MuG Tools API Documentation

Release 0.5

Marco Pasi

Nov 27, 2018

Contents:

CHAPTER 1

Requirements and Installation

1.1 Requirements

1.1.1 Software

- Python 2.7.10+

1.1.2 Python Modules

- pytest

1.2 Installation

Directly from GitHub:

```
git clone https://github.com/Multiscale-Genomics/mg-tool-api.git
```

Using pip:

```
pip install git+https://github.com/Multiscale-Genomics/mg-tool-api.git
```

1.3 Documentation

To build the documentation:

```
pip install Sphinx
pip install sphinx-autobuild
cd docs
make html
```

CHAPTER 2

Tools API

2.1 App Initiating and handling

class basic_modules.app.**App**

The generic App interface.

The App abstracts details of the particular local execution environment in order for Tools to run smoothly. For example, a subclass of App may exist that deals with execution of Tools in a Virtual Machine. Apps should be compatible with all Tools.

In general, App deals with:

1. instantiate and configure the Tool, and
2. call its "run" method

The App.launch() method is called in order to run a Tool within the App, with each call wrapping a single Tool class. The App.launch method calls the Tool.run() method; the App._pre_run() and App._post_run() methods should be called to execute operations before and after, in order to facilitate the accumulation of features in App subclasses in a way similar to the mixin pattern (see for example WorkflowApp).

As Apps need to be compatible with any Tool, it is unpractical to use Apps to combine Tools. Instead, Workflows can be implemented (see Workflow) in order to take advantage of the VRE's capabilities to optimise the data flow according to the specific requirements of the workflow, by ensuring that data is staged/unstaged only once.

This general interface outlines the App's workload, independent of the execution environment and runtime used (e.g. it does not rely on PyCOMPSs, see PyCOMPSsApp).

launch (*tool_class*, *input_files*, *input_metadata*, *output_files*, *configuration*)

Run a Tool with the specified inputs and configuration.

Parameters

- **tool_class** (*class*) – the subclass of Tool to be run;
- **input_files** (*dict*) – a dict of absolute path names of the input data elements required by the Tool, associated with their role;

- **input_metadata** (*dict*) – a dict of metadatas for each of the input data elements required by the Tool, associated with their role;
- **output_files** (*dict*) – a dict of absolute path names of the output data elements created by the Tool, associated with their role;
- **configuration** (*dict*) – a dictionary containing information on how the tool should be executed.

Returns

output_files [dict] a dict of absolute path names of the output data elements created by the Tool, associated with their role;

output_metadata [dict] a dict of metadatas for each of the output data elements created by the Tool, associated with their role.

Return type (output_files, output_metadata)

Example

```
>>> import App, Tool
>>> app = App()
>>> app.launch(Tool, {"input": <input_file>}, {})
```

2.2 Metadata Handling

class basic_modules.metadata.**Metadata**(*data_type=None*, *file_type=None*, *file_path=None*, *sources=None*, *meta_data=None*, *taxon_id=None*)

Object containing all information pertaining to a specific data element.

classmethod get_child(*parents*, *path*)

Generate a stub for the metadata of a new data element generated from the data element described in the specified parents.

Fields "data_type" and "file_type" are taken from the first parent; the "meta_data" fields are merged from all parents, in their respective order (i.e. values in the last parent prevail).

While making a copy, ensure the copy is deep enough that changing the child instance will not affect the parents.

Parameters parents (*list*) – List of Metadata instances

Returns An instance of Metadata generated as described above

Return type *Metadata*

Example

```
>>> import Metadata
>>> metadata1 = Metadata(...)
>>> metadata2 = Metadata(...)
>>> child_metadata =
>>>     Metadata.get_child([metadata1, metadata2], 'child_file')
```

2.3 Tool Definitions

class basic_modules.tool.**Tool**(*configuration=None*)

Abstract class describing a specific operation on a precise input data type to produce a precise output data type.

The tool is executed by calling its "run()" method, which should support multiple inputs and outputs. Inputs and outputs are valid file names locally accessible to the Tool.

The "run()" method also receives an instance of Metadata for each of the input data elements. It is the Tool's responsibility to generate the metadata for each of the output data elements, which are returned in a tuple (see code below).

The "run()" method calls the relevant methods that perform the operations require to implement the Tool's functionality. Each of these methods should be decorated using the "@task" decorator. Further, the task constraints can be configured using the "@constraint" decorator.

See also Workflow.

run(*input_files*, *input_metadata*, *output_files*)

Perform the required operations to achieve the functionality of the Tool. This usually involves: 0. Import tool-specific libraries 1. Perform relevant checks on input data 2. Optionally convert input data to internal formats 3. Perform tool-specific operations 4. Optionally convert output data to the output format 5. Write metadata for the output data 6. Handle failure in any of the above

In case of failure, the Tool should return None instead of the output file name, AND attach an Exception instance to the output metadata (see Metadata.set_exception), to allow the wrapping App to report the error (see App).

Note that this method calls the actual task(s). Ideally, each task should have a unique name that identifies the operation: these will be used by the COMPSs runtime to build a graph and trace.

Parameters

- **input_file** (*dict*) – a dict of absolute path names of the input data elements, associated with their role;
- **input_metadata** (*dict*) – a dict of metadatas for each of the input data elements, associated with their role;
- **output_files** (*dict*) – a dict of absolute path names of the output data elements, associated with their role.

Returns

output_files [dict] a dict of absolute path names of the output data elements created by the Tool, associated with their role;

output_metadata [dict] a dict of metadatas for each of the output data elements created by the Tool, associated with their role;

Return type (output_files, output_metadata)

Example

```
>>> import Tool
>>> tool = Tool(configuration = {})
>>> tool.run(
... {"input1": <input_1>, "input2": <input_2>},
... {"input1": <in_data_1>, "input2": <in_data_2>})
({"output": <output_1>}, {"output": <out_data_1>})
```

2.4 Workflow Definitions

class `basic_modules.workflow.`**`Workflow`**

Abstract class describing a Workflow.

Workflows are similar to Tools in that they are defined as receiving a precise input data type to produce a precise output data type. The main difference is that, instead of performing the operations themselves, they instantiate other Tools and call their “run()” method to define a flow of operations. Workflows can be further nested, to provide easy access to very complex pipelines. Furthermore, Workflows automatically take advantage of the VRE’s ability to optimise the data flow between operations: this is a powerful strategy to implement the most data intensive pipelines.

The Workflow itself is executed by calling its “run()” method; as for Tools, “run()” should support multiple inputs and outputs, which are assumed to be valid file names locally accessible to the Workflow. This allows the Workflow to use the output of Tools as input for other Tools.

The “run()” method of Workflows should keep track of these intermediate outputs by using the “add_intermediate()” method, to allow the wrapping App to unstage these (see App).

As for Tools, Workflows are expected to generate metadata for each of the outputs (as well as for intermediate outputs); generally the metadata generated by the Tools called by the Workflow will be sufficient.

run(*input_files*, *metadata*, *output_files*)

Perform the required operations to achieve the functionality of the Workflow. This usually involves: 0. Perform relevant checks on the input 1. Instantiate a Tool and run it using some input data 2. Add the Tool’s output to the intermediates 3. Repeat from 1 as many times as required 4. Optionally edit the output metadata 5. Return the output files and metadata

See also help(Tool.run).

CHAPTER 3

Utils

3.1 App Initiating and handling

`utils.logger.`**`critical`**(*message*, **args*, ***kwargs*)

Logs a message with level FATAL. The arguments are interpreted as for debug().

`utils.logger.`**`debug`**(*message*, **args*, ***kwargs*)

Logs a message with level DEBUG.

'message' is the message format string, and the args are the arguments which are merged into msg using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

`utils.logger.`**`error`**(*message*, **args*, ***kwargs*)

Logs a message with level ERROR. The arguments are interpreted as for debug().

`utils.logger.`**`fatal`**(*message*, **args*, ***kwargs*)

Logs a message with level FATAL. The arguments are interpreted as for debug().

`utils.logger.`**`info`**(*message*, **args*, ***kwargs*)

Logs a message with level INFO. The arguments are interpreted as for debug().

`utils.logger.`**`progress`**(*message*, **args*, ***kwargs*)

Provides information about Tool progress.

Logs a message containing information about Tool progress, with level `PROGRESS`.

The arguments are interpreted as for `debug()` (see below for exceptions).

This function provides two pre-baked log message formats, that can be activated by specifying the following items in `**kwargs`:

Parameters

- **`status`** (*`str`*) – Status of the Tool logs "MESSAGE - STATUS"
- **`task_id`** (*`int`*) – Current task; requires also the "total" item logs "MESSAGE (TASK_ID/TOTAL)

- **total** (*int*) – Total number of tasks, should be provided in conjunction with task_id logs “MESSAGE (TASK_ID/TOTAL)

Example

```
class TestTool(Tool):
    def run(self, input_files, input_metadata, output_files):
        logger.progress("TestTool starting", status="RUNNING")
        total_tasks = 3

        self.task1()
        logger.progress("TestTool", task_id=1, total=total_tasks)

        self.task2()
        logger.progress("TestTool", task_id=2, total=total_tasks)

        self.task3()
        logger.progress("TestTool", task_id=3, total=total_tasks)

        logger.progress("TestTool", status="DONE")
        return True
```

utils.logger.**warn**(*message*, **args*, ***kwargs*)
Logs a message with level WARNING. The arguments are interpreted as for debug().

utils.logger.**warning**(*message*, **args*, ***kwargs*)
Logs a message with level WARNING. The arguments are interpreted as for debug().

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index