Chapter 1. Cassandra Engine

Table of Contents

Cassandra

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

NoSQLUnit supports *Cassandra* by using next classes:

Table 1.1. Lifecycle Management Rules

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

Table 1.2. Manager Rule

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

Maven Setup

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

Example 1.1. 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 1.2. 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 1.3. Embedded Cassandra

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

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

Table 1.3. 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 1.4. Managed Cassandra

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

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

Table 1.4. 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 1.5. 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 1.6. 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 1.7. 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 1.8. 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 1.9. 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 1.10. 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 1.11. 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"
            }]
        }]
    }]
}
```