Chapter 1. CouchDB Engine

Table of Contents

CouchDB

CouchDB is a *NoSQL* database that stores structured data as *JSON-like* documents with dynamic schemas.

NoSQLUnit supports *CouchDB* by using next classes:

Table 1.1. Lifecycle Management Rules

Managed	`com.lordofthejars.nosqlunit.couchdb.ManagedCouc`

Table 1.2. Manager Rule

NoSQLUnit Management	`com.lordofthejars.nosqlunit.couchdb.CouchDbRule`

Maven Setup

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

Example 1.1. NoSqlUnit Maven Repository

```
<dependency>
 <groupId>com.lordofthejars</groupId>
 <artifactId>nosqlunit-couchdb</artifactId>
 <version>${version.nosqlunit}</version>
</dependency>
```

Dataset Format

Default dataset file format in *CouchDB* module is *json* .

Datasets must have next format :

Example 1.2. Example of CouchDB Dataset

```
{
 "data":
   [
    {"attribute1":"value1", "atribute2":"value2", ...},
    {...}
   ]
}
```

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 **managed** approach or **remote** approach.

There is no *CouchDB* inmemory instance, so only managed or remote lifecycle can be used.

To configure the **managed** way, you should use `ManagedCouchDb` rule and may require some configuration parameters.

Example 1.3. Managed CouchDB

```
import static com.lordofthejars.nosqlunit.couchdb.ManagedCouchDb.ManagedCouchDbRul

@ClassRule
public static ManagedCouchDb managedCouchDb = newManagedCouchDbRule().couchDbPath(
```

By default managed *CouchDB* rule uses next default values:

- *CouchDB* installation directory is retrieved from `COUCHDB_HOME` system environment variable.
- Target path, that is the directory where *CouchDB* server is started, is `target/couchdb-temp` .
- Port where *CouchDB* will be started. Note that this parameter is used only as information, if you change port from configuration file you should change this parameter too. By default*CouchDB* server is started at `5984` .

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 CouchDB Connection

Next step is configuring ***CouchDB*** rule in charge of maintaining *CouchDB* database into known state by inserting and deleting defined datasets. You must register `CouchDbRule` *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.

Table 1.3. Default Managed Configuration Values

URI	http://localhost5984
Authentication	No authentication parameters.
Enable SSL	false.
Relaxed SSL Settings	false.
Caching	True.

Example 1.4. CouchDBRule with managed configuration

```
import static com.lordofthejars.nosqlunit.couchdb.CouchDbRule.CouchDbRuleBuilder.n

@Rule
public CouchDbRule couchDbRule = newCouchDbRule().defaultManagedMongoDb("books");
```

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 1.5. 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 *CouchDB* server:

Example 1.6. Book POJO

```
private CouchDbConnector connector;

 public BookManager(CouchDbConnector connector)  {
  this.connector = connector;
 }

 public void create(Book book) {
  connector.create(MapBookConverter.toMap(book));
 }

 public Book findBookById(String id) {
  Map<String, Object> map = connector.get(Map.class, id);
  return MapBookConverter.toBook(map);
 }
```

And now it is time for testing. In next test we are going to validate that a book is found into database.

Example 1.7. Test with Managed Connection

```
public class WhenYouFindBooksById {

 @ClassRule
 public static ManagedCouchDb managedCouchDb = newManagedCouchDbRule().couchDbPath

 @Rule
 public CouchDbRule couchDbRule = newCouchDbRule().defaultManagedMongoDb("books");

 @Inject
 private CouchDbConnector couchDbConnector;

 @Test
 @UsingDataSet(locations="books.json", loadStrategy=LoadStrategyEnum.CLEAN_INSERT)
 public void identified_book_should_be_returned() {

  BookManager bookManager = new BookManager(couchDbConnector);
  Book book = bookManager.findBookById("1");

  assertThat(book.getTitle(), is("The Hobbit"));
  assertThat(book.getNumberOfPages(), is(293));

 }

}
```

Example 1.8. Initial Dataset

```
{
 "data":
   [
    {"_id":"1", "title":"The Hobbit","numberOfPages":"293"}
   ]
}
```

You can watch full example at github [https://github.com/lordofthejars/nosql-unit/tree/master/nosqlunit-demo] .