django-multiform Documentation

Release 0.1a1

Baptiste Mispelon

September 15, 2016

Contents

Django-multiform is a library that allows you to wrap several forms into one object with a form-like API.

It's compatible with django 1.4 and 1.5.

A lot of care has been put into replicating the same API as Form so that a MultiForm can be used anywhere a regular Form would.

The library consists of two classes: `MultiForm` and `MultiModelForm`.

CHAPTER 1

MultiForm

Wraps up several `Form` into one object, which allows you for example to reuse several existing forms in a generic `FormView`.

```
# forms.py
from django import forms

from multiform import MultiForm

class FooForm(forms.Form):
    foo = forms.CharField()

class BarForm(forms.Form):
    bar = forms.CharField()

class FooBarForm(MultiForm):
    base_forms = [
        ('foo', FooForm),
        ('bar', BarForm),
    ]

# views.py
from django.views import generic
from .forms import FooBarForm

class FooBarView(generic.FormView):
    form_class = FooBarForm

    def form_valid(self, form):
        form.cleaned_data['foo'] # {'foo': ...}
        form.cleaned_data['bar'] # {'bar': ...}
        return super(FooBarView, self).form_valid(form)
```

CHAPTER 2

MultiModelForm

As the name hints, it wraps several `ModelForm` instances into one object.

It's quite similar to `MultiForm`, but it adds a `save` method and it can handle the dispatching of the `instance` attribute that you usually pass to a `ModelForm`.

It's useful for creating related model instances in one step with a generic CreateView for example.

```
# models.py
from django.db import models

class Person(models.Model):
    eye_color = models.CharField(max_length=50)
    user = models.OneToOneField(auth.get_user_model())

# forms.py
from django.contrib.auth.forms import UserCreationForm
from .models import Person

from multiform import MultiModelForm

class PersonUserForm(MultiModelForm):
    base_forms = [
        ('person', PersonForm),
        ('user', UserCreationForm),
    ]

    def dispatch_init_instance(self, name, instance):
        if name == 'person':
            return instance
        return super(PersonUserForm, self).dispatch_init_instance(name, instance)

    def save(self, commit=True):
        """Save both forms and attach the user to the person."""
        instances = super(PersonUserForm, self).save(commit=False)
        instances['person'].user = instances['user']
        if commit:
            for instance in instances.values():
                instance.save()
        return instances
```

CHAPTER 3

Dispatching Parameters

In the event that you want to pass different parameters to some of the wrapped forms, you have two options (that can be used independently):

1. Implement a `dispatch_init_$arg` method on your subclass. This method will be called when builting the keyword arguments passed to a wrapped form's constructor. This method is passed two arguments: the name of the wrapped form being built, and the original value of the $arg keyword argument.
2. Pass a `$name__$arg=foo` keyword argument to the MultiForm's constructor. This will make it so that the wrapped form with the name of `$name` will be passed the `$arg=foo` keyword argument. Note that in case of conflicts, this method has priority over the first one.

Any keyword argument passed to a Multiform's contructor that's not part of the Form's signature and that's not of the form `$name__*` will be passed to all wrapped forms.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search