NoSQLUnit Reference Manual

Alex Soto, www.lordofthejars.com

NoSQLUnit Reference Manual

by Alex Soto

0.7.2

Table of Contents

List of Tables

List of Examples

Part I. NoSQLUnit Core

This chapter provides an explanation of why **NoSQLUnit** should be used for testing applications that use *NoSQL* engines as databases. Also will provide an explanation of the main concepts of **NoSQLUnit**.

Chapter 1. NoSQLUnit Core

Overview

Unit testing is a method by which the smallest testable part of an application is validated. Unit tests must follow the FIRST Rules; these are Fast, Isolated, Repeatable, Self-Validated and Timely.

It is strange to think about a JEE application without persistence layer (typical Relational databases or new *NoSQL* databases) so should be interesting to write unit tests of persistence layer too. When we are writing unit tests of persistence layer we should focus on to not break two main concepts of FIRST rules, the fast and the isolated ones.

Our tests will be *fast* if they don't access network nor filesystem, and in case of persistence systems network and filesystem are the most used resources. In case of RDBMS (*SQL*), many Java in-memory databases exist like Apache Derby , H2 or HSQLDB . These databases, as their name suggests are embedded into your program and data are stored in memory, so your tests are still fast. The problem is with *NoSQL* systems, because of their heterogeneity. Some systems work using Document approach (like MongoDb), other ones Column (like Hbase), or Graph (like Neo4J). For this reason the in-memory mode should be provided by the vendor, there is no a generic solution.

Our tests must be isolated from themselves. It is not acceptable that one test method modifies the result of another test method. In case of persistence tests this scenario occurs when previous test method insert an entry to database and next test method execution finds the change. So before execution of each test, database should be found in a known state. Note that if your test found database in a known state, test will be repeatable, if test assertion depends on previous test execution, each execution will be unique. For homogeneous systems like RDBMS , *DBUnit* exists to maintain database in a known state before each execution. But there is no like *DBUnit* framework for heterogeneous *NoSQL* systems.

NoSQLUnit resolves this problem by providing a *JUnit* extension which helps us to manage lifecycle of NoSQL systems and also take care of maintaining databases into known state.

Requirements

To run **NoSQLUnit** , *JUnit 4.10* or later must be provided. This is because of **NoSQLUnit** is using *Rules* , and they have changed from previous versions to 4.10.

Although it should work with JDK 5 , jars are compiled using JDK 6 .

NoSQLUnit

NoSQLUnit is a *JUnit* extension to make writing unit and integration tests of systems that use NoSQL backend easier and is composed by two sets of *Rules* and a group of annotations.

First set of *Rules* are those responsible of managing database lifecycle; there are two for each supported backend.

- The first one (in case it is possible) it is the **in-memory** mode. This mode takes care of starting and stopping database system in " *in-memory* " mode. This mode will be typically used during unit testing execution.
- The second one is the **managed** mode. This mode is in charge of starting *NoSQL* server but as remote process (in local machine) and stopping it. This will typically used during integration testing execution.

Second set of *Rules* are those responsible of maintaining database into known state. Each supported back-end will have its own, and can be understood as a connection to defined database which will be used to execute the required operations for maintaining the stability of the system.

Note that because *NoSQL* databases are heterogeneous, each system will require its own implementation.

And finally two annotations are provided, @UsingDataSet and @ShouldMatchDataSet , (thank you so much *Arquillian* people for the name).

Seeding Database

@UsingDataSet is used to seed database with defined data set. In brief data sets are files that contain all data to be inserted to configured database. In order to seed your database, use *@UsingDataSet* annotation, you can define it either on the test itself or on the class level. If there is definition on both, test level annotation takes precedence. This annotation has two attributes `locations` and `loadStrategy` .

With `locations` attribute you can specify **classpath** datasets location. Locations are relative to test class location. Note that more than one dataset can be specified.

Also `withSelectiveLocations` attribute can be used to specify datasets location. See Advanced Usage chapter for more information.

If files are not specified explicitly, next strategy is applied:

- First searches for a file on classpath in same package of test class with next file name, `[test class name]#[test method name].[format]` (only if annotation is present at test method).
- If first rule is not met or annotation is defined at class scope, next file is searched on classpath in same package of test class, `[test class name].[default format]`.

Warning

datasets must reside into *classpath* and format depends on *NoSQL* vendor.

Second attribute provides strategies for inserting data. Implemented strategies are:

Table 1.1. Load Strategies

INSERT	Insert defined datasets before executing any test method.

DELETE_ALL	Deletes all elements of database before executing any test method.
CLEAN_INSERT	This is the most used strategy. It deletes all elements of database and then insert defined datasets before executing any test method.

An example of usage:

```
@UsingDataSet(locations="my_data_set.json", loadStrategy=LoadStrategyEnum.INSERT)
```

Verifying Database

Sometimes it might imply a huge amount of work asserting database state directly from testing code. By using *@ShouldMatchDataSet* on test method, **NoSQLUnit** will check if database contains expected entries after test execution. As with *@ShouldMatchDataSet* annotation you can define classpath file location, or using `withSelectiveMatcher` See Advanced Usage chapter for more information.

If it is not dataset is supplied next convention is used:

- First searches for a file on classpath in same package of test class with next file name, `[test class name]#[test method name]-expected.[format]` (only if annotation is present at test method).
- If first rule is not met or annotation is defined at class scope, file is searched on classpath in same package of test class, `[test class name]-expected.[default format]`.

Warning

datasets must reside into *classpath* and format depends on *NoSQL* vendor.

An example of usage:

```
@ShouldMatchDataSet(location="my_expected_data_set.json")
```

Part II. Supported Engines

This chapter provides an overview of supported *NoSQL* databases, and how to write tests for them, using **NoSQLUnit**.

Chapter 2. MongoDb Engine

MongoDb

MongoDb is a *NoSQL* database that stores structured data as *JSON-like* documents with dynamic schemas.

NoSQLUnit supports *MongoDb* by using next classes:

Table 2.1. Lifecycle Management Rules

In Memory	`com.lordofthejars.nosqlunit.mongodb.InMemoryMon`
Managed	`com.lordofthejars.nosqlunit.mongodb.ManagedMong`

Table 2.2. Manager Rule

NoSQLUnit Management	`com.lordofthejars.nosqlunit.mongodb.MongoDbRule`

Maven Setup

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

Example 2.1. NoSqlUnit Maven Repository

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

Note that if you are plannig to use **in-memory** approach an extra dependency is required. **In-memory** mode is implemented using *jmockmongo* . *JMockmongo* is a new project that help with unit testing Java-based MongoDb Applications by starting an in-process *Netty* server that speaks the *MongoDb* protocol and maintains databases and collections in JVM memory. It is not a true embedded mode becuase it will starts a server, but in fact for now it is the best way to write MongoDb unit tests. As his author says it is an incomplete tool and will be improved every time a new feature is required.

Warning

During development of this documentation, current *jmockmongo* version was 0.0.2-SNAPSHOT. Author is imporoving version often so before using one specific version, take a look at its website [https://github.com/thiloplanz/jmockmongo] .

To install add next repository and dependency :

Example 2.2. jmockmongo Maven Repository

```
<repositories>
 <repository>
  <id>thiloplanz-snapshot</id>
  <url>http://repository-thiloplanz.forge.cloudbees.com/snapshot/</url>
 </repository>
</repositories>
```

Example 2.3. jmockmongo Maven Dependency

```
<dependency>
 <groupId>jmockmongo</groupId>
 <artifactId>jmockmongo</artifactId>
 <version>${mongomock.version}</version>
</dependency>
```

Dataset Format

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

Datasets must have next format :

Example 2.4. Example of MongoDb Dataset

```
{
 "name_collection1": [
 {
  "attribute_1":"value1",
  "attribute_2":"value2"
 },
 {
  "attribute_3":2,
  "attribute_4":"value4"
 }
 ],
 "name_collection2": [
  ...
 ],
 ....
}
```

Notice that if attributes value are integers, double quotes are not required.

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 **in-memory** approach, **managed** approach or **remote** approach.

To configure **in-memory** approach you should only instantiate next rule :

Example 2.5. In-memory MongoDb

```
@ClassRule
InMemoryMongoDb inMemoryMongoDb = new InMemoryMongoDb();
```

To configure the **managed** way, you should use `ManagedMongoDb` rule and may require some configuration parameters.

Example 2.6. Managed MongoDb

```
import static com.lordofthejars.nosqlunit.mongodb.ManagedMongoDb.MongoServerRuleBu

@ClassRule
public static ManagedMongoDb managedMongoDb = newManagedMongoDbRule().build();
```

By default managed *MongoDb* rule uses next default values:

- *MongoDb* installation directory is retrieved from `MONGO_HOME` system environment variable.
- Target path, that is the directory where *MongoDb* server is started, is `target/mongo-temp` .
- Database path is at *`{target path}`* `/mongo-dbpath` .
- Because after execution of tests all generated data is removed, in *`{target path}`* `/logpath` will remain log file generated by the server.
- In *Windows* systems executable should be found as `bin/mongod.exe` meanwhile in *MAC OS* and **nix* should be found as `bin/mongod` .

`ManagedMongoDb` can be created from scratch, but for making life easier, a *DSL* is provided using `MongoServerRuleBuilder` class. For example :

Example 2.7. Specific Managed MongoDb Configuration

```
import static com.lordofthejars.nosqlunit.mongodb.ManagedMongoDb.MongoServerRuleBu

@ClassRule
public static ManagedMongoDb managedMongoDb =
newManagedMongoDbRule().mongodPath("/opt/mongo").appendSingleCommandLineArguments(
```

In example we are overriding `MONGO_HOME` variable (in case has been set) and set mongo home at `/opt/mongo` . Moreover we are appending a single argument to *MongoDb* executable, in this case setting log level to number 3 (-vvv). Also you can append *property=value* arguments using `appendCommandLineArguments(String argumentName, String argumentValue)` method.

Warning

when you are specifying command line arguments, remember to add slash (-) and double slash (--) where is necessary.

To stop *MongoDb* instance, **NoSQLUnit** sends a `shutdown` command to server using *Java Mongo AP* I. When this command is sent, the server is stopped and because connection is lost, *Java Mongo API* logs automatically an exception (read here [https://groups.google.com/group/mongodb-user/browse_thread/thread/ac9a4c9ea13f3e81] information about the problem and how to "resolve" it). Do not confuse with a testing failure. You will see something like:

```
java.io.EOFException
 at org.bson.io.Bits.readFully(Bits.java:37)
 at org.bson.io.Bits.readFully(Bits.java:28)
 at com.mongodb.Response.<init>;(Response.java:39)
 at com.mongodb.DBPort.go(DBPort.java:128)
 at com.mongodb.DBPort.call(DBPort.java:79)
 at com.mongodb.DBTCPConnector.call(DBTCPConnector.java:218)
```

```
at com.mongodb.DBApiLayer$MyCollection.__find(DBApiLayer.java:305)
at com.mongodb.DB.command(DB.java:160)
at com.mongodb.DB.command(DB.java:183)
at com.mongodb.DB.command(DB.java:144)
at
com.lordofthejars.nosqlunit.mongodb.MongoDbLowLevelOps.shutdown(MongoDbLowLevelOp
at
com.lordofthejars.nosqlunit.mongodb.ManagedMongoDb.after(ManagedMongoDb.java:157)
at
org.junit.rules.ExternalResource$1.evaluate(ExternalResource.java:48)
at org.junit.rules.RunRules.evaluate(RunRules.java:18)
at org.junit.runners.ParentRunner.run(ParentRunner.java:300)
at
org.apache.maven.surefire.junit4.JUnit4Provider.execute(JUnit4Provider.java:236)
at
org.apache.maven.surefire.junit4.JUnit4Provider.executeTestSet(JUnit4Provider.jav
at
org.apache.maven.surefire.junit4.JUnit4Provider.invoke(JUnit4Provider.java:113)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java
at java.lang.reflect.Method.invoke(Method.java:616)
at
org.apache.maven.surefire.util.ReflectionUtils.invokeMethodWithArray(ReflectionUt
at
org.apache.maven.surefire.booter.ProviderFactory$ProviderProxy.invoke(ProviderFac
at
org.apache.maven.surefire.booter.ProviderFactory.invokeProvider(ProviderFactory.j
at
org.apache.maven.surefire.booter.ForkedBooter.runSuitesInProcess(ForkedBooter.jav
at
org.apache.maven.surefire.booter.ForkedBooter.main(ForkedBooter.java:74)
```

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 MongoDb Connection

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

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

The first one is for configuring a connection to in-memory *jmockmongo* server. Default connection values are:

Table 2.3. Default In-Memory Configuration Values

Host	0.0.0.0

Port	2307

Notice that these values are the default ones of *jmockmongo* project, so if you are thinking to use *jmockmongo* , no modifications are required.

Example 2.8. MongoDbRule with in-memory configuration

```
import static com.lordofthejars.nosqlunit.mongodb.InMemoryMongoDbConfigurationBuil

@Rule
public MongoDbRule remoteMongoDbRule = new MongoDbRule(inMemoryMongoDb().databaseN
```

The second one is for configuring a connection to remote *MongoDb* server. Default values are:

Table 2.4. Default Managed Configuration Values

Host	localhost
Port	27017
Authentication	No authentication parameters.

Example 2.9. MongoDbRule with managed configuration

```
import static com.lordofthejars.nosqlunit.mongodb.MongoDbConfigurationBuilder.mong

@Rule
public MongoDbRule remoteMongoDbRule = new MongoDbRule(mongoDb().databaseName("tes
```

Example 2.10. MongoDbRule with remote configuration

```
import static com.lordofthejars.nosqlunit.mongodb.MongoDbConfigurationBuilder.mong

@Rule
public MongoDbRule remoteMongoDbRule = new MongoDbRule(mongoDb().databaseName("tes
```

Complete Example

Consider a library application, which apart from multiple operations, it allow us to add new books to system. Our model is as simple as:

Example 2.11. Book POJO

```
public class Book {

 private String title;

 private int numberOfPages;

 public Book(String title, int numberOfPages) {
  super();
  this.title = title;
  this.numberOfPages = numberOfPages;
 }

 public void setTitle(String title) {
  this.title = title;
 }

 public void setNumberOfPages(int numberOfPages) {
  this.numberOfPages = numberOfPages;
 }


 public String getTitle() {
  return title;
 }

 public int getNumberOfPages() {
  return numberOfPages;
 }
}
```

Next business class is the responsible of managing access to *MongoDb* server:

Example 2.12. Book POJO

```
public class BookManager {

 private static final Logger LOGGER = LoggerFactory.getLogger(BookManager.class);

 private static final MongoDbBookConverter MONGO_DB_BOOK_CONVERTER = new MongoDbBo
 private static final DbObjectBookConverter DB_OBJECT_BOOK_CONVERTER = new DbObjec


 private DBCollection booksCollection;

 public BookManager(DBCollection booksCollection) {
  this.booksCollection = booksCollection;
 }

 public void create(Book book) {
  DBObject dbObject = MONGO_DB_BOOK_CONVERTER.convert(book);
  booksCollection.insert(dbObject);
 }
}
```

And now it is time for testing. In next test we are going to validate that a book is inserted correctly into database.

Example 2.13. Test with Managed Connection

```
package com.lordofthejars.nosqlunit.demo.mongodb;

public class WhenANewBookIsCreated {

 @ClassRule
 public static ManagedMongoDb managedMongoDb = newManagedMongoDbRule().mongodPath(

 @Rule
 public MongoDbRule remoteMongoDbRule = new MongoDbRule(mongoDb().databaseName("te

 @Test
 @UsingDataSet(locations="initialData.json", loadStrategy=LoadStrategyEnum.CLEAN_I
 @ShouldMatchDataSet(location="expectedData.json")
 public void book_should_be_inserted_into_repository() {

  BookManager bookManager = new BookManager(MongoDbUtil.getCollection(Book.class.g

  Book book = new Book("The Lord Of The Rings", 1299);
  bookManager.create(book);
 }

}
```

In previous test we have defined that *MongoDb* will be managed by test by starting an instance of server located at `/opt/mongo` . Moreover we are setting an initial dataset in file `initialData.json` located at classpath `com/lordofthejars/nosqlunit/demo/mongodb/initialData.json` and expected dataset called `expectedData.json` .

Example 2.14. Initial Dataset

```
{
 "Book":
 [
  {"title":"The Hobbit","numberOfPages":293}
 ]
}
```

Example 2.15. Expected Dataset

```
{
 "Book":
 [
  {"title":"The Hobbit","numberOfPages":293},
  {"title":"The Lord Of The Rings","numberOfPages":1299}
 ]
}
```

You can watch full example at github [https://github.com/lordofthejars/nosql-unit/tree/master/nosqlunit-demo] .

Chapter 3. Neo4j Engine

Neo4j

Neo4j is a high-performance, *NoSQL* graph database with all the features of a mature and robust database.

NoSQLUnit supports *Neo4j* by using next classes:

Table 3.1. Lifecycle Management Rules

In Memory	`com.lordofthejars.nosqlunit.neo4j.InMemoryNeo4j`
Embedded	`com.lordofthejars.nosqlunit.neo4j.EmbeddedNeo4j`
Managed Wrapping	`com.lordofthejars.nosqlunit.neo4j.ManagedWrappi`
Managed	`com.lordofthejars.nosqlunit.neo4j.ManagedNeoSer`

Table 3.2. Manager Rule

NoSQLUnit Management	`com.lordofthejars.nosqlunit.neo4j.Neo4jRule`

Maven Setup

To use **NoSQLUnit** with Neo4j you only need to add next dependency:

Example 3.1. NoSqlUnit Maven Repository

```
<dependency>
 <groupId>com.lordofthejars</groupId>
 <artifactId>nosqlunit-neo4j</artifactId>
 <version>${version.nosqlunit}</version>
</dependency>
```

Dataset Format

Default dataset file format in *Neo4j* module is GraphML [http://graphml.graphdrawing.org/] . *GraphML* is a comprehensive and easy-to-use file format for graphs.

Datasets must have next format :

Example 3.2. Example of GraphML Dataset

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
    <key id="attr1" for="edge" attr.name="attr1" attr.type="float"/>
    <key id="attr2" for="node" attr.name="attr2" attr.type="string"/>
    <graph id="G" edgedefault="directed">
        <node id="1">
            <data key="attr2">value1</data>
        </node>
        <node id="2">
            <data key="attr2">value2</data>
        </node>
        <edge id="7" source="1" target="2" label="label1">
            <data key="attr1">float</data>
        </edge>
    </graph>
</graphml>
```

where:

- *graphml* : the root element of the GraphML document
- *key* : description for graph element properties, you must define if property type is for nodes or relationships, name, and type of element. In our case string, int, long, float, double and boolean are supported.
- *graph* : the beginning of the graph representation. In our case only one level of graphs are supported. Inner graphs will be ignored.
- *node* : the beginning of a vertex representation. Please note that id 0 is reserved for reference node, so cannot be used as id.
- *edge* : the beginning of an edge representation. Source and target attributes are filled with node id. If you want to link with reference node, use a 0 which is the id of root node. Note that label attribute is not in defined in standard definition of GraphML specification; GraphML supports adding new attributes to all GrpahML elements, and label attribute has been added to facilitate the creation of edge labels.
- *data* : the key/value data associated with a graph element. Data value will be validated against type defined in key element.

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 in-memory approach, embedded approach, managed approach or remote approach.

In-memory Lifecycle

To configure **in-memory** approach you should only instantiate next rule :

Example 3.3. In-Memory Neo4j

```
import static com.lordofthejars.nosqlunit.neo4j.InMemoryNeo4j.InMemoryNeo4jRuleBui

@ClassRule
public static InMemoryNeo4j inMemoryNeo4j = newInMemoryNeo4j().build();
```

Embedded Lifecycle

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

Example 3.4. Embedded Neo4j

```
import static com.lordofthejars.nosqlunit.neo4j.EmbeddedNeo4j.EmbeddedNeo4jRuleBui

@ClassRule
public static EmbeddedNeo4j embeddedNeo4j = newEmbeddedNeo4jRule().build();
```

By default embedded *Neo4j* rule uses next default values:

Table 3.3. Default Embedded Values

Target path	This is the directory where *Neo4j* server is started and is `target/neo4j-temp` .

Managed Lifecycle

To configure managed way, two possible approaches can be used:

The first one is using an **embedded database wrapped by a server** . This is a way to give an embedded database visibility through network (internally we are creating a `WrappingNeoServerBootstrapper` instance) :

Example 3.5. Managed Wrapped Neo4j

```
import static com.lordofthejars.nosqlunit.neo4j.ManagedWrappingNeoServer.ManagedWr

@ClassRule
public static ManagedWrappingNeoServer managedWrappingNeoServer = newWrappingNeoSe
```

By default wrapped managed *Neo4j* rule uses next default values, but can be configured programmatically as shown in previous example :

Table 3.4. Default Wrapped Values

Target path	The directory where *Neo4j* server is started and is `target/neo4j-temp` .
Port	Where server is listening incoming messages is 7474.

The second strategy is **starting and stopping an already installed server** on executing machine, by calling start and stop command lines. Next rule should be registered:

Example 3.6. Managed Neo4j

```
import static com.lordofthejars.nosqlunit.neo4j.ManagedNeoServer.Neo4jServerRuleBu

@ClassRule
public static ManagedNeoServer managedNeoServer = newManagedNeo4jServerRule().neo4
```

By default managed *Neo4j* rule uses next default values, but can be configured programmatically as shown in previous example :

Table 3.5. Default Managed Values

Target path	This is the directory where *Neo4j* process will be started and by default is `target/neo4j-temp` .
Port	Where server is listening incoming messages is 7474.
Neo4jPath	*Neo4j* installation directory which by default is retrieved from `NEO4J_HOME` system environment variable.

Warning

Versions prior to *Neo4j* 1.8, port cannot be configured from command line, and port should be changed manually in `conf/neo4j-server.properties` . Although this restriction, if you have configured *Neo4j* to run through a different port, it should be specified too in `ManagedNeoServer` rule.

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 Neo4j Connection

Next step is configuring **Neo4j** rule in charge of maintaining *Neo4j* graph into known state by inserting and deleting defined datasets. You must register `Neo4jRule` *JUnit* rule class, which requires a configuration parameter with information like host, port, uri or target directory.

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

In-Memory/Embedded Connection

The first one is for configuring a connection to in-memory/embedded *Neo4j* .

Example 3.7. Neo4j with embedded configuration

```
import static com.lordofthejars.nosqlunit.neo4j.EmbeddedNeoServerConfigurationBuil

@Rule
public Neo4jRule neo4jRule = new Neo4jRule(newEmbeddedNeoServerConfiguration().bui
```

If you are only registering one embedded *Neo4j* instance like previous example , calling `build` is enough. If you are using more than one *Neo4j* embedded connection like explained in Simultaneous Engine section, `targetPath` shall be provided by using `buildFromTargetPath` method.

If you are using in-memory approach mixed with embedded approach, target path for in-memory instance can be found at `InMemoryNeo4j.INMEMORY_NEO4J_TARGET_PATH` variable.

Remote Connection

The second one is for configuring a connection to remote *Neo4j* server (it is irrelevant at this level if it is wrapped or not). Default values are:

Table 3.6. Default Managed Connection Values

Connection URI	http://localhost:7474/db/data
Authentication	No authentication parameters.

Example 3.8. Neo4j with managed configuration

```
import static com.lordofthejars.nosqlunit.neo4j.ManagedNeoServerConfigurationBuild

@Rule
public Neo4jRule neo4jRule = new Neo4jRule(newManagedNeoServerConfiguration().buil
```

Verifying Graph

`@ShouldMatchDataSet` is also supported for *Neo4j* graphs but we should keep in mind some considerations.

To compare two graphs, stored graph is exported into GraphML format and then is compared with expected *GraphML* using *XmlUnit* framework. This approach implies two aspects to be considered, the first one is that although your graph does not contains any connection to reference node, reference node will appear too with the form (<node id="0"></node>). The other aspect is that id's are *Neo4j's* internal id, so when you write the expected file, remember to follow the same id strategy followed by *Neo4j* so id attribute of each node could be matched correctly with generated output. Inserted nodes' id starts from 1 (0 is reserved for reference node), meanwhile edges starts from 0.

This way to compare graphs may change in future (although this strategy will be always supported).

As I have noted in verification section I find that using `@ShouldMatchDataSet` is a bad approach during testing because test readibility is affected negatively. So as general guide, my advice is to try to avoid using @ShouldMatchDataSet in your tests as much as possible.

Full Example

To show how to use **NoSQLUnit** with *Neo4j* , we are going to create a very simple application that counts Neo's friends.

MatrixManager is the business class responsible of inserting new friends and counting the number of Neo's friends.

Example 3.9. Neo4j with managed configuration

```
public class MatrixManager {

 public enum RelTypes implements RelationshipType {
  NEO_NODE, KNOWS, CODED_BY
 }

 private GraphDatabaseService graphDb;

 public MatrixManager(GraphDatabaseService graphDatabaseService) {
  this.graphDb = graphDatabaseService;
 }

 public int countNeoFriends() {

  Node neoNode = getNeoNode();
  Traverser friendsTraverser = getFriends(neoNode);

  return friendsTraverser.getAllNodes().size();

 }

 public void addNeoFriend(String name, int age) {
  Transaction tx = this.graphDb.beginTx();
  try {
   Node friend = this.graphDb.createNode();
   friend.setProperty("name", name);
   Relationship relationship = getNeoNode().createRelationshipTo(friend, RelTypes.
   relationship.setProperty("age", age);
   tx.success();
  } finally {
   tx.finish();
  }
 }

 private static Traverser getFriends(final Node person) {
  return person.traverse(Order.BREADTH_FIRST, StopEvaluator.END_OF_GRAPH, Returnab
    RelTypes.KNOWS, Direction.OUTGOING);
 }

 private Node getNeoNode() {
  return graphDb.getReferenceNode().getSingleRelationship(RelTypes.NEO_NODE, Direc
 }

}
```

And now one unit test and one integration test is written:

For unit test we are going to use embedded approach:

Example 3.10. Neo4j with managed configuration

```
import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.is;
import static com.lordofthejars.nosqlunit.neo4j.EmbeddedNeo4j.EmbeddedNeo4jRuleBui
import static com.lordofthejars.nosqlunit.neo4j.EmbeddedNeoServerConfigurationBuil

import javax.inject.Inject;

import org.junit.ClassRule;
import org.junit.Rule;
import org.junit.Test;
import org.neo4j.graphdb.GraphDatabaseService;

import com.lordofthejars.nosqlunit.annotation.UsingDataSet;
import com.lordofthejars.nosqlunit.core.LoadStrategyEnum;
import com.lordofthejars.nosqlunit.neo4j.EmbeddedNeo4j;
import com.lordofthejars.nosqlunit.neo4j.Neo4jRule;

public class WhenNeoFriendsAreRequired {

 @ClassRule
 public static EmbeddedNeo4j embeddedNeo4j = newEmbeddedNeo4jRule().build();

 @Rule
 public Neo4jRule neo4jRule = new Neo4jRule(newEmbeddedNeoServerConfiguration().bu

 @Inject
 private GraphDatabaseService graphDatabaseService;

 @Test
 @UsingDataSet(locations="matrix.xml", loadStrategy=LoadStrategyEnum.CLEAN_INSERT)
 public void all_direct_and_inderectly_friends_should_be_counted() {
  MatrixManager matrixManager = new MatrixManager(graphDatabaseService);
  int countNeoFriends = matrixManager.countNeoFriends();
  assertThat(countNeoFriends, is(3));
 }

}
```

And as integration test , the managed one:

Example 3.11. Neo4j with managed configuration

```
import static com.lordofthejars.nosqlunit.neo4j.ManagedWrappingNeoServer.ManagedWr
import static com.lordofthejars.nosqlunit.neo4j.ManagedNeoServerConfigurationBuild

import javax.inject.Inject;

import org.junit.ClassRule;
import org.junit.Rule;
import org.junit.Test;
import org.neo4j.graphdb.GraphDatabaseService;

import com.lordofthejars.nosqlunit.annotation.ShouldMatchDataSet;
import com.lordofthejars.nosqlunit.annotation.UsingDataSet;
import com.lordofthejars.nosqlunit.core.LoadStrategyEnum;
import com.lordofthejars.nosqlunit.neo4j.ManagedWrappingNeoServer;
import com.lordofthejars.nosqlunit.neo4j.Neo4jRule;

public class WhenNeoMeetsANewFriend {

 @ClassRule
 public static ManagedWrappingNeoServer managedWrappingNeoServer = newWrappingNeoS

 @Rule
 public Neo4jRule neo4jRule = new Neo4jRule(newManagedNeoServerConfiguration().bui

 @Inject
 private GraphDatabaseService graphDatabaseService;

 @Test
 @UsingDataSet(locations="matrix.xml", loadStrategy=LoadStrategyEnum.CLEAN_INSERT)
 @ShouldMatchDataSet(location="expected-matrix.xml")
 public void friend_should_be_related_into_neo_graph() {

  MatrixManager matrixManager = new MatrixManager(graphDatabaseService);
  matrixManager.addNeoFriend("The Oracle", 4);
 }

}
```

Note that in both cases we are using the same dataset as initial state, which looks like:

Example 3.12. matrix.xml Neo4j file

```
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
        http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
    <key id="name" for="node" attr.name="name" attr.type="string"/>
    <key id="age" for="edge" attr.name="age" attr.type="int"/>
    <graph id="G" edgedefault="directed">
        <node id="1">
            <data key="name">Thomas Anderson</data>
        </node>
        <node id="2">
            <data key="name">Trinity</data>
        </node>
        <node id="3">
            <data key="name">Morpheus</data>
        </node>
        <node id="4">
            <data key="name">Agent Smith</data>
        </node>
        <node id="5">
            <data key="name">The Architect</data>
        </node>
        <edge id="1" source="0" target="1" label="NEO_NODE">
        </edge>
        <edge id="2" source="1" target="2" label="KNOWS">
            <data key="age">3</data>
        </edge>
        <edge id="3" source="1" target="3" label="KNOWS">
            <data key="age">5</data>
        </edge>
        <edge id="4" source="2" target="3" label="KNOWS">
            <data key="age">18</data>
        </edge>
        <edge id="5" source="3" target="4" label="KNOWS">
            <data key="age">20</data>
        </edge>
        <edge id="6" source="4" target="5" label="CODED_BY">
            <data key="age">20</data>
        </edge>
    </graph>
</graphml>
```

Chapter 4. Cassandra Engine

Cassandra

Cassandra is a BigTable data model running on an Amazon Dynamo-like infrastructure.

NoSQLUnit supports *Cassandra* by using next classes:

Table 4.1. Lifecycle Management Rules

Embedded	`com.lordofthejars.nosqlunit.cassandra.EmbeddedC`
Managed	`com.lordofthejars.nosqlunit.cassandra.ManagedCa`

Table 4.2. Manager Rule

NoSQLUnit Management	`com.lordofthejars.nosqlunit.cassandra.Cassandra`

Maven Setup

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

Example 4.1. NoSqlUnit Maven Repository

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

Dataset Format

Default dataset file format in *Cassandra* module is json. To make compatible **NoSQLUnit** with Cassandra-Unit [https://github.com/jsevellec/cassandra-unit/] file format, `DataLoader` of Cassandra-Unit project is used, so same json format file is used.

Datasets must have next format :

Example 4.2. Example of Casssandra Dataset

```
{
    "name" : "",
    "replicationFactor" : "",
    "strategy" : "",
    "columnFamilies" : [{
        "name" : "",
        "type" : "",
        "keyType" : "",
        "comparatorType" : "",
        "subComparatorType" : "",
        "defaultColumnValueType" : "",
        "comment" : "",
        "compactionStrategy" : "",
        "compactionStrategyOptions" : [{
            "name" : "",
            "value": ""
        }],
        "gcGraceSeconds" : "",
        "maxCompactionThreshold" : "",
        "minCompactionThreshold" : "",
        "readRepairChance" : "",
        "replicationOnWrite" : "",
        "columnsMetadata" : [{
            "name" : "",
            "validationClass : "",
            "indexType" : "",
            "indexName" : ""
        },
        ...
        ]
        "rows" : [{
            "key" : "",
            "columns" : [{
                "name" : "",
                "value" : ""
            },
            ...
            ],
            ...
            // OR
            ...
            "superColumns" : [{
                "name" : "",
                "columns" : [{
                    "name" : "",
                    "value" : ""
                },
                ...
                ]
            },
            ...
            ]
        },
        ...
        ]
    },
    ...
    ]
}
```

See Cassandra-Unit Dataset [https://github.com/jsevellec/cassandra-unit/wiki/What-can-you-set-into-a-dataSet] format for more information.

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 4.3. Embedded Cassandra

```
@ClassRule
public static EmbeddedCassandra embeddedCassandraRule = newEmbeddedCassandraRule()
```

By default embedded *Cassandra* rule uses next default values:

Table 4.3. Default Embedded Values

Target path	This is the directory where *Cassandra* server is started and is `target/cassandra-temp` .
Cassandra Configuration File	Location of yaml configuration file. By default a configuration file is provided with correct default parameters.
Host	localhost
Port	By default port used is 9171. Port cannot be configured, and cannot be changed if you provide an alternative Cassandra Configuration File.

Managed Lifecycle

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

Example 4.4. Managed Cassandra

```
@ClassRule
public static ManagedCassandra managedCassandra = newManagedCassandraRule().build(
```

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

Table 4.4. Default Managed Values

Target path	This is the directory where *Cassandra* server is started and is `target/cassandra-temp` .
CassandraPath	*Cassandra* installation directory which by default is retrieved from `CASSANDRA_HOME` system environment variable.

Port	By default port used is 9160. If port is changed in *Cassandra* configuration file, this port should be configured too here.

Warning

To start *Cassandra*`java.home` must be set. Normally this variable is already configured, you would need to do nothing.

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 Cassandra Connection

Next step is configuring **Cassandra** rule in charge of maintaining *Cassandra* graph into known state by inserting and deleting defined datasets. You must register `CassandraRule` *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 a connection to embedded *Cassandra* .

Example 4.5. Cassandra with embedded configuration

```
import static com.lordofthejars.nosqlunit.cassandra.EmbeddedCassandraConfiguration

@Rule
public CassandraRule cassandraRule = new CassandraRule(newEmbeddedCassandraConfigu
```

Host and port parameters are already configured.

Managed Connection

The first one is for configuring a connection to managed *Cassandra* .

Example 4.6. Cassandra with managed configuration

```
import static com.lordofthejars.nosqlunit.cassandra.ManagedCassandraConfigurationB

@Rule
public CassandraRule cassandraRule = new CassandraRule(newManagedCassandraConfigur
```

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 *Cassandra* .

Example 4.7. Cassandra with remote configuration

```
import static com.lordofthejars.nosqlunit.cassandra.RemoteCassandraConfigurationBu

@Rule
public CassandraRule cassandraRule = new CassandraRule(newRemoteCassandraConfigura
```

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.

Verifying Data

`@ShouldMatchDataSet` is also supported for *Cassandra* data but we should keep in mind some considerations.

Warning

In **NoSQLUnit** , expectations can only be used over data, not over configuration parameters, so for example fields set in dataset file like compactionStrategy, gcGraceSeconds or maxCompactionThreshold are not used. Maybe in future will be supported but for now only data (keyspace, columnfamilyname, columns, supercolumns, ...) are supported.

Full Example

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

PersonManager is the business class responsible of getting and updating person's car.

Example 4.8. PersonCar cassandra with manager.

```
public class PersonManager {

 private ColumnFamilyTemplate<String, String> template;

 public PersonManager(String clusterName, String keyspaceName, String host) {
  Cluster cluster = HFactory.getOrCreateCluster(clusterName, host);
  Keyspace keyspace = HFactory.createKeyspace(keyspaceName, cluster);

        template = new ThriftColumnFamilyTemplate<String, String>(keyspace,
          "personFamilyName",
                                                                  StringSerializer.ge
                                                                  StringSerializer.ge

 }

 public String getCarByPersonName(String name) {
  ColumnFamilyResult<String, String> queryColumns = template.queryColumns(name);
  return queryColumns.getString("car");
 }

 public void updateCarByPersonName(String name, String car) {
  ColumnFamilyUpdater<String, String> createUpdater = template.createUpdater(name)
  createUpdater.setString("car", car);

  template.update(createUpdater);
 }

}
```

And now one unit test and one integration test is written:

For unit test we are going to use embedded approach:

Example 4.9. Cassandra with embedded configuration

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

import static com.lordofthejars.nosqlunit.cassandra.EmbeddedCassandra.EmbeddedCass
import static com.lordofthejars.nosqlunit.cassandra.EmbeddedCassandraConfiguration

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

import com.lordofthejars.nosqlunit.annotation.UsingDataSet;
import com.lordofthejars.nosqlunit.cassandra.CassandraRule;
import com.lordofthejars.nosqlunit.cassandra.EmbeddedCassandra;
import com.lordofthejars.nosqlunit.core.LoadStrategyEnum;

public class WhenPersonWantsToKnowItsCar {

 @ClassRule
 public static EmbeddedCassandra embeddedCassandraRule = newEmbeddedCassandraRule(

 @Rule
 public CassandraRule cassandraRule = new CassandraRule(newEmbeddedCassandraConfig


 @Test
 @UsingDataSet(locations="persons.json", loadStrategy=LoadStrategyEnum.CLEAN_INSER
 public void car_should_be_returned() {

  PersonManager personManager = new PersonManager("Test Cluster", "persons", "loca
  String car = personManager.getCarByPersonName("mary");

  assertThat(car, is("ford"));

 }

}
```

And as integration test , the managed one:

Example 4.10. Cassandra with managed configuration

```
import static com.lordofthejars.nosqlunit.cassandra.ManagedCassandraConfigurationB
import static com.lordofthejars.nosqlunit.cassandra.ManagedCassandra.ManagedCassan

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

import com.lordofthejars.nosqlunit.annotation.ShouldMatchDataSet;
import com.lordofthejars.nosqlunit.annotation.UsingDataSet;
import com.lordofthejars.nosqlunit.cassandra.CassandraRule;
import com.lordofthejars.nosqlunit.cassandra.ManagedCassandra;
import com.lordofthejars.nosqlunit.core.LoadStrategyEnum;

public class WhenPersonWantsToUpdateItsCar {

 static {
  System.setProperty("CASSANDRA_HOME", "/opt/cassandra");
 }

 @ClassRule
 public static ManagedCassandra managedCassandra = newManagedCassandraRule().build

 @Rule
 public CassandraRule cassandraRule = new CassandraRule(newManagedCassandraConfigu

 @Test
 @UsingDataSet(locations="persons.json", loadStrategy=LoadStrategyEnum.CLEAN_INSER
 @ShouldMatchDataSet(location="expected-persons.json")
 public void new_car_should_be_updated() {

  PersonManager personManager = new PersonManager("Test Cluster", "persons", "loca
  personManager.updateCarByPersonName("john", "opel");

 }

}
```

Note that in both cases we are using the same dataset as initial state, which looks like:

Example 4.11. persons.json Cassandra file

```
{
    "name" : "persons",
    "columnFamilies" : [{
        "name" : "personFamilyName",
 "keyType" : "UTF8Type",
 "defaultColumnValueType" : "UTF8Type",
 "comparatorType" : "UTF8Type",
        "rows" : [{
            "key" : "john",
            "columns" : [{
                "name" : "age",
                "value" : "22"
            },
            {
                "name" : "car",
                "value" : "toyota"
            }]
        },
        {
            "key" : "mary",
            "columns" : [{
                "name" : "age",
                "value" : "33"
            },
            {
                "name" : "car",
                "value" : "ford"
            }]
        }]
    }]
}
```

Chapter 5. Redis Engine

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 5.1. Lifecycle Management Rules

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

Table 5.2. Manager Rule

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

Maven Setup

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

Example 5.1. 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 5.2. 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 5.3. Embedded Redis

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

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

Table 5.3. 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 5.4. 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 5.4. 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 5.5. 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 5.6. 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 5.7. 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 5.8. 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 5.9. 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 5.10. 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 5.11. book.json Redis file

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

Chapter 6. HBase Engine

HBase

Apache HBase is an open-source, distributed, versioned, column-oriented store.

NoSQLUnit supports *HBase* by using next classes:

Table 6.1. Lifecycle Management Rules

Embedded	`com.lordofthejars.nosqlunit.hbase.EmbeddedHBase`
Managed	`com.lordofthejars.nosqlunit.hbase.ManagedHBase`

Table 6.2. Manager Rule

NoSQLUnit Management	`com.lordofthejars.nosqlunit.hbase.HBaseRule`

Maven Setup

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

Example 6.1. NoSqlUnit Maven Repository

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

Dataset Format

Default dataset file format in *HBase* module is json. Dataset in HBase is the same used by Cassandra-Unit [https://github.com/jsevellec/cassandra-unit/] but not all fields are supported. Only fields available in TSV HBase application can be set into dataset.

So as summary datasets must have next format :

Example 6.2. Example of HBase Dataset

```
{
    "name" : "tablename",
    "columnFamilies" : [{
        "name" : "columnFamilyName",
        "rows" : [{
            "key" : "key1",
            "columns" : [{
                "name" : "columnName",
                "value" : "columnValue"
            },
            ...
            ]
        },
        ...
        ]
    },
    ...
    ]
}
```

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 6.3. Embedded HBase

```
@ClassRule
public static EmbeddedHBase embeddedHBase = newEmbeddedHBaseRule().build();
```

By default embedded *Embedded* rule uses `HBaseTestingUtility` default values:

Table 6.3. Default Embedded Values

Target path	This is the directory where *HBase* stores data and is `target/data` .
Host	localhost
Port	By default port used is 60000.
File Permissions	Depending on your umask configuration, `HBaseTestingUtility` will create some directories that will not be accessible during runtime. By default this value is set to 775, but depending on your OS you may require a different value.

Managed Lifecycle

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

Example 6.4. Managed HBase

```
@ClassRule
public static ManagedHBase managedHBase = newManagedHBaseServerRule().build();
```

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

Table 6.4. Default Managed Values

Target path	This is the directory where *HBase* server is started and is `target/hbase-temp` .
CassandraPath	*HBase* installation directory which by default is retrieved from `HBASE_HOME` system environment variable.
Port	By default port used is 60000. If port is changed in *HBase* configuration file, this port should be configured too here.

> **Warning**
> To start *HBASE*`JAVA_HOME` must be set. Normally this variable is already configured, so you would need to do nothing.

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 HBase Connection

Next step is configuring **HBase** rule in charge of maintaining *HBase* columns into known state by inserting and deleting defined datasets. You must register `HBaseRule` *JUnit* rule class, which requires a configuration parameter with some information.

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 a connection to embedded *HBase* .

Example 6.5. HBase with embedded configuration

```
import static com.lordofthejars.nosqlunit.hbase.EmbeddedHBase.EmbeddedHBaseRuleBui

@Rule
public HBaseRule hBaseRule = newHBaseRule().defaultEmbeddedHBase();
```

Embedded HBase does not require any special parameter. Configuration object is copied from Embedded rule directly to HBaseRule.

Managed Connection

This is for configuring a connection to managed *HBase* .

Example 6.6. HBase with managed configuration

```
import static com.lordofthejars.nosqlunit.hbase.ManagedHBaseConfigurationBuilder.n

@Rule
public HBaseRule hbaseRule = new HBaseRule(newManagedHBaseConfiguration().build())
```

By default configuration used is the one loaded by calling HBaseConfiguration.create() method. HBaseConfiguration.create() [http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/HBaseConfiguration.html#create()] which uses hbase-site.xml and hbase-default.xml classpath files.

But also a method `setProperty` method is provided to modify any parameter of generated configuration object.

Remote Connection

Configuring a connection to remote *HBase* uses same approach like ManagedHBase configuration object but using `com.lordofthejars.nosqlunit.hbase.RemoteHBaseConfigurationBuilder` class instead of com.lordofthejars.nosqlunit.hbase.ManagedHBaseConfigurationBuilder. .

Warning

Working with Apache HBase required a bit of knowledge about how it works. For example your /etc/hosts file cannot contain a reference to your host name with ip 127.0.1.1.

Moreover **NoSQLUnit** uses *HBase-0.94.1* and this version should be also installed in your computer to work with managed or remote approach. If you install another version, you should exclude these artifacts from **NoSQLUnit** dependencies, and add the new ones manually to your pom file.

Verifying Data

`@ShouldMatchDataSet` is also supported for *HBase* data but we should keep in mind some considerations.

If you plan to verify data with `@ShouldMatchDataSet` in Managed and Remote approach, you should enable Aggregate coprocessor by editing hbase-site-xml file and adding next lines:

Example 6.7. HBase with coprocessor

```
<property>
    <name>hbase.coprocessor.user.region.classes</name>
    <value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>
</property>
```

Full Example

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

PersonManager is the business class responsible of getting and updating person's car.

Example 6.8. PersonCar cassandra with manager.

```
public class PersonManager {

 private Configuration configuration;

 public PersonManager(Configuration configuration) {
  this.configuration = configuration;
 }

 public String getCarByPersonName(String personName) throws IOException {
  HTable table = new HTable(configuration, "person");
  Get get = new Get("john".getBytes());
  Result result = table.get(get);

  return new String(result.getValue(toByteArray().convert("personFamilyName"), toB
 }

 private Converter<String, byte[]> toByteArray() {
  return new Converter<String, byte[]>() {

   @Override
   public byte[] convert(String element) {
    return element.getBytes();
   }
  };
 }

}
```

And now one unit test is written:

For unit test we are going to use embedded approach:

Example 6.9. HBase with embedded configuration

```
public class WhenPersonWantsToKnowItsCar {

 @ClassRule
 public static EmbeddedHBase embeddedHBase = newEmbeddedHBaseRule().build();

 @Rule
 public HBaseRule hBaseRule = newHBaseRule().defaultEmbeddedHBase(this);

 @Inject
 private Configuration configuration;


 @Test
 @UsingDataSet(locations="persons.json", loadStrategy=LoadStrategyEnum.CLEAN_INSER
 public void car_should_be_returned() throws IOException {

  PersonManager personManager = new PersonManager(configuration);
  String car = personManager.getCarByPersonName("john");

  assertThat(car, is("toyota"));
 }

}
```

And dataset used is:

Example 6.10. persons.json HBase file

```
{
    "name" : "person",
    "columnFamilies" : [{
        "name" : "personFamilyName",
        "rows" : [{
            "key" : "john",
            "columns" : [{
                "name" : "age",
                "value" : "22"
            },
            {
                "name" : "car",
                "value" : "toyota"
            }]
        },
        {
            "key" : "mary",
            "columns" : [{
                "name" : "age",
                "value" : "33"
            },
            {
                "name" : "car",
                "value" : "ford"
            }]
        }]
    }]
}
```

Chapter 7. CouchDB Engine

CouchDB

CouchDB is a *NoSQL* database that stores structured data as *JSON-like* documents with dynamic schemas.

NoSQLUnit supports *CouchDB* by using next classes:

Table 7.1. Lifecycle Management Rules

Managed	`com.lordofthejars.nosqlunit.couchdb.ManagedCouc`

Table 7.2. Manager Rule

NoSQLUnit Management	`com.lordofthejars.nosqlunit.couchdb.CouchDbRule`

Maven Setup

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

Example 7.1. NoSqlUnit Maven Repository

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

Dataset Format

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

Datasets must have next format :

Example 7.2. Example of CouchDB Dataset

```
{
 "data":
   [
    {"attribute1":"value1", "atribute2":"value2", ...},
    {...}
   ]
}
```

Notice that if attributes value are integers, double quotes are not required.

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 **managed** approach or **remote** approach.

There is no *CouchDB* inmemory instance, so only managed or remote lifecycle can be used.

To configure the **managed** way, you should use `ManagedCouchDb` rule and may require some configuration parameters.

Example 7.3. Managed CouchDB

```
import static com.lordofthejars.nosqlunit.couchdb.ManagedCouchDb.ManagedCouchDbRul

@ClassRule
public static ManagedCouchDb managedCouchDb = newManagedCouchDbRule().couchDbPath(
```

By default managed *CouchDB* rule uses next default values:

- *CouchDB* installation directory is retrieved from `COUCHDB_HOME` system environment variable.
- Target path, that is the directory where *CouchDB* server is started, is `target/couchdb-temp` .
- Port where *CouchDB* will be started. Note that this parameter is used only as information, if you change port from configuration file you should change this parameter too. By default*CouchDB* server is started at `5984` .

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 CouchDB Connection

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

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

Table 7.3. Default Managed Configuration Values

URI	http://localhost5984
Authentication	No authentication parameters.
Enable SSL	false.
Relaxed SSL Settings	false.
Caching	True.

Example 7.4. CouchDBRule with managed configuration

```
import static com.lordofthejars.nosqlunit.couchdb.CouchDbRule.CouchDbRuleBuilder.n

@Rule
public CouchDbRule couchDbRule = newCouchDbRule().defaultManagedMongoDb("books");
```

Complete Example

Consider a library application, which apart from multiple operations, it allow us to add new books to system. Our model is as simple as:

Example 7.5. Book POJO

```
public class Book {

 private String title;

 private int numberOfPages;

 public Book(String title, int numberOfPages) {
  super();
  this.title = title;
  this.numberOfPages = numberOfPages;
 }

 public void setTitle(String title) {
  this.title = title;
 }

 public void setNumberOfPages(int numberOfPages) {
  this.numberOfPages = numberOfPages;
 }


 public String getTitle() {
  return title;
 }

 public int getNumberOfPages() {
  return numberOfPages;
 }
}
```

Next business class is the responsible of managing access to *CouchDB* server:

Example 7.6. Book POJO

```
private CouchDbConnector connector;

 public BookManager(CouchDbConnector connector)  {
  this.connector = connector;
 }

 public void create(Book book) {
  connector.create(MapBookConverter.toMap(book));
 }

 public Book findBookById(String id) {
  Map<String, Object> map = connector.get(Map.class, id);
  return MapBookConverter.toBook(map);
 }
```

And now it is time for testing. In next test we are going to validate that a book is found into database.

Example 7.7. Test with Managed Connection

```
public class WhenYouFindBooksById {

 @ClassRule
 public static ManagedCouchDb managedCouchDb = newManagedCouchDbRule().couchDbPath

 @Rule
 public CouchDbRule couchDbRule = newCouchDbRule().defaultManagedMongoDb("books");

 @Inject
 private CouchDbConnector couchDbConnector;

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

  BookManager bookManager = new BookManager(couchDbConnector);
  Book book = bookManager.findBookById("1");

  assertThat(book.getTitle(), is("The Hobbit"));
  assertThat(book.getNumberOfPages(), is(293));

 }

}
```

Example 7.8. Initial Dataset

```
{
 "data":
   [
    {"_id":"1", "title":"The Hobbit","numberOfPages":"293"}
   ]
}
```

You can watch full example at github [https://github.com/lordofthejars/nosql-unit/tree/master/nosqlunit-demo] .

Part III. Advanced Usage

This chapter provides some examples of advanced features of **NoSQLUnit** not described in previous chapters.

Chapter 8. Advanced Usage

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 8.1. 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 8.2. 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 8.3. 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 8.4. 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 8.5. 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 8.6. 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 8.7. 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 8.8. 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 8.9. 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 8.10. 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;
```

Part IV. Stay In Touch

This chapter provides information about next releases and how to stay in touch with the project.

Chapter 9. Stay In Touch

Future releases

Version 0.4.0 will have support for *Neo4J and Cassandra.*

Next versions will contain support for *HBase* and *CouchDb* .

Stay in Touch

Table 9.1.

Email:	asotobu at gmail.com
Blog:	Lord Of The Jars [www.lordofthejars.com]
Twitter:	@alexsotob
Github:	NoSQLUnit Github [https://github.com/lordofthe-jars/nosql-unit/]