Chapter 1. Neo4j Engine

Table of Contents

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 1.1. 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 1.2. Manager Rule

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

Maven Setup

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

Example 1.1. 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 1.2. 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 1.3. 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 1.4. 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 1.3. 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 1.5. 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 1.4. 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 1.6. 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 1.5. 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 1.7. 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 1.6. Default Managed Connection Values

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

Example 1.8. 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 1.9. 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 1.10. 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 1.11. 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 1.12. 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>
```