django-ostinato Documentation

Release 1.1

Andre Engelbrecht

June 19, 2016

Contents

"In music, an ostinato (derived from Italian: "stubborn", compare English: obstinate) is a motif or phrase, which is persistently repeated in the same musical voice."

—Wikipedia

CHAPTER 1

Requirements

- django >= 1.4.2
- django-mptt == 0.6.0
- django-appregister == 0.3.1

CHAPTER 2

Introduction

Django-ostinato is a collection of applications that aims to bring together some of the most common features expected from a CMS.

Every feature is contained within a single ostinato app. This helps us to keep functionality focussed on the single feature/task that the app is meant to provide

2.1 The Demo Project

Ostinato comes with a demo project that you can use to play around with the app. The test project uses zc.buildout, which lets you install and run the entire demo, including all dependencies, in an isolated environment.

2.1.1 Setting up the demo project

After checking out or downloading the source, you will see the `demoproject` folder. There should be two files in that folder `bootstrap.py` and `buildout.cfg`. The actual django project is in `demoproject/src/odemo`.

Lets build the project. To do so you bootstrap it using the python version of your choice.

`python bootstrap.py` or you could do, `python2.6 bootstrap.py`. Just remember that ostinato have not been tested with versions lower than 2.6.

Ok, after the bootstrap, you will see there should now be a `bin` folder.

Now run: `./bin/buildout`

This will start to download django, mptt, an any other dependecies required for the project to run.

2.1.2 Running the demo project

Once the buildout has been created, and is finished. A new file will be in the `bin` folder called `odemo`. This is basically a wrapper for `manage.py` that ensures that the project is run within buildout, and not in the system.

So lets sync the database: `./bin/odemo syncdb`

After the sync we can run the dev server: `./bin/odemo runserver`

2.2 ostinato.pages

For the user - Allows for creating a hierarchy of pages, manage publishing, and displaying the pages in the site's navigation.

For the Developer - Allows for creating custom Content for Pages, which can be customized on a per-project-basis.

2.2.1 A quick overview

Pages

In our pages app, a Page is nothing more than a container for content. A Page does have some of it's own field and attributes, but these are mostly to set certain publication details etc.

Page Content

Page Content is a seperate model from pages, and this is the actual content for the page. Two of these models already exist within pages, and you are free to use them out-of-the-box, but you can easilly create your own if you need more control over content in your pages.

2.2.2 Requirements

- django-mptt
- django-appregister

2.2.3 Add `ostinato.pages` to your project

Start by adding the app to your `INSTALLED_APPS`

```
INSTALLED_APPS = (
    ...

    'ostinato',
    'ostinato.pages',
    'mptt',  # Make sure that mptt is after ostinato.pages

    ...
)
```

Now add the `ostinato.pages.urls` to the *end* of your urlpatterns. It is important to add this snippet right at the end of the `urls.py` so that pages doesn't take priority over your other urlpatterns. That is of course unless you want it to, in which case you can add it where-ever you wish.

```
urlpatterns += patterns('',
    url(r'^', include('ostinato.pages.urls')),
)
```

Remember to run `syncdb` after you've done this.

That's it, you now have a basic Pages app. We cannot do anything with it yet, since we first need to create actual templates and content. We'll do this in the next section.

Note: Publication and Timezones

Django 1.4 changed how timezones are treated. So if you mark a page as published, please remember that it is published, relative to the timezone you specified in your settings.py.

2.2.4 Creating and registering page content

Ok, so lets say the client wants a landing page that contains a small `intro` and `content`, and a general page that contains only `content`. It was decided by you that these would all be TextFields.

Lets create these now. You need to place these in your app/project `models.py`.

```
1  from django.db import models
2  from ostinato.pages.models import PageContent
3  from ostinato.pages.regitry import page_content
4
5  @page_content.register
6  class LandingPage(PageContent):  # Note the class inheritance
7      intro = models.TextField()
8      content = models.TextField()
9
10 @page_content.register
11 class GeneralPage(PageContent):
12     content = models.TextField()
```

As you can see, these are standard django models, except that we inherit from `ostinato.pages.models.PageContent`.

You also need to register your model with the `page_content` registry, as you can see on lines 5 and 10.

Note: Since the content you just created are django models, you need to remember to run syncdb.

If you load up the admin now, you will be able to choose a template for the page.

2.2.5 Displaying page content in the templates

By default the template used by the page is determined by the page content. The default template location is `pages/<content_model_name>.html`. So the templates for our two content models (which you'll need to create now) are:

- `pages/landing_page.html`
- `pages/general_page.html`

Note: You can override these templates by using the `ContentOptions` meta class in your page content model.

```
class GeneralPage(PageContent):
    content = models.TextField()

    class ContentOptions:
        template = 'some/custom/template.html'
```

Lets see how we can access the content in the template.

The page view adds `page` to your context, which is the current page instance. Using that it's very easy to do something like this:

```
<h1>{{ page.title }}</h1>
<p class="byline">Author: {{ page.author }}</p>
```

That's all fine, but we have content for a page as well, which is stored in a different model. We include a field on the page called `contents`, which will get the related page content for you.

In the following example, we assume that you are editing your `landing_page.html`.

```
<p>{{ page.contents.intro }}</p>
<p>{{ page.contents.content }}</p>
```

Note: You can also access the content using the django related field lookups, but this method is very verbose and requires a lot of typing. The related name is in the format of, `<app_label>_<model>_content`.

```
<p>{{ page.myapp_landingpage_content.intro }}</p>
<p>{{ page.myapp_landingpage_content.content }}</p>
```

2.2.6 Creating a custom view for your content

There are cases where you may want to have a custom view to render your template rather than just using the default view used by `ostinato.pages`.

One use case for this may be that one of your pages can have a contact form. So you will need a way to add this form to the page context. You also want this page to handle the post request etc.

First you create your view. Note that `ostinato.pages` makes use of django's class based views. If you haven't used them before, then it would help to read up on them.

```
from ostinato.pages.views import PageView

class ContactView(PageView):  # Note we are subclassing PageView

    def get(self, *args, **kwargs):
        c = self.get_context_data(**kwargs)
        c['form'] = ContactForm()
        return self.render_to_response(c)

    def post(self, *args, **kwargs):
        c = self.get_context_data(**kwargs)
        ## Handle your form ...
        return http.HttpResponseRedirect('/some/url/')
```

In the example above, we created our own view that will add the form to the context, and will also handle the post request. There is nothing special here. It's just the standard django class based views in action.

One thing to note is that our `ContactView` inherits from `PageView` (which in turn inherits from `TemplateView`). You don't *have* to inherit from PageView, but if you don't, then you need to add the `page` instance to the context yourself, whereas `PageView` takes care of that for you.

Next we need to tell the page content model to use this view when it's being rendered. We do this in the `ContentOptions` meta class for the page content.

Using our `LandingPage` example from earlier, we change it like so:

```
from django.db import models
from ostinato.pages.models import PageContent

```

```
class LandingPage(PageContent):
    intro = models.TextField()
    content = models.TextField()

    class ContentOptions:
        view = 'myapp.views.ContactView'  # Full import path to your view
```

2.2.7 Custom forms for Page Content

`ostinato.pages` also allows you to specify a custom form for page content. You do this in the ContentOptions class like the example below:

```
from django.db import models
from ostinato.pages.models import PageContent

class LandingPage(SEOContentMixin, PageContent):
    intro = models.TextField()
    content = models.TextField()

    class ContentOptions:
        form = 'myapp.forms.CustomForm'  # Full import path to your form
```

As you can see we just added that at the end. Just create your custom form on the import path you specified, and the admin will automatically load the correct form for your page content.

2.2.8 Creating complex page content with mixins

Sometimes you may have two different kinds of pages that share other fields. Lets say for example we have two or more pages that all needs to update our meta title and description tags for SEO.

It is a bit of a waste to have to add those two fields to each of our content models manually, not to mention that it introduces a larger margin for errors.

We use mixins to solve this:

```
from django.db import models
from ostinato.pages.models import PageContent

class SEOContentMixin(models.Model):  # Note it's a standard model...
    keywords = models.CharField(max_length=200)
    description = models.TextField()

    class Meta:
        abstract = True  # ...and _must_ be abstract


class LandingPage(SEOContentMixin, PageContent):
    intro = models.TextField()
    content = models.TextField()
```

The two points you have to be aware of here:

1. The mixin should be a normal django model
2. The mixin *must* be abstract

2.2.9 Custom Statemachine for Pages

`ostinato.pages.workflow` provides a default statemachine that is used by the page model. Sometimes, you may want to create a different workflow for the pages based on client requirements.

To do this, you just create your custom statemachine as mentioned in the `ostinato.statemachine` documentation, and then tell `ostinato.pages` which class to use by adding the following in your settings.py:

```
OSTINATO_PAGES_WORKFLOW_CLASS = "myapp.customworkflow.SomeWorkflow"
```

When creating your custom workfow, do remember that the Page state is stored as an IntegerField, so make sure that you use the `IntegerStatemachine` so subclass your own statemachine class.

2.2.10 Extra Inline Fields for a Page in the Admin

There are cases where you want a specific page to have extra inline fields, based on the chosen template. We have provided you with this capability through the PageContent model.

First you need to create the model that should be related to your page.

```
from django.db import models
from ostinato.pages.models import Page

class Contributor(models.Model):
    page = models.ForeignKey(Page)
    name = models.CharField(max_lenght=50)
```

Next, you need to create your inline class (usually done in admin.py).

```
from django.contrib import admin

class ContributorInline(admin.StackedInline):
    model = Contributor
```

Right, after a quick syncdb, we are ready to add this to our page content. Lets say that we want to add contributors to our `LandingPage` from earlier:

```
...

class LandingPage(SEOContentMixin, PageContent):
    intro = models.TextField()
    content = models.TextField()

    class ContentOptions:
        admin_inlines = ['myapp.admin.ContributorInline']
...
```

If you load up the django admin now, and edit a Landing Page, you should see the extra inline model fields below your PageContent.

To access the related set in your template, just do it as normal.

```
{% for contributor in page.contributor_set.all %}
{{ contributor.name }}
{% endfor %}
```

2.2.11 Template tags and filters

`ostinato.pages` comes with a couple of tempalate tags and filters to help with some of the more common tasks.

navbar(for_page=None)

A inclusion tag that renders the navbar, for the root by default. It will render all child pages for the node. This tag will only render pages that has `show_in_nav` selected and is published.

```
{% load pages_tags %}
{% navbar %}
```

This inclusion tag uses `pages/navbar.html` to render the nav, just in case you want to customize it.

This inclusion tag can also take a extra arument to render the children for a specific page.

```
{% load pages_tags %}
{% navbar for_page=page %}
```

breadcrumbs(for_page=None, obj=None)

This tag will by default look for `page` in the context. If found it will render the breadcrumbs for this page's ancestors.

```
{% load pages_tags %}
{% breadcrumbs %}
```

If you want to manually specify the page for which to render the breadcrumbs, you can do that using `for_page`.

```
{% load pages_tags %}
{% breadcrumbs for_page=custom_page %}
```

Sometimes you may have a object that does not belong to the standard page hierarchy. This could be a model like a BlogEntry, but when viewing the detail template for this entry, you may still want to relate this object to a page. For this you can use `obj`.

```
{% load pages_tags %}
{% breadcrumbs for_page=blog_landingpage obj=entry %}
```

One thing to note about the custom object is that the model must have a `title` attribute, and a `get_absolute_url()` method.

filter_pages(kwargs)**

This tag will filter the pages by `**kwargs` and return the the queryset.

```
{% load pages_tags %}
{% filter_pages state=5 as published_pages %}
{% for p in published_pages %}
    <p>{{ p.title }}</p>
{% endfor %}
```

get_page(kwargs)**

Same as `filter_pages`, except that this tag will return the first item found.

```
{% load pages_tags %}
{% get_page slug='page-1' as mypage %}
<h1>{{ mypage.title }}</h1>
```

2.2.12 Multi Site Support

Ostinato pages can be used for multiple sites. We suggest you read the django documentation on the sites framework for a background on how we use it here.

Every site will have it's own Tree. So you create one root page for every site, and then all descendants will belong to this site.

Additionally, in the `settings.py` file for each site, you need to specify which TreeID belongs to that site.

`OSTINATO_PAGES_SITE_TREEID = 1` will tell tell your project that the current projects pages are all located in the first Tree.

2.2.13 Pages Settings

```
OSTINATO_PAGES_SETTINGS = {
    'CACHE_NAME': 'default',
    'DEFAULT_STATE': 5,
}
```

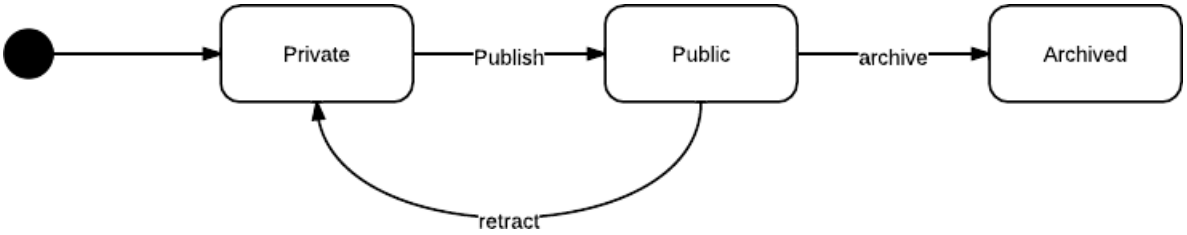
2.3 ostinato.statemachine

2.3.1 Overview

Ostinato includes a statemachine that will allow you to create complex workflows for your models. A common workflow, for example, is a publishing workflow where an item can be either `private` or `public`. The change from the one state to the next is called a transition.

In ostinato our main aim was to have the ability to "attach" a statemachine to a model, without having to change any fields on that model. So you can create your States and StateMachines completely independent of your models, and just *attach* it when needed.

Ok, lets build an actual statemachine so you can see how it works. For this example we will create the following statemachine:



For our example we will assume you are creating a statemachine for the following model:

```
class NewsItem(models.Model):
    title = models.CharField(max_length=150)
    content = models.TextField()
    publish_date = models.DateTimeField(null=True, blank=True)
    state = models.CharField(max_length=50, default='private')
```

We start by creating our States...

```
from ostinato.statemachine import State, StateMachine

class Private(State):
    verbose_name = 'Private'
    transitions = {'publish': 'public'}

class Public(State):
    verbose_name = 'Public'
    transitions = {'retract': 'private', 'archive': 'archived'}

```

```
class Archived(State):
    verbose_name = 'Archived'
    transitions = {}
```

This is simple enough. Every state is a subclass of `ostinato.statemachine.core.State` and each of these states specifies two attributes.

- `verbose_name` is just a nice human readable name.
- **`transitions` is a dict where the *keys* are transition/action names, and** the *values* is the target state for the transition.

Now we have to glue these states together into a statemachine.

```
class NewsWorkflow(StateMachine):
    state_map = {'private': Private, 'public': Public, 'archived': Archived}
    initial_state = 'private'
```

- **`state_map` is a dict where *keys* are unique id's/names for the states;** *values* are the actual `State` subclass
- `initial_state` is the starting state *key*

Thats all you need to set up a fully functioning statemachine.

Lets have a quick look at what this allows you to do:

```
>>> from odemo.news.models import NewsItem, NewsWorkflow

# We need an instance to work with. We just get one from the db in this case
>>> item = NewsItem.objects.get(id=1)
>>> item.state
u'public'

# Create a statemachine for our instance
>>> sm = NewsWorkflow(instance=item)

# We can see that the statemachine automatically takes on the state of the
# newsitem instance.
>>> sm.state
'Public'

# We can view available actions based on the current state
>>> sm.actions
['retract', 'archive']

# We can tell the statemachine to take action
>>> sm.take_action('retract')

# State is now changed in the statemachine ...
>>> sm.state
'Private'

# ... and we can see that our original instance was also updated.
>>> item.state
'private'
>>> item.save()  # Now we save our news item
```

2.3.2 Custom Action methods

You can create custom *action methods* for states, which allows you to do extra stuff, like updating the publish_date.

Our example `NewsItem` already has a empty `publish_date` field, so lets create a method that will update the publish date when the `publish` action is performed.

```
1  from django.utils import timezone
2
3  class Private(State):
4      verbose_name = 'Private'
5      transitions = {'publish': 'public'}
6
7      def publish(self, **kwargs):
8          if self.instance:
9              self.instance.publish_date = timezone.now()
```

Now, whenever the `publish` action is called on our statemachine, it will update the `publish_date` for the instance that was passed to the `StateMachine` when it was created.

Note: The name of the method is important. The `State` class tries to look for a method with the same name as the `transition` *key*.

You can use the `kwargs` to pass extra arguments to your custom methods. These arguments are passed through from the `StateMachine.take_action()` method eg.

```
sm.take_action('publish', author=request.user)
```

2.3.3 Admin Integration

Integrating your statemachine into the admin is quite simple. You just need to use the statemachine form factory function that generates the form for your model, and then use that form in your ModelAdmin.

```
1  from odemo.news.models import NewsItem, NewsWorkflow
2  from ostinato.statemachine.forms import sm_form_factory
3
4
5  class NewsItemAdmin(admin.ModelAdmin):
6      form = sm_form_factory(NewsWorkflow)
7
8      list_display = ('title', 'state', 'publish_date')
9      list_filter = ('state',)
10     date_hierarchy = 'publish_date'
11
12
13 admin.site.register(NewsItem, NewsItemAdmin)
```

Lines 2 and 6 are all that you need. `sm_form_factory` takes as it's first argument your Statemachine Class.

2.3.4 Custom `state_field`

The statemachine assumes by default that the model field that stores the state is called, `state`, but you can easilly tell the statemachine (and the statemachine form factory function) what the field name for the state will be.

- Statemachine - `sm = NewsWorkflow(instance=obj, state_field='field_name')`

- Form Factory - `sm_form_factory(NewsWorkflow, state_field='field_name')`

2.4 ostinato.blog

2.4.1 Overview

A blog is a very common application that are installed for most websites these days. There are a couple of common features that most blogging apps provide, but the use cases of every project can be quite different.

Because of this, we decided to bundle a simple skeleton for building your own blog, and this is what `ostinato.blog` does.

2.4.2 How to use `ostinago.blog`

Start by creating your own blogging application, and within it your own `BlogEntry` model.

```
from ostinato.blog.models import BlogEntryBase

class Entry(BlogEntryBase):
    pass
```

`BlogEntryBase` provides the following fields for your `Entry` Model.

```
title = models.CharField(max_length=255)
slug = models.SlugField(unique=True)
content = models.TextField()
state = models.IntegerField(default=1)
author = models.ForeignKey(User)
created_date = models.DateTimeField(auto_now_add=True)
modified_date = models.DateTimeField(auto_now=True, null=True, blank=True)
publish_date = models.DateTimeField(null=True, blank=True)
archived_date = models.DateTimeField(null=True, blank=True)

allow_comments = models.BooleanField(default=True)
```

Those are the most basic fields that any blog might require, but of course you can extend this to include any other fields that you may require.

```
from ostinato.blog.models import BlogEntryBase

class Entry(BlogEntryBase):

    contributors = models.ManyToManyField(User, null=True, blank=True)
    preview_image = models.Imagefield(upload_to='uploads', null=True, blank=True)
```

So now you have a blog entry with two extra fields.

2.4.3 Using the custom manager

`published()` - Returns a queryset containing published blog entries

2.4.4 Wrapping up

Since blogs can vary in use case so much, we have decided to provide only the bare minimum to get you going and you still need to create your own urls, views and templates.

The reason for this approach is that we still wish to maintain flexability, and we feel that this is the best way to approach this.

2.5 ostinato.contentfilters

The `ostinato.contentfilters` app provides you with a easy way to apply a list of filters to content, which is rendered in your templates. The basic functionality is the same as standard django template filters, except that they are appllied as a group. This is handy if you want to apply a whole range of filters to a single piece of content.

We also include a couple of filters for some common use cases.

2.5.1 Writing a Contentfilter

Start by creating a standard django template tag library. In our case we will call this `custom_filters.py`.

```
from django import template
register = template.Library()
```

Now we need to create our filter. For this example we will create a simple filter that will convert the content to uppercase.

```
from django import template
from ostinato.contentfilters import ContentMod

register = template.Library()

def to_upper(content):
    return content.upper()

ContentMod.register('upper', to_upper)
```

As you can see you just create a basic function, which takes a single argument, `content`. You then do some processing on your content, and return the result.

The last thing you need to do is register your modifier with `ostinato.contentfilters.ContentMod`. The first argument here is the unique name for the filter. The second argument is the function to use.

2.5.2 Using the filters in your templates

Now that you have your content filters registered, you can use them in your templates.

```
{% load content_filters custom_filters %}
{{ 'some lowercase content'|modify }}
```

Firstly note that we need to load both template tag libraries for `content_filters` and `custom_filters`.

Secondly you will see we just applied a single `modify` filter to our content. Calling `modify` without any arguments will run through *all* registered filters.

You can apply specific filters by passing it as arguments to `modify`:

```
{% load content_filters custom_filters %}
{{ 'some lowercase content'|modify:"upper,another_filter" }}
```

You can also tell it to exclude filters. The following will use *all* registered filters, except for `upper` and `youtube`. Note the exclamation mark at the start of the filter list.

```
{% load content_filters custom_filters %}
{{ 'some lowercase content'|modify:"!upper,another_filter" }}
```

2.5.3 Default content filters included with ostinato.contentfilters

- **`youtube` - Searches for a youtube url in the content, and replaces it with** the html embed code.
- **`snip` - Searches for a string, `{{{snip}}}` in the content, and if found** it truncates the content at that point.
- **`hide_snip` - Searches for a string, `{{{snip}}}` in the content, and if** found, removes it from the content.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search