Python Finite State Machine

Release 2.0.0

Sep 10, 2018

Contents

CHAPTER 1

Overview

docs	
tests	
package	

Minimal state machine

- Free software: BSD license

```
import fsm

class MyTasks(fsm.FiniteStateMachineMixin):
    """An example to test the state machine.

    Contains transitions to everywhere, nowhere and specific states.
    """

    state_machine = {
        'created': '__all__',
        'pending': ('running',),
        'running': ('success', 'failed'),
        'success': None,
        'failed': ('retry',),
        'retry': ('pending', 'retry'),
    }

    def __init__(self, state):
        """Initialize setting a state."""
        self.state = state
```

(continues on next page)

(continued from previous page)

```
    def on_before_pending(self):
        print("I'm going to a pending state")
```

```
In [4]: m = MyTasks(state='created')

In [5]: m.change_state('pending')
I'm going to a pending state
Out[5]: 'pending'
```

```
In [6]: m.change_state('failed')  # Let's try to transition to an invalid state
---------------------------------------------------------------------------
InvalidTransition                         Traceback (most recent call last)
<ipython-input-6-71d2461eee74> in <module>()
----> 1 m.change_state('failed')

~/pyfsm/src/fsm/fsm.py in change_state(self, next_state, **kwargs)
     90             msg = "The transition from {0} to {1} is not valid".
↪format(previous_state,
     91                                                                    next_
↪state)
---> 92             raise InvalidTransition(msg)
     93
     94         name = 'pre_{0}'.format(next_state)

InvalidTransition: The transition from pending to failed is not valid
```

Contents

1.1 Installation

```
pip install fsmpy
```

1.2 Usage

1. Define in a class the `state_machine`
2. Initialize `state`, either with a value, using `__init__` or as a django field
3. Add hooks:

Method	Description
on_before_change_state	Before transitioning to the state
on_change_state	After transitioning to the state, if no failure, runs for every state
pre_<state_name>	Runs before a particular state, where `state_name` is the specified name in the `state_machine`
post_<state_name>	Runs after a particular state, where `state_name` is the specified name in the `state_machine`

This hooks will receive any extra argument given to `change_state`

E.g:

Running `m.change_state('pending', name='john')` will trigger `pre_pending(name='john')`

1.3 Django integration

```
import fsm
from django.db import models


class MyModel(models.Model, fsm.FiniteStateMachineMixin):
    """An example to test the state machine.

    Contains transitions to everywhere, nowhere and specific states.
    """

    CHOICES = (
        ('created', 'CREATED'),
        ('pending', 'PENDING'),
        ('running', 'RUNNING'),
        ('success', 'SUCCESS'),
        ('failed', 'FAILED'),
        ('retry', 'RETRY'),
    )

    state_machine = {
        'created': '__all__',
        'pending': ('running',),
        'running': ('success', 'failed'),
        'success': None,
        'failed': ('retry',),
        'retry': ('pending', 'retry'),
    }

    state = models.CharField(max_length=30, choices=CHOICES, default='created')

    def on_change_state(self, previous_state, next_state, **kwargs):
        self.save()
```

1.3.1 Django Rest Framework

If you are using `serializers`, they usually perform the `save`, so saving inside `on_change_state` is not necessary.

One simple solution is to do this:

```
class MySerializer(serializers.ModelSerializer):

    def update(self, instance, validated_data):
        new_state = validated_data.get('state', instance.state)
        try:
            instance.change_state(new_state)
        except fsm.InvalidTransition:
            raise serializers.ValidationError("Invalid transition")
        instance = super().update(instance, validated_data)
        return instance
```

1.4 Documentation

https://pyfsm.readthedocs.org/

1.5 Development

To run the tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER 2

Installation

At the command line:

```
pip install fsmpy
```

CHAPTER 3

Usage

1. Define in a class the `state_machine`
2. Initialize `state`, either with a value, using `__init__` or as a django field
3. Add hooks:

Method	Description
on_before_change_state	Before transitioning to the state
on_change_state	After transitioning to the state, if no failure, runs for every state
pre_<state_name>	Runs before a particular state, where `state_name` is the specified name in the `state_machine`
post_<state_name>	Runs after a particular state, where `state_name` is the specified name in the `state_machine`

This hooks will receive any extra argument given to `change_state`

E.g:

Running `m.change_state('pending', name='john')` will trigger `pre_pending(name='john')`

3.1 In your code

To use Python Finite State Machine in a project:

```
import fsm
```

Then add the Mixin to any class where the state machine is required.

```
class Foo(fsm.FiniteStateMachineMixin):

    state_machine = {
        'my_first_state': '__all__',
```

(continues on next page)

(continued from previous page)

```
        'my_state': ('my_second_state',),
        'my_second_state': ('my_state', 'my_second_state', 'last_state'),
        'last_state': None
    }

    state = 'my_first_state'
```

Instanciate the class and use it. Remember that in order to work as intended, `change_state` must be used to transition from one state to the other.

```
>>> foo = Foo()

>>> foo.current_state()
'my_first_state'

>>> foo.change_state('my_state')
'my_state'

>>> foo.current_state()
'my_state'

>>> foo.can_change('last_state')
False

>>> foo.get_valid_transitions()
('my_second_state',)
```

You can also use `BaseFiniteStateMachineMixin` for more flexibility. Implementing `current_state` and `set_state` is required. Doing this allows using more complex behavior, but it is **not recommended**.

CHAPTER 4

Django integration

```python
import fsm
from django.db import models


class MyModel(models.Model, fsm.FiniteStateMachineMixin):
    """An example to test the state machine.

    Contains transitions to everywhere, nowhere and specific states.
    """

    CHOICES = (
        ('created', 'CREATED'),
        ('pending', 'PENDING'),
        ('running', 'RUNNING'),
        ('success', 'SUCCESS'),
        ('failed', 'FAILED'),
        ('retry', 'RETRY'),
    )

    state_machine = {
        'created': '__all__',
        'pending': ('running',),
        'running': ('success', 'failed'),
        'success': None,
        'failed': ('retry',),
        'retry': ('pending', 'retry'),
    }

    state = models.CharField(max_length=30, choices=CHOICES, default='created')

    def on_change_state(self, previous_state, next_state, **kwargs):
        self.save()
```

4.1 Django Rest Framework

If you are using `serializers`, they usually perform the `save`, so saving inside `on_change_state` is not necessary.

One simple solution is to do this:

```
class MySerializer(serializers.ModelSerializer):

    def update(self, instance, validated_data):
        new_state = validated_data.get('state', instance.state)
        try:
            instance.change_state(new_state)
        except fsm.InvalidTransition:
            raise serializers.ValidationError("Invalid transition")
        instance = super().update(instance, validated_data)
        return instance
```

CHAPTER 5

Reference

5.1 fsm package

5.1.1 Submodules

5.1.2 fsm module

class fsm.**InvalidTransition**
Bases: exceptions.Exception

Moving from an state to another is not possible.

class fsm.**FiniteStateMachineMixin**
Bases: fsm.fsm.BaseFiniteStateMachineMixin

A drop in implementation. Ready to be used.

Replace FIELD_NAME in order to automatically retrieve or set from a different field.

In order to use with django, just add a field state or as defined in FIELD_NAME and remember to use change_state instead of simply assigning it

FIELD_NAME = 'state'

current_state()

set_state(*state*)

class fsm.**BaseFiniteStateMachineMixin**
Base Mixin to add a state_machine behavior.

Represents the state machine for the object.

The states and transitions should be specified in the following way:

```
state_machine = {
    'some_state': '__all__'
    'another_state': ('some_state', 'one_more_state')
    'one_more_state': None
}
```

Requires the implementation of `current_state` and `set_state`

can_change(*next_state*)
Validates if the next_state can be executed or not.

It uses the state_machine attribute in the class.

change_state(*next_state*, ***kwargs*)
Performs a transition from current state to the given next state if possible.

Callbacks will be exacuted before an after changing the state. Specific state callbacks will also be called if they are implemented in the subclass.

Parameters **next_state** (*str or int*) – where the state must go

Returns new state.

Return type str or int

Raises ***InvalidTransition*** – If transitioning is not possible

current_state()
Returns the current state in the FSM.

get_valid_transitions()
Return possible states to whom a product can transition.

Returns valid transitions

Return type tuple or list

on_before_change_state(*previous_state*, *next_state*, ***kwargs*)
Called before a state changes.

Parameters

- **previous_state** (*str or int*) – type depends on the definition of the states.
- **next_state** (*str or int*) – type depends on the definition of the states.

on_change_state(*previous_state*, *next_state*, ***kwargs*)
Called after a state changes.

Parameters

- **previous_state** (*str or int*) – type depends on the definition of the states.
- **next_state** (*str or int*) – type depends on the definition of the states.

set_state(*state*)
Update the internal state field.

Parameters **state** (*str or int*) – type depends on the definition of the states.

state_machine = None

5.1.3 Module contents

Initializing package.

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

6.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.2 Documentation improvements

Python Finite State Machine could always use more documentation, whether as part of the official Python Finite State Machine docs, in docstrings, or even on the web in blog posts, articles, and such.

6.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/Woile/pyfsm/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

6.4 Development

To set up *pyfsm* for local development:

1. Fork pyfsm (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/pyfsm.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

6.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

6.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 7

Authors

- Santiago Fraire Willemoes - https://github.com/woile

CHAPTER 8

Changelog

8.1 2.0.0 (2018-08-26)

- BREAKING: Simpler implementation
- Docs and README updated
- Usage of pyproject.toml

8.2 1.0.0 (2018-07-03)

- Renamed hooks to `pre_<state_name>` and `post_<state_name>`

8.3 0.1.3 (2017-15-09)

- Updated docs
- Corrections to code
- ci updated

8.4 0.1.0 (2016-04-18)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index