also works in expectations annotation:

Example 1.7. Selective expectation example

```
@ShouldMatchDataSet(withSelectiveMatcher =
    { @SelectiveMatcher(identifier = "one", location = "test3")
    })
```

When you use more than one engine at a time you should take under consideration next rules:

- If location attribute is set, it will use it and will ignore `withSelectiveMatcher` attribute data. Location data is populated through all registered systems.
- If location is not set, then system tries to insert data defined in `withSelectiveMatcher` attribute to each backend.
- If `withSelectiveMatcher` attribute is not set, then default strategy (explained in section) is taken. Note that default strategy will replicate all datasets to defined engines.

You can also use the same approach for inserting data into same engine but in different databases. If you have one MongoDb instance with two databases, you can also write tests for both databases at one time. For example:

Example 1.8. Multiple connections example

```
@Rule
public MongoDbRule remoteMongoDbRule1 = new MongoDbRule(mongoDb()
     .databaseName("test").connectionIdentifier("one").build() ,this);

@Rule
public MongoDbRule remoteMongoDbRule2 = new MongoDbRule(mongoDb()
     .databaseName("test2").connectionIdentifier("two").build() ,this);

@Test
@UsingDataSet(withSelectiveLocations = {
  @Selective(identifier = "one", locations = "json.test"),
  @Selective(identifier = "two", locations = "json3.test") },
 loadStrategy = LoadStrategyEnum.CLEAN_INSERT)
public void my_test() {...}
```

Support for JSR-330

NoSQLUnit supports two annotations of JSR-330 aka Dependency Injection for Java. Concretely `@Inject` and `@Named` annotations.

During test execution you may need to access underlying class used to load and assert data to execute extra operations to backend. **NoSQLUnit** will inspect `@Inject` annotations of test fields, and try to set own driver to attribute. For example in case of MongoDb , `com.mongodb.Mongo` instance will be injected.

Example 1.9. Injection example

```
@Rule
public MongoDbRule remoteMongoDbRule1 = new MongoDbRule(mongoDb()
      .databaseName("test").build() ,this);

@Inject
private Mongo mongo;
```

Warning

Note that in example we are setting `this` as second parameter to the Rule. This is only required in versions of JUnit prior to 4.11. In new versions is no longer required passing the this parameter.

But if you are using more than one engine at same time (see chapter) you need a way to distinguish each connection. For fixing this problem, you must use `@Named` annotation by putting the identifier given in configuration instance. For example:

Example 1.10. Named injection example

```
@Rule
public MongoDbRule remoteMongoDbRule1 = new MongoDbRule(mongoDb()
     .databaseName("test").connectionIdentifier("one").build() ,this);

@Rule
public MongoDbRule remoteMongoDbRule2 = new MongoDbRule(mongoDb()
     .databaseName("test2").connectionIdentifier("two").build() ,this);

@Named("one")
@Inject
private Mongo mongo1;

@Named("two")
@Inject
private Mongo mongo2;
```