signac project template Documentation

Release 0.2.0

Carl Simon Adorf, Paul Dodd

Mar 27, 2017

Contents

This is the documentation for the **signac-project-template** designed for rapid project development based on the data management framework signac and the workflow extension signac-flow.

Note: Before reading this manual you should be familiar with the basic concepts of signac.

Contents:

CHAPTER 1

Quickstart

This project is based on the basic workflow implemented in the signac tutorial. Being familiar with the tutorial will help in understanding the logic of this template.

The project requires the signac-flow package, which implements the core logic of the example workflow within a `flow.FlowProject` class. In addition it adds functionality to work with schedulers in a cluster environment.

The Basics

This is a list of key things you need to know in order to efficiently work with this project:

1. All modules are part of the `my_project` package located in the directory of the same name.
2. The project execution logic is implemented within the `project.MyProject` class.
3. All jobs are classified via `str`-labels with the `MyProject.classify()` method.
4. The *next operation* is identified via the `MyProject.next_operation()` method.
5. The project **status** may be examined by executing the `status` module.
6. Job-operations may be submitted to a scheduler via the `submit` module.
7. Python-based operations are implemented within the `scripts/operations.py` module.
8. Operations defined in the `scripts/operations.py` module can be executed directly via the `scripts/run.py` script.

A complete overview of all modules and functions an be found in the *API* chapter.

Step-by-step

This is a description on how to execute the complete workflow of this project.

Initialize the data space using a random number or string, e.g. your username:

```
$ python -m my_project.init $USER  # (or $ python my_project.init 42)
```

You can check the status of your project:

```
$ python -m my_project.status -d
Query scheduler...
Determine job stati...
Generate output...

Status project 'MyProject':
Total # of jobs: 10
label    progress
-------  ----------

Detailed view:
job_id                            S    next_op     labels
--------------------------------  ---  ----------  --------
6c57f630f0b62d449349ee2322cc16b6  U !  initialize
e0cf9aa968b48b22c66bbfda41d46129  U !  initialize
1677c153f81290d2e6e8b97a4f1d4297  U !  initialize
a230567b8a54d5c44d88b806b390b426  U !  initialize
3904431a51a3d3e4a31358f24b69d43f  U !  initialize
...

Abbreviations used:
!: requires_attention
S: status
U: unknown
```

We initialize the jobs for hoomd-blue:

```
$ python scripts/run.py initialize
```

Notice that the next_op and labels have changed if you check the status again:

```
$ python -m my_project.status -d
Query scheduler...
Determine job stati...
Generate output...

Status project 'MyProject':
Total # of jobs: 10
label        progress
-----------  ------------------------------------------------
initialized  |########################################| 100.00%

Detailed view:
job_id                            S    next_op    labels
--------------------------------  ---  ---------  -----------
6c57f630f0b62d449349ee2322cc16b6  U !  estimate   initialized
e0cf9aa968b48b22c66bbfda41d46129  U !  estimate   initialized
1677c153f81290d2e6e8b97a4f1d4297  U !  estimate   initialized
a230567b8a54d5c44d88b806b390b426  U !  estimate   initialized
3904431a51a3d3e4a31358f24b69d43f  U !  estimate   initialized
...

Abbreviations used:
!: requires_attention
```

```
S: status
U: unknown
```

Compute the ideal gas estimate, just like in the tutorial:

```
$ python scripts/run.py estimate
```

Execute a molecular dynamics simulation using hoomd-blue with:

```
$ python scripts/run.py sample 6c57
```

where *6c57* is the first few characters of the *job id*.

Note: When no *job id* is provided as argument, the specified operation is executed for **all** jobs.

Instead of running the operations directly, we can also submit them to a scheduler:

```
$ python -m my_project.submit -j sample
```

In this case we explicitly specified which operation to submit. If we omit the argument, the *next operation* for each job will be submitted.

Tip: Use the `--pretend` argument to print the submission script to the screen instead of submitting it during debugging.

The scheduler is determined from the environment with the `environment` module. If your environment does not have a scheduler or it is not configured, signac-flow will raise an exception. However, you can use a test environment with `--test` argument, which will mock an actual submission process.

CHAPTER 2

Reference

Introduction

A signac project manages a data space which is divided into segments, where each segment is strongly associated with a unique set of parameters: a *state point*. The signac-flow extension provides means to implement a workflow via the `flow.FlowProject` which inherits from `signac.Project`. This workflow is based on two core concepts: job *classification* and data space *operations*.

Classfication

We classify the state of a `Job` using text *labels*. These labels can be determined by a simple generator function, e.g.:

```
def classify(job):
   if job.isfile('init.txt'):
       yield 'initialized'
```

Operations

A *data space operation* is any action that will modify the data space.

This is an example for an operation implemented in python:

```
def initialize(job):
    with job:
        with open('init.txt', 'w') as file:
            file.write('Hello world!')
```

The *initialize* operation will create a file called `init.txt` within a `job`'s workspace.

The default workflow

Combining the concepts of *classification* and *operations* we can define the workflow logic of a `flow.FlowProject` by implementing the `classify()` and the `next_operation()` method:

```
from flow import FlowProject
from flow import JobOperation

class MyProject(FlowProject):

    def classify(self, job):
        if job.isfile('init.txt'):
            yield 'initialized'
        if job.isfile('dump.txt'):
            yield 'processed'

    def next_operation(self, job):
        labels = set(self.classify(job))

        def op(name):
            return JobOperation(name, job, 'python scripts/run.py {} {}'.format(name,␣
↪job))

        if 'initialized' not in labels:
            return op('initialize')
        if 'processed' not in labels:
            return op('process')
```

The `next_operation()` returns the **default operation** to execute **next** for a job in the identified state. This operation is a command, which can be executed on the command line. In the template, all operations are defined in the `scripts/operations.py` module and are executed by the `scripts/run.py` script.

We can get a quick overview of our project's status via the `print_status()` method:

```
>>> project = MyProject()
>>> project.print_status(detailed=True, params=('a',))
Status project 'MyProject':
Total # of jobs: 10
label        progress
-----------  ------------------------------------------------
initialized  |########------------------------------------| 20.00%
processed    |####----------------------------------------| 10.00%

Detailed view:
job_id                            S    next_op       a  labels
--------------------------------  ---  ----------  ---  ----------------------
108ef78ec381244447a108f931fe80db  U !  sample      1 1  processed, initialized
be01a9fd6b3044cf12c4a83ee9612f84  U !  process     3 2  initialized
32764c28ef130baefebeba76a158ac4e  U !  initialize  2.3
# ...
```

Tip: You can print the project's status from the command line by executing `$ python -m my_project.status`.

Running operations

All python-based *operations* are implemented in the `scripts/operations.py` module. We can use the `scripts/run.py` script to execute them directly, e.g.:

```
$ python scripts/run.py initialize 108e
```

This command will execute the *initialize* operation for the job identified by the *108e...* id.

Scheduling

To take full advantage of the workflow management, it is advantagous to use a `Scheduler` which schedules the execution of *job-operations* for us. The **project template** attempts to detect available schedulers through the `environment` module, but might require some tweaking based off your particular computing environment.

To submit job-operations to a scheduler, call the `submit()` method.

Tip: You can submit *job operations* to a scheduler from the command line, by executing `$ python my_project.submit`.

The `submit()` method will schedule the execution of operations for specified jobs by generating and submitting a *jobscript* to the scheduler.

Every *job submission script* has the same basic structure:

1. environment dependent header (e.g. scheduler options)
2. operation-agnostic header (e.g. switching into the project root directory)
3. commands to execute operations

The *scheduler header* will vary across different scheduler implementations and should be configured via the `environment` module.

In summary, if we only execute *operations* defined in the `operations` module, we can run them either directly or submit them to a scheduler:

```
python scripts/run.py OPERATION [JOBID] ...
python -m my_project.submit [-j OPERATION] [JOBID] ...
```

CHAPTER 3

API

Module contents

my_project.project module

my_project.init module

my_project.status module

my_project.submit module

my_project.environment module

my_project.switch_workspace module

CHAPTER 4

Indices and tables

- genindex
- modindex
- search