NoSQLUnit

Version 0.3.0

Alex Soto, www.lordofthejars.com

2012

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. 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. 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 if it is not 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")
```

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. Lifecycle Management Rules

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

Table 3. Manager Rule

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

Maven Setup

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

Example 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. jmockmongo Maven Repository

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

Example 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 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 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 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 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 4. 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 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 5. Default Managed Configuration Values

Host	localhost
Port	27017
Authentication	No authentication parameters.

Example 9. MongoDbRule with managed configuration

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

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

Example 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 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 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 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 14. Initial Dataset

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

Example 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] .

Future releases

In next project release 0.3.1, a new feature will be implemented. As features *@Inject* will be able to be used to have access to underlying connection into tests.

Version 0.4.0 there will be support for *Neo4J and Cassandra.*

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

Stay in Touch

Table 6.

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