Djam Documentation

Release 0.1.0

Participatory Culture Foundation

December 24, 2013

Contents

Djam is a third-party admin with the following goals:

- Provide basic CRUD functionality similar to the current admin.
- Ease the transition by supporting auto-generation of an admin based on installed ModelAdmins.
- Support most of the basic ModelAdmin options out-of-the-box.
- Make it easy to add new custom areas to the admin that don't fit into the traditional CRUD model.
- Use the latest front-end tools and design patterns.

The name is alliterative; the beginning of Djam is pronounced the same as the beginning of Django.

CHAPTER 1

Links

docs http://django-djam.readthedocs.org/

code https://github.com/django-djam/django-djam/

irc #django-djam on irc.freenode.net

build status

CHAPTER 2

Getting Started

2.1 Quick Start

Trying out djam is easy, especially if you're already using `django.contrib.admin`. First, just install it.

```
$ pip install django-djam==0.1.0
```

Djam requires Django 1.5+ and Django-Floppyforms 1.1+.

2.1.1 urls.py

Next, go into the urls.py file for your project. There is probably something there along these lines:

```
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
)
```

All you need to do here is change it to look more like the following:

```
import djam
djam.autodiscover()

urlpatterns = patterns('',
    url(r'^admin/', include(djam.admin.urls)),
)
```

Note: It's also entirely possible to run `djam` and `django.contrib.admin` side-by-side, if you want to compare the two. The test project does it!

2.1.2 settings.py

Now go into your settings file.

First, make sure that `djam` and `floppyforms` are somewhere in your `INSTALLED_APPS`. For example:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Other apps...
    'djam',
    'floppyforms',
)
```

Now make sure that `django.core.context_proocessors.request` is in your `TEMPLATE_CONTEXT_PROCESSORS`. If you just add it to Django's defaults, you'll get:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.core.context_processors.tz",
    "django.contrib.messages.context_processors.messages",
    "django.core.context_processors.request",
)
```

2.1.3 Congratulations!

You're now using djam. Go ahead and log in; have a look around. All of the `ModelAdmins` you registered with `django.contrib.admin` should already be showing up on the dashboard. Whenever you feel ready, you can move on to *Extending the Admin* and find out why that is.

2.2 Extending the Admin

> **Riff** *n.* (in popular music and jazz) A short repeated phrase, frequently played over changing chords or used as a background to a solo improvisation.

Like a riff is a building block of jazz music, a `Riff` is the building block of the Djam admin. Riffs are essentially modular, decoupled chunks of namespaced functionality which can be attached to the main admin site. ModelAdmins (and the Django admin site) work on the same principle. The main difference in `djam` is that riffs don't necessarily map one-to-one to models. There can be multiple registered riffs related to the same model, and there can be riffs registered which don't relate to any model at all.

2.2.1 ModelRiffs

We'll start out with `ModelRiffs`, which are essentially analogous to `ModelAdmins`. Here is how you could set up some basic ModelRiffs:

```
# myapp/riffs.py
from djam.riffs.models import ModelRiff
```

```
from myapp.models import MyModel, MyOtherModel


class MyModelRiff(ModelRiff):
    model = MyModel


class MyModelRiff2(ModelRiff):
    model = MyModel
    namespace = 'myapp_mymodel2'
    display_name = 'MyModel 2'
    slug = 'mymodel-2'


class MyOtherModelRiff(ModelRiff):
    model = MyOtherModel
    use_modeladmin = False


# Djam looks for this variable during :func:`.autodiscovery <autodiscover>`.
riffs = [MyModelRiff, MyOtherModelRiff]
```

Each `ModelRiff` class has an associated model. If a ModelAdmin has been registered for that model, the riff will by default inherit a number of options from that ModelAdmin, unless `use_modeladmin` is `False`.

2.2.2 Autodiscovery

djam.**autodiscover**(*self*, *with_batteries=True*, *with_modeladmins=True*)

`djam.autodiscover()` loops through `INSTALLED_APPS` and loads any `riffs.py` modules in those apps - similar to the django admin's autodiscovery.

On top of that, `djam` will by default include `ModelRiffs` which have been auto-generated from registered ModelAdmins. This can be turned of by passing in `with_modeladmins=False`.

For some commonly-used models (such as `django.contrib.auth.User`) `djam` provides a Riff which handles some functionality that would otherwise be lost during the conversion from ModelAdmin to `ModelRiff`. This can be disabled by passing in `with_batteries=False`.

The order that these are loaded is: app riff modules, "batteries", and ModelAdmins. If two riffs are registered using the same namespace, the first one registered will take precedence; any others will be ignored.

2.2.3 Great, so it can replace the admin.

Why yes, it can. But there's also a lot you can do that goes *beyond the capabilities of Django's admin*.

2.3 Beyond `django.contrib.admin`

2.3.1 A Model-less Riff

Here's an example of a riff which is simple to implement in Djam, but which in Django's admin would require (at the very least) overriding several base templates and using a custom AdminSite subclass. This code is taken directly from the test project's example app.

```
# example_app/riffs.py
from django.conf.urls import patterns, url
from djam.riffs.base import Riff

from test_project.example_app.views import HelloView, HelloFinishedView


class HelloRiff(Riff):
    display_name = "Hello"

    def get_extra_urls(self):
        return patterns('',
            url(r'^$',
                HelloView.as_view(**self.get_view_kwargs()),
                name='hello'),
            url(r'^(?P<slug>[\w-]+)/$',
                HelloFinishedView.as_view(**self.get_view_kwargs()),
                name='hello_finished'),
        )

    def get_default_url(self):
        return self.reverse('hello')


riffs = [HelloRiff]
```

Essentially, this riff has one view which asks a user for their name (a slug) and another view which says "Hello <name>". It doesn't use any models. Or more to the point, it doesn't provide a CRUD interface for interacting with a particular model.

```
# example_app/forms.py
from django import forms


class HelloWorldForm(forms.Form):
    name = forms.SlugField()

# example_app/views.py
from django.http import HttpResponseRedirect
from djam.views.generic import FormView, TemplateView

from test_project.example_app.forms import HelloWorldForm


class HelloView(FormView):
    form_class = HelloWorldForm
    template_name = 'djam/form.html'

    def form_valid(self, form):
        url = self.riff.reverse('hello_finished',
                                slug=form.cleaned_data['name'])
        return HttpResponseRedirect(url)


class HelloFinishedView(TemplateView):
    template_name = 'example_app/hello.html'

    def get_crumbs(self):
```

```
        crumbs = super(HelloFinishedView, self).get_crumbs()
        return crumbs + [(None, self.kwargs['slug'])]
```

The notable difference here is that we're importing our generic views from `djam.views.generic` rather than `django.views.generic`. These views have some riff-specific functionality, including providing the riff itself – and breadcrumbs for navigation – to the template context.

```
{# example_app/templates/example_app/hello.html #}
{% extends 'djam/__base.html' %}

{% load webdesign djam %}

{% block main %}
    <h1>Hello, {{ slug }}!</h1>

    {% lorem 3 p random %}

    <a href="{% riff_url riff 'hello' %}" class='btn btn-success btn-large'>Once more!</a>
{% endblock %}
```

In this template, you can see the `{% riff_url %}` template tag being used. It reverses the given view name (`'hello'`) within the namespace of the given riff (in this case the current riff).

And that's it!

CHAPTER 3

Contributing

3.1 Contributing to Djam

3.1.1 Coding style

- Unless otherwise specified, follow PEP 8. Since we're a new project, there shouldn't be any cases (apart from those outlined here) where you need to instead conform to surrounding code.
- Use four spaces for indentation.
- In docstrings, use "action words". For example, "Calculates the number of apples" rather than "Calculate the number of apples".
- Empty lines should not contain indentation.
- Always use absolute imports. They're easier to debug and are 3.0 forwards-compatible.
- Be sure to include `from __future__ import unicode_literals` at the top of any file that uses strings.

3.1.2 Git practices

We follow the same guidelines for using git (and github) as Django, with one exception: we prefer ticket branches be named `ticket/xxxx`.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*