simpleflow Documentation

Release 0.11.9

Greg Leclercq

August 05, 2016

Contents

Release v0.11.9. (*Installation*)

Python library for dataflow programming with Amazon SWF

CHAPTER 1

Guide

1.1 License

1.2 Installation

1.2.1 From the PyPI

```
$ pip install -U simpleflow
```

1.2.2 From Source

simpleflow is actively developed on Github.

You can clone the public repo:

```
git clone https://github.com/botify-labs/simpleflow.git
```

Or download one of the following:

- tarball
- zipball

Once you have the source, you can install it into your site-packages with

```
$ python setup.py install
```

1.3 Quickstart

TODO

1.4 SWF Quickstart

swf is a wrapper for Amazon Simple Workflow service. It aims to provide some abstractions over Boto library SWF API implementation, like querysets and objects over commonly used concepts: `Domains`, `Workflows`, `Activities`, and so on.

1.4.1 Settings

Mandatory:

- aws_access_key_id
- aws_secret_access_key

Optional:

- region

Settings are found respectively in:

A credential file a `.swf` file in the user's home directory:

```
[credentials]
aws_access_key_id=<aws_access_key_id>
aws_secret_access_key=<aws_secret_access_key>

[defaults]
region=us-east-1
```

The following environment variables

- *AWS_ACCESS_KEY_ID*
- *AWS_SECRET_ACCESS_KEY*
- *region*

If neither of the previous methods were used, you can still set the AWS credentials with `swf.settings.set()`:

```
>>> import swf.settings
>>> swf.settings.set(aws_access_key_id='MYAWSACCESSKEYID',
...                  aws_secret_access_key='MYAWSSECRETACCESSKEY',
...                  region='REGION')
# And then you're good to go...
>>> queryset = DomainQuery()
>>> queryset.all()
[Domain('test1'), Domain('test2')]
```

1.4.2 Batteries Included

Models

Simple Workflow entities such as domains, workflow types, workflow executions and activity types are to be manipulated through swf using `models`. They are immutable `swf` objects representations providing an interface to objects attributes, local/remote objects synchronization and changes watch between these local and remote objects.

```
# Models resides in swf.models module
>>> from swf.models import Domain, WorkflowType, WorkflowExecution, ActivityType

# Once imported you're ready to create a local model instance
>>> D = Domain(
    "my-test-domain-name",
    description="my-test-domain-description",
    retention_period=60
)

# a Domain model local instance has been created, but nothing has been
# sent to amazon. To do so, you have to save it.
>>> D.save()
```

Now you have a local `Domain` model object, and if no errors were raised, the `save` method have saved amazon-side. But, sometimes, you won't be able to know if the model you're manipulating has an upstream version: whether you've acquired it through a queryset, or the remote object has been deleted for example. Fortunately, models are shipped with a set of functions to make sure your local objects keep synced and consistent.

```
# Exists method let's you know if you're model instance has an upstream version
>>> D.exists
True

# What if changes have been made to the remote object?
# synced  and changes methods help ensuring local and remote models
#are still synced and which changes have been maid.
>>> D.is_synced
True
>>> D.changes
ModelDiff()
```

What if your local object is out of sync? Models `upstream` method will fetch the remote version of your object and will build a new model instance using it's attributes.

```
>>> D.is_synced
False
>>> D.changes
ModelDiff(
    Difference('status', 'REGISTERED', 'DEPRECATED')
)

# Let's pull the upstream version
>>> D = D.upstream()
>>> D.is_synced
True
>>> D.changes
ModelDiff()
```

QuerySets

Models can be retrieved and instantiated via querysets. To continue over the django comparison, they're behaving like django managers.

```
# As querying for models needs a valid connection to amazon service,
# Queryset objects cannot act as classmethods proxy and have to be instantiated;
# most of the time against a Domain model instance
>>> from swf.querysets import DomainQuerySet, WorkflowTypeQuerySet

# Domain querysets can be instantiated directly
>>> domain_qs = DomainQuerySet()
>>> workflow_domain = domain_qs.get("MyTestDomain")  # and specific model retieved via .get method
>>> workflow_qs = WorkflowTypeQuerySet(workflow_domain)  # queryset built against model instance exa

>>> workflow_qs.all()
[WorkflowType("TestType1"), WorkflowType("TestType2"),]

>>> workflow_qs.filter(status=DEPRECATED)
[WorkflowType("DeprecatedType1"),]
```

Events

(coming soon)

History

(coming soon)

Decisions

(coming soon)

Actors

Swf workflows are based on a worker-decider pattern. Every actions in the flow is executed by a worker which runs supplied activity tasks. And every actions is the result of a decision taken by the decider reading the workflow events history and deciding what to do next. In order to ease the development of such workers and decider, swf exposes base classes for them located in `swf.actors` submodule.

- An `Actor` must basically implement a `start` and `stop` method and can actually inherits from whatever runtime implementation you need: thread, gevent, multiprocess...

```
class Actor(ConnectedSWFObject):
    def __init__(self, domain, task_list)
    def start(self):
    def stop(self):
```

- `Decider` base class implements the core functionality of a swf decider: polling for decisions tasks, and sending back a decision task copleted decision. Every other special needs implementations are left up to the user.

```
class Decider(Actor):
    def __init__(self, domain, task_list)
```

```
    def complete(self, task_token, decisions=None, execution_context=None)
    def poll(self, task_list=None, identity=None, maximum_page_size=None)
```

- `Worker` base class implements the core functionality of a swf worker whoes role is to process activity tasks. It is basically able to poll for new activity tasks to process, send back a heartbeat to swf service in order to let it know it hasn't failed or crashed, and to complete, fail or cancel the activity task it's processing.

```
class ActivityWorker(Actor):
    def __init__(self, domain, task_list)
    def cancel(self, task_token, details=None)
    def complete(self, task_token, result=None)
    def fail(self, task_token, details=None, reason=None)
    def heartbeat(self, task_token, details=None)
    def poll(self, task_list=None, **kwargs)
```

1.5 Execution of Tasks as Programs

1.5.1 Introduction

The `simpleflow.execute` module allows to define functions that will be executed as a program.

There are two modes:

- Convert the definition of a fonction as a command line.
- Execute a Python function in another process.

Please refer to the `simpleflow.tests.test_activity` test module for further examples.

1.5.2 Executing a function as a command line

Let's take the example of `ls`:

```
@execute.program()
def ls():
    pass
```

Calling `ls()` in Python will execute the `ls` command. Here the purpose of the function definition is only to describe the command line. The reason for this is to map a call in a workflow definition to a program to execute on the command line. The program may be written in any language whereas the workflow definition is in Python.

1.5.3 Executing a Python function in another process

The rationale for this feature is to execute a function with another interpreter (such as pypy) or in another environment (virtualenv).

```
@execute.python(interpreter='pypy')
def inc(xs):
    return [x + 1 for x in xs]
```

Calling `inc(range(10))` in Python will execute the function with the `pypy` interpreter found in the `$PATH`.

1.5.4 Limitations

The main limitation comes from the need to serialize the arguments and the return values to pass them as strings. Hence all arguments and return values must be convertible into JSON values.

CHAPTER 2

API Reference

CHAPTER 3

Project info

3.1 Contributing guidelines

3.1.1 In General

- PEP 8, when sensible.
- Test ruthlessly. Write docs for new features.
- Even more important than Test-Driven Development–*Human-Driven Development*.

3.1.2 In Particular

Questions, Feature Requests, Bug Reports, and Feedback. . .

. . .should all be reported on the Github Issue Tracker .

Setting Up for Local Development

1. Fork simpleflow on Github.
2. Clone your fork:

```
$ git clone git@github.com/botify-labs/simpleflow.git
```

3. Make your virtualenv and install dependencies. If you have virtualenv and virtualenvwrapper, run:

```
$ mkvirtualenv simpleflow
$ cd simpleflow
$ pip install -r requirements-dev.txt
```

- If you don't have virtualenv and virtualenvwrapper, you can install both using virtualenv-burrito.

Git Branch Structure

simpleflow used to have a separated `devel` branch but is now using only one, main branch `master`, that contains what will be released in the next version. This branch is (hopefully) always stable.

Pull Requests

1. Create a new local branch.

```
$ git checkout -b name-of-feature
```

2. Commit your changes. Write good commit messages.

   $ git commit -m "Detailed commit message" $ git push origin name-of-feature

2. Before submitting a pull request, check the following:

- If the pull request adds functionality, it should be tested and the docs should be updated.
- The pull request should work on Python 2.7 and PyPy. Use `tox` to verify that it does.

3. Submit a pull request to the `master` branch.

Running tests

To run all the tests in your current virtual environment:

```
$ ./script/test
```

This is what Travis CI does on each environment.

If you want to simulate what Travis CI does, you can approach that by running a container from them:

```
$ ./script/test-travis python2.7
$ ./script/test-travis pypy
```

This can help you simulate locally what Travis CI would do. Be aware though that tests may fail depending on your OS, so Travis CI is the reference gate for the project. For instance, installing `subprocess32` in a Pypy environment doesn't work on Mac OSX.

Documentation

Contributions to the documentation are welcome. Documentation is written in reStructured Text (rST). A quick rST reference can be found here. Builds are powered by Sphinx.

To build docs:

```
$ invoke build_docs -b
```

The `-b` (for "browse") automatically opens up the docs in your browser after building.

3.2 Credits

3.2.1 Development Lead

- Greg Leclercq <greg@botify.com>

3.2.2 Contributors

None yet. Why not be the first?

3.3 Changelog

3.3.1 0.11.9

- Add a –repair option to simpleflow standalone (#100)

3.3.2 0.11.8

- Retry boto.swf connection to avoid frequent errors when using IAM roles (#99)

3.3.3 0.11.7

Same as 0.11.6 but the 0.11.6 on pypi is broken (pushed something similar to 0.11.5 by mistake)

3.3.4 0.11.6

- Add **issubclass_** method (#96)
- Avoid duplicate logs if root logger has an handler (#97)
- Allow passing SWF domain via the SWF_DOMAIN environment variable (#98)

3.3.5 0.11.5

- Don't mask activity cancel exception (#84)
- Propagate all decision response attributes up to Executor.replay() (#76, #94)

3.3.6 0.11.4

- ISO dates in workflow history #91
- Fix potential infinite retry loop #90

3.3.7 0.11.3

- Fix replay hooks introduced in 0.11.2 (#86)
- Remove python3 compatibility from README (which was not working for a long time)

3.3.8 0.11.2

- Add new workflow hooks (#79)

3.3.9 0.11.1

- Fix logging when an exception occurs

3.3.10 0.11.0

- Merge `swf` package into simplefow for easier maintenance.

3.3.11 0.10.4 and below

Sorry changes were not documented for simpleflow <= 0.10.x.