Chapter 1. MongoDb Engine

Table of Contents

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 1.1. Lifecycle Management Rules

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

Table 1.2. Manager Rule

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

Maven Setup

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

Example 1.1. 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 1.2. jmockmongo Maven Repository

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

Example 1.3. 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 1.4. 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 1.5. 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 1.6. 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 1.7. 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 1.3. 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 *jmockmongo* , no modifications are required.

Example 1.8. 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 1.4. Default Managed Configuration Values

Host	localhost
Port	27017
Authentication	No authentication parameters.

Example 1.9. MongoDbRule with managed configuration

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

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

Example 1.10. 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 1.11. 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 1.12. 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 1.13. 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 1.14. Initial Dataset

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

Example 1.15. 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] .