
django-fluent-pages Documentation

Release 2.0.7

Diederik van der Boor

Oct 24, 2021

Contents

1	Preview	3
2	Getting started	5
2.1	Quick start guide	5
2.2	Configuration	10
2.3	The template tags	12
2.4	Sitemaps integration	15
2.5	Multilingual support	16
2.6	Management Commands	18
3	Using the page type plugins	21
3.1	Bundled Page Type Plugins	21
3.2	Creating new page types	32
4	API documentation	43
4.1	Low-level API's	43
4.2	API documentation	45
4.3	Package dependencies	65
4.4	Changelog	66
5	Indices and tables	77
	Python Module Index	79
	Index	81

This module provides a page tree, where each node type can be a different model. This allows you to structure your site CMS tree as you see fit. For example:

- Build a tree of flat pages, with a WYSIWYG editor.
- Build a tree with widget-based pages, by integrating [django-fluent-contents](#).
- Build a tree structure of RST pages, by defining a `RstPage` type.
- Build a tree of a *homepage*, *subsection*, and *article* node, each with custom fields like professional CMSes have.

Each node type can have it's own custom fields, attributes, URL patterns and rendering.

In case you're building a custom CMS, this module might just be suited for you, since it provides the tree for you, without bothering with anything else. The actual page contents is defined via page type plugins. 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

Add Page

Title:	<input type="text"/>
Slug:	<input type="text"/> <small>The slug is used in the URL of the page</small>
Status:	<input type="radio"/> Published <input checked="" type="radio"/> Draft
Layout:	Standard-2col <input style="font-size: 0.8em; vertical-align: middle;" type="button" value="+"/>
<input checked="" type="checkbox"/> Show in navigation	

Main Sidebar

Text item ↕ ↔

B *I* U ABC | | | | Paragraph

| x_2 x^2 |

Path: p

Comments area ↕ ↔

Allow posting new comments

2.1 Quick start guide

2.1.1 Installing django-fluent-pages

Make sure you have the base packages installed:

```
pip install Django
pip install django-fluent-pages
```

This command installs the base dependencies. As you add more of the *Bundled Page Type Plugins*, additional packages may be required. This is explained in the documentation for each plugin.

Tip: For optional dependency management, it is strongly recommended that you run the application inside a *virtualenv*.

2.1.2 Starting a project

For a quick way to have everything configured at once, use our template:

```
mkdir example.com
django-admin.py startproject "myexample" "example.com" -e "py,rst,example,gitignore" -
↳-template="https://github.com/edoburu/django-project-template/archive/django-fluent.
↳zip"
```

And install it's packages:

```
mkvirtualenv example.com
pip install -r example.com/requirements.txt
```

Otherwise, continue with the instructions below:

2.1.3 Basic Settings

In your existing Django project, the following settings need to be added to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_pages',
    'mptt',
    'parler',
    'polymorphic',
    'polymorphic_tree',

    # And optionally add the desired page types with their dependencies:

    # - flat pages
    'fluent_pages.pagetypes.flatpage',
    'django_wysiwyg',

    # - redirect nodes
    'fluent_pages.pagetypes.redirectnode',
    'any_urlfield', # optional but recommended
)
```

The following applications are used here:

- The main `fluent_pages` package that you always need.
- The main dependencies.
- A selection of page type plugins, and their dependencies.

Since some extra page types are used here, make sure their dependencies are installed:

```
pip install django-fluent-pages[flatpage,redirectnode]
```

Afterwards, you can setup the database:

```
./manage.py migrate # use 'syncdb' for Django 1.6 and below
```

Note: Each page type is optional. Only the `fluent_pages` application is required, allowing to write custom models and plugins. Since a layout with the `flatpage` and `redirectnode` page types provides a good introduction, these are added here.

Each plugin is easily swappable for other implementations, exactly because everything is optional! You can use a different page type, or invert new page types with custom fields. It makes the CMS configurable in the way that you see fit.

2.1.4 URL configuration

The following needs to be added to `urls.py`:

```
urlpatterns += patterns('',
    url(r'', include('fluent_pages.urls'))
)
```

See also:

- To add sitemaps support, see the *Sitemaps integration* documentation about that.

- Multilingual support may also require changes, see *Multilingual support*.

2.1.5 Template structure

The page output is handled by templates. When creating large websites, you'll typically have multiple page templates. That's why it's recommended to have a single base template for all pages. This can expose the SEO fields that are part of every HTML page. As starting point, the following structure is recommended:

```
templates/
  base.html
  pages/
    base.html
    default.html
    ...
```

Now, create a `pages/base.html` template:

```
{% extends "base.html" %}

{% block full-title %}{% if page.meta_title %}{{ page.meta_title }}{% else %}{{ block.super }}{% endif %}{% endblock %}
{% block meta-keywords %}{{ page.meta_keywords }}{% endblock %}
{% block meta-description %}{{ page.meta_description }}{% endblock %}

{% block extrahead %}{{ block.super }}{% if page.meta_robots %}
  <meta name="robots" content="{{ page.meta_robots }}" />
{% endif %}{% endblock %}
```

These blocks should appear in your `base.html` template off course.

Your site `base.html` template could look something like this:

```
{% load fluent_pages_tags %}
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="keywords" content="{% block meta-keywords %}{% endblock %}" />
  <meta name="description" content="{% block meta-description %}{% endblock %}" />
  <title>{% block full-head-title %}{% block head-title %}Untitled{% endblock %} | My_
  ↳site{% endblock %}</title>
  {% block extrahead %}{% endblock %}
</head>
<body>
  <header>
    {% render_menu %}
  </header>

  <section id="contents">
    <div id="main">
      <h1>{{ page.title }}</h1>

      {% render_breadcrumb %}

      {% block main %}{% endblock %}
    </div>
  </section>
```

(continues on next page)

(continued from previous page)

```
</body>
</html>
```

This base template does the following:

- Expose the placeholders for SEO fields.
- Add a main menu using `{% render_menu %}`
- Add a breadcrumb using `{% render_breadcrumb %}`

Tip: Whether `page.title` should be included is your own decision. You can also let clients enter the `<h1>` in the WYSIWYG page content, and reserve `page.title` for menu titles alone. This works really well in practise.

2.1.6 Adding page content

This package is very flexible when it comes to choosing page content. There are several page type plugins available:

- `fluent_pages.pagetypes.flatpage` - a simple page with WYSIWYG text box.
- `fluent_pages.pagetypes.fluentpage` - a page with flexible content blocks.
- *Other known page types*, such as a FAQ index or Blog index page.

The tree can also contain other node types, e.g.:

- `fluent_pages.pagetypes.redirectnode` - a redirect.
- `fluent_pages.pagetypes.text` - a plain text file, e.g. to add a `humans.txt` file.
- or any *custom page type* you create.

See also:

In this quick-start manual, we'll discuss the most important options briefly below. See the *Bundled Page Type Plugins* for the full documentation about each page type.

Using the flatpage plugin

The *Flat page* page type displays a simple WYSIWYG text box. To use it, install the packages and desired plugins:

```
pip install django-fluent-pages[flatpage]
pip install django-tinymce
```

Tip: You can also use CKEditor, Redactor or an other WYSIWYG editor, but for convenience TinyMCE is used as example. See the documentation of the *The flatpage page type* for details.

Add the following settings:

```
INSTALLED_APPS += (
    'fluent_pages.pagetypes.flatpage',
    'django_wysiwyg',
    'tinymce',
)
```

(continues on next page)

(continued from previous page)

```
DJANGO_WYSIWYG_FLAVOR = "tinymce"    # or "tinymce_advanced"
FLUENT_TEXT_CLEAN_HTML = True
FLUENT_TEXT_SANITIZE_HTML = True
```

Make sure the database tables are created:

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

To render the output properly, create a `fluent_pages/base.html` file so the *Flat page* pages can map the block names to the ones you use in `base.html`:

```
{% extends "pages/base.html" %}

{% block head-title %}{% block title %}{% endblock %}{% endblock %}

{% block main %}{% block content %}{% endblock %}{% endblock %}
```

Using the fluentpage plugin

The *Fluent page* page type can fill parts of the page with flexible content blocks. To use it, install the packages and desired plugins:

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

Configure the settings:

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

    # Page types
    'fluent_pages.pagetypes.fluentpage',
    'fluent_pages.pagetypes.flatpage',
    'fluent_pages.pagetypes.redirectnode',

    # Several content plugins
    'fluent_contents.plugins.text',           # requires django-wysiwyg
    'fluent_contents.plugins.code',         # requires pygments
    'fluent_contents.plugins.gist',
    'fluent_contents.plugins.google-docs-viewer',
    'fluent_contents.plugins.iframe',
    'fluent_contents.plugins.markup',
    'fluent_contents.plugins.rawhtml',
)

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

Make sure the database tables are created:

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

The template can be filled with the “placeholder” tags from `django-fluent-contents`:

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

{% block main %}
    <section id="main">
        <article>
            {% block pagetitle %}<h1 class="pagetitle">{{ page.title }}</h1>{%
↵endblock %}
            {% page_placeholder "main" role='m' %}
        </article>

        <aside>
            {% page_placeholder "sidebar" role='s' %}
        </aside>
    </section>
{% endblock %}
```

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.

2.2 Configuration

A quick overview of the available settings:

```
FLUENT_PAGES_BASE_TEMPLATE = "fluent_pages/base.html"

FLUENT_PAGES_TEMPLATE_DIR = TEMPLATE_DIRS[0]

FLUENT_PAGES_RELATIVE_TEMPLATE_DIR = True

FLUENT_PAGES_DEFAULT_IN_NAVIGATION = True

FLUENT_PAGES_KEY_CHOICES = ()

# Advanced
FLUENT_PAGES_PREFETCH_TRANSLATIONS = False
FLUENT_PAGES_FILTER_SITE_ID = True
FLUENT_PAGES_PARENT_ADMIN_MIXIN = None
FLUENT_PAGES_CHILD_ADMIN_MIXIN = None
ROBOTS_TXT_DISALLOW_ALL = DEBUG
```

2.2.1 Template locations

FLUENT_PAGES_BASE_TEMPLATE

The name of the base template. This setting can be overwritten to point all templates to another base template. This can be used for the *Flat page* page type.

FLUENT_PAGES_TEMPLATE_DIR

The template directory where the “Layouts” model can find templates. By default, this is the first path in `TEMPLATE_DIRS`. It can also be set explicitly, for example:

```
FLUENT_PAGES_TEMPLATE_DIR = os.path.join(SRC_DIR, 'frontend', 'templates')
```

FLUENT_PAGES_RELATIVE_TEMPLATE_DIR

Whether template paths are stored as absolute or relative paths. This defaults to relative paths:

```
FLUENT_PAGES_RELATIVE_TEMPLATE_DIR = True
```

2.2.2 Preferences for the admin

FLUENT_PAGES_DEFAULT_IN_NAVIGATION

This defines whether new pages have the “Show in Navigation” checkbox enabled by default. It makes sense for small sites to enable it, and for larger sites to disable it:

```
FLUENT_PAGES_DEFAULT_IN_NAVIGATION = False
```

FLUENT_PAGES_KEY_CHOICES

Pages can be “tagged” to be easily found in the page tree. Example value:

```
FLUENT_PAGES_KEY_CHOICES = (
    # Allow to tag some pages, so they can be easily found by templates.
    ('search', _("Search")),
    ('contact', _("Contact")),
    ('terms', _("Terms and Conditions")),
    ('faq', _("FAQ page")),
    ('impactmap', _("Impact map")),
)
```

When this value is defined, a “Page identifier” option appears in the “Publication settings” fieldset.

Pages which are marked with an identifier can be found using `Page.objects.get_for_key()`.

2.2.3 Performance optimizations

FLUENT_PAGES_PREFETCH_TRANSLATIONS

Enable this to prefetch all translations at a regular page. This is useful to display a language choice menu:

```
FLUENT_PAGES_PREFETCH_TRANSLATIONS = True
```

2.2.4 SEO settings

ROBOTS_TXT_DISALLOW_ALL

When using *RobotsTxtView*, enable this setting for beta websites. This makes sure such site won't be indexed by search engines. Off course, it's recommended to add HTTP authentication to such site, to prevent accessing the site at all.

2.2.5 Advanced admin settings

FLUENT_PAGES_FILTER_SITE_ID

By default, each `Site` model has it's own page tree. 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
```

FLUENT_PAGES_PARENT_ADMIN_MIXIN / FLUENT_PAGES_CHILD_ADMIN_MIXIN

By setting this value, this module will insert your class in the admin. This can be used to override methods, or provide integration other third party applications such as *django-guardian*.

- The “parent admin” handles the list display for pages.
- The “child admin” handles the edit and delete views for pages.

Example setting:

```
FLUENT_PAGES_PARENT_ADMIN_MIXIN = 'apps.auth_utils.page_admin.  
↳FluentPagesParentAdminMixin'  
FLUENT_PAGES_CHILD_ADMIN_MIXIN = 'apps.auth_utils.page_admin.  
↳FluentPagesChildAdminMixin'
```

Your project needs to provide those classes, and can implement or override admin methods there.

2.2.6 Advanced language settings

The language settings are copied by default from the *django-parler* variables. If you have to provide special settings (basically fork the settings), you can provide the following values:

```
FLUENT_DEFAULT_LANGUAGE_CODE = PARLER_DEFAULT_LANGUAGE_CODE = LANGUAGE_CODE  
  
FLUENT_PAGES_DEFAULT_LANGUAGE_CODE = FLUENT_DEFAULT_LANGUAGE_CODE  
FLUENT_PAGES_LANGUAGES = PARLER_LANGUAGES
```

2.3 The template tags

The template tags provide a way to include a menu, or breadcrumb in the website. Load the tags using:


```
{% load fluent_pages_tags %}
```

2.3.1 The breadcrumb

The breadcrumb of the current page can be rendered using:

```
{% render_breadcrumb %}
```

It's possible to render the breadcrumb using a custom template:

```
{% render_breadcrumb template="fluent_pages/parts/breadcrumb.html" %}
```

The breadcrumb template could look like:

```
{% if breadcrumb %}
<ul>
{% for item in breadcrumb %}
  <li{% if forloop.last %} class="last"{% endif %}><a href="{{ item.url }}">{{ item.
  ↳title }}</a></li>
{% endfor %}
</ul>
{% endif %}
```

2.3.2 The menu

The menu of the site can be rendered using:

```
{% render_menu %}
```

The number of levels can be limited using the `depth` parameter:

```
{% render_menu depth=1 %}
```

Custom menu template

The template parameter offers a way to define your own menu layout. For example:

```
{% render_menu max_depth=1 template="fluent_pages/parts/menu.html" %}
```

The menu template could look like:

```
{% load mptt_tags %}
{% if menu_items %}
  <ul>
    {% recursetree menu_items %}
    <li class="{% if node.is_active or node.is_child_active %}active{% endif %}{% if_
    ↳node.is_draft %} draft{% endif %}">
      <a href="{{ node.url }}">{{ node.title }}</a>
      {% if children %}<ul>{{ children }}</ul>{% endif %}
    </li>{% endrecursetree %}
  </ul>
{% else %}
```

(continues on next page)

(continued from previous page)

```
<!-- Menu is empty -->
{% endif %}
```

The node variable is exposed by the `{% recursetree %}` tag. It's a `PageNavigationNode` object.

To use a different template, either override the `fluent_pages/parts/menu.html` template in your project, or use the `template` variable (recommended). For example, for a Bootstrap 3 project, you can use the following template:

```
{% load mptt_tags %}
{% if menu_items %}
<ul class="nav navbar-nav">
  {% recursetree menu_items %}
  <li class="{% if node.is_active or node.is_child_active %}active{% endif %}">
    {% if children %}
      <a href="{{ node.url }}" class="dropdown-toggle" data-toggle="dropdown" role=
      ↪ "button" aria-haspopup="true" aria-expanded="false">{{ node.title }} <span class=
      ↪ "caret"></span></a>
      <ul class="dropdown-menu" role="menu">{{ children }}</ul>
    {% else %}
      <a href="{{ node.url }}">{{ node.title }}</a>
    {% endif %}
  </li>{% endrecursetree %}
</ul>
{% else %}
  <!-- Menu is empty -->
{% endif %}
```

Rendering side menu's

You can render a subsection of the menu using the `parent` keyword argument. It expects a page object, URL path or page ID of the page you want to start at. Combined with the `template` argument, this gives

```
{% render_menu parent=page max_depth=1 template="partials/side_menu.html" %}
{% render_menu parent='/documentation/' max_depth=1 %}
{% render_menu parent=8 max_depth=1 %}
```

2.3.3 Advanced features

Fetching 'site' and 'page' variables

The templates receive a `site` and `page` variable by default. In case the template is rendered outside the regular loop, these fields can be fetched:

```
{% get_fluent_page_vars %}
```

Locating custom page type views

When a custom page type provides additional views, these can be fetched using:

```
{% load appurl_tags %}

{% appurl "my_viewname" %}

{% appurl "my_viewname" arg1 arg2 %}

{% appurl "my_viewname" kwarg1=value kwarg2=value %}
```

These tags locate the page in the page tree, and resolve the view URL from there.

2.4 Sitemaps integration

The pages can be included in the sitemap that `django.contrib.sitemaps` provides. This makes it easier for search engines to index all pages.

Add the following in `urls.py`:

```
from fluent_pages.sitemaps import PageSitemap
from fluent_pages.views import RobotsTxtView

sitemaps = {
    'pages': PageSitemap,
}

urlpatterns += patterns('',
    url(r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': _
↪sitemaps}),
    url(r'^robots.txt$', RobotsTxtView.as_view()),
)
```

The `django.contrib.sitemaps` should be included in the `INSTALLED_APPS` off course:

```
INSTALLED_APPS += (
    'django.contrib.sitemaps',
)
```

The pages should now be visible in the `sitemap.xml`.

A sitemap is referenced in the `robots.txt` URL. When using the bundled `RobotsTxtView` in the example above, this happens by default.

The contents of the `robots.txt` URL can be overwritten by overriding the `robots.txt` template. Note that the `robots.txt` file should point to the sitemap with the full domain name included:

```
Sitemap: http://full-website-domain/sitemap.xml
```

For more details about the `robots.txt` URL, see the documentation at <http://www.robotstxt.org/> and <https://support.google.com/webmasters/answer/6062608?hl=en&rd=1>

Note: When using Nginx, verify that `robots.txt` is also forwarded to your Django application.

For example, when using `location = /robots.txt { access_log off; log_not_found off; }`, the request will not be forwarded to Django because this replaces the standard `location / { .. }` block.

2.5 Multilingual support

This package supports creating content in multiple languages. This feature is based on `django-parler`. Historical anecdote: `django-parler` was created to make this CMS multilingual.

To enable multiple languages, configuring `django-parler` is sufficient.

2.5.1 Configuration

```
LANGUAGES = (
    ('en', _("Global Site")),
    ('en-us', _("US Site")),
    ('it', _("Italian")),
    ('nl', _("Dutch")),
    ('fr', _("French")),
    ('es', _("Spanish")),
)

PARLER_DEFAULT_LANGUAGE_CODE = 'en' # defaults to LANGUAGE_CODE

SITE_ID = None

PARLER_LANGUAGES = {
    None: (
        # Default SITE_ID, all languages
        {'code': lang[0]} for lang in LANGUAGES
    ),
    2: (
        # SITE_ID 2: only english/french
        {'code': 'en',},
        {'code': 'fr',},
    ),
    'default': {
        # This is applied to each entry in this setting:
        'hide_untranslated': False,
        'hide_untranslated_menu_items': False,
        # 'fallback': 'en' # set by PARLER_DEFAULT_LANGUAGE_CODE
    }
}
```

There are two extra values that can be used:

- `hide_untranslated`: if set to `True`, untranslated pages are not accessible.
- `hide_untranslated_menu_items`: if set to `True`, untranslated pages are not visible in the menu.

These values can be used in the “default” section, or in each dictionary entry per site.

2.5.2 Accessing content

There are several ways to expose translated content. One way is adding a subpath in the URL by using `i18n_patterns()`:

Using i18n_patterns

Add the following to `settings.py`:

```
MIDDLEWARE_CLASSES += (
    'django.middleware.locale.LocaleMiddleware', # or your own override/replacement
)
```

Add to `urls.py`:

```
from django.conf.urls import patterns, url
from django.conf.urls.i18n import i18n_patterns
from django.contrib import admin
from fluent_pages.sitemaps import PageSitemap

sitemaps = {
    # Place sitemaps here
    'pages': PageSitemap,
}

admin.autodiscover()

urlpatterns = patterns('',
    # All URLs that should not be prefixed with the country code,
    # e.g. robots.txt or django admin.
) + i18n_patterns('',
    # All URLs inside the i18n_patterns() get prefixed with the country code:
    # Django admin
    url(r'^admin/', admin.site.urls),

    # SEO API's per language
    url(r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': _
↪sitemaps}),

    # CMS modules
    url(r'', include('fluent_pages.urls')),
)
```

Using custom middleware

Nothing prevents you from writing custom middleware that sets the frontend language. For example:

Add the following to `settings.py`:

```
LANGUAGE_CODE = 'en' # default, e.g. for the admin
FRONTEND_LANGUAGE_CODE = 'de'

MIDDLEWARE_CLASSES += (
    'mysite.middleware.FrontendLanguageMiddleware',
)
```

The custom middleware code:

```
from django.conf import settings
from django.utils import translation
from django.urls import reverse_lazy
```

(continues on next page)

```

class FrontendLanguageMiddleware(object):
    """
    Change the active language when visiting a frontend page.
    """
    def __init__(self):
        # NOTE: not locale aware, assuming the admin stays at a single URL.
        self._admin_prefix = reverse_lazy('admin:index', prefix='/')

    def process_request(self, request):
        if request.path_info.startswith(str(self._admin_prefix)):
            return # Excluding the admin

        if settings.FRONTEND_LANGUAGE_CODE != settings.LANGUAGE_CODE:
            translation.activate(settings.FRONTEND_LANGUAGE_CODE)

```

This could even include detecting the sub-domain, and setting the language accordingly.

All queries that run afterwards read the active language setting, and display the content in the given language.

You can take this further and make Django aware of the sub-domain in its URLs by overriding `ABSOLUTE_URL_OVERRIDES` in the settings. The `Page` provides a `default_url` attribute for this specific use-case. You'll also have to override the sitemap, as it won't take absolute URLs into account.

2.6 Management Commands

The following management commands are provided for administrative utilities:

2.6.1 make_language_redirects

When a language is unmaintained at the site, use this command to generate the URL redirects. The command outputs a script for the web server (currently only in Nginx format).

Options:

- `--from=language`: the old language
- `--to=language`: the new language
- `--format=nginx`: the format
- `--site=id`: the site for which redirects are created.

Example:

```
python manage.py make_language_redirects --from=it --to=en --format=nginx --site=1
```

2.6.2 rebuild_page_tree

In the unlikely event that the page tree is broken, this utility repairs the tree. This happened in earlier releases (before 1.0) when entire trees were moved in multi-lingual sites.

It regenerates the MPTT fields and URLs.

Options:

- `-p / --dry-run`: tell what would happen, but don't make any changes.
- `-m / --mptt-only`: only regenerate the MPTT fields, not the URLs of the tree.

Example:

```
python manage.py rebuild_page_tree
```

2.6.3 remove_stale_pages

New in version 1.1.2.

In the unlikely event that a page type was removed, but its page nodes still exist, this command can be used to repair the tree. It removes the old pages that point to content types that no longer exist.

Options:

- `-p / --dry-run`: tell what would happen, but don't make any changes.

Example:

```
python manage.py remove_stale_pages --dry-run
```


3.1 Bundled Page Type 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 The flatpage page type

The *flatpage* provides a simple page type with a WYSIWYG (“What You See is What You Get”) editor.

Add Flat Page

Title:	<input type="text"/>
Slug:	<input type="text"/> The slug is used in the URL of the page
Status:	<input type="radio"/> Published <input checked="" type="radio"/> Draft
<input checked="" type="checkbox"/> Show in navigation	
Contents	
Content:	<div><p>B <i>I</i> <u>U</u> ABC Paragraph HTML</p><p> x₂ x² —</p></div> <p>Path: p</p>
SEO settings (Show)	
Menu structure (Show)	
Publication settings (Show)	
<input type="button" value="Save and add another"/> <input type="button" value="Save and continue editing"/>	

The WYSIWYG editor is provided by [django-wysiwyg](#), making it possible to switch to any WYSIWYG editor of your choice.

Note: This page type may seem a bit too simply for your needs. However, in case additional fields are needed, feel free to create a different page type yourself. This page type can serve as canonical example.

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-pages[flatpage]
```

This installs the [django-wysiwyg](#) package.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_pages.pagetypes.flatpage',
    'django_wysiwyg',
)
```

Using CKEditor

To use [CKEditor](#), install `django-ckeditor`:

```
pip install django-ckeditor
```

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'ckeditor',
)

DJANGO_WYSIWYG_FLAVOR = "ckeditor"
```

Using TinyMCE

To use [TinyMCE](#), install `django-tinymce`:

```
pip install django-tinymce
```

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'tinymce',
)

DJANGO_WYSIWYG_FLAVOR = "tinymce"    # or "tinymce_advanced"
```

Using Redactor

To use [Redactor](#), tell `django-wysiwyg` where to find the static files. This is done on purpose to respect the commercial license.

```
DJANGO_WYSIWYG_FLAVOR = "redactor"
DJANGO_WYSIWYG_MEDIA_URL = "/static/vendor/imperavi/redactor/"
```

Template layout

To integrate the output of the page into your website design, overwrite `fluent_pages/base.html`. The following blocks have to be mapped to your website theme base template:

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

- **meta-keywords** - the value for the meta-keywords tag.

In case your website base template uses different names for those blocks, create a `fluent_pages/base.html` file to map the names:

```
{% extends "pages/base.html" %}

{% block head-title %}{% block title %}{% endblock %}{% endblock %}

{% block main %}{% block content %}{% endblock %}{% endblock %}
```

Further output tuning

The name of the base template can also be changed using the `FLUENT_PAGES_BASE_TEMPLATE` setting. The page type itself is rendered using `fluent_pages/pagetypes/flatpage/default.html`, which extends the `fluent_pages/base.html` template.

Configuration settings

The following settings are available:

```
DJANGO_WYSIWYG_FLAVOR = "yui_advanced"

FLUENT_TEXT_CLEAN_HTML = True
FLUENT_TEXT_SANITIZE_HTML = True
```

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`.

FLUENT_TEXT_CLEAN_HTML

If `True`, the HTML tags will be rewritten to be well-formed. This happens using either one of the following packages:

- `html5lib`

- `pytidylib`

FLUENT_TEXT_SANITIZE_HTML

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

3.1.2 The `fluentpage` page type



The *fluentpage* provides a page type where parts of the page can be filled with flexible content blocks.

Add Page

Title:

Slug:
The slug is used in the URL of the page









Status: Published Draft







Layout:  

Show in navigation

Main **Sidebar**

Text item


B **I** **U** **ABC** |   |   |   | Paragraph   HTML

  | x_2 x^2 |    

Path: p

Comments area

Allow posting new comments

Comments area 

SEO settings (Show)

Menu structure (Show)

Publication settings (Show)

This feature is provided by `django-fluent-contents`.

The combination of `django-fluent-pages` and `django-fluent-contents` provides the most flexible page layout. It's possible to use a mix of standard plugins (e.g. `text`, `code`, `forms`) and customly defined plugins to facilitate complex website designs. See the documentation of `django-fluent-contents` for more details.

Installation

Install the dependencies via *pip*:

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

This installs the `django-fluent-contents` package.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_pages.pagetypes.fluentpage',
    'fluent_contents',

    # The desired plugins for django-fluent-contents, e.g:
    'fluent_contents.plugins.text',
    'django_wysiwyg',
)
```

Template design

To render the page, include the tags that `django-fluent-contents` uses to define placeholders. For example:

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

{% block main %}
    <section id="main">
        <article>
            {% block pagetitle %}<h1 class="pagetitle">{{ page.title }}</h1>{%_
→endblock %}
            {% page_placeholder "main" role='m' %}
        </article>

        <aside>
            {% page_placeholder "sidebar" role='s' %}
        </aside>
    </section>
{% endblock %}
```

These placeholders will be detected and displayed in the admin pages.

Place the template in the template folder that `FLUENT_PAGES_TEMPLATE_DIR` points to. By default, that is the first path in `TEMPLATE_DIRS`.

Configuration

The page type itself doesn't provide any configuration options, everything can be fully configured by configuring `django-fluent-contents`. See the documentation of each of these [bundled content plugins](#) to use them:

- The code plugin
- The commentsarea plugin
- The Disquscommentsarea plugin
- The formdesignerlink plugin

- The gist plugin
- The googledocsviewer plugin
- The iframe plugin
- The markup plugin
- The oembeditem plugin
- The rawhtml plugin
- The sharedcontent plugin
- The text plugin
- The twitterfeed plugin

Creating new plugins

A website with custom design elements can be easily editable by creating custom plugins.

Creating new plugins is not complicated at all, and simple plugins can easily be created within 15 minutes.

The documentation of [django-fluent-contents](#) explains [how to create new plugins](#) in depth.

Advanced features

This module also provides the `FluentPageBase` and `FluentPageAdmin` classes, which can be used as base classes for *custom page types* that also use the same layout mechanisms.

3.1.3 The redirectnode page type

The *redirectnode* allows adding a URL path that redirects the website visitor.

Add Redirect

Title:	<input type="text"/>
Slug:	<input type="text"/> The slug is used in the URL of the page
Status:	<input type="radio"/> Published <input checked="" type="radio"/> Draft
	<input checked="" type="checkbox"/> Show in navigation
Contents	
New URL:	<input checked="" type="radio"/> External URL <input type="radio"/> Page <input type="text"/>
Redirect type:	<input checked="" type="radio"/> Normal redirect <input type="radio"/> Permanent redirect (for SEO ranking) Use 'normal redirect' unless you want to transfer SEO ranking to the new page.
Menu structure (Show)	
Publication settings (Show)	
<input type="button" value="Save and add another"/> <input type="button" value="Save and continue edit"/>	

Installation

Install the dependencies via *pip*:

```
pip install django-fluent-pages[redirectnode]
```

This installs the `django-any-urlfield` package.

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_pages.pagetypes.redirectnode',
    'any_urlfield',
)
```

Configuration

This page type works out of the box.

By default, the admin can choose between an “External URL” and “Page”. Other models can also be included too, as long as they have a `get_absolute_url` method. Register the respective models to `django-any-urlfield`:

```
from any_urlfield.models import AnyUrlField
AnyUrlField.register_model(Article)
```

See the `any_urlfield.models` documentation for details.

3.1.4 The textfile page type

The *textfile* allows adding a URL node that displays plain text.

Add Plain text file

Title:	<input type="text"/>
Slug:	<input type="text"/> <small>The slug is used in the URL of the page</small>
Status:	<input type="radio"/> Published <input checked="" type="radio"/> Draft
	<input checked="" type="checkbox"/> Show in navigation
Contents	
File contents:	<div style="border: 1px solid #ccc; height: 150px;"></div>
File type:	<input type="text" value="Plain text"/>
Menu structure (Show)	
Publication settings (Show)	
<input type="button" value="Save and add another"/> <input type="button" value="Save and continue edit"/>	

This page type serves as simple demo, and can also be used to add a custom `robots.txt`, `humans.txt` file or `README` file to the page tree.

Note: Currently, it's still required to use the "Override URL" field in the form to include a file extension, as the "Slug" field does not allow this.

Installation

Add the following settings to `settings.py`:

```
INSTALLED_APPS += (
    'fluent_pages.pagetypes.textfile',
)
```

3.1.5 Other known page types

Blog page

The `django-fluent-blogs` module provides a “Blog page” type, which can be used to include a “Blog” in the page tree.

To integrate it with this module, configure it using:

```
INSTALLED_APPS += (
    'fluent_blogs',
    'fluent_blogs.pagetypes.blogpage',
)
```

See the documentation of `django-fluent-blogs` for details.

FAQ page

The `django-fluent-faq` module provides a “FAQ page” type, which displays a list of FAQ questions and categories.

To integrate it with this module, configure it using:

```
INSTALLED_APPS += (
    'fluent_faq',
    'fluent_faq.pagetypes.faqpage',
)
```

See the documentation of `django-fluent-faq` for details.

Open ideas

Other page types can also be written, for example:

- a “Portfolio” page type.
- a “Split test” page type.
- a “Flat page” with `reStructuredText` content.
- a “Web shop” page type.
- a “Subsite section” page type.

See the next chapter, *Creating new page types* to create such plugins.

3.2 Creating new page types

This module is specifically designed to easily add custom page types.

Typically, a project consists of some standard modules, and perhaps one or two custom types. Creating these is easy, as shown in the following sections:

3.2.1 Example plugin code

A plugin is a standard Django/Python package. As quick example, let's create a webshop page.

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

Example code

For the `pagetypes.shoppage` package, the following files are needed:

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

`models.py`

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

```
from django.db import models
from fluent_pages.models import Page
from myshop.models import ProductCategory

class ProductCategoryPage(Page):
    product_category = models.ForeignKey(ProductCategory)

    class Meta:
        verbose_name = 'Product category page'
        verbose_name_plural = 'Product category pages'
```

This `Page` class provides the basic fields to integrate the model in the tree.

`page_type_plugins.py`

The `page_type_plugins.py` file can contain multiple plugins, each should inherit from the `PageTypePlugin` class.

```
from django.conf.urls import patterns, url
from fluent_pages.extensions import PageTypePlugin, page_type_pool
from .models import ProductCategoryPage

@page_type_pool.register
```

(continues on next page)

(continued from previous page)

```

class ProductCategoryPagePlugin(PageTypePlugin):
    """
    A new page type plugin that binds the rendering and model together.
    """
    model = ProductCategoryPage
    render_template = "products/productcategorypage.html"

    # Custom URLs
    urls = patterns('myshop.views',
        url('^(?P<slug>[/]+)$', 'product_details'),
    )

```

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

The other fields, such as the `urls` are optional.

productcategorypage.html

The default `get_response()` code renders the page with a template.

This can be used to generate the HTML:

```

{% extends "pages/base.html" %}

{% block headtitle %}{{ page.title }}{% endblock %}

{% block main %}
    <p>
        Contents of the category: {{ page.product_category }} ({{ page.product_category.
        ↳products.count }} products).
    </p>

    <div id="products">
        ....
    </div>
{% endblock %}

```

Note how the page variable is available, and the extra `product_category` field can be accessed directly.

Wrapping up

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

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

Now, the plugin will be visible in the "Add page" dialog:

Add Page

Page type:
 Page
 Plain text file
 Product category page
 Redirect

[Submit](#)

After adding it, the admin interface will be visible:

Change Product category page

[History](#)
[View on site](#)

Title:

Slug:
The slug is used in the URL of the page

Status: Published Draft

Show in navigation

Contents

Product category: +

[Menu structure \(Show\)](#)

[Publication settings \(Show\)](#)

✖ Delete
[Save and add another](#)
[Save and continue editing](#)
[Save](#)

The appearance on the website depends on the site’s CSS theme, of 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 page type is made of two classes:

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

The plugin class renders the model instance using:

- A custom `get_response()` method.
- The `render_template` attribute, `get_render_template()` method and optionally `get_context()` method.

Simply stated, a plugin provides the “view” of the “page”.

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:

```
@page_type.register
class MyPageType(PageTypePlugin):
    # ...

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

To add more context data, overwrite the `get_context` method.

Custom rendering

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

The `textfile` and `redirectnode` page types use this for example:

```
def get_response(self, request, redirectnode, **kwargs):
    response = HttpResponseRedirect(redirectnode.new_url)
    response.status_code = redirectnode.redirect_type
    return response
```

The standard `get_response()` method basically does the following:

```
def get_response(self, request, page, **kwargs):
    render_template = self.get_render_template(request, page, **kwargs)
    context = self.get_context(request, page, **kwargs)
    return self.response_class(
        request = request,
        template = render_template,
        context = context,
    )
```

- It takes the template from `get_render_template()`.
- It uses the context provided by `get_context()`.
- It uses `response_class()` class to output the response.

Note: The `PageTypePlugin` class is instantiated once, just like the `ModelAdmin` class. Unlike the Django class based views, it's not possible to store state at the local instance.

3.2.3 Customizing the admin interface

The admin rendering of a page type is fully customizable.

Changed in version 1.2: It's no longer necessary to define the `model_admin` attribute. Registering the custom admin class instead using `admin.site.register()` or the `@admin.register()` decorator.

```
@page_type_pool.register
class ProductCategoryPagePlugin(PageTypePlugin):
    """
    A new page type plugin that binds the rendering and model together.
    """
    model = ProductCategoryPage
    render_template = "products/productcategorypage.html"

    model_admin = ProductCategoryPageAdmin # only required for fluent-pages 1.1 and
    ↪below.
```

The admin class needs to inherit from one of the following classes:

- `fluent_pages.admin.PageAdmin`
- `fluent_pages.admin.HtmlPageAdmin` - in case the model extends from `HtmlPage`
- `fluent_pages.pagetypes.fluentpage.admin.FluentPageAdmin` - in case the model extends from `FluentPageBase`

The admin can be used to customize the “add” and “edit” fields for example:

```
from django.contrib import admin
from fluent_pages.admin import PageAdmin
from .models import ProductCategoryPage

@admin.register(ProductCategoryPage)
class ProductCategoryPageAdmin(PageAdmin):
    raw_id_fields = PageAdmin.raw_id_fields + ('product_category',)
```

Despite being registered in the admin, the model won’t show up in the index page. The “list” page is never used, as this is rendered by the main `PageAdmin` class. Only the “add” and “edit” page are exposed by the `PageAdmin` class too.

Customizing fieldsets

To deal with model inheritance, the fieldsets are not set in stone in the `fieldsets` attribute. Instead, the fieldsets are created dynamically using the `base_fieldsets` value as starting point. Any unknown fields (e.g. added by derived models) will be added to a separate “Contents” fieldset.

The default layout of the `PageAdmin` class is:

```
base_fieldsets = (
    PageAdmin.FIELDSET_GENERAL,
    PageAdmin.FIELDSET_MENU,
    PageAdmin.FIELDSET_PUBLICATION,
)
```

The default layout of the `HtmlPageAdmin` is:

```
base_fieldsets = (
    HtmlPageAdmin.FIELDSET_GENERAL,
    HtmlPageAdmin.FIELDSET_SEO,
    HtmlPageAdmin.FIELDSET_MENU,
    HtmlPageAdmin.FIELDSET_PUBLICATION,
)
```


The title of the custom “Contents” fieldset is configurable with the `extra_fieldset_title` attribute.

Customizing the form

Similar to the `base_fieldsets` attribute, there is a `base_form` attribute to use for the form.

Inherit from the `PageAdminForm` class to create a custom form, so all base functionality works.

3.2.4 Adding custom URLs

Page types can provide custom URL patterns. These URL patterns are relative to the place where the page is added to the page tree.

This feature is useful for example to:

- Have a “Shop” page type where all products are sub pages.
- Have a “Blog” page type where all articles are displayed below.

To use this feature, provide a `URLconf` or an inline `patterns()` list in the page type plugin.

Basic example

To have a plugin with custom views, add the `urls` attribute:

```
@page_type_pool.register
class ProductCategoryPagePlugin(PageTypePlugin):
    # ...

    urls = patterns('myshop.views',
        url('^(?P<slug>[^/]+)/$', 'product_details'),
    )
```

The view is just a plain Django view:

```
from django.http import HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from myshop.models import Product

def product_details(request, slug):
    product = get_object_or_404(Product, slug=slug)
    return render(request, 'products/product_details.html', {
        'product': product
    })
```

Other custom views can be created in the same way.

Resolving URLs

The URLs can’t be resolved using the standard `reverse()` function unfortunately. The main reason is that it caches results internally for the lifetime of the WSGI container, meanwhile pages may be rearranged by the admin.

Hence, a `app_reverse()` function is available. It can be used to resolve the product page:

```
from fluent_pages.urlresolvers import app_reverse

app_reverse('product_details', kwargs={'slug': 'myproduct'})
```

In templates, there is an `appurl` tag which accomplishes the same effect:

```
{% load appurl_tags %}

<a href="{% appurl 'product_details' slug='myproduct' %}">My Product</a>
```

See also:

The [example application](#) in the source demonstrates this feature.

Compatibility with regular URLconf

An application can provide a standard `urls.py` for regular Django support, and still support page type URLs too. For this special case, the `mixed_reverse()` function is available. It attempts to resolve the view in the standard URLconf first, and falls back to `app_reverse()` if the view is not found there.

A `mixedurl` template tag has to be included in the application itself. Use the following code as example:

```
@register.tag
def mixedurl(parser, token):
    if 'fluent_pages' in settings.INSTALLED_APPS:
        from fluent_pages.templatetags.appurl_tags import appurl
        return appurl(parser, token)
    else:
        from django.template.defaulttags import url
        return url(parser, token)
```

See also:

The [django-fluent-blogs](#) application uses this feature to optionally integrate the blog articles to the page tree.

3.2.5 Integration with fluent-contents

The bundled *fluent page type* provides a page type where parts of the page can be filled with flexible content blocks. This feature can be used in your custom page types as well. The `fluent_pages.integration.fluent_contents` package provides all classes to make this integration painless.

Note: The support for those flexible blocks is provided by the stand-alone [django-fluent-contents](#) package, which is an optional dependency. Both *fluent-pages* and *fluent-contents* are stand-alone packages, which you can mix and match freely with other software.

Example case: donation page

In some pages, the user is guided through several steps. At each step, staff members have to be able to enter CMS page content.

This can be handled in a smart way by exposing all situations through a single page.

In this simple example, a “Donation Page” was created as custom page type. This allowed editing the opening view, and “thank you view” as 2 separate area’s.

`models.py`

```

from fluent_pages.integration.fluent_contents.models import FluentContentsPage

class DonationPage(FluentContentsPage):
    """
    It has a fixed template, which can be used to enter the contents for all wizard_
    ↪ steps.
    """
    class Meta:
        verbose_name = _("Donation Page")
        verbose_name_plural = _("Donation Pages")

```

`admin.py`

```

from fluent_pages.integration.fluent_contents.admin import FluentContentsPageAdmin

class DonationPageAdmin(FluentContentsPageAdmin):
    """
    Admin for "Donation Page" in the CMS.
    """
    # This template is read to fetch the placeholder data, which displays the tabs.
    placeholder_layout_template = 'pages/donation.html'

```

`page_type_plugins.py`

```

from django.conf.urls import url
from fluent_pages.extensions import page_type_pool
from fluent_pages.integration.fluent_contents.page_type_plugins import_
↪ FluentContentsPagePlugin
from .admin import DonationPageAdmin
from .models import DonationPage
from .views import DonationSuccessView

@page_type_pool.register
class DonationPagePlugin(FluentContentsPagePlugin):
    """
    Custom page type for the donation page

    This page type can be inserted somewhere within the page tree,
    and all it's wizard sub-pages will be read from it.
    """
    model_admin = DonationPageAdmin
    model = DonationPage

    urls = [
        # root = donation starting page (handled as standard page)
        url(r'^step1/', DonationStep1View.as_view(), name='donation-step1'),
        url(r'^step2/', DonationStep2View.as_view(), name='donation-step2'),
        url(r'^thank-you/', DonationSuccessView.as_view(), name='donation-success'),
    ]

```

views.py

```

from django.views.generic import TemplateView
from fluent_pages.views import CurrentPageTemplateMixin

class DonationViewBase(CurrentPageTemplateMixin):
    # There is no need to redeclare the template here,
    # it's auto selected from the plugin/admin by CurrentPageTemplateMixin.
    #template_name = 'pages/donation.html'
    render_tab = ''

    def get_context_data(self, **kwargs):
        context = super(DonationViewBase, self).get_context_data(**kwargs)
        context['render_tab'] = self.render_tab
        return context

class DonationStep1(DonationViewBase, FormView):
    """
    Success page
    """
    view_url_name = 'donation-step1' # for django-parler's {% get_translated_url %}
    render_tab = 'step1' # for the template
    template_name = ""

    # ...

class DonationSuccessView(DonationViewBase, TemplateView):
    """
    Success page
    """
    view_url_name = 'donation-success'
    render_tab = 'success'
    template_name = ""

```

templates/pages/donation.html

```

{% extends "pages/base.html" %}{% load fluent_contents_tags %}
{% comment %}
    This template implements a sort-of "wizard" like view.
    By exposing all variations in the placeholders,
    the CMS view will display tabs for each option.
{% endcomment %}

{% block main %}
    <div class="constrained-subtle">
        <div class="container">

            {% if not render_tab %}
                {% page_placeholder "donation-intro" title="Donation intro" role="m"
↳fallback=True %}
            {% elif render_tab == 'step1' %}
                {% page_placeholder "donation-step1" title="Step 1" role="s"
↳fallback=True %}
            {% elif render_tab == 'success' %}

```

(continues on next page)

(continued from previous page)

```

        {% page_placeholder "donation-success" title="Success page" role="s"
↪ fallback=True %}
        {% endif %}

    </div>
</div>
</div>
{% endblock %}

```

This template leverages the features of `django-fluent-contents`. Each step can now be filled in by a staff member with CMS content. Even the form can now be added as a “Content plugin”. By using `FLUENT_CONTENTS_PLACEHOLDER_CONFIG`, the allowed plugin types can be limited per step. For example:

```

FLUENT_CONTENTS_PLACEHOLDER_CONFIG = {
    # ...

    # The 'pages/donation.html' template:
    'donation-intro': {
        'plugins': (
            'DonateButtonPlugin', 'TextPlugin',
        ),
    },
    'donation-step1': {
        'plugins': (
            'DonationForm1Plugin', 'TextPlugin',
        ),
    },
    'giveone-success': {
        'plugins': (
            'ThankYouPlugin',
            'TextPlugin',
            'RawHtmlPlugin', # For social media embed codes
        ),
    },
}
})

```


4.1 Low-level API's

This package provides internal API's, so projects can use those to query the tree or even prefill it.

Note: When using the Python shell, make sure the activate a language first.

```
from django.conf import settings
from django.utils.translations import activate

activate(settings.LANGUAGE_CODE)
```

4.1.1 Query pages

When you query the general *Page* or *UrlNode* model, the pages are returned in their specific type.

```
>>> from fluent_pages.models import Page
>>> Page.objects.published()
<Queryset [ <FluentPage: Homepage>, <BlogPage: Blog>, <FluentPage: Contact> ]>
```

To filter the results, use one of these methods:

- `parent_site()` - select a different site.
- `get_for_path()` - find the node for a path.
- `best_match_for_path()` - find the node starting with

Tip: Finding pages by ID

When `FLUENT_PAGES_KEY_CHOICES` is set, specific pages can be fetched using `Page.objects.get_for_key()`.

4.1.2 Creating pages

The tree can hold different page types. Always create the specific type needed, for example:

```
from django.contrib.auth import get_user_model
from fluent_pages.pagetypes.flatpage.models import FlatPage

User = get_user_model()
admin = User.objects.get(active=True, username='admin')

page = FlatPage.objects.create(
    # Standard fields
    title="Flat test",
    slug="flat-test",
    status=FlatPage.PUBLISHED,
    author=admin,

    # Type specific fields:
    content="This page is created via the API!"
)
```

Now the page will appear too:

```
>>> from fluent_pages.models import Page
>>> Page.objects.published()
<Queryset [<FluentPage: Homepage>, <BlogPage: Blog>, <FluentPage: Contact>,
↳<FlatPage: Flat test>]>
```

The same approach can be used for other page types. Review the model API to see which fields can be used:

- `FlatPage` (provide content and optionally, `template_name`).
- `RedirectNode` (provide `new_url` and optionally, `redirect_type`).
- `TextFile` (provide content and optionally, `content_type`).

Pages with visible HTML content also inherit from `HtmlPage`, which makes the `meta_keywords`, `meta_description` and optional `meta_title` available too.

Fluent content pages

A similar way can be used for pages with block content. This uses the `django-fluent-contents` and `django-parler` API's too:

```
from django.contrib.auth import get_user_model
from fluent_pages.pagetypes.fluentpage.models import FluentPage
from fluent_contents.plugins.textitem.models import TextItem
from fluent_contents.plugins.oembeditem.models import OEmbedItem

User = get_user_model()
admin = User.objects.get(active=True, username='admin')

page = FluentPage.objects.language('en').create(
```

(continues on next page)

(continued from previous page)

```

# Standard fields
title="Fluent test",
slug="fluent-test",
status=FluentPage.PUBLISHED,
author=admin,
)

# Create the placeholder
placeholder = page.create_placeholder('main')

# Create the content items
TextItem.objects.create_for_placeholder(placeholder, text="Hello, World!")
OEmbedItem.objects.create_for_placeholder(placeholder, embed_url="https://vimeo.com/
↪channels/952478/135740366")

# Adding another language:
page.create_translation('nl')
TextItem.objects.create_for_placeholder(placeholder, language_code="nl", text="Hello,
↪World NL!")
OEmbedItem.objects.create_for_placeholder(placeholder, language_code="nl", embed_url=
↪"https://vimeo.com/channels/952478/135740366")

```

The `.language('en')` is not required, as the current language is selected. However, it's good to be explicit in case your project is multilingual. When no language code is given to `create_for_placeholder()`, it uses the current language that the parent object (i.e. the page) has.

4.2 API documentation

4.2.1 `fluent_pages.adminui`

A set of base classes, to build custom admin pages, for your page types.

These classes are separate from the `fluent_pages.admin` module on purpose. Custom page type plugins can inherit these classes to provide their enhanced admin interface. If this module could be called `fluent_pages.admin`, it would invoke the app registry and prevent any further model initialization.

The `PageAdmin` class

```
fluent_pages.adminui.PageAdmin
    alias of fluent_pages.adminui.pageadmin.DefaultPageChildAdmin
```

```
class fluent_pages.adminui.DefaultPageChildAdmin (model, admin_site, *args, **kwargs)
    Bases: fluent_pages.adminui.urlnodechildadmin.UrlNodeChildAdmin
```

The base class for administrating pages. When a custom page type implements a custom admin, use this class as its base. See the code in `fluent_pages/pagetypes/*/admin.py` for examples. To deal with model inheritance, define the fieldsets using the `base_fieldsets` option. For example:

```

base_fieldsets = (
    PageAdmin.FIELDSET_GENERAL,
    PageAdmin.FIELDSET_MENU,
    PageAdmin.FIELDSET_PUBLICATION,
)

```

By using `base_fieldsets` instead of the `ModelAdmin.fieldsets` attribute, any additional fields from a derived model will be displayed in a separate fieldset automatically. The title of the fieldset is configurable with the `extra_fieldset_title` attribute. It's "Contents" by default.

The admin class can be extended with mixins by defining `FLUENT_PAGES_PARENT_ADMIN_MIXIN / FLUENT_PAGES_CHILD_ADMIN_MIXIN`.

base_form

alias of `PageAdminForm`

base_model

alias of `fluent_pages.models.db.Page`

formfield_for_foreignkey (*db_field, request=None, **kwargs*)

Get a form Field for a ForeignKey.

render_change_form (*request, context, add=False, change=False, form_url="", obj=None*)

Insert the language tabs.

base_change_form_template = `'admin/fluent_pages/page/base_change_form.html'`

The default template name, which is available in the template context. Use `{% extend base_change_form_template %}` in templates to inherit from it.

change_form_template

Dynamic property to support transition to regular models.

This automatically picks `admin/parler/change_form.html` when the admin uses a translatable model.

The `PageAdminForm` class

class `fluent_pages.adminui.PageAdminForm` (**args, **kwargs*)

The base class for all admin forms.

This form validates the "Slug" and "Override URL" fields.

The `HtmlPageAdmin` class

class `fluent_pages.adminui.HtmlPageAdmin` (*model, admin_site, *args, **kwargs*)

Bases: `fluent_pages.adminui.pageadmin.DefaultPageChildAdmin`

The modeladmin configured to display `HtmlPage` models. The `HtmlPage` also displays a keywords and description field.

This admin class defines another fieldset: `FIELDSET_SEO`. The default fieldset layout is:

```
base_fieldsets = (
    HtmlPageAdmin.FIELDSET_GENERAL,
    HtmlPageAdmin.FIELDSET_SEO,
    HtmlPageAdmin.FIELDSET_MENU,
    HtmlPageAdmin.FIELDSET_PUBLICATION,
)
```

formfield_for_dbfield (*db_field, **kwargs*)

Allow `formfield_overrides` to contain field names too.

4.2.2 fluent_pages.adminui.utils

Utility functions related to admin views.

`fluent_pages.adminui.utils.get_page_admin_url(page)`

Return the admin URL for a page.

`fluent_pages.adminui.utils.get_current_edited_page(request)`

Return the *Page* object which is currently being edited in the admin. Returns *None* if the current view isn't the "change view" of the the *Page* model.

4.2.3 fluent_pages.extensions

Special classes to extend the module; e.g. page type 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 PageTypePlugin class

class `fluent_pages.extensions.PageTypePlugin`

The base class for a page type plugin.

To create a new plugin, derive from this class and call `page_type_pool.register` to enable it. For example:

```
from fluent_pages.extensions import PageTypePlugin, page_type_pool
from mycms.models import MyCustomPage

@page_type_pool.register
class MyCustomPagePlugin(PageTypePlugin):
    model = MyCustomPage
    render_template = "mycustompage/example.html"
```

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

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

To customize the admin, define the `model_admin` attribute. The provided class should inherit from the *PageAdmin* class.

The output of a page is fully customizable in the page type plugin. By default, `render_template` will be used but you can also override `get_render_template()`, `get_context()` or even `get_response()`. The latter gives full control over the *HttpResponse* to send to the client.

Page types can provide additional views, relative to the location where the page is placed in the tree. A shopping module for example, can display products as sub pages. Provide an URLconf to the `urls` attribute to use this feature, and resolve those URLs using the `fluent_pages.urlresolvers` module.

model_admin

alias of `fluent_pages.adminui.pageadmin.DefaultPageChildAdmin`

response_class

alias of `django.template.response.TemplateResponse`

__init__()

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

get_context() (*request, page, **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.

Note that this function can also be called by custom views when those views implement the `CurrentPageMixin` or `CurrentPageTemplateMixin`

get_model_instances()

Return all `Page` instances that are has created using this page types.

get_render_template() (*request, page, **kwargs*)

Return the template to render for the specific `page` or `request`, By default it uses the `render_template` attribute.

get_response() (*request, page, **kwargs*)

Render the page, and return the Django `HttpResponse`.

This is the main function to generate output of the page. By default, it uses `get_render_template()`, `get_context()` and `response_class` to generate the output of the page. The behavior can be replaced completely by overriding this function.

get_url_resolver()

Access the URL resolver of the page type.

get_view_response() (*request, page, view_func, view_args, view_kwargs*)

Render the custom view that was exposed by the extra plugin URL patterns. This gives the ability to add extra middleware logic.

can_be_root = True

New in version 1.1.

Define whether this page type can be a root node.

can_have_children = True

Defines whether users are allowed to place sub pages below this node. When `is_file` is `True`, this is never possible.

child_types = []

New in version 1.1.

Defines which pages can be children of this node. List of values similar to those values accepted in a model `ForeignKey`.

default_in_sitemaps = True

New in version 0.9.

Tell whether the page type should be displayed in the sitemaps by default. This value can be changed for most pages in the admin interface.

is_file = False

Defines the page type represents a file; it neither has appended slash or does it allow children.

model = None

Defines the model to use to store the custom fields. It must derive from *Page*.

render_template = None

Defines the template to use for rendering the page.

sort_priority = 100

The sorting priority for the page type in the “Add Page” dialog of the admin.

type_id

Returns the *ContentType* id of the model.

type_name

Return the class name of the model, this is mainly provided for templates.

urls = None

Defines the URLs that the page provides relative to the current node. This can either be the name of a Python module with *urlpatterns* in it, or a direct inline *patterns()* list.

verbose_name

Returns the title for the plugin, by default it reads the *verbose_name* of the model.

The *PageTypePool* class

class *fluent_pages.extensions.PageTypePool*

The central administration of plugins.

__init__()

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

get_file_types()

Return the *ContentType* id’s of page types that act like files (no slash or children).

get_folder_types()

Return the *ContentType* id’s of page types that operate as a container for sub pages.

get_model_classes()

Return all model classes which are exposed by page types. Each model derives from *Page*.

get_plugin_by_model(model_class)

Return the corresponding *PageTypePlugin* for a given model.

get_plugins()

Return the *PageTypePlugin* instances which are loaded.

get_url_pattern_plugins()

Return the *PageTypePlugin* instances that provide URL patterns.

get_url_pattern_types()

Return the *ContentType* id’s of page types that provide URL patterns.

register(plugin)

Make a page type plugin known by the CMS.

Parameters plugin – The plugin class, deriving from *PageTypePlugin*.

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

The `page_type_pool` attribute

`fluent_pages.extensions.page_type_pool`
 The global plugin pool, a instance of the `PluginPool` class.

Other classes

exception `fluent_pages.extensions.PageTypeAlreadyRegistered`
 Raised when attempting to register a plugin twice.

exception `fluent_pages.extensions.PageTypeNotFound`
 Raised when the plugin could not be found in the rendering process.

4.2.4 `fluent_pages.integration.fluent_contents`

The base classes to create a page which can display *django-fluent-contents* models.

The API to interface with *django-fluent-contents* is public, and documented for this reason. In fact, this module is just a tiny bridge between the page type plugins and the *django-fluent-pages* API. It can be used to create custom page types that display `ContentItem` objects.

The following parts are provided:

- The admin class; `admin.FluentContentsPageAdmin`
- The page type model: `models.FluentContentsPage`
- The plugin class: `page_type_plugins.FluentContentsPagePlugin`

These classes can be imported from their respective subpackages:

```
from fluent_pages.integration.fluent_contents.admin import FluentContentsPageAdmin
from fluent_pages.integration.fluent_contents.models import FluentContentsPage
from fluent_pages.integration.fluent_contents.page_type_plugins import _
↪FluentContentsPagePlugin
```

The `FluentContentsPageAdmin` class

```
class fluent_pages.integration.fluent_contents.admin.FluentContentsPageAdmin (model,
                                                                              ad-
                                                                              min_site,
                                                                              *args,
                                                                              **kwargs)
```

This admin is a small binding between the pagetypes of *django-fluent-pages* and page contents of *django-fluent-contents*.

Use `{% extends base_change_form_template %}` in your page template, and it will all work properly.

get_all_allowed_plugins ()
 By default, all plugins are allowed, unless a placeholder puts a limit on this. The page will load much faster if the plugin types are limited globally here.

get_placeholder_data (*request*, *obj=None*)
 Read the placeholder data to display in the template. This reads `placeholder_layout` and `placeholder_layout_template`. It can be overwritten to return the layout depending on the page or request.

Tip: if the object is given, this could read `obj.plugin.get_render_template(request, obj)` too.

get_translation_objects (*request, language_code, obj=None, inlines=True*)

Make sure the translated ContentItem objects are also deleted when a translation is removed.

all_allowed_plugins = None

A static list of all allowed plugin names. This is read by `get_all_allowed_plugins()`

placeholder_layout = None

A fixed defined placeholder layout, which can be defined statically. This should be a list of `PlaceholderData` objects.

placeholder_layout_template = None

A fixed template, from which the placeholder data can be read. The `placeholder_layout` will be read automatically from the template.

The FluentContentsPage class

class `fluent_pages.integration.fluent_contents.models.FluentContentsPage` (**args, **kwargs*)

The base model to create a Page object which hosts placeholders and content items.

Parameters

- **id** (*AutoField*) – Id
- **polymorphic_ctype** (*ForeignKey to ContentType*) – Polymorphic ctype
- **parent** (*PageTreeForeignKey to UrlNode*) – Parent. You can also change the parent by dragging the page in the list.
- **parent_site** (*ForeignKey to Site*) – Parent site
- **status** (*CharField*) – Status
- **publication_date** (*DateTimeField*) – Publication date. When the page should go live, status must be “Published”.
- **publication_end_date** (*DateTimeField*) – Publication end date
- **in_navigation** (*BooleanField*) – Show in navigation
- **in_sitemaps** (*BooleanField*) – Include in search engine sitemaps
- **key** (*SlugField*) – Page identifier. A unique identifier that is used for linking to this page.
- **author** (*ForeignKey to User*) – Author
- **creation_date** (*DateTimeField*) – Creation date
- **modification_date** (*DateTimeField*) – Last modification
- **lft** (*PositiveIntegerField*) – Lft
- **rght** (*PositiveIntegerField*) – Rght
- **tree_id** (*PositiveIntegerField*) – Tree id
- **level** (*PositiveIntegerField*) – Level
- **urlnode_ptr** (*OneToOneField to UrlNode*) – Urlnode ptr
- **placeholder_set** (*PlaceholderRelation*) – Placeholder set

- `contentitem_set` (*ContentItemRelation*) – Contentitem set

`create_placeholder` (*slot*, *role='m'*, *title=None*)

Create a placeholder on this page.

To fill the content items, use `ContentItemModel.objects.create_for_placeholder()`.

Return type `Placeholder`

`get_content_items_by_slot` (*slot*)

Return all content items of the page, which are stored in the given slot name. :rtype: `ContentItemQuerySet`

`get_placeholder_by_slot` (*slot*)

Return a placeholder that is part of this page. :rtype: `Placeholder`

`contentitem_set`

Model field: contentitem set, accesses the M2M `FluentContentsPage` model.

`placeholder_set`

Model field: placeholder set, accesses the M2M `FluentContentsPage` model.

The `FluentContentsPagePlugin` class

`class` `fluent_pages.integration.fluent_contents.page_type_plugins.FluentContentsPagePlugin`

Base plugin to render a page with content items.

`model_admin`

alias of `fluent_pages.integration.fluent_contents.admin.FluentContentsPageAdmin`

`get_render_template` (*request*, *fluentpage*, ***kwargs*)

Overwritten to automatically pick up the template used in the admin.

`model = None`

Defines the model to use to store the custom fields. It must derive from `FluentContentsPage`.

4.2.5 `fluent_pages.models`

The data layer of the CMS, exposing all database models.

The objects can be imported from the main package. There are several sub packages:

db: The database models managers: Additional manager classes
 modeldata: Classes that expose model data in a sane way (for template designers)
 navigation: The menu navigation nodes (for template designers)

The `UrlNode` class

`class` `fluent_pages.models.UrlNode` (**args*, ***kwargs*)

The base class for all nodes; a mapping of an URL to content (e.g. a HTML page, text file, blog, etc..)

Parameters

- `id` (*AutoField*) – Id
- `polymorphic_ctype` (*ForeignKey to ContentType*) – Polymorphic ctype
- `parent` (*PageTreeForeignKey to UrlNode*) – Parent. You can also change the parent by dragging the page in the list.

- **parent_site** (ForeignKey to *Site*) – Parent site
- **status** (*CharField*) – Status
- **publication_date** (*DateTimeField*) – Publication date. When the page should go live, status must be “Published”.
- **publication_end_date** (*DateTimeField*) – Publication end date
- **in_navigation** (*BooleanField*) – Show in navigation
- **in_sitemaps** (*BooleanField*) – Include in search engine sitemaps
- **key** (*SlugField*) – Page identifier. A unique identifier that is used for linking to this page.
- **author** (ForeignKey to *User*) – Author
- **creation_date** (*DateTimeField*) – Creation date
- **modification_date** (*DateTimeField*) – Last modification
- **lft** (*PositiveIntegerField*) – Lft
- **rght** (*PositiveIntegerField*) – Rght
- **tree_id** (*PositiveIntegerField*) – Tree id
- **level** (*PositiveIntegerField*) – Level

exception DoesNotExist

exception MultipleObjectsReturned

__init__ (*args, **kwargs)

Replace Django’s inheritance accessor member functions for our model (`self.__class__`) with our own versions. We monkey patch them until a patch can be added to Django (which would probably be very small and make all of this obsolete).

If we have inheritance of the form ModelA -> ModelB -> ModelC then Django creates accessors like this:
 - ModelA: modelb - ModelB: modela_ptr, modelb, modelc - ModelC: modela_ptr, modelb, modelb_ptr, modelc

These accessors allow Django (and everyone else) to travel up and down the inheritance tree for the db object at hand.

The original Django accessors use our polymorphic manager. But they should not. So we replace them with our own accessors that use our appropriate `base_objects` manager.

delete (*args, **kwargs)

Calling `delete` on a node will delete it as well as its full subtree, as opposed to reattaching all the subnodes to its parent node.

There are no argument specific to a MPTT model, all the arguments will be passed directly to the django’s `Model.delete`.

`delete` will not return anything.

get_absolute_url ()

Return the URL to this page.

get_absolute_urls ()

Return all available URLs to this page.

is_publication_date_active (*date=None*)

Return whether a configured publication date is within range.

New in version 2.0.4.

save (**args, **kwargs*)

Save the model, and update caches.

save_translation (*translation, *args, **kwargs*)

Update the fields associated with the translation. This also rebuilds the decedent URLs when the slug changed.

breadcrumb

Return the breadcrumb; all parent pages leading to the current page, including current page itself.

can_be_root

Return `True` when the page type can be used as root page.

can_have_children

Return `True` when the node can have child nodes.

child_types

Return a list of content type ids of nodes that can be children of this node.

default_url

The internal implementation of `get_absolute_url()`. This function can be used when overriding `get_absolute_url()` in the settings. For example:

```
ABSOLUTE_URL_OVERRIDES = {
    'fluent_pages.Page': lambda o: "http://example.com" + o.default_url
}
```

is_draft

Return whether the node is still a draft.

is_file

Return `True` when the node represents a file (can't have children, doesn't have a layout).

is_first_child

Return `True` when the node is the first sibling.

is_last_child

Return `True` when the node is the last sibling.

is_published

Return whether the node is published.

last_modified

Return the last modification date of the page. Currently this is the last time the page was saved. This is implemented as separate property, to be extended to page content in the future.

page_key

Ensure `get_child_types` is run once per plugin model.

plugin

Access the parent plugin which renders this model.

url

The URL of the page, provided for template code.

The Page class

class `fluent_pages.models.Page(*args, **kwargs)`

The base class for all all `UrlNode` subclasses that display pages.

This is a proxy model that changes the appearance of the node in the admin. The `UrlNode` displays the URL path, while this model displays the title.

Parameters

- **id** (`AutoField`) – Id
- **polymorphic_ctype** (`ForeignKey` to `ContentType`) – Polymorphic ctype
- **parent** (`PageTreeForeignKey` to `UrlNode`) – Parent. You can also change the parent by dragging the page in the list.
- **parent_site** (`ForeignKey` to `Site`) – Parent site
- **status** (`CharField`) – Status
- **publication_date** (`DateTimeField`) – Publication date. When the page should go live, status must be “Published”.
- **publication_end_date** (`DateTimeField`) – Publication end date
- **in_navigation** (`BooleanField`) – Show in navigation
- **in_sitemaps** (`BooleanField`) – Include in search engine sitemaps
- **key** (`SlugField`) – Page identifier. A unique identifier that is used for linking to this page.
- **author** (`ForeignKey` to `User`) – Author
- **creation_date** (`DateTimeField`) – Creation date
- **modification_date** (`DateTimeField`) – Last modification
- **lft** (`PositiveIntegerField`) – Lft
- **rght** (`PositiveIntegerField`) – Rght
- **tree_id** (`PositiveIntegerField`) – Tree id
- **level** (`PositiveIntegerField`) – Level

exception `DoesNotExist`

exception `MultipleObjectsReturned`

The HtmlPage class

class `fluent_pages.models.HtmlPage(*args, **kwargs)`

The `HtmlPage` is the base for all page types that display HTML. This is a proxy model, which adds translatable SEO fields and a customizable title.

Parameters

- **id** (`AutoField`) – Id
- **polymorphic_ctype** (`ForeignKey` to `ContentType`) – Polymorphic ctype
- **parent** (`PageTreeForeignKey` to `UrlNode`) – Parent. You can also change the parent by dragging the page in the list.

- **parent_site** (ForeignKey to *Site*) – Parent site
- **status** (*CharField*) – Status
- **publication_date** (*DateTimeField*) – Publication date. When the page should go live, status must be “Published”.
- **publication_end_date** (*DateTimeField*) – Publication end date
- **in_navigation** (*BooleanField*) – Show in navigation
- **in_sitemaps** (*BooleanField*) – Include in search engine sitemaps
- **key** (*SlugField*) – Page identifier. A unique identifier that is used for linking to this page.
- **author** (ForeignKey to *User*) – Author
- **creation_date** (*DateTimeField*) – Creation date
- **modification_date** (*DateTimeField*) – Last modification
- **lft** (*PositiveIntegerField*) – Lft
- **rght** (*PositiveIntegerField*) – Rght
- **tree_id** (*PositiveIntegerField*) – Tree id
- **level** (*PositiveIntegerField*) – Level

exception DoesNotExist

exception MultipleObjectsReturned

delete (**args, **kwargs*)

Calling `delete` on a node will delete it as well as its full subtree, as opposed to reattaching all the subnodes to its parent node.

There are no argument specific to a MPTT model, all the arguments will be passed directly to the django’s `Model.delete`.

`delete` will not return anything.

meta_robots

The value for the `<meta name="robots" content=".."/>` tag. It defaults to `noindex` when `in_sitemaps` is `False`.

The PageLayout class

class `fluent_pages.models.PageLayout` (**args, **kwargs*)

A `PageLayout` object defines a template that can be used by a page.

Parameters

- **id** (*AutoField*) – Id
- **key** (*SlugField*) – Key. A short name to identify the layout programmatically
- **title** (*CharField*) – Title
- **template_path** (*TemplateFilePathField*) – Template file

exception DoesNotExist

exception MultipleObjectsReturned

get_template()
Return the template to render this layout.

The `UrlNodeManager` class

class `fluent_pages.models.UrlNodeManager`
Extra methods attached to `UrlNode.objects` and `Page.objects`.

queryset_class
alias of `UrlNodeQuerySet`

best_match_for_path (*path*, *language_code=None*)
Return the `UrlNode` that is the closest parent to the given path.

`UrlNode.objects.best_match_for_path('/photos/album/2008/09')` might return the page with url `'/photos/album/'`.

Changed in version 0.9: This filter only returns the pages of the current site.

get_for_key (*key*)
New in version 0.9: Return the `UrlNode` for the given key.

The key can be a slug-like value that was configured in `FLUENT_PAGES_KEY_CHOICES`.

get_for_path (*path*, *language_code=None*)
Return the `UrlNode` for the given path. The path is expected to start with an initial slash.

Raises `UrlNode.DoesNotExist` when the item is not found.

Changed in version 0.9: This filter only returns the pages of the current site.

in_navigation (*for_user=None*)
Return only pages in the navigation.

in_sitemaps ()
New in version 0.9.

Return only pages in the navigation.

parent_site (*site*)
New in version 0.9: Filter to the given site.

published (*for_user=None*)
Return only published pages for the current site.

Changed in version 0.9: This filter only returns the pages of the current site.

toplevel ()
Return all pages which have no parent.

toplevel_navigation (*current_page=None*, *for_user=None*, *language_code=None*)
Return all toplevel items, ordered by menu ordering.

When *current_page* is passed, the object values such as `'is_current'` will be set.

url_pattern_types ()
Return only page types which have a custom `URLpattern` attached.

4.2.6 `fluent_pages.models.navigation`

The data model to walk through the site navigation.

These objects only return the relevant data for the menu/breadcrumb in a fixed, minimalistic, API so template designers can focus on that.

To walk through the site content in Python code, use the *Page* model directly. It offers properties such as `parent` and `children` (a `RelatedManager`), and methods such as `get_parent()` and `get_children()` through the *MPTTModel* base class.

The `NavigationNode` class

class `fluent_pages.models.navigation.NavigationNode`

The base class for all navigation nodes, whether model-based or virtually inserted ones.

get_children()

Provided for compatibility with mptt recursetree

get_level()

Provided for compatibility with mptt recursetree

children

The list of children.

has_children

Whether the node has children.

is_active

True if the node is the currently active page.

is_child_active

True if a child of this node is the currently active page.

is_draft

True if the node is a draft item.

is_published

True if the node is a normal published item.

level

The depth of the menu level.

parent

The parent node.

slug

The slug of the node.

title

The title of the node.

url

The URL of the node.

The `PageNavigationNode` class

class `fluent_pages.models.navigation.PageNavigationNode` (*page*, *parent_node=None*,
max_depth=9999,
current_page=None,
for_user=None)

An implementation of the *NavigationNode* for *Page* models.

`__init__` (*page, parent_node=None, max_depth=9999, current_page=None, for_user=None*)
Initialize the node with a Page.

children
The list of children.

has_children
Whether the node has children.

is_active
True if the node is the currently active page.

is_child_active
True if a child of this node is the currently active page.

is_draft
True if the node is a draft item.

is_published
True if the node is a normal published item.

page
New in version 0.9: Provide access to the underlying page object, if it exists.

parent
The parent node.

4.2.7 fluent_pages.templatetags.appurl_tags

Template tag to resolve page URLs which have an URLconf attached to them. Load this module using:

```
{% load appurl_tags %}
```

Usage:

```
{% appurl "my_viewname" %}
{% appurl "my_viewname" arg1 arg2 %}
{% appurl "my_viewname" kwarg1=value kwarg2=value as varname %}
```

4.2.8 fluent_pages.templatetags.fluent_pages_tags

Template tags to request fluent page content in the template. Load this module using:

```
{% load fluent_pages_tags %}
```

The `render_breadcrumb` tag

Render the breadcrumb of the site, starting at the current page. This function either uses the default template, or a custom template if the `template` argument is provided.

```
{% render_breadcrumb template="fluent_pages/parts/breadcrumb.html" %}
```

The `render_menu` tag

Render the menu of the site. The `max_depth`, `parent` and `template` arguments are optional.

```
{% render_menu max_depth=1 parent="/documentation/" template="fluent_pages/parts/menu.
↪html" %}
```

The `get_fluent_page_vars` tag

Introduces the `site` and `page` variables in the template. This can be used for pages that are rendered by a separate application.

```
{% get_fluent_page_vars %}
```

4.2.9 `fluent_pages.sitemaps`

This module provides integration for the `django.contrib.sitemaps` module. This can be done using:

```
from fluent_pages.sitemaps import PageSitemap

sitemaps = {
    'pages': PageSitemap,
}

urlpatterns += [
    url(r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': ↪
↪sitemaps}),
]
```

The `PageSitemap` class

class `fluent_pages.sitemaps.PageSitemap`

The sitemap definition for the pages created with *django-fluent-pages*. It follows the API for the `django.contrib.sitemaps` module.

items ()

Return all items of the sitemap.

lastmod (*urlnode*)

Return the last modification of the page.

location (*urlnode*)

Return url of a page.

4.2.10 `fluent_pages.urlresolvers`

URL Resolving for dynamically added pages.

`fluent_pages.urlresolvers.app_reverse` (*viewname*, *args=None*, *kwargs=None*, *multiple=False*, *ignore_multiple=False*, *current_page=None*, *language_code=None*)

Locate an URL which is located under a page type.

`fluent_pages.urlresolvers.mixed_reverse` (*viewname*, *args=None*, *kwargs=None*, *current_app=None*, *current_page=None*, *language_code=None*, *multiple=False*, *ignore_multiple=False*)

Attempt to reverse a normal URLconf URL, revert to `app_reverse()` on errors.

`fluent_pages.urlresolvers.clear_app_reverse_cache()`

Clear the cache for the `app_reverse()` function. This only has to be called when doing bulk update/delete actions that circumvent the individual model classes.

Other classes

exception `fluent_pages.urlresolvers.MultipleReverseMatch`

Raised when an `app_reverse()` call returns multiple possible matches.

exception `fluent_pages.urlresolvers.PageTypeNotMounted`

Raised when the `app_reverse()` function can't find the required plugin in the page tree.

4.2.11 fluent_pages.views

All views of the CMS

The CurrentPageMixin class

class `fluent_pages.views.CurrentPageMixin`

Access the current page. This can be used for views which are defined as page type views in `PageTypePlugin.urls`.

The template context will have the same variables as the regular page templates would have, which are: * page * site * `FLUENT_PAGES_BASE_TEMPLATE`

get_context_data (***kwargs*)

Add the plugin context to the template.

get_current_page ()

Return the current page.

get_view_url ()

When using the `ViewUrlMixin.view_url_name` feature of *django-parler*, this makes sure that mounted pages are also supported.

It uses `fluent_pages.urlresolvers.mixed_reverse()` function to resolve the `view_url_name`.

The CurrentPageTemplateMixin class

class `fluent_pages.views.CurrentPageTemplateMixin`

Automatically reuse the template of the current page for the URL pattern view.

get_template_names ()

Auto-include the template of the CMS page.

The RobotsTxtView class

class `fluent_pages.views.RobotsTxtView` (**kwargs)
 Exposing a robots.txt template in the Django project.

Add this view to the `urls.py`:

```
from fluent_pages.views import RobotsTxtView

urlpatterns = [
    # ...

    url(r'^robots.txt$', RobotsTxtView.as_view()),
]
```

Naturally, this pattern should be placed outside `i18n_patterns()` as it should appear at the top level.

A robots.txt template is included by default, which you have override in your own project. Possible templates could look like:

Simple:

```
Sitemap: {{ ROOT_URL }}sitemap.xml
{% if ROBOTS_TXT_DISALLOW_ALL %}
User-agent: *
Disallow: /
{% endif %}
```

Sitemaps per language for usage with `i18n_patterns()`:

```
{% for language in language_codes %}Sitemap: {{ ROOT_URL }}{{ language }}/sitemap.
→xml
{% endfor %}
```

Alternative:

```
{% for sitemap_url in sitemap_urls %}Sitemap: {{ sitemap_url }}
{% endfor %}
```

get_i18n_patterns_codes ()

Return the possible values that `i18n_patterns()` support.

get_sitemap_urls (root_url)

Return all possible sitemap URLs, which the template can use.

has_i18n_patterns_urls ()

Check whether something like `i18n_patterns()` is used.

render_to_response (context, **response_kwargs)

Return a response, using the `response_class` for this view, with a template rendered with the given context.

Pass `response_kwargs` to the constructor of the response class.

content_type = 'text/plain'

The `content_type` to return.

template_name = 'robots.txt'

The template to render. You can override this template.

4.2.12 Deprecated modules

fluent_pages.pagetypes.fluentpage.admin

The `FluentPageAdmin` class

```
class fluent_pages.pagetypes.fluentpage.admin.FluentPageAdmin (model, admin_site, *args, **kwargs)
```

This admin is a small binding between the pagetypes of *django-fluent-pages* and page contents of *django-fluent-contents*.

Note: To create custom page types that combine both apps, consider using *fluent_pages.integration.fluent_contents.admin.FluentContentsPageAdmin* instead. In fact, the code in this class concerns with the layout mechanism that is specific for this implementation.

To build a variation of this page, see the API documentation of [Creating a CMS system](#) in the *django-fluent-contents* documentation to implement the required APIs.

`base_form`

alias of `FluentPageAdminForm`

formfield_for_foreignkey (*db_field*, *request=None*, ***kwargs*)

Get a form Field for a ForeignKey.

get_layout_view (*request*, *id*)

Return the metadata about a layout

get_page_template (*page*)

Return the template that is associated with the page.

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

Provides a list of `fluent_contents.models.PlaceholderData` classes, that describe the contents of the template.

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

Determine which fields are readonly. This includes the shared fields if the user has no permission to change them.

get_urls ()

Introduce more urls

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

Whether the user can change the page layout.

fluent_pages.pagetypes.fluentpage.models

The `AbstractFluentPage` class

```
class fluent_pages.pagetypes.fluentpage.models.AbstractFluentPage (*args, **kwargs)
```

A `FluentPage`` represents one HTML page of the site.

Note: If you really like to use the `layout` field in our custom applications, inherit from this class. Otherwise, please use *fluent_pages.integration.fluent_contents.models.FluentContentsPage*

instead.

This class is abstract, so it's easy to reuse the same CMS functionality in your custom page types without introducing another table/join indirection in the database. Naturally, the same layout mechanism is used. In case the `layout` should be handled differently, please consider building a variation of this page type application.

Parameters

- **id** (*AutoField*) – Id
- **polymorphic_ctype** (*ForeignKey to ContentType*) – Polymorphic ctype
- **parent** (*PageTreeForeignKey to UrlNode*) – Parent. You can also change the parent by dragging the page in the list.
- **parent_site** (*ForeignKey to Site*) – Parent site
- **status** (*CharField*) – Status
- **publication_date** (*DateTimeField*) – Publication date. When the page should go live, status must be “Published”.
- **publication_end_date** (*DateTimeField*) – Publication end date
- **in_navigation** (*BooleanField*) – Show in navigation
- **in_sitemaps** (*BooleanField*) – Include in search engine sitemaps
- **key** (*SlugField*) – Page identifier. A unique identifier that is used for linking to this page.
- **author** (*ForeignKey to User*) – Author
- **creation_date** (*DateTimeField*) – Creation date
- **modification_date** (*DateTimeField*) – Last modification
- **lft** (*PositiveIntegerField*) – Lft
- **rght** (*PositiveIntegerField*) – Rght
- **tree_id** (*PositiveIntegerField*) – Tree id
- **level** (*PositiveIntegerField*) – Level
- **urlnode_ptr** (*OneToOneField to UrlNode*) – Urlnode ptr
- **layout** (*ForeignKey to PageLayout*) – Layout
- **placeholder_set** (*PlaceholderRelation*) – Placeholder set
- **contentitem_set** (*ContentItemRelation*) – Contentitem set

The `FluentPage` class

```
class fluent_pages.pagetypes.fluentpage.models.FluentPage (*args, **kwargs)
    A `FluentPage` represents one HTML page of the site.
```

Parameters

- **id** (*AutoField*) – Id
- **polymorphic_ctype** (*ForeignKey to ContentType*) – Polymorphic ctype

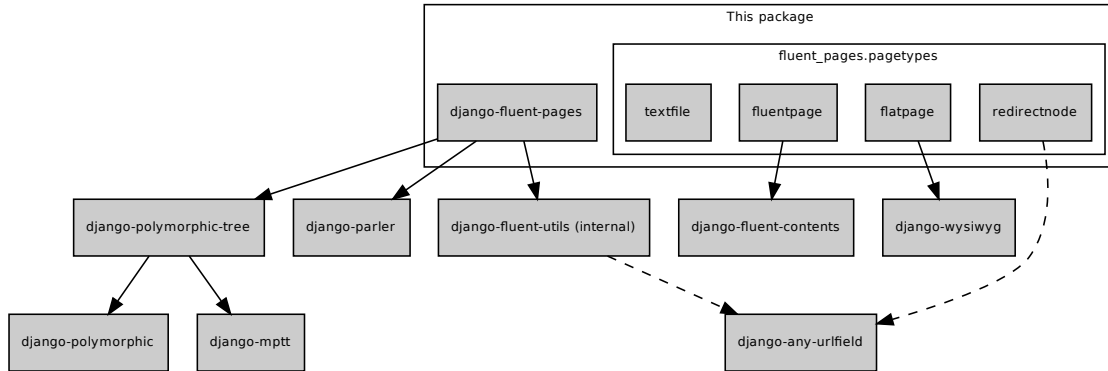
- **parent** (*PageTreeForeignKey* to *UrlNode*) – Parent. You can also change the parent by dragging the page in the list.
- **parent_site** (*ForeignKey* to *Site*) – Parent site
- **status** (*CharField*) – Status
- **publication_date** (*DateTimeField*) – Publication date. When the page should go live, status must be “Published”.
- **publication_end_date** (*DateTimeField*) – Publication end date
- **in_navigation** (*BooleanField*) – Show in navigation
- **in_sitemaps** (*BooleanField*) – Include in search engine sitemaps
- **key** (*SlugField*) – Page identifier. A unique identifier that is used for linking to this page.
- **author** (*ForeignKey* to *User*) – Author
- **creation_date** (*DateTimeField*) – Creation date
- **modification_date** (*DateTimeField*) – Last modification
- **lft** (*PositiveIntegerField*) – Lft
- **rght** (*PositiveIntegerField*) – Rght
- **tree_id** (*PositiveIntegerField*) – Tree id
- **level** (*PositiveIntegerField*) – Level
- **urlnode_ptr** (*OneToOneField* to *UrlNode*) – Urlnode ptr
- **layout** (*ForeignKey* to *PageLayout*) – Layout
- **placeholder_set** (*PlaceholderRelation*) – Placeholder set
- **contentitem_set** (*ContentItemRelation*) – Contentitem set

exception `DoesNotExist`

exception `MultipleObjectsReturned`

4.3 Package dependencies

This is a quick overview of all used Django packages:



The used packages are:

django-any-urlfield: An URL field which can also point to an internal Django model.

django-fluent-contents: The widget engine for flexible block positions.

django-fluent-utils: Internal utilities for code sharing between django-fluent modules.

django-mptt: The structure to store tree data in the database.

Note that *django-fluent-pages* doesn't use a 100% pure MPTT tree, as it also stores a `parent_id` and `_cached_url` field in the database. These fields are added for performance reasons, to quickly resolve parents, children and pages by URL.

django-parler: Translation support for all models.

django-polymorphic: Polymorphic inheritance for Django models, it lets queries return the derived models by default.

django-polymorphic-tree The tree logic, where each node can be a different model type.

django-wysiwyg: A flexible WYSIWYG field, which supports various editors.

4.4 Changelog

4.4.1 Changes in 2.0.7 (2020-01-04)

- Fix Django 3.0 compatibility by removing `django.utils.six` dependency.
- Bump setup requirements to ensure Django 3.0 compatibility.

4.4.2 Version 2.0.6 (2019-06-11)

- Confirmed Django 2.2 support.
- Fixed recursion in `.only()` queries, also happened with `django-mptt` 0.10 on deleting nodes.
- Fixed pagetype plugin admin import errors, caused by ordering in `INSTALLED_APPS`.
- Reformatted all files with `black`

4.4.3 Version 2.0.5 (2018-08-28)

- Confirmed Django 2.1 support.
- Fixed `RedirectNode` in combination with `django-any-urlfield` on Python 3.
- Bumped dependency versions to their latest bugfix releases.
- Optimized the query logic to see whether untranslated fields should be shown.

4.4.4 Version 2.0.4 (2018-04-05)

- Added `page.is_publication_date_active()` method to check whether a page is published at a given date.
- Added `Page.objects.get_for_id()` that limits the choices to the current site.
- Fixed `page.is_draft` and `page.is_published` properties to take `publication_date / publication_end_date` into account.
- Fixed displayed `verbose_name` for `FluentPage`, which showed “fluent page” in `django-staff-toolbar`.
- Improved page chooser form message when selecting an unpublished page.
- Bumped minimal `django-slug-preview` version for proper Django 2.0 support.

4.4.5 Version 2.0.3 (2018-02-05)

- Added missing migration for the new `on_delete=SET_NULL` behavior for the author field.
- Added `Meta.manager_inheritance_from_future = True` to all page 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 when using a `subclass.objects.all()`.

4.4.6 Version 2.0.2 (2018-01-22)

- Fixed adding pages when visiting the direct child admin URL (e.g. `/admin/redirectnode/redirectnode/add/` instead of `/admin/fluent_pages/page/add/?ct_id=`) [backported to 1.1.4].

4.4.7 Version 2.0.1 (2018-01-22)

- Fix admin list crash.
- Fixed setup classifiers.

4.4.8 Version 2.0 (2018-01-22)

- Added Django 2.0 support.
- Fixed fetching page layouts on `django-polymorphic` 1.3 / 2.0 [backported to 1.1.4].
- Dropped Django 1.7, 1.8 and 1.9 support, since `django-polymorphic-tree` also dropped this.

4.4.9 Version 1.1.4 (2018-01-18)

Backported fixes from 2.0 release in case Django 1.8 support is still needed.

- Fixed fetching page layouts on `django-polymorphic` 1.3 / 2.0.
- Fixed adding pages when visiting the direct child admin URL (e.g. `/admin/redirectnode/redirectnode/add/` instead of `/admin/fluent_pages/page/add/?ct_id=`)

4.4.10 Version 1.1.3 (2017-11-22)

- Added `HtmlPage.meta_image` field to specify an Facebook `og:image` for the current page
- Fixed meta keywords/description showing `None` when some fields were left empty.
- Fixed compatibility with upcoming `django-polymorphic` 2.0.
- Allow to register the model admin classes of page types directly in the admin.
- Removed `prefix_pagetypes` management command as it no never worked beyond Django 1.7.

4.4.11 Version 1.1.2 (2017-08-01)

- Added `manage.py remove_stale_pages` command that helps to clear removed page type models from the database.
- Upgraded minimal `django-polymorphic-tree` version from 1.4 to to 1.4.1, to include a required bugfix.
- Fixed unwanted migrations created by Django 1.10+
- Fixed unselected active menu item in Django 1.11

4.4.12 Version 1.1.1 (2017-02-24)

- Fixed `native_str` usage in the admin template resolving.
- Fixed more Django 1.10 issues with reverted default managers in abstract models. This also fixes the `django-fluent-blogs` admin page for the `BlogPage` model on Django 1.10.

4.4.13 Version 1.1 (2017-02-18)

- Added `child_types` and `can_be_root` options to limit the allowed model types in the plugin. This allows limiting which child types can be used by plugins!
- Added support for `{% appurl .. as varname %}`.
- Added `ParentTranslationDoesNotExist` exception to improve error handling
- Fixed Django 1.10 issue for the `FluentPage` type with an invalid default manager in the admin.
- Fix multiple fallback languages support in `rebuild_page_tree`.
- Fixed migration string types for Python 3.
- Fixed using `os.path.sep` in `FLUENT_PAGES_TEMPLATE_DIR`
- Fixed recursion bug in `RedirectNodeAdmin`.
- Dropped Python 2.6 and Django 1.6 support

Note: Creating child nodes in a language that doesn't yet exist for the parent node is no longer supported.

While past versions tried to resolve such situation with fallback URLs, it turns out to be very prone to bugs when moving page branches or changing the translated parent slug slugs.

4.4.14 Version 1.0.1 (2016-08-07)

- Fixed bug that broke Django 1.7 support.
- Avoid installing `django-mptt` 0.8.5, which breaks pickling deferred queriesets.

4.4.15 Version 1.0 (2016-08-07)

This release provides compatibility with newer package versions. Many fixes add to the stability of this release, especially when extending it with custom page types.

Major features:

- Django 1.9 support.
- Django 1.10 support is underway (it awaits fixes in our dependencies)
- Support for multiple fallback languages.
- Nicer slug previews in the admin.
- Menu template improvements:
 - Added `is_child_active` variable to fix menu highlights.
 - Added `draft` and `active` CSS classes.
 - The `fluent_pages.pagetypes.textfile` content can be translated.
 - Old unmaintained languages can be redirected with the `make_language_redirects` command.
- Dropped Django 1.4, 1.5 and Python 3.2 support.
- **Backwards incompatible:** The `FluentPageBase` class is now removed, use `AbstractFluentPage` instead.

Note: Make sure to add the `slug_preview` package to your `INSTALLED_APPS`.

`django-mptt` 0.8.5 has a bug that prevents pickling deferred queriesets, hence this version is explicitly excluded as requirement. Use version 0.8.4 instead.

4.4.16 Changes in 1.0b3 (2016-05-17)

- Dropped Django 1.5 support.
- Fixed displaying new empty translation page.
- Fixed page moving bug due to old caches on previous errors.

Changes in 1.0b3 (2016-05-17)

- Fixed showing “View on site” link for draft pages, since staff has access to it.
- Fixed `node.is_child_active` for selected parent menu’s.
- Fixed applying `FLUENT_PAGES_FILTER_SITE_ID` setting in the admin.
- Improved `RobotsTxtView` to handle `il8n_patterns()` automatically.

Changes as of 1.0b2 (2016-02-23)

- Fixed published admin icon for Django 1.9
- Fixed truncating long `db_table` names.
- Added `class="active"` in the default menu template for menu’s where a child item is active.
- Added automatic configuration for `django-staff-toolbar`.

Changes as of version 1.0b1 (2015-12-30)

- Added Django 1.9 support
- Added translation support to the `fluent_pages.pagetypes.textfile` type, to translate the content (but not the type).
- Added `draft` CSS class to unpublished menu items that are only visible for staff members.
- Added `FluentPagesConfig` to use Django 1.7 `appconfigs`.
- Added multiple fallback language support for `django-parler` 1.5.
- Added `make_language_redirects` management command for redirecting an unmaintained language to another.
- Added `is_child_active` variable in `PageNavigationNode` for menu templates.
- Added `django-slug-preview` for nicer slug appearance in the admin.
- Improve error messages when URLs can’t be created.
- Improve performance of `PageSitemap` for sites with a lot of pages.
- Temporary fix: Block moving pages to untranslated sub nodes, until a design decision can be made how to handle this.
- Temporary fix: Hide subpages when searching in the admin, to avoid errors with partial MPTT trees.
- Fixed Django 1.8 issues in the “Change Page” view.
- Fixed migrations to prevent Django from creating additional ones when settings change.
- Fixed silent behavior of using `.parent_site()` too late in an already filtered queryset.
- Fixed unicode handling in `rebuild_page_tree`.
- Fixed importing `mixed_reverse_lazy()` from django settings.
- Fixed showing pages when there is no translation is created yet.
- Fixed JavaScript event binding for dynamic related-lookup fields.
- Fixed `welcome.json` fixture

- Dropped Django 1.4 and Python 3.2 support.
- **Backwards incompatible:** The `FluentPageBase` class is now removed, use `AbstractFluentPage` instead.

4.4.17 Version 0.9 (2015-04-13)

- Added Django 1.8 support
- Non-published pages can now be seen by staff members
- Fix initial migrations on MySQL with InnoDB/utf8 charset.
- Fix missing `robots.txt` in the PyPI package.
- Fix behavior of `Page.objects.language(..).get_for_path()` and `best_match_for_path()`, use the currently selected language. This is similar to `django-parler`'s `TranslatableModel.objects.language(..).create(..)` support.
- Fix skipping mount-points in `app_reverse()` when the root is not translated.
- **Backwards incompatible** with previous beta releases: split the `fluent_pages.integration.fluent_contents` package. You'll need to import from the `.models.`, `.admin` and `.page_type_plugins` explicitly. This removes many cases where projects suffered from circular import errors.

Released in 0.9c1 (2015-01-19)

- Fix deleting pages which have SEO fields filled in (the `HtmlPageTranslation` model).
- Fix `UrlNode.DoesNotExist` exception when using `{% render_breadcrumb %}` on 404 pages.
- Change slug size to 100 characters.
- Added `RobotsTxtView` for easier sitemaps integration
- Added `FluentContentsPage.create_placeholder(slot)` API.
- Added `--mptt-only` option to `manage.py rebuild_page_tree` command.
- Added lazy-resolver functions: `app_reverse_lazy()` / `mixed_reverse_lazy()`.

Released in 0.9b4 (2014-11-06)

- Fix South migrations for flexible `AUTH_USER_MODEL`

Released in 0.9b3 (2014-11-06)

- Added preliminary Django 1.7 support, migrations are not fully working yet.
- Added translation support for the SEO fields (meta keywords/description/title) and redirect URL.
- All base models are proxy models now; there will be no more need to update south migrations in your own apps.
- Added `fluent_pages.integration.fluent_contents` to simplify creating custom
- Added `CurrentPageMixin` and `CurrentPageTemplateMixin` for custom views.
- Added `HtmlPage.meta_robots` property to automatically add `noindex` to pages outside the sitemaps.

- Added `in_sitemaps` flag, which is now `false` for the `RedirectNode` by default. pagetypes that reuse the `django-fluent-contents` integration that the `fluent_pages.pagetypes.fluentpage` has.
- Fixed stale translated `ContentItem` objects from `django-fluent-contents` when deleting a translation of a page.
- Fixed support for: `future` `>= 0.13`.
- Fixed support for: `django-polymorphic` `>= 0.6`.
- Fixed support for: `django-parler` `>= 1.2`.
- API: use `FluentContentsPage` instead of `AbstractFluentPage`.

Upgrade notices:

Due to Django 1.7 support, the following changes had to be made:

- `fluent_pages.admin` is renamed to `fluent_pages.adminui`.
- South 1.0 is now required to run the migrations (or set `SOUTH_MIGRATION_MODULES` for all plugins).

Secondly, there were database changes to making the SEO-fields translatable. Previously, the SEO fields were provided by abstract models, requiring projects to upgrade their apps too.

All translated SEO fields are now managed in a single table, which is under the control of this app. Fortunately, this solves any future migration issues for changes in the `HtmlPage` model.

If your page types inherited from `HtmlPage`, `FluentContentsPage` or its old name `FluentPage`, you'll have to migrate the data of your apps one more time. The bundled pagetypes have two migrations for this: `move_seo_fields` and `remove_untranslated_fields`. The first migration moves all data to the `HtmlPageTranslation` table (manually added to the datamigration). The second migration can simply be generated with `./manage.py schemamigration <yourapp> --auto "remove_untranslated_fields"`.

If you have overridden `save_translation()` in your models, make sure to check for `translation.related_name`, as both the base object and derived object translations are passed through this method now.

The `SeoPageMixin` from 0.9b1 was removed too, instead inherit directly from `HtmlPage`.

Released in 0.9b2 (2014-06-28)

- Added Python 3 support!
- Added `key` field to allow linking to specific user-created pages (e.g. a Terms and Conditions page). This feature is only visible when `FLUENT_PAGES_KEY_CHOICES` is configured.
- Fix support for `il8n_patterns()` in the `override_url` field.
- Added `hide_untranslated_menu_items` setting in `FLUENT_PAGES_LANGUAGES / PARLER_LANGUAGES`.
- Added page variable for menu items in `PageNavigationNode`.
- Add “change Override URL permission” flag. South users: run `manage.py syncdb --all` to create the permission
- Fix resolving pages under their fallback language URL when a translated URL does exist.
- Fix exception in `PageNavigationNode.has_children`.
- Fix moving pages in the admin list (changes were undone).

- Fix missing “ct_id” GET parameter for Django 1.6 when filtering in the admin (due to the `_changelist_filters` parameter).
- Updated dependencies to their Python 3 compatible versions.
- Optimize queries for rendering menu’s
- nodes without children no need a query in `PageNavigationNode.children`.
- avoid polymorphic behavior for child menu nodes (unless the parent node was polymorphic).

Released in 0.9b1 (2014-04-14)

- Added multisite support.
- Added multilingual support, using `django-parler`.
- Added hooks for patching the admin; `FLUENT_PAGES_PARENT_ADMIN_MIXIN` and `FLUENT_PAGES_CHILD_ADMIN_MIXIN`. Note that using this feature is comparable to monkey-patching, and future compatibility can’t be fully guanteed.
- Added “Can change Shared fields” permission for all page types.
- Added “Can change Page layout” permission for `fluent_pages.pagetypes.fluentpage`.
- Allow `formfield_overrides` to contain field names too.
- API: added `SeoPageMixin` model with `meta_title`, `meta_keywords` and `meta_description` fields.
- API: renamed `FluentPageBase` to `AbstractFluentPage`.
- API: added `get_view_response` to the `PageTypePlugin` class, allow adding middleware to custom views.
- API: **Backwards incompatible:** when inheriting from the abstract `HtmlPage` model, your app needs a South migration.
- Fixed calling `reverse()` on the resolved page urls.
- Dropped Django 1.3 and 1.4 support.

Upgrade notices:

- When using custom page types that inherit from inherited from `HtmlPage`, `FluentPageBase` or `FluentContentsPage`, please add a South migration to your application to handle the updated fields.
- The `keywords` field was renamed to `meta_keywords`.
- The `description` field was renamed to `meta_description`.
- The `meta_title` field was added.
- The South `rename_column` function can be used in the migration:

```
db.rename_column('your_model_table', 'keywords', 'meta_keywords')
db.rename_column('your_model_table', 'description', 'meta_description')
```

- API: renamed `FluentPageBase` to `FluentContentsPage`. The old name is still available.

4.4.18 Version 0.8.7 (2014-12-30)

- Add support of django-polymorphic 0.6.
- Add `page` variable for menu items in `PageNavigationNode`.

4.4.19 Version 0.8.6 (2014-01-21)

- Add `FLUENT_PAGES_DEFAULT_IN_NAVIGATION` setting to change the “in navigation” default value.
- Fix django-mptt 0.6 support.
- Fix using `{% appurl %}` for modules with multiple results.
- Widen “modification date” column, to support other languages.

4.4.20 Version 0.8.5 (2013-08-15)

- Added intro page for empty sites.
- Support Django 1.6 transaction management.
- Fix NL translation of “Slug”.
- Fix the `@admin` redirect for application URLs (e.g. `/page/app-url/@admin` should redirect to `/page/app-url/`).
- Fix URL dispatcher for app urls when a URL prefix is used (e.g. `/en/..`)
- Fix Django 1.5 custom user model support in migrations

4.4.21 Version 0.8.4 (2013-05-28)

- Fix running at Django 1.6 alpha 1
- Remove filtering pages by `SITE_ID` in `PageChoiceField` as there is no proper multi-site support yet.
- Remove `X-Object-Type` and `X-Object-Id` headers as Django 1.6 removed it due to caching issues.

4.4.22 Version 0.8.3 (2013-05-15)

- Fix circular imports for some setups that import `fluent_pages.urlresolvers` early.
- Fix initial south migrations, added missing dependencies.
- Fix using `{% render_menu %}` at 404 pages.

4.4.23 Version 0.8.2 (2013-04-25)

- Add `parent` argument to `{% render_menu %}`, to render sub menu's.
- Add `page`, `site` variable in template of `{% render_breadcrumb %}`.
- Add `request`, `parent` (the parent context) variables to templates of `{% render_breadcrumb %}` and `{% render_menu %}`.
- Bump version requirement of `django-mptt` to 0.5.4, earlier versions have bugs.

- Fix `{% get_fluent_page_vars %}` to skip the `django-haystack` `page` variable.
- Fix `{% get_fluent_page_vars %}` when a `site` variable is already present.
- Fix unit test suite in Django 1.3

4.4.24 Version in 0.8.1 (2013-03-07)

- Add “Flat page” page type.
- Add support for `django-any-urlfield`.
- Add `X-Object-Type` and `X-Object-Id` headers to the response in development mode (similar to `django.contrib.flatpages`).
- Add Django 1.5 Custom User model support.
- Added lots of documentation.
- Moved the template tag parsing to a separate package, `django-tag-parser`.
- Improve error messages on initial project setup.
- Improve ability to extend the `page_change_form` template.
- Improve layout of `keywords` and `description` fields in the admin.
- Fixed 500 error on invalid URLs with unicode characters.
- Fixed `app_reverse()` function for Django 1.3.
- Fixed `appurl` tag for template contexts without `page` variable.
- Fixed `NavigationNode.is_active` property for sub menu nodes.
- Fixed `NavigationNode.parent` property for root node.
- Fixed `runtests.py` script.
- Fixed `Page.objects.best_match_for_path()` for pages without a slash.
- Fixed generated URL path for “file” node types in sub folders.
- Fix Django dependency in `setup.py`, moved from `install_requires` to the `requires` section.
- Bump version of `django-polymorphic-tree` to 0.8.6 because it fixes issues with moving pages in the admin.

4.4.25 Version 0.8.0 (2012-11-21)

First public release

- Support for custom page types.
- Optional integration with `django-fluent-contents`.
- Refactored tree logic to `django-polymorphic-tree`.
- Unit tests included.

CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`

f

`fluent_pages.adminui`, 45
`fluent_pages.adminui.utils`, 47
`fluent_pages.extensions`, 47
`fluent_pages.integration.fluent_contents`,
50
`fluent_pages.models`, 52
`fluent_pages.models.navigation`, 57
`fluent_pages.pagetypes.fluentpage.admin`,
63
`fluent_pages.pagetypes.fluentpage.models`,
63
`fluent_pages.sitemaps`, 60
`fluent_pages.templatetags.appurl_tags`,
59
`fluent_pages.urlresolvers`, 60
`fluent_pages.views`, 61

Symbols

- `__init__()` (*fluent_pages.extensions.PageTypePlugin* method), 48
 - `__init__()` (*fluent_pages.extensions.PageTypePool* method), 49
 - `__init__()` (*fluent_pages.models.UrlNode* method), 53
 - `__init__()` (*fluent_pages.models.navigation.PageNavigationNode* method), 58
- ## A
- `AbstractFluentPage` (class in *fluent_pages.pagetypes.fluentpage.models*), 63
 - `all_allowed_plugins` (*fluent_pages.integration.fluent_contents.admin.FluentContentsPageAdmin* attribute), 51
 - `app_reverse()` (in module *fluent_pages.urlresolvers*), 60
- ## B
- `base_change_form_template` (*fluent_pages.adminui.DefaultPageChildAdmin* attribute), 46
 - `base_form` (*fluent_pages.adminui.DefaultPageChildAdmin* attribute), 46
 - `base_form` (*fluent_pages.pagetypes.fluentpage.admin.FluentPageAdmin* attribute), 63
 - `base_model` (*fluent_pages.adminui.DefaultPageChildAdmin* attribute), 46
 - `best_match_for_path()` (*fluent_pages.models.UrlNodeManager* method), 57
 - `breadcrumb` (*fluent_pages.models.UrlNode* attribute), 54
- ## C
- `can_be_root` (*fluent_pages.extensions.PageTypePlugin* attribute), 48
 - `can_be_root` (*fluent_pages.models.UrlNode* attribute), 54
 - `can_have_children` (*fluent_pages.extensions.PageTypePlugin* attribute), 48
 - `can_have_children` (*fluent_pages.models.UrlNode* attribute), 54
 - `change_form_template` (*fluent_pages.adminui.DefaultPageChildAdmin* attribute), 46
 - `child_types` (*fluent_pages.extensions.PageTypePlugin* attribute), 48
 - `child_types` (*fluent_pages.models.UrlNode* attribute), 54
 - `children` (*fluent_pages.models.navigation.NavigationNode* attribute), 58
 - `children` (*fluent_pages.models.navigation.PageNavigationNode* attribute), 59
 - `clear_app_reverse_cache()` (in module *fluent_pages.urlresolvers*), 61
 - `content_type` (*fluent_pages.views.RobotsTxtView* attribute), 62
 - `contentitem_set` (*fluent_pages.integration.fluent_contents.models.FluentContentsPageAdmin* attribute), 52
 - `create_placeholder()` (*fluent_pages.integration.fluent_contents.models.FluentContentsPageAdmin* method), 52
 - `CurrentPageMixin` (class in *fluent_pages.views*), 61
 - `CurrentPageTemplateMixin` (class in *fluent_pages.views*), 61
- ## D
- `default_in_sitemaps` (*fluent_pages.extensions.PageTypePlugin* attribute), 48
 - `default_url` (*fluent_pages.models.UrlNode* attribute), 54
 - `DefaultPageChildAdmin` (class in *fluent_pages.adminui*), 45

delete() (*fluent_pages.models.HtmlPage method*), 56
 delete() (*fluent_pages.models.UrlNode method*), 53
 django-any-urlfield:, 66
 django-fluent-contents:, 66
 django-fluent-utils:, 66
 django-mptt:, 66
 django-parler:, 66
 django-polymorphic-tree, 66
 django-polymorphic:, 66
 django-wysiwyg:, 66

F

fluent_pages.adminui (*module*), 45
 fluent_pages.adminui.utils (*module*), 47
 fluent_pages.extensions (*module*), 47
 fluent_pages.integration.fluent_contents (*module*), 50
 fluent_pages.models (*module*), 52
 fluent_pages.models.navigation (*module*), 57
 fluent_pages.pagetypes.fluentpage.admin (*module*), 63
 fluent_pages.pagetypes.fluentpage.models (*module*), 63
 fluent_pages.sitemaps (*module*), 60
 fluent_pages.templatetags.appurl_tags (*module*), 59
 fluent_pages.urlresolvers (*module*), 60
 fluent_pages.views (*module*), 61
 FluentContentsPage (*class in fluent_pages.integration.fluent_contents.models*), 51
 FluentContentsPageAdmin (*class in fluent_pages.integration.fluent_contents.admin*), 50
 FluentContentsPagePlugin (*class in fluent_pages.integration.fluent_contents.page_type_plugins*), 52
 FluentPage (*class in fluent_pages.pagetypes.fluentpage.models*), 64
 FluentPage.DoesNotExist, 65
 FluentPage.MultipleObjectsReturned, 65
 FluentPageAdmin (*class in fluent_pages.pagetypes.fluentpage.admin*), 63
 formfield_for_dbfield() (*fluent_pages.adminui.HtmlPageAdmin method*), 46
 formfield_for_foreignkey() (*fluent_pages.adminui.DefaultPageChildAdmin method*), 46
 formfield_for_foreignkey() (*fluent_pages.pagetypes.fluentpage.admin.FlulentPageAdmin method*), 63

G

get_absolute_url() (*fluent_pages.models.UrlNode method*), 53
 get_absolute_urls() (*fluent_pages.models.UrlNode method*), 53
 get_all_allowed_plugins() (*fluent_pages.integration.fluent_contents.admin.FlulentContentsPage method*), 50
 get_children() (*fluent_pages.models.navigation.NavigationNode method*), 58
 get_content_items_by_slot() (*fluent_pages.integration.fluent_contents.models.FlulentContentsPage method*), 52
 get_context() (*fluent_pages.extensions.PageTypePlugin method*), 48
 get_context_data() (*fluent_pages.views.CurrentPageMixin method*), 61
 get_current_edited_page() (*in module fluent_pages.adminui.utils*), 47
 get_current_page() (*fluent_pages.views.CurrentPageMixin method*), 61
 get_file_types() (*fluent_pages.extensions.PageTypePool method*), 49
 get_folder_types() (*fluent_pages.extensions.PageTypePool method*), 49
 get_for_key() (*fluent_pages.models.UrlNodeManager method*), 57
 get_for_path() (*fluent_pages.models.UrlNodeManager method*), 57
 get_i18n_patterns_codes() (*fluent_pages.views.RobotsTxtView method*), 62
 get_layout_view() (*fluent_pages.pagetypes.fluentpage.admin.FlulentPageAdmin method*), 63
 get_level() (*fluent_pages.models.navigation.NavigationNode method*), 58
 get_model_classes() (*fluent_pages.extensions.PageTypePool method*), 49
 get_model_instances() (*fluent_pages.extensions.PageTypePlugin method*), 48
 get_page_admin_url() (*in module fluent_pages.adminui.utils*), 47
 get_page_template() (*fluent_pages.extensions.PageTypePlugin method*), 48

ent_pages.pagetypes.fluentpage.admin.FluentPageAdmin.*AdminViewUrl* (*fluent_pages.views.CurrentPageMixin* method), 63
ent_pages.views.CurrentPageMixin method), 61
get_placeholder_by_slot () (*fluent_pages.integration.fluent_contents.models.FluentContentsPage* method), 52
H
get_placeholder_data () (*fluent_pages.integration.fluent_contents.admin.FluentContentsPageAdmin* method), 50
ent_pages.pagetypes.fluentpage.admin.FluentPageAdmin method), 63
get_placeholder_data () (*fluent_pages.models.navigation.NavigationNode* attribute), 58
ent_pages.pagetypes.fluentpage.admin.FluentPageAdmin attribute), 58
get_plugin_by_model () (*fluent_pages.extensions.PageTypePool* method), 49
fluent_pages.models.navigation.PageNavigationNode attribute), 59
has_change_page_layout_permission () (*fluent_pages.pagetypes.fluentpage.admin.FluentPageAdmin* method), 63
has_children (*fluent_pages.models.navigation.NavigationNode* attribute), 58
has_children (*fluent_pages.models.navigation.PageNavigationNode* attribute), 59
has_i18n_patterns_urls () (*fluent_pages.views.RobotsTxtView* method), 62
get_plugins () (*fluent_pages.extensions.PageTypePool* method), 49
fluent_pages.views.RobotsTxtView method), 62
get_readonly_fields () (*fluent_pages.pagetypes.fluentpage.admin.FluentPageAdmin* method), 63
fluent_pages.models.navigation.PageNavigationNode attribute), 59
get_render_template () (*fluent_pages.extensions.PageTypePlugin* method), 48
fluent_pages.pagetypes.fluentpage.admin.FluentPageAdmin method), 63
get_render_template () (*fluent_pages.integration.fluent_contents.page_type_plugins.FluentContentsPagePlugin* method), 52
fluent_pages.pagetypes.fluentpage.admin.FluentPageAdmin method), 63
get_response () (*fluent_pages.extensions.PageTypePlugin* method), 48
fluent_pages.models.UrlNodeManager method), 57
is_active (*fluent_pages.models.navigation.NavigationNode* attribute), 58
is_active (*fluent_pages.models.navigation.PageNavigationNode* attribute), 59
get_sitemap_urls () (*fluent_pages.views.RobotsTxtView* method), 62
is_child_active (*fluent_pages.models.navigation.NavigationNode* attribute), 58
is_child_active (*fluent_pages.models.navigation.PageNavigationNode* attribute), 59
get_template () (*fluent_pages.models.PageLayout* method), 56
is_draft (*fluent_pages.models.navigation.NavigationNode* attribute), 58
is_draft (*fluent_pages.models.navigation.PageNavigationNode* attribute), 59
get_template_names () (*fluent_pages.views.CurrentPageTemplateMixin* method), 61
is_draft (*fluent_pages.models.UrlNode* attribute), 54
is_file (*fluent_pages.extensions.PageTypePlugin* attribute), 48
is_file (*fluent_pages.models.UrlNode* attribute), 54
get_translation_objects () (*fluent_pages.integration.fluent_contents.admin.FluentContentsPageAdmin* method), 51
is_first_child (*fluent_pages.models.UrlNode* attribute), 54
is_last_child (*fluent_pages.models.UrlNode* attribute), 54
get_url_pattern_plugins () (*fluent_pages.extensions.PageTypePool* method), 49
is_last_child (*fluent_pages.models.UrlNode* attribute), 54
is_publication_date_active () (*fluent_pages.models.UrlNode* method), 53
is_file (*fluent_pages.extensions.PageTypePlugin* attribute), 48
is_first_child (*fluent_pages.models.UrlNode* attribute), 54
is_last_child (*fluent_pages.models.UrlNode* attribute), 54
get_urls () (*fluent_pages.pagetypes.fluentpage.admin.FluentPageAdmin* method), 63
is_published (*fluent_pages.models.navigation.NavigationNode* attribute), 58
get_view_response () (*fluent_pages.extensions.PageTypePlugin* method), 48

- is_published (*fluent_pages.models.navigation.PageNavigationNode* attribute), 59
 - is_published (*fluent_pages.models.UrlNode* attribute), 54
 - items () (*fluent_pages.sitemaps.PageSitemap* method), 60
- ## L
- last_modified (*fluent_pages.models.UrlNode* attribute), 54
 - lastmod () (*fluent_pages.sitemaps.PageSitemap* method), 60
 - level (*fluent_pages.models.navigation.NavigationNode* attribute), 58
 - location () (*fluent_pages.sitemaps.PageSitemap* method), 60
- ## M
- meta_robots (*fluent_pages.models.HtmlPage* attribute), 56
 - mixed_reverse () (in module *fluent_pages.urlresolvers*), 60
 - model (*fluent_pages.extensions.PageTypePlugin* attribute), 48
 - model (*fluent_pages.integration.fluent_contents.page_type_plugins.PlugintContentsPagePlugin* attribute), 52
 - model_admin (*fluent_pages.extensions.PageTypePlugin* attribute), 47
 - model_admin (*fluent_pages.integration.fluent_contents.page_type_plugins.PlugintContentsPagePlugin* attribute), 52
 - MultipleReverseMatch, 61
- ## N
- NavigationNode (class in *fluent_pages.models.navigation*), 58
- ## P
- Page (class in *fluent_pages.models*), 55
 - page (*fluent_pages.models.navigation.PageNavigationNode* attribute), 59
 - Page.DoesNotExist, 55
 - Page.MultipleObjectsReturned, 55
 - page_key (*fluent_pages.models.UrlNode* attribute), 54
 - page_type_pool (*fluent_pages.extensions.fluent_pages.extensions* attribute), 50
 - PageAdmin (in module *fluent_pages.adminui*), 45
 - PageAdminForm (class in *fluent_pages.adminui*), 46
 - PageLayout (class in *fluent_pages.models*), 56
 - PageLayout.DoesNotExist, 56
 - PageLayout.MultipleObjectsReturned, 56
 - PageNavigationNode (class in *fluent_pages.models.navigation*), 58
 - PageSitemap (class in *fluent_pages.sitemaps*), 60
 - PageTypeAlreadyRegistered, 50
 - PageTypeNotFound, 50
 - PageTypeNotMounted, 61
 - PageTypePlugin (class in *fluent_pages.extensions*), 47
 - PageTypePool (class in *fluent_pages.extensions*), 49
 - parent (*fluent_pages.models.navigation.NavigationNode* attribute), 58
 - parent (*fluent_pages.models.navigation.PageNavigationNode* attribute), 59
 - parent_site () (*fluent_pages.models.UrlNodeManager* method), 57
 - placeholder_layout (*fluent_pages.integration.fluent_contents.admin.FluentContentsPage* attribute), 51
 - placeholder_layout_template (*fluent_pages.integration.fluent_contents.admin.FluentContentsPage* attribute), 51
 - placeholder_set (*fluent_pages.integration.fluent_contents.models.FluentContentsPage* attribute), 52
 - plugin (*fluent_pages.models.UrlNode* attribute), 54
 - published () (*fluent_pages.models.UrlNodeManager* method), 57
- ## Q
- queryset_class (*fluent_pages.models.UrlNodeManager* attribute), 57
- ## R
- register () (*fluent_pages.extensions.PageTypePool* method), 49
 - render_change_form () (*fluent_pages.adminui.DefaultPageChildAdmin* method), 46
 - render_template (*fluent_pages.extensions.PageTypePlugin* attribute), 49
 - render_to_response () (*fluent_pages.views.RobotsTxtView* method), 62
 - response_class (*fluent_pages.extensions.PageTypePlugin* attribute), 48
 - RobotsTxtView (class in *fluent_pages.views*), 62
- ## S
- save () (*fluent_pages.models.UrlNode* method), 54
 - save_translation () (*fluent_pages.models.UrlNode* method), 54
 - slug (*fluent_pages.models.navigation.NavigationNode* attribute), 58

`sort_priority` (*fluent_pages.extensions.PageTypePlugin* attribute), 49

T

`template_name` (*fluent_pages.views.RobotsTxtView* attribute), 62

`title` (*fluent_pages.models.navigation.NavigationNode* attribute), 58

`toplevel()` (*fluent_pages.models.UrlNodeManager* method), 57

`toplevel_navigation()` (*fluent_pages.models.UrlNodeManager* method), 57

`type_id` (*fluent_pages.extensions.PageTypePlugin* attribute), 49

`type_name` (*fluent_pages.extensions.PageTypePlugin* attribute), 49

U

`url` (*fluent_pages.models.navigation.NavigationNode* attribute), 58

`url` (*fluent_pages.models.UrlNode* attribute), 54

`url_pattern_types()` (*fluent_pages.models.UrlNodeManager* method), 57

`UrlNode` (class in *fluent_pages.models*), 52

`UrlNode.DoesNotExist`, 53

`UrlNode.MultipleObjectsReturned`, 53

`UrlNodeManager` (class in *fluent_pages.models*), 57

`urls` (*fluent_pages.extensions.PageTypePlugin* attribute), 49

V

`verbose_name` (*fluent_pages.extensions.PageTypePlugin* attribute), 49