Database Rider Getting Started

Table of Contents

This guide is a `20~30 minutes read` which will introduce Database Rider and make a tour among its main features.

Chapter 1. Introduction

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

Most inspiration of Database Rider 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/database-rider/getting-started

Chapter 2. Setup Database Rider

First thing to do is to add Database Rider core module to your test classpath:

```
        <dependency>
            <groupId>com.github.database-rider</groupId>
            <artifactId>rider-core</artifactId>
            <version>1.0.0-RC2</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>2.3.4</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>4.2.20.Final</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>4.2.20.Final</version>
            <scope>test</scope>
        </dependency>
```

Database Rider don't depend on JPA, it only needs a JDBC connection to ride database through DBUnit.

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="riderDB" transaction-type="RESOURCE_LOCAL">

        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>com.github.database.rider.gettingstarted.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" /> ①
        <property name="hibernate.show_sql" value="true" />
        </properties>

    </persistence-unit>

</persistence>
```

① We're creating the database from our JPA entities, but we could use a database migration tool like flyway to do this work, see example here.

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 ride our database tests!

Chapter 3. Example

Create a yaml file which will be used to prepare database (with two users) before the test:

src/test/resources/dataset/users.yml

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

① Table name is followed by `:`, we can have multiple tables in the same file.

② Table rows are separated by `-`.

Be careful with **spaces**, wrong indentation can lead to invalid dataset (principally in `yaml` datasets).

For more dataset examples, look here.

You can generate datasets based on database tables, look at Exporting DataSets section.

And the JUnit test:

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

    @Rule
    public EntityManagerProvider emProvider = EntityManagerProvider.instance("riderDB
");  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().
                isNotEmpty().
                hasSize(2);
    }
}
```

① EntityManagerProvider is a JUnit rule that initializes a JPA entity manager before each **test class**. `riderDB` 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. Note that you can provide a comma separated list of datasets names here.

④ **em()** is a shortcut (`import static com.github.database.rider.util.EntityManagerProvider.em;`) for the EntityManager that was initialized by EntityManagerProvider rule.

There is a lot of example tests here.

Chapter 4. Transactions

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

```
    @Test
    @DataSet(value="users.yml", disableConstraints=true)
    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: ①
  - id: 1 ②
    name: "@realpestano"
  - 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.database.rider.assertion.DataSetAssert.compareData(DataSetA
ssert.java:94)
```

5.1. Regular expressions

Expected datasets also alow `regexp` 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 (by default) for all declared tables in dataset. This is why we didn't needed `cleanBefore` in any other example tests.

Chapter 6. Scriptable datasets

Database Rider 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.DATE)
    private Date 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) em().createQuery("select t from Tweet t where t.id = 1"
).getSingleResult();
        assertThat(tweet).isNotNull();
        assertThat(tweet.getLikes()).isEqualTo(50);
    }
```

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.database.rider.dataset.DataSetExecutorImpl.createDataSet(DataSetExecutorImp
l.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) 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);
    }
```

Chapter 7. Configuration

There are two types of configuration in Database Rider: `DataSet` and `DBUnit`.

DataSet Configuration

this basically setup the `dataset` which will be used. The only way to configure a dataset is using **@DataSet** annotation.

It can be used at **class** or **method** level:

```
@Test
@DataSet(value ="users.yml", strategy = SeedStrategy.UPDATE,
       disableConstraints = true,cleanAfter = true,transactional = true)
public void shouldLoadDataSetConfigFromAnnotation(){

 }
```

Here are possible values:

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 on provided dataset.
useSequenceFiltering	If true dbunit will look at constraints and dataset to try to determine the correct ordering for the SQL statements.	true
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

Name	Description	Default
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 test 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 `;`.	{}

DBUnit Configuration

this basically setup `DBUnit` itself. It 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
cacheTableNames: true
leakHunter: false
properties:
  batchedStatements:  false
  qualifiedTableNames: false
  caseSensitiveTableNames: false
  batchSize: 100
  fetchSize: 100
  allowEmptyFields: false
  escapePattern:
connectionConfig:
  driver: ""
  url: ""
  user: ""
  password: ""
```

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

Chapter 8. Multiple databases

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

```
package com.github.database.rider.gettingstarted;

import com.github.database.rider.core.DBUnitRule;
import com.github.database.rider.core.api.dataset.DataSet;
import com.github.database.rider.core.api.dataset.DataSetExecutor;
import com.github.database.rider.core.configuration.DataSetConfig;
import com.github.database.rider.core.connection.ConnectionHolderImpl;
import com.github.database.rider.core.dataset.DataSetExecutorImpl;
import com.github.database.rider.core.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 DataSetConfig("users.yml"));

        exec2.createDataSet(new DataSetConfig("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 Database Rider.

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` instead of using annotations.

Chapter 9. Riding database in JUnit 5 tests

JUnit 5 is the new version of JUnit and comes with a new extension model, so instead of `rules` you will use `extensions` in your tests. Database Rider comes with a JUnit 5 extension which enables DBUnit.

9.1. Configuration

Just add following dependency to your classpath:

pom.xml

```
<dependency>
   <groupId>com.github.database-rider</groupId>
   <artifactId>rider-junit5</artifactId>
   <version>1.0.0-RC2</version>
   <scope>test</scope>
</dependency>
```

9.2. Example

```
@ExtendWith(DBUnitExtension.class) ①
@RunWith(JUnitPlatform.class) ②
public class DBUnitJUnit5Test {

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

    @Test
    @DataSet("users.yml")
    public void shouldListUsers() {
        List<User> users = 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.

Chapter 10. Riding database in CDI tests

For CDI based tests we are going to use DeltaSpike test control module and Database Rider CDI.

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

10.1. Classpath dependencies

First we need DBUnit CDI: .pom.xml

```
        <dependency>
            <groupId>com.github.database-rider</groupId>
            <artifactId>rider-cdi</artifactId>
            <version>1.0.0-RC2</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>
            <groupId>org.apache.deltaspike.modules</groupId>
            <artifactId>deltaspike-data-module-impl</artifactId>
            <version>${ds.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.deltaspike.modules</groupId>
            <artifactId>deltaspike-data-module-api</artifactId>
            <version>${ds.version}</version>
        </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 bootstraping CDI container

④ OpenWebBeans as CDI implementation

10.2. Configuration

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

/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>
```

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

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

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

The test itself must be a CDI bean so Database Rider can intercept it.

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

```
package com.github.database.rider.gettingstarted.cdi;



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

import static com.github.database.rider.core.util.EntityManagerProvider.instance;

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

    private EntityManager em;


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

}
```

This entityManager will be used as a bridge to JDBC connection needed by Database Rider.

10.3. Example

Here is a test example:

```
Unresolved directive in index.adoc - include::/home/travis/build/database-rider
/getting-started/src/test/java/com/github/database/rider/gettingstarted/cdi
/DBUnitRulesCDITest.java[tags=sample]
}
```

① DeltaSpike JUnit runner that enables CDI in tests;

② Activates DBUnitInterceptor which will read `@DataSet` annotation in order to seed database before test;

③ The EntityManager we produced in previous steps;

④ 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.

For more examples, look at CDI module tests here.

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
    @DataSet(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();
    }
```

2

Chapter 11. Riding database in BDD tests

BDD and DBUnit are integrated by Database Rider Cucumber. It's a Cucumber runner which is CDI aware.

11.1. Configuration

Just add following dependency to your classpath:

pom.xml

```
        <dependency>
            <groupId>com.github.database-rider</groupId>
            <artifactId>rider-cucumber</artifactId>
            <version>1.0.0-RC2</version>
            <scope>test</scope>
        </dependency>
```

Now you just need to use **CdiCucumberTestRunner** to have `Cucumber`, `CDI` and `DBUnit` on your BDD tests.

11.2. Example

First we need a feature file:

src/test/resources/features/search-users.feature

```
Feature: Search users
In order to find users quickly
As a recruiter
I want to be able to query users by its tweets.

Scenario Outline: Search users by tweet content

Given We have two users that have tweets in our database

When I search them by tweet content <value>

Then I should find <number> users
Examples:
| value    | number |
| "dbunit" | 1      |
| "rules"  | 2      |
```

Then a dataset to prepare our database:

src/test/resources/datasets/usersWithTweet.json

```
{
  "USER": [
    {
      "id": 1,
      "name": "@realpestano"
    },
    {
      "id": 2,
      "name": "@dbunit"
    }
  ],
  "TWEET": [
    {
      "id": 1,
      "content": "dbunit rules json example",
      "date": "2013-01-20",
      "user_id": 1
    },
    {
      "id": 2,
      "content": "CDI rules",
      "date": "2016-06-20",
      "user_id": 2
    }
  ]
}
```

Now a Cucumber runner test entry point:

```
package com.github.database.rider.gettingstarted.bdd;

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

/**
 * Created by rmpestano on 4/17/16.
 */
@RunWith(CdiCucumberTestRunner.class)
@CucumberOptions(features ="src/test/resources/features/search-users.feature")
public class DatabaseRiderBddTest {
}
```

And finally our cucumber step definitions:

```
package com.github.database.rider.gettingstarted.bdd;
```

```
import com.github.database.rider.cdi.api.DBUnitInterceptor;
import com.github.database.rider.core.api.dataset.DataSet;
import com.github.database.rider.gettingstarted.User;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;
import org.hibernate.Session;
import org.hibernate.criterion.MatchMode;
import org.hibernate.criterion.Restrictions;
import org.hibernate.sql.JoinType;

import javax.inject.Inject;
import javax.persistence.EntityManager;
import java.util.List;

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

/**
 * Created by pestano on 20/06/16.
 */
@DBUnitInterceptor
public class SearchUsersSteps {

    @Inject
    EntityManager entityManager;

    List<User> usersFound;

    @Given("^We have two users that have tweets in our database$")
    @DataSet("usersWithTweet.json")
    public void We_have_two_users_in_our_database() throws Throwable {
    }

    @When("^I search them by tweet content \"([^\"]*)\"$")
    public void I_search_them_by_tweet_content_value(String tweetContent) throws
Throwable {
        Session session = entityManager.unwrap(Session.class);
        usersFound = session.createCriteria(User.class).
        createAlias("tweets","tweets", JoinType.LEFT_OUTER_JOIN).
        add(Restrictions.ilike("tweets.content",tweetContent, MatchMode.ANYWHERE))
.list();
    }

    @Then("^I should find (\\d+) users$")
    public void I_should_find_number_users(int numberOfUsersFound) throws Throwable {
        assertThat(usersFound).
                isNotNull().
                hasSize(numberOfUsersFound).
                contains(new User(1L));//examples contains user with id=1
    }
```

```
}
```

Living documentation of *Database Rider* is based on its BDD tests, you can access it here: http://rmpestano.github.io/database-rider/latest/documentation.html.

Chapter 12. Exporting DataSets

Creating dataset files is a very error prone task when done manually. Using DataSetExporter component you can generate **datasets from database** in YML, JSON, XML, CSV and XLS formats.

There are two ways to generate datasets with database rider, programmatically or from your IDE using a JBoss Forge addon:

12.1. Generating datasets programmatically

```
    @Test
    @DataSet(cleanBefore=true)
    public void shouldExportYMLDataSetUsingDataSetExporter() throws SQLException,
DatabaseUnitException{
        tx().begin();
        User u1 = new User();
        u1.setName("u1");
        em().persist(u1); //just insert a user and assert it is present in exported
dataset
        tx().commit();
        DataSetExporterImpl.getInstance().
        export(emProvider.connection(), ①
            new DataSetExportConfig().outputFileName("target/user.yml")); ②
        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
                          );

    }
```

① JDBC connection;

② the second required parameter is a ExporterConfig which only requires output file name attribute;

You can use @ExportDataSet to make extraction even easier:

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

    }
```

① Not required, its here only to add some data to be exported after test execution.

Full example above (and other related tests) can be found here.

12.2. Export Datasets using DBUnit Addon

DBUnit Addon is a JBoss forge addon which lets you export datasets from within your IDE through a nice GUI interface.

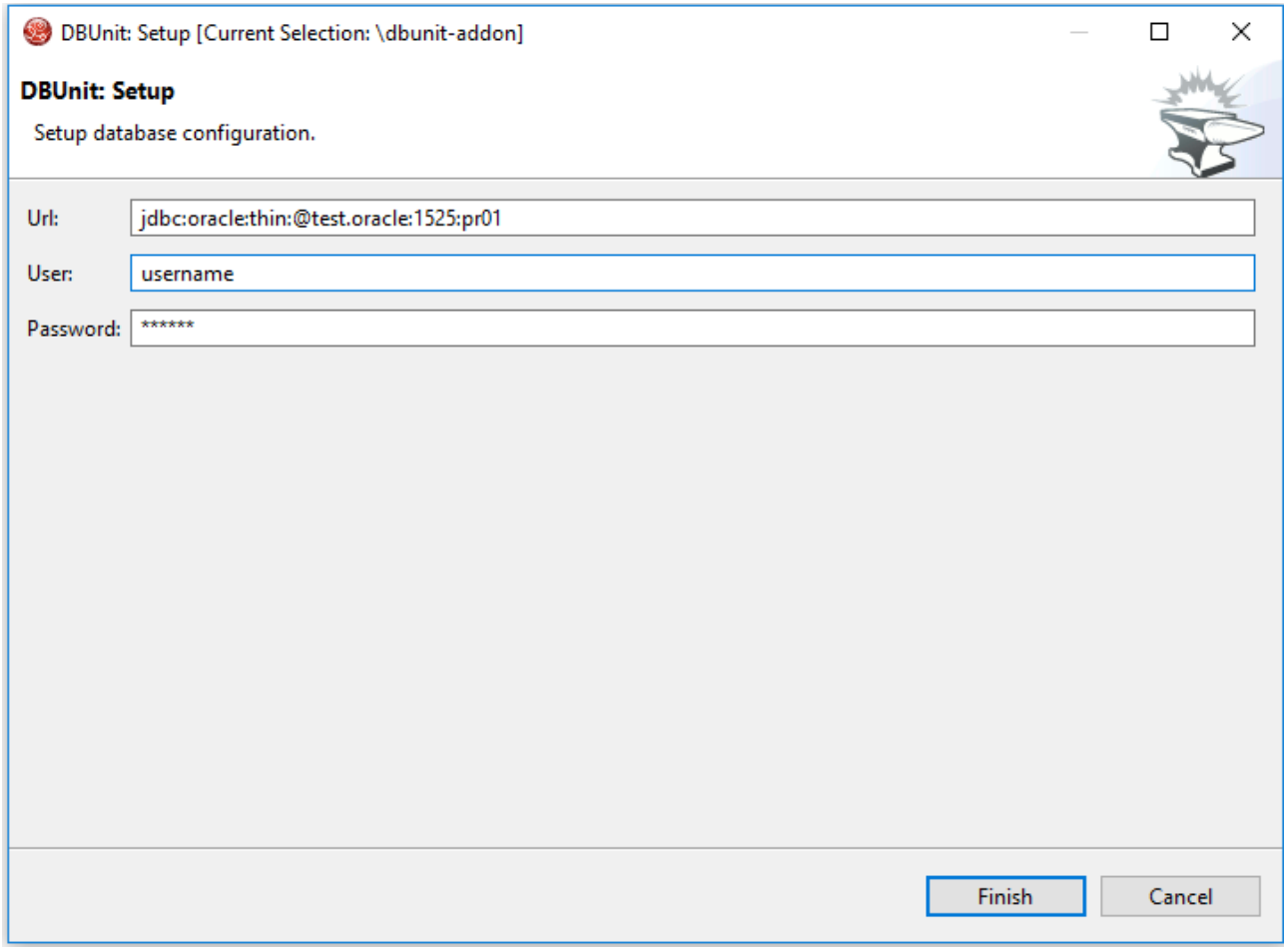
12.2.1. Installation

You will need Forge installed in your IDE or operating system. After that use install addon from git command:

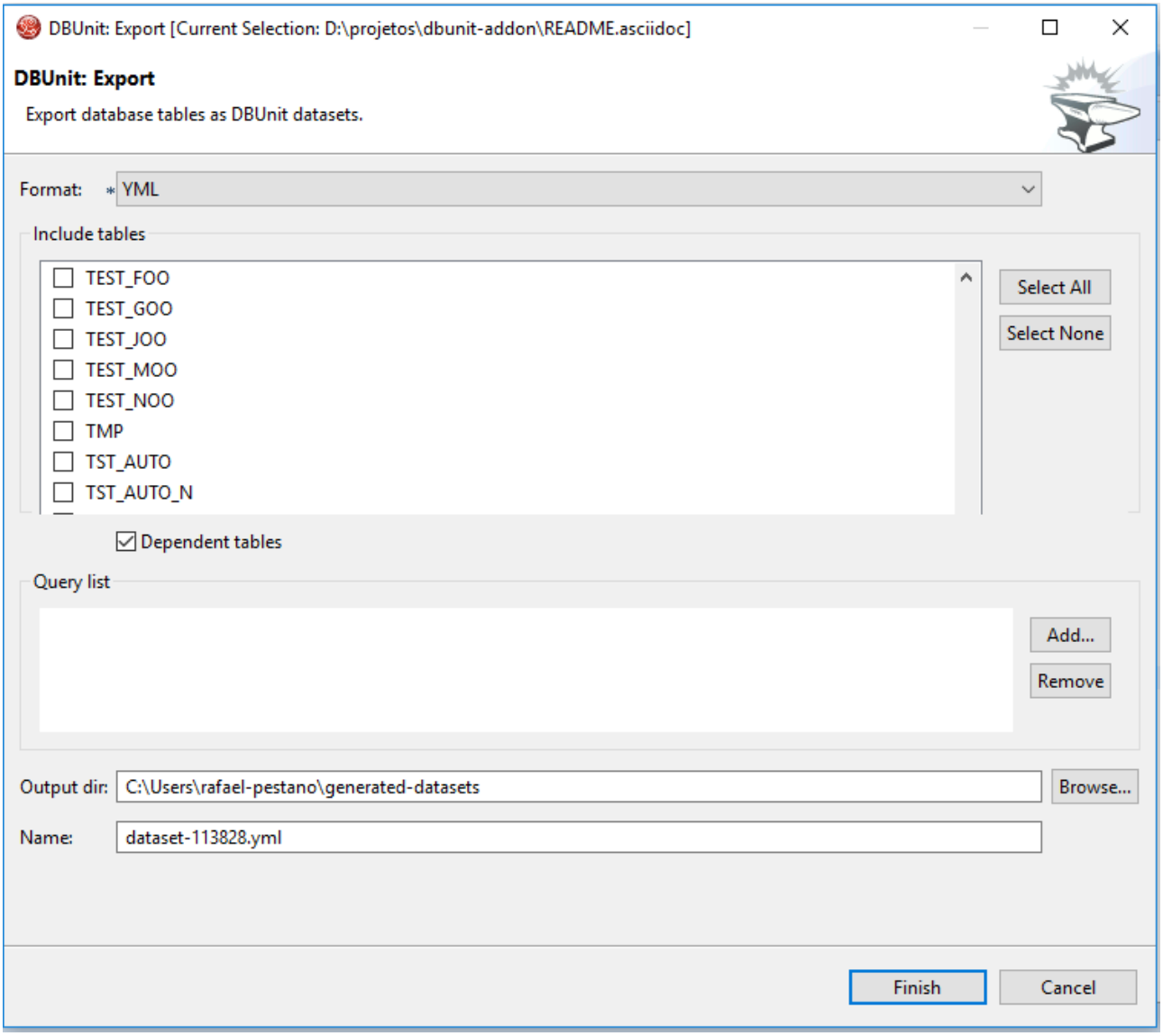
```
addon-install-from-git --url https://github.com/database-rider/dbunit-addon.git
```

12.2.2. Usage

1. Setup database connection



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



12.3. Configuration

Following table shows dataset 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.	""

Chapter 13. Detecting connection leaks

Database Rider provides a component that counts JDBC connection before and after test execution.

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

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

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

    @BeforeClass
    public static void initDB() {
        //trigger db initialization
        Persistence.createEntityManagerFactory("rules-it");
    }

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

     @Test
     @DataSet("yml/user.yml")
     public void shouldFindTwoConnectionLeaks()  {
         exception.expect(LeakHunterException.class);
         exception.expectMessage("Execution of method shouldFindTwoConnectionLeaks
left 2 open connection(s).");
         createLeak();
         createLeak();
     }

     @Test
     @DataSet("yml/user.yml")
     @DBUnit(leakHunter = false)
     public void shouldNotFindConnectionLeakWhenHunterIsDisabled() {
           createLeak();
     }

    private Connection getConnection() {
          try {
               return DriverManager.getConnection(
```

```
"jdbc:hsqldb:mem:test;DB_CLOSE_DELAY=-1", "sa", "");
            } catch (SQLException e) {
                e.printStackTrace();
                return null;
            }
        }

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