Integration Tests

New: Testing can now be done with mvn, too. Detailled information will follow. Here some short hints

- In folder "test" Create a new feature with name "de.uniol.inf.is.odysseus.test.feature"
- For each test group (see below) create a new feature and bundle in the test folder
- Add each feature to the test feature
- Use maven to create a test product for the local platform

```
mvn clean verify -Ptest,solobuild,currentOS -Dtargetfilename=${PLATFORMTARGETFILE}
// change to dir, where product is placed (here linux example)
cd ${WORKSPACE}/odysseus_dev/test/test.product/target/products/de.uniol.inf.is.odysseus.test.runner.product
/linux/gtk/x86_64/
// run test
java -Dosgi.requiredJavaVersion=1.8 -Xms1024m -Xmx1024m -jar plugins/org.eclipse.equinox.launcher_1.4.0.
v20161219-1356.jar
```

See PLATFORMTARGETFILE for further information about targetfilename.

In combination with our build tooling (Jenkins), we are running some integration tests for checking Odysseus functionalities. After each hourly build a certain server product is built and run. This instance is used for different sets of queries (and sometimes some expected output) to test different things like query languages, the executor or operators. This article shows the currently existing test components and how a new component can be integrated.

Overview of the Integration Testing

The basic stuff for the integration testing can be found in the bundle called de.uniol.inf.is.odysseus.test. It contains the runner that executes the tests and some reusable abstract classes that can be used for own test components.

Structure

The structure of the testing is as follows. There is a test-application that binds different test components (implementations of ITestComponent) via a declarative service and executes them. Each test component may have any number of so called "sub tests". For example, the nexmark-test is a test component with 5 queries. Each query is a subtest and is tested for its own. However, if one sub test fails, then the whole test component fails too. There are already some test components with a certain functionality that are explained in the following section. A sub test is bundled into a "test set". There are different test sets depending on the type of the test component. For a query test component, for example, there is a query test set that contains a query that should be executed. An "expected output test component" (which checks if the results of the query matches to an expected list of tuples) uses an "expected output test set" instead to combine a query to a certain expected output.

Furthermore, there is the possibility to manage a context. This can be used, for example, to run the same tests with a different context. A predefined basic context holds the current user (which is the "System" user) and the data-root-path (which is by default the root of the bundle). However, you may use this, e.g. to run the same tests with different users.

Running Test Suite

There is a TestRunnerProduct which can be used to run the test in Eclipse.

Adding new Queries

There are some basic concepts for adding new queries:

- The bundles of the three basic component tests are automatically scanned for new test sets
- If a test component tests the processing results with a given list (called expected output), the file of the expected output must be a csv-file in the same folder like the qry file. Furthermore, it may have the same name like the qry (for example: "example.csv" for "example.qry") . Alternatively, the name of the expected output could also one of "output.csv", "expected.csv" or "expected_output.csv".
- You can use ${BUNDLE-ROOT} within the qry-file, if you want to use some files from the same bundle (e.g. as a data-input for access operators)
- currently, only the Timeinterval is considered for metadata
- It is important, that the bundles (which are jar files after the build) are unpacked. Therefore, you have to activate this option in the de.uniol.inf.is.odysseus.test.feature under "Plug-ins" for the components (it is called "Unpack the plug-in archive after installation" on the right hand side). It is also important to add the testdata and queries to your build properties (or via the MANIFEST.MF of the test component bundle in the "Build" tab) so that they are part of the "binary build"!

Example: Adding a new Test window_sliding

We use the Test-Component under to the bundle: `de.unol.inf.is.odysseus.test.component.operator`

Here you can find the folder: testdaten containing tests.

First, you have to define a new input data set. In most cases, the easiest way would be a CSV file. It can be copied from other tests. We use input0.csv from the aggregate_time test

Second, you have to define a new query. This should be done in Odysseus Studio.In our Example we use: window_sliding.qry

```
///OdysseusScript
#PARSER PQL
#TRANSCFG Standard
#RUNQUERY
percentage = ACCESS({
    source='percentage',
    wrapper='GenericPull',
    transport='file',
    protocol='SimpleCSV',
    dataHandler='Tuple',
    options=[
        ['filename', '${WORKSPACEPROJECT}\input0.csv'],
        ['csv.delimiter', ';'],
        ['csv.trim', 'true']
        ],
    schema=[
        ['timestamp', 'STARTTIMESTAMP'],
        ['percentage', 'INTEGER']
]})
window0 = window({size = 1, type = 'time'}, percentage)
```

Here we test a simple window, that creates windows of 1 millisecond.

Start the query and look if everything compiles correct. When the query terminates use Show Stream Elements - List - Show all last elements on the query and let the query run again.

YOU SHOULD NOW CAREFULLY CHECK THE RESULT. IS THIS THE OUTPUT YOU EXPECT. This step is very important, else the test makes no sense!

If everthings seems ok, copy the whole ouput of the list window and paste it into a file window_sliding.csv.

Now go to the testbundle: `de.unol.inf.is.odysseus.test.component.operator`

- In the folder: testdaten add new new folder for your test: window_sliding
- copy the input0.csv, window_sliding.csv and window_sliding.qry into this new folder
- Change in windows_sliding.qry the filename as follows: `['filename', '${BUNDLE-ROOT}\testdaten\window_sliding\input0.csv']`

This test is now part of the test suite and will be run each time, the test components run.

Testing Behavior

Some remarks for the testing behavior.

- Simple query tests (which uses AbstractQueryTestComponent, e.g. like the compile test component) are only try to execute OdysseusScript by calling executor.addQuery. It does not check any results, semantics or processing. It only looks, if the executor fails (throws an exception) or not
- The expected output tests (which uses AbstractQueryExpectedTestCompoent, e.g. like the nexmark and the operator test component) firstly tries to execute the OdysseusScript (like a simple query test above) and then adds new sinks to the "old" sinks of the queries (instances of TICompareSink). The TICompareSInk contains already all expected output data as a list of tuples. The expected output data is read from the corresponding csv-file and is wrapped by the TupleHandler with the schema of the the "old" sink. If the test runs, the TICompareSink tries to match the result of the query with the content of the expected output data list. After the query is done, the list of the expected output must be empty (so all expected data was part of the processing results) and the test is done.
- The StatusCode enum contains all possible failure codes (+ OK if nothing fails)

Existing Test Components

test.component.compile

This test tries to execute queries, but does not check any processing results. Therefore, this test is used for testing query languages, for example, to check if the syntax is still ok and a query plan can be built by the parser. To add your own tests, simply add a qry-file within the bundles subfolder called "testdata".

test.component.nexmark

This test starts nexmark (not by using the generator but using dedicated files) and runs different queries. Furthermore, it checks, if the result of a query matches to a given expected output. Its behavior is very similar to the "operator test component" by running a query and testing the results of the query with an expected output. For example, the query1.qry is executed and the test component checks, if the results are (semantically) equal to the given expected output query1.csv. You can extend this by adding new pairs like query6.qry/query6.csv to testdata and change the NexmarkTestComponent class - the method createTestSets must be changed to return the new test set (the query6 pair).

test.component.operator

This test has very short queries, because it only tests a single operator (or some more if they are needed - like windows for aggregations 😃). Each sub folder corresponds to a sub test and each sub test is a pair of the query and the expected output. The folder is searched recursively for new pairs, so you may simply add a new subfolder with a pair of qry- and csv-file to create a new operator-test or you can use the Testcase Generator to create a new test case for an operator.

test.component.keyvalue

This test checks if the key value operators work and have the expected results. For now key value project and select and the keyvaluetotuple and tupletokeyvalue operators are tested. The input and output data is given as json files.

test.component.probabilistic

This test checks the operators and transformation rule of the probabilistic feature. The test consists of all relational operators and tests for the correct estimation of stochastic models.

test.component.jira

This test checks reported issues. Each test is named after a JIRA issue to make sure the reported issue remains fixed in future releases.

Creating New Test Components

To add your own test component you have to do the following:

- Create a new bundle using the naming convention "de.uniol.inf.is.odysseus.test.component...."
- add odysseus.core, odysseus.core.server and odysseus.test to the required plugins
- Add a new class and inherit from one the following abstract classes:
    - AbstractTestComponent for a very basic test component
    - AbstractQueryTestComponent for a test component that should install a query
    - AbstractQueryExptectedTestComponent for a test that should compare the query results with an expected output by automatically adding TICompareSinks (look above under "Testing behavior").
- Implement the needed methods
    - The createTestSets should return the list of sub tests that should be executed by the test component. The type of the test set depends on the type you choose before. So, if you used AbstractQueryTestComponent, you have to return a list of QueryTestSet, which contains the query to be executed. You can use the TestSetFactory class. It offers some methods to search and create test sets.
    - You can override the getName to name the test component.
- create a service component description for the test component class (which inherits from on the abstract classes). It has to provide the "ITestComponent" service.
- additionally, you have to check for lazy loading, so that your bundle is not started if one class is needed.
- Look also at the tips from "Adding new Queries" above.
- If the tests should run periodically, add the the component to `de.uniol.inf.is.odysseus.test.feature`.