

testium

Introduction training

François Dausseur

May 1, 2026



Summary

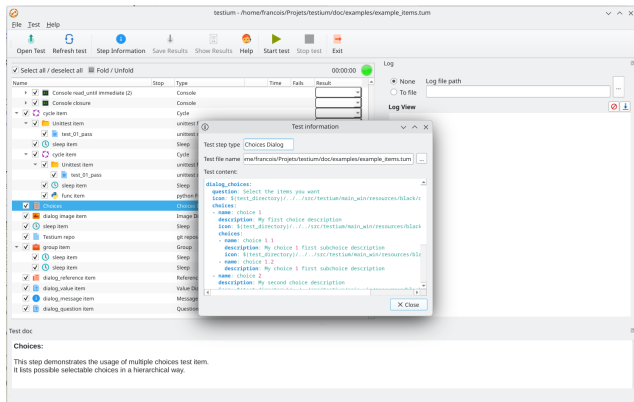
- 1 Introduction
 - Main features
 - Advanced features
 - Documentation and example code
 - License
- 2 Setting up *testium*
- 3 Test items
 - Tests items summary
 - Tests items common features
 - Tests items
- 4 *testium* functionalities
- 5 Conclusion

Summary

- 1 Introduction
 - Main features
 - Advanced features
 - Documentation and example code
 - License
- 2 Setting up *testium*
- 3 Test items
 - Tests items summary
 - Tests items common features
 - Tests items
- 4 *testium* functionalities
- 5 Conclusion

testium main features

- Automates the execution of tests
- GUI or console mode
- Stores logs
- Generates test reports
- Usable for production and engineering testings
- Mainly based on:
 - Python3
 - QT6
 - Jinja2



testium input files

- Test scripts (.tum)
 - Files written in **YAML**
 - Wide range of test items
 - Loops, Variables, conditional execution
 - Modular templating using **Jinja2**
- Configuration files (.yaml, .yml)
 - **YAML** test configuration / parameters

```
# All sub directory in items are evaluated as a list
items: <| [os.path.basename(f.path) for f in
↳ os.scandir(os.path.join("${test_directory}", "items")) if f.is_dir()] |>

# Parameters of the validation execution
validation_report_path: ${test_directory}/../tmp/
validation_report_file: validation

# various parameters
windows_prompt: ">"
linux_prompt: "$ "
[...]
```

Listing: Example of config file

```
config_file:
  - param.yaml
  {% for item in items %}
    - "items/{{ item }}/param.yaml"
  {% endfor %}
main:
  name: testium validation suite
  steps:
    - group:
      name: Test preparation
      steps:
        - let:
          name: Set test variables for Linux
          condition: "'${os}' == 'Linux'"
          values:
            - terminal_prompt: ${linux_prompt}
            - psep: "/"
        - let:
          name: Set test variables for Windows
          condition: "'${os}' == 'Widarkbluendows'"
          values:
[...]
```

Listing: Example of test file

- **testium** can be invoked with the following options:
 - **-h**: command line help
 - **-b**: batch mode (execution of the test script without GUI)
 - **-c**: Use a specific config file
 - **-i**: Set a specific include path
 - **-g**: Run in debug mode (for vscode debug only)
 - **-r**: Run the test and close

Test scripts

- Hierarchic list of test items
 - **main** is the root of the test script
 - **config_file** list of config/params files
 - **report** global report parameters section
- **main** contains test items
 - executed iteratively from top to bottom
 - container items can have children
 - **loop**
 - **group**
- **!include** directive
 - include sub-tests
 - allow the use of templates

```
config_file:  
  - param.yaml  
  
main:  
  name: Training file demo  
  steps:  
    - item1:  
      [...]  
  
    - loop:  
      name: Training file demo  
      steps:  
        - item2:  
          [...]  
  
    - item3:  
      [...]  
  
  [...]  
  
  - !include path/to/my/subtest.tum  
  
  [...]  
  
report:  
  [...]
```

Listing: A main test file

Variable expansion & global dictionary

- Persistence between steps
- Variable expansion: `$(my_variable)`
 - The `my_variable` value is taken in the global dictionary if the key exists
- Accessed from python functions:
 - `tm.gd`
 - `tm.setgd`
 - `tm.delgd`
- Built-in variables:
 - `test_directory`: the folder of the main tum file
 - `test_main_file`: the main tum file
 - `os`: The OS used by the system (Linux or Windows)
 - `host_name`: The hostname of the machine
 - `home`: The home directory of the user
 - `test_outputs`: The list of the paths of the test log and test report
 - `testrun_date`: The date when the test started YYYY-MM-DD
 - `testrun_time`: The time when the test started HH:MM:SS

Inline evaluation of python statements

- `<| statement |>`
 - Evaluation of the python *statement* enclosed by `<|` and `|>`
 - Evaluated in:
 - YAML config files
 - `.tum` files
 - compatible with variables expansion
 - compatible with variables expansion

```
[...]
- loop:
  name: Test loop
  stop_on_failure: false
  iterator: ${stuff_number}
  steps:

    - json_rpc:
      name: Start a important step
      udp:
        server: ${dut_eth0_ip}
        snd_port: ${ctrl_snd_port}
        rcv_port: ${ctrl_rcv_port}
      timeout: 30
      mute: False
      steps:
        - query:
          method: record.start
          params:
            - slot: <|${loop_index} + 1|>
              input: <|${test_data})[${loop_index}]|>
              quality: medium
[...]
```

Listing: variables expansion and evaluation examples

Inline evaluation of python statements - Examples

- simple

- `<|$(loop_index)+1|>`

- table indexing

- `<|$(my_table))[$(loop_index)]|>`

- string

- `<|"$(video_path)+"$(vstream_formats))[0]+".mkv"|>`

- Random list of integers

- `<|[random.sample(range(0,8), k=4)]|>`

Links for documentation and example code

- Git repository
 - https://git.beafrancois.fr/v_and_v/testium
 - Also possibility to do suggestions or raise tickets
- Reference documentation
- *testium* validation suite

- Distributed under the **EUPL-1.2**
 - European Union Public Licence v1.2
 - Open-source, OSI / FSF approved
 - Governed by the law of the licensor's EU country (France)
- SPDX identifier: **EUPL-1.2**
- Files in the repository:
 - **LICENSE** (full legal text)
 - **CONTRIBUTING.md** (contribution rules)
- Contributions follow the *inbound = outbound* rule
 - Any patch you submit is licensed under the same EUPL-1.2
 - No CLA, no copyright assignment
- **You may:**
 - Use **testium** for any purpose, including commercial
 - Read, modify, redistribute the source code
 - Combine with code under GPL, AGPL, LGPL, MPL, CeCILL, EPL... (compatibility appendix)
- **You must:**
 - Keep the copyright and license notices
 - Redistribute modified versions under the EUPL (or a compatible license)
 - Publish your modifications even when offering **testium** as a network service (SaaS clause)
- **No warranty, no liability of the author**

Summary

- 1 Introduction
 - Main features
 - Advanced features
 - Documentation and example code
 - License
- 2 Setting up *testium*
- 3 Test items
 - Tests items summary
 - Tests items common features
 - Tests items
- 4 *testium* functionalities
- 5 Conclusion

Setting up *testium*

- Binary installation

- Place the binary file of testium in a directory
- Add the testium directory in the PATH

- Automatic setup from sources

- run **run.bat**, **run.ps1** or **run.sh** depending on your OS and command interpreter

```
cd C:\path\to\testium
run.bat
```

- Manual setup from sources

- Create the python virtual environnement

```
python3 -m venv C:\path\to\venv
```

- activate the environnement

```
C:\path\to\venv\Scripts\activate.bat
```

- install the requirements

```
cd C:\path\to\testium
pip install -r src\requirements.txt
```

- run **testium**

```
python -m src/testium
```

Summary

- 1 Introduction
 - Main features
 - Advanced features
 - Documentation and example code
 - License
- 2 Setting up *testium*
- 3 Test items
 - Tests items summary
 - Tests items common features
 - Tests items
- 4 *testium* functionalities
- 5 Conclusion

Test items - 1/2

Item	Description
<code>check</code>	Checks for a value or expression
<code>console</code>	Console steps (serial, terminal, telnet, tcp, ssh)
<code>dialog_choices</code>	Asks for a choice in list
<code>dialog_image</code>	Displays an image
<code>dialog_message</code>	Displays a message
<code>dialog_note</code>	Asks to enter a note
<code>dialog_question</code>	Asks a question with a yes/no choice
<code>dialog_references</code>	Asks for references
<code>dialog_value</code>	Asks for a value, entered manually
<code>py_func</code>	Python function call (from a file)
<code>lua_func</code>	Lua function call (from a file)

Test items - 2/2

Item	Description
<code>group</code>	container for grouping things
<code>jsonrpc</code>	JSON-RPC test item
<code>let</code>	Defining variables
<code>loop</code>	Container for repeating things
<code>parallel</code>	Container for running branches concurrently
<code>plot</code>	Runtime plot utility
<code>report</code>	Extract a report file
<code>run</code>	Runs a new instance of <i>testium</i>
<code>sleep</code>	Wait (with or without dialog)
<code>unittest_file</code>	Python unittest file

Items common attributes

- Mandatory
 - **name** : The test item name
- Optional
 - **stop_on_failure**: Stop the test if there is one failure
 - **execute_on_stop** : Execution of the item when the test is stopped
 - **skipped** : **true** to skip the execution of the test
 - **doc** : The documentation of the test item
 - **key** : A Key used to filter the reports
 - **condition** : If **true**, will execute the test step. Otherwise, the test step is skipped
 - **process_result** : Processes the result
 - **expected_result** : Expected result of the item
 - **store_result** : Stores the result in a global variable
 - **no_fail** : If **true**, forces the step result to be **PASS**

Items common variables

- for all test items
 - `last_step_result` : contains the last test step result
 - if no result returned by the test
 - `PASS`, `FAIL` or `SKIPPED`
 - `ts_start_<item_name>` : The timestamp at the beginning of the item name
 - `ts_end_<item_name>` : The timestamp at the beginning of the item name
- `console` test item specific variables
 - `cn_<item_name>` : Containing the last data which has been read in the console
- `py_func` test item specific variables
 - `pfn_<item_name>` : The returned value of the last `py_func` test item execution
- `lua_func` test item specific variables
 - `lfn_<item_name>` : The returned value of the last `lua_func` test item execution
- `loop` test item specific variables
 - `loop_index` : loop index (starting from 0)
 - `loop_index_inverse` : loop index in reverse order
 - `loop_count` : total number of iterations
 - `loop_param` : current value of the loop iterator

- **open** : Open a console using the specified protocol
 - telnet
 - ssh
 - serial
 - rawtcp
 - terminal
- **close** : Close the console
- **write** : Write the string in the console
- **writeln** : Write the string in the console with \n
- **read_until** : Read until an expected string
 - **expected** : The expected string to be received
 - **timeout** : The timeout to read the expected until it **FAILs**
 - **no_fail** : Do not **FAIL** even if a timeout occurs

```
- console:
  name: test name in GUI
  console_name: console name in dict
  steps:
    - open:
      protocol: telnet
      telnet_host: ${target_ip}
      telnet_port: ${target_port}
    - writeln: reset
    - read_until: {expected: U-Boot, timeout: 50}
    - write: ${boot_vxworks_1}
    - writeln: ${boot_vxworks_2}
    - read_until:
      expected: U-Boot
      timeout: 15
    - read_until:
      expected: Something that will never occurs
      timeout: 5
      no_fail: True
      mute: True
    - close:
```

Listing: **console** usage example

py_func test item function call

- The python file must import helpers module `py_func.tm`
- Calls the function `exec` from a Python class
 - The Python class must be inherited from `tm.FunctionItem`

```
import py_func.tm as tm

class TestItemPyFunc(tm.FunctionItem)

    def exec(param1, param2, param4, param4):
        ...
        self.reportValue('my_reported_value', reported_value)
        print(self.reportedValues())
        return 10
```

- Calls a Python function by its name

```
def dummy_func(param1, param2, param4, param4):
    ...
    return 10
```

- To get a variable from the global dictionary : `tm.gd("global_dict_key")`
 - To set a variable from the global dictionary : `tm.setgd("global_dict_key", value)`
- `context_id`: share a persistent subprocess across `py_func` items

lua_func test item function call

- The lua file must import helper's module `testium`
- Calls a Lua function by its name

```
tm = require("testium")

function dummy_func(param1, param2)
    local value = tm.gd("global_dict_key")
    ...
    return 10
end
```

- To get a variable from the global dictionary : `tm.gd("global_dict_key")`
 - To set a variable from the global dictionary : `tm.setgd("global_dict_key", value)`
- `context_id`: share a persistent subprocess across `lua_func` items

context_id shared persistent subprocess

- By default, each `py_func`/`lua_func` call spawns its own subprocess
 - module-level state is **not** preserved between calls
- `context_id`: items sharing the same id reuse the **same** persistent subprocess
 - subprocess lives until the end of the test run
 - required for non-JSON-serializable objects (connections, handles...)
- `tm.setgd`/`tm.gd` always work across all items for serializable values
 - non-serializable values (`py_func` only) are kept locally in the subprocess

```
- py_func:
  name: open connection
  file: my_script.py
  func_name: open_connection
  context_id: my_context
  expected_result: ok

- py_func:
  name: use connection
  file: my_script.py
  func_name: use_connection
  context_id: my_context
  expected_result: open

- lua_func:
  name: produce value
  file: my_script.lua
  func_name: produce
  context_id: my_context
  param:
    - hello

- lua_func:
  name: consume value
  file: my_script.lua
  func_name: consume
  context_id: my_context
  expected_result: hello
```

- **steps:** The list of items in the group

```
- group:
  name: Group Item
  condition: "'$(OS)' == 'Linux'"
  steps:
    - unittest_file:
      test_file: test_prod_rio6_8093.py
      test_method:
        ...
    - sleep:
      timeout: 10
```


loop test item

- iterator
 - The number of iterations of the loop
 - The list of each iteration parameter
 - Not defined: infinite loop
- **steps**: The list of items to loop
- **exit_condition**
 - calls a Python function and stop the **loop** if it does not return **False**
- Variables:
 - **loop_param**: The loop iterator
 - **loop_index**: The loop index

```
- loop:
  name: Cycle Temperature
  iterator: 10
  steps:
    - unittest_file:
      test_file: test_prod_rio6_8093.py
    - py_func:
      name: function test item
      file: scriptTestFile.py
      func_name: funcToBeExecuted
      param:
        - $(loop_param)
  exit_condition:
    file: script_name.py
    func_name: methodName
```

Listing: **loop** usage example

parallel test item

- **branches**: list of concurrent branches; each branch has its own **steps**
- **sync**
 - **all** (default): wait for every branch
 - **any**: stop the others as soon as one finishes
- **wait_for** (per branch): poll a condition before running the branch's steps
- Each branch runs in its own thread:
 - Live output prefixed [**<branch>**]
 - One report row per branch (clean, no interleaving)

```
- parallel:
  name: Read sensors
  sync: all
  branches:
    - name: Temperature
      steps:
        - py_func:
            name: read temp
            file: sensors.py
            func_name: read_temp
    - name: Pressure
      wait_for:
        condition: <| "$(temp_ready)" == "1" |>
        timeout: 5
      steps:
        - py_func:
            name: read pressure
            file: sensors.py
            func_name: read_pressure
```

Listing: **parallel** usage example

dialog_x test items

- Dialogs pop-up a message box with a question, information or more
- Dialogs require an action of the operator
- Available dialogs are:
 - `dialog_message` : Display a message to the operator
 - `dialog_question` : Ask a question and **FAIL** if the answer is no
 - `dialog_note` : The operator can write a small note
 - `dialog_value` : The operator enter a value
 - `dialog_references` : Ask for the reference, revision and serial number
 - `dialog_image` : A `dialog_question` with an image
 - `dialog_choices` : A choice of items to check

- Add or set a variable in the global dictionary
 - **values**: list of key/values which are to be recorded

```
- let:  
  name: Let Item  
  values:  
    key1: value1  
    key2: value2
```

- A simple test item returning **FAIL** if one of its **values** is **false**

```
- check:  
  name: check test item example  
  values:  
    - True  
    - <|$(last_step_result) == 3|>  
    - <|"my string" in "$(my_global_variable)"|>
```

- sleeps the requested time
 - **timeout**: the time to sleep in seconds
 - **dialog**: if the sleep shows a dialog (**false** by default)

```
- sleep:  
  name: sleeps 10 secs  
  timeout: 10  
  dialog: true
```

json_rpc test item

- JSON-RPC is a remote procedure call standard
- transport protocol can be chosen among:
 - udp
 - a previously opened **console**

```
- json_rpc:
  name: JSONRPC UDP query waiting for reception
  key: $(test)_PASS
  udp: {server: localhost, snd_port: 4323, rcv_port: 8765}
  timeout: 1
  steps:
    - open:
    - query:
      name: echo
      method: echo
      params:
        - Hello World
        - {a: 1, b: "hello"}
      timeout: 1
    - close:
```

Listing: **json_rpc** query example, waiting for server answer

```
- json_rpc:
  name: JSONRPC UDP query not waiting (only send)
  key: $(test)_PASS
  udp: {server: localhost, snd_port: 4323, rcv_port: 8765}
  timeout: 1
  steps:
    - open:
    - query:
      method: echo
      params:
        - {b: "olleh", a: -1}
        - World Hello
      id: 3095372
      no_wait: True

[...]
- json_rpc:
  name: JSONRPC UDP Reception
  key: $(test)_PASS
  udp: {server: localhost, snd_port: 4323, rcv_port: 8765}
  timeout: 1
  steps:
    - receive:
      id: 3095372
    - close:
```

Listing: **json_rpc** query and answer separated

Other test items

- `plot`: Displays and stores time series data
- `run`: Execute a new instance of *testium*
- `unittest_file`: Runs a Python file based on the unittest framework

Summary

- 1 Introduction
 - Main features
 - Advanced features
 - Documentation and example code
 - License
- 2 Setting up *testium*
- 3 Test items
 - Tests items summary
 - Tests items common features
 - Tests items
- 4 *testium* functionalities
- 5 Conclusion

Post-processing test items results

- You can process the result directly in the item using :
 - `process_result`: To process the result of an item Done before `expected_result`
 - `expected_result`: Set the expected result of the item
- `process_result`
 - Python evaluation of the expression
 - `$(result)`: is *mandatory* in the expression
 - It can typically
 - Extract the relevant data from a complex result
 - Change the type of the result
 - Simplify the result
- `expected_result`
 - Check if it equals the test item result (processed or not)

Result processing example

```
- py_func:
  name: Get device status
  file: device.py
  func_name: get_status
  process_result: <|$(result)['temperature']|>
  expected_result: 25.0
```

```
- console:
  name: Read firmware version
  console_name: serial_console
  steps:
    - writeln: version
    - read_until:
      expected: "version:"
      timeout: 5
  process_result: <|$(result).split()[-1]|>
  expected_result: "1.2.3"
```

store_result attribute

- Stores the item result into a named global variable
 - accessible via `$(variable_name)` in subsequent items
- If the item returns a value (e.g. `py_func`), that value is stored
- If `process_result` is also set, the post-processed value is stored
- If the item returns no value, stores the status string "PASS" or "FAIL"
 - evaluated *after* `expected_result` but *before* `no_fail`
 - captures the real outcome even when `no_fail: True`

```
- py_func:  
  name: Read sensor  
  func_name: read_temperature  
  store_result: temperature
```

```
- py_func:  
  name: Check temperature in range  
  func_name: check_range  
  param: [$(temperature), 20, 30]
```

```
- console:  
  name: Send command  
  console_name: device  
  steps:  
    - writeln: reboot  
    - read_until: {expected: "ready", timeout: 10}  
  store_result: reboot_status
```

```
- py_func:  
  name: Use reboot status  
  func_name: log_status  
  param: [$(reboot_status)]
```

Includes

- a `.tum` file can be included from another `.tum` file
- There is no limit on the number of included files
- The included file must not have a `main` defined

```
#include with the sub-sequence reference mechanism
sequence: &included_sequence
  !include included_file.tum

main:
  name: Test example
  steps:
    - test_item1:
        name: test_1
    - *included_sequence

#include can also be inserted directly within the steps list
- !include included_file.tum
```

Templating

- A template engine is a files preprocessor
- applies to all *testium* files
 - main test file
 - any included file
- Files are processed at the moment the test is opened
- recursively on all included sub-tests

Template engine syntax

- Any test file can use the Jinja syntax for templates
 - [Template Documentation](#)
- expressions
 - `{{ my_expression }}`
- control sequences
 - `{% if condition %} ... {% endif %}`
 - `{% for iterator %} ... {% endfor %}`
 - ...
- All config-file parameters can be used in templates

```
items:
- common
- check
- console
- cycle
- dialogs

config_file:
- param.yaml

main:
[...]
```

```
{% for item in items %}
# item test
- let:
  name: test constants
  values:
    test: {{ item }}
    test_path: items/${test}
- group:
  name: {{ item }} test
  action:
    - !include items/{{ item }}/test.seq
    - !include items/report.seq
{% endfor %}
[...]
```

Listing: `param.xml` followed by `main.tum`

Template parameters

- main file

```
main:
  name: Test example
  steps:
    - test_item1:
        name: test_1
    - !include
        file: included_template_file.tum
        inclusion_parameter_1: param1
        inclusion_parameter_2: param2
```

- included_template_file.tum

```
- test_item:
    name: {{ inclusion_parameter_1 }}
- {{ inclusion_parameter_2 }}:
    name: test_3
# The following construction is not allowed and will fail to load:
- test_item:
    name: {{ $(inclusion)_parameter_3 }}
```


JINJA syntax vs *testium* syntax

- Jinja and *testium* syntax super-imposed
 - Jinja is evaluated first, when the test is loaded
 - then *testium* displays the result of template pre-processing
- loops
 - Jinja : loops result in a flat sequence of steps
 - *testium* : The loop is a test item and its iterations can depend on runtime parameters
- conditional execution
 - Jinja : The item appears or not, depending on static configuration parameter, or jinja syntax
 - *testium* : The condition can depend on runtime parameters

report section and test items

- Supports sqlite, Junit, JSON, txt formats
- Designed to ease the addition of new formats if any specific request
- Added to the root of the main .tum file
 - **path**: The report path
 - **file_name**: The name of the report file
 - **export**: The type of export
 - **log_stored**: Stores the log extracts in the report
- the **report** test item can also generate some intermediate reports
 - the **key** test item parameter allows to filter reported steps

```
report:
  enabled: True
  log_stored: True
  export:
    junit:
      path: $(validation_report_path)
      file_name: $(validation_report_file).junit
```

Listing: **report** example

Summary

- 1 Introduction
 - Main features
 - Advanced features
 - Documentation and example code
 - License
- 2 Setting up *testium*
- 3 Test items
 - Tests items summary
 - Tests items common features
 - Tests items
- 4 *testium* functionalities
- 5 Conclusion

Thank you

- François Dausseur
- fdausseur@free.fr
- https://git.beafrancois.fr/v_and_v/testium