Chapter 1. NoSQLUnit Core

Table of Contents

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")
```