signac project template Documentation

Release 0.2.0

Carl Simon Adorf, Paul Dodd

February 18, 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. Data Space operations are implemented within the `scripts/operations.py` module.
3. The project execution logic is implemented within the `project.MyProject` class.
4. All jobs are classified via `str`-labels with the `MyProject.classify()` method.
5. The *next operation* is identified via the `MyProject.next_operation()` method.
6. Job-operations may be executed directly via the `scripts/run.py` script.
7. Job-operations may be submitted to a scheduler via the `submit` module.
8. The project **status** may be examined by executing the `status` module.

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

You can check the status of your project:

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

```
Status project 'MyProject':
Total # of jobs: 5
label        progress
-----------  ------------------------------------------------
initialized  |######################################| 100.00%

Detailed view:
job_id                            status    next_operation    labels
--------------------------------  --------  ----------------  -----------
8921709098d990fc70b19895653b7f01  unknown   estimate          initialized
8deb24c26dcb0bf0322cbf45c6b3198f  unknown   estimate          initialized
b76e21a18c46a90ed52ec3f1e2cd6250  unknown   estimate          initialized
ed41e3073b4a4133c05bf7ed050ebceb  unknown   estimate          initialized
fc89c69cb0f09b84f0b7f08c39bde326  unknown   estimate          initialized
```

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

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

Or execute a molecular dynamics simulation using hoomd-blue with:

```
$ python scripts/run.py equilibrate 8921
```

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 my_project.submit -j equilibrate
```

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

Note: 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 default to a *fake scheduler*, which prints the job scripts to screen.

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

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

The `next_operation()` returns the **default operation** to execute **next** for a job in the identified state.

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 *jobscript* has the same structure:

1. scheduler header
2. project header
3. operations

The *scheduler header* will vary across different scheduler implementations and can be configured via the `header` module. The *header* contains commands which should only be executed *once* per submission, such as setting up the correct software environment.

By default only those job-operations are submitted where the *operation* is equal to the *next operation*. This policy is implemented within the `eligible()` method. Think of it as *eligible for submission*. You can of course change the function to implement whatever policy you prefer.

In summary, we can execute *operations* defined in the `operations` module either directly or we can 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.header module

my_project.switch_workspace module

CHAPTER 4

Indices and tables

- genindex
- modindex
- search