django-fancypages Documentation

Release 0.1.0

Sebastian Vetter

December 05, 2014

Contents

The core principle of *fancypages* (FP) is to provide the user with a way to edit and enhance content without giving them too much control over style and layout. The objective is to maintain the overall design of the website.

The project was born out of the need to add content editing capabilities to an e-commerce project based on django-oscar.

Warning: **Django 1.7** support is currently only available for the standalone version of fancypages. The `fancypages.contrib.oscar_fancypages` integration package doesn't support it yet because django-oscar doesn't support it yet which makes it impossible to create migrations for `oscar_fancypages`.

Contents:

CHAPTER 1

Installation

You can use *django-fancypages* as standalone app in your Django project or you can integrate it with your django-oscar shop using the included extension module. In the following sections, the standalone setup of django-fancypages will be referred to as *FP* and the Oscar integration as *OFP*.

#Most of the installation steps are exactly the same for both so let's #go through these steps first. After you have completed them, follow the

Note: The two sandbox sites in FP show an example integration with django-oscar and as standalone project. They both use django-configurations maintained by the awesome Jannis Leidel to make dealing with Django settings much simpler. Using it is not a requirement for *django-fancypages* it's just a personal preference. The following settings explain the setup using the basic Django `settings.py` but I recommend checking out *django-configurations*.

1.1 Installing Fancypages

For both FP and OFP, you have to install the python package *django-fancypages* which is available on PyPI and can be installed with:

```
$ pip install django-fancypages
```

or you can install the latest version directly from the github repo:

```
$ pip install git+https://github.com/tangentlabs/django-fancypages.git
```

1.2 Standalone Setup

Let's start with adding all required apps to you `INSTALLED_APPS`. FP relies on several third-party apps in addition to the `fancypages` app itself. For convenience, FP provides two functions `get_required_apps` and `get_fancypages_apps` that make it easy to add all apps in one additional line of code:

```
from fancypages import get_required_apps, get_fancypages_apps

INSTALLED_APPS = [
    ...
] + get_required_apps() + get_fancypages_apps()
```

Note: FP supports **Django 1.7** which replaces `South` migrations with a new migration system integrated in Django. The `fancypages.migrations` module containse the *new-style* migrations and will only work for Django 1.7+.

For **Django 1.5 and 1.6**, you have to add `south` to your installed apps and specify an alternative migrations module in the `SOUTH_MIGRATION_MODULES` settings. Add the following to your settings when using either of these versions:

```
SOUTH_MIGRATION_MODULES = {
    'fancypages': "fancypages.south_migrations",
}
```

It will then behave in exactly the same way as before.

Next you have add a piece of middleware that provide the content editor functionality on pages that are managed by FP. The content editor works similar to django-debug-toolbar and uses the same middleware mechanism to inject additional mark up into every FP-enabled page if the current user has admin privileges. Add the FP middleware to the end of your `MIDDLEWARE_CLASSES`:

```
MIDDLEWARE_CLASSES = (
    ...
    'fancypages.middleware.EditorMiddleware',
)
```

Fancypages requires several default settings to be added. To make sure that you have all the default settings in your settings, you can use the defaults provided by fancypages itself. Add the following in your settings file **before** you overwrite specific settings:

```
...
from fancypages.defaults import *

# override the defaults here (if required)
...
```

Finally, you have to add URLs to your `urls.py` to make the fancypages dashboard and all FP-enabled pages available on your sight. FP uses a very broad matching of URLs to ensure that you can have nicely nested URLs with your pages. This will match **all** URLs it encounters, so make sure that you add them as the very last entry in your URL patterns:

```
urlpatterns = patterns('',
    ...
    url(r'^', include('fancypages.urls')),
)
```

If you would like the home page of your project to be an FP-enabled page as well, you have to add one additional URL pattern:

```
urlpatterns = patterns('',
    url(r'^$', views.HomeView.as_view(), name='home'),
    ...
    url(r'^', include('fancypages.urls')),
)
```

This view behaves slightly different from a regular `FancyPageView`: if no `FancyPage` instance exists with the name `Home` (and the corresponding slug `home`), this page will be created automatically as a "Draft" page. Make sure that you publish the page to be able to see it as non-admin user.

1.3 Setup Alongside Oscar

Note: The following instructions assume that you have Oscar set up succesfully by following Oscar's documentation. Addressing Oscar-specific set up details aren't considered here. We recommend that you take a close look at Oscar's documentation before continuing.

Setting up *django-fancypages* alongside your django-oscar shop is very similar to the standalone setup. You also have to add extra apps to your `INSTALLED_APPS` and once again, you can use the convenience function provided by fancypages. Note that we pass `use_with_oscar=True` to ensure that the `fancypages.contrib.oscar_fancypages` app is added:

```
from fancypages import get_required_apps, get_fancypages_apps

INSTALLED_APPS = [
    ...
] + fp.get_required_apps() \
  + fp.get_fancypages_apps(use_with_oscar=True) \
  + get_core_apps()
```

Note: Once again, FP ships the *new-style* migrations for Django 1.7+ by default. If you are using Django 1.5 or 1.6, you have to make sure that you have `south` in your `INSTALLED_APPS` and add the following setting to point to the alternative South migrations:

```
SOUTH_MIGRATION_MODULES = {
    'fancypages': "fancypages.south_migrations",
    'oscar_fancypages': 'fancypages.contrib.oscar_fancypages.south_migrations',  # noqa
}
```

You can now use `syncdb` and `migrate` as you would normally.

Next you have add a piece of middleware that provide the content editor functionality on pages that are managed by FP. The content editor works similar to django-debug-toolbar and uses the same middleware mechanism to inject additional mark up into every FP-enabled page if the current user has admin privileges. Add the FP middleware to the end of your `MIDDLEWARE_CLASSES`:

```
MIDDLEWARE_CLASSES = (
    ...
    'fancypages.middleware.EditorMiddleware',
)
```

Similar to the standalone setup, you have to import the default settings for FP in your `settings.py`. However, to make the integration with Oscar seamless, you have to set the `FP_NODE_MODEL` to Oscar's `Category` model. The reason for this is, that categories in Oscar already provide a tree-structure on the site that we can leverage. Switching the page node from FP's internal model to Oscar's `Category` is as easy as:

```
...
from fancypages.defaults import *

FP_NODE_MODEL = 'catalogue.Category'
FP_PAGE_DETAIL_VIEW = 'fancypages.contrib.oscar_fancypages.views.FancyPageDetailView'
...
```

In addition, you should integrate the page management dashboard with Oscar's builtin dashboard. We recommend replacing the entry "Catalogue > Categories" with FP's page management by replacing:

```
OSCAR_DASHBOARD_NAVIGATION = [
    ...
        {
            'label': _('Categories'),
            'url_name': 'dashboard:catalogue-category-list',
        },
    ...
]
```

with:

```
OSCAR_DASHBOARD_NAVIGATION = [
    ...
        {
            'label': _('Pages / Categories'),
            'url_name': 'fp-dashboard:page-list',
        },
    ...
]
```

This usually means, you have to copy the entire `OSCAR_DASHBOARD_NAVIGATION` dictionary from `oscar.defaults` to overwrite it with your own.

The last thing to configure is the URLs for the pages. Conceptually, a `FancyPage` is equivalent to a `Category` in Oscar, therefore, a `FancyPage` wraps the `Category` model and adds FP-specific behaviour. Therefore, we have to modify Oscar's URLs to replace the category URLs with those for our FP pages. This sounds more complicated than it actually is:

```
from fancypages.app import application as fancypages_app
from fancypages.contrib.oscar_fancypages import views

from oscar.app import Shop
from oscar.apps.catalogue.app import CatalogueApplication


class FancyCatalogueApplication(CatalogueApplication):
    category_view = views.FancyPageDetailView


class FancyShop(Shop):
    catalogue_app = FancyCatalogueApplication()


urlpatterns = patterns('',
    ...
    url(r'', include(FancyShop().urls)),
    ...
    url(r'^', include(fancypages_app.urls)),
)
```

All we are doing here is, replacing the `CategoryView` in Oscar with the `FancyPageDetailView` from OFP, which will display the same details as Oscar's template.

Replacing the home page with a FP page works exactly the same way as described in *Standalone Setup*.

1.4 Running Migrations

Before you are ready to go, make sure that you've applied the migrations for FP and OFP (depending on your setup) by running:

```
$ ./manage.py migrate
```

CHAPTER 2

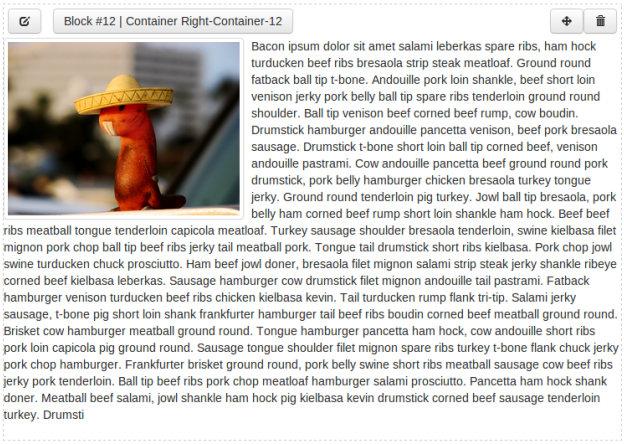
Basic Concepts

Before we get started with the installation and setup of *fancypages* (FP), let's take a look a the underlying concepts to understand the components it's assembled of. There are three major components in FP that you should know about are *containers*, *blocks* and *pages*.

Containers and blocks are strongly related and form the most important part of fancypages. As the name suggests, a container is an object that holds an arbitrary number of other objects, in our case the *blocks*. These, in turn, are the basic building blocks of FP.

2.1 Blocks

A content block in FP is a Django model that defines specific content that is editable by the user. This can be a simple `CharField`, an `ImageField` or any model field really. These fields are then editable on the front-end through the editor panel.



In addition to the actual content, each block also defines a template that provides that specific layout for this content block. This works similar to views in Django itself where `template_name` on a `TemplateView` can be used to specify the path to a template file relative to the template directory.

A simple content block providing editable rich text might look similar to this:

```
from fancypages.models.blocks import ContentBlock
from fancypages.library import register_content_block

@register_content_block
class TextBlock(ContentBlock):
```

```
    name = _("Text")
    code = 'text'
    template_name = "fancypages/blocks/textblock.html"

    text = models.TextField(_("Text"), default="Your text goes here.")
```

2.2 Containers

To be able to place content block on a page we need to be able to specify where these blocks can be placed on any given page. That's where containers come in. They are basically placeholders in a template file that define where blocks can be added. They are agnostic of their surrounding and simply expand to the maximum area they can occupy within their enclosing HTML element.

Adding a container to a template will make sure that all block added to this container are rendered whenever the template is rendered. Additionally, they are an indication for FP to display the editor panel to users with the right permissions.

Let's look at how you can define containers in your own templates to get a better idea of how they work. FP knows *two different types* of containers and to illustrate the difference and how you can use them we'll look at the following two examples:

1. The **named container** behaves similar to a variable. You specify a name for you container and wherever you use that name in a template the **same** container including all it's blocks is rendered. A simple example could look something like this:

   ```
   {% load fp_container_tags %}
   ...
   <div class="row">
       {% fp_container my-first-container %}
   </div>
   ...
   ```

   This defines a container named `my-first-container`. When you now go to the page that uses this template, the FP editor panel will be displayed (assuming you are logged in as admin user) and you can add blocks to the container. *Note:* You don't have to create the container yourself. The first time the template is rendered the container is created if it doesn't already exist.

2. The **model-related container** is similar in that it has to have a name. The difference is that we attach this container to a specific model instance by passing it into the template tag. Where does that make sense? Let's look at a simple blog app that contains a template for the detail page of each blog post. If you use a regular *named container* the exact same container with the exact same blocks will show up on **all** blog post pages. But that might not be what you want. If you want to be able to enhance the content of block posts individually you have to use a *model-related container* and attach it to the individual blog post. In a template it would look like this (assuming the blog post instance in the context is called `post`):

   ```
   {% load fp_container_tags %}
   ...
   <h1>{{ post.title }}</h1>

   {% fp_object_container blog-post-container post %}

   <div class="content">{{ post.content|safe }}</div>
   ```

CHAPTER 3

Content Blocks

3.1 Form Block

Generating freely configurable form on the front-end is difficult to get right and usually has drawbacks in terms of validation of the content that is passed through. The form block in FP provides a more restrictive way of defining form by only allowing pre-configurated forms to be selected. All available forms are configurated as settings and can be selected when editing the form block.

3.1.1 Defining Selectable Forms

The *FormBlock <fancypages.models.blocks.content.FormBlock>* uses a setting named `FP_FORM_BLOCK_CHOICES` to specify all available forms with their respective *action* URLs and (optionally) a template to render the form. An example would be:

```
FP_FORM_BLOCK_CHOICES = {
    'contact-us': {
        'name': "Contact Us Form",
        'form': 'contact_us.forms.ContactUsForm',
        'url': 'contact-us',
        'template_name': 'contact_us/contact_us_form.html',
    }
}
```

The key `contact-us` is the unique identifier used to store the form used in a form block. This value will be stored on the block model. Each of the keys has to provide at least `name`, `form` and `url` in its configuration.

`name`	The name displayed in the form block selection.
`form`	Dotted path to a form class subclassing `fancypages.form.BaseBlockForm`.
`url`	The URL used in th‘‘action‘‘ attribute of the form. This can be a Django URL pattnern name that can be used in `reverse` or a fully qualified URL including a valid scheme.

In addition to these mandatory options, a `template_name` can be specified that will be used instead of the default form template `fancypages/templates/fancypages/blocks/formblock.html`.

CHAPTER 4

Recipes

4.1 Create a Custom Template Block

Start off by creating a new app in your project, e.g. a `blocks` app. Conent blocks in fancypages are basically Django models that require a few additional attributes and definitions.

Let's assume we want to create a simple widget that displays a custom template without providing any additional data that can be edited. All we need to do is define the following model:

```
from fancypages.models.blocks import ContentBlock
from fancypages.library import register_content_block

@register_content_block
class MyTemplateBlock(ContentBlock):
    name = _("My template")
    code = u'my-template'
    group = u'My Blocks'
    template_name = u'blocks/my_template_block.html'

    def __unicode__(self):
        return self.name
```

The first three attributes `name`, `code` and `group` are important and have to be specified on every new content block.

`name`	Display name of the content block
`code`	**Unique** code for the block to be identified by
`group`	Blocks can be grouped by using the same group name here

4.2 Changing Rich Text Editor

Fancypages uses Trumbowyg as the rich text editor by default. It is an open-source tool licensed under the MIT license and provides the basics required for rich text editing in the fancypages editor panel.

Alternatively, other rich text editors can be used instead. Fancypages comes with an alternative setup for Froala. Although Froala is a more comprehensive editor, it is not the default because of its license. It is only free to use for personal and non-profit project, commercial projects require a license.

4.2.1 Switching to Froala

The Froala editor can be enabled in three simple steps but before we get started, you have to download Froala from their website and unpack it.

Step 1: Copy the files `froala_editor.min.js` and `froala_editor.min.css` into your project's static file directory. This would usually be something like `static/libs/froala/`.

Step 2: Override the fancypages partials that define JavaScript and CSS files required to the editor panel. Copy the following three files from fancypages into your template directory:

```
templates/fancypages/editor/head.html
templates/fancypages/editor/partials/cdn_scripts.html
templates/fancypages/editor/partials/extrascripts.html
```

Remove the `trumbowyg.css` and `trumbowyg.min.js` files forom the `head.html` and `extrascripts.html` respectively and replace them with the corresponding CSS and JavaScript files for Froala. You'll also need to add Font Awesome to the `cdn_scripts.html`, e.g.:

```
<link href="//maxcdn.bootstrapcdn.com/font-awesome/4.1.0/css/font-awesome.min.css" rel="stylesheet">
```

Step 3: Set the rich text editor to `Froala` when initialising the Fancypages app in the editor panel by overwriting `templates/fancypages/editor/body.html` and starting the application using:

```
$(document).ready(function(){
    FancypageApp.start({'editor': 'froala'});
});
```

The rich text editors in the editor panel should now use Froala instead of the default Trumbowyg editor.

4.2.2 Using a custom editor

You can also use your favourite editor by adding all the JavaScript and CSS requirements similar to the Froala example and providing a Backbone/Marionette view class that provides the necessary initialisations. For an example, take a look at the `FroalaEditor` and `TrumbowygEditor` views in the Marionette views for Fancypages. To enable your editor set the `editor` option for the Fancypages app to `custom` and pass you view class as the `editorView`. An example might look like this:

```
$(document).ready(function(){
    FancypageApp.start({
        editor: 'custom',
        editorView: myownjavascript.Views.FavouriteEditor
    });
});
```

4.3 Customising Rich Text Editor

In addition to choose the editor you want to use for rich text editing, you can also configure the way the editor behaves by passing editor-specific options to the fancypages app when it is initialised in the `fancypages/editor/body.html` template. Simply overwrite the template and update the script section at the bottom with something like this:

```
.. code-block:: javascript

    $(document).ready(function(){
```

```
    FancypageApp.start({ editor: 'trumbowyg', editorOptions: {
            fullscreenable: true btns: [
              'viewHTML', 'l', 'formatting', 'l', 'link', 'l', 'insertImage', 'l', 'insertHorizontal-
              Rule'
            ]
        },
    });
});
```

CHAPTER 5

Contributing

5.1 Integration Tests

The test that are based on Django's `LiveServerTestCase` are considered integration test. We use splinter that runs on top of Selenium for that. All integration tests are based on the `SplinterTestCase <fancypages.test.testcases.SplinterTestCase>` and carry the py.test marker `integration` that is excluded from the default running of tests. To run integration tests run:

```
py.test -m integration
```

There are a couple of settings that allow changes to the way selenium/splinter is run. Setting `SPLINTER_WEBDRIVER` to a valid Selenium webdriver allows for changing the default webdriver to whatever you want (assuming the required driver is installed). Another helpful variable is `SPLINTER_DEBUG` which prevents the Selenium browser from being closed after finishing a test run so you can inspect the state of the site. Using both settings a test could be run like this:

```
SPLINTER_DEBUG=true SPLINTER_WEBDRIVER=chrome py.test -m integration
```

CHAPTER 6

API Reference

fancypages.**get_fancypages_paths**(*path*, *use_with_oscar=False*)
Get absolute paths for *path* relative to the project root

6.1 Models

class fancypages.abstract_models.**AbstractPageGroup**(**args*, ***kwargs*)
A page group provides a way to group fancy pages and retrieve only pages within a specific group.

Parameters

- **uuid** (*ShortUUIDField*) – Unique id
- **name** (*CharField*) – Name
- **slug** (*SlugField*) – Slug

class fancypages.abstract_models.**AbstractPageNode**(**args*, ***kwargs*)
Define the tree structure properties of the fancy page. This is a separate abstract class to make sure that it can be easily replaced by another tree handling library or none if needed.

Parameters

- **path** (*CharField*) – Path
- **depth** (*PositiveIntegerField*) – Depth
- **numchild** (*PositiveIntegerField*) – Numchild
- **name** (*CharField*) – Name
- **slug** (*SlugField*) – Slug
- **image** (*ImageField*) – Image
- **description** (*TextField*) – Description

move(*target*, *pos=None*)
Moves the current node and all its descendants to a new position relative to another node.

See https://tabo.pe/projects/django-treebeard/docs/1.61/api.html

6.2 Mixins

class fancypages.mixins.**TemplateNamesModelMixin**

Mixin that provides a generalised way of generating template names for a a Django model. It uses relies on at least one of two class attributes: `template_name` and `default_template_names` to generate a list of templates to look for according to Django's rules for template lookup.

The `template_name` attribute specifies a specific template to be used when rendering this model. If this attribute is not `None` it takes precedence over all other template names and therefore will appear at the top of the templates. The `default_template_names` is a list of template names that provides default behaviour and a fallback in case no template name is given or it can't be found by the template engine. Specifying both a template name and a list of default templates will result in a list of template names similar to this:

```
>>> from django.db import models
>>> from fancypages.mixins import TemplateNamesModelMixin
>>>
>>> class Container(TemplateNamesModelMixin, models.Model):
...     template_name = 'container.html'
...     default_template_names = ['default_container.html']
...
...     class Meta: app_label = 'fakeapp'
>>>
>>>
>>> c = Container()
>>> c.get_template_names()
['container.html', 'default_container.html']
```

Each template name provided in `template_name` or `default_template_names` is also run through standard Python string formatting providing the model name as provide in `self._meta.module_name` which allows parametrized template names. Additional keyword arguments can be passed into `get_template_names` to provide additional formatting keywords. Here's an example:

```
>>> class Pony(TemplateNamesModelMixin, models.Model):
...     template_name = 'container_{module_name}_{magic}.html'
...
...     class Meta: app_label = 'fakeapp'
>>>
>>> c = Pony()
>>> c.get_template_names(magic='rainbow')
['container_pony_rainbow.html']
```

In addition to the above, language-specific template names are added if the model has a `language_code` attribute specified. This allows different templates for different languages to customise the appearance of the rendered data based on the language. This makes sense for langugages such as Persian where the reading direction is from left to right. Language-specific templates have the corresponding language code added as a suffix to the filename just before the file extension. In cases such as English where the language is split up into different regions such as British (en-gb) and American English (en-us) a generic template for 'en' is added as well. For a British language code this will be the list of templates:

```
>>> class Pony(TemplateNamesModelMixin, models.Model):
...     template_name = '{module_name}.html'
...     language_code = models.CharField(max_length=6)
...
...     class Meta: app_label = 'fakeapp'
>>>
>>> c = Pony(language_code='en-gb')
>>> c.get_template_names()
['pony_en-gb.html', 'pony_en.html', 'pony.html']
```

get_template_names(***kwargs*)

Get a list of template names in order of precedence as used by the Django template engine. Keyword argument passed in are used during string formatting of the template names. This fails silently if a argument is specified in a template name but is not present in `kwargs`.

Rtype list A list of template names (unicode).

6.3 Blocks

class fancypages.models.blocks.content.**CarouselBlock**(**args*, ***kwargs*)

CarouselBlock(id, uuid, container_id, display_order, contentblock_ptr_id)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **link_url_1** (*CharField*) – Link url 1
- **link_url_2** (*CharField*) – Link url 2
- **link_url_3** (*CharField*) – Link url 3
- **link_url_4** (*CharField*) – Link url 4
- **link_url_5** (*CharField*) – Link url 5
- **link_url_6** (*CharField*) – Link url 6
- **link_url_7** (*CharField*) – Link url 7
- **link_url_8** (*CharField*) – Link url 8
- **link_url_9** (*CharField*) – Link url 9
- **link_url_10** (*CharField*) – Link url 10
- **image_1_id** (*AssetKey*) – Image 1
- **image_2_id** (*AssetKey*) – Image 2
- **image_3_id** (*AssetKey*) – Image 3
- **image_4_id** (*AssetKey*) – Image 4
- **image_5_id** (*AssetKey*) – Image 5
- **image_6_id** (*AssetKey*) – Image 6
- **image_7_id** (*AssetKey*) – Image 7
- **image_8_id** (*AssetKey*) – Image 8
- **image_9_id** (*AssetKey*) – Image 9
- **image_10_id** (*AssetKey*) – Image 10

class fancypages.models.blocks.content.**ContentBlock**(**args*, ***kwargs*)
ContentBlock(id, uuid, container_id, display_order)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order

class fancypages.models.blocks.content.**FormBlock**(**args*, ***kwargs*)
FormBlock(id, uuid, container_id, display_order, contentblock_ptr_id, form_selection)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **form_selection** (*CharField*) – Form selection

class fancypages.models.blocks.content.**ImageAndTextBlock**(**args*, ***kwargs*)
ImageAndTextBlock(id, uuid, container_id, display_order, contentblock_ptr_id, title, alt_text, link, image_asset_id, text)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **title** (*CharField*) – Image title
- **alt_text** (*CharField*) – Alternative text
- **link** (*CharField*) – Link url
- **image_asset_id** (*AssetKey*) – Image asset
- **text** (*TextField*) – Text

class fancypages.models.blocks.content.**ImageBlock**(**args*, ***kwargs*)
ImageBlock(id, uuid, container_id, display_order, contentblock_ptr_id, title, alt_text, link, image_asset_id)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr

- **title** (*CharField*) – Image title
- **alt_text** (*CharField*) – Alternative text
- **link** (*CharField*) – Link url
- **image_asset_id** (*AssetKey*) – Image asset

class fancypages.models.blocks.content.**PageNavigationBlock**(**args*, ***kwargs*)
PageNavigationBlock(id, uuid, container_id, display_order, contentblock_ptr_id, depth, origin)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **depth** (*PositiveIntegerField*) – Navigation depth
- **origin** (*CharField*) – Navigation origin

class fancypages.models.blocks.content.**TextBlock**(**args*, ***kwargs*)
TextBlock(id, uuid, container_id, display_order, contentblock_ptr_id, text)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **text** (*TextField*) – Text

class fancypages.models.blocks.content.**TitleTextBlock**(**args*, ***kwargs*)
TitleTextBlock(id, uuid, container_id, display_order, contentblock_ptr_id, title, text)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **title** (*CharField*) – Title
- **text** (*TextField*) – Text

class fancypages.models.blocks.layouts.**FourColumnLayoutBlock**(**args*, ***kwargs*)
FourColumnLayoutBlock(id, uuid, container_id, display_order, contentblock_ptr_id)

Parameters

- **id** (*AutoField*) – Id

- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr

class fancypages.models.blocks.layouts.**HorizontalSeparatorBlock**(**args*, ***kwargs*)
HorizontalSeparatorBlock(id, uuid, container_id, display_order, contentblock_ptr_id)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr

class fancypages.models.blocks.layouts.**TabBlock**(**args*, ***kwargs*)
TabBlock(id, uuid, container_id, display_order, contentblock_ptr_id)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr

class fancypages.models.blocks.layouts.**ThreeColumnLayoutBlock**(**args*, ***kwargs*)
ThreeColumnLayoutBlock(id, uuid, container_id, display_order, contentblock_ptr_id)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr

class fancypages.models.blocks.layouts.**TwoColumnLayoutBlock**(**args*, ***kwargs*)
TwoColumnLayoutBlock(id, uuid, container_id, display_order, contentblock_ptr_id, left_width)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **left_width** (*PositiveIntegerField*) – Left width

left_span
Returns the bootstrap span class for the left container.

right_span
Returns the bootstrap span class for the left container.

class fancypages.models.blocks.social.**TwitterBlock**(**args*, ***kwargs*)
TwitterBlock(id, uuid, container_id, display_order, contentblock_ptr_id, username, max_tweets)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **username** (*CharField*) – Twitter username
- **max_tweets** (*PositiveIntegerField*) – Maximum tweets

class fancypages.models.blocks.social.**VideoBlock**(**args*, ***kwargs*)
VideoBlock(id, uuid, container_id, display_order, contentblock_ptr_id, source, video_code)

Parameters

- **id** (*AutoField*) – Id
- **uuid** (*ShortUUIDField*) – Unique id
- **container_id** (*ForeignKey*) – Container
- **display_order** (*PositiveIntegerField*) – Display order
- **contentblock_ptr_id** (*OneToOneField*) – Contentblock ptr
- **source** (*CharField*) – Video type
- **video_code** (*CharField*) – Video code

6.4 Template Tags

fancypages.templatetags.fp_container_tags.**fp_block_container**(*parser*, *token*)
Template tag for convenience to use within templates for e.g. layout blocks where the container is assigned to the widget rather then the object in the context. The same could be achieved using:

```
{% fp_object_container some-name fp_block %}
```

fancypages.templatetags.fp_container_tags.**fp_object_container**(*parser*, *token*)
Template tag specifying a fancypages container to be rendered in the template at the given location. It takes up to three arguments. The first argument is the name of the container which is mandatory. The object that this tag is attached to is the second argument and is optional. If it is not specified, the 'object' variable in the current context is used. The third argument is an optional language code that specify the language that should be used for the container. Without a language code specified, the current language is retrieved using Django's internationalisation helpers.

Valid template tags are:

```
{% fp_object_container container-name %}
```

and with a specific object:

```
{% fp_object_container container-name my_object %}
```

and with a language code:

```
{% fp_object_container container-name my_object "de-de" %}
{% fp_object_container container-name object_name=my_object language="de-de" %}
```

fancypages.templatetags.fp_container_tags.**parse_arguments**(*parser*, *token*, *params=None*)

Parse positional arguments and keyword arguments into a dictionary for the known arguments given in *params* in the given order. If the number of arguments in *token* is greater than the known number of arguments, a `TemplateSyntaxError` is raised. The same is true if no tokens are provided.

Parameters

- **parser** – Parser as passed into the template tag.
- **token** – Token object as passed into the template tag.
- **params** – List of expected arguments in the order in which they appear when not using keyword arguments. Default to ['container_name', 'object_name', 'language'].

Rtype dict containing the parsed content for the arguments above.

6.5 Editor Middleware

fancypages.middleware.**replace_insensitive**(*string*, *target*, *replacement*)

Similar to string.replace() but is case insensitive Code borrowed from: http://forums.devshed.com/python-programming-11/case-insensitive-string-replace-490921.html

CHAPTER 7

Indices and tables

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

f

V