django-fluent-contents Documentation

Release 2.0.7

Diederik van der Boor

Oct 24, 2021

Contents

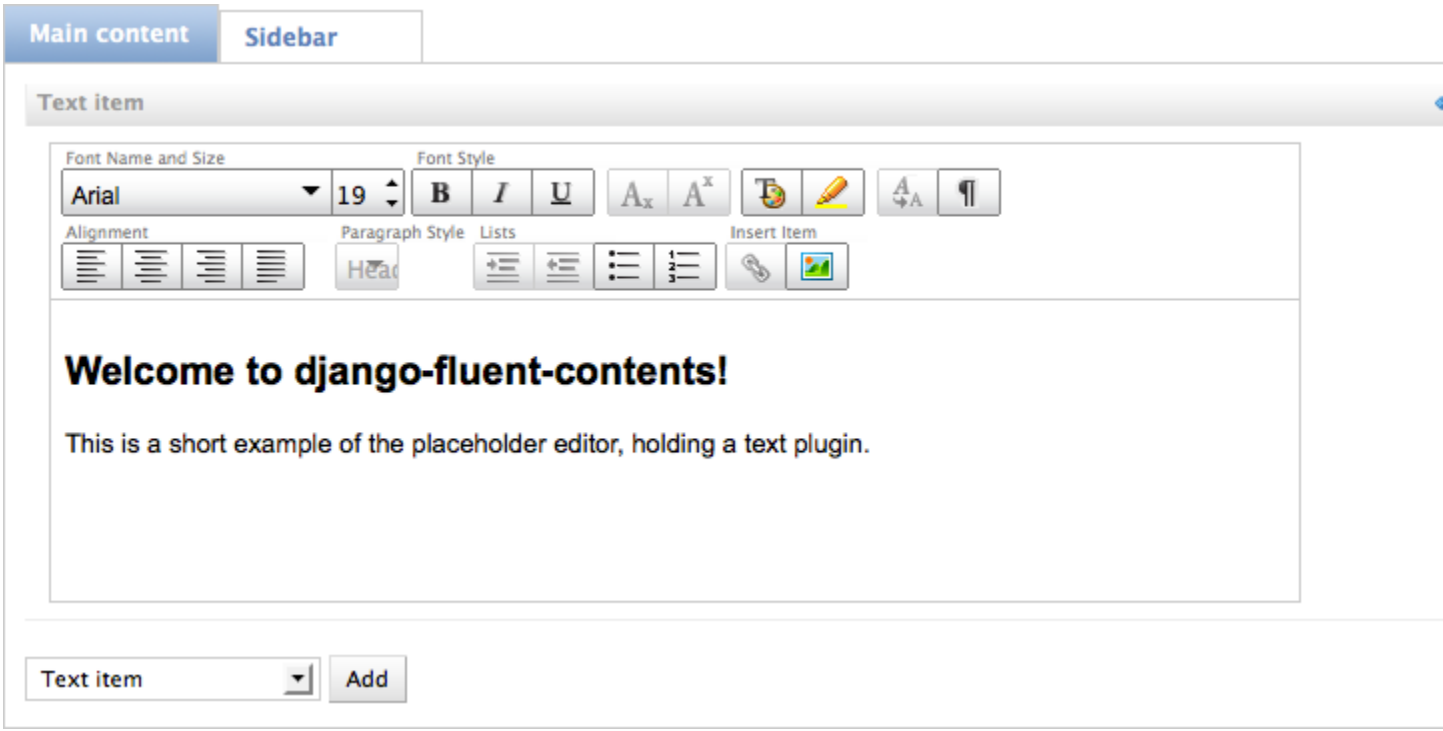
This documentation covers the latest release of django-fluent-contents, a collection of applications to build an end user CMS for the Django administration interface. django-fluent-contents includes:

- A `PlaceholderField` to display various content on a model.
- A `PlaceholderEditorAdmin` to build CMS interfaces.
- A default set of plugins to display WYSIWYG content, reStructuredText, highlighted code, Gist snippets and more.
- an extensible plugin API.

To get up and running quickly, consult the *quick-start guide*. The chapters below describe the configuration of each specific plugin in more detail.

CHAPTER 1

Preview



CHAPTER 2

Getting started

2.1 Quick start guide

2.1.1 Installing django-fluent-contents

The base installation of django-fluent-contents requires Django version 1.3 or higher and django-polymorphic 0.2 or higher.

The package can be installed using:

```
pip install django-fluent-contents
```

The additional plugins may add additional requirements; the plugins will warn about then when they are used for the first time. For optional dependency management, it is strongly recommended that you run the application inside a *virtualenv*.

2.1.2 Basic setup

Next, create a project which uses the module. The basic module can be installed and optional plugins can be added:

```
INSTALLED_APPS += (
    'fluent_contents',
    'django_wysiwyg',

    # And optionally add the desired plugins:
    'fluent_contents.plugins.text',                # requires django-wysiwyg
    'fluent_contents.plugins.code',                # requires pygments
    'fluent_contents.plugins.gist',
    'fluent_contents.plugins.googledocsviewer',
    'fluent_contents.plugins.iframe',
    'fluent_contents.plugins.markup',
    'fluent_contents.plugins.rawhtml',
)
```

Since some extra plugins are used here, make sure their applications are installed:

```
pip install django-fluent-contents[text,code]
```

Note: Each plugin is optional. Only the `fluent_contents` application is required, allowing to write custom models and plugins. Since a layout with the *text* and *code* plugin form a good introduction, these are added here.

Afterwards, you can setup the database:

```
./manage.py syncdb
```

2.1.3 Displaying content

Finally, it needs a model or application that displays the content. The most simply way, is adding a *PlaceholderField* to a model:

```
class Article(models.Model):
    title = models.CharField("Title", max_length=200)
    content = PlaceholderField("article_content")

    class Meta:
        verbose_name = "Article"
        verbose_name_plural = "Articles"

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

Make sure the admin screen the *PlaceholderFieldAdmin* class. This makes sure additional inlines are added the the admin screen:

```
class ArticleAdmin(PlaceholderFieldAdmin):
    pass

admin.site.register(Article, ArticleAdmin)
```

No extra configuration is required, the field will simply blend in with the rest of the form fields. Gven that the article is displayed by a template (i.e. `article/details.html`) it can use the `fluent_contents_tags` to display the contents:

```
{% load fluent_contents_tags %}
{% render_placeholder article.content %}
```

That's it!

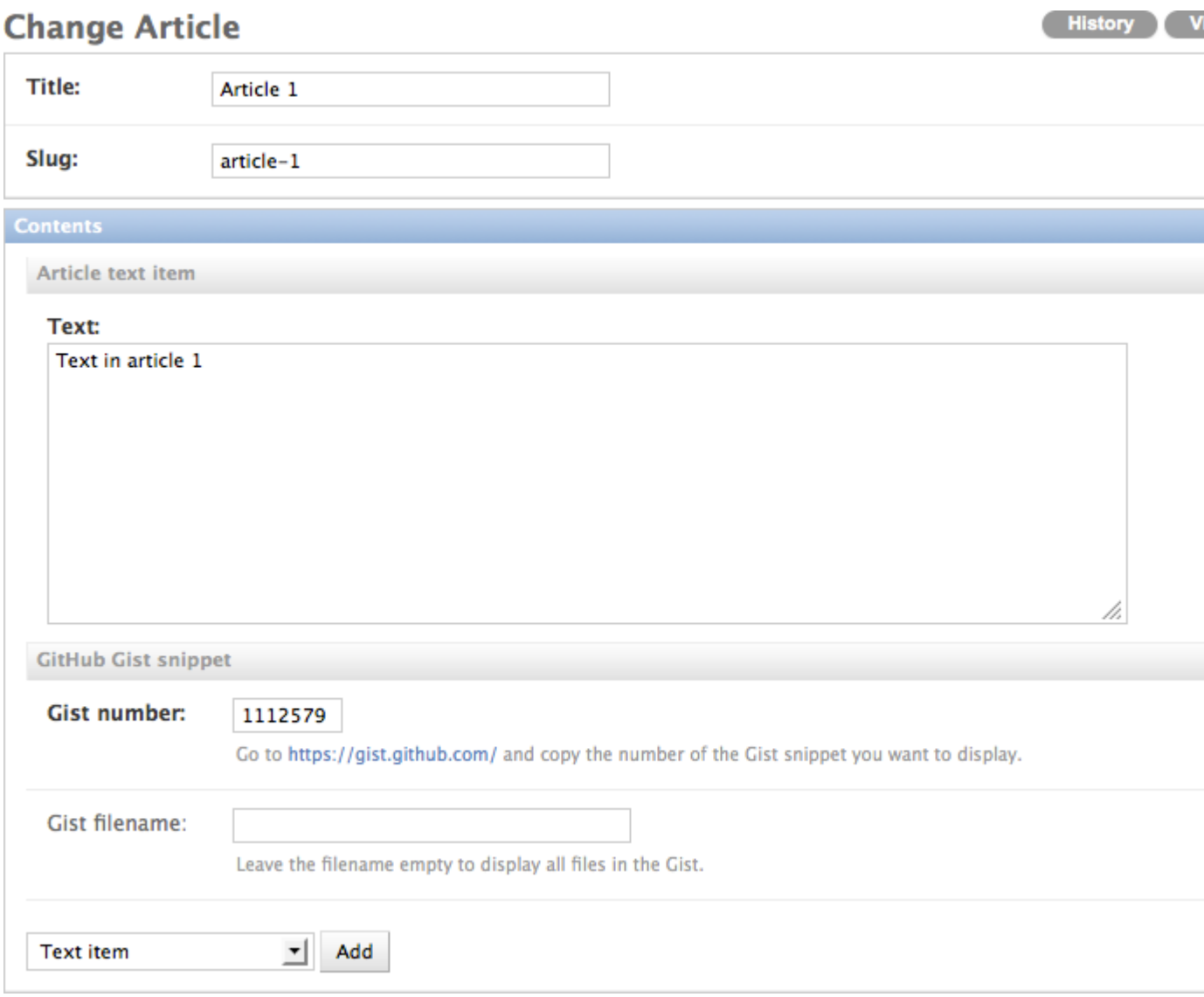
2.1.4 Fieldset layout

With a small change in the `fieldsets` configuration, the admin interface could look like this:

When the placeholder is used in a separate `fieldset` that has a `plugin-holder` class name, the field will be displayed without a label in front of it:

```
class ArticleAdmin(PlaceholderFieldAdmin):
    prepopulated_fields = {'slug': ('title',)}

    fieldsets = (
        (None, {
            'fields': ('title', 'slug'),
        }),
        ("Contents", {
            'fields': ('content',),
            'classes': ('plugin-holder',),
        })
    )
```



2.1.5 Optional features

To add even more plugins, use:

```
INSTALLED_APPS += (
    'fluent_contents',

    # Dependencies for plugins:
    'disqus',
    'django.contrib.comments',
    'django_wysiwyg',
    'form_designer',

    # All plugins:
    'fluent_contents.plugins.text',                # requires django-wysiwyg
    'fluent_contents.plugins.code',                # requires pygments
    'fluent_contents.plugins.gist',
```

(continues on next page)

(continued from previous page)

```
    'fluent_contents.plugins.googledocsviewer',
    'fluent_contents.plugins.iframe',
    'fluent_contents.plugins.markup',
    'fluent_contents.plugins.rawhtml',

    'fluent_contents.plugins.commentsarea',        # requires django.contrib.comments␣
↪+ templates
    'fluent_contents.plugins.disquswidgets',       # requires django-disqus + DISQUS_
↪API_KEY
    'fluent_contents.plugins.formdesignerlink',    # requires django-form-designer-ai
)

DISQUS_API_KEY = '...'
DISQUS_WEBSITE_SHORTNAME = '...'

FLUENT_MARKUP_LANGUAGE = 'reStructuredText'        # can also be markdown or textile
```

Most of the features are glue to existing Python or Django modules, hence these packages need to be installed:

- django-wysiwyg (for the *text* plugin)
- Pygments (for the *code* plugin)
- docutils (for the *markup* plugin)
- django-disqus (for the *disquscommentsarea* plugin)
- django-form-designer-ai (for the *formdesignerlink* plugin)

They can be installed using:

```
pip install django-fluent-contents[text,code,markup,disquscommentsarea,
↪formdesignerlink]
```

The reason that all these features are optional is make them easily swappable for other implementations. You can use a different comments module, or invert new content plugins. It makes the CMS configurable in the way that you see fit.

Some plugins, like the *commentsarea* based on django.contrib.comments, might make a bad first impression because they have no default layout. This turns out however to be by design, to make them highly adaptable to your design and requirements.

2.1.6 Creating a CMS system

The django-fluent-contents package also offers a `PlaceholderEditorAdmin` class which allows CMS-developers to display the content plugins at various locations of a CMS page. For more information, see the *Creating a CMS system*.

2.1.7 Testing your new shiny project

Congrats! At this point you should have a working installation. Now you can just login to your admin site and see what changed.

2.1.8 Production notes

When deploying the project to production, enable the following setting:

```
FLUENT_CONTENTS_CACHE_PLACEHOLDER_OUTPUT = True
```

This improves the performance of the *template tags*.

2.2 Configuration

A quick overview of the available settings:

```
FLUENT_CONTENTS_CACHE_OUTPUT = True                # disable sometimes for development
FLUENT_CONTENTS_CACHE_PLACEHOLDER_OUTPUT = False  # enable for production

FLUENT_CONTENTS_PLACEHOLDER_CONFIG = {
    'slot_name': {
        'plugins': ('TextPlugin', 'PicturePlugin', 'OEmbedPlugin',
↪'SharedContentPlugin', 'RawHtmlPlugin',)
    },
    'shared_content': {
        'plugins': ('TextPlugin',)
    }
    'blog_contents': {
        'plugins': ('TextPlugin', 'PicturePlugin',)
    }
}

FLUENT_CONTENTS_DEFAULT_LANGUAGE_CODE = LANGUAGE_CODE  # e.g. "en"
FLUENT_CONTENTS_FILTER_SITE_ID = True
```

2.2.1 Rendering options

FLUENT_CONTENTS_CACHE_OUTPUT

Typically, most web site pages look the same for every visitor. Hence, this module takes the following default policies:

- the output of plugins is cached by default.
- the cache is refreshed when a staff member updates saves changes in in the Django admin.
- each `ContentPlugin` can specify custom *caching settings* to influence this.
- caching is even enabled for development (to test production setup), but changes in templates are detected.

Caching greatly improves the performance of the web site, as very little database queries are needed. If you have a few custom plugins that should not be cached per request (e.g. a contact form), update the *output caching* settings of that specific plugin.

However, in case this is all too complex, you can disable the caching mechanism entirely. The caching can be disabled by setting this option to `False`.

FLUENT_CONTENTS_CACHE_PLACEHOLDER_OUTPUT

This setting can be used to enable the output of entire placeholders. Preferably, this should be enabled for production.

This feature is impractical in development, because any changes to code or templates won't appear. Hence, the default value is `False`.

Without this setting, only individual content plugins are cached. The surrounding objects will still be queried from the database. That includes:

- The *Placeholder*
- Any `SharedContent` model.
- The base class of each *ContentItem* model.

FLUENT_CONTENTS_DEFAULT_LANGUAGE_CODE

The default language for items. This value is used as:

- The fallback language for rendering content items.
- The default language for the `ContentItem.language_code` model field when the parent object is not translatable.
- The initial migration of data when you migrate a 0.9 project to v1.0

When this value is not defined, the following settings will be tried:

- `FLUENT_DEFAULT_LANGUAGE_CODE`
- `PARLER_DEFAULT_LANGUAGE_CODE`
- `LANGUAGE_CODE`

2.2.2 HTML Field Settings

FLUENT_TEXT_CLEAN_HTML

If `True`, the HTML content returned by the *PluginHtmlField* will be rewritten to be well-formed using html5lib.

FLUENT_TEXT_SANITIZE_HTML

if `True`, unwanted HTML tags will be removed server side using html5lib.

FLUENT_TEXT_POST_FILTERS, FLUENT_TEXT_PRE_FILTERS

These settings accept a list of callable function names, which are called to update the HTML content. For example:

```
FLUENT_TEXT_PRE_FILTERS = (
    'myapp.filters.cleanup_html',
    'myapp.filters.validate_html',
    'fluent_contents.plugins.text.filters.smartypants.smartypants_filter',
)

FLUENT_TEXT_POST_FILTERS = (
    'fluent_contents.plugins.text.filters.softhypen.softhypen_filter',
)
```

The pre-filters and post-filters differ in a slight way:

- The *pre*-filters updates the source text, so they should be idempotent.

- The *post*-filters only affect the displayed output, so they may manipulate the HTML completely.

See also:

The *HTML Text Filters* documentation.

2.2.3 Admin settings

FLUENT_CONTENTS_PLACEHOLDER_CONFIG

This setting limits which plugins can be used in a given placeholder slot. For example, a "homepage" slot may include add a "slideshow", while the "sidebar" slot can be limited to other elements. By default, all plugins are allowed to be used everywhere.

The list of plugins can refer to class names, or point to the actual classes themselves. When a list of plugins is explicitly passed to a `PlaceholderField`, it overrides the defaults given via the settings file.

2.2.4 Advanced admin settings

FLUENT_CONTENTS_FILTER_SITE_ID

By default, contents is displayed for the current site only.

By default, each `Site` model has it's own contents. This enables the multi-site support, where you can run multiple instances with different sites. To run a single Django instance with multiple sites, use a module such as django-multisite.

You can disable it using this by using:

```
FLUENT_PAGES_FILTER_SITE_ID = False
```

This completely disables the multisite support, and should only be used as last resort. The `SharedContent` model is unsplit, making all content available for all sites.

Note: The "Shared Content" module also provides a "Share between all sites" setting to share a single object explicitly between multiple sites. Enable it using the *FLUENT_SHARED_CONTENT_ENABLE_CROSS_SITE* setting. Using that feature is recommended above disabling multisite support completely.

2.3 The template tags

The template tags provide the rendering of placeholder content. Load the tags using:

```
{% load fluent_contents_tags %}
```

To render a placeholder for a given object, use:

```
{% render_placeholder myobject.placeholder_field %}
```

2.3.1 CMS Page placeholders

To define the placeholders for a *cms* page, use:

```
{% page_placeholder currentpage "slotname" %}
```

If the currentpage variable is named `page`, it can be left out.

Using a custom template

To customize the placeholder contents, a template can be specified:

```
{% page_placeholder currentpage "slotname" template="mysite/parts/slot_placeholder.
↪html" %}
```

That template should loop over the content items, and include additional HTML. For example:

```
{% for contentitem, html in contentitems %}
  {% if not forloop.first %}<div class="splitter"></div>{% endif %}
  {{ html }}
{% endfor %}
```

The following variables are available:

- `contentitems` - a list with:
- the *ContentItem* model. You can access `contentitem.plugin.name` to read the actual plugin name. The model itself is generally not downcasted to the actual model.
- the rendered HTML code
- `parent_object` - the parent object, this may be `None` if `render_items()` was used instead of *render_placeholder()*.

Note: When a template is used, the system assumes that the output can change per request. Hence, even though *FLUENT_CONTENTS_CACHE_PLACEHOLDER_OUTPUT* may be set, but the final merged output will no longer be cached. Add `cachable=1` to enable output caching for templates too.

The output of individual items will always be cached, as that is subject to the *FLUENT_CONTENTS_CACHE_OUTPUT* setting.

Admin Meta information

Extra meta information can be provided for the admin interface:

```
{% page_placeholder currentpage "slotname" title="Tab title" role="main %}
```

The metadata can be extracted with the *PagePlaceholderNode* class, and *fluent_contents.analyzer* module.

Fallback languages

New in version 1.0: For multilingual sites, the contents of the active translation will be displayed only. To render the fallback language for empty placeholders, use the `fallback` parameter:

```
{% page_placeholder currentpage "slotname" fallback=1 %}
```

This can be used to display the "english" content everywhere by default for example, until a translator fills the contents of the page. The fallback language is defined in the *FLUENT_CONTENTS_DEFAULT_LANGUAGE_CODE* setting.

2.3.2 Frontend media

To render the CSS/JS includes of content items, use:

```
{% render_content_items_media %}
```

This tag should be placed at the bottom of the page, after all plugins are rendered.

Optionally, specify to render only the CSS or JavaScript content:

```
{% render_content_items_media css %}
{% render_content_items_media js %}
{% render_content_items_media js internal %}
{% render_content_items_media js external %}
```

By adding the `local` or `external` flag, the media files will be split into:

- externally hosted files which should *not* be compressed (e.g. a plugin that includes the Google Maps API).
- locally hosted files which can be compressed.

This way, the contents can be minified too, using django-compressor for example:

```
{% load compress fluent_contents_tags %}

{% render_content_items_media css external %}
{% compress css %}
    {% render_content_items_media css internal %}
{% endcompress %}

{% render_content_items_media js external %}
{% compress js %}
    {% render_content_items_media js local %}
    {% block extra_scripts %}{% endblock %}
{% endcompress %}
```

2.3.3 Note for existing projects

Deprecated since version 1.0: Previously, the template tag library was called *placeholder_tags*. Using the old style import still works. It's recommended to change it:

```
{% load placeholder_tags %}
```

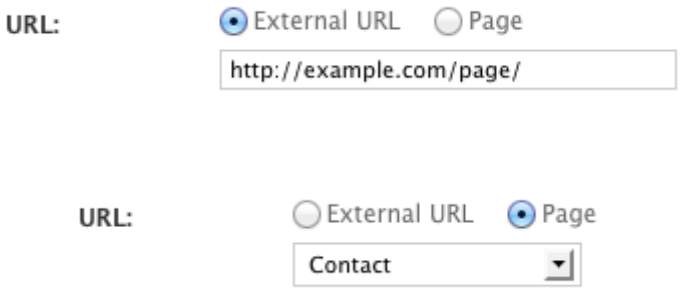
2.4 Optional integration with other packages

New in version 0.9.0.

By default, this package works without including other packages in `INSTALLED_APPS`. There is no requirement to use a specific file manager or WYSIWYG editor in your project. However, the features of this package can be enhanced by installing a few additional applications.

2.4.1 Custom URL fields

By installing django-any-urlfield, the URL fields can select both internal and external URLs:



The *picture plugin* and `PluginUrlField` class use this to display the URL field.

Install it via:

```
pip install django-any-urlfield
```

And include it in the settings:

```
INSTALLED_APPS += (
    'any_urlfield',
    'any_imagefield',
)
```

Each model which provides a `get_absolute_url` field can be used as form option.

For more information, see the documentation of django-any-urlfield.

2.4.2 Custom file and image fields

By installing django-any-imagefield, file and image fields can be replaced with a file browser:

The *picture plugin*, `PluginFileField` and `PluginImageField` class use this to display the file field.

Install it via:

```
pip install django-any-imagefield
pip install -e git+git://github.com/smacker/django-filebrowser-no-grappelli-django14.
↪git#egg=django-filebrowser
```

And include it in the settings:

```
INSTALLED_APPS += (
    'any_imagefield',
    'filebrowser',
)
```

And update `urls.py`:

```
from filebrowser.sites import site as fb_site

urlpatterns += [
    url(r'^admin/filebrowser/', include(fb_site.urls)),
]
```

This package either uses the standard Django `ImageField`, or the image field from any other supported application. When sorl-thumbnail is installed, it will be used; when django-filebrowser-no-grappelli-django14 is available it's used instead.

For more information, see the documentation of django-any-imagefield.

2.4.3 Custom HTML / WYSIWYG fields

The *text plugin* and *`PluginHtmlField`* use django-wysiwyg to display a WYSIWYG editor.

It's possible to switch to any WYSIWYG editor of your choice. The default editor is the YUI editor, because it works out of the box. Other editors, like the CKEditor, Redactor and TinyMCE are supported with some additional configuration.

For more information, see the documentation of django-wysiwyg.

2.4.4 Debug toolbar

During development, the rendered items can be displayed in a special django-debug-toolbar panel. Include `'fluent_contents.panels.ContentPluginPanel'` in the `DEBUG_TOOLBAR_PANELS` setting.

2.5 HTML Text Filters

Plugins that deal with HTML content, can opt-in to the filtering framework. This helps to improve HTML output consistently everywhere.

2.5.1 Bundled filters

This package has a two standard filters included:

- `fluent_contents.plugins.text.filters.smartypants.smartypants_filter()` can be added to *FLUENT_TEXT_POST_FILTERS, FLUENT_TEXT_PRE_FILTERS* to replace regular quotes with curly quotes. It needs smartypants to be installed.
- `fluent_contents.plugins.text.filters.softhypen.softhypen_filter()` can be added to *FLUENT_TEXT_POST_FILTERS, FLUENT_TEXT_PRE_FILTERS* to insert soft-hyphenation characters in the text (`&shy;` as HTML entity). It needs django-softhyphen to be installed.

2.5.2 Filter types

A filter may be included in one of these settings:

- Pre-filters are listed in *FLUENT_TEXT_POST_FILTERS, FLUENT_TEXT_PRE_FILTERS*
- Post-filters are listed in *FLUENT_TEXT_POST_FILTERS, FLUENT_TEXT_PRE_FILTERS*

While some filters could be used in both settings, there is a semantic difference.

Pre-filters

Any changes made by pre-filters affect the original text. These changes are visible in the WYSIWYG editor after saving. Thus, the pre-filter should be idempotent; it should be able to run multiple times over the same content. Typical use cases of a pre-filter are:

- Validate HTML
- Sanitize HTML (using bleach)
- Replace ``"``regular quotes``"`` with curly "smart" quotes.

Post-filters

The changes made by post-filters are *not* stored in the original text, and won't be visible in the WYSIWYG editor. This allows a free-form manipulation of the text, for example to:

- Add soft-hyphens in the code for better line breaking.
- Improve typography, such as avoiding text widows, highlighting ampersands, etc.. (using django-typogrify).
- Highlight specific words.
- Parse "short codes" - if you really must do so. *Please consider short codes a last resort.* It's recommended to create new plugins instead for complex integrations.

Since post-filters never change the original text, any filter function can be safely included as post-filter. When there is an unwanted side-effect, simply remove the post-filter and resave the text.

2.5.3 Creating filters

Each filter is a plain Python function that receives 2 parameters:

- The `ContentItem` model, and
- The HTML text it can update.

For example, see the smartypants filter:

```
from smartypants import smartypants

def smartypants_filter(contentitem, html):
    return smartypants(html)
```

Since the original `ContentItem` model is provided, a filter may read fields such as `contentitem.language_code`, `contentitem.placeholder` or `contentitem.parent` to have context.

The filters may also raise a `ValidationError` to report any errors in the text that should be corrected by the end-user.

2.5.4 Supporting filters in custom plugins

The bundled *text plugin* already uses the filters out of the box. When creating a custom plugin that includes a HTML/WYSIWYG-field, overwrite the `full_clean()` method in the model:

```
from django.db import models
from fluent_contents.extensions import PluginHtmlField
from fluent_contents.models import ContentItem
from fluent_contents.utils.filters import apply_filters


class WysiwygItem(ContentItem):
    html = PluginHtmlField("Text")
    html_final = models.TextField(editable=False, blank=True, null=True)

    def full_clean(self, *args, **kwargs):
        super(TextItem, self).full_clean(*args, **kwargs)
        self.html, self.html_final = apply_filters(self, self.html, field_name='html')
```

The *PluginHtmlField* already provides the standard cleanup and sanitation checks that the *FLUENT_TEXT_CLEAN_HTML* and *FLUENT_TEXT_SANITIZE_HTML* settings enable.

CHAPTER 3

Using the plugins

3.1 Bundled Content Plugins

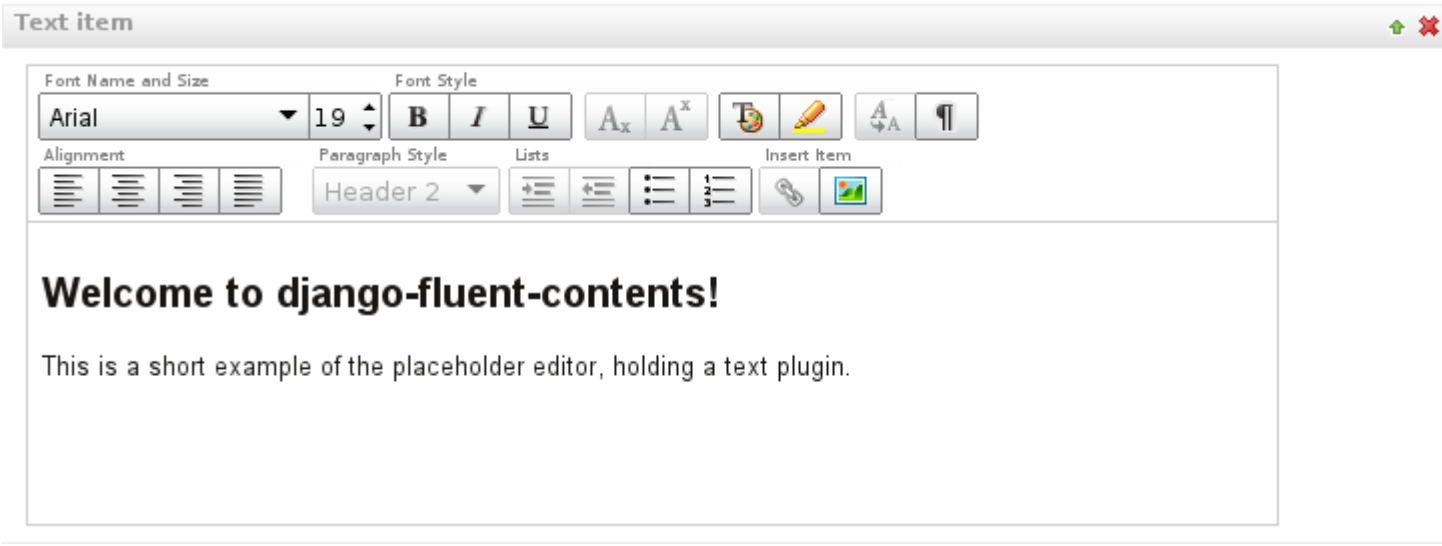
This module ships has a set of plugins bundled by default, as they are useful for a broad range of web sites. The plugin code also serves as an example and inspiration to create your own modules, so feel free browse the source code of them.

The available plugins are:

3.1.1 Standard plugins

The text plugin

The *text* plugin provides a standard WYSIWYG ("What You See is What You Get") editor in the administration panel, to add HTML contents to the page.



It features:

- Fully replaceable/customizable WYSIWYG editor.
- Text pre- and post-processing hooks.
- HTML sanitation/cleanup features.

The plugin is built on top of django-wysiwyg, making it possible to switch to any WYSIWYG editor of your choice. The default editor is the YUI editor, because it works out of the box. Other editors, like the CKEditor, Redactor and TinyMCE are supported with some additional configuration. See the django-wysiwyg documentation for details.

Important: There is no reason to feel constrained to a specific editor. Firstly, the editor can be configured by configuring django-wysiwyg. Secondly, it's possible to create a different text plugin yourself, and let this plugin serve as canonical example.

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-contents[text]
```

This installs the django-wysiwyg package.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.text',
    'django_wysiwyg',
)
```

Configuration

The following settings are available:

```
DJANGO_WYSIWYG_FLAVOR = "yui_advanced"
```

Also check the following settings in the global section:

- *FLUENT_TEXT_CLEAN_HTML*
- *FLUENT_TEXT_SANITIZE_HTML*
- *FLUENT_TEXT_POST_FILTERS, FLUENT_TEXT_PRE_FILTERS*
- *FLUENT_TEXT_POST_FILTERS, FLUENT_TEXT_PRE_FILTERS*

Those settings allow post-processing of the HTML, sanitation and making the HTML well-formed. It can be used for example to fix typography, such as replacing regular quotes with curly quotes.

DJANGO_WYSIWYG_FLAVOR

The `DJANGO_WYSIWYG_FLAVOR` setting defines which WYSIWYG editor will be used. As of django-wysiwyg 0.5.1, the following editors are available:

- **ckeditor** - The CKEditor, formally known as FCKEditor.
- **redactor** - The Redactor editor (requires a license).
- **tinymce** - The TinyMCE editor, in simple mode.
- **tinymce_advanced** - The TinyMCE editor with many more toolbar buttons.
- **yui** - The YAHOO editor (the default)
- **yui_advanced** - The YAHOO editor with more toolbar buttons.

Additional editors can be easily added, as the setting refers to a set of templates names:

- django_wysiwyg/**flavor**/includes.html
- django_wysiwyg/**flavor**/editor_instance.html

For more information, see the documentation of django-wysiwyg about extending django-wysiwyg.

TinyMCE integration example

The WYSIWYG editor can be configured to allow end-users to make minimal styling choices. The following configuration has proven to work nicely for most web sites, save it as `django_wysiwyg/tinymce_advanced/includes.html` in a Django template folder. This code has the following features:

- django-filebrowser integration.
- Unnecessary styling is removed.
- Styling choices are limited to a single "format" box.
- It reads `/static/frontend/css/tinymce.css`, allowing visual consistency between the editor and frontend web site.
- It defines `body_class` so any `.text` CSS selectors that style this plugin output work as expected.

```
{% extends "django_wysiwyg/tinymce/includes.html" %}

<script>{# <- dummy element for editor formatting #}
{% block django_wysiwyg_editor_config %}
```

(continues on next page)

(continued from previous page)

```
    var django_wysiwyg_editor_config = {
        plugins: 'paste,autoresize,inlinepopups',
        strict_loading_mode: true,  // for pre 3.4 releases

        // Behavioral settings
        document_base_url: '/',
        relative_urls: false,
        custom_undo_redo_levels: 10,
        width: '610px',

        // Toolbars and layout
        theme: "advanced",
        theme_advanced_toolbar_location: 'top',
        theme_advanced_toolbar_align: 'left',
        theme_advanced_buttons1: 'styleselect,removeformat,cleanup,|,link,unlink,|,
↪bullist,numlist,|,undo,redo,|,outdent,indent,|,sub,sup,|,image,charmap,anchor,hr,|,
↪code',
        theme_advanced_buttons2: '',
        theme_advanced_buttons3: '',
        theme_advanced_blockformats: 'h3,h4,p',
        theme_advanced_resizing : true,

        // Integrate custom styling
        content_css: "{{ STATIC_URL }}frontend/css/tinymce.css",
        body_class: 'text',

        // Define user configurable styles
        style_formats: [
            {title: "Header 2", block: "h2"},
            {title: "Header 3", block: "h3"},
            {title: "Header 4", block: "h4"},
            {title: "Paragraph", block: "p"},
            {title: "Quote", block: "blockquote"},
            {title: "Bold", inline: "strong"},
            {title: "Emphasis", inline: "em"},
            {title: "Strikethrough", inline: "s"},
            {title: "Highlight word", inline: "span", classes: "highlight"},
            {title: "Small footnote", inline: "small"}
            //{title: "Code example", block: "pre"},
            //{title: "Code keyword", inline: "code"}
        ],

        // Define how TinyMCE formats things
        formats: {
          underline: {inline: 'u', exact: true}
          //strikethrough: {inline: 'del'},
        },
        //inline_styles: false,
        fix_list_elements: true,
        keep_styles: false,

        // Integrate filebrowser
        file_browser_callback: 'djangoFileBrowser'
    };

    function djangoFileBrowser(field_name, url, type, win) {
        var url = "{% url 'filebrowser:fb_browse' %}?pop=2&type=" + type;
```

(continues on next page)

(continued from previous page)

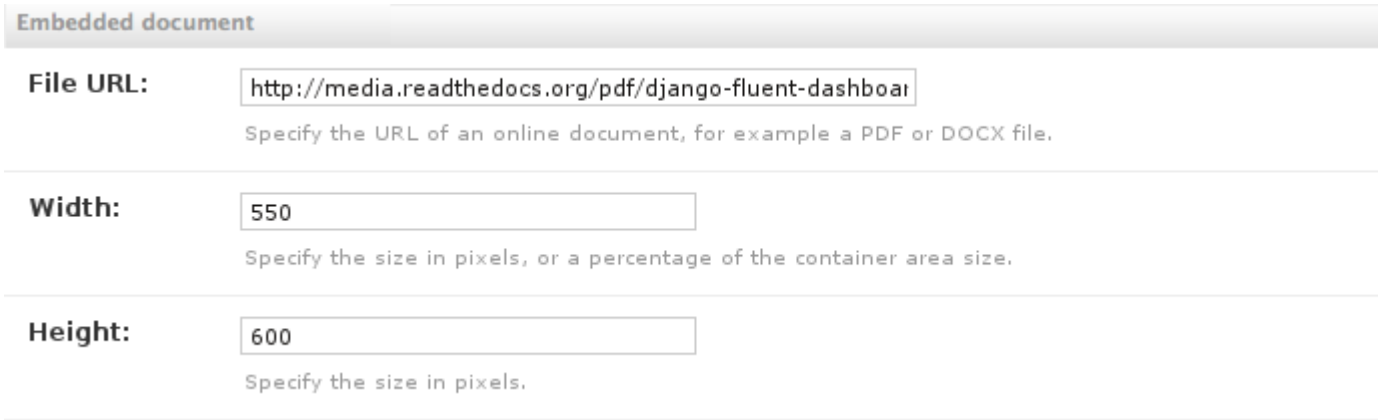
```
        tinyMCE.activeEditor.windowManager.open(
        {
            'file': url,
            'width': 880,
            'height': 500,
            'resizable': "yes",
            'scrollbars': "yes",
            'inline': "no",
            'close_previous': "no"
        },
        {
            'window': win,
            'input': field_name,
            'editor_id': tinyMCE.selectedInstance.editorId
        });
        return false;
    }

{% endblock %}
</script>
```

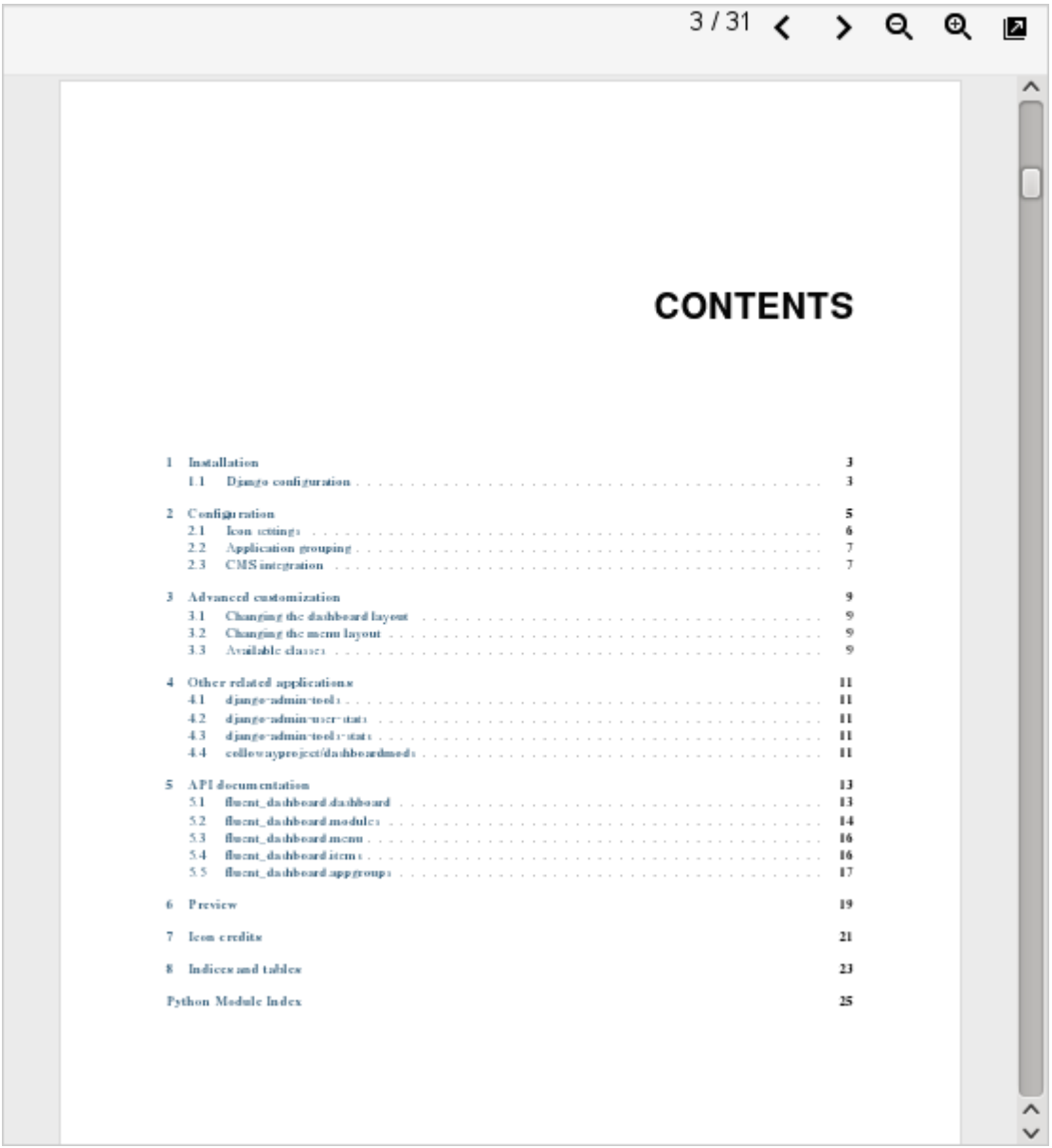
3.1.2 Media

The googledocsviewer plugin

The *googledocsviewer* plugin allows inserting an embedded Google Docs Viewer in the page. This can be used to display various files - like PDF or DOCX files - inline in the HTML page.



The document is rendered by Google:



Installation

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.googledocsviewer',
)
```

The plugin does not provide additional configuration options, nor does it have dependencies on other packages.

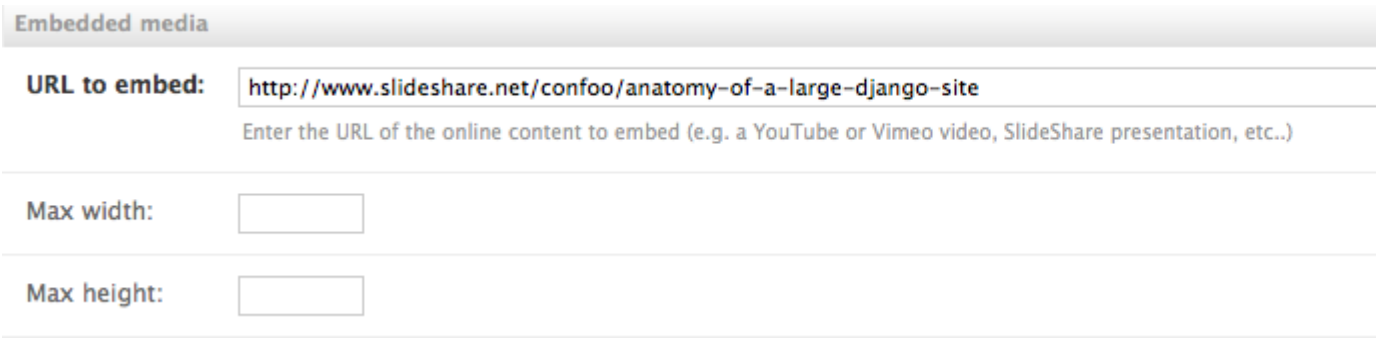
The Google Docs Viewer is rendered as `<iframe>` tag. The appearance of the `<iframe>` tag depends on the CSS style of the web site. Generally, at least a `border` should be defined.

Note: When using the Google Docs viewer on your site, Google assumes you agree with the Terms of Service, see:

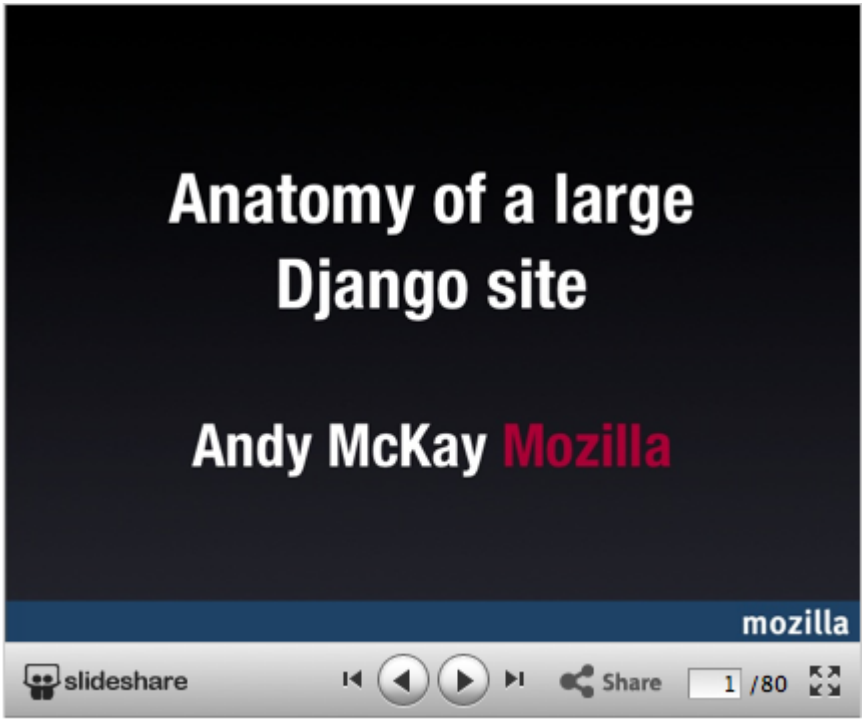
https://docs.google.com/viewer/TOS

The oembeditem plugin

The *oembeditem* plugin allows inserting an embedded online content in the page, such as a YouTube video, SlideShare presentation, Twitter status, Flickr photo, etc..



The presentation is rendered with the embed code:



Anatomy of a large Django site from ConFoo

By default, the following services are supported:

- Blip.tv
- DailyMotion
- Flickr (both videos and images)
- FunnyOrDie
- Instagram

- Imgur
- GitHub Gists
- Hulu
- Mobypicture
- Photobucket
- Polldaddy
- Qik
- Revision3
- Scribd
- Slideshare
- SmugMug
- SoundCloud
- Speaker Desk
- Twitter (status messages)
- Viddler
- Vimeo
- Wordpress.tv
- yfrog
- YouTube (public videos and playlists)

By using the paid-for embed.ly service, many other sites can be supported too.

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-contents[oembeditem]
```

This installs the micawber package.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.oembeditem',
)
```

Configuration

The following settings are available:

```
FLUENT_OEMBED_SOURCE = 'basic'    # "list", "basic" or "embedly"

FLUENT_OEMBED_EXTRA_PROVIDERS = (
    (r'http://\S+.wordpress\.com/\S*',  'http://public-api.wordpress.com/oembed/?
↪for=my-domain-name'),
```

(continues on next page)

(continued from previous page)

```
    (r'http://\S+.wp\.me/\S*',          'http://public-api.wordpress.com/oembed/?
↪for=my-domain-name'),
)

FLUENT_OEMBED_FORCE_HTTPS = False

MICAWBER_EMBEDLY_KEY = ''

FLUENT_OEMBED_PROVIDER_LIST = (
    (r'https?://(www\.)?youtube\.com/watch\S*',  'http://www.youtube.com/oembed'),
    (r'http://youtu\.be/\S*',                    'http://www.youtube.com/oembed'),
    (r'http://blip\.tv/\S*',                     'http://blip.tv/oembed/'),
    (r'https?://(www\.)?vimeo\.com/\S*',         'http://vimeo.com/api/oembed.json'),

    # ...
)
```

FLUENT_OEMBED_SOURCE

The source to use for the OEmbed provider list. This can be one the following values:

- **basic** Use the list of well-known providers in the micawber package.
- **noembed** Use the embed service from noembed
- **embedly** Use the embed service from embed.ly
- **list** Use the provides defined in `FLUENT_OEMBED_PROVIDER_LIST`.

The embed.ly service contains many providers, including sites which do not have an OEmbed implementation themselves. The service does cost money, and requires an API key. For a list of providers supported by embed.ly see http://embed.ly/providers

The *basic* setting is the default, and contains well known services that provide an OEmbed endpoint.

FLUENT_OEMBED_EXTRA_PROVIDERS

The OEmbed providers in this setting will be added to the existing set that `FLUENT_OEMBED_SOURCE` contains. Each item is a tuple with the regular expression and endpoint URL.

FLUENT_OEMBED_FORCE_HTTPS

New in version 1.1.9.

Enforce that the generated embed URLs are served over secure HTTP. This flag is enabled by default when `SECURE_SSL_REDIRECT` is set.

MICAWBER_EMBEDLY_KEY

The key to access the embed.ly service.

FLUENT_OEMBED_PROVIDER_LIST

A fixed hard-coded list of providers. Specify this setting to override the complete set of default OEmbed providers. To add additional providers to any existing source, use `FLUENT_OEMBED_EXTRA_PROVIDERS` instead.

Each item is a tuple with two fields:

- The regular expression to match the URL.
- The OEmbed provider endpoint.

Note that the regular expressions never test for `.*` but use `\S*` instead so micawber can also detect the URL within a larger fragment.

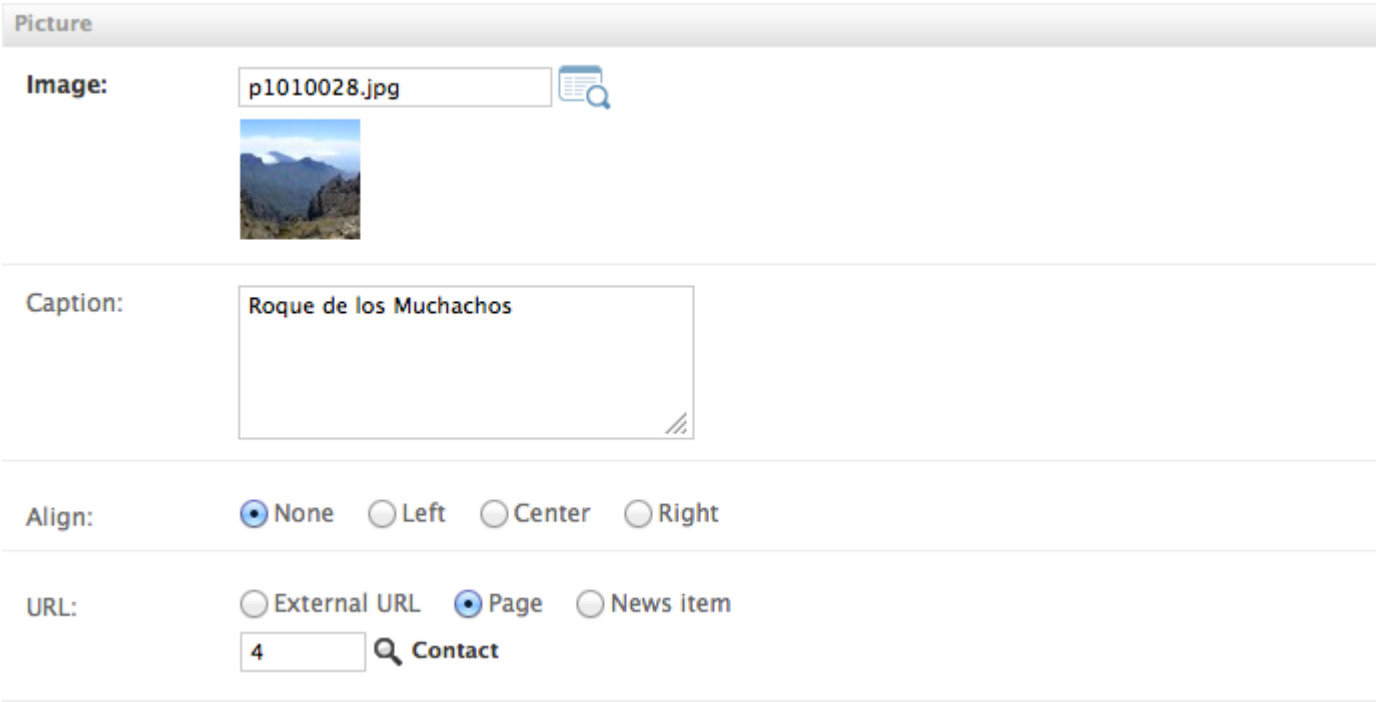
Security considerations

Note that an OEmbed element is fetched from another server, which specifies how the embed code looks like. Hence, only known online services are whitelisted via the `FLUENT_OEMBED_PROVIDER_LIST` setting. This reduces the risks for Cross-site scripting (XSS) attacks.

Hence, the OEmbed discovery protocol is not supported either.

The picture plugin

New in version 0.9.0: The *picture* plugin provides a simple structured way to add images to the page content.



Pictures are outputted in a `<figure>` element and can optionally have a `<figcaption>`:

Roque de los Muchachos

Installation

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.picture',
)
```

The plugin does not provide additional configuration options.

Using improved models fields

By default, the standard Django `ImageField` and `URLField` are used. By including django-any-imagefield and django-any-urlfield in the project, this application will use those fields instead. See the *optional integration with other packages* chapter to enable these features.

Configuration

The following settings are available:

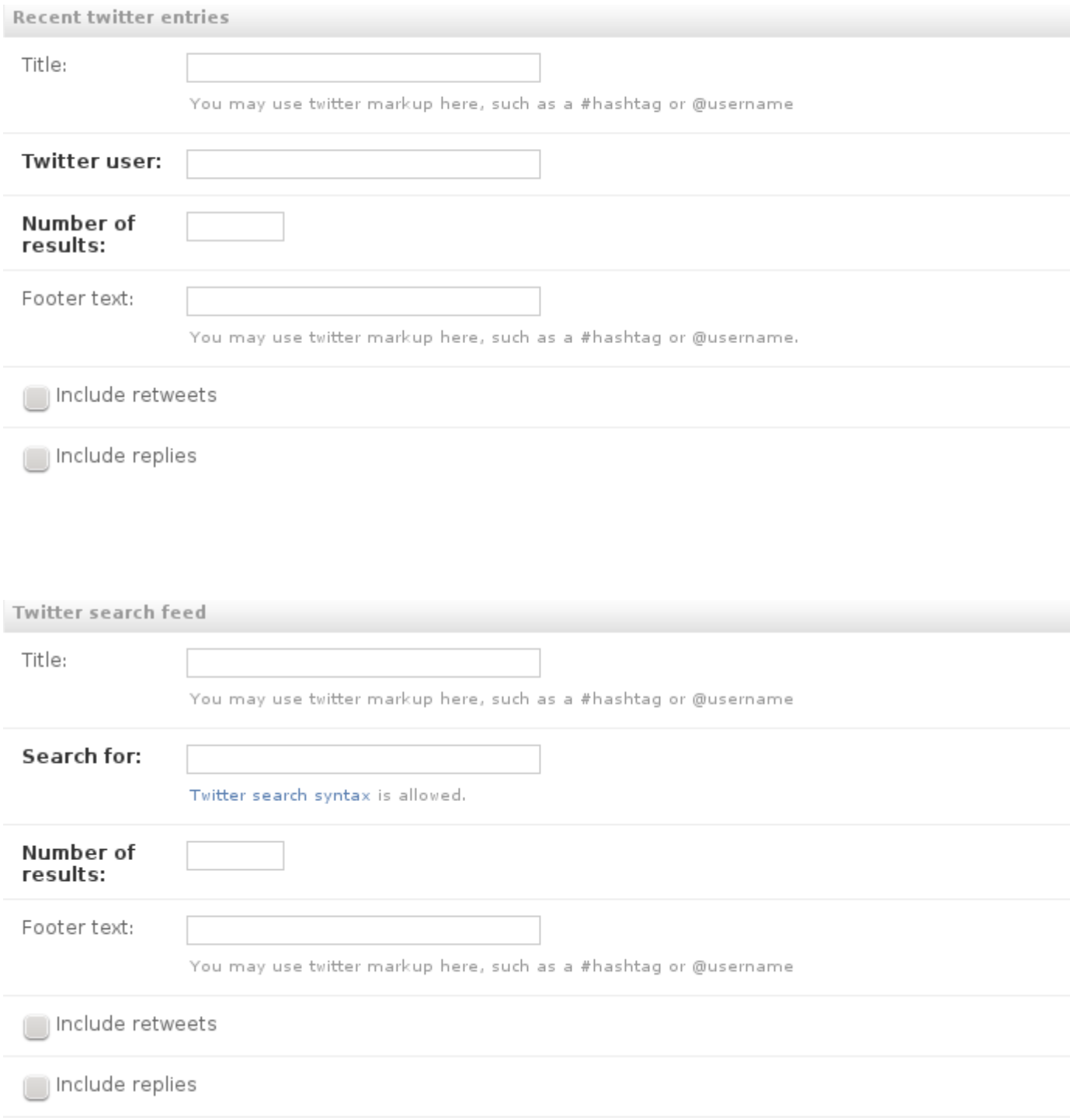
```
FLUENT_PICTURE_UPLOAD_TO = '.'
```

FLUENT_PICTURE_UPLOAD_TO

The upload folder for all pictures. Defaults to the root of the media folder.

The twitterfeed plugin

The *twitterfeed* plugin provides two widgets to display at the web site:



The twitterfeed is fetched client-side by JavaScript, and can be styled using CSS.

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-contents[twitterfeed]
```

This installs the twitter-text-py package.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.twitterfeed',
)
```

Configuration

The following settings are available:

```
FLUENT_TWITTERFEED_AVATAR_SIZE = 32
FLUENT_TWITTERFEED_REFRESH_INTERVAL = 0
FLUENT_TWITTERFEED_TEXT_TEMPLATE = "{avatar}{text} {time}"
```

FLUENT_TWITTERFEED_AVATAR_SIZE

Define the size of the user avatar to display. Typical sizes are 16, 32 or 48 pixels.

FLUENT_TWITTERFEED_REFRESH_INTERVAL

Define after how many seconds all feeds should refresh.

FLUENT_TWITTERFEED_TEXT_TEMPLATE

Define the text layout of all twitter feeds. Various fields of the JavaScript code are available to use. The most relevant are:

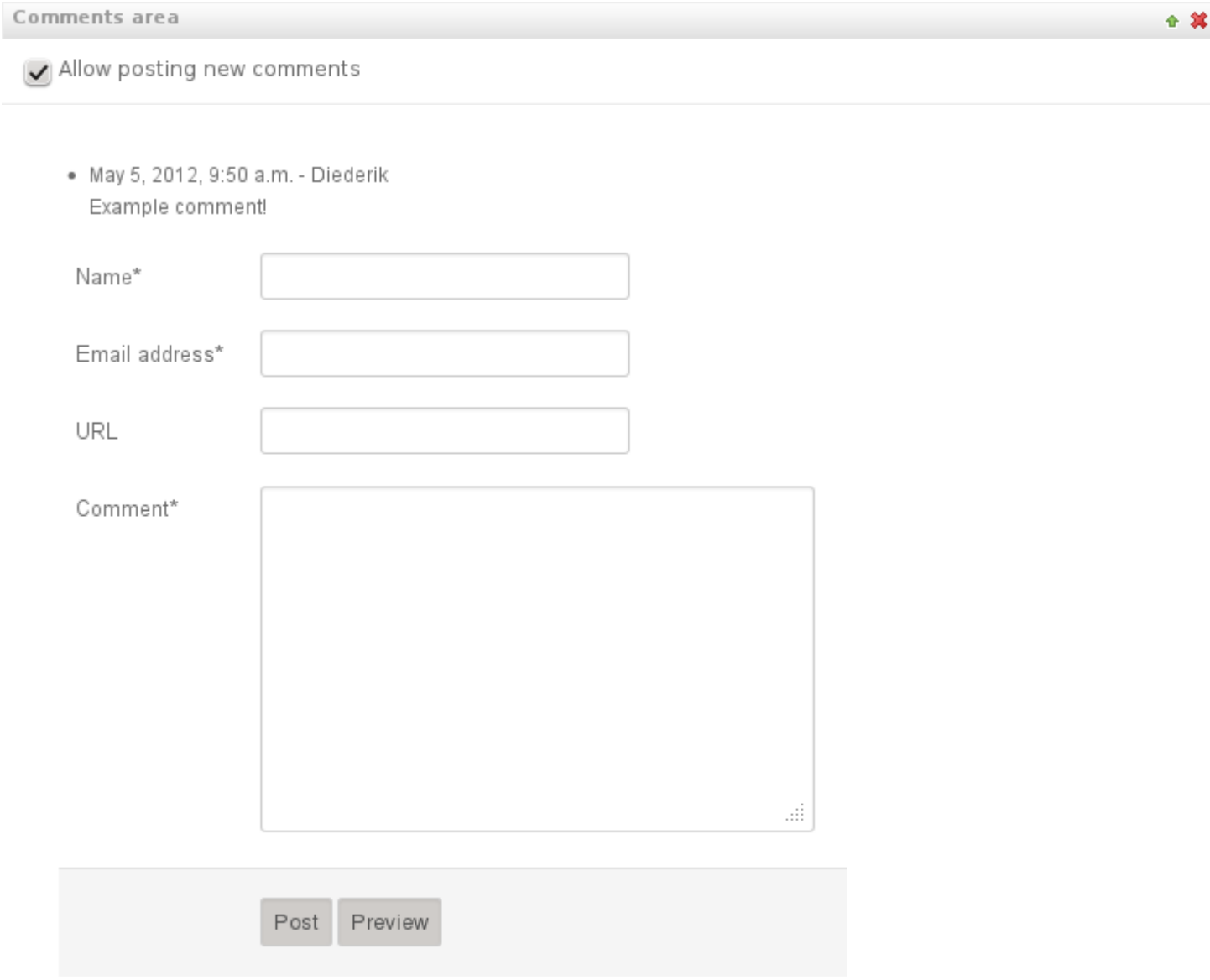
- `{screen_name}`: The user name.
- `{avatar}`: The avatar image tag.
- `{text}`: The text of the tweet
- `{time}`: the relative time
- `{user_url}`: The URL to the user profile
- `{tweet_url}`: The permalink URL of the twitter status.

3.1.3 Interactivity

The commentsarea plugin

The *commentsarea* plugin displays the form and messagelist that django-contrib-comments (or django.contrib.comments) renders.

The displays a list of comments, with a comments area below:



By default, the displayed comments will look very plain. This is however, not an accident. The django.contrib.comments module provides these defaults to make it fully customizable to the design and workflow of the web site where it is being used. Hence, this plugin depends on a properly configured django.contrib.comments module.

Tip: For an advanced plug&play setup, you can use the django-fluent-comments application which includes features such as Ajax-based posting. Make sure to include it's JavaScript and CSS files somewhere in the page.

Installation

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'django_comments',
    'fluent_contents.plugins.commentsarea',
)
```

The django.contrib.comments module also requires a location for it's pages. Add the following to `urls.py`:

```
urlpatterns += [
    url(r'^blog/comments/', include('django_comments.urls')),
]
```

This URL can be anything off course, like `/comments/`, `/respond/comments/` or `/blog/comments/` for example.

Note: As of Django 1.8, the django.contrib.comments module is no longer bundled with Django. It's provided as separate application that can be installed from PyPI.

For older Django projects, replace `django_comments` with `django.contrib.comments` in the example above.

Configuration

After the installation, each page can be enriched with a comments area. Posting a comment however, produces an almost blank page. That's because `comments/base.html` should be overwritten.

To get a usable comments module, the least you need to do, is providing two templates:

- `comments/base.html`
- `comments/posted.html`

Note: As with other plugins of *django-fluent-contents*, the output of the plugin is cached. Only when a comment is posted, the output will be refreshed. To change templates in a development/runserver environment, set *FLUENT_CONTENTS_CACHE_OUTPUT* to `False` in the settings.

The base.html template

The `comments/base.html` template is used by every template of the comments module. It needs to provide two blocks;

- **title**: the sub title to display in the `<title>` tag.
- **content**: the content to display in the `<body>` tag.

These blocks can be mapped to your site template. It's contents could be something like:

```
{% extends "mysite/base.html" %}{% load i18n %}

{% block headtitle %}{% block title %}{% trans "Responses for page" %}{% endblock %}{
↪% endblock %}

{% block main %}
          <div id="content" class="clearfix">
```

(continues on next page)

(continued from previous page)

```
            {% block content %}{% endblock %}
          </div>
{% endblock %}
```

The `comments/base.html` file can be stored in the `templates` folder of your site theme.

The posted.html template

The final "Thank you for posting" page is also quite plain. Replace it be something more fresh by overriding the `comments/posted.html` template. For example, try something like:

```
{% extends "comments/base.html" %}{% load i18n %}

{% block title %}{% trans "Thanks for commenting" %}{% endblock %}

{% block extrahead %}
{{ block.super }}
    <meta http-equiv="Refresh" content="5; url={{ comment.content_object.get_absolute_
↪url }}#c{{ comment.id }}" />
{% endblock %}

{% block content %}
    <h1>Thank you for responding</h1>
    <p>
      We have received your comment, and added it to the web site.<br />
      You will be sent back to the article...
    </p>

    {# Using identical formatting to normal comment list #}
    <dl id="comments">
      <dt id="c{{ comment.id }}">
        {{ comment.submit_date }} - {{ comment.name }}
      </dt>
      <dd>
        <p>{{ comment.comment }}</p>
      </dd>
    </dl>

    <p><a href="{{ comment.content_object.get_absolute_url }}#c{{ comment.id }}">Back␣
↪to the article</a></p>
{% endblock %}
```

The template now contains links back to the blog page, and no longer appears as dead end. It will automatically redirect back to the blog in a few seconds.

Additional configuration

The django.contrib.comments module can be further extended with other modules. In fact, django.contrib.comments only establishes a standard methodology for integrating comments to a Django site. The framework also supports moderation, flagging, and RSS feeds too. More documentation can be found at:

- Django's comments framework
- Customizing the comments framework

- Example of using the in-built comments app

Some Django applications already implement these features. For example:

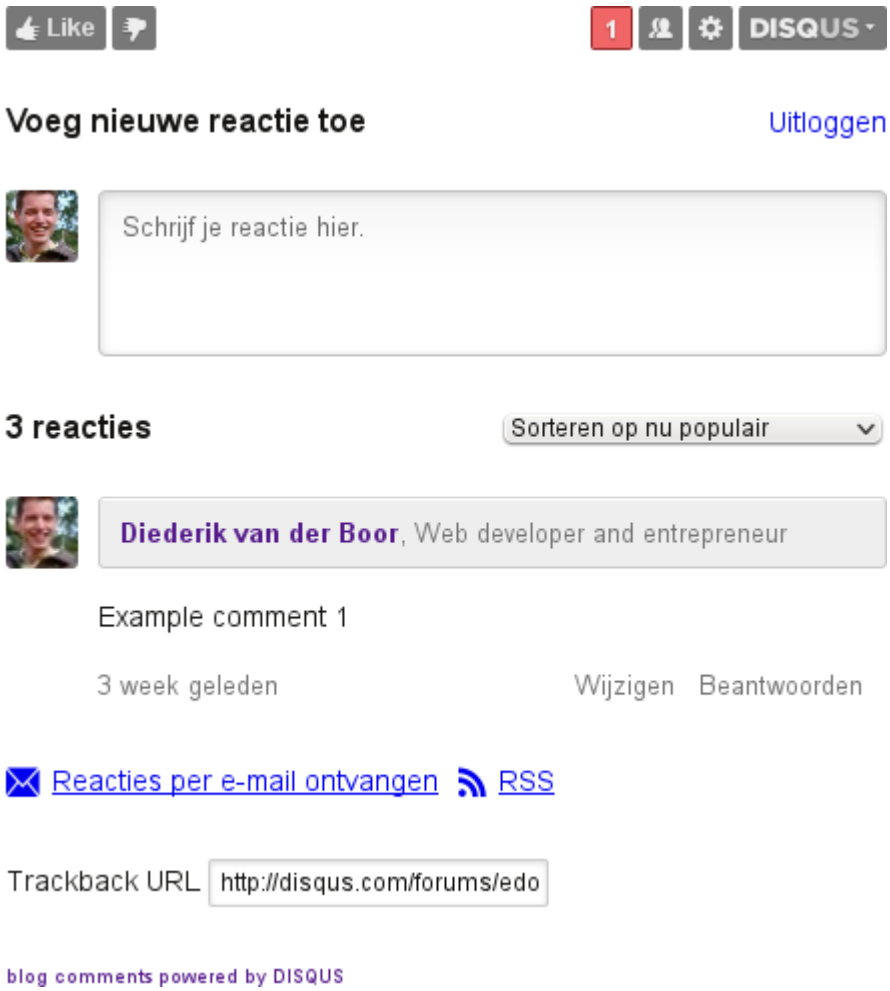
- django-fluent-comments, which includes:
- Ajax-based previews and posting of comments.
- Comment moderation, and Akismet based filtering.
- E-mail notifications.
- django-threadedcomments
- django-comments-spamfighter
- django-myrecaptcha

These modules can enhance the commentsarea even further.

The disquscommentsarea plugin

The *disquscommentsarea* plugin displays a comments area powed by DISQUS.

This displays the DISQUS comments area:



In the background, DISQUS uses a JavaScript include to embed the comments. Google indexes the comments nevertheless.

The plugin uses django-disqus internally to render the HTML output. The django-disqus module also offers management commands to import the comments of django.contrib.comments to DISQUS, or export the comments from DISQUS as JSON or WXR feed.

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-contents[disquscommentsarea]
```

This installs django-disqus.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'disqus',
    'fluent_contents.plugins.disquscommentsarea',
)
```

(continues on next page)

(continued from previous page)

```
DISQUS_API_KEY = '..'      # Insert API key here.
DISQUS_SHORTNAME = '..'    # Insert the website shortname.
```

Configuration

The plugin does not provide any additional configuration, it fully relies on the templatetags of django-disqus to provide a proper comments area.

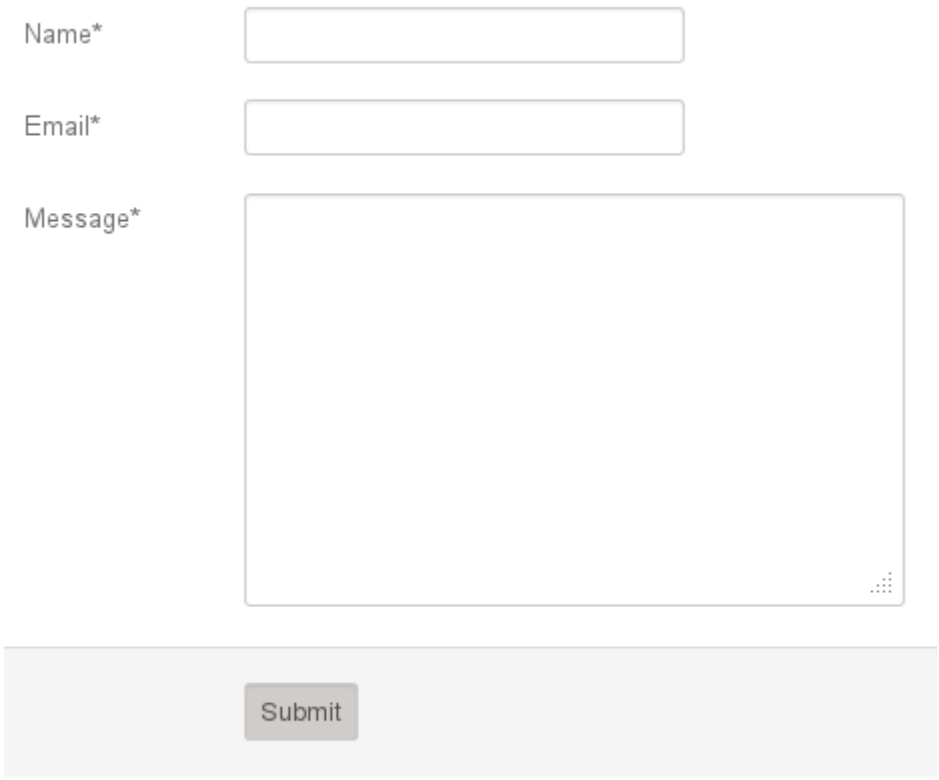
- The API key can be created at the DISQUS website. You can get your API key here (you must be logged in on the DISQUS website). To see the shortname of your website, navigate to Settings->General on the DISQUS website.
- Secondly, make sure the *django.contrib.sites* framework is configured, including the domain name where the pages should be displayed.

Tip: While the DISQUS include provides all default styles, it can be useful to specify some default styling as well. This avoids any undesired minor jumps by page elements when DISQUS is loading.

The formdesignerlink plugin

The *formdesignerlink* plugin displays a form, created by the django-form-designer-ai module.

The form is displayed at the website:



Note: While the *form_designer* interface may not be fully up to the "UI standards" of *django-fluent-contents*, it is however a popular module, and hence this plugin is provided!

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-contents[formdesignerlink]
```

This installs django-form-designer-ai.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'form_designer',
    'fluent_contents.plugins.formdesignerlink',
)
```

To display previews, the *form_designer* application also requires an additional line in `urls.py`:

```
urlpatterns += [
    url(r'^forms/', include('form_designer.urls')),
]
```

Each page can now be enriched with a form, that was created by the *form_designer* application.

Configuration

To customize the output, configure the `django-form-designer-ai` application via the settings file. Some relevant settings are:

`FORM_DESIGNER_DEFAULT_FORM_TEMPLATE` Defines the default template to use to render a form. For example, the template can be rendered with `django-uni-form`.

`FORM_DESIGNER_FORM_TEMPLATES` Defines a list of choices, to allow users to select a template.

`FORM_DESIGNER_FIELD_CLASSES` A list of choices, to define which Django field types are allowed.

`FORM_DESIGNER_WIDGET_CLASSES` A list of choices, to define which Django widget types are allowed.

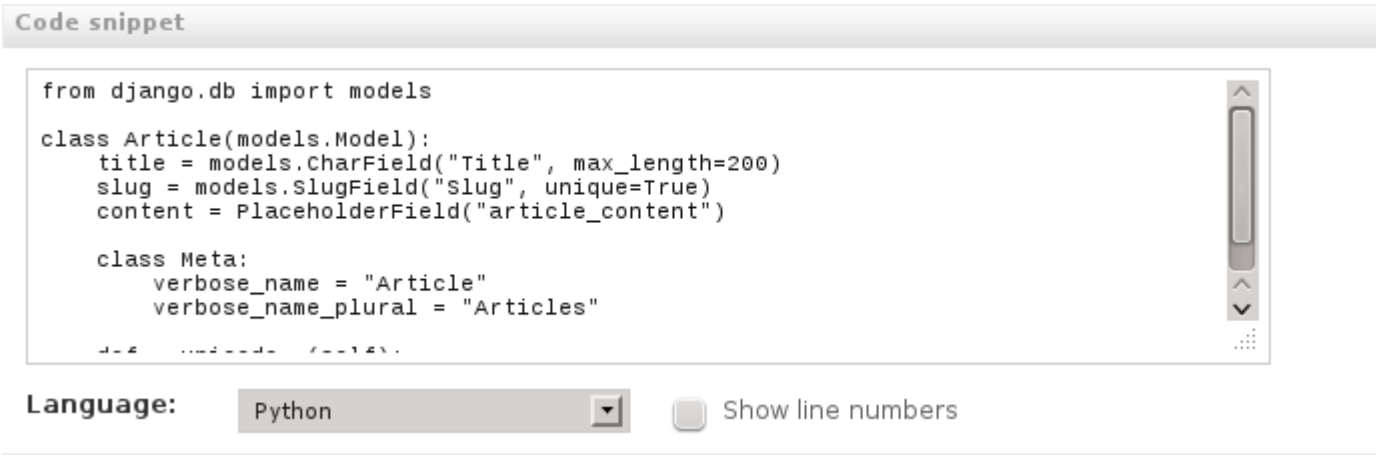
It is also highly recommended to overwrite the `form_designer/templates/html/formdefinition/base.html` template, which is used to provide previews for the form in the admin interface.

Further information can be found in the source code of *django-formdesigner*. (e.g. settings.py).

3.1.4 Programming

The code plugin

The *code* plugin provides highlighting for programming code.



The plugin uses Pygments as backend to perform the highlighting:

```
from django.db import models

class Article(models.Model):
    title = models.CharField("Title", max_length=200)
    slug = models.SlugField("Slug", unique=True)
    content = PlaceholderField("article_content")

    class Meta:
        verbose_name = "Article"
        verbose_name_plural = "Articles"

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

The color theme can be configured in the settings.

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-contents[code]
```

This installs the Pygments package.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.code',
)
```

Configuration

No settings have to be defined. For further tuning however, the following settings are available:

```
FLUENT_CODE_DEFAULT_LANGUAGE = 'html'
FLUENT_CODE_DEFAULT_LINE_NUMBERS = False

FLUENT_CODE_STYLE = 'default'

FLUENT_CODE_SHORTLIST = ('python', 'html', 'css', 'js')
FLUENT_CODE_SHORTLIST_ONLY = False
```

FLUENT_CODE_DEFAULT_LANGUAGE

Define which programming language should be selected by default.

This setting is ideally suited to set personal preferences. By default this is “HTML”, to be as neutral as possible.

FLUENT_CODE_DEFAULT_LINE_NUMBERS

Define whether line number should be enabled by default for any new plugins.

FLUENT_CODE_STYLE

The desired highlighting style. This can be any of the themes that Pygments provides.

Each style name refers to a python module in the `pygments.styles` package. The styles provided by Pygments 1.4 are:

- *autumn*
- *borland*
- *bw* (black-white)
- *colorful*
- *default*
- *emacs*
- *friendly*
- *fruity*
- *manni*
- *monokai*
- *murphy*
- *native*
- *pastie*
- *perldoc*
- *tango*
- *trac*
- *vim*
- *vs* (Visual Studio colors)

Note: This setting cannot be updated per plugin instance, to avoid a mix of different styles used together. The entire site uses a single consistent style.

FLUENT_CODE_SHORTLIST

The plugin displays a shortlist of popular programming languages in the "Language" selectbox, since Pygments provides highlighting support for many many programming languages.

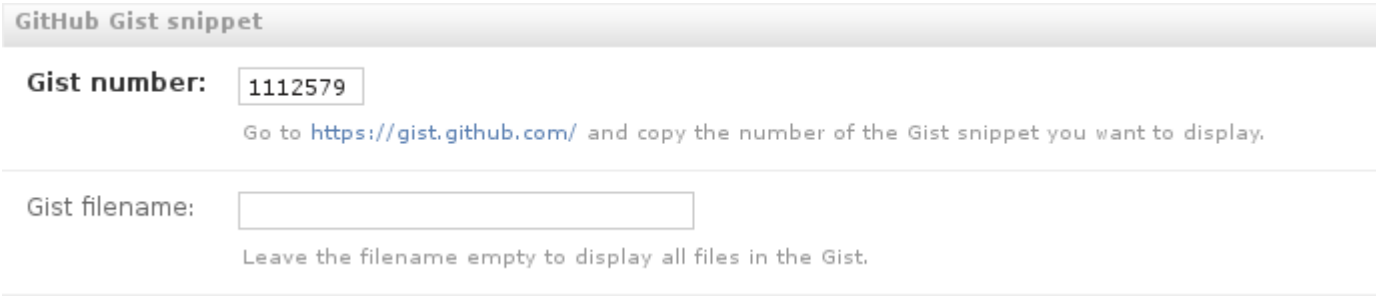
This settings allows the shortlist to be customized.

FLUENT_CODE_SHORTLIST_ONLY

Enable this setting to only show the programming languages of the shortlist. This can be used to simplify the code plugin for end users.

The gist plugin

The *gist* plugin provides highlighting of programming code snippets (referred to as Gists), which are hosted at GitHub.



Gist snippets are built by a JavaScript include, which renders the Gist with syntax highlighting:

```
class PersonalModule(modules.LinkList):
    title = _('Welcome,')
    draggable = False
    deletable = False
    collapsible = False
    template = 'ecms_dashboard/modules/personal.html'

    def init_with_context(self, context):
        super(PersonalModule, self).init_with_context(context)

        current_user = context['request'].user
        site_name = get_admin_site_name(context)

        # Personalize
        self.title = _('Welcome,') + ' ' + (current_user.first

        # Expose links
        self.pages_link = reverse('%s:ecms_cmsobject_changelis
        self.password_link = reverse('%s:password_change' % s
        self.logout_link = reverse('%s:logout' % site_name)

    def is_empty(self):
        return False
```

This Gist brought to you by GitHub. dashboard.py view raw

Installation

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.gist',
)
```

The plugin does not provide additional configuration options, nor does it have dependencies on other packages.

The markup plugin

The *markup* plugin provides support for lightweight "markup" languages.

reStructuredText

```
Welcome to django-fluent-contents' documentation!
=================================================

This documentation covers the latest release of django-fluent-contents, a
collection of applications to build an end user CMS for the `Django
<http://www.djangoproject.com>`_ administration interface.
django-fluent-contents includes:

* A ``PlaceholderField`` to display various content on a model.
* A ``PlaceholderEditorAdmin`` to build CMS interfaces.
* A default set of plugins to display WYSIWYG content, reStructuredText,
```

The markup language is rendered as HTML:

Welcome to django-fluent-contents' documentation!

This documentation covers the latest release of django-fluent-contents, a collection of applications to build an end user CMS for the Django administration interface. django-fluent-contents includes:

- A PlaceholderField to display various content on a model.
- A PlaceholderEditorAdmin to build CMS interfaces.
- A default set of plugins to display WYSIWYG content, reStructuredText, highlighted code, Gist snippets and more.
- an extensible plugin API.

The plugin provided support for the markup languages:

- **reStructuredText**: The syntax known for Python documentation.
- **Markdown**: The syntax known for GitHub and Stack Overflow comments (both do have a dialect/extended version)
- **Textile**: The syntax known for Redmine and partially used in Basecamp and Google+.

For technical savy users, this makes it easy to write pages in a consistent style, or publish wiki/RST documentation online.

The plugin uses django.contrib.markup internally to provide the translation of the markup to HTML, hence it supports any markup language that django.contrib.markup has a filter for.

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-contents[markup]
```

Depending on the choosen markup language, the required dependencies can also installed separately using *pip*:

For **reStructuredText**, use:

```
pip install docutils
```

For **Markdown**, use:

```
pip install Markdown
```

For **Textile**, use:

```
pip install textile
```

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.markup',
)

FLUENT_MARKUP_LANGUAGES = ['restructuredtext', 'markdown', 'textile']
```

Configuration

The following settings are available:

```
FLUENT_MARKUP_LANGUAGES = ['restructuredtext', 'markdown', 'textile']
FLUENT_MARKUP_MARKDOWN_EXTRAS = ["extension1_name", "extension2_name", "..."]
```

FLUENT_MARKUP_LANGUAGES

Define which markup language should be used.

This is a list/tuple, which can contain the following values:

- *restructuredtext*
- *markdown*
- *textile*

By default, all languages are added.

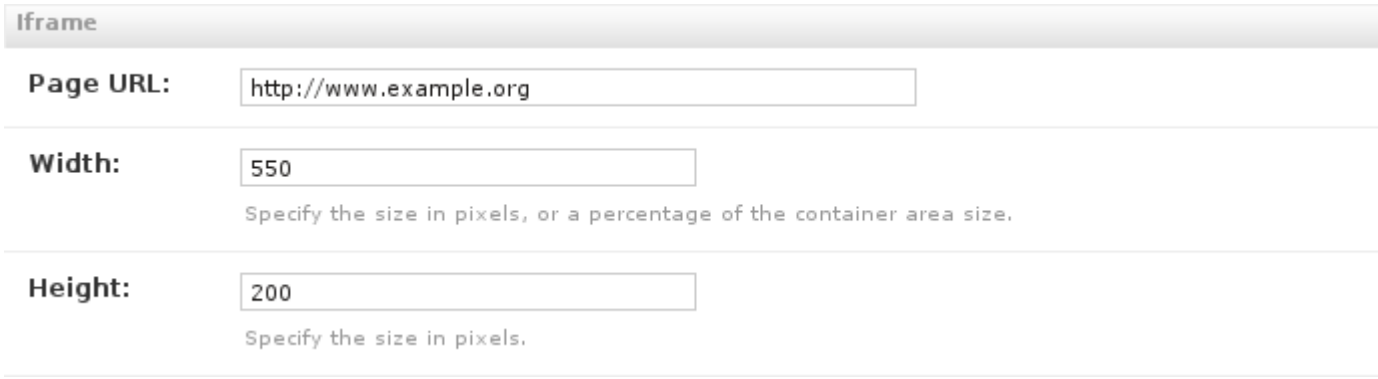
FLUENT_MARKUP_MARKDOWN_EXTRAS

Define the markdown extensions to use.

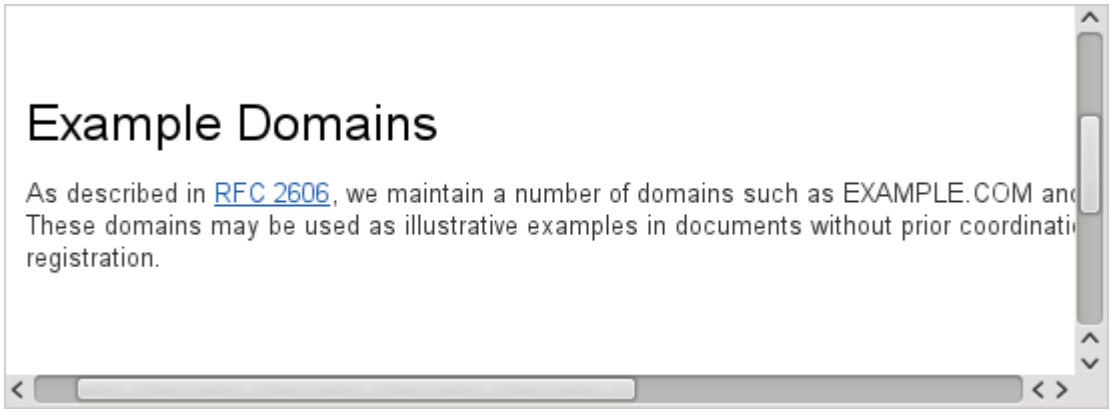
3.1.5 Advanced

The iframe plugin

The *iframe* plugin allows inserting IFrames to the page.



The frame is displayed at the website:



Generally, such feature is useful for large web sites where a specific service or contact form needs to be embedded. In case a variation is needed, this plugin can easily be used as starting point for writing a custom plugin.

Installation

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.iframe',
)
```

The plugin does not provide additional configuration options, nor does it have dependencies on other packages.

The appearance of the `<iframe>` tag depends on the CSS style of the web site. Generally, at least a `border` should be defined.

The rawhtml plugin

The *rawhtml* plugin allows inserting raw HTML code in the page.

HTML code

```
<iframe width="420" height="315" src="http://www.youtube.com/embed/A-S0tqpPga4"
frameborder="0" allowfullscreen></iframe>
```

Enter the HTML code to display, like the embed code of an online widget.

The code is included as-is at the frontend:

This plugin can be used for example for:

- prototyping extra markup quickly
- including "embed codes" in a page, in case the *oembeditem* plugin does not support it.
- including jQuery snippets in a page:

HTML code

```
<!-- raw inserted HTML -->
<link rel="stylesheet" type="text/css" href="/static/edoburu/extlib/fancybox
/jquery.fancybox-1.3.4.css" />
<script type="text/javascript" src="/static/edoburu/extlib/fancybox/jquery.fancybox-
1.3.4.js"></script>
<script type="text/javascript">
  $(document).ready(function(){
    $("#middle-text a > img").parent().fancybox();
  });
</script>
```

Vul de HTML code in om weer te geven, zoals de embed code van een online widget.

Installation

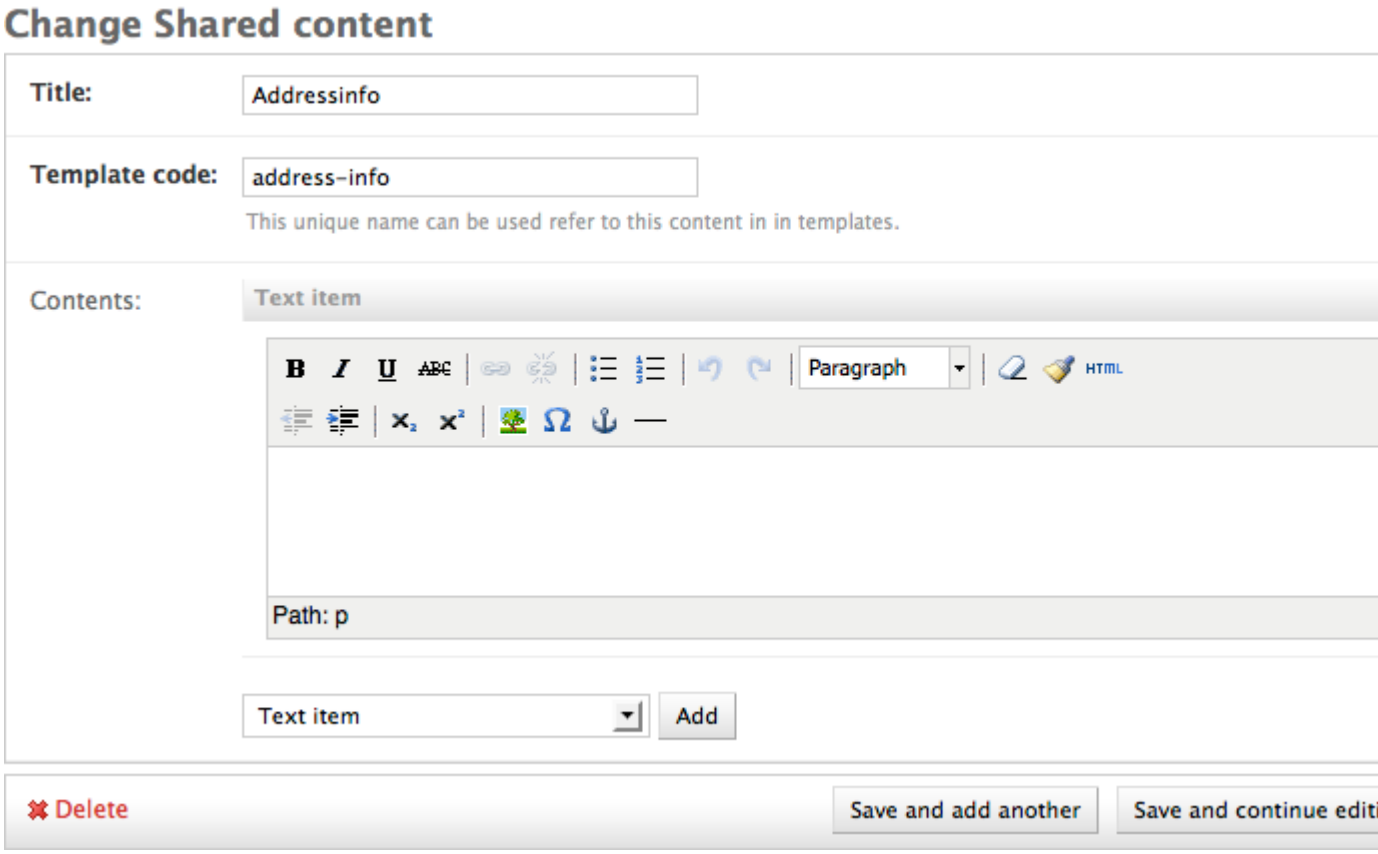
Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.rawhtml',
)
```

The plugin does not provide additional configuration options, nor does it have dependencies on other packages.

The sharedcontent plugin

New in version 0.8.5: The *sharedcontent* plugin allows inserting an content at multiple locations in the site.



The shared content can be included in the site:

Installation

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_contents.plugins.sharedcontent',
)
```

The plugin does not have dependencies on other packages.

Configuration

No settings have to be defined. For further tuning however, the following settings are available:

```
FLUENT_SHARED_CONTENT_ENABLE_CROSS_SITE = False
```

FLUENT_SHARED_CONTENT_ENABLE_CROSS_SITE

By default, each `Site` model has it's own contents. Each site has it's own set of shared contents objects.

By enabling this flag, the admin interface provides a "Share between all sites" setting. This can be found in the collapsed "Publication settings" tab. Once enabled, a shared content item is visible across all websites that the Django instance hosts.

This can be enabled to support content such as the text of an "About" or "Terms and Conditions" page.

When *FLUENT_CONTENTS_FILTER_SITE_ID* is disabled, this feature is nullified, as all contents is always available across all sites.

Usage in templates

To hard-code the shared content in a template, use:

```
{% load sharedcontent_tags %}

{% sharedcontent "template-code-name" %}
```

To customize the placeholder contents, a template can be specified:

```
{% sharedcontent "template-code-name" template="mysite/parts/slot_placeholder.html" %}
```

That template should loop over the content items, and include additional HTML. For example:

```
{% for contentitem, html in contentitems %}
  {% if not forloop.first %}<div class="splitter"></div>{% endif %}
  {{ html }}
{% endfor %}
```

Note: When a template is used, the system assumes that the output can change per request. Hence, the output of individual items will be cached, but the final merged output is no longer cached. Add `cachable=1` to enable output caching for templates too.

3.2 Creating new plugins

The idea of this module is that you can create a custom layout by defining "content items" for each unique design.

Typically, a project consists of some standard modules, and additional custom items. Creating custom items is easy, and the following pages explain the details.

3.2.1 Example plugin code

A plugin is a standard Django/Python package. As quick example, let's create an announcement block.

This is a typical module that is found on many websites; a title text, some intro text and "call to action" button at the bottom. Such item could be created in a WYSIWYG editor, but in this case we'll provide a clear interface for the editorial content.

The plugin can be created in your Django project, in a separate app which can be named something like `plugins.announcementblock` or `mysite.contentitems`.

Example code

For the `plugins.announcementblock` package, the following files are needed:

- `__init__.py`, naturally.
- `models.py` for the database model.
- `content_plugins.py` for the plugin definition.

models.py

The models in `models.py` needs to inherit from the *ContentItem* class, the rest is just standard Django model code.

```
from django.db import models
from django.utils.translation import ugettext_lazy as _
from fluent_contents.models import ContentItem

class AnnouncementBlockItem(ContentItem):
    """
    Simple content item to make an announcement.
    """
    title = models.CharField(_("Title"), max_length=200)
    body = models.TextField(_("Body"))

    button_text = models.CharField(_("Text"), max_length=200)
    button_link = models.URLField(_("URL"))

    class Meta:
        verbose_name = _("Announcement block")
        verbose_name_plural = _("Announcement blocks")

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

This *ContentItem* class provides the basic fields to integrate the model in a placeholder. The `verbose_name` and `__unicode__` fields are required to display the model in the admin interface.

content_plugins.py

The `content_plugins.py` file can contain multiple plugins, each should inherit from the *ContentPlugin* class.

```
from django.utils.translation import ugettext_lazy as _
from fluent_contents.extensions import plugin_pool, ContentPlugin
from .models import AnnouncementBlockItem


@plugin_pool.register
class AnnouncementBlockPlugin(ContentPlugin):
    model = AnnouncementBlockItem
    render_template = "plugins/announcementblock.html"
    category = _("Simple blocks")
    cache_output = False    # Temporary set for development

    fieldsets = (
        (None, {
            'fields': ('title', 'body',)
        }),
        (_("Button"), {
            'fields': ('button_text', 'url',)
        })
    )
```

The plugin class binds all parts together; the model, metadata, and rendering code. Either the *render()* function can be overwritten, or a *render_template* can be defined.

The other fields, such as the `fieldsets` are optional. The *plugin_pool.register* decorator registers the plugin.

announcementblock.html

The default *render()* code makes the model instance available as the `instance` variable. This can be used to generate the HTML:

```
<div class="announcement">
    <h3>{{ instance.title }}</h3>
    <div class="text">
        {{ instance.body|linebreaks }}
    </div>
    <p class="button"><a href="{{ instance.button_url }}">{{ instance.button_text }}</
↪a></p>
</div>
```

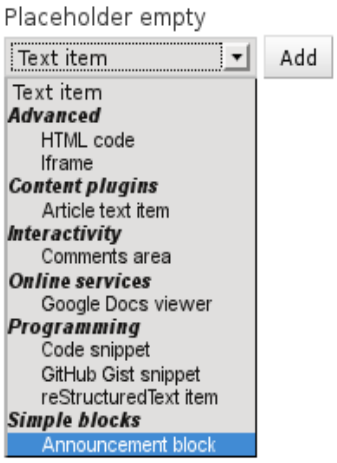
Important: By default, the output of plugins is cached; changes to the template file are only visible when the model is saved in the Django admin. You can set *FLUENT_CONTENTS_CACHE_OUTPUT* to `False`, or use the *cache_output* setting temporary in development. The setting is enabled by default to let plugin authors make a conscious decision about caching and avoid unexpected results in production.

Wrapping up

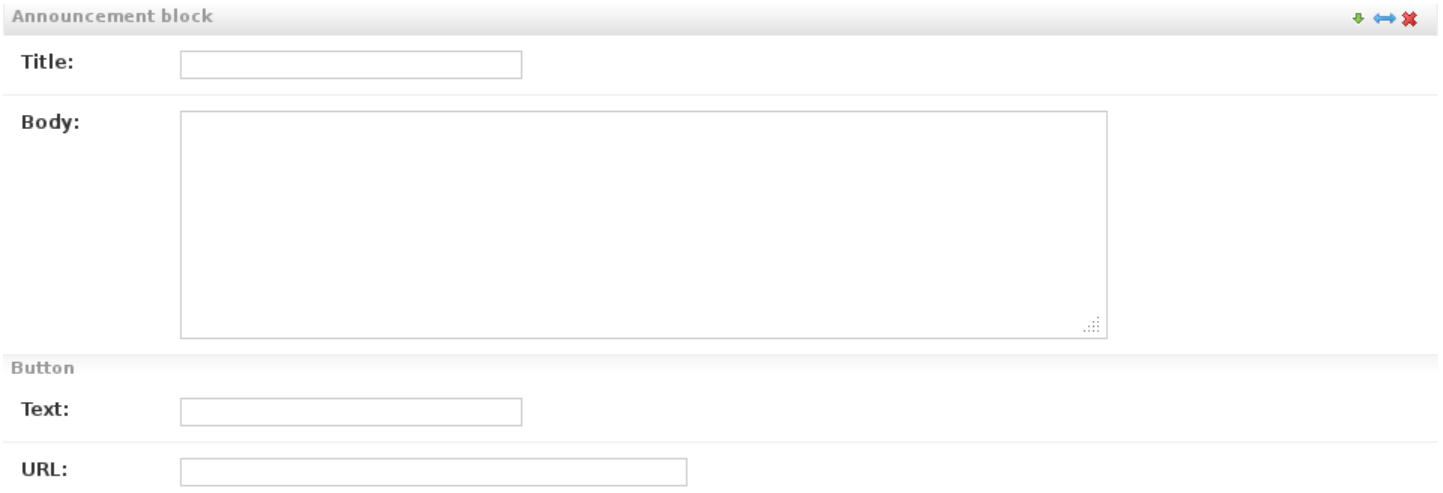
The plugin is now ready to use. Don't forget to add the `plugins.announcementblock` package to the `INSTALLED_APPS`, and create the tables:

```
./manage.py syncdb
```

Now, the plugin will be visible in the editor options:



After adding it, the admin interface will be visible:



The appearance at the website, depends on the sites CSS theme off course.

This example showed how a new plugin can be created within 5-15 minutes! To continue, see *Customizing the frontend rendering* to implement custom rendering.

3.2.2 Customizing the frontend rendering

As displayed in the *Example plugin code* page, a plugin is made of two classes:

- A model class in `models.py`.
- A plugin class in `content_plugins.py`.

The plugin class renders the model instance using:

- A custom *render()* method.
- The *render_template* attribute, *get_render_template()* method and optionally *get_context()* method.

Simply stated, a plugin provides the "view" of a "model".

Simple rendering

To quickly create plugins with little to no effort, only the *render_template* needs to be specified. The template code receives the model object via the `instance` variable.

To switch the template depending on the model, the *get_render_template()* method can be overwritten instead. For example:

```
@plugin_pool.register
class MyPlugin(ContentPlugin):
    # ...

    def get_render_template(self, request, instance, **kwargs):
        return instance.template_name or self.render_template
```

To add more context data, overwrite the *get_context* method. The *twitterfeed* plugins use this for example to pass settings to the template:

```
@plugin_pool.register
class MyPlugin(ContentPlugin):
    # ...

    def get_context(self, request, instance, **kwargs):
        context = super(MyPlugin, self).get_context(request, instance, **kwargs)
        context.update({
            'AVATAR_SIZE': int(appsettings.FLUENT_TWITTERFEED_AVATAR_SIZE),
            'REFRESH_INTERVAL': int(appsettings.FLUENT_TWITTERFEED_REFRESH_INTERVAL),
            'TEXT_TEMPLATE': appsettings.FLUENT_TWITTERFEED_TEXT_TEMPLATE,
        })
        return context
```

For most scenario's, this provides simple flexibility with a DRY approach.

Custom rendering

Instead of only providing extra context data, the whole *render()* method can be overwritten as well.

It should return a string with the desired output. For example, this is the render function of the *text* plugin:

```
def render(self, request, instance, **kwargs):
    return mark_safe('<div class="text">' + instance.text + '</div>\n')
```

The standard *render()* method basically does the following:

```
def render(self, request, instance, **kwargs):
    template = self.get_render_template(request, instance, **kwargs)
    context = self.get_context(request, instance, **kwargs)
    return self.render_to_string(request, template, context)
```

- It takes the template from *get_render_template()*.
- It uses the the context provided by *get_context()*.
- It uses *render_to_string()* method which adds the STATIC_URL and MEDIA_URL variables in the template.

The output will be escaped by default, so use Django's format_html() or mark_safe() when content should not be escaped. Hence, it's preferred to use a template unless that makes things more complex.

Internally, the *render_to_string()* method wraps the rendering context in a *PluginContext()*. which is similar to the RequestContext that Django provides.

Form processing

An entire form with GET/POST can be handled with a plugin. This happens again by overwriting *render()* method.

For example, a "Call me back" plugin can be created using a custom *render()* function:

```
@plugin_pool.register
class CallMeBackPlugin(ContentPlugin):
    model = CallMeBackItem
    category = _("Contact page")
    render_template = "contentplugins/callmeback/callmeback.html"
    cache_output = False   # Important! See "Output caching" below.

    def render(self, request, instance, **kwargs):
        context = self.get_context(request, instance, **kwargs)
        context['completed'] = False

        if request.method == 'POST':
            form = CallMeBackForm(request.POST, request.FILES)
            if form.is_valid():
                instance = form.save()
                return self.redirect(reverse('thank-you-page'))
        else:
            form = CallMeBackForm()

        context['form'] = form
        return self.render_to_string(request, self.render_template, context)
```

Note: The *cache_output* attribute is False to disable the default output caching. The POST screen would return the cached output instead.

To allow plugins to perform directs, add *fluent_contents.middleware.HttpRedirectRequestMiddleware* to MIDDLEWARE_CLASSES.

Frontend media

Plugins can specify additional JS/CSS files which should be included. For example:

```
@plugin_pool.register
class MyPlugin(ContentPlugin):
    # ...
```

(continues on next page)

(continued from previous page)

```
    class FrontendMedia:
        css = {
            'all': ('myplugin/all.css',)
        }
        js = (
            'myplugin/main.js',
        )
```

Equally, there is a *frontend_media* property, and *get_frontend_media* method.

Output caching

By default, plugin output is cached and only refreshes when the administrator saves the page. This greatly improves the performance of the web site, as very little database queries are needed, and most pages look the same for every visitor anyways.

- When the plugin output is dynamic set the *cache_output* to `False`.
- When the plugin output differs per `SITE_ID` only, set *cache_output_per_site* to `True`.
- When the plugin output differs per language, set *cache_output_per_language* to `True`.
- When the output should be refreshed more often, change the *cache_timeout*.
- As last resort, the caching can be disabled entirely project-wide using the *FLUENT_CONTENTS_CACHE_OUTPUT* setting. This should be used temporary for development, or special circumstances only.

Most plugins deliver exactly the same content for every request, hence the setting is tuned for speed by default. Further more, this lets plugin authors make a conscious decision about caching, and to avoid unexpected results in production.

When a plugin does a lot of processing at render time (e.g. requesting a web service, parsing text, sanitizing HTML, or do XSL transformations of content), consider storing the intermediate rendering results in the database using the `save()` method of the model. The *code plugin* uses this for example to store the highlighted code syntax. The *render()* method can just read the value.

Development tips

In `DEBUG` mode, changes to the *render_template* are detected, so this doesn't affect the caching. Some changes however, will not be detected (e.g. include files). A quick way to clear memcache, is by using nc/ncat/netcat:

```
echo flush_all | nc localhost 11211
```

When needed, include `FLUENT_CONTENTS_CACHE_OUTPUT = False` in the settings file.

3.2.3 Customizing the admin interface

The admin rendering of a plugin is - by design - mostly controlled outside the plugin class. However, the *ContentPlugin* class does provide a controlled set of options to configure the admin interface. For example, the `fieldsets` attribute was already mentioned in the previous *example code* page:

```
@plugin_pool.register
class AnnouncementBlockPlugin(ContentPlugin):
    model = AnnouncementBlockItem
```

(continues on next page)

(continued from previous page)

```
    render_template = "plugins/announcementblock.html"
    category = _("Simple blocks")

    fieldsets = (
        (None, {
            'fields': ('title', 'body',)
        }),
        (_("Button"), {
            'fields': ('button_text', 'url',)
        })
    )
```

The other options are documented here.

Internally, the plugin is rendered in the admin as an inline model, but this is out of scope for the plugin code. In a different context (e.g. frontend editing) this could easily by replaced by another container format.

General metadata

The following attributes control the appearance in the plugin `<select>` box:

- *`verbose_name`* - The title of the plugin, which reads the `verbose_name` of the model by default.
- *`category`* - The title of the category.

Changing the admin form

The following attributes control the overall appearance of the plugin in the admin interface:

- *`admin_form_template`* - The template to render the admin interface with.
- *`admin_init_template`* - This optional template is inserted once at the top of the admin interface.
- *`form`* - The form class to use in the admin form. It should inherit from the *`ContentItemForm`* class.
- `fieldsets` - A tuple of fieldsets, similar to the `ModelAdmin` class.

The *`admin_form_template`* is used for example by the *code* plugin. It displays the "language" and "line number" fields at a single line. Other plugins, like the *rawhtml* and *text* plugins use this setting to hide the form labels. The available templates are:

- `admin/fluent_contents/contentitem/admin_form.html` (the default template)
- `admin/fluent_contents/contentitem/admin_form_without_labels.html` aka *`ContentPlugin.ADMIN_TEMPLATE_WITHOUT_LABELS`*.

The *`admin_init_template`* can be used by plugins that need to add some template-based initialization. The *text* plugin uses this for example to initialize the WYSIWYG editor.

Changing the admin fields

The following attributes control the overall appearance of form fields in the admin interface:

- *`raw_id_fields`* - A tuple of field names, which should not be displayed as a selectbox, but as ID field.
- *`filter_vertical`* - A tuple of field names to display in a vertical filter.
- *`filter_horizontal`* - A tuple of field names to display in a horizontal filter.

- *radio_fields* - A dictionary listing all fields to display as radio choice. The key is the field name, the value can be `admin.HORIZONTAL` / `admin.VERTICAL`.
- *prepopulated_fields* - A dictionary listing all fields to auto-complete. This can be used for slug fields, and works identically to the `prepopulated_fields` attribute of the `ModelAdmin` class.
- *formfield_overrides* - A dictionary to override form field attributes. Unlike the regular `ModelAdmin` class, both classes and field names can be used as dictionary key. For example, to specify the `max_value` of an `IntegerField` use:

```
formfield_overrides = {
    'fieldname': {
        'max_value': 900
    },
}
```

- *readonly_fields* - A list of fields to display as readonly.

Custom model fields

New in version 0.9.0.

To maintain consistency between plugins, this package provides a few additional model fields which plugins can use. By default, these fields use the standard Django model fields. When one of the *optional packages* is installed, the fields will use those additional features:

- *fluent_contents.extensions.PluginFileField* - The file field uses `FileField` by default. It displays a file browser when django-any-imagefield is installed.
- *fluent_contents.extensions.PluginHtmlField* - The HTML field displays the WYSIWYG editor, which is also used by the *text plugin*.
- *fluent_contents.extensions.PluginImageField* - The file field uses `ImageField` by default. It displays a file browser when django-any-imagefield is installed.
- *fluent_contents.extensions.PluginUrlField* - The URL field uses `URLField` by default. It displays a URL selector for internal models when django-any-urlfield is installed.

Whenever your plugin uses an image field, file field, WISYWIG editor, or URL field that may refer to internal URL's, consider using these classes instead of the regular Django fields.

Adding CSS to the admin interface

The plugin allows to define a `class Media` with the CSS files to include in the admin interface. For example:

```
class Media:
    css = {
        'screen': ('plugins/myplugin/plugin_admin.css',)
    }
```

By default, all paths are relative to the `STATIC_URL` of the Django project.

Each content item has a `.inline-ModelName` class in the admin interface. This can be used to apply CSS rules to the specific plugin only.

For example, the `<textarea>` of a `RawHtmlItem` model form can be styled using:

```
.inline-RawHtmlItem textarea.vLargeTextField {
  /* allow the OS to come up with something better then Courier New */
  font-family: "Consolas", "Menlo", "Monaco", "Lucida Console", "Liberation Mono",
↪"DejaVu Sans Mono", "Bitstream Vera Sans Mono", "Courier New", monospace;
  padding: 5px; /* 3px is standard */
  width: 606px;
}
```

Adding JavaScript behavior

In a similar way, JavaScript can be added to the admin interface:

```
class Media:
    js = (
        'plugins/myplugin/plugin_admin.js',
    )
```

Note however, that content items can be dynamically added or removed in the admin interface. Hence the JavaScript file should register itself as "view handler". A view handler is called whenever the user adds or removes a content item in the admin interface.

In case of the Announcement Block plugin, the general layout of the file would look like:

```
(function($){

  var AnnouncementBlockHandler = {

      enable: function($contentitem) {
          var inputs = $contentitem.find("input");
          // ... update the items
      },

      disable: function($contentitem) {
          // deinitialize, if needed
      }
  };

  // Register the view handler for the 'AnnouncementBlockItem' model.
  fluent_contents.plugins.registerViewHandler('AnnouncementBlockItem',␣
↪AnnouncementBlockHandler);

})(window.jQuery || django.jQuery);
```

This mechanism can be used to initialize a WYSIWYG editor, or bind custom events to the DOM elements for example. While it's not demonstrated by the current bundled plugins, you can implement an inline preview/edit switch this way!

The view handler is a JavaScript object, which should have the following methods:

enable(*$contentitem*)
: The `enable()` function is called whenever the user adds the content item. It receives a `jQuery` object that points to the root of the content item object in the DOM.

disable(*$contentitem*)
: The `disable()` function is called just before the user removes the content item. It receives a `jQuery` object that points to the root of the content item object in the DOM.

initialize(*$formset_group*)
: The `initialize()` function is not required. In case it exists, it will be called once at the start of the page. It's called right after all plugins are initialized, but just before they are moved to the proper location in the DOM.

Note that the `enable()` and `disable()` functions can be called multiple times, because it may also be called when a content item is moved to another placeholder.

In the `initialize()` function, the following jQuery selectors can be used:

- `.inline-contentitem-group` selects all formsets which contain content item forms.
- `.inline-ModelName-group` selects the formset which contain the content items of a plugin.
- `.inline-ModelName` selects each individual formset item in which a form is displayed, including the empty form placeholder.
- `.inline-ModelName:not(.empty-form)` selects all active formset items.
- `.inline-ModelName.empty-form` selects the placeholder formset which is used to create new items.

After the `initialize()` function has run, the items are moved to the apropriate placeholders, so the group selectors can only select all items reliably in the `initialize()` function. The other selectors remain valid, as they operate on individual elements.

CHAPTER 4

Advanced topics

4.1 Multilingual support

New in version 1.0.

django-fluent-contents supports creating page content in multiple languages. As all content items are added to a "parent" object, some support on the parent side is needed to use this feature. The benefit however, is that you can use *django-fluent-contents* with any multilingual package.

4.1.1 Installation

All created content items have a `language_code` attribute, to identify the language an item was created in. This attribute is filled using the language of the parent object those content items are created for.

The current language can be detected by providing:

- A `get_current_language()` method on the parent object. This covers all models that use django-parler for translation support.
- A `language_code` attribute or property on the parent object.

When such properly is found on the parent, all rendering functions use that information too. For example, only content items that match the parent's language will be rendered.

4.1.2 Rendering items

Django projects can use a different language per request or URL, using any of these methods:

- Running a second instance with a different `LANGUAGE_CODE` setting.
- Using `LocaleMiddleware` in combination with `i18n_patterns()`.
- Writing code that calls `translation.activate()` directly.

By default, all content items are rendered in the language they are created in. When you switch the language, the item will still appear in the original language. It uses `translation.override` during the rendering.

Without such strict policy, you risk combining database content of the original language, and `{% trans ".." %}` content in the current language. The output would become a mix of languages, and even be stored in the cache this way.

When you deal with this explicitly, this behavior can be disabled. There are some plugin settings available:

- *`cache_output_per_language`* - Cache each rendering in the currently active language. This is perfect for using `{% trans ".." %}` tags in the template and items rendered in fallback languages too.
- *`render_ignore_item_language`* - Don't change the language at all. This is typically desired when *`cache_output`* is completely disabled. The `get_language()` points to the active site language.

4.1.3 Fallback languages

When a page is not translated, the system can be instructed to render the fallback language. The fallback language has to be defined in *FLUENT_CONTENTS_DEFAULT_LANGUAGE_CODE*.

In the templates, use:

```
{% render_placeholder ... fallback=True %}
```

The *`render_placeholder()`* function also has a `fallback_language` parameter.

4.2 Creating a CMS system

Besides the *`PlaceholderField`* class, the *fluent_contents* module also provides additional admin classes to build a CMS interface.

The main difference between the CMS interface, and *`PlaceholderField`* class is that the placeholders will be created dynamically based on the template of the current page. Instead of displaying placeholders inline in the form, the placeholder content is displayed in a separate tabbar interface.

The features include:

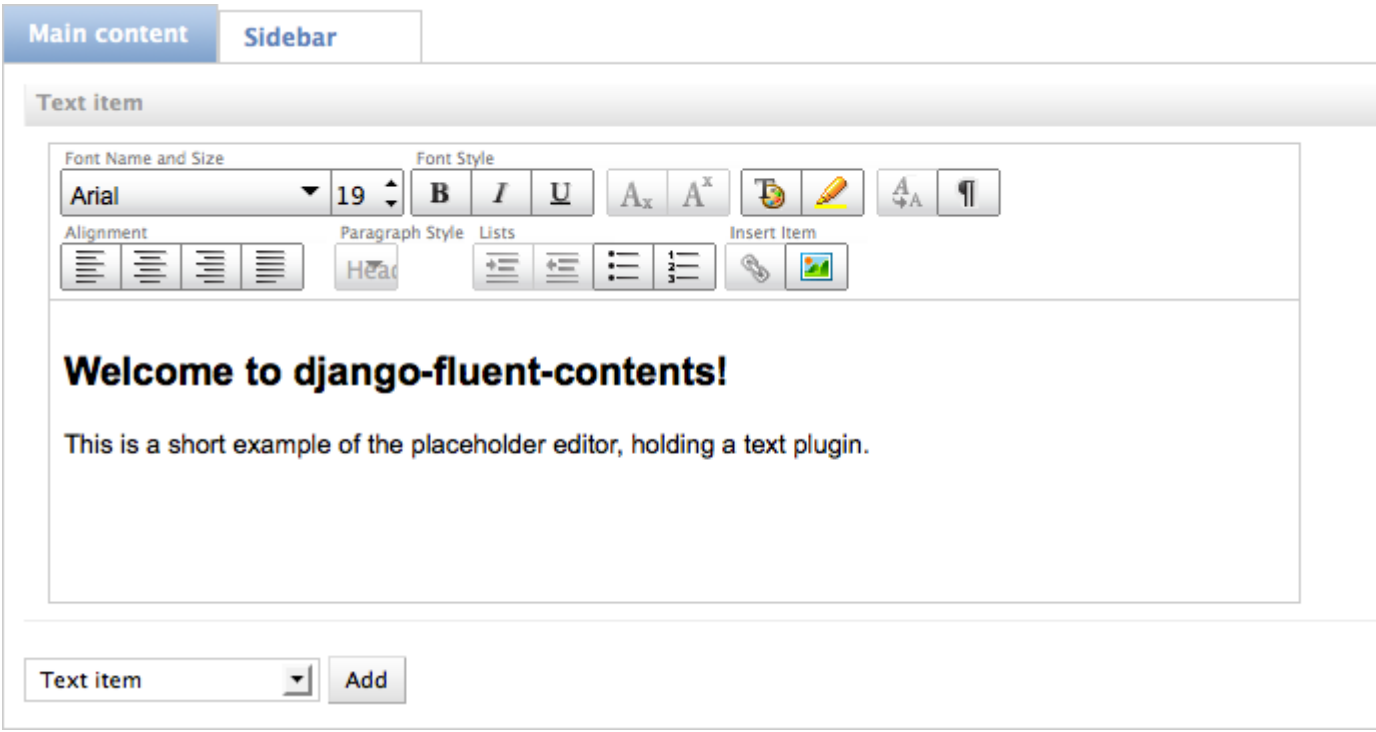
- Detecting placeholders from a Django template.
- Automatically rearrange content items when the layout changes.
- Allow usage with any parent model.

In the source distribution, see the `example.simplecms` package for a working demonstration.

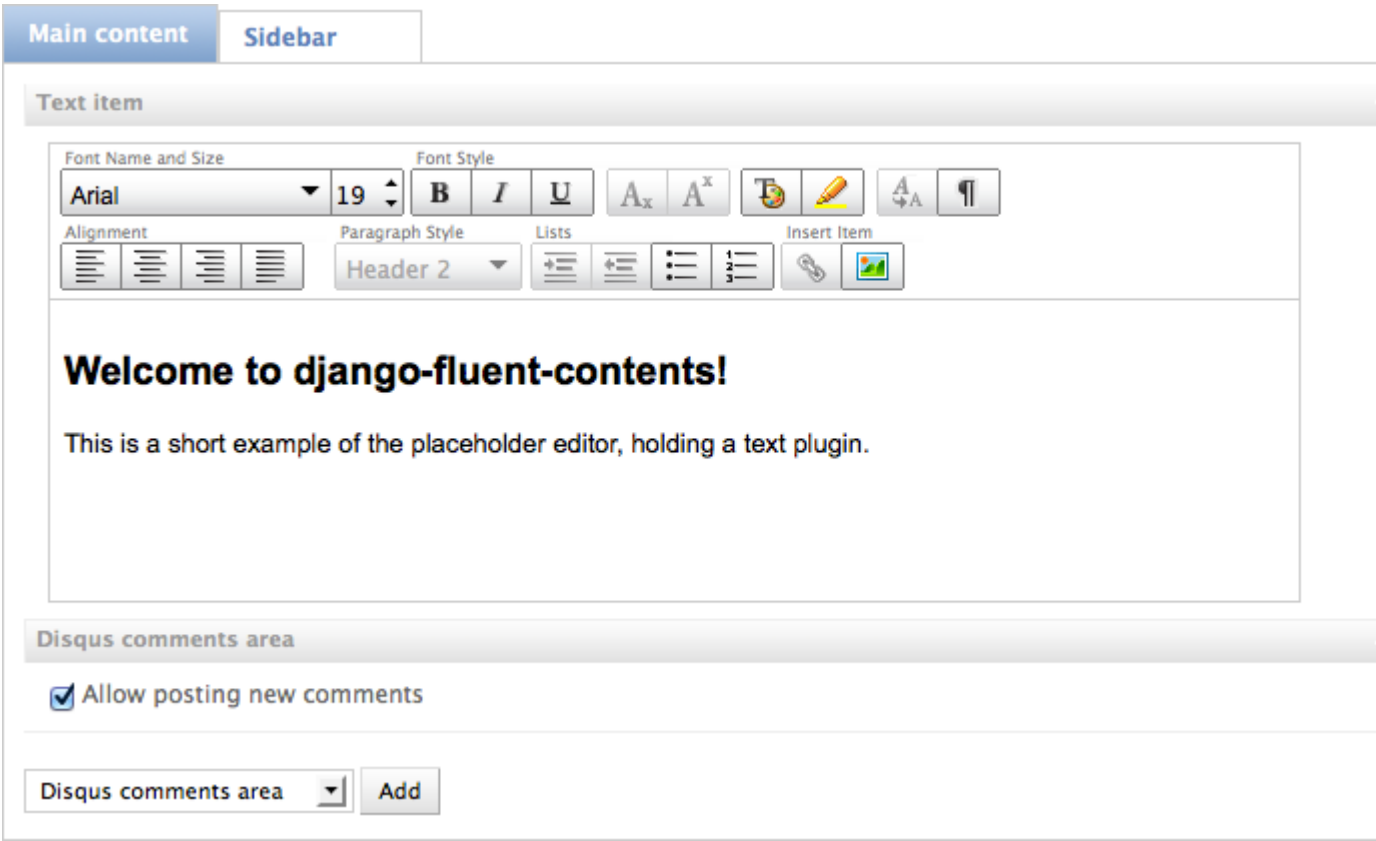
See also:

The django-fluent-pages application is built on top of this API, and provides a ready-to-use CMS that can be implemented with minimal configuration effort. To build a custom CMS, the API documentation of the *fluent_contents.admin* module provides more details of the classes.

The final appearance of the CMS would look something like:



Users can add various plugins to the page, for example by the DISQUS plugin:



4.2.1 The basic setup

The start of the admin interface, is the current interface for the CMS *Page* object. To display the "placeholder editor", the admin screen needs to inherit from *PlaceholderEditorAdmin*, and implement the `get_placeholder_data()` method.

```
from django.contrib import admin
from fluent_contents.admin import PlaceholderEditorAdmin
from fluent_contents.analyzer import get_template_placeholder_data
from .models import Page

class PageAdmin(PlaceholderEditorAdmin):

    def get_placeholder_data(self, request, obj):
        # Tell the base class which tabs to create
        template = self.get_page_template(obj)
        return get_template_placeholder_data(template)


    def get_page_template(self, obj):
        # Simple example that uses the template selected for the page.
        if not obj:
            return get_template(appconfig.SIMPLECMS_DEFAULT_TEMPLATE)
        else:
            return get_template(obj.template_name or appconfig.SIMPLECMS_DEFAULT_
↪TEMPLATE)

admin.site.register(Page, PageAdmin)
```

Now, the placeholder editor will show tabs for each placeholder. The placeholder editor is implemented as a `InlineModelAdmin`, so it will be displayed nicely below the standard forms.

The `get_placeholder_data()` method tells the "placeholder editor" which tabbar items it should create. It can use the *get_template_placeholder_data()* function for example to find the placeholders in the template.

Variation for django-mptt

For CMS systems that are built with django-mptt, the same *PlaceholderEditorAdmin* can be used thanks to the method resolution order (MRO) that Python has:

```
from mptt.admin import MPTTModelAdmin
from fluent_contents.admin import PlaceholderEditorAdmin

class PageAdmin(PlaceholderEditorAdmin, MPTTModelAdmin):

    def get_placeholder_data(self, request, obj):
        # Same code as above
        pass
```

Optional model enhancements

The *Page* object of a CMS does not require any special fields.

Optionally, the *PlaceholderRelation* and *ContentItemRelation* fields can be added to allow traversing from the parent model to the *Placeholder* and *ContentItem* classes. This also causes the admin to display any *Placeholder* and *ContentItem* objects that will be deleted on removing the page.

```
from django.db import models
from fluent_contents.models import PlaceholderRelation, ContentItemRelation
from . import appconfig


class Page(models.Model):
    title = models.CharField("Title", max_length=200)
    template_name = models.CharField("Layout", max_length=255, choices=appconfig.
↪SIMPLECMS_TEMPLATE_CHOICES)

    # ....

    placeholder_set = PlaceholderRelation()
    contentitem_set = ContentItemRelation()
```

4.2.2 Dynamic layout switching

The `example` application also demonstrates how to switch layouts dynamically. This happens entirely client-side. There is a public JavaScript API available to integrate with the layout manager.

`fluent_contents.layout.`**`onInitialize`**(*callback*)
: Register a function this is called when the module initializes the layout for the first time.

  By letting the handler return `true`, it will abort the layout initialization. The handler will be required to call `fluent_contents.loadLayout()` manually instead. This feature is typically used to restore a previous client-side selection of the user, instead of loading the last known layout at the server-side.

`fluent_contents.layout.`**`expire`**`()`
: Hide the placeholder tabs, but don't remove them yet. This can be used when the new layout is being fetched; the old content will be hidden and is ready to move.

`fluent_contents.layout.`**`load`**(*layout*)
: Load the new layout, this will create new tabs and move the existing content items to the new location. Content items are migrated to the apropriate placeholder, first matched by slot name, secondly matched by role.

  The `layout` parameter should be a JSON object with a structure like:

```
var layout = {
    'placeholders': [
        {'title': "Main content", 'slot': "main", 'role': "m", 'allowed_plugins':␣
↪["TextPlugin"]},
        {'title': "Sidebar", 'slot': "sidebar", 'role': "s", 'fallback_language':␣
↪true, 'allowed_plugins': ["TextPlugin", "PicturePlugin"]},
    ]
}
```

  The contents of each placeholder item is identical to what the *`as_dict()`* method of the *`PlaceholderData`* class returns.

`fluent_contents.tabs.`**`show`**(*animate*)
: Show the content placeholder tab interface.

`fluent_contents.tabs.`**`hide`**(*animate*)
: Hide the content placeholder tab interface. This can be used in case no layout is selected.

Note: Other JavaScript functions of the content placeholder editor that live outside the `fluent_contents` names-

pace are private, and may be changed in future releases.

CHAPTER 5

API documentation

5.1 API documentation

5.1.1 fluent_contents.admin

The admin classes of *fluent_contents* provide the necessary integration with existing admin interfaces.

The placeholder interfaces are implemented by model admin inlines, so they can be added to almost any admin interface. To assist in configuring the admin interface properly, there are a few base classes:

- For models with a `PlaceholderField`, use the `PlaceholderFieldAdmin` as base class.
- For CMS models, use the `PlaceholderEditorAdmin` as base class.

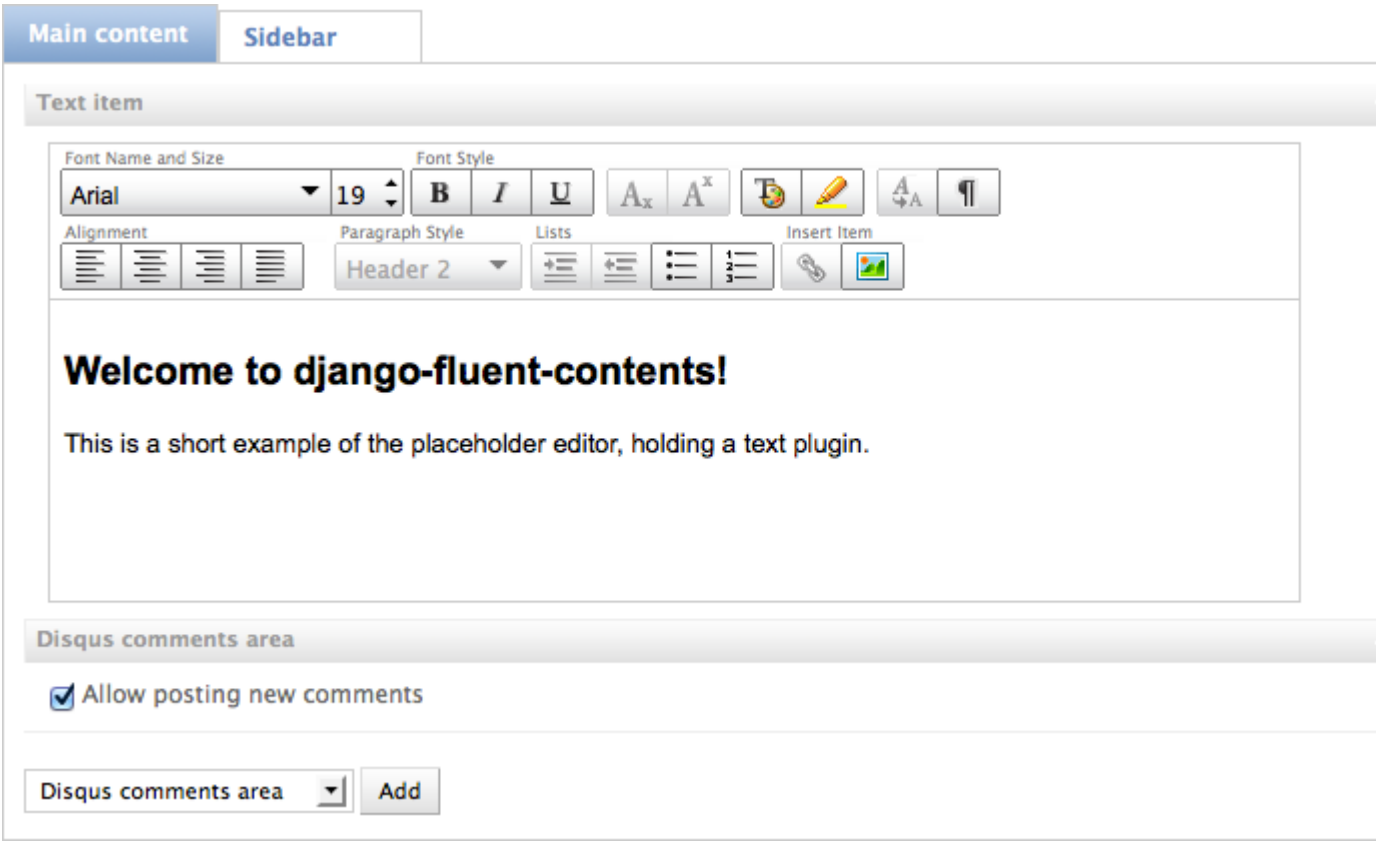
Both classes ensure the placeholders and inlines are properly setup.

The `PlaceholderEditorAdmin` class

class fluent_contents.admin.**PlaceholderEditorAdmin**(*model*, *admin_site*)

The base functionality for `ModelAdmin` dialogs to display a placeholder editor with plugins. It loads the inlines using `get_extra_inlines()`.

It loads the `PlaceholderEditorInline`, which displays each placeholder in separate tabs:



get_extra_inlines()
Return the extra inlines for the placeholder editor. It loads the *placeholder_inline* first, followed by the inlines for the *ContentItem* classes.

get_inline_instances(*request*, **args*, ***kwargs*)
Create the inlines for the admin, including the placeholder and contentitem inlines.

get_placeholder_data_view(*request*, *object_id*)
Return the placeholder data as dictionary. This is used in the client for the "copy" functionality.

placeholder_inline
alias of *PlaceholderEditorInline*

save_formset(*request*, *form*, *formset*, *change*)
Given an inline formset save it to the database.

The PlaceholderFieldAdmin class

class fluent_contents.admin.**PlaceholderFieldAdmin**(*model*, *admin_site*)
The base functionality for ModelAdmin dialogs to display placeholder fields.

This class loads the *ContentItem* inlines, and initializes the frontend editor for the *PlaceholderField*. The placeholder will be displayed in the admin:

formfield_for_dbfield(*db_field*, ***kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

get_all_allowed_plugins()
Return which plugins are allowed by the placeholder fields.

get_form(*request*, *obj=None*, ***kwargs*)
Return a Form class for use in the admin add view. This is used by add_view and change_view.

get_placeholder_data(*request*, *obj=None*)
Return the data of the placeholder fields.

placeholder_inline
alias of `PlaceholderFieldInline`

The `PlaceholderEditorBaseMixin` class

class fluent_contents.admin.**PlaceholderEditorBaseMixin**

Base interface/mixin for a `ModelAdmin` to provide the *`PlaceholderEditorInline`* with initial data. This class is implemented by the *`PlaceholderEditorAdmin`* and *`PlaceholderFieldAdmin`* classes.

get_all_allowed_plugins()

Return all plugin categories which can be used by placeholder content. By default, all plugins are allowed. Individual slot names may further limit the plugin set.

Return type list of *`ContentPlugin`*

get_placeholder_data(*request*, *obj=None*)

Return the placeholders that the editor should initially display. The function should return a list of *`PlaceholderData`* classes. These classes can either be instantiated manually, or read from a template using the *fluent_contents.analyzer* module for example.

The `PlaceholderEditorInline` class

class fluent_contents.admin.**PlaceholderEditorInline**(*parent_model*, *admin_site*)

The placeholder editor, implemented as an admin inline. It displays tabs for each inline placeholder, and displays *`ContentItem`* plugins in the tabs.

It should be inserted in the `ModelAdmin.inlines` before the inlines that the *`get_content_item_inlines()`* function generates. The ContentItem inlines look for the `Placeholder` object that was created just before their invocation.

To fetch the initial data, the inline will attempt to find the parent model, and call *`get_placeholder_data()`*. When the admin models inherit from *`PlaceholderEditorAdmin`* or *`PlaceholderFieldAdmin`* this will be setup already.

formset

alias of `PlaceholderInlineFormSet`

get_all_allowed_plugins()

Return *all* plugin categories which can be used by placeholder content. This is the sum of all allowed plugins by the various slots on the page. It accesses the parent *`PlaceholderEditorBaseMixin`* by default to request the information. This field is used in the template.

get_formset(*request*, *obj=None*, ***kwargs*)

Pre-populate formset with the initial placeholders to display.

model

alias of `fluent_contents.models.db.Placeholder`

The `get_content_item_inlines` function

fluent_contents.admin.**get_content_item_inlines**(*plugins=None*, *base=<class 'fluent_contents.admin.contentitems.BaseContentItemInline'>*)

Dynamically generate genuine django inlines for all registered content item types. When the *plugins* parameter is `None`, all plugin inlines are returned.

5.1.2 fluent_contents.analyzer

Analyze the templates for placeholders of this module.

fluent_contents.analyzer.**get_template_placeholder_data**(*template*)
Return the placeholders found in a template, wrapped in a `PlaceholderData` object.

This function looks for the *PagePlaceholderNode* nodes in the template, using the `get_node_instances()` function of django-template-analyzer.

Parameters **template** – The Template object, or nodelist to scan.

Return type list of *PlaceholderData*

5.1.3 fluent_contents.cache

Functions for caching.

fluent_contents.cache.**get_placeholder_cache_key**(*placeholder*, *language_code*)
Return a cache key for an existing placeholder object.

This key is used to cache the entire output of a placeholder.

fluent_contents.cache.**get_placeholder_cache_key_for_parent**(*parent_object*, *placeholder_name*, *language_code*)
Return a cache key for a placeholder.

This key is used to cache the entire output of a placeholder.

fluent_contents.cache.**get_rendering_cache_key**(*placeholder_name*, *contentitem*)
Return a cache key for the content item output.

See also:

The `ContentItem.clear_cache()` function can be used to remove the cache keys of a retrieved object.

5.1.4 fluent_contents.extensions

Special classes to extend the module; e.g. plugins.

The extension mechanism is provided for projects that benefit from a tighter integration then the Django URLconf can provide.

The API uses a registration system. While plugins can be easily detected via `__subclasses__()`, the register approach is less magic and more explicit. Having to do an explicit register ensures future compatibility with other API's like reversion.

The `ContentPlugin` class

class fluent_contents.extensions.**ContentPlugin**
The base class for all content plugins.

A plugin defines the rendering for a *ContentItem*, settings and presentation in the admin interface. To create a new plugin, derive from this class and call *plugin_pool.register* to enable it. For example:

```
from fluent_contents.extensions import plugin_pool, ContentPlugin

@plugin_pool.register
class AnnouncementBlockPlugin(ContentPlugin):
    model = AnnouncementBlockItem
    render_template = "plugins/announcementblock.html"
    category = _("Simple blocks")
```

As minimal configuration, specify the *model* and *render_template* fields. The *model* should be a subclass of the *ContentItem* model class.

Note: When the plugin is registered in the `plugin_pool`, it will be instantiated only once. It is therefore not possible to store per-request state at the plugin object. This is similar to the behavior of the `ModelAdmin` classes in Django.

To customize the admin, the *admin_form_template* and *form* can be defined. Some well known properties of the `ModelAdmin` class can also be specified on plugins; such as:

- `fieldsets`
- `filter_horizontal`
- `filter_vertical`
- `prepopulated_fields`
- `radio_fields`
- `raw_id_fields`
- `readonly_fields`
- A `class Media` to provide extra CSS and JavaScript files for the admin interface.

The rendered output of a plugin is cached by default, assuming that most content is static. This also avoids extra database queries to retrieve the model objects. In case the plugin needs to output content dynamically, include `cache_output = False` in the plugin definition.

ADMIN_TEMPLATE_WITHOUT_LABELS = 'admin/fluent_contents/contentitem/admin_form_without_
Alternative template for the view.

ADVANCED = 'Advanced'
New in version 1.1.

Category for advanced plugins (e.g. raw HTML, iframes)

HORIZONTAL = 1
New in version 0.8.5.

The `HORIZONTAL` constant for the *radio_fields*.

INTERACTIVITY = 'Interactivity'
New in version 1.1.

Category for interactive plugins (e.g. forms, comments)

MEDIA = 'Media'
New in version 1.1.

Category for media

PROGRAMMING = 'Programming'
New in version 1.1.

Category for programming plugins

VERTICAL = 2
New in version 0.8.5.

The `VERTICAL` constant for the *radio_fields*.

admin_form_template = 'admin/fluent_contents/contentitem/admin_form.html'
The template to render the admin interface with

admin_init_template = None
An optional template which is included in the admin interface, to initialize components (e.g. JavaScript)

cache_output = True
By default, rendered output is cached, and updated on admin changes.

cache_output_per_language = False
New in version 1.0.

Cache the plugin output per language. This can be useful for sites which either:

- Display fallback content on pages, but still use `{% trans %}` inside templates.
- Dynamically switch the language per request, and *share* content between multiple languages.

This option does not have to be used for translated CMS pages, as each page can have it's own set of *ContentItem* objects. It's only needed for rendering the *same* item in different languages.

cache_output_per_site = False
New in version 0.9.

Cache the plugin output per SITE_ID.

cache_supported_language_codes = ['af', 'ar', 'ar-dz', 'ast', 'az', 'bg', 'be', 'bn',
New in version 1.0.

Tell which languages the plugin will cache. It defaults to the language codes from the LANGUAGES setting.

cache_timeout = <object object>
Set a custom cache timeout value

category = None
The category title to place the plugin into. This is only used for the "Add Plugin" menu. You can provide a string here, `ugettext_lazy()` or one of the predefined constants (*MEDIA*, `INTERACTIVITY:`, *PROGRAMMING* and *ADVANCED*).

filter_horizontal = ()
The fields to display in a horizontal filter

filter_vertical = ()
The fields to display in a vertical filter

form
alias of `fluent_contents.forms.ContentItemForm`

formfield_overrides = {}
Overwritten formfield attributes, e.g. the 'widget'. Allows both the class and fieldname as key.

frontend_media
New in version 1.0.

The frontend media, typically declared using a `class FrontendMedia` definition.

get_cached_output(*placeholder_name*, *instance*)
New in version 0.9: Return the cached output for a rendered item, or `None` if no output is cached.

This method can be overwritten to implement custom caching mechanisms. By default, this function generates the cache key using *get_output_cache_key()* and retrieves the results from the configured Django cache backend (e.g. memcached).

get_context(*request*, *instance*, ***kwargs*)
Return the context to use in the template defined by `render_template` (or *get_render_template()*). By default, it returns the model instance as `instance` field in the template.

get_frontend_media(*instance*)
Return the frontend media for a specific instance. By default, it returns `self.frontend_media`, which derives from the `class FrontendMedia` of the plugin.

get_model_instances()
Return the model instances the plugin has created.

get_output_cache_base_key(*placeholder_name*, *instance*)
New in version 1.0: Return the default cache key, both *get_output_cache_key()* and *get_output_cache_keys()* rely on this. By default, this function generates the cache key using *get_rendering_cache_key()*.

get_output_cache_key(*placeholder_name*, *instance*)
New in version 0.9: Return the default cache key which is used to store a rendered item. By default, this function generates the cache key using *get_output_cache_base_key()*.

get_output_cache_keys(*placeholder_name*, *instance*)
New in version 0.9: Return the possible cache keys for a rendered item.

This method should be overwritten when implementing a function *set_cached_output()* method or when implementing a *get_output_cache_key()* function. By default, this function generates the cache key using *get_output_cache_base_key()*.

get_render_template(*request*, *instance*, ***kwargs*)
Return the template to render for the specific model *instance* or *request*, By default it uses the `render_template` attribute.

get_search_text(*instance*)
Return a custom search text for a given instance.

Note: This method is called when *search_fields* is set.

---

model = None
The model to use, must derive from *fluent_contents.models.ContentItem*.

name
Return the classname of the plugin, this is mainly provided for templates. This value can also be used in *PluginPool()*.

prepopulated_fields = {}
Fields to automatically populate with values

radio_fields = {}
The fields to display as radio choice. For example:

```
radio_fields = {
    'align': ContentPlugin.VERTICAL,
}
```

The value can be *ContentPlugin.HORIZONTAL* or *ContentPlugin.VERTICAL*.

raw_id_fields = ()
The fields to display as raw ID

readonly_fields = ()
The fields to display as readonly.

redirect(*url*, *status=302*)
New in version 1.0.

Request a redirect to be performed for the user. Usage example:

```
def get_context(self, request, instance, **kwargs):
    context = super(IdSearchPlugin, self).get_context(request, instance,␣
↪**kwargs)

    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            self.redirect("/foo/")
    else:
        form = MyForm()

    context['form'] = form
    return context
```

To handle redirects, *fluent_contents.middleware.HttpRedirectRequestMiddleware* should be added to the MIDDLEWARE_CLASSES.

render(*request*, *instance*, ***kwargs*)
The rendering/view function that displays a plugin model instance.

Parameters

- **instance** – An instance of the model the plugin uses.
- **request** – The Django HttpRequest class containing the request parameters.
- **kwargs** – An optional slot for any new parameters.

To render a plugin, either override this function, or specify the *render_template* variable, and optionally override *get_context()*. It is recommended to wrap the output in a tag, to prevent the item from being displayed right next to the previous plugin.

New in version 1.0: The function may either return a string of HTML code, or return a *ContentItemOutput* object which holds both the CSS/JS includes and HTML string. For the sake of convenience and simplicity, most examples only return a HTML string directly.

When the user needs to be redirected, simply return a HttpResponseRedirect or call the *redirect()* method.

To render raw HTML code, use mark_safe() on the returned HTML.

render_error(*error*)
A default implementation to render an exception.

render_ignore_item_language = False
New in version 1.0.

By default, the plugin is rendered in the language_code it's written in. It can be disabled explicitly in case the content should be rendered language agnostic. For plugins that cache output per language, this will be done already.

See also: *cache_output_per_language*

render_template = None
The template to render the frontend HTML output.

render_to_string(*request*, *template*, *context*, *content_instance=None*)
Render a custom template with the *PluginContext* as context instance.

search_fields = []
Define which fields could be used for indexing the plugin in a site (e.g. haystack)

search_output = None
Define whether the full output should be used for indexing.

set_cached_output(*placeholder_name*, *instance*, *output*)
New in version 0.9: Store the cached output for a rendered item.

This method can be overwritten to implement custom caching mechanisms. By default, this function generates the cache key using *get_rendering_cache_key()* and stores the results in the configured Django cache backend (e.g. memcached).

When custom cache keys are used, also include those in *get_output_cache_keys()* so the cache will be cleared when needed.

Changed in version 1.0: The received data is no longer a HTML string, but *ContentItemOutput* object.

type_id
Shortcut to retrieving the ContentType id of the model.

type_name
Return the classname of the model, this is mainly provided for templates.

verbose_name
The title for the plugin, by default it reads the `verbose_name` of the model.

The `PluginPool` class

class fluent_contents.extensions.**PluginPool**
The central administration of plugins.

__init__()
Initialize self. See help(type(self)) for accurate signature.

get_allowed_plugins(*placeholder_slot*)
Return the plugins which are supported in the given placeholder name.

get_model_classes()
Return all *ContentItem* model classes which are exposed by plugins.

get_plugin_by_model(*model_class*)
Return the corresponding plugin for a given model.

You can also use the *ContentItem.plugin* property directly. This is the low-level function that supports that feature.

get_plugins()
Return the list of all plugin instances which are loaded.

get_plugins_by_name(**names*)
Return a list of plugins by plugin class, or name.

register(*plugin*)
Make a plugin known to the CMS.

Parameters **plugin** (*ContentPlugin*) – The plugin class, deriving from *ContentPlugin*.

The plugin will be instantiated once, just like Django does this with `ModelAdmin` classes. If a plugin is already registered, this will raise a *`PluginAlreadyRegistered`* exception.

The `plugin_pool` attribute

`fluent_contents.extensions.`**`plugin_pool`**
The global plugin pool, a instance of the *`PluginPool`* class.

Model fields

New in version 0.9.0.

The model fields ensure a consistent look and feel between plugins. It's recommended to use these fields instead of the standard Django counterparts, so all plugins have a consistent look and feel. See *Optional integration with other packages* for more details.

class `fluent_contents.extensions.`**`PluginFileField`**(*verbose_name=None*, *name=None*, *upload_to=''*, *storage=None*, ***kwargs*)
A file upload field for plugins.

Use this instead of the standard `FileField`.

class `fluent_contents.extensions.`**`PluginHtmlField`**(**args*, ***kwargs*)
A HTML field for plugins.

This field is replaced with a django-wysiwyg editor in the admin.

`__init__`(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

`formfield`(***kwargs*)
Return a django.forms.Field instance for this field.

`to_python`(*html*)
Convert the input value into the expected Python data type, raising django.core.exceptions.ValidationError if the data can't be converted. Return the converted value. Subclasses should override this.

class `fluent_contents.extensions.`**`PluginImageField`**(*verbose_name=None*, *name=None*, *width_field=None*, *height_field=None*, ***kwargs*)
An image upload field for plugins.

Use this instead of the standard `ImageField`.

class `fluent_contents.extensions.`**`PluginUrlField`**(**args*, ***kwargs*)
An URL field for plugins.

Use this instead of the standard `URLField`.

Classes for custom forms

New in version 0.9.0: The canonical place to import this class has moved, previously it was available via `fluent_contents.forms`.

class fluent_contents.extensions.**ContentItemForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

The base form for custom *ContentItem* types. It displays the additional meta fields as hidden fields.

When creating custom admin forms (e.g. to add validation for specific fields), use this class as base to ensure all fields are properly set up.

Other classes

class fluent_contents.extensions.**PluginContext**(*request*, *dict=None*, *current_app=None*)

A template Context class similar to `RequestContext`, that enters some pre-filled data. This ensures that variables such as `STATIC_URL` and `request` are available in the plugin templates.

__init__(*request*, *dict=None*, *current_app=None*)

Initialize self. See help(type(self)) for accurate signature.

exception fluent_contents.extensions.**PluginAlreadyRegistered**

Raised when attemping to register a plugin twice.

exception fluent_contents.extensions.**PluginNotFound**

Raised when the plugin could not be found in the rendering process.

5.1.5 fluent_contents.middleware

The `HttpRedirectRequestMiddleware` class

class fluent_contents.middleware.**HttpRedirectRequestMiddleware**(*get_response=None*)

New in version 1.0.

Middleware that handles requests redirects, requested by plugins using *ContentPlugin.redirect()*. This needs to be added to `MIDDLEWARE_CLASSES` for redirects to work.

Some background:

Since plugins can't return a `HttpResponseRedirect` response themselves, because they are part of the template rendering process. Instead, Python exception handling is used to abort the rendering and return the redirect.

When *ContentPlugin.redirect()* is called, a `HttpRedirectRequest` exception is raised. This middleware handles the exception, and returns the proper redirect response.

---

process_exception(*request*, *exception*)

Return a redirect response for the `HttpRedirectRequest`

process_template_response(*request*, *response*)

Patch a `TemplateResponse` object to handle the `HttpRedirectRequest` exception too.

5.1.6 fluent_contents.models

The *fluent_contents* package defines two models, for storing the content data:

- *Placeholder*
- *ContentItem*

Secondly, there are a few possible fields to add to parent models:

- *PlaceholderField*
- *PlaceholderRelation*
- *ContentItemRelation*

Finally, to exchange template data, a *PlaceholderData* object is available which mirrors the relevant fields of the *Placeholder* model.

The `Placeholder` class

class fluent_contents.models.**Placeholder**(**args*, ***kwargs*)

The placeholder groups various *ContentItem* models together in a single compartment. It is the reference point to render custom content. Each placeholder is identified by a *slot* name and *parent* object.

Optionally, the placeholder can have a *title*, and *role*. The role is taken into account by the client-side placeholder editor when a page switches template layouts. Content items are migrated to the apropriate placeholder, first matched by slot name, secondly matched by role.

The parent object is stored in a generic relation, so the placeholder can be used with any custom object. By adding a *PlaceholderRelation* field to the parent model, the relation can be traversed backwards. From the placeholder, the `contentitem_set` can be traversed to find the associated *ContentItem* objects. Since a *ContentItem* is polymorphic, the actual sub classes of the *ContentItem* will be returned by the query. To prevent this behavior, call `non_polymorphic()` on the *QuerySet*.

Parameters

- **id** (*AutoField*) – Id
- **slot** (*SlugField*) – Slot. A short name to identify the placeholder in the template code.
- **role** (*CharField*) – Role. This defines where the object is used.
- **parent_type** (ForeignKey to `ContentType`) – Parent type
- **parent_id** (*IntegerField*) – Parent id
- **title** (*CharField*) – Admin title

exception DoesNotExist

exception MultipleObjectsReturned

get_absolute_url()

Return the URL of the parent object, if it has one. This method mainly exists to support cache mechanisms (e.g. refreshing a Varnish cache), and assist in debugging.

get_allowed_plugins()

Return the plugins which are supported in this placeholder.

get_content_items(*parent=None*, *limit_parent_language=True*)

Return all models which are associated with this placeholder. Because a *ContentItem* is polymorphic, the actual sub classes of the content item will be returned by the query.

By passing the `parent` object, the items can additionally be filtered by the parent language.

get_search_text (*fallback_language=None*)
Get the search text for all contents of this placeholder.

Parameters **fallback_language** (*bool|str*) – The fallback language to use if there are no items in the current language. Passing `True` uses the default *FLUENT_CONTENTS_DEFAULT_LANGUAGE_CODE*.

Return type str

The `PlaceholderManager` class

class fluent_contents.models.**PlaceholderManager**
Extra methods for the `Placeholder.objects`.

create_for_object (*parent_object*, *slot*, *role='m'*, *title=None*)
Create a placeholder with the given parameters

get_by_slot (*parent_object*, *slot*)
Return a placeholder by key.

parent (*parent_object*)
Return all placeholders which are associated with a given parent object.

The `ContentItem` class

class fluent_contents.models.**ContentItem**(**args*, ***kwargs*)
A *ContentItem* represents a content part (also called pagelet in other systems) which is displayed in a *Placeholder*. To use the *ContentItem*, derive it in your model class:

```
class ExampleItem(ContentItem):
    # any custom fields here

    class Meta:
        verbose_name = "Example item"
```

All standard Django model fields can be used in a *ContentItem*. Some things to note:

- There are special fields for URL, WYSIWYG and file/image fields, which keep the admin styles consistent. These are the *PluginFileField*, *PluginHtmlField*, *PluginImageField* and *PluginUrlField* fields. See the *Custom model fields* section for more details.
- When adding a M2M field, make sure to override *copy_to_placeholder*, so the M2M data will be copied.

When querying the objects through the ORM, the derived instances will be returned automatically. This happens because the *ContentItem* class is polymorphic:

```
>>> from fluent_contents.models import ContentItem
>>> ContentItem.objects.all()
[<ArticleTextItem: Main article>, <RawHtmlItem: test>, <CodeItem: def foo():␣
↪print 1>,
<AnnouncementBlockItem: Test>, <ArticleTextItem: Text in sidebar>]
```

Note that the *django-polymorphic* application is written in such way, that this requires the least amount of queries necessary. When access to the polymorphic classes is not needed, call `non_polymorphic()` on the *QuerySet* to prevent this behavior:

```
>>> from fluent_contents.models import ContentItem
>>> ContentItem.objects.all().non_polymorphic()
[<ContentItem: Article text item#1 in 'Main content'>, <ContentItem: HTML code#5␣
↪in 'Main content'>, <ContentItem: Code snippet#6 in 'Main content'>,
<ContentItem: Announcement block#7 in 'Main content'>, <ContentItem: Article text␣
↪item#4 in 'Sidebar'>]
```

Being polymorphic also means the base class provides some additional methods such as:

- `get_real_instance()`
- `get_real_instance_class()`

Each *ContentItem* references both it's parent object (e.g. a page, or article), and the placeholder. While this is done mainly for internal reasons, it also provides an easy way to query all content items of a parent. The parent object is stored in a generic relation, so the *ContentItem* can be used with any custom object. By adding a *ContentItemRelation* field to the parent model, the relation can be traversed backwards.

Because the *ContentItem* references it's parent, and not the other way around, it will be cleaned up automatically by Django when the parent object is deleted.

To use a *ContentItem* in the *PlaceholderField*, register it via a plugin definition. see the *ContentPlugin* class for details.

The rendering of a *ContentItem* class happens in the associate *ContentPlugin* class. To render content items outside the template code, use the *fluent_contents.rendering* module to render the items.

Parameters

- **id** (*AutoField*) – Id
- **polymorphic_ctype** (ForeignKey to ContentType) – Polymorphic ctype
- **parent_type** (ForeignKey to ContentType) – Parent type
- **parent_id** (*IntegerField*) – Parent id
- **language_code** (*CharField*) – Language code
- **placeholder** (ForeignKey to Placeholder) – Placeholder
- **sort_order** (*IntegerField*) – Sort order

exception DoesNotExist

exception MultipleObjectsReturned

copy_to_placeholder(*placeholder*, *sort_order=None*, *in_place=False*)
Note: if you have M2M relations on the model, override this method to transfer those values.

get_absolute_url()
Return the URL of the parent object, if it has one. This method mainly exists to refreshing cache mechanisms.

get_cache_keys()
Get a list of all cache keys associated with this model. This queries the associated plugin for the cache keys it used to store the output at.

move_to_placeholder(*placeholder*, *sort_order=None*)
The object is saved afterwards.

plugin
Access the parent plugin which renders this model.

Return type *ContentPlugin*

save(**args*, ***kwargs*)
Calls `pre_save_polymorphic()` and saves the model.

The `PlaceholderField` class

class fluent_contents.models.**PlaceholderField**(*slot*, *plugins=None*, ***kwargs*)
The model field to add *ContentItem* objects to a model.

Parameters

- **slot** (*str*) – A programmatic name to identify the placeholder.
- **plugins** (*list*) – Optional, define which plugins are allowed to be used. This can be a list of names, or *ContentPlugin* references.

This class provides the form fields for the field. Use this class in a model to use it:

```
class Article(models.Model):
    contents = PlaceholderField("article_contents")
```

The data itself is stored as reverse relation in the *ContentItem* object. Hence, all contents will be cleaned up properly when the parent model is deleted.

The placeholder will be displayed in the admin:

__init__(*slot*, *plugins=None*, ***kwargs*)
Initialize the placeholder field.

contribute_to_class(*cls*, *name*, ***kwargs*)
Internal Django method to associate the field with the Model; it assigns the descriptor.

formfield(***kwargs*)
Returns a `PlaceholderFormField` instance for this database Field.

plugins
Get the set of plugins that this field may display.

rel_class
alias of `PlaceholderRel`

value_from_object(*obj*)
Internal Django method, used to return the placeholder ID when exporting the model instance.

The `PlaceholderRelation` class

class fluent_contents.models.**PlaceholderRelation**(***kwargs*)

A `GenericRelation` which can be applied to a parent model that is expected to be referenced be a *`Placeholder`*. For example:

```
class Page(models.Model):
    placeholder_set = PlaceholderRelation()
```

__init__(***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

The `ContentItemRelation` class

class fluent_contents.models.**ContentItemRelation**(***kwargs*)

A `GenericRelation` which can be applied to a parent model that is expected to be referenced by the *`ContentItem`* classes. For example:

```
class Page(models.Model):
    contentitem_set = ContentItemRelation()
```

Adding this relation also causes the admin delete page to list the *`ContentItem`* objects which will be deleted.

__init__(***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

bulk_related_objects(*objs*, *using='default'*)

Return all objects related to `objs` via this `GenericRelation`.

The `PlaceholderData` class

class fluent_contents.models.**PlaceholderData**(*slot*, *title=None*, *role=None*, *fallback_language=None*)

A wrapper with data of a placeholder node. It shares the `slot`, `title` and `role` fields with the *`Placeholder`* class.

__init__(*slot*, *title=None*, *role=None*, *fallback_language=None*)

Create the placeholder data with a slot, and optional title and role.

as_dict()

Return the contents as dictionary, for client-side export. The dictionary contains the fields:

- `slot`
- `title`
- `role`
- `fallback_language`
- `allowed_plugins`

The `ContentItemOutput` class

class fluent_contents.models.**ContentItemOutput**(*html*, *media=None*, *cacheable=True*, *cache_timeout=<object object>*)

A wrapper with holds the rendered output of a plugin, This object is returned by the *render_placeholder()* and *ContentPlugin.render()* method.

Instances can be treated like a string object, but also allows reading the `html` and `media` attributes.

__init__(*html*, *media=None*, *cacheable=True*, *cache_timeout=<object object>*)

Initialize self. See help(type(self)) for accurate signature.

The `get_parent_lookup_kwargs` function

fluent_contents.models.**get_parent_lookup_kwargs**(*parent_object*)

Return lookup arguments for the generic `parent_type` / `parent_id` fields.

Parameters **parent_object** (*Model*) – The parent object.

5.1.7 fluent_contents.rendering

This module provides functions to render placeholder content manually.

The functions are available outside the regular templatetags, so it can be called outside the templates as well.

Contents is cached in memcache whenever possible, only the remaining items are queried. The templatetags also use these functions to render the *ContentItem* objects.

fluent_contents.rendering.**get_cached_placeholder_output**(*parent_object*, *placeholder_name*)

Return cached output for a placeholder, if available. This avoids fetching the Placeholder object.

fluent_contents.rendering.**render_placeholder**(*request*, *placeholder*, *parent_object=None*, *template_name=None*, *cachable=None*, *limit_parent_language=True*, *fallback_language=None*)

Render a *Placeholder* object. Returns a *ContentItemOutput* object which contains the HTML output and `Media` object.

This function also caches the complete output of the placeholder when all individual items are cacheable.

Parameters

- **request** (*HttpRequest*) – The current request object.
- **placeholder** (*Placeholder*) – The placeholder object.
- **parent_object** – Optional, the parent object of the placeholder (already implied by the placeholder)
- **template_name** (*str | None*) – Optional template name used to concatenate the placeholder output.
- **cachable** (*bool | None*) – Whether the output is cachable, otherwise the full output will not be cached. Default: False when using a template, True otherwise.
- **limit_parent_language** (*bool*) – Whether the items should be limited to the parent language.

- **fallback_language** (*bool/str*) – The fallback language to use if there are no items in the current language. Passing `True` uses the default *FLUENT_CONTENTS_DEFAULT_LANGUAGE_CODE*.

Return type *ContentItemOutput*

`fluent_contents.rendering.`**render_content_items**(*request*, *items*, *template_name=None*, *cachable=None*)

Render a list of *ContentItem* objects as HTML string. This is a variation of the *render_placeholder()* function.

Note that the items are not filtered in any way by parent or language. The items are rendered as-is.

Parameters

- **request** (*HttpRequest*) – The current request object.
- **items** (list or queryset of *ContentItem*.) – The list or queryset of objects to render. Passing a queryset is preferred.
- **template_name** (*Optional[str]*) – Optional template name used to concatenate the placeholder output.
- **cachable** (*Optional[bool]*) – Whether the output is cachable, otherwise the full output will not be cached. Default: False when using a template, True otherwise.

Return type *ContentItemOutput*

`fluent_contents.rendering.`**render_placeholder_search_text**(*placeholder*, *fallback_language=None*)

Render a *Placeholder* object to search text. This text can be used by an indexer (e.g. haystack) to produce content search for a parent object.

Parameters

- **placeholder** (*Placeholder*) – The placeholder object.
- **fallback_language** (*bool|str*) – The fallback language to use if there are no items in the current language. Passing `True` uses the default *FLUENT_CONTENTS_DEFAULT_LANGUAGE_CODE*.

Return type str

`fluent_contents.rendering.`**get_frontend_media**(*request*)

Return the media that was registered in the request object.

Note: The output of plugins is typically cached. Changes the the registered media only show up after flushing the cache, or re-saving the items (which flushes the cache).

`fluent_contents.rendering.`**register_frontend_media**(*request*, *media*)

Add a `Media` class to the current request. This will be rendered by the `render_plugin_media` template tag.

`fluent_contents.rendering.`**is_edit_mode**(*request*)

Return whether edit mode is enabled; output is wrapped in `<div>` elements with metadata for frontend editing.

`fluent_contents.rendering.`**set_edit_mode**(*request*, *state*)

Enable the edit mode; placeholders and plugins will be wrapped in a `<div>` that exposes metadata for frontend editing.

5.1.8 fluent_contents.templatetags.fluent_contents_tags

Changed in version 1.0: The template tag library was called `placeholder_tags` in the past. The `fluent_contents_tags` module provides two template tags for rendering placeholders: It can be loaded using:

```
{% load fluent_contents_tags %}
```

A placeholder which is stored in a *`PlaceholderField`* can be rendered with the following syntax:

```
{% render_placeholder someobject.placeholder %}
```

To support CMS interfaces, placeholder slots can be defined in the template. This is done using the following syntax:

```
{% page_placeholder currentpage "slotname" %}
{% page_placeholder currentpage "slotname" title="Admin title" role="main" %}
```

The CMS interface can scan for those tags using the *fluent_contents.analyzer* module.

The `PagePlaceholderNode` class

class fluent_contents.templatetags.fluent_contents_tags.**PagePlaceholderNode**(*tag_name*, *as_var*, *parent_expr*, *slot_expr*, ***kwargs*)

The template node of the `page_placeholder` tag. It renders a placeholder of a provided parent object. The template tag can also contain additional metadata, which can be returned by scanning for this node using the *fluent_contents.analyzer* module.

__init__(*tag_name*, *as_var*, *parent_expr*, *slot_expr*, ***kwargs*)
The constructor receives the parsed arguments. The values are stored in `tagname`, `args`, `kwargs`.

get_fallback_language()
Return whether to use the fallback language.

get_role()
Return the string literal that is used in the template. The role can be "main", "sidebar" or "related", or shorted to "m", "s", "r".

get_slot()
Return the string literal that is used for the placeholder slot in the template. When the variable is not a string literal, `None` is returned.

get_title()
Return the string literal that is used in the template. The title is used in the admin screens.

get_value(*context*, **tag_args*, ***tag_kwargs*)
Return the value for the tag.

Parameters

- **tag_args** –
- **tag_kwargs** –

classmethod parse(*parser*, *token*)
Parse the node syntax:

```
{% page_placeholder parentobj slotname title="test" role="m" %}
```

The RenderPlaceholderNode class

class fluent_contents.templatetags.fluent_contents_tags.**RenderPlaceholderNode**(*tag_name*, *as_var*, **args*, ***kwargs*)

The template node of the `render_placeholder` tag. It renders the provided placeholder object.

get_value(*context*, **tag_args*, ***tag_kwargs*)
Return the value for the tag.

Parameters

- **tag_args** –
- **tag_kwargs** –

classmethod validate_args(*tag_name*, **args*, ***kwargs*)
Validate the syntax of the template tag.

The RenderContentItemsMedia class

class fluent_contents.templatetags.fluent_contents_tags.**RenderContentItemsMedia**(*tag_name*, **args*, ***kwargs*)

The template node of the `render_plugin_media` tag. It renders the media object object.

render_tag(*context*, *media_type=None*, *domain=None*)
Render the tag, with all arguments resolved to their actual values.

classmethod validate_args(*tag_name*, **args*, ***kwargs*)
Validate the syntax of the template tag.

5.1.9 fluent_contents.utils

This module provides additional small utility functions which plugins can use.

fluent_contents.utils.**validate_html_size**(*value*)
Validate whether a value can be used in a HTML `width` or `height` value. The value can either be a number, or end with a percentage sign. Raises a `ValidationError` if the value is invalid.

5.2 Changelog

5.2.1 Changes in 2.0.7 (2020-01-04)

- Fix Django 3.0 compatibility by removing `curry()` call.
- Bump setup requirements to ensure Django 3.0 compatibility.

5.2.2 Changes in 2.0.6 (2019-06-11)

- Fix Django 2.x compatibility with `disquswidgets` and `formdesignerlink` migrations.
- Fix Python error when adding a `ContentItem` on a parent model that doesn't have a `parent_site` field.
- Replace *django-form-designer* GitHub dependency with updated django-form-designer-ai PyPI release.
- Reformat all files with isort, black and prettier.

5.2.3 Changes in 2.0.5 (2019-04-12)

- Fixed compatibility with Django 2.2

5.2.4 Changes in 2.0.4 (2018-08-27)

- Fixed showing languages in the copy button which are not part of `PARLER_LANGUAGES`.
- Fixed storing `MarkupItem` under it's proxy class type ID.
- Fixed missing return value from `CachedModelMixin.delete()`.
- Fixed reading `context.request` for content plugin templates.
- Fixed including `cp_tabs.js` in `PlaceholderFieldAdmin`.
- Fixed HTML comment output escaping for missing database tables.
- Fixed errors in `get_plugins_by_name()` when object instances are passed.
- Fixed `Placeholder.DoesNotExist` warning to display proper ContentType ID for proxy models.
- Fixed leaving empty `.form-row` elements in the admin page.
- Fixed `start_content_plugin` command, template was missing in `MANIFEST`.
- Fixed Python 3 support for `Placeholder.__repr__()`.

5.2.5 Changes in 2.0.3 (2018-05-14)

- Fixed twitter-text extra requires dependency for Python 3 support Use `twitter-text` instead of the outdated `twitter-text-py`.

5.2.6 Changes in 2.0.2 (2018-02-12)

- Fixed JavaScript media file ordering for Django 2.0

5.2.7 Changes in 2.0.1 (2018-02-05)

- Added `Meta.manager_inheritance_from_future = True` to all `ContentItem` subclasses that define a `Meta` class. This avoids warnings in the latest django-polymorphic 2.0.1 release. It also makes sure all sub-sub classes are correctly fetched (an unlikely use-case though).
- Fixed deprecation warnings for Django 2.1
- Fixed setup classifiers

5.2.8 Changes in 2.0 (2018-01-22)

- Added Django 2.0 support.
- Removed compatibility with very old `django-form-designer` versions.
- Dropped Django 1.7, 1.8, 1.9 support.

5.2.9 Changes in 1.2.2 (2017-11-22)

- Fixed compatibility with upcoming django-polymorphic release.
- Delayed twittertext plugin checks.
- Removed unneeded `fluent_utils.django_compat` imports.

5.2.10 Changes in 1.2.1 (2017-08-10)

- Fixed Django 1.10+ wanting to create new migrations.
- Fixed inconsistent ordering of "markup item" language choices.
- Fixed str/bytes differences in migrations between Python 2 and 3.

5.2.11 Changes in 1.2 (2017-05-01)

- Django 1.11 support.
- Fixd garbled placeholder data in admin forms submitted with errors
- Fixed JavaScript `django_wysiwyg` error when text plugin is not included.
- Dropped Django 1.5, 1.6 and Python 2.6 support.

5.2.12 Changes in 1.1.11 (2016-12-21)

- Fixed "Copy language" button for Django 1.10+
- Fix slot parameter for tests `fluent_contents.tests.factories.create_placeholder()`

5.2.13 Changes in 1.1.10 (2016-12-15)

- Fixed "Copy language" button for Django 1.9+

5.2.14 Changes in 1.1.9 (2016-12-06)

- Added `find_contentitem_urls` management command to index URL usage.
- Added `remove_stale_contentitems --remove-unreferenced` option to remove content items that no longer point to an existing page.
- Make sure the OEmbed plugin generates links with `https://` when `SECURE_SSL_REDIRECT` is set, or `FLUENT_OEMBED_FORCE_HTTPS` is enabled.
- Fixed loosing jQuery event bindings in the admin.

5.2.15 Changes in 1.1.8 (2016-11-09)

- Added `remove_stale_contentitems` command for cleaning unused `ContentItem` objects. This also allows the migrations to remove the stale `ContentType` models afterwards.
- Fixed `start_content_plugin` command for Django 1.7
- Fixed `MiddlewareMixin` usage for Django 1.10 middleware support
- Fixed `is_template_updated()` check for some Django 1.8 template setups.

5.2.16 Changes in 1.1.7 (2016-10-05)

- Added animations when moving content items, using the up/down buttons.
- Added drag&drop support on the title bar for reordering content items.

Although new feature additions usually mandate a new point release ("1.2'), these two improvements are too wonderful to delay further. Hence they are backported from development.

5.2.17 Changes in 1.1.6 (2016-09-11)

- Added `start_content_plugin` management command.
- Fixed running *clear_cache()* too early on a new model; it executed before saving/retrieving a primary key.
- Fixed unwanted HTML escaping for output comments that report stale models.
- Fixed Python errors during debugging when the debug toolbar panel finds stale models.
- Fixed errors by `context.flatten()` on plugin rendering (e.g. when using *django-crispy-forms*).
- Fixed `ContentPlugin.ADMIN_TEMPLATE_WITHOUT_LABELS` template when displaying multiple fields on a single line.

5.2.18 Changes in 1.1.5 (2016-08-06)

- Fixed usage of deprecated `context_instance` for Django 1.10 compatibility.
- Fixed delete dialog in the Django admin when the page has stale context items.
- Fixed compatibility with html5lib 0.99999999/1.0b9

BACKWARDS INCOMPATIBLE: the custom merging template that's used in `{% page_placeholder .. template=".." %}` no longer receives any custom context processor data defined in `context_processors / TEMPLATE_CONTEXT_PROCESSORS`. Only the standard Django context processors are included (via the `PluginContext`). The standard template values like `{{ request }}`, `{{ STATIC_URL }}` and `{% csrf_token %}` still work.

5.2.19 Changes in 1.1.4 (2016-05-16)

- Added `fluent_contents.tests.factories` methods for easier plugin testing.
- Added missing django-fluent-comments media files for `contentarea` plugin. This is configurable with the `FLUENT_COMMENTSAREA_INCLUDE_STATIC_FILES` setting, that defaults to `FLUENT_BLOGS_INCLUDE_STATIC_FILES` (`True`).
- Fixed appearance in django-flat-theme / Django 1.9.

- Fixed proxy model support for `ContentItem` models.
- Fixed Markup plugin rendering.
- Fixed reStructuredText rendering, avoid rendering the whole HTML document.

5.2.20 Changes in 1.1.3 (2016-05-11)

- Fixed `{% csrf_token %}` support in plugin templates.
- Fixed django-debug-toolbar support for skipped items.
- Fixed error handling of missing content items in the database.

5.2.21 Changes in 1.1.2 (2016-03-25)

- Fix truncating long `db_table` names, just like Django does.
- Fix various Django 1.9 warnings that would break once Django 1.10 is out.
- Enforce newer versions on dependencies to ensure all bugfixes are installed.

5.2.22 Changes in 1.1.1 (2016-01-04)

- Fixed errors when rendering pages with missing items

5.2.23 Changes in 1.1 (2015-12-29)

- Added Django 1.9 support
- Added django-debug-toolbar panel: `fluent_contents.panels.ContentPluginPanel'`.
- Added `Placeholder.get_search_text()` API for full text indexing support.
- Added `FLUENT_TEXT_POST_FILTERS` and `FLUENT_TEXT_PRE_FILTERS` to the text plugin for further processing of the text.
- **BACKWARDS INCOMPATIBLE:** as text filters became global, the settings in `fluent_contents.plugins.text.appsettings` moved to `fluent_contents.appsettings`.
- Dropped Django 1.4 support

5.2.24 Changes in 1.0.4 (2015-12-17)

- Prevent caching complete placeholder/sharedcontent output when there are items with `cache_output_per_site`. This only occurs in environments where `FLUENT_CONTENTS_CACHE_PLACEHOLDER_OUTPUT` is enabled.
- Fix Django migration unicode issues in Python 3
- Fix error in `get_output_cache_keys()` when reading the `pk` field during deletion.
- Fix compatibility with django-polymorphic 0.8.

5.2.25 Changes in 1.0.3 (2015-10-01)

- Improve styling with django-flat-theme theme.
- Fix choices listing of the "Copy Language" button.
- Fix form field order so CSS can select `.form-row:last-child`.

5.2.26 Version 1.0.2

- Added `ContentItem.move_to_placeholder()` and `ContentItem.objects.move_to_placeholder()` API functions
- Added check against bad html5lib versions that break HTML cleanup.
- Fix using `ContentItemInline.get_queryset()` in Django 1.6/1.7/1.8
- Fix Python 3.4 support for development (fixed `_is_template_updated` / "is method overwritten" check)
- Fix support for returning an `HttpRedirectRequest` in the `ContentPlugin.render()` method.
- Fix `copy_to_placeholder()` to accidently setting an empty "FK cache" entry for the `ContentItem.parent` field.
- Fix `TypeError` when abstract `ContentItem` class has no `__str__()` method.
- Fix initial migration for sharedcontent plugin.
- Fix handling of `SharedContent.__str__()` for missing translations.

5.2.27 Version 1.0.1

- Fix rendering in development for Django 1.4 and 1.5
- Fix placeholder cache timeout values, take `ContentPlugin.cache_output` into account. This is only an issue when using `FLUENT_CONTENTS_CACHE_PLACEHOLDER_OUTPUT = True`.
- Fix migration files that enforced using django-any-urlfield / django-any-imagefield. NOTE: all migrations now explicitly refer to `PluginUrlField` / `PluginImageField`. You can either generate new Django migrations, or simply replace the imports in your existing migrations.

5.2.28 Version 1.0

- Added Django 1.8 support.
- Added caching support for the complete `{% render_placeholder %}`, `{% page_placeholder %}` and `{% sharedcontent %}` tags.
- Added `as var` syntax for `{% render_placeholder %}`, `{% page_placeholder %}` and `{% sharedcontent %}` tags.
- Added `ContentItem.copy_to_placeholder()` and `ContentItem.objects.copy_to_placeholder()` API functions
- Fix handling `CheckboxSelectMultiple` in admin form widgets.
- Fix missing API parameters for `ContentItem.objects.create_for_placeholder()` and `Placeholder.objects.create_for_parent()`.
- Fix static default `SITE_ID` value for `SharedContent`, for compatibility with django-multisite.

- Fix cache invalidation when using `render_ignore_item_language`.
- Fix adding a second `PlaceholderField` to a model in a later stage.

Released on 1.0c3:

- Added Django 1.7 support.
- Added option to share `SharedContent` objects across multiple websites.
- Allow passing `SharedContent` object to `{% sharedcontent %}` template tag.
- Added `SharedContent.objects.published()` API for consistency between all apps.
- Fixed rendering content items in a different language then the object data is saved as. This can be overwritten by using `render_ignore_item_language = True` in the plugin.
- Fixed support for: future >= 0.13.
- Improve default value of `ContentPlugin.cache_timeout` for Django 1.6 support.
- Fix frontend media support for `{% sharedcontent %}` tag.
- **BACKWARDS INCOMPATIBLE:** South 1.0 is required to run the migrations (or set `SOUTH_MIGRATION_MODULES` for all plugins).
- **BACKWARDS INCOMPATIBLE:** Content is rendered in the language that is is being saved as, unless `render_ignore_item_language` is set.

Note: Currently, Django 1.7 doesn't properly detect the generated `db_table` value properly for ContentItem objects. This needs to be added manually in the migration files.

Released on 1.0c2:

- Fix JavaScript errors with `for i in` when `Array.prototype` is extended. (e.g. when using django-taggit-autosuggest).

Released on 1.0c1:

- Fix saving content item sorting.

Released on 1.0b2:

- Added Python 3 support!
- Fixed Django 1.6 compatibility.
- Fixed disappearing contentitems issue for PlaceholderField on add-page
- Fixed orphaned content for form errors in the add page.
- Fixed no tabs selected on page reload.

Released on 1.0b1:

- Added multilingual support, using django-parler.
- Added multisite support to sharedcontent plugin.
- Added frontend media support.
- Added "Open in new window" option for the "picture" plugin.
- Added `HttpRedirectRequest` exception and `HttpRedirectRequestMiddleware`.
- Added `cache_output_per_language` option to plugins.
- Content items are prefixed with "content:" during syncdb, a `prefix_content_item_types` management command can be run manually too.
- **API Change:** Renamed template tag library `placeholder_tags` to `fluent_contents_tags` (the old name still works).
- **API Change:** `render_placeholder()` and `render_content_items()` return a `ContentItemOutput` object, which can be treated like a string.
- **API Change:** both `get_output_cache_key()` and `get_output_cache_keys()` should use `get_output_cache_base_key()` now.
- Fix showing non-field-errors for inlines.
- Fix server error on using an invalid OEmbed URL.
- Fix gist plugin, allow UUID's now.
- Fix missing `alters_data` annotations on model methods.
- Removed unneeded `render_comment_list` templatetag as it was upstreamed to django-threadedcomments 0.9.

5.2.29 Version 0.9

- Dropped Django 1.3 support, added Django 1.6 support.
- Added `FLUENT_CONTENTS_PLACEHOLDER_CONFIG` variable to limit plugins in specific placeholder slots.
- Added model fields for plugin developers, to have a consistent interface. The model fields integrate with django-any-urlfield, django-any-imagefield and django-wysiwyg.
- Added picture plugin.
- Added development (`DEBUG=True`) feature, changes in plugin templates update the stored version in the output cache.
- Added cache methods to plugins which can be overwritten (`get_output_cache_key()`, `get_cached_output()`, etc..)
- Added `cache_output_per_site` option to plugins.
- Fix admin appearance of plugins without fields.
- Fix initial south migrations, added missing dependencies.

5.2.30 Version 0.8.6

- Fixed metaclass errors in markup plugin for Django 1.5 / six.
- Fix initial south migrations, added missing dependencies.
- Fixed cache clearing of sharedcontent plugin.
- Updated django-polymorphic version to 0.4.2, addressed deprecation warnings.
- Updated example app to show latest features.

5.2.31 Version 0.8.5

- Added support for shared content.
- Added `ContentPlugin.HORIZONTAL` and `ContentPlugin.VERTICAL` constants for convenience.
- Added support for noembed in `FLUENT_OEMBED_SOURCE` setting.
- Added `FLUENT_OEMBED_EXTRA_PROVIDERS` setting to the OEmbed plugin.
- Fix Django 1.5 compatibility.
- Fix *code* plugin compatibility with Pygments 1.6rc1.
- Fix escaping slot name in templates
- Fix https support for OEmbed plugin.
- Fix maxwidth parameter for OEmbed plugin.
- Fix updating OEmbed code after changing maxwidth/maxheight parameters.
- Moved the template tag parsing to a separate package, django-tag-parser.
- Bump version of django-wysiwyg to 0.5.1 because it fixes TinyMCE integration.
- Bump version of micawber to 0.2.6, which contains an up to date list of known OEmbed providers.
- **BIC:** As micawber is actively updated, we no longer maintain a local list of known OEmbed providers. This only affects installations where `FLUENT_OEMBED_SOURCE = "list"` was explicitly defined in `settings.py`, without providing a list for `FLUENT_OEMBED_PROVIDER_LIST`. The new defaults are: `FLUENT_OEMBED_SOURCE = "basic"` and `FLUENT_OEMBED_PROVIDER_LIST = ()`.

5.2.32 Version 0.8.4

- Fix 500 error when content items get orphaned after switching layouts.
- Fix plugin dependencies installation via the optional dependency specifier (e.g. `django-fluent-contents[text]`).
- Fix missing dependency check for OEmbed plugin
- Fix Django dependency in `setup.py`, moved from `install_requires` to the `requires` section.
- Fix template name for django-threadedcomments to `comment/list.html`, to be compatible with the pull request at https://github.com/HonzaKral/django-threadedcomments/pull/39.

5.2.33 Version 0.8.3

- Fixed `fluent_contents.rendering.render_content_items()` to handle models without a PK.
- Make sure the client-side `sort_order` is always consistent, so external JS code can read/submit it.

5.2.34 Version 0.8.2

- Fixed `PlaceholderField` usage with inherited models.

5.2.35 Version 0.8.1

- Fixed missing files for oembed and markup plugins.
- Clarified documentation bits

5.2.36 Version 0.8.0

First PyPI release.

The module design has been stable for quite some time, so it's time to show this module to the public.

CHAPTER 6

Roadmap

The following features are on the radar for future releases:

- Frontend editing support
- Bridging other plugin systems, like Django CMS
- Inline support (e.g. building a linklist plugin).

Please contribute your improvements or work on these area's!

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

Symbols

A

B

C

D

E

F

G

H

I

M

N

P

R

S

T

V