# NoSQLUnit Reference Manual

**Alex Soto, www.lordofthejars.com**

# NoSQLUnit Reference Manual

by Alex Soto

0.3.2

# Table of Contents

# List of Tables

# List of Examples

# Part I. What's New?

For those who are already familiar with **NoSQLUnit**, this chapter provides a brief overview of the new features. In this section, only last 5 versions will be highlighted.

# Chapter 1. What's New?

## Neo4j Support

**NoSQLUnit** supports *Neo4j* by using next classes:

### Table 1.1. Lifecycle Management Rules

| | |
|---|---|
| Embedded | `com.lordofthejars.nosqlunit.neo4j.EmbeddedNeo4j` |
| Managed Wrapping | `com.lordofthejars.nosqlunit.neo4j.ManagedWrappi` |
| Managed | `com.lordofthejars.nosqlunit.neo4j.ManagedNeoSer` |

### Table 1.2. Manager Rule

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

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.

### Example 1.1. 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>
```

A simple example of using embedded *Neo4j* lifecycle management could be:

### Example 1.2. Embedded Neo4j

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

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

And for configuring *Neo4j* connection:

**Example 1.3. Neo4j with embedded configuration**

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

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

## 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 1.4. 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 1.5. 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 1.6. Selective expectation example**

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

For more information see chapter about advanced features .

## 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 1.7. 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.

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 1.8. 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;
```

For more information see advanced features chapter.

# Part II. 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 2. 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 backend 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 2.1. 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. |
| REFRESH | Insert all data defined in datasets that are not present on database. |

An example of usage:

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

## 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 III. Supported Engines

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

# Chapter 3. 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 3.1. Lifecycle Management Rules**

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

**Table 3.2. Manager Rule**

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

## Maven Setup

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

**Example 3.1. 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 3.2. jmockmongo Maven Repository**

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

**Example 3.3. 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 3.4. 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 3.5. 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 3.6. 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` .
- *Mongodb* is started with *fork* option.
- 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 3.7. 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 3.3. 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 *jmock-mongo* , no modifications are required.

### Example 3.8. 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 3.4. Default Managed Configuration Values

| | |
|---|---|
| Host | localhost |
| Port | 27017 |
| Authentication | No authentication parameters. |

### Example 3.9. MongoDbRule with managed configuration

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

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

### Example 3.10. 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 3.11. 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 3.12. 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 3.13. 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 3.14. Initial Dataset

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

### Example 3.15. 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 4. 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 4.1. Lifecycle Management Rules**

| | |
|---|---|
| Embedded | `com.lordofthejars.nosqlunit.neo4j.EmbeddedNeo4j` |
| Managed Wrapping | `com.lordofthejars.nosqlunit.neo4j.ManagedWrappi` |
| Managed | `com.lordofthejars.nosqlunit.neo4j.ManagedNeoSer` |

**Table 4.2. Manager Rule**

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

## Maven Setup

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

**Example 4.1. 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 4.2. 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 embedded approach, managed approach or remote approach. Note that there is no implementation of a *Neo4j* in-memory database at this time, but embedded strategy for unit tests will be the better one.

#### Embedded Lifecycle

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

### Example 4.3. 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 4.3. 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 `WrappingNeoServerBootstrap-per` instance) :

### Example 4.4. 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 4.4. 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 4.5. 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 4.5. 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.

### Embedded Connection

The first one is for configuring a connection to embedded *Neo4j* .

#### Example 4.6. 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.

### 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 4.6. Default Managed Connection Values

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

**Example 4.7. 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 4.8. 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 4.9. 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 4.10. 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 4.11. Neo4j with managed configuration**

```
<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>
```

# Part IV. Advanced Usage

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

# Chapter 5. Advanced Usage

## 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 5.1. 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 5.2. 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 5.3. 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 5.4. 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 5.5. 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.

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 5.6. 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 V. Stay In Touch

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

# Chapter 6. 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 6.1.**

| | |
|---|---|
| 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/] |