django-transitions Documentation

Release 0.1

Christian Ledermann

Jan 17, 2019

Contents:

CHAPTER 1

Overview

A wrapper of pytransitions for django You do not *need* django-transitions to integrate django and pytransitions. It is meant to be a lightweight wrapper (it has just over 50 logical lines of code) and documentation how to go about using pytransitions inside a django application.

This package provides:

- Example workflow implementation.
- **Base classes and mixins to**
    - Keep it DRY
    - Keep transitions consistent
    - Reduce cut and paste
    - Avoid boiler plate.
- Admin mixin to add workflow actions to the django admin.
- Admin templates

CHAPTER 2

Quickstart

Lets implement the following state machine.

- The object starts of as ‘under development’ which can then be made ‘live’.
- From the ‘live’ state it can be marked as ‘under maintenance’.
- From all states the object can be marked as ‘deleted’.
- A ‘deleted’ object can be recovered into the ‘under maintenance’ state.
- Whenever a transition occurs the datetime will be recorded in a datefield.

Import the dependencies:

```
from django_transitions.workflow import StateMachineMixinBase
from django_transitions.workflow import StatusBase
from transitions import Machine
```

2.1 States and Transitions

We start by defining the states and transitions

```
class LiveStatus(StatusBase):
    """Workflow for Lifecycle."""

    # Define the states as constants
    DEVELOP = 'develop'
    LIVE = 'live'
    MAINTENANCE = 'maintenance'
    DELETED = 'deleted'

    # Give the states a human readable label
```

(continues on next page)

(continued from previous page)

```
    STATE_CHOICES = (
        (DEVELOP, 'Under Development'),
        (LIVE, 'Live'),
        (MAINTENANCE, 'Under Maintenance'),
        (DELETED, 'Deleted'),
    )

    # Define the transitions as constants
    PUBLISH = 'publish'
    MAKE_PRIVATE = 'make_private'
    MARK_DELETED = 'mark_deleted'
    REVERT_DELETED = 'revert_delete'

    # Give the transitions a human readable label and css class
    # which will be used in the django admin
    TRANSITION_LABELS = {
        PUBLISH : {'label': 'Make live', 'cssclass': 'default'},
        MAKE_PRIVATE: {'label': 'Under maintenance'},
        MARK_DELETED: {'label': 'Mark as deleted', 'cssclass': 'deletelink'},
        REVERT_DELETED: {'label': 'Revert Delete', 'cssclass': 'default'},
    }

    # Construct the values to pass to the state machine constructor

    # The states of the machine
    SM_STATES = [
        DEVELOP, LIVE, MAINTENANCE, DELETED,
    ]

    # The machines initial state
    SM_INITIAL_STATE = DEVELOP

    # The transititions as a list of dictionaries
    SM_TRANSITIONS = [
        # trigger, source, destination
        {
            'trigger': PUBLISH,
            'source': [DEVELOP, MAINTENANCE],
            'dest': LIVE,
        },
        {
            'trigger': MAKE_PRIVATE,
            'source': LIVE,
            'dest': MAINTENANCE,
        },
        {
            'trigger': MARK_DELETED,
            'source': [
                DEVELOP, LIVE, MAINTENANCE,
            ],
            'dest': DELETED,
        },
        {
            'trigger': REVERT_DELETED,
            'source':  DELETED,
            'dest': MAINTENANCE,
        },
```

(continues on next page)

(continued from previous page)

```
    ]
```

2.2 Statemachine Mixin

Next we create a mixin to create a state machine for the django model.

Note: The mixin or the model *must* provide a state property. In this implementation state is mapped to the django model field `wf_state`

The mixin **must** override the `machine` of the `StateMachineMixinBase` class. The minimum boilerplate to achieve this is:

```
machine = Machine(
    model=None,
    **status_class.get_kwargs()
)
```

In the example we also define a `wf_finalize` method that will set the date when the last transition occurred on every transaction.

```
class LifecycleStateMachineMixin(StateMachineMixinBase):
    """Lifecycle workflow state machine."""

    status_class = LiveStatus

    machine = Machine(
        model=None,
        finalize_event='wf_finalize',
        auto_transitions=False,
        **status_class.get_kwargs()  # noqa: C815
    )

    @property
    def state(self):
        """Get the items workflowstate or the initial state if none is set."""
        if self.wf_state:
            return self.wf_state
        return self.machine.initial

    @state.setter
    def state(self, value):
        """Set the items workflow state."""
        self.wf_state = value
        return self.wf_state

    def wf_finalize(self, *args, **kwargs):
        """Run this on all transitions."""
        self.wf_date = timezone.now()
```

2.3 Model

Set up the django model

```
class Lifecycle(LifecycleStateMachineMixin, models.Model):
    """
    A model that provides workflow state and workflow date fields.

    This is a minimal example implementation.
    """

    class Meta:  # noqa: D106
        abstract = False

    wf_state = models.CharField(
        verbose_name = 'Workflow Status',
        null=False,
        blank=False,
        default=LiveStatus.SM_INITIAL_STATE,
        choices=LiveStatus.STATE_CHOICES,
        max_length=32,
        help_text='Workflow state',
    )

    wf_date =  models.DateTimeField(
        verbose_name = 'Workflow Date',
        null=False,
        blank=False,
        default=timezone.now,
        help_text='Indicates when this workflowstate was entered.',
    )
```

We can now inspect the behaviour of the model model with `python manage.py shell`

```
>>> from testapp.models import Lifecycle
>>> lcycle = Lifecycle()
>>> lcycle.state
'develop'
>>> lcycle.publish()
True
>>> lcycle.state
'live'
>>> lcycle.publish()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/christian/devel/django-transitions/.venv/lib/python3.5/site-packages/
↪transitions/core.py", line 383, in trigger
    return self.machine._process(func)
  File "/home/christian/devel/django-transitions/.venv/lib/python3.5/site-packages/
↪transitions/core.py", line 1047, in _process
    return trigger()
  File "/home/christian/devel/django-transitions/.venv/lib/python3.5/site-packages/
↪transitions/core.py", line 397, in _trigger
    raise MachineError(msg)
transitions.core.MachineError: "Can't trigger event publish from state live!"
>>> lcycle.save()
>>> graph = lcycle.get_wf_graph()
```

(continues on next page)

(continued from previous page)

```
>>> graph.draw('lifcycle_state_diagram.svg', prog='dot')  # This produces the above␣
↪diagram
```

2.4 Admin

Set up the django admin to include the workflow actions.

```
# -*- coding: utf-8 -*-
"""Example django admin."""

from django_transitions.admin import WorkflowAdminMixin
from django.contrib import admin

from .models import Lifecycle


class LifecycleAdmin(WorkflowAdminMixin, admin.ModelAdmin):
    """
    Minimal Admin for Lifecycles Example.

    You probably want to make the workflow fields
    read only so yo can not change these values
    manually.

    readonly_fields = ['wf_state', 'wf_date']
    """

    list_display = ['wf_date', 'wf_state']
    list_filter = ['wf_state']


admin.site.register(Lifecycle, LifecycleAdmin)
```

CHAPTER 3

Mixins and Base Classes

3.1 Transition Base Classes

Mixins for transition workflows.

3.1.1 StatusBase

class `django_transitions.workflow.`**`StatusBase`**

Base class for transitions and status definitions.

classmethod get_kwargs`()`

Get the kwargs to initialize the state machine.

3.1.2 StateMachineMixinBase

class `django_transitions.workflow.`**`StateMachineMixinBase`**

Base class for state machine mixins.

Class attributes:

- `status_class` must provide `TRANSITION_LABELS` property and the `get_kwargs` class method (see `StatusBase`).
- `machine` is a transition machine e.g:

```
machine = Machine(
    model=None,
    finalize_event='wf_finalize',
    auto_transitions=False,
    **status_class.get_kwargs()  # noqa: C815
)
```

The transition events of the machine will be added as methods to the mixin.

`get_available_events``()`

Get available workflow transition events for the current state.

Returns a dictionary:

- `transition`: transition event.
- `label`: human readable label for the event
- `cssclass`: css class that will be applied to the button

`get_wf_graph``()`

Get the graph for this machine.

3.2 Django Admin Mixins

Mixins for the django admin.

`class` `django_transitions.admin.`**`WorkflowAdminMixin`**

A mixin to provide workflow transition actions.

It will create an admin log entry.

`response_change``(`*request*, *obj*`)`

Add actions for the workflow events.

CHAPTER 4

Templates

To use the templates you have to include `'django_transitions'` in `INSTALLED_APPS` in the projects `settings.py` file:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    ...
    'django_transitions', # this is only needed to find the templates.
]
```

The `change_form` template adds workflow buttons to the admin change form, and also provides the 'save' and 'delete' buttons. This template can be applied to the django admin class:

```
change_form_template = 'transitions/change_form.html'
```

```
{% extends 'admin/change_form.html' %}

{% block submit_buttons_bottom %}

  <!-- add the save and delete buttons -->
  {{ block.super }}

  <!-- Add buttons for available transitions -->
  <div class="submit-row">
    {% for event in original.get_available_events %}
      <input type="submit" class="{{ event.cssclass }}" value="{{ event.label }}"␣
↪name="_{{ event.transition.name }}">
    {% endfor %}
  </div>

{% endblock %}
```

The `read_only_change_form` template adds workflow buttons to the admin change form, and removes the 'save' and 'delete' buttons. This template can be applied to the django admin class:

```
change_form_template = 'transitions/read_only_change_form.html'
```

```
{% extends 'admin/change_form.html' %}

{% block submit_buttons_bottom %}

  <div class="submit-row">
    {% for event in original.get_available_events %}
      <input type="submit" class="{{ event.cssclass }}" value="{{ event.label }}"␣
↪name="_{{ event.transition.name }}">
    {% endfor %}
  </div>

{% endblock %}
```

CHAPTER 5

Frequently asked questions

5.1 What are the advantages of django-transitions over other django workflow applications?

Personally I like to have all the information about my workflow in one place.

5.2 Are there other packages that provide this functionality?

The packages I know of are (in no specific order):

- django-fsm
- viewflow
- ActivFlow
- Django-XWorkflows
- Django River

You should evaluate if one of the above packages are a better match for your needs.

5.3 What is the history of django and pytransitions integration?

The code from this package was lifted from the discussion in django and transitions

CHAPTER 6

Changelog

6.1 0.2 (2019/01/17)

- Add optional css class to `TRANSITION_LABELS`

6.2 0.1 (2018/11/13)

- Initial release

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index