django-xworkflows Documentation

Release 0.11.0

Raphaël Barrois

February 25, 2017

Contents

django-xworkflows is a django application adding xworkflows functionnalities to django models.

CHAPTER 1

Getting started

First, install the required packages:

```
pip install django-xworkflows
```

In your `settings.py`, add `django_xworkflows` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    '...',
    'django_xworkflows',
)
```

Define a workflow:

```
from django_xworkflows import models as xwf_models

class MyWorkflow(xwf_models.Workflow):
    log_model = ''  # Disable logging to database

    states = (
        ('new', _(u"New")),
        ('old', _(u"Old")),
    )
    transitions = (
        ('get_old', 'new', 'old'),
    )
    initial_state = 'new'
```

And add it to a model:

```
from django import models
from django_xworkflows import models as xwf_models

class MyModel(xwf_models.WorkflowEnabled, models.Model):

    state = xwf_models.StateField(MyWorkflow)
```

The `state` field of `MyModel` is now defined as a `django.db.models.CharField`, whose `choices` and `default` are configured according to the related *`django_xworkflows.models.Workflow`*.

CHAPTER 2

Integration with django

After each successful transition, a `save()` is performed on the object. This behaviour is controlled by passing the extra argument `save=False` when calling the transition method.

If the *Workflow* has a definition for the *log_model* attribute (as a `<app>.<Model>` string), an instance of that model will be created for each successful transition.

If the `django_xworkflows.xworkflow_log` application is installed, *log_model* defaults to *TransitionLog*. Otherwise, it defaults to `''` (db logging disabled).

This behaviour can be altered by:

- Setting the *log_model* attribute to `''`
- Calling the transition method with `log=False` (no logging to database)
- Overriding the *db_log()* method of the *Workflow*.
- Overriding the *log_transition()* method of the *Workflow*; this controls both `log` and `save` behaviours.

CHAPTER 3

Contents

Library internals

This page documents the internal mechanisms of django_xworkflows.

Binding to a workflow

The mechanism to bind a django model to a `xworkflows.Workflow` relies on the *`WorkflowEnabled`* and *`StateField`* classes.

class `django_xworkflows.models.`**`StateField`**(*django.db.models.Field*)
This class is a simple Django `Field`, specifically tuned for a *`Workflow`*.

It is internally backed by a `CharField` containing the `name` of the state.

Reading the value always returns a `xworkflows.base.StateWrapper`, writing checks that the value is a valid state or a valid state name.

`workflow`
Mandatory; holds the *`Workflow`* to which this *`StateField`* relates

`choices`
The workflow states, as a list of `(name, title)` tuples, for use in forms.

`default`
The name of the inital state of the workflow

`max_length`
The length of the longest state name in the workflow.

`blank`
Such a field cannot be blanked (otherwise, the workflow wouldn't have a meaning).

`null`
Since the field cannot be empty, is cannot be null either.

`south_field_triple`(*self*)
Returns the south description of this field. When unfreezing, a fake *`Workflow`* will be retrieved with the same states and initial_state as present at freezing time.

This allows reading states that no longer exist in the workflow.

class `django_xworkflows.models.`**`WorkflowEnabled`**(*models.Model*)
This class inherits from Django's `Model` class, performing some transformations on the subclass: each `attr`

`= StateField(SomeWorkflow, ...)` attribute will enable XWorkflows' transition detection and wrapping.

Most of this job is performed through *WorkflowEnabledMeta*.

_get_FIELD_display(*self*, *field*)
This method overrides the default django one to retrieve the `title` from a *StateField* field.

Transitions

Transitions mostly follow XWorkflows' mechanism.

Implementation wrappers

django_xworkflows provides two custom implementation wrappers specially suited for Django:

class `django_xworkflows.models.`**DjangoImplementationWrapper**(*xworkflows.base.ImplementationWrapper*)
This wrapper simply adds two special attributes for interpretation in Django templates:

alters_data
Set to `True` to prevent Django templating system to call a transition, e.g in `{{ foo.confirm }}`

do_not_call_in_templates
This attribute signals Django templating system (starting from Django 1.4) that the transition implementation should not be called, but its attributes should be made available.

This allows such constructs:

```
{% if obj.confirm.is_available %}
<form method="POST" action="">
    <input type="submit" value="Confirm" />
</form>
{% endif %}
```

class `django_xworkflows.models.`**TransactionalImplementationWrapper**(*DjangoImplementationWrapper*)
This specific wrapper runs all transition-related code, including `hooks`, in a single database transaction.

The *TransactionalImplementationWrapper* can be enabled by setting it to the `implementation_class` attribute of a `xworkflows.Workflow` or of a *Workflow*:

```
class MyWorkflow(models.Workflow):
    implementation_class = models.TransactionalImplementationWrapper
```

Workflow and logging

class `django_xworkflows.models.`**Workflow**(*xworkflows.Workflow*)
This `xworkflows.Workflow` subclass performs a few customization:

- Logging transition logs in database
- Saving updated objects after the transition

log_model
This holds the name of the model to use to log to the database. If empty, no database logging is performed.

log_model_class
This holds the class of the model to use to log to the database.

Takes precedence over *log_model*. If this attribute is empty but *log_model* has been provided, it will be filled at first access.

db_log(*self*, *transition*, *from_state*, *instance*, **args*, ***kwargs*)
Logs the transition into the database, saving the following elements:

- Name of the transition
- Name of the initial state
- `GenericForeignKey` to the modified instance
- `ForeignKey` to the user responsible for the transition
- timestamp of the operation

The default *TransitionLog* model is *django_xworkflows.xworkflow_log.models.TransitionLog*, but an alternative one can be specified in *log_model* or *log_model_class*.

Hint: Override this method to log to a custom TransitionLog with complex fields and storage.

log_transition(*self*, *transition*, *from_state*, *instance*, *save=True*, *log=True*, **args*, ***kwargs*)
In addition to `xworkflows.Workflow.log_transition()`, additional actions are performed:

- If `save` is `True`, the instance is saved.
- If `log` is `True`, the *db_log()* method is called to register the transition in the database.

Transition database logging

Transition logs can be stored in the database. This is performed by the *db_log()* method of the *Workflow* class.

The default method will save informations about the transition into an adapted model. The actual model to log will be:

- The model whose class is set to the *Workflow.log_model_class* attribute
- The model whose name (in an `app_label.ModelClass` format) is set to the *Workflow.log_model* attribute
- The *django_xworkflows.xworkflow_log.models.TransitionLog* model if `django_xworkflows.xworkflow_log` belongs to `settings.INSTALLED_APPS`
- Nothing if none of the above match

Such models are expected to have a few fields, a good basis for writing your own is to inherit from either *BaseTransitionLog* or *GenericTransitionLog* (which provides a default storage through a `GenericForeignKey`).

The *BaseTransitionLog* class provides all required fields for logging a transition.

class `django_xworkflows.models.`**BaseTransitionLog**(*models.Model*)
This class provides minimal functions for logging a transition to the database.

transition
This attribute holds the name of the performed transition, as a string.

from_state
Name of the source state, as a string.

to_state
Name of the target state, as a string.

timestamp
Timestamp of the operation, as a `DateTimeField`.

MODIFIED_OBJECT_FIELD
Name of the field where the modified instance should be passed. Logging the transition will likely fail if this is not provided.

EXTRA_LOG_ATTRIBUTES
It may be useful to log extra transition kwarg (`user`, ...) to the database. This attribute describes how to log those extra keyword arguments.

It takes the form of a list of 3-tuples `(db_field, kwarg, default)`. When logging to the database, the `db_field` attribute of the *BaseTransitionLog* instance will be filled with the keyword argument passed to the transition at `kwarg`, if any. Otherwise, `default` will be used.

get_modified_object(*self*)
Abstract the lookup of the modified object through *MODIFIED_OBJECT_FIELD*.

log_transition(*cls*, *transition*, *from_state*, *to_state*, *modified_object*, ***kwargs*)
Save a new transition log from the given transition name, origin state name, target state name, modified object and extra fields.

class `django_xworkflows.models.`**GenericTransitionLog**(*BaseTransitionLog*)
An extended version of *BaseTransitionLog* uses a `GenericForeignKey` to store the modified object.

content_type
A foreign key to the `ContentType` of the modified object

content_id
The primary key of the modified object

modified_object
The `GenericForeignKey` pointing to the modified object.

class `django_xworkflows.models.`**BaseLastTransitionLog**(*BaseTransitionLog*)
This alternate *BaseTransitionLog* has been tuned to store only the last transition log for an object, typically with a `OneToOneField`.

It handles update or creation on its own.

class `django_xworkflows.models.`**GenericLastTransitionLog**(*BaseLastTransitionLog*)
This class is to *BaseLastTransitionLog* what *GenericTransitionLog* is to *BaseTransitionLog*. It holds the modified object through a `GenericForeignKey`, with the adequate `unique_together` setting.

Here is an example of a custom `TransitionLog` model:

```
# Note that we inherit from BaseTransitionLog, not GenericTransitionLog.
class MyDocumentTransitionLog(django_xworkflows.models.BaseTransitionLog):

    # This is where we'll store the modified object
    document = models.ForeignKey(Document)

    # Extra data to keep about transitions
    user = models.ForeignKey(auth_models.User, blank=True, null=True)
    client = models.ForeignKey(api_models.Client, blank=True, null=True)
    source_ip = models.CharField(max_length=24, blank=True)

    # Set the name of the field where the modified object goes
    MODIFIED_OBJECT_FIELD = 'document'

    # Define extra logging attributes
```

```
    EXTRA_LOG_ATTRIBUTES = (
        ('user', 'user', None),
        ('client', 'api_client', None),  # Transitions are called with 'api_client' kwarg
        ('source_ip', 'ip', ''),  # Transitions are called with 'ip' kwarg
    )
```

An example *TransitionLog* model is available in the `django_xworkflows.xworkflow_log` application. Including it to `settings.INSTALLED_APPS` will enable database logging of transitions for all *WorkflowEnabled* subclasses.

class `django_xworkflows.xworkflow_log.models.`**`TransitionLog`**(*GenericTransitionLog*)
This specific *GenericTransitionLog* also stores the user responsible for the transition, if provided.

The exact `Model` to use for that foreign key can be set in the `XWORKFLOWS_USER_MODEL` django setting (defaults to `'auth.User'`, which uses `django.contrib.auth.models.User`).

Internals

Note: These classes are private API.

class `django_xworkflows.models.`**`WorkflowEnabledMeta`**(*xworkflows.base.WorkflowEnabledMeta*)
This metaclass is responsible for parsing a class definition, detecting all *StateField* and collecting/defining the associated *TransactionalImplementationWrapper*.

`_find_workflows`(*mcs*, *attrs*)
Collect all *StateField* from the given `attrs` (the default version collects *Workflow* subclasses instead)

`_add_workflow`(*mcs*, *field_name*, *state_field*, *attrs*)
Perform necessay actions to register the *Workflow* stored in a *StateField* defined at `field_name` into the given attributes dict.

It differs from the base implementation which adds a `StateProperty` instead of keeping the *StateField*.

Parameters

- **`field_name`** (*str*) – The name of the attribute at which the *StateField* was defined
- **`state_field`** (*StateField*) – The *StateField* wrapping the *Workflow*
- **`attrs`** (*dict*) – The attributes dictionary to update.

ChangeLog

0.11.0 (2017-02-25)

New:

- Add compatibility for Django up to 1.10 and Python 3.6

Note: This version drops support for Python 3.0-3.3 and Django<1.8.

0.10.1 (2016-06-26)

Bugfix:

- Don't choke on dj.db.migration passing in `unicode` instead of `str`
- Fix packaging / docs layout

0.10.0 (2016-02-15)

New:

- Add compatibility for Django up to 1.9

Bugfix:

- Fix invalid default TransitionLog ModelAdmin (lacking `readonly_fields`), thanks to btoueg
- Fix updating *`timestamp`* on *`BaseLastTransitionLog`* instances, thanks to tanyunshi

Note: This version drops support for Python 2.6; the next one will drop Django<1.7.

0.9.4 (2014-11-24)

Bugfix:

- Add support for `django.db.migrations` (Django >= 1.7)

0.9.3 (2014-06-04)

Packaging:

- #12: Prevent 'egg' packaging

0.9.2 (2013-09-25)

Bugfix:

- Fix migrations to take into account Django's `AUTH_USER_MODEL` setting.

0.9.1 (2013-08-14)

Bugfix:

- Fix packaging

0.9.0 (2013-05-16)

New:

- #10: Ask Django' templates to not call transitions, and give access to sub-methods (e.g `is_available()`). Contributed by kanu.

0.8.1 (2012-11-30)

Bugfix:

- #7: allow more than one *GenericTransitionLog* in the same project.

0.8.0 (2012-10-12)

New:

- Provide a base *BaseLastTransitionLog* and a *GenericLastTransitionLog*, useful for storing only the *last* transition log for a given model.

0.7.1 (2012-09-10)

Bugfix:

- Use `django.utils.timezone.now()` instead of `datetime.datetime.now()` with Django >= 1.4

0.7.0 (2012-08-17)

New:

- Provide a base *BaseTransitionLog* without `GenericForeignKey`.
- Ease specification of transition kwargs to store in custom *TransitionLog* classes
- Allow settings *log_model_class* explicitly (thus bypassing the lookup performed by *log_model*).

0.6.0 (2012-08-02)

New:

- Enable support for XWorkflows 0.4.0

0.5.0 (2012-07-14)

New:

- Add rebuild_transitionlog_states management command to refill *from_state* and *to_state*.
- Add indexes on various *django_xworkflows.models.BaseTransitionLog* fields

Bugfix:

- Fix *django_xworkflows.models.WorkflowEnabled* inheritance

0.4.5 (2012-06-12)

Bugfix:

- Don't default to *TransactionalImplementationWrapper* when using a *django_xworkflows.models.Workflow*.

0.4.4 (2012-05-29)

Bugfix:

- Serialize unicode of `xworkflows.base.State.title` in south ORM freezing.

0.4.3 (2012-05-29)

Bugfix:

- Include migrations in package

0.4.2 (2012-05-29)

Bugfix:

- Fix log=False/save=False when calling transitions

0.4.1 (2012-05-29)

Bugfix:

- Avoid circular import issues when resolving *`log_model`* to a `Model`
- Log source and target state names in *`BaseTransitionLog`*

0.4.0 (2012-04-29)

New:

- Improve south support
- Run transition implementations in a database transaction

0.3.1 (2012-04-15)

New:

- Introduce *`StateField`* for adding a *`Workflow`* to a model
- Adapt to xworkflows-0.3.0

CHAPTER 4

Resources

- Package on PyPI: http://pypi.python.org/pypi/django-xworkflows
- Repository and issues on GitHub: http://github.com/rbarrois/django_xworkflows
- Doc on https://django-xworkflows.readthedocs.io/
- XWorkflows on GitHub: http://github.com/rbarrois/xworkflows
- XWorkflows doc on https://xworkflows.readthedocs.io/

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

d

Symbols

A

B

C

D

E

F

G

L

M

N

S

T

W