

# Ruling Database Testing with DBUnit Rules

# Table of Contents

1. Introduction .....	2
2. Setup DBUnit Rules .....	3
3. Example .....	5
4. Transactions .....	6
5. Database assertion with ExpectedDataSet .....	7
5.1. Regular expressions .....	8
6. Scriptable datasets .....	10
6.1. Javascript scriptable dataset .....	10
6.2. Groovy scriptable dataset .....	12
7. Multiple databases .....	13
8. Ruling database in CDI tests .....	15
8.1. Classpath dependencies .....	15
8.2. Configuration .....	16
8.3. Example .....	17
9. Ruling database in BDD tests .....	20

In this post I am going to talk about [DBUnit Rules](#), a small opensource project I maintain which aims to simplify database testing [1: In the context of this article, database testing stands for [JUnit](#) integration tests which depend on a **relational** database so application business logic that depend on a database can be tested without mocking.].

# Chapter 1. Introduction

**DBUnit Rules** integrates **JUnit** and **DBUnit** through **JUnit rules** and, in case of **CDI** based tests, a **CDI interceptor**. This powerful combination lets you easily prepare the database state for testing through **xml**, **json**, **xls** or **yaml** files.

Most inspiration of DBUnit Rules was taken from **Arquillian extension persistence** a library for database **in-container integration tests**.

Source code for the upcoming examples can be found at github here:  
<https://github.com/rmpestano/dbunit-rules-sample>

## Chapter 2. Setup DBUnit Rules

First thing to do is to add DBUnit Rules core module to your test classpath:

```
<dependency>
  <groupId>com.github.dbunit-rules</groupId>
  <artifactId>core</artifactId>
  <version>${dbunit-rules.version}</version>
  <scope>test</scope>
</dependency>
```

Secondly we need a database, for testing I recommend [HSQLDB](#) which is a very fast in-memory database, here is its maven dependency:

```
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.3.3</version>
  <scope>test</scope>
</dependency>
```

Later A JPA provider will be needed, in this case Hibernate will be used:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.2.8.Final</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.2.8.Final</version>
  <scope>test</scope>
</dependency>
```

And the entity manager persistence.xml:

*src/test/resources/META-INF/persistence.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="rulesDB" transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.github.dbunit.rules.sample.User</class>

    <properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"
/>
    <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"
/>
    <property name="javax.persistence.jdbc.url" value=
"jdbc:hsqldb:mem:test;DB_CLOSE_DELAY=-1" />
    <property name="javax.persistence.jdbc.user" value="sa" />
    <property name="javax.persistence.jdbc.password" value="" />
    <property name="hibernate.hbm2ddl.auto" value="create-drop" />
    <property name="hibernate.show_sql" value="true" />
    </properties>

  </persistence-unit>
</persistence>
```

and finally the JPA entity which our tests will work on:

```
@Entity
public class User {

    @Id
    @GeneratedValue
    private long id;

    private String name;
```

Now we are ready to rule our database tests!

# Chapter 3. Example

Create a yaml file which will be used to prepare database before the test:

*src/test/resources/dataset/users.yml*

```
user:
  - id: 1
    name: "@realpestano"
  - id: 2
    name: "@dbunit"
```

And the JUnit test:

```
@RunWith(JUnit4.class)
public class DBUnitRulesCoreTest {

    @Rule
    public EntityManagerProvider emProvider = EntityManagerProvider.instance("rulesDB"); ①

    @Rule
    public DBUnitRule dbUnitRule = DBUnitRule.instance(emProvider.connection()); ②

    @Test
    @DataSet("users.yml") ③
    public void shouldListUsers() {
        List<User> users = em(). ④
            createQuery("select u from User u").
            getResultList();
        assertThat(users).
            isNotNull().
            isEmpty().
            hasSize(2);
    }
}
```

- ① EntityManagerProvider is a JUnit rule that initializes a JPA entity manager before each **test class**. **rulesDB** is the name of persistence unit;
- ② DBUnit rule reads **@DataSet** annotations and initializes database before each **test method**. This rule only needs a **JDBC** connection to be created.
- ③ The dataSet configuration itself, [see here](#) for all available configuration options.
- ④ **em()** is a shortcut (`import static com.github.dbunit.rules.util.EntityManagerProvider.em;`) for the EntityManager that was initialized by EntityManagerProvider rule.

# Chapter 4. Transactions

EntityManagerProvider rule provides entity manager transactions so you can insert/delete entities in your tests:

```
@Test
@DataSet("users.yml")
public void shouldUpdateUser() {
    User user = (User) em().
        createQuery("select u from User u where u.id = 1").
        getSingleResult();
    assertThat(user).isNotNull();
    assertThat(user.getName()).isEqualTo("@realpestano");
    tx().begin(); ①
    user.setName("@rmpestano");
    em().merge(user);
    tx().commit();
    assertThat(user.getName()).isEqualTo("@rmpestano");
}

@Test
@DataSet("users.yml")
public void shouldDeleteUser() {
    User user = (User) em().
        createQuery("select u from User u where u.id = 1").
        getSingleResult();
    assertThat(user).isNotNull();
    assertThat(user.getName()).isEqualTo("@realpestano");
    tx().begin();
    em().remove(user);
    tx().commit();
    List<User> users = em().
        createQuery("select u from User u ").
        getResultList();
    assertThat(users).
        hasSize(1);
}
```

① `tx()` is a shortcut for the entity manager transaction provided by EntityManagerProvider.



# Chapter 5. Database assertion with ExpectedDataSet

Consider the following datasets:

*src/test/resources/dataset/users.yml*

```
user:
  - id: 1
    name: "@realpestando"
  - id: 2
    name: "@dbunit"
```

and expected dataset:

*src/test/resources/dataset/expectedUser.yml*

```
user:
  - id: 2
    name: "@dbunit"
```

And the following test:

```
@Test
@DataSet("users.yml")
@ExpectedDataSet(value = "expectedUser.yml", ignoreCols = "id") ①
public void shouldAssertDatabaseUsingExpectedDataSet() {
    User user = (User) em().
        createQuery("select u from User u where u.id = 1").
        getSingleResult();
    assertThat(user).isNotNull();
    tx().begin();
    em().remove(user);
    tx().commit();
}
```

① Database state after test will be compared with dataset provided by `@ExpectedDataSet`.

If database state is not equal then an assertion error is thrown, example imagine in test above we've deleted user with `id=2`, error would be:



```
junit.framework.ComparisonFailure: value (table=USER, row=0, col=name)
Expected :@dbunit
Actual   :@realpestano
<Click to see difference>
    at
org.dbunit.assertion.JUnitFailureFactory.createFailure(JUnitFailureFac
tory.java:39)
    at
org.dbunit.assertion.DefaultFailureHandler.createFailure(DefaultFailur
eHandler.java:97)
    at
org.dbunit.assertion.DefaultFailureHandler.handle(DefaultFailureHandle
r.java:223)
    at
com.github.dbunit.rules.assertion.DataSetAssert.compareData(DataSetAss
ert.java:94)
```

## 5.1. Regular expressions

Expected datasets also allow `regex` in datasets:

*src/test/resources/dataset/expectedUsersRegex.yml*

```
user:
- id: "regex:\\d+"
  name: regex:^expected user.* #expected user1
- id: "regex:\\d+"
  name: regex:.*user2$ #expected user2
```

```
@Test
@DataSet(cleanBefore = true) ①
@ExpectedDataSet("expectedUsersRegex.yml")
public void shouldAssertDatabaseUsingRegex() {
    User u = new User();
    u.setName("expected user1");
    User u2 = new User();
    u2.setName("expected user2");
    tx().begin();
    em().persist(u);
    em().persist(u2);
    tx().commit();
}
```

① You don't need to initialize a dataset but can use `cleanBefore` to clear database before testing.



When you use a dataset like users.yml in @DataSet dbunit will use `CLEAN_INSERT` seeding strategy for all declared tables in dataset. This is why we didn't needed `cleanBefore` in any other example tests.

# Chapter 6. Scriptable datasets

DBUnit Rules enables scripting in dataset for languages that implement JSR 233 - Scripting for the Java Platform, [see this article](#) for more information.

For this example we will introduce another JPA entity:

```
@Entity
public class Tweet {

    @Id
    @GeneratedValue
    private long id;

    @Size(min = 1, max = 140)
    private String content;

    private Integer likes;

    @Temporal(TemporalType.TIMESTAMP)
    private Calendar date;

    @ManyToOne(fetch = FetchType.LAZY)
    User user;
```

## 6.1. Javascript scriptable dataset

Following is a dataset which uses Javascript:

*src/test/resources/datasets/dataset-with-javascript.yml*

```
tweet:
- id: 1
  content: "dbunit rules!"
  likes: "js:(5+5)*10/2" ①
  user_id: 1
```

① **js:** prefix enables javascript in datasets.

and the junit test:

```

@Test
@DataSet(value = "dataset-with-javascript.yml",
        cleanBefore = true, ①
        disableConstraints = true) ②
public void shouldSeedDatabaseUsingJavaScriptInDataset() {
    Tweet tweet = (Tweet) emProvider.em().createQuery("select t from Tweet t where
t.id = 1").getSingleResult();
    assertThat(tweet).isNotNull();
    assertThat(tweet.getLikes()).isEqualTo(50);
}

@Test
@DataSet(value = "dataset-with-groovy.yml",
        cleanBefore = true,
        disableConstraints = true)
public void shouldSeedDatabaseUsingGroovyInDataset() throws ParseException {
    Tweet tweet = (Tweet) emProvider.em().createQuery("select t from Tweet t where
t.id = '1'").getSingleResult();
    assertThat(tweet).isNotNull();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");//remove time
    Date now = sdf.parse(sdf.format(new Date()));
    assertThat(tweet.getDate()).isEqualTo(now);
}
}

```

- ① As we don't declared `User` table in dataset it will not be cleared by `CLEAN_INSERT` seeding strategy so we need `cleanBefore` to avoid conflict with other tests that insert users.
- ② Disabling constraints is necessary because `Tweet` table depends on `User`.

if we do not disable constraints we will receive the error below on dataset creation:

```

Caused by: org.dbunit.DatabaseUnitException: Exception processing table name='TWEET'
    at
    org.dbunit.operation.AbstractBatchOperation.execute(AbstractBatchOperation.java:232)
    at org.dbunit.operation.CompositeOperation.execute(CompositeOperation.java:79)
    at
    com.github.dbunit.rules.dataset.DataSetExecutorImpl.createDataSet(DataSetExecutorImpl.
    java:127)
    ... 21 more
Caused by: java.sql.SQLIntegrityConstraintViolationException: integrity constraint
violation: foreign key no parent; FK_OH8MF7R69JSK6IISPTIAOCC6L table: TWEET
    at org.hsqldb.jdbc.JDBCUtil.sqlException(Unknown Source)

```



If we declare `User` table in `dataset-with-javascript.yml` dataset we can remove `cleanBefore` and `disableConstraints` attributes.

## 6.2. Groovy scriptable dataset

Javascript comes by default in JDK but you can use other script languages like **Groovy**, to do so you need to add it to test classpath:

*pom.xml*

```
<dependency>
  <groupId>org.codehaus.groovy</groupId>
  <artifactId>groovy-all</artifactId>
  <version>2.4.6</version>
  <scope>test</scope>
</dependency>
```

If Groovy is not present in classpath we'll receive a *warn message* (maybe we should fail, what do you think?):

```
WARNING: Could not find script engine with name groovy in classpath
```

Here's our Groovy based dataset:

*src/test/resources/datasets/dataset-with-groovy.yml*

```
tweet:
- id: "1"
  content: "dbunit rules!"
  date: "groovy:new Date()" ①
  user_id: 1
```

① **groovy**: prefix enables javascript in datasets.

And here is the test:

```
@Test
@DataSet(value = "dataset-with-groovy.yml",
         cleanBefore = true,
         disableConstraints = true)
public void shouldSeedDatabaseUsingGroovyInDataset() throws ParseException {
    Tweet tweet = (Tweet) emProvider.em().createQuery("select t from Tweet t where
t.id = '1'").getSingleResult();
    assertNotNull(tweet);
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");//remove time
    Date now = sdf.parse(sdf.format(new Date()));
    assertEquals(tweet.getDate(),now);
}
```

# Chapter 7. Multiple databases

Multiple databases can be tested by using multiple DBUnit rule and Entity manager providers:

```
package com.github.dbunit.rules.sample;

import com.github.dbunit.rules.DBUnitRule;
import com.github.dbunit.rules.api.dataset.DataSet;
import com.github.dbunit.rules.api.dataset.DataSetExecutor;
import com.github.dbunit.rules.api.dataset.DataSetModel;
import com.github.dbunit.rules.connection.ConnectionHolderImpl;
import com.github.dbunit.rules.dataset.DataSetExecutorImpl;
import com.github.dbunit.rules.util.EntityManagerProvider;
import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

import static org.assertj.core.api.Assertions.assertThat;

/**
 * Created by pestano on 23/07/15.
 */

@RunWith(JUnit4.class)
public class MultipleDataBasesTest {

    @Rule
    public EntityManagerProvider emProvider = EntityManagerProvider.instance("pu1");

    @Rule
    public EntityManagerProvider emProvider2 = EntityManagerProvider.instance("pu2");

    @Rule
    public DBUnitRule rule1 = DBUnitRule.instance("rule1", emProvider.connection()); ①

    @Rule
    public DBUnitRule rule2 = DBUnitRule.instance("rule2", emProvider2.connection());

    @Test
    @DataSet(value = "users.yml", executorId = "rule1") ②
    public void shouldSeedDatabaseUsingPu1() {
        User user = (User) emProvider.em().
            createQuery("select u from User u where u.id = 1").getSingleResult();
        assertThat(user).isNotNull();
        assertThat(user.getId()).isEqualTo(1);
    }

    @Test
```

```

@DataSet(value = "users.yml", executorId = "rule2")
public void shouldSeedDatabaseUsingPu2() {
    User user = (User) emProvider2.em().
        createQuery("select u from User u where u.id = 1").getSingleResult();
    assertThat(user).isNotNull();
    assertThat(user.getId()).isEqualTo(1);
}

@Test ③
public void shouldSeedDatabaseUsingMultiplePus() {
    DataSetExecutor exec1 = DataSetExecutorImpl.
        instance("exec1", new ConnectionHolderImpl(emProvider.connection()));
    DataSetExecutor exec2 = DataSetExecutorImpl.
        instance("exec2", new ConnectionHolderImpl(emProvider2.connection()));

    //programmatic seed db1
    exec1.createDataSet(new DataSetModel("users.yml"));

    exec2.createDataSet(new DataSetModel("dataset-with-javascript.yml")); //seed
    db2

    //user comes from database represented by pu1
    User user = (User) emProvider.em().
        createQuery("select u from User u where u.id = 1").getSingleResult();
    assertThat(user).isNotNull();
    assertThat(user.getId()).isEqualTo(1);

    //tweets comes from pu2
    Tweet tweet = (Tweet) emProvider.em().createQuery("select t from Tweet t where
t.id = 1").getSingleResult();
    assertThat(tweet).isNotNull();
    assertThat(tweet.getLikes()).isEqualTo(50);
}
}

```

- ① rule1 is the id of DataSetExecutor, the component responsible for database initialization in DBUnit Rules.
- ② here we match dataset executor id in @DataSet annotation so in this test we are going to use database from pu1.
- ③ For multiple databases in same test we need to initialize database state programmatically.



# Chapter 8. Ruling database in CDI tests

For CDI based tests we are going to use [DeltaSpike test control module](#) and [DBUnit rules CDI](#).

The first enables CDI in JUnit tests and the second enables DBUnit through a CDI interceptor.

## 8.1. Classpath dependencies

First we need DBUnit CDI: .pom.xml

```
<dependency>
  <groupId>com.github.dbunit-rules</groupId>
  <artifactId>cdi</artifactId>
  <version>${dbunit-rules.version}</version>
  <scope>test</scope>
</dependency>
```

And also DeltaSpike control module:

```

<dependency> ①
  <groupId>org.apache.deltaspike.core</groupId>
  <artifactId>deltaspike-core-impl</artifactId>
  <version>${ds.version}</version>
  <scope>test</scope>
</dependency>

<dependency> ②
  <groupId>org.apache.deltaspike.modules</groupId>
  <artifactId>deltaspike-test-control-module-api</artifactId>
  <version>${ds.version}</version>
  <scope>test</scope>
</dependency>

<dependency> ②
  <groupId>org.apache.deltaspike.modules</groupId>
  <artifactId>deltaspike-test-control-module-impl</artifactId>
  <version>${ds.version}</version>
  <scope>test</scope>
</dependency>

<dependency> ③
  <groupId>org.apache.deltaspike.cdictrl</groupId>
  <artifactId>deltaspike-cdictrl-owb</artifactId>
  <version>${ds.version}</version>
  <scope>test</scope>
</dependency>

<dependency> ④
  <groupId>org.apache.openwebbeans</groupId>
  <artifactId>openwebbeans-impl</artifactId>
  <version>1.6.2</version>
  <scope>test</scope>
</dependency>

```

- ① DeltaSpike core module is base of all DeltaSpike modules
- ② Test control module api and impl
- ③ CDI control OWB dependency, it is responsible for bootstrapping CDI container
- ④ OpenWebBeans as CDI implementation

## 8.2. Configuration

For configuration we will need a beans.xml which enables DBUnit CDI interceptor:

*/src/test/resources/META-INF/beans.xml*

```
include::{metainf}/beans.xml
```

And `apache-deltaspike.properties` to set our tests as CDI beans:

*/src/test/resources/META-INF/apache-deltaspike.properties*

```
include::{metainf}/apache-deltaspike.properties
```

The test itself must be a CDI bean so DBUnit Rules can intercept it.

The last configuration needed is to produce a EntityManager for tests:

```
package com.github.dbunit.rules.sample.cdi;

import com.github.dbunit.rules.util.EntityManagerProvider;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;
import javax.persistence.EntityManager;

/**
 * Created by pestano on 09/10/15.
 */
@ApplicationScoped
public class EntityManagerProducer {

    private EntityManager em;

    @Produces
    public EntityManager produce() {
        return EntityManagerProvider.instance("rulesDB").em();
    }

}
```

This entityManager will be used as a bridge to JDBC connection needed by DBUnit Rules.

## 8.3. Example

Here is a test example:

```

@RunWith(CdiTestRunner.class) ❶
public class DBUnitRulesCDITest {

    @Inject
    EntityManager em; ❷

    @Test
    @UsingDataSet("users.yml") ❸
    public void shouldListUsers() {
        List<User> users = em().
            createQuery("select u from User u").
            getResultList();
        assertThat(users).
            isNotNull().
            isEmpty().
            hasSize(2);
    }
}

```

- ❶ DeltaSpike JUnit runner that enables CDI in tests;
- ❷ The EntityManager we produced in previous steps;
- ❸ This annotation enables DBUnit CDI interceptor which will prepare database state before the test execution.

All other features presented earlier, **except multiple databases**, are supported by DBUnit CDI.

Here is **ExpectedDataSet** example:

*src/test/resources/datasets/expectedUsers.yml*

```

user:
  - id: 1
    name: "expected user1"
  - id: 2
    name: "expected user2"

```

And the test:

```
@Test
@UsingDataSet(cleanBefore = true) //needed to activate interceptor (can be at
class level)
@ExpectedDataSet(value = "expectedUsers.yml",ignoreCols = "id")
public void shouldMatchExpectedDataSet() {
    User u = new User();
    u.setName("expected user1");
    User u2 = new User();
    u2.setName("expected user2");
    em.getTransaction().begin();
    em.persist(u);
    em.persist(u2);
    em.getTransaction().commit();
}
```

## Chapter 9. Ruling database in BDD tests