

# Table of Contents

- Picom mechanism** ..... 1
- What is a Picom ?** ..... 1
- Principle** ..... 1
  - File-based dialog ..... 1
  - Context ..... 2
- Client API** ..... 3
  - Submitting a Picom Task ..... 3
  - Creating a new Picom ..... 3



# Picom mechanism

## What is a Picom ?

Picom stands for Process Intelligent COMponent. A picom is a component run as a task (PicomTask) that runs a user-defined script in the name of the user that is able to interact with Gateway and in particular to submit other tasks.

Initially, the picom was designed to be a very generic mechanism that would be specialized into different standard workflow control components (Loop, Switch/Case, Containers, Parsers). The mechanism has proven that it is much more powerful (business process integration, dynamic workflows, recursive workflows, AI oriented workflows...)

## Principle

### File-based dialog

The principle of the picom script is to do a file-based dialog with Gateway. The picom script reads in the runlog all the information :

- about itself,
- the context it's running in,
- the template tasks it can use as model for the new tasks it,
- the tasks it has spawned previously

The runlog is the center of the dialog between Gateway and the picom script.

The runlog contains :

- the picom script (python)
- the picom description (JSON)
- the tasks templates (JSONs)
- the log of the different script executions

```
$ tree -L 2 /home/michael/hpcgateway/runlog/michael/2018/02/21/picom_4857
/home/michael/hpcgateway/runlog/michael/2018/02/21/picom_4857
├── picom.json
├── picom.log
├── script.py
├── steps
│   ├── 0
│   ├── 1
│   ├── 2
│   ├── 3
│   └── 4
└── templates
```

```
├── template_0000.json
├── template_0001.json
└── template_0002.json
```

There is also a sub-directory for each 'step' of the picom execution. Each step sub-directory contains:

- the context description (read and written by the script)
- the description of all the tasks before the script execution
- the description of the tasks the must be submitted after the script execution (written by the script)
- the status of the picom task (written by the script)

```
$ tree -L 2
/home/michael/hpcgateway/runlog/michael/2018/02/21/picom_4857/steps/3
/home/michael/hpcgateway/runlog/michael/2018/02/21/picom_4857/steps/3
├── context.json
├── lastTasks
│   ├── task_0000.json
│   ├── task_0001.json
│   ├── task_0002.json
│   ├── task_0003.json
│   └── task_0004.json
├── nextTasks
│   ├── task_000.json
│   ├── task_001.json
│   ├── task_002.json
│   ├── task_003.json
│   └── task_004.json
└── status.json
```

## Context

The executions of the picom tasks and its sub-tasks is influenced by the *context* of the execution. The context is keep the trace of the executions and it propagated to the sub-tasks of the picom task.

## Step and maxStep

The *step* correspond to the different executions of the picom script.

The step is limited by a field *maxStep* in the context (default is 100).

## Depth and maxDepth

The *depth* is the depth of the task hierarchy. The picom task that is created by the user has a depth 0 and its immediate children have a depth 1.

The depth is limited by a field *maxDepth* in the context (default is 3).

The depth and its limitation are crucial in the recursive picoms to avoid the picoms to run wild.

## Client API

### Submitting a Picom Task

### Creating a new Picom

The code below is an example of Picom creation from the client API.

```
# step 0 - define the picom script
def picom_body(ex):
    """
    The function defining the content of the Picom
    :param ex: the picom executor
    :return:
    """

    sleep = picom_task.get_input('sleep')
    if ex.context.step > 5:
        ex.stop()

    if not ex.tasks.last_tasks:
        for i in range(0, 1):
            task = ex.tasks.new(template='test task', name='task ' + str(i),
submit=True)
            task.set_input('sleep', sleep)
            task.context.num = i
    else:
        i = ex.tasks.last_tasks[-1].context.num
        for last_task in ex.tasks.last_tasks:
            task = ex.tasks.new(template=last_task, name='task ' + str(i),
submit=True)
            task.clear_dependencies()
            task.add_dependency(last_task)

            task.context.num = i
            i += 1
    return

t = Torii()

# step 0 - create the picom to edit
name = 'test'
```

```
version = '0.1'
picom = t.picoms.new(name=name, version=version)

# define the script
picom.set_script(picom_body)

# prepare the task template
application = t.applications.get(name='Test App 001')
template = t.tasks.new(name = 'test task', description = 'wait for a while',
application=application)
picom.templateTasks = [template]

# define the icon
picom.icon = ICON

# save the picom
picom.update()
```

There are two major elements in this example.

First, the script that is set using a function reference. The following section explains how the function is transformed in a script.

Second, we give a template task to the picom for the script to have a model for spawning new tasks.

### **Picom script and picom executor**

The code below correspond to the script generated by the client API in the former example. The client API add to the function that was passed as argument a 'main' body and the creation of a *picom executor* that mimics the behavior of a Torii client.

The idea is to have exactly the same syntax for the picom script than the one we would use in the client API. In particular the picom executor offers a pseudo-task service (*OfflineTaskService*) that allows the picom to spawn new tasks.

In fact the picom executor read the context, templates and picom descriptions from the runlog and automatically generate the json files of the tasks-to-spawned in the step directory.

Keep in mind that the picom execution is completely offline. The executor cannot get any data from Gateway except the data that was written in the runlog.

```
from torii import PicomExecutor

def picom_body(ex):
    """
    The function defining the content of the Picom
    :param ex: the picom executor
    :return:
    """
```

```
sleep = picom_task.get_input('sleep')
if ex.context.step > 5:
    ex.stop()

if not ex.tasks.last_tasks:
    for i in range(0, 1):
        task = ex.tasks.new(template='test task', name='task ' + str(i),
submit=True)
        task.set_input('sleep', sleep)
        task.context.num = i
    else:
        i = ex.tasks.last_tasks[-1].context.num
        for last_task in ex.tasks.last_tasks:
            task = ex.tasks.new(template=last_task, name='task ' + str(i),
submit=True)
            task.clear_dependencies()
            task.add_dependency(last_task)

            task.context.num = i
            i += 1
    return

if __name__ == "__main__":

    ex = PicomExecutor()
    picom_body(ex)
```

From:  
<http://docs.fujitsu.fr/HPCGateway/current/> - **HPC Gateway**

Permanent link:  
<http://docs.fujitsu.fr/HPCGateway/current/doku.php?id=fujitsu:hpcgateway:guides:internals:picoms>

Last update: **2018/03/26 14:13**

