Living Documentation

Version 1.2.2-SNAPSHOT

Table of Contents

1. **Introduction** . . . 1
2. **Seeding database** . . . 2
   - 2.1. Seed database with `DBUnit Rule` . . . 2
   - 2.2. Seed database with `DBUnit Interceptor` . . . 4
   - 2.3. Seed database with `JUnit 5 extension` . . . 7
   - 2.4. Seeding database in BDD tests with `Rider Cucumber` . . . 10
3. **DataSet creation** . . . 17
   - 3.1. Creating a `YAML` dataset . . . 17
   - 3.2. Creating a `JSON` dataset . . . 18
   - 3.3. Creating a `XML` dataset . . . 20
   - 3.4. Creating a `XLS` dataset . . . 21
   - 3.5. Creating a `CSV` dataset . . . 21
4. **Configuration** . . . 24
   - 4.1. DataSet configuration . . . 24
   - 4.2. DBUnit configuration . . . 25
5. **DataSet assertion** . . . 28
   - 5.1. Assertion with yml dataset . . . 28
   - 5.2. Assertion with regular expression in expected dataset . . . 29
   - 5.3. Database assertion with seeding before test execution . . . 30
   - 5.4. Failing assertion . . . 32
   - 5.5. Assertion using automatic transaction . . . 33
6. **Dynamic data using scritable datasets** . . . 35
   - 6.1. Seed database with groovy script in dataset . . . 35
   - 6.2. Seed database with javascript in dataset . . . 36
7. **Database connection leak detection** . . . 38
   - 7.1. Detecting connection leak . . . 38
8. **DataSet export** . . . 40
   - 8.1. Export dataset with `@ExportDataSet` annotation . . . 40
   - 8.2. Programmatic export . . . 40
   - 8.3. Configuration . . . 41
   - 8.4. Export using DBUnit Addon . . . 41

Chapter 1. Introduction

Database Rider aims for bringing DBUnit closer to your JUnit tests so **database testing will feel like a breeze!**. Here are the main features:

- JUnit rule to integrate with DBUnit via annotations:

```
    @Rule
    public DBUnitRule dbUnitRule = DBUnitRule.instance(jdbcConnection);①

    @Test
    @DataSet(value = "datasets/yml/users.yml")
    public void shouldSeedDataSet(){
        //database is seed with users.yml dataset
    }
```

① The rule depends on a JDBC connection.

- CDI integration via interceptor to seed database without rule instantiation;
- JSON, YAML, XML, XLS, and CSV support;
- Configuration via annotations or yml files;
- Cucumber integration;
- Multiple database support;
- Date/time support in datasets;
- Scriptable datasets with groovy and javascript;
- Regular expressions in expected datasets;
- JUnit 5 integration;
- DataSet export;
- Connection leak detection;
- Lot of examples.

The project is composed by 5 modules:

- Core: Contains the dataset executor and JUnit rule;
- CDI: provides the DBUnit interceptor;
- Cucumber: a CDI aware cucumber runner;
- JUnit5: Comes with an extension for JUnit5.
- Examples module.

Chapter 2. Seeding database

> In order to insert data into database before test execution
> As a developer
> I want to easily use DBUnit in JUnit tests.

Database Rider brings DBUnit to your JUnit tests by means of:

- JUnit rules (for JUnit4 tests);
- CDI interceptor (for `CDI` based tests)
- JUnit5 extension (for JUnit5 tests).

2.1. Seed database with `DBUnit Rule`

JUnit4 integrates with DBUnit through a JUnit rule called `DBUnitRule` which reads **@Dataset** annotations in order to prepare the database state using DBUnit behind the scenes.

The rule just needs a JDBC connection in order to be created.

Dependencies

To use it add the following maven dependency:

```
<dependency>
   <groupId>com.github.database-rider</groupId>
   <artifactId>rider-core</artifactId>
   <version>1.2.2-SNAPSHOT</version>
   <scope>test</scope>
</dependency>
```

Given

The following junit rules 👍

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

    @Rule
    public EntityManagerProvider emProvider =
EntityManagerProvider.instance("rules-it"); ①

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

① `EntityManagerProvider` is a simple Junit rule that creates a JPA entityManager for each test. DBUnit rule don't depend on EntityManagerProvider, it only needs a **JDBC connection**.

② DBUnit rule is responsible for reading `@DataSet` annotation and prepare the database for each test.

And

The following dataset 👍

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

```
USER:
  - ID: 1
    NAME: "@realpestano"
  - ID: 2
    NAME: "@dbunit"
TWEET:
  - ID: abcdef12345
    CONTENT: "dbunit rules!"
    DATE: "[DAY,NOW]"
    USER_ID: 1
FOLLOWER:
  - ID: 1
    USER_ID: 1
    FOLLOWER_ID: 2
```

When

The following test is executed: 👍

```
    @Test
    @DataSet(value = "datasets/yml/users.yml", useSequenceFiltering =
true)
    public void shouldSeedUserDataSet() {
        User user = (User) EntityManagerProvider.em().
                createQuery("select u from User u join fetch u.tweets
join fetch u.followers where u.id = 1").getSingleResult();
        assertThat(user).isNotNull();
        assertThat(user.getId()).isEqualTo(1);
        assertThat(user.getTweets()).isNotNull().hasSize(1);
        Tweet tweet = user.getTweets().get(0);
        assertThat(tweet).isNotNull();
        Calendar date = tweet.getDate();
        Calendar now = Calendar.getInstance();
        assertThat(date.get(Calendar.DAY_OF_MONTH)).
                isEqualTo(now.get(Calendar.DAY_OF_MONTH));
    }
```

Source code of the above example can be found here.

Then

The database should be seeded with the dataset content before test execution 👍

2.2. Seed database with `DBUnit Interceptor`

DBUnit CDI [1: Contexts and dependency for the Java EE] integration is done through a CDI interceptor which reads `@DataSet` to prepare database in CDI tests.

Dependencies

To use this module just add the following maven dependency:

```
<dependency>
   <groupId>com.github.database-rider</groupId>
   <artifactId>rider-cdi</artifactId>
   <version>1.2.2-SNAPSHOT</version>
   <scope>test</scope>
</dependency>
```

Given

DBUnit interceptor is enabled in your test beans.xml: 👍

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
     <interceptors>

<class>com.github.database.rider.cdi.DBUnitInterceptorImpl</class>
     </interceptors>
</beans>
```

Your test itself must be a CDI bean to be intercepted. if you're using Deltaspike test control just enable the following property in `test/resources/META-INF/apache-deltaspike.properties`:

```
deltaspike.testcontrol.use_test_class_as_cdi_bean=true
```

And

The following dataset 👍

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

```
USER:
  - ID: 1
    NAME: "@realpestano"
  - ID: 2
    NAME: "@dbunit"
TWEET:
  - ID: abcdef12345
    CONTENT: "dbunit rules!"
    USER_ID: 1
  - ID: abcdef12233
    CONTENT: "dbunit rules!"
    USER_ID: 2
  - ID: abcdef1343
    CONTENT: "CDI for the win!"
    USER_ID: 2
FOLLOWER:
  - ID: 1
    USER_ID: 1
    FOLLOWER_ID: 2
```

When

The following test is executed: 👍

```
@RunWith(CdiTestRunner.class) ①
@DBUnitInterceptor ②
@DataSet(value = "yml/users.yml")
public class DBUnitCDIIt {
    @Test
    @DataSet("yml/users.yml")
    public void shouldSeedUserDataSetUsingCdiInterceptor() {
        List<User> users = em.createQuery("select u from User u order
by u.id asc").getResultList();
        User user1 = new User(1);
        User user2 = new User(2);
        Tweet tweetUser1 = new Tweet();
        tweetUser1.setId("abcdef12345");
        assertThat(users).isNotNull().hasSize(2).contains(user1,
user2);
        List<Tweet> tweetsUser1 = users.get(0).getTweets();

assertThat(tweetsUser1).isNotNull().hasSize(1).contains(tweetUser1);
    }
```

① CdiTestRunner is provided by Apache Deltaspike but you should be able to use other CDI test runners.

② Needed to activate DBUnit interceptor

Source code of the above example can be found here.

Then

The database should be seeded with the dataset content before test execution 👍

2.3. Seed database with `JUnit 5 extension`

DBUnit is enabled in JUnit 5 tests through an extension named **DBUnitExtension**.

Dependencies

To use the extension just add the following maven dependency:

```
<dependency>
   <groupId>com.github.database-rider</groupId>
   <artifactId>rider-junit5</artifactId>
   <version>1.2.2-SNAPSHOT</version>
   <scope>test</scope>
</dependency>
```

Given

The following dataset 👍

src/test/resources/dataset/users.yml

```
USER:
  - ID: 1
    NAME: "@realpestano"
  - ID: 2
    NAME: "@dbunit"
```

When

The following junit5 test is executed 👍

```
@ExtendWith(DBUnitExtension.class) ①
@RunWith(JUnitPlatform.class) ②
@DataSet(cleanBefore = true)
public class DBUnitJUnit5It {

    private ConnectionHolder connectionHolder = () -> ③
            EntityManagerProvider.instance("junit5-
pu").clear().connection();④

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

① Enables DBUnit.

② JUnit 5 runner;

③ As JUnit5 requires **Java8** you can use lambdas in your tests;

④ DBUnitExtension will get connection by reflection so just declare a field or a method with `ConnectionHolder` as return type.

Source code of the above example can be found here.

Another way to activate DBUnit in JUnits 5 test is using **@DBRider** annotation (at method or class level):

```
    @DBRider //shortcut for
@ExtendWith(DBUnitExtension.class) and @Test
    @DataSet(value = "usersWithTweet.yml")
    public void shouldListUsers() {
        List users = EntityManagerProvider.em().
                createQuery("select u from User
u").getResultList();

assertThat(users).isNotNull().isNotEmpty().hasSize(2);
        assertThat(users.get(0)).isEqualTo(new User(1));
    }
```

① Shortcut for `@Test` and `@ExtendWith(DBUnitExtension.class)`

Then

The database should be seeded with the dataset content before test execution 👍

2.4. Seeding database in BDD tests with `Rider Cucumber`

DBUnit enters the BDD world through a dedicated JUNit runner which is based on Cucumber and Apache DeltaSpike.

This runner just starts CDI within your BDD tests so you just have to use Database Rider CDI interceptor on Cucumber steps, here is the so called Cucumber CDI runner declaration:

```
package com.github.database.rider.examples.cucumber;

import com.github.database.rider.cucumber.CdiCucumberTestRunner;
import cucumber.api.CucumberOptions;
import org.junit.runner.RunWith;


@RunWith(CdiCucumberTestRunner.class)
@CucumberOptions(
        features = {"src/test/resources/features/contacts.feature"},
        plugin = {"json:target/cucumber.json"}
        //glue = "com.github.database.rider.examples.glues"
)
public class ContactFeature {
}
```

As cucumber doesn't work with JUnit Rules, see this issue, you won't be able to use Cucumber runner with *DBUnit Rule*, but you can use DataSetExecutor in `@Before`, see example here.

Dependencies

Here is a set of maven dependencies needed by Database Rider Cucumber:

Most of the dependencies, except CDI container implementation, are brought by Database Rider Cucumber module transitively.

```
<dependency>
   <groupId>com.github.database-rider</groupId>
   <artifactId>rider-cucumber</artifactId>
   <version>1.2.2-SNAPSHOT</version>
   <scope>test</scope>
</dependency>
```

Cucumber dependencies

```
<dependency> ①
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>1.2.4</version>
    <scope>test</scope>
</dependency>
<dependency> ①
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>1.2.4</version>
    <scope>test</scope>
</dependency>
```

① You don't need to declare because it comes with Database Rider Cucumber module dependency.

DeltaSpike and CDI dependency

```
<dependency> ①
    <groupId>org.apache.deltaspike.modules</groupId>
    <artifactId>deltaspike-test-control-module-api</artifactId>
    <version>${ds.version}</version>
    <scope>test</scope>
</dependency>

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

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

<dependency> ②
    <groupId>org.apache.deltaspike.cdictrl</groupId>
    <artifactId>deltaspike-cdictrl-owb</artifactId>
    <version>${ds.version}</version>
    <scope>test</scope>
</dependency>

<dependency>  ②
    <groupId>org.apache.openwebbeans</groupId>
    <artifactId>openwebbeans-impl</artifactId>
    <version>1.6.2</version>
    <scope>test</scope>
</dependency>
```

① Also comes with Rider Cucumber.

② You can use CDI implementation of your choice.

Given

The following feature 👍

```
Feature: Contacts test
  As a user of contacts repository
  I want to crud contacts
  So that I can expose contacts service


  Scenario Outline: search contacts
    Given we have a list of contacts
    When we search contacts by name "<name>"
    Then we should find <result> contacts

    Examples: examples1
      | name     | result |
      | delta    | 1      |
      | sp       | 2      |
      | querydsl | 1      |
      | abcd     | 0      |


  Scenario: delete a contact

    Given we have a list of contacts
    When we delete contact by id 1
    Then we should not find contact 1
```

And

The following dataset 👍

```
CONTACT:
  - ID: 1
    NAME: "deltaspike"
    EMAIL: "users@deltaspike.apache.org"
    COMPANY_ID: 1
  - ID: 2
    NAME: "querydsl"
    EMAIL: "info@mysema.com"
    COMPANY_ID: 2
  - ID: 3
    NAME: "Spring"
    EMAIL: "spring@pivotal.io"
    COMPANY_ID: 3

COMPANY:
  - ID: 1
    NAME: "Apache"
  - ID: 2
    NAME: "Mysema"
  - ID: 3
    NAME: "Pivotal"
  - ID: 4
    NAME: "Google"
```

And

The following Cucumber test 👍

```
package com.github.database.rider.examples.cucumber;

import com.github.database.rider.cucumber.CdiCucumberTestRunner;
import cucumber.api.CucumberOptions;
import org.junit.runner.RunWith;


@RunWith(CdiCucumberTestRunner.class)
@CucumberOptions(
        features = {"src/test/resources/features/contacts.feature"},
        plugin = {"json:target/cucumber.json"}
        //glue = "com.github.database.rider.examples.glues"
)
public class ContactFeature {
}
```

When

The following cucumber steps are executed 👍

```
package com.github.database.rider.examples.cucumber;

import com.github.database.rider.core.api.dataset.DataSet;
import com.github.database.rider.cdi.api.DBUnitInterceptor;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;
import org.example.jpadomain.Contact;
import org.example.jpadomain.Contact_;
import org.example.service.deltaspike.ContactRepository;

import javax.inject.Inject;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNull;

@DBUnitInterceptor ①
public class ContactSteps {

    @Inject
    ContactRepository contactRepository; ②

    Long count;
```

```
    @Given("^we have a list of contacts$")
    @DataSet("datasets/contacts.yml") ③
    public void given() {
        assertEquals(contactRepository.count(), new Long(3));
    }

    @When("^we delete contact by id (\\d+)$")
    public void we_delete_contact_by_id(long id) throws Throwable {
        contactRepository.remove(contactRepository.findBy(id));
    }

    @Then("^we should not find contact (\\d+)$")
    public void we_should_not_find_contacts_in_database(long id) throws
Throwable {
        assertNull(contactRepository.findBy(id));
    }

    @When("^^we search contacts by name \"([^\"]*)\"$")
    public void we_search_contacts_by_name_(String name) throws
Throwable {
        Contact contact = new Contact();
        contact.setName(name);
        count = contactRepository.countLike(contact, Contact_.name);
    }

    @Then("^we should find (\\d+) contacts$")
    public void we_should_find_result_contacts(Long result) throws
Throwable {
        assertEquals(result, count);
    }
}
```

① Activates DBUnit CDI interceptor

② As the Cucumber cdi runner enables CDI, you can use injection into your Cucumber steps.

③ Dataset is prepared before step execution by `@DBUnitInterceptor`.

Source code for the example above can be found here.

Then

The database should be seeded with the dataset content before step execution 👍

Chapter 3. DataSet creation

In order to create datasets to feed tables
As a developer
I want to declare database state in external files.

It is a good practice to move database preparation or any infrastructure code outside test logic, it increases test maintainability.

3.1. Creating a `YAML` dataset

YAML stands for `yet another markup language` and is a very simple, lightweight yet powerful format.

YAML is based on spaces indentation so be careful because any missing or additional space can lead to an incorrect dataset.

Source code of the examples below can be found here.

Given

The following dataset 👍

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

```
USER:
  - ID: 1
    NAME: "@realpestano"
  - ID: 2
    NAME: "@dbunit"
TWEET:
  - ID: abcdef12345
    CONTENT: "dbunit rules!"
    DATE: "[DAY,NOW]"
    USER_ID: 1
FOLLOWER:
  - ID: 1
    USER_ID: 1
    FOLLOWER_ID: 2
```

When

The following test is executed: 👍

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

Then

The database should be seeded with the dataset content before test execution 👍

3.2. Creating a JSON dataset

Given

The following dataset 👍

src/test/resources/dataset/json/users.json

```
{
  "USER": [
    {
      "id": 1,
      "name": "@realpestano"
    },
    {
      "id": 2,
      "name": "@dbunit"
    }
  ],
  "TWEET": [
    {
      "id": "abcdef12345",
      "content": "dbunit rules json example",
      "date": "2013-01-20",
      "user_id": 1
    }
  ],
  "FOLLOWER": [
    {
      "id": 1,
      "user_id": 1,
      "follower_id": 2
    }
  ]
}
```

When

The following test is executed: 👍

```
@Test
@DataSet("json/users.json")
public void shouldSeedDatabaseWithJSONDataSet() {
   List<User> users = em().createQuery("select u from User
u").getResultList();
   assertThat(users).isNotNull().isNotEmpty().hasSize(2);
}
```

Then

The database should be seeded with the dataset content before test execution 👍

3.3. Creating a XML dataset

Given

The following dataset 👍

src/test/resources/dataset/xml/users.xml

```
<dataset>
    <USER id="1" name="@realpestano" />
    <USER id="2" name="@dbunit" />
    <TWEET id="abcdef12345" content="dbunit rules flat xml example"
user_id="1"/>
    <FOLLOWER id="1" user_id="1" follower_id="2"/>
</dataset>
```

When

The following test is executed: 👍

```
@Test
@DataSet("xml/users.xml")
public void shouldSeedDatabaseWithXMLDataSet() {
   List<User> users = em().createQuery("select u from User
u").getResultList();
   assertThat(users).isNotNull().isNotEmpty().hasSize(2);
}
```

Then

The database should be seeded with the dataset content before test execution 👍

3.4. Creating a XLS dataset

Given

The following dataset 👍

src/test/resources/dataset/xls/users.xls

```
ID  NAME
1   @realpestano
2   @dbunit
```

Each Excell `sheet` name is the **table name**, first row is **columns names** and remaining rows/cells are values.

When

The following test is executed: 👍

```
@Test
@DataSet("xls/users.xls")
public void shouldSeedDatabaseWithXLSDataSet() {
    List<User> users = em().createQuery("select u from User
u").getResultList();
    assertThat(users).isNotNull().isNotEmpty().hasSize(2);
}
```

Then

The database should be seeded with the dataset content before test execution 👍

3.5. Creating a CSV dataset

Given

The following dataset 👍

src/test/resources/dataset/csv/USER.csv

```
ID, NAME
"1","@realpestano"
"2","@dbunit"
```

src/test/resources/dataset/csv/TWEET.csv

```
ID, CONTENT, DATE, LIKES, USER_ID
"abcdef12345","dbunit rules!","2016-09-12 22:46:20.0",null,"1"
```

File name is **table name** and first row is **column names**.

src/test/resources/dataset/csv/table-ordering.txt

```
FOLLOWER
TWEET
USER
```

CSV datasets are composed by multiple files (one per table) and a table ordering descriptor declaring the order of creation.

Also note that each csv dataset must be declared in its own folder because DBUnit will read all csv files present in dataset folder.

When

The following test is executed: 👍

```
@Test
@DataSet("datasets/csv/USER.csv") ①
public void shouldSeedDatabaseWithCSVDataSet() {
    List<User> users = em().createQuery("select u from User
u").getResultList();
    assertThat(users).isNotNull().isNotEmpty().hasSize(2);
}
```

① You need to declare just one csv dataset file. Database rider will take parent folder as dataset folder.

Then

> The database should be seeded with the dataset content before test execution 👍

Chapter 4. Configuration

> In order to handle various use cases
> As a developer
> I want to be able to configure DataBase Rider

4.1. DataSet configuration

DataSet configuration is done via **@DataSet** annotation at **class** or **method** level:

```
@Test
@DataSet(value ="users.yml", strategy = SeedStrategy.UPDATE,
  disableConstraints = true,cleanAfter = true,
  useSequenceFiltering = true, tableOrdering = {"TWEET","USER"},
  executeScriptsBefore = "script.sql", executeStatementsBefore = "DELETE from USER
where 1=1"
  transactional = true, cleanAfter=true)
public void shouldCreateDataSet(){

}
```

Table below illustrate the possible configurations:

Name	Description	Default
value	Dataset file name using test resources folder as root directory. Multiple, comma separated, dataset file names can be provided.	""
executorId	Name of dataset executor for the given dataset.	DataSetExecutorImpl.DEFAULT_EXECUTOR_ID
strategy	DataSet seed strategy. Possible values are: CLEAN_INSERT, INSERT, REFRESH and UPDATE.	CLEAN_INSERT, meaning that DBUnit will clean and then insert data in tables present in provided dataset.
useSequenceFiltering	If true dbunit will look at constraints and dataset to try to determine the correct ordering for the SQL statements.	true

Name	Description	Default
tableOrdering	A list of table names used to reorder DELETE operations to prevent failures due to circular dependencies.	""
disableConstraints	Disable database constraints.	false
cleanBefore	If true Database Rider will try to delete database before test in a smart way by using table ordering and brute force.	false
cleanAfter	If true Database Rider will try to delete database after test in a smart way by using table ordering and brute force.	false
transactional	If true a transaction will be started before and committed after test execution.	false
executeStatementsBefore	A list of jdbc statements to execute before test.	{}
executeStatementsAfter	A list of jdbc statements to execute after test.	{}
executeScriptsBefore	A list of sql script files to execute before test. Note that commands inside sql file must be separated by `;`.	{}
executeScriptsAfter	A list of sql script files to execute after test. Note that commands inside sql file must be separated by `;`.	{}

4.2. DBUnit configuration

`DBUnit`, the tool doing the dirty work the scenes, can be configured by **@DBUnit** annotation (class or method level) and **dbunit.yml** file present in `test resources` folder.

```
@Test
@DBUnit(cacheConnection = true, cacheTableNames = false, allowEmptyFields =
true,batchSize = 50)
public void shouldLoadDBUnitConfigViaAnnotation() {

}
```

Here is a dbunit.yml example, also the default values:

src/test/resources/dbunit.yml

```
cacheConnection: true ①
cacheTableNames: true ②
leakHunter: false ③
caseInsensitiveStrategy:
!!com.github.database.rider.core.api.configuration.Orthography 'UPPERCASE' ④
properties:
  batchedStatements:  false ⑤
  qualifiedTableNames: false ⑥
  caseSensitiveTableNames: false ⑦
  batchSize: 100 ⑧
  fetchSize: 100 ⑨
  allowEmptyFields: false ⑩
  escapePattern: ⑪
connectionConfig: ⑫
  driver: ""
  url: ""
  user: ""
  password: ""
```

① Database connection will be reused among tests

② Caches table names to avoid query connection metadata unnecessarily

③ Activate connection leak detection. In case a leak (open JDBC connections is increased after test execution) is found an exception is thrown and test fails.

④ When `caseSensitiveTableNames` is `false` will apply letter case based on configured strategy. Valid values are `UPPERCASE` and `LOWERCASE`.

⑤ Enables usage of JDBC batched statement

⑥ Enable or disable multiple schemas support. If enabled, Dbunit access tables with names fully qualified by schema using this format: SCHEMA.TABLE.

⑦ Enable or disable case sensitive table names. If enabled, Dbunit handles all table names in a case sensitive way.

⑧ Specifies the size of JDBC batch updates

⑨ Specifies statement fetch size for loading data into a result set table.

⑩ Allow to call INSERT/UPDATE with empty strings ('').

⑪ Allows schema, table and column names escaping. The property value is an escape pattern where the ? is replaced by the name. For example, the pattern "[?]" is expanded as "[MY_TABLE]" for a table named "MY_TABLE". The most common escape pattern is ""?"" which surrounds the table name with quotes (for the above example it would result in ""MY_TABLE""). As a fallback if no questionmark is in the given String and its length is one it is used to surround the table name on the left and right side. For example the escape pattern """ will have the same effect as the escape pattern ""?"".

⑫ JDBC connection configuration, it will be used in case you don't provide a connection inside test (except in CDI test where connection is inferred from entity manager).

`@DBUnit` annotation takes precedence over `dbunit.yml` global configuration which will be used only if the annotation is not present.

Since version 1.1.0 you can define only the properties of your interest, example:

```
cacheConnection: false
properties:
  caseSensitiveTableNames: true
  escapePattern: ""?""
```

Chapter 5. DataSet assertion

In order to verify database state after test execution
As a developer
I want to assert database state with datasets.

Complete source code of examples below can be found here.

5.1. Assertion with yml dataset

Given

The following dataset 👍

expectedUsers.yml

```
USER:
  - ID: 1
    NAME: "expected user1"
  - ID: 2
    NAME: "expected user2"
```

When

The following test is executed: 👍

```
@RunWith(JUnit4.class)
@DBUnit(cacheConnection = true)
public class ExpectedDataSetIt {

    @Rule
    public EntityManagerProvider emProvider =
EntityManagerProvider.instance("rules-it");

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


    @Test
    @DataSet(cleanBefore = true)①
    @ExpectedDataSet(value = "yml/expectedUsers.yml",ignoreCols = "id")
    public void shouldMatchExpectedDataSet() {
        EntityManagerProvider instance =
EntityManagerProvider.newInstance("rules-it");
        User u = new User();
        u.setName("expected user1");
        User u2 = new User();
        u2.setName("expected user2");
        instance.tx().begin();
        instance.em().persist(u);
        instance.em().persist(u2);
        instance.tx().commit();
    }
```

① Clear database before to avoid conflict with other tests.

Then

Test must pass because database state is as in expected dataset. 👍

5.2. Assertion with regular expression in expected dataset

Given

The following dataset 👍

expectedUsersRegex.yml

```
USER:
  - ID: "regex:\\d+"
    NAME: regex:^expected user.* #expected user1
  - ID: "regex:\\d+"
    NAME: regex:.*user2$ #expected user2
```

When

The following test is executed: 👍

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

Then

Test must pass because database state is as in expected dataset. 👍

5.3. Database assertion with seeding before test execution

Given

The following dataset 👍

user.yml

```
USER:
  - ID: 1
    NAME: "@realpestano"
  - ID: 2
    NAME: "@dbunit"
```

And

The following dataset 👍

expectedUser.yml

```
USER:
  - ID: 2
    NAME: "@dbunit"
```

When

The following test is executed: 👍

```
@Test
@DataSet(value = "yml/user.yml", disableConstraints = true)
@ExpectedDataSet(value = "yml/expectedUser.yml", ignoreCols = "id")
public void shouldMatchExpectedDataSetAfterSeedingDataBase() {
    tx().begin();
    em().remove(EntityManagerProvider.em().find(User.class,1L));
    tx().commit();
}
```

Then

Test must pass because database state is as in expected dataset. 👍

5.4. Failing assertion

Given

The following dataset 👍

expectedUsers.yml

```
USER:
  - ID: 1
    NAME: "expected user1"
  - ID: 2
    NAME: "expected user2"
```

When

The following test is executed: 👍

```
@Test
@ExpectedDataSet(value = "yml/expectedUsers.yml",ignoreCols = "id")
public void shouldNotMatchExpectedDataSet() {
    User u = new User();
    u.setName("non expected user1");
    User u2 = new User();
    u2.setName("non expected user2");
    EntityManagerProvider.tx().begin();
    EntityManagerProvider.em().persist(u);
    EntityManagerProvider.em().persist(u2);
    EntityManagerProvider.tx().commit();
}
```

Then

Test must fail with following error: 👍

junit.framework.ComparisonFailure: value (table=USER, row=0, col=name) expected:<[]expected user1> but was:<[non]expected user1> at org.dbunit.assertion.JUnitFailureFactory.createFailure(JUnitFailureFactory.java:39) at org.dbunit.assertion.DefaultFailureHandler.createFailure(DefaultFailureHandler.java:97) at org.dbunit.assertion.DefaultFailureHandler.handle(DefaultFailureHandler.java:223) at ...

5.5. Assertion using automatic transaction

Given

The following dataset 👍

expectedUsersRegex.yml

```
USER:
  - ID: "regex:\\d+"
    NAME: regex:^expected user.* #expected user1
  - ID: "regex:\\d+"
    NAME: regex:.*user2$ #expected user2
```

When

The following test is executed: 👍

```
@Test
@DataSet(cleanBefore = true, transactional = true,executorId =
"TransactionIt")
@ExpectedDataSet(value = "yml/expectedUsersRegex.yml")
@DBUnit(cacheConnection = true)
public void shouldManageTransactionAutomatically() {
    User u = new User();
    u.setName("expected user1");
    User u2 = new User();
    u2.setName("expected user2");
    EntityManagerProvider.em().persist(u);
    EntityManagerProvider.em().persist(u2);
}
```

`Transactional` attribute will make Database Rider start a transaction before test and commit the transaction **after** test execution but **before** expected dataset comparison.

Then

Test must pass because inserted users are commited to database and database state matches expected dataset. 👍

Chapter 6. Dynamic data using scritable datasets

In order to have dynamic data in datasets
As a developer
I want to use scripts in DBUnit datasets.

Scritable datasets are backed by JSR 223. [3: Scripting for the Java Platform, for more information access the official docs here].

Complete source code of examples below can be found here.

6.1. Seed database with groovy script in dataset

Given

Groovy script engine is on test classpath 👍

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

And

The following dataset 👍

```
TWEET:
  - ID: "1"
    CONTENT: "dbunit rules!"
    DATE: "groovy:new Date()" ①
    USER_ID: 1
```

① Groovy scripting is enabled by `groovy:` string.

When

The following test is executed: 👍

```
@Test
@DataSet(value = "datasets/yml/groovy-with-date-
replacements.yml",cleanBefore = true, disableConstraints = true,
executorId = "rules-it")
public void shouldReplaceDateUsingGroovyInDataset() {
    Tweet tweet = (Tweet) emProvider.em().createQuery("select t from
Tweet t where t.id = '1'").getSingleResult();
    assertThat(tweet).isNotNull();
    assertThat(tweet.getDate().get(Calendar.DAY_OF_MONTH)).
            isEqualTo(now.get(Calendar.DAY_OF_MONTH));
    assertThat(tweet.getDate(). get(Calendar.HOUR_OF_DAY)).
            isEqualTo(now.get(Calendar.HOUR_OF_DAY));
}
```

Then

Dataset script should be interpreted while seeding the database 👍

6.2. Seed database with javascript in dataset

Javascript engine comes within JDK so no additional classpath dependency is necessary.

Given

The following dataset 👍

```
TWEET:
  - ID: "1"
    CONTENT: "dbunit rules!"
    LIKES: "js:(5+5)*10/2" ①
    USER_ID: 1
```

① Javascript scripting is enabled by `js:` string.

When

The following test is executed: 👍

```
@Test
@DataSet(value = "datasets/yml/js-with-calc-
replacements.yml",cleanBefore = true, disableConstraints = true,
executorId = "rules-it")
public void shouldReplaceLikesUsingJavaScriptInDataset() {
    Tweet tweet = (Tweet) emProvider.em().createQuery("select t from
Tweet t where t.id = '1'").getSingleResult();
    assertThat(tweet).isNotNull();
    assertThat(tweet.getLikes()).isEqualTo(50);
}
```

Then

Dataset script should be interpreted while seeding the database 👍

Chapter 7. Database connection leak detection

> In order to find JDBC connection leaks
> As a developer
> I want to make Database Rider monitor connections during tests execution.

Leak hunter is a Database Rider component, based on this blog post, which counts open jdbc connections before and after test execution.

Complete source code of example below can be found here.

7.1. Detecting connection leak

```
@RunWith(JUnit4.class)
@DBUnit(leakHunter = true) ①
public class LeakHunterIt {

    @Rule
    public DBUnitRule dbUnitRule = DBUnitRule.instance(new
ConnectionHolderImpl(getConnection()));

    @Rule
    public ExpectedException exception = ExpectedException.none();

    @Test
    @DataSet("yml/user.yml")
    public void shouldFindConnectionLeak() throws SQLException {
        exception.expect(LeakHunterException.class);
        exception.expectMessage("Execution of method shouldFindConnectionLeak left 1
open connection(s).");
        createLeak();
    }

    private void createLeak() throws SQLException {
        Connection connection = getConnection();
        try (Statement stmt = connection.createStatement()) {
            ResultSet resultSet = stmt.executeQuery("select count(*) from user");
            assertThat(resultSet.next()).isTrue();
            assertThat(resultSet.getInt(1)).isEqualTo(2);
        }
    }
```

① Enables connection leak detection.

If number of connections after test execution are greater than before then a **LeakHunterException** will be raised.

Chapter 8. DataSet export

In order to easily create `dataset files`
As a developer
I want generate datasets based on database state.

Manual creation of datasets is a very error prone task. In order to export database state after test execution into datasets files one can use @ExportDataSet Annotation or use DataSetExporter component or even using a JBoss Forge addon.

Complete source code of examples below can be found here.

8.1. Export dataset with `@ExportDataSet` annotation

```
@Test
@DataSet("datasets/yml/users.yml") ①
@ExportDataSet(format = DataSetFormat.XML, outputName =
"target/exported/xml/allTables.xml")
public void shouldExportAllTablesInXMLFormat() {
}
```

① Used here just to seed database, you could insert data manually or connect to a database which already has data.

After above test execution all tables will be exported to a xml dataset.

XML, YML, JSON, XLS and CSV formats are supported.

8.2. Programmatic export

```
@Test
@DataSet(cleanBefore = true)
public void shouldExportYMLDataSetProgrammatically() throws SQLException,
DatabaseUnitException {
    tx().begin();
    User u1 = new User();
    u1.setName("u1");
    EntityManagerProvider.em().persist(u1);
    tx().commit();
    DataSetExporter.getInstance().export(emProvider.connection(), new
DataSetExportConfig().outputFileName("target/user.yml"));
    File ymlDataSet = new File("target/user.yml");
    assertThat(ymlDataSet).exists();
    assertThat(contentOf(ymlDataSet)).
            contains("USER:" + NEW_LINE +
                            "  - ID: 1" + NEW_LINE +
                            "    NAME: \"u1\"" + NEW_LINE
            );
}
```

8.3. Configuration

Following table shows all exporter configuration options:

Name	Description	Default
format	Exported dataset file format.	YML
includeTables	A list of table names to include in exported dataset.	Default is empty which means **ALL tables**.
queryList	A list of select statements which the result will used in exported dataset.	{}
dependentTables	If true will bring dependent tables of declared includeTables.	false
outputName	Name (and path) of output file.	""

8.4. Export using DBUnit Addon

DBUnit Addon exports DBUnit datasets based on a database connection.

Pre requisites

You need JBoss Forge installed in your IDE or available at command line.

Installation

Use install addon from git command:

```
addon-install-from-git --url https://github.com/database-rider/dbunit-addon.git
```

Usage

1. Setup database connection

   [Setup command] | *https://raw.githubusercontent.com/database-rider/dbunit-*

addon/master/setup_cmd.png

2. Export database tables into **YAML**, **JSON**, **XML**, **XLS** and **CSV** datasets.

   [Export command] | *https://raw.githubusercontent.com/database-rider/dbunit-*

addon/master/export_cmd.png

Export configuration

- `Format`: Dataset format.
- `Include tables`: Name of tables to include in generated dataset. If empty all tables will be exported.
- `Dependent tables`: If true will bring dependent included tables. Works in conjunction with `includeTables`.
- `Query list`: A list of SQL statements which resulting rows will be used in generated dataset.
- `Output dir`: directory to generate dataset.
- `Name`: name of resulting dataset. Format can be ommited in dataset name.