Table Of Contents

Testing Framework

[Unit-Tests 2](#_Toc365555911)

[Workflow Tests 2](#_Toc365555912)

[Quickstart 2](#_Toc365555913)

[Detailed Description 3](#_Toc365555914)

[Workflow setup 3](#_Toc365555915)

[Providing Input Files 5](#_Toc365555916)

[Disturber node 5](#_Toc365555917)

[Required Messages and Failing Nodes 5](#_Toc365555918)

[Upload 6](#_Toc365555919)

[Running regression tests 7](#_Toc365555920)

[Running from the Command Line 7](#_Toc365555921)

[Notes 8](#_Toc365555922)

KNIME’s testing framework offers support fort two different kinds of test: unit tests (using JUnit) and workflow tests. Unit tests are suitable for testing functionality that is not tied to a node, e.g. classes to parse certain input files or algorithms. Workflow tests are the easiest way to check whether complete nodes behave as expected.

# Unit-Tests

Since KNIME is based on Eclipse which has its plug-in concept with dependencies and special classloaders, unit tests cannot be run as normal Java processes but must run within Eclipse. If you are running a JUnit test from your SDK, Eclipse keeps automatically runs the test inside a special Eclipse application. However, if you want to run unit tests in an automated way, say for nightly tests, KNIME offers a special application that runs all known unit tests at once. You need to do some preparation, though:

1. A good practice is to put unit tests for a plug-in into a fragment for the plug-in under test (e.g. *org.knime.core.testing* for *org.knime.core*). This ensures that the tests can access also classes from the plug-in which are not exported (fragments become part of the host plug-in at runtime).
2. In order for the testing application to find all testcases, you need to add a dependency to *org.knime.testing* and create a subclass of *org.knime.testing.core.AbstractTestcaseCollector* somewhere in the plug-in (e.g. *MyPluginsTestcaseCollector*). This class must be registered with the extension point *org.knime.testing.TestcaseCollector* (similar to how KNIME nodes are registered in the plug-in).
3. Add the fragment to an existing feature or create a new feature for the test fragment only and install it into KNIME
4. Run the application *org.knime.testing.NGUnittestRunner.*

The unit test runner application executes all testcases it can find and creates XML files with the results. The XML files can be analyzed e.g. by Jenkins or the JUnit View in Eclipse (via “Import…” in the menu). The application needs one command line argument:

* -xmlResultDir <dir\_name> the directory into which the results are written. It will be created if it does not exist.

# Workflow Tests

Workflow tests execute one or several nodes in workflow and therefore test a great part of the node’s functionality in a realistic setup.

## Quickstart

If you just need to refresh your memory: here’s the checklist. Further down is a more comprehensive tour.

* Create your flow, use a reader/writer combination to store the golden table, use the *Table Difference Checker* node to compare test results (or any other difference checker).
* Remove all writers from the flow.
* Reset all nodes that should be tested.
* Optionally add a *Testflow Configuration* node to your workflow, open its dialog, and
  + Set the workflow owner
  + Add required log messages (errors, warnings, infos) if applicable
  + Set required node messages (error, warnings) and if a node is supposed to fail
* Run the test locally on your computer and make sure it passes.
* Upload the workflow to the testflow server

## Detailed Description

A workflow test consists of several phases, some of which are optional:

1. The workflow is loaded. Any error that occurs during loading are reported as failures.
2. Optionally the workflow is saved immediately after loading and the re-loaded again. Any errors occurring during save and the second load are reported as failures. This test is useful for checking whether upgrades to new versions works properly (workflow format, node settings, etc.).
3. Optionally any deprecated nodes in the workflow are reported as failures (e.g. for example workflows which should contain only recent nodes).
4. Optionally all node views are opened before execution. Any errors/exceptions that occur are reported as failures. The views are kept open during execution and are updated.
5. The workflow is executed. After execution all nodes should be executed, except nodes which are supposed to fail. If either case does not hold a failure is reported. Also, all executed node should not have any error or warning messages, except if explicitly defined. Nodes that were already executed when the workflow was loaded are ignored (e.g. reader nodes which may complain about missing input files).
6. Optionally all dialogs are opened and closed. Any errors are reported as failures. Also here nodes that were saved executed are not tested.
7. All open views are closed. Any errors during close are reported as failures.
8. Optionally the executed workflow is saved into a different location. Any errors are reported as failures.
9. The workflow is closed. Any errors are reported as failures.
10. Optionally all log messages that have been written during the above steps are analysed. Any ERROR or FATAL messages are reported as failures, expect if they are expected. If any expected messages (from all categories) are missing they are also reported are failures.
11. Any uncaught exceptions during the above steps are reported as failures. This is mainly useful for exceptions in the AWT Event Thread, e.g. while updating a view or dialog.
12. Optionally all available nodes that have not been used in a test workflow are reported (as failures) in the end.

In order to create a test workflow, you first need to install the “KNIME Testing Framework” (or get the projects *org.knime.testing* and *org.knime.testing.ui* from Subversion).

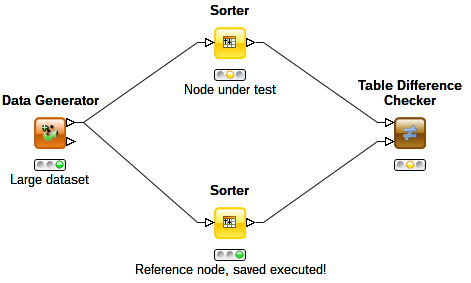
### Workflow setup

During test execution all nodes of a flow that are not already executed are run. Meaning the test flow – at the time of upload – should be in a state where all nodes that should *not* be tested must be *executed* and those that should be tested must be *executable*.

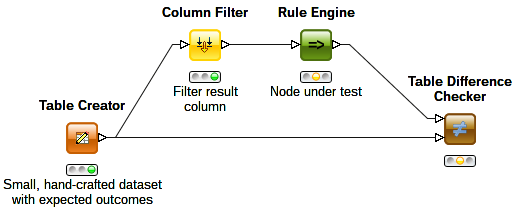
The general idea of a test workflow is to compare the current output of a node with a reference output. The reference can either be the output of the same node at a certain point in time (usually when the workflow was created) or data from “external” sources (e.g. manually created or produced by another program). The former tests are useful to ensure that a node consistently creates the same output over time (regression tests) whereas the latter tests are for checking if the output is correct. The latter is of course preferred since checking whether a node creates the same – wrong – results in the next few years is a bit pointless.

In order to test the output of a node add the *Difference Checker* node to the flow (category *Testing*). It has two inputs and compares the two incoming tables (actual data and reference data) during execute. For each column a specific checker can be configured. The default is to check for equality, i.e. the values in the test table and the reference table must be identical. For certain column types the check can be relaxed a bit, e.g. numeric columns allow for specifying an epsilon by which the two values may differ. In multiline string columns the line delimiters can be ignored (useful is tests are run on different operating systems). Additional checkers may exist for other column types.

Now connect the node to test to the “Test Table” input of the *Difference Checker* and the reference table to the “Reference Table” input. The two workflows depicted below show a test workflow using external reference data and self-created reference data, respectively.



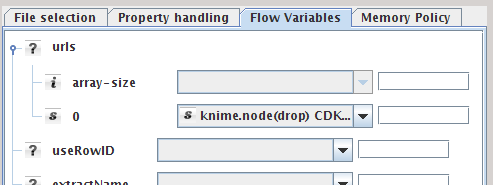
Testflow with self-created reference data



Testflow with external reference data

### Providing Input Files

Usually your testflow needs some kind of input that is read by a reader node. Since the read file usually does not exist on other systems, these files should be shipped with the workflow. This can either be done with the *drop* mechanism or workflow-relative URLs (if the reader supports URLs). For the drop-mechanism, for build for workflow save and close it. Then create a directory called *drop* inside the node’s directory in the saved workflow, e.g. File Reader (#1)/drop. Then open your workflow again and select the file in the drop folder as input file in the dialog. Now go to the *Flow Variables* tab in the node’s dialog, locate the setting for the input file and select the file in the drop folder in the selection box.



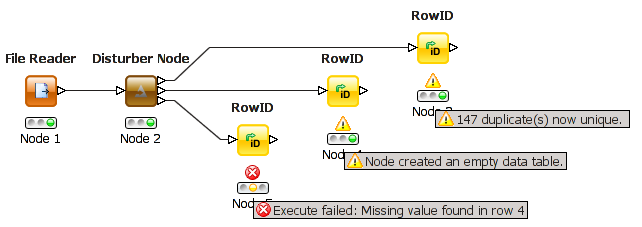
An easier way is using workflow-relative URLs. Just place the input file somewhere inside the workflow and use *knime://knime.workflow/file.txt* as URL (knime.workflow is a fixed identifier).

### Disturber node

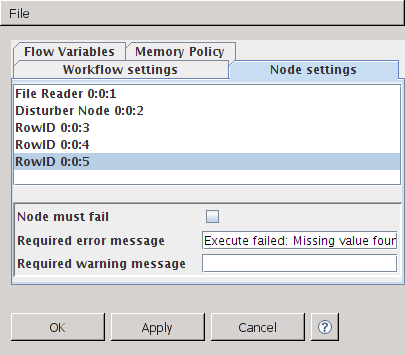
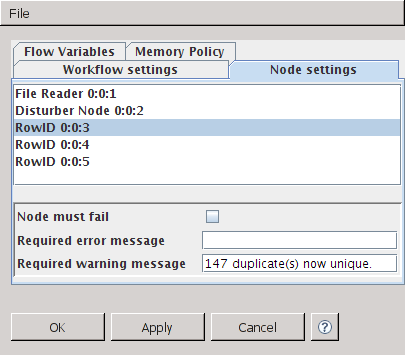
The *Disturber Node* can be used to test the robustness of your code. This node has one input and three outputs: An output providing the unaltered input table, another delivering an empty table, and the third delivers the input table with randomly injected missing values. You can copy the nodes of your flow three times and connect each copy to an output of the disturber. All nodes should be able to gracefully handle empty tables and missing values somehow.

### Required Messages and Failing Nodes

To ensure that a warning and/or an error appears on a node in the workflow, or to check for messages in the log file/console you can use the Testflow Configuration node. In the dialog enter the messages in the corresponding places. Suppose you have a workflow that has the following status after it has been executed:



In the Testflow Configuration node enter the warning and error messages on the nodes as follows:

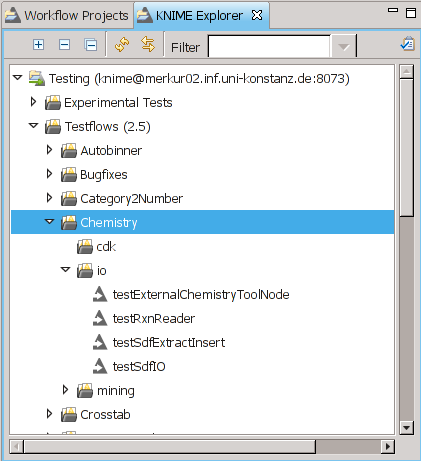


This message fields are pre-filled with the current error message of the selected node (if there is any), indicated by the grey text colour. If you want to really use this message click in the text field and possibly edit it. Once the text colour is black, the message is used.

If you want to assure that the log file (or console) contains certain error, warning, or info messages you can enter them in the “Workflow settings” tab of the dialog.

If the expected outcome of your workflow is that all nodes are executed, no and no errors or warnings have occurred, than you don’t need a Testflow Configuration node.

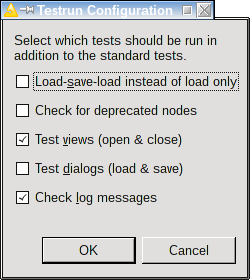
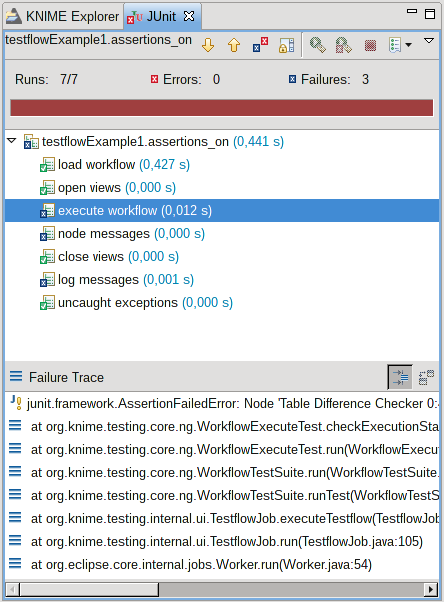
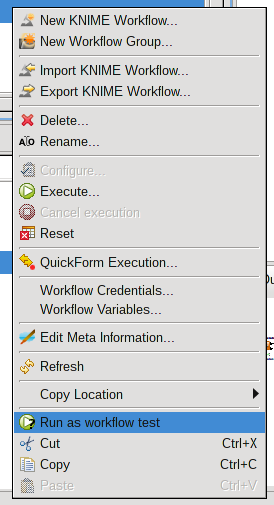
### Upload

****Upload the flow into an appropriate workflow group on your testflow server. All workflows in this or any sub-groups will be executed.

## Executing Workflow tests

Workflow tests can either be run within KNIME (useful while building a test workflow), or with a headless application (useful for automated tests)

### Running inside KNIME

In order to run the tests locally, you can right-click on a workflow in the KNIME Explorer and select “Run as workflow test”. A dialog will then pop up, in which you can select the different test steps that should be run (see their explanation above). The testflow is now run and the results are display in the JUnit view.

### Running from the Command Line

The workflow tests can also be run without a graphical interface from the command line. This is achieved by calling KNIME with the following arguments:

**knime -application org.knime.testing.NGTestflowRunner –nosplash -consoleLog**

This starts the Eclipse framework (activating all installed plug-ins – without showing the splash screen) and runs the testing application. You can pass the following arguments to the application (after the above parameters):

* The **-root** argument specifies the root directory of where the tests are located. The application traverses this directory and all sub-directories and runs all workflows it finds. Workflows may also be packed in ZIP files.
* The **–server** argument specifies the URI of a workflow group on a KNIME server where the testing workflows are stored. The format of the URI is **knimefs://<user>:<password>@host[:port]/workflowGroup1**

Either **–root** or **–sever** must be provided.

* With the **-pattern** argument a subset of tests can be selected. Only testcases whose directory name (the directory the **workflow.knime** file is located in) matches the specified pattern (which is a regular expression) are run. Specifying “**.\***” matches all tests.
* -xmlResultDir specifies a directory into which the test results (one per workflow) are written.
* -xmlResultFile specifies a single file where all results are written to. Either –xmlResultDir or –xmlResultFile must be specified.
* If want to load-save-load the workflow before execution (instead of loading it only once), pass –loadSaveLoad.
* If –deprecated is given then all deprecated nodes are reported as failures.
* -views opens all views prior to execution.
* -dialogs opens all dialogs after execution
* All log messages are analysed if you pass –logMessages.
* If you to report all nodes that have not been tested in a workflow, pass –untestedNodes. This argument takes a regular expression of node factory names. Only untested nodes matching the expression will be reported.
* If you want to save the executed testflows, pass –save with a destination directory name.
* By default workflows are canceled after 5 minutes. With –timeout you can specify a different default timeout.

# Notes

* If a node should be tested with different settings or different input data you need to create one test for each set of settings/input data. You can either create several testflows with identical structure but different settings or add several independent branches into one workflow.
* It is possible to test multiple nodes in one regression test (as all nodes that are executable will be executed) – it’s not advisable though. If one node is broken, there would be many tests failing and it would be hard to figure out which node is actually causing the pain. It’s just harder to debug. Create one test per node. The more the merrier.

© Copyright 2003 – 2012, Chair for Bioinformatics and Information Mining (Prof. M. Berthold) and KNIME GmbH, Konstanz, Germany.