feincms3 Documentation

Release 0.30-1-gf18fe38

Feinheit AG

Mar 18, 2019

Contents

Version 0.30-1-gf18fe38

feincms3 offers tools and building blocks which make building a CMS that is versatile, powerful and tailor-made at the same time for each project a reachable reality.

It builds on other powerful tools such as Django itself and its admin interface, django-content-editor to allow creating and editing structured content and django-tree-queries for querying hierarchical data such as page trees.

The tools can be used for a page CMS, but also work well for other types of content such as news magazines or API backends for mobile apps.

Note: Despite its version number feincms3 is already used in production on many sites and backwards compatibility isn't broken lightly.

Note: This documentation uses Python 3's keyword-only syntax in a few places.

While Python 3 is strongly recommended feincms3 still supports Python 2. Keyword-only usage is enforced by function wrappers for now.

CHAPTER 1

First steps

Start here if you want to know what feincms3 is and build your first CMS based on feincms3.

1.1 Introduction

1.1.1 Philosophy

feincms3 follows the library-not-framework approach. Inversion of control is avoided as much as possible, and great care is taken to provide useful functionality which can still be easily replaced if anyone wishes to do so.

Replacing functionality should not require using extension points or configuration but simply different glue code, which should be short and obvious enough to be repeated in different projects.

The idea is not necessarily to avoid code, but to avoid all sorts of complexity, whether obvious or not. The cost of abstractions is that there always comes a moment where you have understand the layers beneath, and often the learning curve gets steep quickly.

feincms3 is your Do It Yourself kit for CMS building, as Django is your Do It Yourself kit for website building. Getting started isn't easy at first but it pays off later.

feincms3 only has abstract model classes and mixins. Any concrete classes (for example, the page model) **have** to be added by you in your own project. This is by design, and a quick win for customization.

1.1.2 Standing on the shoulders

Django's builtin admin application provides a really good and usable administration interface for managing structured content. django-content-editor extends Django's inlines mechanism with tools and an interface for managing heterogenous collections of content as are often necessary for content management systems. For example, articles may be composed of text blocks with images and videos interspersed throughout. Those content elements are called plugins.

django-tree-queries provides a smart way to efficiently fetch tree-shaped data in a relational database supporting Common Table Expressions.

Note: What we are calling plugins is called a content type in FeinCMS. This can be easily confused with Django's own contenttypes, therefore the name was changed for feincms3.

Using django-mptt or other tree libraries is possible with feincms3 as well if you don't want to use CTEs. Reimplementing the abstract page class with a diffefent library should be straightforward.

1.1.3 The parts of feincms3

feincms3 has the following main parts:

- A base class for your own **pages** model if you want to use django-tree-queries to build a hierarchical page tree.
- Model **mixins** for common tasks such as building several navigation menus from a page tree, multilingual sites and selectable templates.
- A few ready-made **plugins** for rich text, images, oEmbed and template snippets.
- A HTML sanitization and **cleansing** function and a rich text widget building on html-sanitizer and django-ckeditor.
- Facilities for embedding **apps** through the admin interface, such as any interactive content (forms) or apps with subpages (e.g. an articles app).
- A **renderer** and associated helpers and template tags.
- **admin** classes for visualizing and modifying the tree hierarchy.
- Various utilities (**shortcuts** and **template tags**).

1.2 Prerequisites and installation

feincms3 runs on Python 2.7 and Python 3.4 or better. The minimum required Django version is 1.11. Database engine support is constrained by django-tree-queries use of recursive common table expressions. At the time of writing, this means that PostgreSQL, sqlite3 (>3.8.3) and MariaDB (>10.2.2) are supported. MySQL 8.0 should work as well, but is not being tested.

The best way to install feincms3 is:

```
pip install feincms3[all]
```

This installs all optional dependencies as required by the bundled rich text, image and oEmbed plugins. A more minimal installation can be selected by only running `pip install feincms3`.

1.3 Build your CMS

This guide shows step by step how to use the tools provided by feincms3 to build your own CMS.

Note: If you just want to quickly check out what feincms3 is capable of, have a look at the feincms3-example project. It shows how everything works together, but also uses advanced functionality which might be confusing to newcomers and is not necessary for smaller CMS projects.

1.3.1 Getting started

Install feincms3 and all recommended dependencies:

```
pip install feincms3[all]
```

Add the following settings:

```
INSTALLED_APPS = [
    ...
    "feincms3",
    "content_editor",
     # Optional, but not for this guide:
    "ckeditor",
    "imagefield",
]
```

1.3.2 Configure the rich text editor

The bundled rich text plugin (which we're going to integrate) uses *feincms3.cleanse.CleansedRichTextField* which always sends HTML through html-sanitizer. The default configuration of HTML sanitizer is really restrictive and removes images (besides other things such as normalizing the HTML and removing script tags etc.)

The corresponding django-ckeditor configuration follows. It should also be added to your settings:

```
# Configure django-ckeditor
CKEDITOR_CONFIGS = {
    "default": {
        "toolbar": "Custom",
        "format_tags": "h1;h2;h3;p;pre",
        "toolbar_Custom": [[
            "Format", "RemoveFormat", "-",
            "Bold", "Italic", "Subscript", "Superscript", "-",
            "NumberedList", "BulletedList", "-",
            "Anchor", "Link", "Unlink", "-",
            "HorizontalRule", "SpecialChar", "-",
            "Source",
        ]],
    },
}
CKEDITOR_CONFIGS["richtext-plugin"] = CKEDITOR_CONFIGS["default"]
```

Note: HTML copy-pasted from other sources (e.g. Word) is often messy. It is generally a good idea to sanitize HTML on the server side to prevent XSS attacks or even just the general uglyness that results from giving website editors too much freedom.

We almost never allow embedding images, tables etc. into rich text elements on our sites. It is just too easy to add a 10MB JPEG or even a BMP file and scale it down to 50x50. Adding images as a separate plugin has other benefits too: No parsing of rich texts to replace images, it's much easier to e.g. create a lightbox, use the first image on the site as teaser image or whatever comes to your mind.

That being said, adding your own rich text plugin which allows whatever you want is quite straightforward and completely supported.

1.3.3 Models

The page model and a few plugins could be defined as follows:

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

from content_editor.models import Region, create_plugin_base

from feincms3 import plugins
from feincms3.pages import AbstractPage


class Page(AbstractPage):
    regions = [
        Region(key="main", title=_("Main")),
    ]


PagePlugin = create_plugin_base(Page)


class RichText(plugins.richtext.RichText, PagePlugin):
    pass


class Image(plugins.image.Image, PagePlugin):
    caption = models.CharField(_("caption"), max_length=200, blank=True)
```

1.3.4 Views and URLs

You're completely free to define your own views and URLs. That being said, the `AbstractPage` class already has a `get_absolute_url` implementation which expects something like this:

```
from django.conf.urls import url

from app.pages import views


app_name = "pages"
urlpatterns = [
    url(r"^(?P<path>[-\w/]+)/$", views.page_detail, name="page"),
    url(r"^$", views.page_detail, name="root"),
]
```

If you don't like this, you're completely free to write your own views, URLs and `get_absolute_url` method.

With the URLconf above the view in the `app.pages.views` module would look as follows:

```
from django.shortcuts import get_object_or_404, render

from feincms3.regions import Regions

from .models import Page
from .renderer import renderer
```

(continues on next page)

(continued from previous page)

```
def page_detail(request, path=None):
    page = get_object_or_404(
        Page.objects.active(),
        path="/{}/".format(path) if path else "/",
    )
    return render(request, "pages/standard.html", {
        "page": page,
        "regions": Regions.from_item(
            page, renderer=renderer, timeout=60
        ),
    })
```

Note: FeinCMS provided request and response processors and several ways how plugins (in FeinCMS: content types) could hook into the request-response processing. This isn't necessary with feincms3 – simply put the functionality into your own views code.

1.3.5 Rendering and templates

Here's an example how plugins could be rendered, `app.pages.renderer`:

```
from django.utils.html import format_html, mark_safe

from feincms3.renderer import TemplatePluginRenderer

from .models import Page, RichText, Image


renderer = TemplatePluginRenderer()
renderer.register_string_renderer(
    RichText,
    lambda plugin: mark_safe(plugin.text),
)
renderer.register_string_renderer(
    Image,
    lambda plugin: format_html(
        '<figure><img src="{}" alt=""/><figcaption>{}</figcaption></figure>',
        plugin.image.url,
        plugin.caption,
    ),
)
```

Of course if you'd rather let plugins use templates, do this:

```
renderer.register_template_renderer(
    Image,
    "plugins/image.html",
)
```

And the associated template:

```
<figure>
  <img src="{{ plugin.image.url }}" alt="{{ plugin.caption }}"/>
```

(continues on next page)

(continued from previous page)

```
  {% if plugin.caption %}<figcaption>{{ plugin.caption }}</figcaption>{% endif %}
</figure>
```

The default image field also offers built-in support for thumbnailing and cropping with a PPOI (primary point of interest); have a look at the django-imagefield docs to find out how.

And a `pages/standard.html` template:

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

{% load feincms3 %}

{% block title %}{{ page.title }} – {{ block.super }}{% endblock %}

{% block content %}
  <main>
    <h1>{{ page.title }}</h1>
    {% render_region regions "main" %}
  </main>
{% endblock %}
```

1.3.6 Admin classes

Here's an example how the `app.pages.admin` module might look like:

```
from django.contrib import admin

from content_editor.admin import ContentEditor
from feincms3 import plugins
from feincms3.admin import TreeAdmin

from app.pages import models


class PageAdmin(ContentEditor, TreeAdmin):
    list_display = ["indented_title", "move_column", "is_active"]
    prepopulated_fields = {"slug": ("title",)}
    raw_id_fields = ["parent"]

    inlines = [
        plugins.richtext.RichTextInline.create(models.RichText),
        plugins.image.ImageInline.create(models.Image),
    ]

    # fieldsets = ... (Recommended! No example here though. Note
    # that the content editor not only allows collapsed, but also
    # tabbed fieldsets -- simply add 'tabbed' to the 'classes' key
    # the same way you'd add 'collapse'.

    # class Media: ... (Add font-awesome from a CDN and nicely
    # looking buttons for plugins as is described in
    # django-content-editor's documentation -- search for
    # "plugin_buttons.js")


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

CHAPTER 2

Guides

These guides discuss key concepts and show how to approach common tasks. They do not have to be read in order and in general only build on the knowledge imparted in *Build your CMS*.

2.1 Writing plugins

Plugins for a given model extend an abstract base class created by django-content-editor's `create_plugin_base` function. A minimal example follows here:

```
from content_editor import Region, create_plugin_base

class Article(models.Model):
    regions = [Region(...)]
    title = models.CharField(...)

ArticlePlugin = create_plugin_base(Article)

class Text(ArticlePlugin):
    text = models.TextField()

class Download(ArticlePlugin):
    download = models.FileField(upload_to="downloads/")
    caption = models.CharField(blank=True, max_length=200)
```

The `create_plugin_base` creates an abstract model with the following fields and methods in the example above:

- `parent`: A foreign key to `Article`.
- `region`: A char field ready for holding the region's key it belongs to.
- `ordering`: An integer field which orders the list of content elements. The value of the ordering field should be treated as opaque in that you should not depend on exact values and gaps in the ordering field values.

- `get_queryset`: A classmethod without arguments which is used to fetch a queryset of plugin instances. If you have a plugin with a foreign key (not to `parent` but to other instances) it would probably be a really good idea to override this classmethod with one that adds a `select_related()` call.

Note: FeinCMS 1's `create_content_type` method could not be avoided because it added the dynamically created (concrete!) model to a few lists for bookkeeping.

By contrast using `create_plugin_base` is not strictly necessary. However, django-content-editor and by extension feincms3 assume a few properties which you'd have to replicate by hand such as the model fields, the `related_name` pattern etc.

2.2 Templates and regions

The build-your-CMS guide only used one region and one template. However, this isn't sufficient for many sites. Some sites have a moodboard region and maybe a sidebar region; some sites at least have a different layout on the home page and so on.

2.2.1 More regions

django-content-editor requires a `regions` attribute or property on the model containing a list of `Region` instances. The *Build your CMS* guide presented a page model with only a single region, `"main"`. It is of course possible to specify more regions:

```
class Page(AbstractPage):
    regions = [
        Region(key="main", title=_("Main")),
        Region(key="sidebar", title=_("Sidebar"), inherited=True),
    ]
```

Regions may also be marked as `inherited`. This means that pages deeper down in the tree may inherit content from some other page (normally the page's ancestors) in case the page region itself does not define any content.

The `page_detail` view presented in the guide also works with more than one region. However, for region inheritance to work you have to provide the pages whose content should be inherited yourself. There isn't much to do though, just add the `inherit_from` keyword argument to the `Regions.from_item` factory method:

```
from feincms3.regions import Regions

def page_detail(request, path=None):
    page = ...
    return render(request, "pages/standard.html", {
        "page": page,
        "regions": Regions.from_item(
            page,
            renderer=renderer,
            inherit_from=page.ancestors().reverse(),
        ),
    })
```

`page.ancestors().reverse()` returns ancestors ordered from the page's parent to the root of the tree. We want pages to inherit content from their closest possible ancestors.

2.2.2 Making templates selectable

As written in the introduction above, sometimes a single template or layout isn't enough. Enter the *TemplateMixin*:

```
from django.utils.translation import ugettext_lazy as _
from content_editor.models import Template, Region
from feincms3.mixins import TemplateMixin
from feincms3.pages import AbstractPage

class Page(TemplateMixin, AbstractPage):
    TEMPLATES = [
        Template(
            key="standard",
            title=_("standard"),
            template_name="pages/standard.html",
            regions=(
                Region(key="main", title=_("Main")),
            ),
        ),
        Template(
            key="with-sidebar",
            title=_("with sidebar"),
            template_name="pages/with-sidebar.html",
            regions=(
                Region(key="main", title=_("Main")),
                Region(key="sidebar", title=_("Sidebar"), inherited=True),
            ),
        ),
    ]
```

The `regions` attribute is provided by the `TemplateMixin` and must be removed from the `Page` definition. Additionally, the `TemplateMixin` provides a `template` property returning the currently selected template. Instead of hard-coding the template value we should now change the `page_detail` view to render the selected template, `page.template.template_name`:

```
def page_detail(request, path=None):
    page = ...
    return render(request, page.template.template_name, {
        "page": page,
        "regions": Regions.from_item(
            page,
            renderer=renderer,
            inherit_from=page.ancestors().reverse(),
        ),
    })
```

2.3 Redirects

The *feincms3.mixins.RedirectMixin* allows redirecting pages to other pages or arbitrary URLs. Inheriting the mixin adds two fields, the char field `redirect_to_url` and the self-referencing foreign key `redirect_to_page`:

```
from feincms3.mixins import RedirectMixin
from feincms3.pages import AbstractPage
```

(continues on next page)

(continued from previous page)

```
class Page(RedirectMixin, AbstractPage):
    pass
```

At most one of `redirect_to_url` or `redirect_to_page` may be set, never both at the same time. The actual redirecting is not provided. This has to be implemented in the page view:

```
from django.shortcuts import redirect

def page_detail(request, path):
    page = ...
    if page.redirect_to_url or page.redirect_to_page:
        return redirect(page.redirect_to_url or page.redirect_to_page)
    # Default rendering continues here.
```

2.4 Navigation generation recipes

This guide provides a few examples and snippets for generating the navigation for a site dynamically.

feincms3's *AbstractPage* model inherits the methods of `tree_queries.models.TreeNode`, notably `page.ancestors()` and `page.descendants()`. These methods and the attributes added by django-tree-queries, `tree_path` and `tree_depth` will prove useful for generating navigations.

feincms3 does not have a concrete page model and does not provide any tags to fetch page instances from the template. You'll have to provide the context variables yourself, preferrably by writing your own template tags.

This guide assumes that the concrete page model is available at `app.pages.models.Page`, and that you'll add a `menus` template tag library somewhere where it may be used by Django.

2.4.1 A simple main menu

Add the following template tag to the `menus.py` file:

```
from django import template
from app.pages.models import Page

register = template.Library()

@register.simple_tag
def main_menu():
    return Page.objects.with_tree_fields().active().filter(parent=None)
```

The template (where the `page` variable holds the current page):

```
{% load menus %}
{% main_menu as menu %}
<nav class="main-menu">
{% for p in menu %}
  <a
    {% if p.id in page.tree_path %}class="active"{% endif %}
    href="{{ p.get_absolute_url }}">{{ p.title }}</a>
{% endfor %}
</nav>
```

The `tree_path` attribute contains the list of all ancestors' primary keys including the key of the node itself. The `active` class is added to all menu entries that are either an ancestor of the current page or are the current page itself.

2.4.2 Breadcrumbs

Breadcrumbs for the current page are generated easily, without the help of a template tag. Ancestors are returned starting from the root node (again, the `page` variable holds the current page):

```
<nav class="breadcrumbs">
{% for ancestor in page.ancestors %}
  {% if forloop.last %}
    {{ ancestor.title }}
  {% else %}
    <a href="{{ ancestor.get_absolute_url }}">{{ ancestor.title }}</a>
    &gt;
  {% endif %}
{% endfor %}
</nav>
```

2.4.3 Main menu with two levels and meta navigation

For this example, the page class should also inherit the *feincms3.mixins.MenuMixin* with a `MENUS` value as follows:

```
from django.utils.translation import ugettext_lazy as _
from feincms3.mixins import MenuMixin
from feincms3.pages import AbstractPage

class Page(MenuMixin, ..., AbstractPage):
    MENUS = (
        ('main', _('main navigation')),
        ('meta', _('meta navigation')),
    )
```

Let's write a template tag which returns all required nodes at once:

```
from collections import defaultdict
from django import template
from app.pages.models import Page

register = template.Library()

@register.simple_tag
def all_menus():
    menus = defaultdict(list)
    pages = Page.objects.with_tree_fields().active().exclude(
        menu=""
    ).extra(
        where=["tree_depth<=1"]
    )
    for page in pages:
        menus[page.menu].append(page)
    return menus
```

The template tag removes all pages that aren't added to a menu and filters for the first two levels in the tree. `tree_depth` is only available as an `.extra()` field, so you cannot use `.filter()` to do this.

Next, let's add a template filter which returns parents bundled together with their children:

```
@register.filter
def group_by_tree(iterable):
    parent = None
    children = []
    depth = -1

    for element in iterable:
        if parent is None or element.tree_depth == depth:
            if parent:
                yield parent, children
                parent = None
                children = []

            parent = element
            depth = element.tree_depth
        else:
            children.append(element)

    if parent:
        yield parent, children
```

Now, a possible use of those two tags in the template looks as follows:

```
{% load menus %}
{% all_menus as menus %}

<nav class="nav-main">
{% for main, children in menus.main|group_by_tree %}
  <a
    {% if page and main.id in page.tree_path %}class="active"{% endif %}
    href="{{ main.get_absolute_url }}">{{ main.title }}</a>
    {% if children %}
    <nav>
      {% for child in children %}
        <a
          {% if page and child.id in page.tree_path %}class="active"{% endif %}
          href="{{ child.get_absolute_url }}">{{ child.title }}</a>
      {% endfor %}
    </nav>
  {% endif %}
{% endfor %}
</nav>

{# ... and an analogous block for the meta menu, maybe without the children loop #}
```

2.5 Rendering

The default behavior of feincms3's renderer is to concatenate the rendered result of individual plugins into one long HTML string.

That may not always be what you want. This guide also describes a few alternative methods of rendering plugins that may or may not be useful.

2.5.1 Rendering plugins

The `feincms3.renderer.TemplatePluginRenderer` offers two fundamental ways of rendering content, string renderers and template renderers. The former simply return a string, the latter work similar to `{% include %}`.

String renderers

You may register a rendering function which returns a HTML string:

```
from django.utils.html import mark_safe
from app.pages.models import RichText

renderer = TemplatePluginRenderer()
renderer.register_string_renderer(
    RichText,
    lambda plugin: mark_safe(plugin.text)
)
```

Template renderers

Or you may choose to render plugins using a template:

```
renderer.register_template_renderer(
    Image,
    "plugins/image.html",
)
```

The configured template receives the plugin instance as `"plugin"` (fittingly).

If you need more flexibility you may also pass a callable instead of a template path as `template_name`. The callable receives the plugin instance as its only argument:

```
def external_template_name(plugin):
    if "youtube" in plugin.url:
        return "plugin/youtube.html"
    elif "vimeo" in plugin.url
        return "plugin/vimeo.html"
    return "plugin/external.html"

renderer.register_template_renderer(
    External,
    external_template_name
)
```

Often, having the surrounding template context and the plugin instance available inside the template is enough. However, you might want to provide additional context variables. This can be achieved by specifying the `context` function. The function receives the plugin instance and the surrounding template context:

```
def plugin_context(plugin, context):
    return {
        "plugin": plugin,  # Recommended, but not required.
        "additional": ....
    }
```

(continues on next page)

(continued from previous page)

```
renderer.register_template_renderer(
    Plugin,
    "plugin/plugin.html",
    plugin_context,
)
```

Rendering individual plugins

Rendering individual plugin instances is possible using the `render_plugin_in_context` method. Except if you're using a non-standard `Regions` class used to encapsulate the fetching of plugins and rendering of regions you won't have to know about this method, but see below under *Rendering some plugins differently*.

Regions instances

Because fetching plugins may be expensive (at least one database query per plugin type) it makes sense to avoid fetching plugins if there is a valid cached version. The *`feincms3.regions.Regions`* which handles the specifics of rendering plugins belonging to specific regions has a factory method, `Regions.from_item`, which automatically creates a lazily evaluated `content_editor.contents.Contents` instance.

By inspecting the plugins registered with the renderer the regions instance automatically knows which plugins to load. It also supports inherited regions introduced in the *More regions* section of the *Templates and regions* guide.

Note: The regions of this `Regions` class have a different meaning than the `Region` class used to define regions for the content editor.

The former encapsulates plugin instances and their fetching and rendering (per region of course), the latter describes the region itself.

The Regions instance has one method which we'll concern ourselves with right now, `Regions.render(region)`. This method is used to render one single region. When passing a `timeout` argument to the `Regions.from_item` factory method all return values of `Regions.render(region)` are automatically cached.

Rendering regions in the template

To render regions in the template, the template first requires the `regions` instance:

```
from feincms3.regions import Regions

def page_detail(request, path=None):
    page = ...
    ...
    return render(request, ..., {
        "page": page,
        "regions": Regions.from_item(page, renderer, timeout=60),
    })
```

In the template you can now use the template tag:

```
{% load feincms3 %}

{% render_region regions "main" %}
```

Using the template tag is advantageous because it automatically provides the surrounding template context to individual plugins' templates, meaning that they could for example access the `request` instance if e.g. an API key would be different for different URLs.

Note: Caching either works for all regions in a `Regions` instance or for none at all.

2.5.2 Rendering some plugins differently

Suppose you're building a site where some plugins should go over the full width of the browser window, but most plugins are constrained inside a container. One way to solve this problem would be to make each plugin open and close its own container. That may work well. A different possibility would be to make the renderer smarter. Let's build a custom `Regions` subclass which knows how to make some plugins escape the container:

```
from django.utils.html import mark_safe

from feincms3.regions import Regions, matches


class ContainerAwareRegions(Regions):
    def handle_fullwidth(self, items, context):
        yield "</div>"  # Close the surrounding container
        while True:
            # The first item in the items deque has caused this
            # handler to be started so it can always safely be
            # consumed ...
            yield self.renderer.render_plugin_in_context(
                items.popleft(), context
            )
            # ... the test whether this handler should continue
            # should come after processing the leftmost item to
            # avoid infinite looping.
            #
            # items may be empty now or the next item might not
            # be a "full_width" plugin:
            if not items or not matches(items[0], subregions={"full_width"}):
                break
        yield '<div class="container">'  # Reopen a new container

class FullWidthPlugin(models.Model):
    subregion = "full_width"

    class Meta:
        abstract = True

# Instantiate renderer and register plugins
renderer = TemplatePluginRenderer()

# Use our new regions class, not the default
regions = ContainerAwareRegions.from_item(page, renderer=renderer)
```

2.5.3 Grouping plugins into subregions

The `Regions` class supports rendering subregions differently. Plugins may be grouped automatically by their type or by some attribute they share.

Let's make an example. Assume that we want to group adjacent teaser elements. We have several teaser plugins but they all share the same `subregion` attribute value:

```
class ArticleTeaser(PagePlugin):
    subregion = "teaser"
    article = models.ForeignKey(...)

class ProjectTeaser(PagePlugin):
    subregion = "teaser"
    project = models.ForeignKey(...)
```

Next, we have to define a regions class which knows how to handle those teasers. The name of the handler has to match the subregion attribute exactly:

```
from feincms3.regions import Regions, matches

class SmartRegions(Regions):
    def handle_teaser(self, items, context):
        # Start the teasers element:
        yield '<div class="teasers">'
        while True:
            # items is a deque, render the leftmost item:
            yield self.renderer.render_plugin_in_context(
                items.popleft(), context
            )
            if not items:
                break
            if not matches(items[0], plugins=(ArticleTeaser, ProjectTeaser)):
                break
        yield "</div>"
```

Now you'll have to use `SmartRegions.from_item()` instead of `Regions.from_item()`, and that's all there is to it.

2.5.4 Generating JSON

A different real-world example is generating JSON instead of HTML. This is possible with a custom `Regions` class too:

```
from feincms3.regions import Region, cached_render

class JSONRegions(Regions):
    @cached_render
    def render(self, region, context=None):
        return [
            dict(
                self._renderer.render_plugin_in_context(plugin),
                type=plugin.__class__.__name__,
            )
            for plugin in self.contents[region]
        ]

        # Alternatively (In this case the ``type`` key above would have to be
        # provided by the renderers themselves):
        # return list(self.generate(self.contents[region], context))
```

(continues on next page)

(continued from previous page)

```
def page_content(request, pk):
    page = get_object_or_404(Page, pk=pk)

    renderer = TemplatePluginRenderer()
    renderer.register_string_renderer(
        RichText,
        lambda plugin: {"text": plugin.text},
    )
    renderer.register_string_renderer(
        Image,
        lambda plugin: {"image": request.build_absolute_uri(plugin.image.url)},
    )

    return JsonResponse({
        "title": page.title,
        "content": Regions.from_item(page, renderer=renderer, timeout=60),
    })
```

In this particular example `register_string_renderer` is a bit of a misnomer. For string renderers, `renderer.render_plugin_in_context` returns the return value of the individual renderer as-is.

Note: A different method would have been to use lower-level methods from django-content-editor. A short example follows, however there's more left to do to reach the state of the example above such as caching:

```
from content_editor.contents import contents_for_items

renderers = {
    RichText: lambda plugin: {
        "text": plugin.text
    },
    Image: lambda plugin: {
        "image": request.build_absolute_uri(plugin.image.url)
    },
}
contents = contents_for_item(page, [RichText, Image])
data = [
    dict(
        renderers[plugin.__class__](plugin),
        type=plugin.__class__.__name__
    )
    for plugin in contents.main
]
# etc...
```

2.6 Multilingual sites

2.6.1 Making the page language selectable

Pages may come in varying languages. `LanguageMixin` helps with that. It adds a `language_code` field to the model which allows selecting the language based on `settings.LANGUAGES`. The first language is set as default:

```
from django.utils.translation import ugettext_lazy as _
from feincms3.mixins import LanguageMixin
from feincms3.pages import AbstractPage

class Page(LanguageMixin, AbstractPage):
    pass
```

2.6.2 Activating the language

The `activate_language` method is the preferred way to activate the page's language for the current request. It runs `django.utils.translation.activate` and sets `request.LANGUAGE_CODE` to the value of `django.utils.translation.get_language`, the same things Django's `LocaleMiddleware` does.

Note that `activate` may fail and `get_language` might return a different language, however that's not specific to feincms3.

```
def page_detail(request, path):
    page = ...  # MAGIC! (or maybe get_object_or_404...)
    page.activate_language(request)
    ...
```

Note: `page.activate_language` does not persist the language across requests as Django's `django.views.i18n.set_language` does. (`set_language` modifies the session and sets cookies.) That is mostly what you want though since the page's language is tied to its URL.

2.6.3 Page tree tips

I most often add a root page per language, which means that the main navigation's `tree_depth` would be `1`, not `0`. The menu template tags described in *Navigation generation recipes* would also require an additional `.filter(language_code=django.utils.translation.get_language())` statement to only return pages in the current language.

A page tree might look as follows then:

```
Home (EN)
- About us
- News

Startseite (DE)
- Über uns
- Neuigkeiten

Page d'acceuil (FR)
- A propos de nous
- Actualité
```

2.7 Meta and OpenGraph tags

The recommended way to add meta and open graph tags information to pages and other CMS objects is using feincms3-meta.

2.7.1 Installation and configuration

Install the package:

```
pip install feincms3-meta
```

Make the page model inherit the mixin:

```
from feincms3.pages import AbstractPage
from feincms3_meta.models import MetaMixin

class Page(MetaMixin, ..., AbstractPage):
    pass
```

If you define `fieldsets` on a `ModelAdmin` subclass, you may want to use the helper `MetaMixin.admin_fieldset()`, or maybe not.

Add settings (optional, but recommended):

```
META_TAGS = {
    "site_name": "My site",
    "title": "Default title",
    "description": (
        "The default description,"
        " maybe long."
    ),
    "image": "/static/app/logo.png",
    # "author": "...",
    "robots": "index,follow,noodp",
}

# Only for translations
INSTALLED_APPS.append("feincms3_meta")
```

2.7.2 Rendering

The dictionary subclass returned by `feincms3_meta.utils.meta_tags` can either be used as a dictionary, or rendered directly (its `__str__` method returns a HTML fragment):

```
from feincms3_meta.utils import meta_tags

def page_detail(request, path=None):
    page = ...
    return render(request, ..., {
        "page": page,
        "regions": ...,
        ...
        "meta_tags": meta_tags([page], request=request),
    })
```

`meta_tags` also supports overriding or removing individual tags using keyword arguments. Falsy values are discarded, `None` causes the complete removal of the tag from the dictionary.

If you want to inherit meta tags from ancestors (or from other objects) provide more than one object to the `meta_tags` function:

```
ancestors = list(page.ancestors().reverse())
tags = meta_tags([page] + ancestors, request=request)
```

2.8 Multisite setup

feincms3-sites allows running a feincms3 site on several domains, with separate page trees etc. on each (if so desired).

2.8.1 Installation and configuration

Install the package:

```
pip install feincms3-sites
```

Inherit feincms3-sites's page model instead of the default feincms3 abstract page. The only difference is that this `AbstractPage` model has an additional `site` foreign key, and path uniqueness is enforced per-site:

```
from feincms3_sites.models import AbstractPage

class Page(..., AbstractPage):
    pass
```

Add `feincms3_sites` to `INSTALLED_APPS` and run migrations afterwards:

```
INSTALLED_APPS.append("feincms3_sites")
```

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

If you're using feincms3 apps currently, replace `feincms3.apps.apps_middleware` with `feincms3_sites.middleware.apps_middleware` in your `MIDDLEWARE`. Otherwise, you may want to add `feincms3_sites.middleware.site_middleware` near the top. Both middleware functions either set `request.site` to the current `feincms3_sites.models.Site` instance or raise a `Http404` exception.

The default behavior allows matching a single host. The advanced options fieldset in the administration panel of feincms3-sites allows specifying your own regex, allowing matching several hostnames. In this case you may also want to add `feincms3_sites.middleware.redirect_to_site_middleware` after the middleware mentioned above. If you're also using the `SECURE_SSL_REDIRECT` of Django's own `SecurityMiddleware` you have to add the `redirect_to_site_middleware` *before* `SecurityMiddleware`.

It is also possible to specify a default site. In this case, when no site's regex matches, the default site is selected instead as a fallback. The code does not prevent you from setting more than one site as the default but sites are deterministically ordered so the same site will always be selected.

2.8.2 Multisite support throughout your code

Since feincms3-sites 0.6 a contextvar automatically provides the current site when inside either `site_middleware` or `apps_middleware`. The default implementation of `Page.objects.active()` filters by the current site. When you're running queries on pages outside of a middleware you'll have to use the contextvar facility yourself by running your code inside a `with feincms3_sites.middleware.set_current_site(site)` block.

2.8.3 Default languages for sites

In some configurations it may be useful to specify a default language per site. In this case you should replace `django.middleware.locale.LocaleMiddleware` with `feincms3_sites.middleware.default_language_middleware`. This middleware has to be placed after the `site_middleware` or `apps_middleware`.

CHAPTER 3

Embedding apps

feincms3 allows content managers to freely place pre-defined applications in the page tree. Examples for apps include forms, or a news app with archives, detail pages etc.

The apps documentation is meant to be read in order.

3.1 Introduction to apps

CMS plugins consist of static content. Backend code around plugins is restricted to rendering (except if you add a thing or two in your own views, of course).

However, wouldn't it be awesome if it were possible to add contact forms and even more complicated apps to the page tree through the CMS?

That's exactly what feincms3 apps are for.

Apps are defined by a list of URL patterns specific to this app. A simple contact form would probably only have a single URLconf entry (`r'^$'`), a news app would at least have two entries (the archive and the detail URL).

The activation of apps happens through a dynamically created URLconf module (probably the trickiest code in all of feincms3, *`apps_urlconf()`*). The *`apps_middleware`* assigns the module to `request.urlconf` which ensures that apps are available for resolving and URL reversing. No page code runs at all, control is directly passed to the app views.

Please note that apps do not have to take over the page where the app itself is attached. If the app does not have a URLconf entry for `r'^$'` the standard page rendering still happens. because of the recommended catch-all URLconf entry for pages at the end.

3.2 Adding a form builder app

The following example app uses form_designer to provide a forms builder integrated with the pages app described above. Apart from installing form_designer itself the following steps are necessary.

3.2.1 Extending the page model

Make the page model inherit `AppsMixin` and `LanguageMixin` and add an `APPLICATIONS` attribute to the class:

```
from feincms3.apps import AppsMixin
from feincms3.mixins import LanguageMixin
from feincms3.pages import AbstractPage

class Page(AppsMixin, LanguageMixin, ..., AbstractPage):
    # ...
    APPLICATIONS = [
        ("forms", _("forms"), {
            # Required: A module containing urlpatterns
            "urlconf": "app.forms",

            # The "form" field on the page is required when
            # selecting the forms app
            "required_fields": ("form",),

            # Not necessary, but helpful for finding a form's URL using
            # reverse_app("forms-{}".format(form.pk), "form")
            "app_instance_namespace": lambda page: "{}-{}".format(
                page.application,
                page.form_id,
            ),
        }),
        # ...
    ]
    form = models.ForeignKey(
        "form_designer.Form",
        on_delete=models.SET_NULL,
        blank=True, null=True,
        verbose_name=_("form"),
    )
```

Note: The `LanguageMixin` is required, but if you have a site where there's only one language, you don't even have to show the `language_code` field in your administration panel. Simply make sure that the `LANGUAGES` setting contains only the one language and nothing else.

3.2.2 The application

Add the `app/forms.py` module itself. Note that since control is directly handed to the application view and no page view code runs you'll have to load the page instance yourself and do the necessary language setup and provide the page etc. to the rendering context. The best way to load the page instance responsible for the current app is by calling *`feincms3.apps.page_for_app_request()`*:

```
from django.conf.urls import url
from django.http import HttpResponseRedirect
from django.shortcuts import render

from feincms3.apps import page_for_app_request
from feincms3.regions import Regions

from app.pages.renderer import renderer
```

(continues on next page)

(continued from previous page)

```
def form(request):
    page = page_for_app_request(request)
    page.activate_language(request)

    context = {}

    if "ok" not in request.GET:
        form_class = page.form.form()

        if request.method == "POST":
            form = form_class(request.POST)

            if form.is_valid():
                # Discard return values from form processing.
                page.form.process(form, request)
                return HttpResponseRedirect("?ok")

        else:
            form = form_class()

        context["form"] = form

    context.update({
        "page": page,
        "regions": Regions.from_item(
            page,
            renderer=renderer,
            inherit_from=page.ancestors().reverse(),
            timeout=60,
        )
    })

    return render(request, "form.html", context)


app_name = "forms"
urlpatterns = [
    url(r"^$", form, name="form"),
]
```

Add the required template:

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

{% load feincms3 %}

{% block content %}
  {% render_region regions 'main' %}

  {% if form %}
    <form method="post" action=".#form" id="form">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Submit</button>
    </form>
```

(continues on next page)

(continued from previous page)

```
  {% else %}
    <h1>Thank you!</h1>
  {% endif %}
{% endblock %}
```

Of course if you'd rather add another URL for the "thank you" page you're free to add a second entry to the `urlpatterns` list and redirect to this URL instead.

3.2.3 Outlook

The example above shows how to add a contact form at the end of the rest of the content. However, it would be quite easy to e.g. add a placeholder plugin which content managers can use to place the form somewhere in-between. An outline how this might be done follows:

The plugin model definition:

```
class Placeholder(PagePlugin):
    identifier = models.CharField(choices=[("form", "form")], ...)
```

The app:

```
def form(request):
    page = ...

    context = {}

    if "ok" not in request.GET:
        context.setdefault("placeholders", {})["form"] = form

    context.update({
        "page": page,
        "regions": renderer.regions(
            page, inherit_from=page.ancestors().reverse()),
    })

    return render(request, "form.html", context)
```

The rendering of the placeholder:

```
renderer.register_template_renderer(
    models.Placeholder,
    lambda plugin: "placeholder/{}.html".format(plugin.identifier),
    lambda plugin, context: {
        "plugin": plugin,
        "placeholder": context["placeholders"].get(plugin.identifier),
    },
)
```

The `placeholder/form.html` template:

```
<form method="post" action=".#form" id="form">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Submit</button>
</form>
```

The rest of the steps is left as an exercise to the reader. The success message is missing, and missing is also what happens if the placeholder plugin hasn't been added to the page.

3.3 Apps and instances

Applications can be added to the tree several times. Django itself supports this through the differentiation between application namespaces and instance namespaces. feincms3 builds on this functionality.

The *feincms3.apps.apps_urlconf()* function generates a dynamic URLconf Python module including all applications in their assigned place and adding the `urlpatterns` from `ROOT_URLCONF` at the end (or returns the value of `ROOT_URLCONF` directly if there are no active applications).

3.3.1 Application namespaces and instance namespace

The application URLconfs are included using nested namespaces:

- The outer application namespace is `"apps"` by default.
- The outer instance namespace is `"apps-" + LANGUAGE_CODE`.
- The inner namespace is the app namespace, specified by the value of `app_name` în the apps' URLconf module. The string must correspond with the value used in the `APPLICATIONS` list on the page model. application's name in the `APPLICATIONS` list.
- The inner instance namespace is the same as the app namespace, except if you return a different value in the applications `app_instance_namespace` function specified in the `APPLICATIONS` list.

Apps are contained in nested URLconf namespaces which allows for URL reversing using Django's `reverse()` mechanism. The inner namespace is the app itself, the outer namespace the language. (Currently the apps code depends on *LanguageMixin* and cannot be used without it.) *reverse_app()* hides a good part of the complexity of finding the best match for a given view name since apps will often be added several times in different parts of the tree, especially on sites with more than one language.

3.3.2 Reversing application URLs

The best way for reversing app URLs is by using *feincms3.apps.reverse_app()*. The method expects at least two arguments, a namespace and a viewname. The namespace argument also supports passing a list of namespaces which is useful in conjunction with the `app_instance_namespace` option of applications.

reverse_app() first generates a list of viewnames and passes them on to *feincms3.apps.reverse_any()* (which returns the first viewname that can be reversed to a URL).

For the sake of an example let's assume that our site is configured with english, german and french as available languages and that we're trying to reverse the article list page, and that we are processing a german page:

```
from feincms3.apps import reverse_app

def page_detail(request, path=None):
    page = ...
    page.activate_language(request)

    articles_list_url = reverse_app("articles", "article-list")
    ...
```

The list of viewnames reversed is in order:

- apps-de.articles.article-list
- apps-en.articles.article-list
- apps-fr.articles.article-list

The german apps namespace comes first in the list. If the german part of the site does not contain an articles app, the reversing continues in all other languages.

If the namespace argument to *reverse_app()* was a list (or tuple), the list is even longer. Suppose that variants of the articles app may be added to the tree where only a single category is shown:

```
class Page(AppsMixin, LanguageMixin, ..., AbstractPage):
    APPLICATIONS = [
        ("articles", _("Articles"), {
            "urlconf": "app.articles.urls",
            "app_instance_namespace": lambda page: "{}-{}".format(
                page.application, page.category_id or "all"
            ),
        }),
        ...
    ]

    category = models.ForeignKey(
        "articles.Category",
        blank=True,
        null=True,
        ...
    )
```

In this case we might prefer the URL of a specific categories' articles app, but also be content with an articles app without a specific category:

```
reverse_app(
    ["articles-{}".format(category.pk), "articles"],
    "article-list"
)
```

The list of viewnames in this case is (assuming that the category has a `pk` value of 42):

- apps-de.articles-42.article-list
- apps-de.articles.article-list
- apps-en.articles-42.article-list
- apps-en.articles.article-list
- apps-fr.articles-42.article-list
- apps-fr.articles.article-list

As you can see `reverse_app` prefers apps in the current language to apps with the closer matching instance namespace.

Note: Some of the time Django's stock `reverse()` function works as well for reversing app URLs, e.g:

```
from django.urls import reverse

reverse("apps:articles:article-list")
```

However, it's still recommended to use `reverse_app`. `reverse` may not find apps because Django is content with the first match when searching for matching namespaces. Also, `reverse` may not find the best match in the presence of several app instances, be it because of several languages on the site or because of other factors.

3.3.3 Reversing URLs outside the request-response cycle

Outside the request-response cycle, respectively outside *`feincms3.apps.apps_middleware()`*'s `request.urlconf` assignment, the reversing functions only use the URLconf module configured using the `ROOT_URLCONF` setting. In this case applications are impossible to find. However, all reversing functions support specifying the root URLconf module used for reversing:

```
from feincms3.apps import apps_urlconf, reverse_app

reverse_app("articles", "article-list", urlconf=apps_urlconf())
```

CHAPTER 4

Reference

4.1 Admin classes (`feincms3.admin`)

class feincms3.admin.**TreeAdmin**(*model*, *admin_site*)

ModelAdmin subclass for managing models using django-tree-queries trees.

Shows the tree's hierarchy and adds a view to move nodes around. To use this class the two columns `indented_title` and `move_column` should be added to subclasses `list_display`:

```
class NodeAdmin(TreeAdmin):
    list_display = ('indented_title', 'move_column', ...)

admin.site.register(Node, NodeAdmin)
```

get_queryset(*request*)

Return a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

get_urls()

Add our own `move` view.

indented_title(*instance*)

Use Unicode box-drawing characters to visualize the tree hierarchy.

move_column(*instance*)

Show a `move` link which leads to a separate page where the move destination may be selected.

class feincms3.admin.**MoveForm**(**args*, ***kwargs*)

Allows making the node the left or right sibling or the first or last child of another node.

Requires the node to be moved as `obj` keyword argument.

clean()

Hook for doing any extra form-wide cleaning after Field.clean() has been called on every field. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field named '__all__'.

class feincms3.admin.**CloneForm**(**args*, ***kwargs*)

clean()
Hook for doing any extra form-wide cleaning after Field.clean() has been called on every field. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field named '__all__'.

class feincms3.admin.**AncestorFilter**(*request*, *params*, *model*, *model_admin*)
Only show the subtree of an ancestor

By default, the first two levels are shown in the `list_filter` sidebar. This can be changed by setting the `max_depth` class attribute to a different value.

Usage:

```
class NodeAdmin(TreeAdmin):
    list_display = ('indented_title', 'move_column', ...)
    list_filter = ('is_active', AncestorFilter, ...)

admin.site.register(Node, NodeAdmin)
```

lookups(*request*, *model_admin*)
Must be overridden to return a list of tuples (value, verbose value)

queryset(*request*, *queryset*)
Return the filtered queryset.

4.2 Apps (`feincms3.apps`)

class feincms3.apps.**AppsMixin**(**args*, ***kwargs*)
The page class should inherit this mixin. It adds an `application` field containing the selected application, and an `app_instance_namespace` field which contains the instance namespace of the application. Most of the time these two fields will have the same value. This mixin also ensures that applications can only be activated on leaf nodes in the page tree. Note that currently the *`LanguageMixin`* is a required dependency of *`feincms3.apps`*.

`APPLICATIONS` contains a list of application configurations consisting of:

- Application name (used as instance namespace)
- User-visible name
- Options dictionary

Available options include:

- `urlconf`: The path to the URLconf module for the application. Besides the `urlpatterns` list the module should probably also specify a `app_name`.
- `required_fields`: A list of page class fields which must be non-empty for the application to work. The values are checked in `AppsMixin.clean_fields`.
- `app_instance_namespace`: A callable which receives the page instance as its only argument and returns a string suitable for use as an instance namespace.

Usage:

```
from django.utils.translation import ugettext_lazy as _
from feincms3.apps import AppsMixin
from feincms3.mixins import LanguageMixin
from feincms3.pages import AbstractPage

class Page(AppsMixin, LanguageMixin, AbstractPage):
    APPLICATIONS = [
        ("publications", _("publications"), {
            "urlconf": "app.articles.urls",
        }),
        ("blog", _("blog"), {
            "urlconf": "app.articles.urls",
        }),
        ("contact", _("contact form"), {
            "urlconf": "app.forms.contact_urls",
        }),
        ("teams", _("teams"), {
            "urlconf": "app.teams.urls",
            "app_instance_namespace": lambda page: "%s-%s" % (
                page.application,
                page.team_id,
            ),
            "required_fields": ("team",),
        }),
    ]
```

LANGUAGE_CODES_NAMESPACE = 'apps'
Override this to set a different name for the outer namespace.

application_config()
Returns the selected application options dictionary, or `None` if no application is selected or the application does not exist anymore.

clean_fields(*exclude=None*)
Checks that application nodes do not have any descendants, and that required fields for the selected application (if any) are filled out, and that app instances with the same instance namespace and same language only exist once on a site.

static fill_application_choices(*sender*, ***kwargs*)
Fills in the choices for `application` from the `APPLICATIONS` class variable. This method is a receiver of Django's `class_prepared` signal.

save(**args*, ***kwargs*)
Updates `app_instance_namespace`.

feincms3.apps.**apps_middleware**(*get_response*)
This middleware must be put in `MIDDLEWARE`; it simply assigns the return value of *apps_urlconf()* to `request.urlconf`. This middleware should probably be one of the first since it has to run before any resolving happens.

feincms3.apps.**apps_urlconf**(***, *apps=None*)
Generates a dynamic URLconf Python module including all applications in their assigned place and adding the `urlpatterns` from `ROOT_URLCONF` at the end. Returns the value of `ROOT_URLCONF` directly if there are no active applications.

Since Django uses an LRU cache for URL resolvers, we try hard to only generate a changed URLconf when application URLs actually change.

The application URLconfs are put in nested namespaces:

- The outer application namespace is `apps` by default. This value can be overridden by setting the `LANGUAGE_CODES_NAMESPACE` class attribute of the page class to a different value. The instance namespaces consist of the `LANGUAGE_CODES_NAMESPACE` value with a language added at the end. As long as you're always using `reverse_app` you do not have to know the specifics.
- The inner namespace is the app namespace, where the application namespace is defined by the app itself (assign `app_name` in the same module as `urlpatterns`) and the instance namespace is defined by the application name (from `APPLICATIONS`).

Modules stay around as long as the Python (most of the time WSGI) process lives. Unloading modules is tricky and probably not worth it since the URLconf modules shouldn't gobble up much memory.

The set of applications can be overridden by passing a list of `(path, application, app_instance_namespace, language_code)` tuples.

feincms3.apps.**page_for_app_request**(*request*, *, *queryset=None*)

Returns the current page if we're inside an app. Should only be called while processing app views. Will pass along exceptions caused by non-existing or duplicated apps (this should never happen inside an app because *apps_urlconf()* wouldn't have added the app in the first place if a matching page wouldn't exist, but still.)

Example:

```
def article_detail(request, slug):
    page = page_for_app_request(request)
    page.activate_language(request)
    instance = get_object_or_404(Article, slug=slug)
    return render(request, "articles/article_detail.html", {
        "article": article,
        "page": page,
    })
```

It is possible to override the queryset used to fetch a page instance. The default implementation simply uses the first concrete subclass of *AppsMixin*.

feincms3.apps.**reverse_any**(*viewnames*, *urlconf=None*, *args=None*, *kwargs=None*, **fargs*, ***fkwargs*)

Tries reversing a list of viewnames with the same arguments, and returns the first result where no `NoReverseMatch` exception is raised.

Usage:

```
url = reverse_any((
    'blog:article-detail',
    'articles:article-detail',
), kwargs={'slug': 'article-slug'})
```

feincms3.apps.**reverse_app**(*namespaces*, *viewname*, **args*, *languages=None*, ***kwargs*)

Reverse app URLs, preferring the active language.

`reverse_app` first generates a list of viewnames and passes them on to `reverse_any`.

Assuming that we're trying to reverse the URL of an article detail view, that the project is configured with german, english and french as available languages, french as active language and that the current article is a publication, the viewnames are:

- `apps-fr.publications.article-detail`
- `apps-fr.articles.article-detail`
- `apps-de.publications.article-detail`
- `apps-de.articles.article-detail`

- `apps-en.publications.article-detail`
- `apps-en.articles.article-detail`

reverse_app tries harder returning an URL in the correct language than returning an URL for the correct instance namespace.

Example:

```
url = reverse_app(
    ("category-1", "blog"),
    "post-detail",
    kwargs={"year": 2016, "slug": "my-cat"},
)
```

`feincms3.apps.`**`reverse_fallback`**(*fallback*, *fn*, **args*, ***kwargs*)

Returns the result of `fn(*args, **kwargs)`, or `fallback` if the former raises a `NoReverseMatch` exception. This is especially useful for reversing app URLs from outside the app and you do not want crashes if the app isn't available anywhere.

The following two examples are equivalent, choose whichever you like best:

```
reverse_fallback("/", lambda: reverse_app(
    ("articles",),
    "article-detail",
    kwargs={"slug": self.slug},
))

reverse_fallback(
    "/",
    reverse_app
    ("articles",),
    "article-detail",
    kwargs={"slug": self.slug},
)
```

`feincms3.templatetags.feincms3.`**`reverse_app`**(*parser*, *token*)

Reverse app URLs, preferring the active language.

Usage:

```
{% load feincms3 %}
{% reverse_app 'blog' 'detail' [args] [kw=args] [fallback='/'] %}
```

`namespaces` can either be a list or a comma-separated list of namespaces. `NoReverseMatch` exceptions can be avoided by providing a `fallback` as a keyword argument or by saving the result in a variable, similar to `{% url 'view' as url %}` does:

```
{% reverse_app 'newsletter' 'subscribe-form' fallback='/newsletter/' %}
```

Or:

```
{% reverse_app 'extranet' 'login' as login_url %}
```

4.3 HTML cleansing (`feincms3.cleanse`)

HTML cleansing is by no means only useful for user generated content. Managers also copy-paste content from word processing programs, the rich text editor's output isn't always (almost never) in the shape we want it to be, and a strict allowlist based HTML sanitizer is the best answer I have.

class feincms3.cleanse.**CleansedRichTextField**(**args*, ***kwargs*)

This is a subclass of django-ckeditor's `RichTextField`. The recommended configuration is as follows:

```
CKEDITOR_CONFIGS = {
    "default": {
        "toolbar": "Custom",
        "format_tags": "h1;h2;h3;p;pre",
        "toolbar_Custom": [[
            "Format", "RemoveFormat", "-",
            "Bold", "Italic", "Subscript", "Superscript", "-",
            "NumberedList", "BulletedList", "-",
            "Anchor", "Link", "Unlink", "-",
            "HorizontalRule", "SpecialChar", "-",
            "Source",
        ]],
    },
}

# Settings for feincms3.plugins.richtext.RichText
CKEDITOR_CONFIGS["richtext-plugin"] = CKEDITOR_CONFIGS["default"]
```

The corresponding `HTML_SANITIZERS` configuration for html-sanitizer would look as follows:

```
HTML_SANITIZERS = {
    "default": {
        "tags": {
            "a", "h1", "h2", "h3", "strong", "em", "p",
            "ul", "ol", "li", "br", "sub", "sup", "hr",
        },
        "attributes": {
            "a": ("href", "name", "target", "title", "id", "rel"),
        },
        "empty": {"hr", "a", "br"},
        "separate": {"a", "p", "li"},

        # Additional default settings not listed here.
    },
}
```

At the time of writing those are the defaults of html-sanitizer, so you don't have to do anything.

If you want or require a different cleansing function, simply override the default with `CleansedRichTextField(cleanse=your_function)`. The cleansing function receives the HTML as its first and only argument and returns the cleansed HTML.

clean(*value*, *instance*)

Convert the value's type and run validation. Validation errors from to_python() and validate() are propagated. Return the correct value if no error is raised.

feincms3.cleanse.**cleanse_html**(*html*)

Pass ugly HTML, get nice HTML back.

4.4 Mixins (feincms3.mixins)

class feincms3.mixins.**LanguageMixin**(**args*, ***kwargs*)
Pages may come in varying languages. LanguageMixin helps with that.

activate_language(*request*)
activate() the page's language and set request.LANGUAGE_CODE

class feincms3.mixins.**MenuMixin**(**args*, ***kwargs*)
The MenuMixin is most useful on pages where there are menus with differing content on a single page, for example the main navigation and a meta navigation (containing contact, imprint etc.)

static fill_menu_choices(*sender*, ***kwargs*)
Fills in the choices for menu from the MENUS class variable. This method is a receiver of Django's class_prepared signal.

class feincms3.mixins.**RedirectMixin**(**args*, ***kwargs*)
The RedirectMixin allows adding redirects in the page tree.

clean_fields(*exclude=None*)
Ensure that redirects are configured properly.

class feincms3.mixins.**TemplateMixin**(**args*, ***kwargs*)
It is sometimes useful to have different templates for CMS models such as pages, articles or anything comparable. The TemplateMixin provides a ready-made solution for selecting django-content-editor Template instances through Django's administration interface.

static fill_template_key_choices(*sender*, ***kwargs*)
Fills in the choices for menu from the MENUS class variable. This method is a receiver of Django's class_prepared signal.

regions
Return the selected template instances' regions attribute, falling back to an empty list if no template instance could be found.

template
Return the selected template instance if the template_key field matches, or None.

4.5 Pages (feincms3.pages)

class feincms3.pages.**AbstractPage**(**args*, ***kwargs*)
Short version: If you want to build a CMS with a hierarchical page structure, use this base class.

It comes with the following fields:

- parent: (a nullable tree foreign key) and a position field for relatively ordering pages. While it is technically possible for position to be 0, e.g., data bulk imported from another CMS, it is not recommended, as the save() method will override values of 0 if you manipulate pages using the ORM.
- is_active: Boolean field. The save() method ensures that inactive pages never have any active descendants.
- title and slug
- path: The complete path to the page, starting and ending with a slash. The maximum length of path (1000) should be enough for everyone (tm, famous last words). This field also has a unique index, which means that MySQL with its low limit on unique indexes will not work with this base class. Sorry.

- `static_path`: A boolean which, when `True`, allows you to fill in the `path` field all by yourself. By default, `save()` ensures that the `path` fields are always composed of a concatenation of the parent's `path` with the page's own `slug` (with slashes of course). This is especially useful for root pages (set `path` to `/`) or, when building a multilingual site, for language root pages (i.e. `/en/`, `/de/`, `/pt-br/` etc.)

clean_fields(*exclude=None*)
Check for path uniqueness problems.

get_absolute_url()
Return the page's absolute URL using `reverse()`

If path is `/`, reverses `pages:root` without any arguments, alternatively reverses `pages:page` with an argument of `path`. Note that this `path` is not the same as `self.path` – slashes are stripped from the beginning and the end of the string to make building an URLconf more straightforward.

save(*self*, ..., *save_descendants=None*)
Saves the page instance, and traverses all descendants to update their `path` fields and ensure that inactive pages (`is_active=False`) never have any descendants with `is_active=True`.

By default, descendants are only updated when any of `is_active` and `path` change. This can be overridden by either forcing updates using `save_descendants=True` or skipping them using `save_descendants=False`.

class feincms3.pages.**AbstractPageManager**
Defines a single method, `active`, which only returns pages with `is_active=True`.

active()
Return only active pages

This function is used in *apps_urlconf()* and is the recommended way to fetch active pages in your code as well.

get_queryset()
Return a new QuerySet object. Subclasses can override this method to customize the behavior of the Manager.

4.6 Plugins (feincms3.plugins)

Note: The content types in FeinCMS had ways to process requests and responses themselves, the `.process()` and `.finalize()` methods. feincms3 plugins do not offer this. The feincms3 way to achieve the same thing is by using *apps* or by adding the functionality in your own views (which are much simpler than the view in FeinCMS was).

4.6.1 External

Uses the Noembed oEmbed service to embed (almost) arbitrary URLs. Depends on requests.

class feincms3.plugins.external.**External**(**args*, ***kwargs*)
External content plugin

class feincms3.plugins.external.**ExternalInline**(*parent_model*, *admin_site*)
Content editor inline using the `ExternalForm` to verify whether the given URL is embeddable using oEmbed or not.

form
alias of `ExternalForm`

feincms3.plugins.external.**oembed_json**(*url*, *, *cache_failures=True*)
Asks Noembed for the embedding HTML code for arbitrary URLs. Sites supported include Youtube, Vimeo, Twitter and many others.

Successful embeds are always cached for 30 days.

Failures are cached if `cache_failures` is `True` (the default). The durations are as follows:

- Connection errors are cached 60 seconds with the hope that the connection failure is only transient.
- HTTP errors codes and responses in an unexpected format (no JSON) are cached for 24 hours.

The return value is always a dictionary, but it may be empty.

feincms3.plugins.external.**oembed_html**(*url*, *, *cache_failures=True*)
Wraps *oembed_json()*, but only returns the HTML part of the OEmbed response.

The return value is always either a HTML fragment or an empty string.

feincms3.plugins.external.**render_external**(*plugin*, ***kwargs*)
Render the plugin, embedding it in the appropriate markup for Foundation's responsive-embed element (https://foundation.zurb.com/sites/docs/responsive-embed.html)

4.6.2 HTML

Plugin providing a simple textarea where raw HTML, CSS and JS code can be entered.

Most useful for people wanting to shoot themselves in the foot.

class feincms3.plugins.html.**HTML**(**args*, ***kwargs*)
Raw HTML plugin

class feincms3.plugins.html.**HTMLInline**(*parent_model*, *admin_site*)
Just available for consistency, absolutely no difference to a standard `ContentEditorInline`.

feincms3.plugins.html.**render_html**(*plugin*, ***kwargs*)
Return the HTML code as safe string so that it is not escaped. Of course the contents are not guaranteed to be safe at all

4.6.3 Images

Provides an image plugin with support for setting the primary point of interest. This is very useful especially when cropping images. Depends on django-imagefield.

class feincms3.plugins.image.**Image**(**args*, ***kwargs*)
Image plugin

class feincms3.plugins.image.**ImageInline**(*parent_model*, *admin_site*)
Image inline

feincms3.plugins.image.**render_image**(*plugin*, ***kwargs*)
Return a simple, unscaled version of the image

4.6.4 Rich text

Provides a rich text area whose content is automatically cleaned using a very restrictive allowlist of tags and attributes.

Depends on django-ckeditor and html-sanitizer.

class feincms3.plugins.richtext.**RichText**(**args*, ***kwargs*)
Rich text plugin

To use this, a django-ckeditor configuration named `richtext-plugin` is required. See the section *HTML cleansing* for the recommended configuration.

class feincms3.plugins.richtext.**RichTextInline**(*parent_model*, *admin_site*)
The only difference with the standard `ContentEditorInline` is that this inline adds the `feincms3/plugin_ckeditor.js` file which handles the CKEditor widget activation and deactivation inside the content editor.

feincms3.plugins.richtext.**render_richtext**(*plugin*, ***kwargs*)
Return the text of the rich text plugin as a safe string (`mark_safe`)

4.6.5 Snippets

Plugin for including template snippets through the CMS

class feincms3.plugins.snippet.**Snippet**(**args*, ***kwargs*)
Template snippet plugin

static fill_template_name_choices(*sender*, ***kwargs*)
Fills in the choices for `template_name` from the `TEMPLATES` class variable. This method is a receiver of Django's `class_prepared` signal.

classmethod register_with(*renderer*)
This helper registers the snippet plugin with a `TemplatePluginRenderer` while adding support for template-specific context functions. The templates specified using the `TEMPLATES` class variable may contain a callable which receives the plugin instance and the template context and returns a context dictionary.

class feincms3.plugins.snippet.**SnippetInline**(*parent_model*, *admin_site*)
Snippet inline does nothing special, it simply exists for consistency with the other feincms3 plugins

feincms3.plugins.snippet.**render_snippet**(*plugin*, ***kwargs*)
Renders the selected template using `render_to_string`

4.6.6 Versatile images

Provides an image plugin with support for setting the primary point of interest. This is very useful especially when cropping images. Depends on django-versatileimagefield.

Note: While this plugin works well too the recommended image plugin is *feincms3.plugins.image*.

class feincms3.plugins.versatileimage.**Image**(**args*, ***kwargs*)
Image plugin

class feincms3.plugins.versatileimage.**AlwaysChangedModelForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

This `ModelForm`'s `has_changed` method always returns `True`. This is a workaround for the problem where Django's inlines do not detect changes in `MultiValueField` (which is used to set the PPOI – primary point of interest – in django-versatileimagefield).

https://github.com/respondcreate/django-versatileimagefield/issues/44

has_changed()

Return True if data differs from initial.

class feincms3.plugins.versatileimage.**ImageInline**(*parent_model*, *admin_site*)

Image inline using the `AlwaysChangedModelForm` to work around a bug where PPOI modifications were not picked up.

form

alias of *AlwaysChangedModelForm*

feincms3.plugins.versatileimage.**render_image**(*plugin*, ***kwargs*)

Return a simple, unscaled version of the image

4.7 Regions (`feincms3.regions`)

class feincms3.regions.**Regions**(*, *contents*, *renderer*, *cache_key=None*, *timeout=None*)

`Regions` uses `content_editor.contents.Contents` and the `feincms3.renderer.TemplatePluginRenderer` to convert a list of plugins into a rendered representation, most often a HTML fragment.

This class may also be instantiated directly but using the factory methods (starting with `from_`) below is probably more comfortable.

classmethod from_contents(*contents*, *, *renderer*, ***kwargs*)

Create and return a regions instance using the bare minimum of a contents instance and a renderer. Additional keyword arguments are forwarded to the regions constructor.

classmethod from_item(*item*, *, *renderer*, *inherit_from=None*, *timeout=None*, ***kwargs*)

Create and return a regions instance for an item (for example a page, an article or anything else managed by django-content-editor).

The item's plugins are determined by what is registered with the renderer. The plugin instances themselves are loaded lazily, and loading every time can be avoided completely by specifying a `timeout`.

generate(*items*, *context*)

Inspects all items in the region for a `subregion` attribute and passes control to the subregions' respective rendering handler, named `handle_<subregion>`. If `subregion` is not set or is falsy `handle_default` is invoked instead. This method raises a `KeyError` exception if no matching handler exists.

You probably want to call this method when overriding `render`.

handle_default(*items*, *context*)
Renders items from the queue using the renderer instance as long as the items either have no `subregion` attribute or whose `subregion` attribute is an empty string.

render(*region*, *context=None*)
Main function for rendering.

Starts the generator and assembles all fragments into a safe HTML string.

feincms3.regions.**matches**(*item*, *, *plugins=None*, *subregions=None*)
Checks whether the item matches zero or more constraints.

`plugins` should be a tuple of plugin classes or `None` if the type shouldn't be checked.

`subregions` should be set of allowed `subregion` attribute values or `None` if the `subregion` attribute shouldn't be checked at all. Include `None` in the set if you want `matches` to succeed also when encountering an item without a `subregion` attribute.

feincms3.regions.**cached_render**(*fn*)
Decorator for `Regions.render` methods implementing caching behavior

feincms3.templatetags.feincms3.**render_region**(*context*, *regions*, *region*, ***kwargs*)
Render a single region. See *Regions* for additional details. Any and all keyword arguments are forwarded to the `render` method of the `Regions` instance.

Usage:

```
{% render_region regions "main" %}
```

4.8 Renderer (feincms3.renderer)

class feincms3.renderer.**TemplatePluginRenderer**(*, *regions_class=<class 'feincms3.renderer.Regions'>*)
This renderer allows registering functions, templates and context providers for plugins. It also supports rendering plugins' templates using the rendering context of the surrounding template without explicitly copying required values into the local rendering context.

plugins()
Return a list of all registered plugins, and is most useful when passed directly to one of django-content-editor's contents utilities:

```
page = get_object_or_404(Page, ...)
contents = contents_for_item(page, renderer.plugins())
```

regions(*item*, *, *inherit_from=None*, *regions=None*)
Return a `Regions` instance which lazily wraps the `contents_for_item` call. This is especially useful in conjunction with the `render_region` template tag. The `inherit_from` argument is directly forwarded to `contents_for_item` to allow regions with inherited content.

The `Regions` type may be overridden by passing a `regions_class` keyword argument when instantiating the `TemplatePluginRenderer` or by setting the `regions` argument of this method.

register_string_renderer(*plugin*, *renderer*)
Register a rendering function which is passed the plugin instance and returns a HTML string:

```
renderer.register_string_renderer(
    RichText,
    lambda plugin: mark_safe(plugin.text),
)
```

register_template_renderer(*self*, *plugin*, *template_name*, *context=default_context*)

Register a renderer for `plugin` using a template. The template uses the same mechanism as `{% include %}` meaning that the full template context is available to the plugin renderer.

`template_name` can be one of:

- A template path
- A list of template paths
- An object with a `render` method
- A callable receiving the plugin as only parameter and returning any of the above.

`context` must be a callable receiving the plugin instance and the template context and returning a dictionary. The default implementation simply returns a dictionary containing a single key named `plugin` containing the plugin instance.

```
# Template snippets have access to everything in the template
# context, including for example ``page``, ``request``, etc.
renderer.register_template_renderer(
    Snippet,
    lambda plugin: plugin.template_name,
)

# Additional context can be provided:
renderer.register_template_renderer(
    Team,
    'pages/plugins/team.html',  # Can also be a callable
    lambda plugin, context: {
        'persons': Person.objects.filter(
            # Assuming that the page has a team foreign key:
            team=plugin.parent.team,
        ),
    },
)
```

render_plugin_in_context(*plugin*, *context=None*)

Render a plugin, passing on the template context into the plugin's template (if the plugin uses a template renderer).

feincms3.renderer.**cached_render**(*fn*)

Decorator for `Regions.render` methods implementing caching behavior

This decorator consumes the `timeout` keyword argument to the `render` method.

feincms3.renderer.**default_context**(*plugin*, *context*)

Return the default context for plugins rendered with a template, which simply is a single variable named `plugin` containing the plugin instance.

class feincms3.renderer.**Regions**(*, *item*, *contents*, *renderer*)

Note: `feincms3.renderer.Regions` has been deprecated in favor of *feincms3.regions.*

Regions.

Wrapper for a `content_editor.contents.Contents` instance with support for caching the potentially somewhat expensive plugin loading and rendering step.

A view using this facility would look as follows:

```
def page_detail(request, slug):
    page = get_object_or_404(Page, slug=slug)
    return render(request, 'page.html', {
        'page': page,
        'regions': renderer.regions(
            page,
            # Optional:
            inherit_from=page.ancestors().reverse(),
            # Optional too:
            timeout=15,
        ),
        # Note! No 'contents' and no 'renderer' necessary in the
        # template.
    })
```

The template itself should contain the following snippet:

```
{% load feincms3 %}

{% block content %}

<h1>{{ page.title }}</h1>
<main>
  {% render_region regions "main" timeout=60 %}
</main>
<aside>
  {% render_region regions "sidebar" timeout=60 %}
</aside>

{% endblock %}
```

Caching is, of course, completely optional. When you're caching regions though you should probably cache them all, because accessing the content of a single region loads the content of all regions. (It might still make sense if the rendering is the expensive part, not the database access.)

Note: You should probably always let the renderer instantiate this class and not depend on the API, especially since the lazyness happens in the renderer, not in the `Regions` instance.

cache_key(*region*)
Return a cache key suitable for the given `region` passed

render(*self*, *region*, *context=None*, ***, *timeout=None*)
Render a single region using the context passed

If `timeout` is `None` caching is disabled.

`feincms3.templatetags.feincms3.`**render_region**(*context*, *regions*, *region*, ***kwargs*)
Render a single region. See *Regions* for additional details. Any and all keyword arguments are forwarded to the `render` method of the `Regions` instance.

Usage:

```
{% render_region regions "main" %}
```

4.9 Shortcuts (`feincms3.shortcuts`)

For me, the most useful part of Django's generic class based views is the template name generation and the context variable naming for list and detail views, and also the pagination.

The rest of the CBV is less flexible than I'd like them to be, i.e. integrating forms on detail pages can be a hassle.

Because of this, `render_list` and `render_detail`.

`feincms3.shortcuts.`**`template_name`**(*model*, *template_name_suffix*)

Given a model and a template name suffix, return the resulting template path:

```
>>> template_name(Article, '_detail')
'articles/article_detail.html'
>>> template_name(User, '_form')
'auth/user_form.html'
```

`feincms3.shortcuts.`**`render_list`**(*request*, *queryset*, *context=None*, ***, *template_name_suffix='_list'*, *paginate_by=None*)

Render a list of items

Usage example:

```
def article_list(request, ...):
    queryset = Article.objects.published()
    return render_list(
        request,
        queryset,
        paginate_by=10,
    )
```

You can also pass an additional context dictionary and/or specify the template name suffix. The query parameter `page` is hardcoded for specifying the current page if using pagination.

The queryset (or the page if using pagination) are passed into the template as `object_list` AND `<model_name>_list`, i.e. `article_list` in the example above.

`feincms3.shortcuts.`**`render_detail`**(*request*, *object*, *context=None*, ***, *template_name_suffix='_detail'*)

Render a single item

Usage example:

```
def article_detail(request, slug):
    article = get_object_or_404(Article.objects.published(), slug=slug)
    return render_detail(
        request,
        article,
    )
```

An additional context dictionary is also supported, and specifying the template name suffix too.

The `Article` instance in the example above is passed as `object` AND `article` (lowercased model name) into the template.

4.10 Utils (`feincms3.utils`)

Note: The utils module is meant purely for feincms3's internal use. Utilities may be added and removed without prior warning and without a deprecation period.

If you depend on some functionality from this module copy the code into your project (according to the very permissive license of course).

`feincms3.utils.`**`validation_error`**(*error*, *, *field*, *exclude*, **kwargs*)

Return a validation error that is associated with a particular field if it isn't excluded from validation.

See https://github.com/django/django/commit/e8c056c31 for some background.

CHAPTER 5

Project links

5.1 Change log

5.1.1 Next version

- Added copying of `handler400`, `handler403`, `handler404` and `handler500` from `ROOT_URLCONF` to the URLconf module created by `apps_urlconf`.

5.1.2 0.30 (2019-03-18)

- Fixed overflowing tree structure boxes in the `TreeAdmin`.
- Switched to emitting `DeprecationWarning` warnings not `Warning`, even though their visibility sucks.
- Added a `languages` argument to `reverse_app` which allows overriding languages and their order.
- Made `TreeAdmin` and `MoveForm` only require that the default manager is a `TreeQuerySet` and not that the model itself also extends `TreeNode`.
- Made `plugin_ckeditor.js`'s dependency on `django.jQuery` explicit. This is necessary for Django 2.2's new `Media.merge` algorithm.

5.1.3 0.29 (2019-02-07)

- Deprecated the `feincms3_apps` and `feincms3_renderer` template tag library. `render_region` and `reverse_app` have been made available as `feincms3`. The `render_plugin` and `render_plugins` tags will be removed completely.
- Changed `feincms3.regions.matches` to the effect that `None` has to be provided explicitly as an allowed subregion if items with no `subregion` attribute should be matched too.
- Removed an use of six which is unnecessary now that we only support Python 3.
- Imported `lru_cache` from the Python library.

- Replaced `concrete_model` calls to determine the concrete subclass of `AppsMixin` with capturing the model instance locally in the `class_prepared` signal handler.
- Removed the now unused `concrete_model` and `iterate_subclasses` utilities.
- Replaced two more occurrences of `.objects` with `._default_manager`.
- Deprecated accessing the backwards compatibility properties `AbstractPage.depth` and `AbstractPage.cte_path`.
- Deprecated `feincms3.apps.AppsMiddleware` in favor of `feincms3.apps.apps_middleware`.

5.1.4 0.28 (2019-02-03)

- **(not yet) BACKWARDS INCOMPATIBLE** Deprecated `TemplatePluginRenderer`'s `regions` method, the `regions_class` attribute and `feincms3.renderer.Regions`. Introduce the more versatile `feincms3.regions.Regions` class instead which also replaces the `feincms3.incubator.subrenderer` functionality and does not suffer from a software design problem where the regions and the renderer classes knew too much about each other. This has been bothering me for a long time already but became impossible to overlook in the subrenderer implementation.
- Updated the Travis CI matrix to cover more versions of Django and Python while reducing the total job count to speed up builds.
- Made the default textarea used for editing the HTML plugin smaller.
- Added documentation for the new `reenter` subrenderer hook.
- Augmented the snippet plugin with a way to specify a template-specific plugin context callable.

5.1.5 0.27 (2019-01-15)

- Fixed the CKEditor plugin script to resize the widget to fit the width of the content editor area.
- Added configuration for easily running prettier and ESLint on the frontend code.
- Dropped Python 2 compatibility, again. The first attempt was made almost 30 months ago.
- Changed the subrenderer to use yielding instead of returning fragments.

5.1.6 0.26 (2018-11-22)

- Removed tree fields when loading applications.
- Stopped mentioning the `AppsMixin` in the reference documentation.
- Fixed a few typos and converted more string quotes in the docs.
- Changed the docs to use allow/deny instead of black/white.
- Changed `feincms3.plugins` do not hide import errors from our own modules anymore (again).
- Added a cloning functionality to copy the values of individual fields and also of the pages' content onto other pages.
- Fixed a problem where `Snippet.__str__` would unexpectedly (for Django) return lazy strings.
- Changed the type of `RedirectMixin.redirect_to_page` to `TreeNodeForeignKey` so that the hierarchy is shown in the dropdown.
- Added more careful detection of chain redirects and improved the error messages a bit.

- Made it clearer that `AbstractPage.position`'s value should probably be greater than zero. Thanks to Hannah Cushman for the contribution!

5.1.7 0.25 (2018-09-07)

- **BACKWARDS INCOMPATIBLE** Removed the imports of plugins into `feincms3.plugins`. Especially with the image plugins it could be non-obvious whether the plugin uses django-imagefield or django-versatileimagefield. Instead, the modules are imported so that classes and functions can be referenced using e.g. `plugins.image.Image` instead of `plugins.Image` as before.
- Moved the documentation from autodoc to a more guide-oriented format.
- Changed `TemplatePluginRenderer.render_plugin_in_context` to raise a specific `PluginNotRegistered` exception upon encountering unregistered plugins instead of a generic `KeyError`.
- Made it possible to pass fixed strings (not callables) to `TemplatePluginRenderer.register_string_renderer`.
- Added an incubator in `feincms3.incubator` for experimental modules with absolutely no compatibility guarantees.
- Changed the `TreeAdmin.move_view` to return a redirect to the admin index page instead of a 404 for missing nodes (as the Django admin's views also do since Django 1.11).
- Fixed an edge case in `apps_urlconf` which would generate a few nonsensical URLs if no language is activated currently.
- Made it an error to add redirects to a page which is already the target of a different redirect. Adding redirects to a page which itself already redirects was already an error.

5.1.8 0.24 (2018-08-25)

- Fixed one use of removed API.
- Fixed a bug where the move form "Save" button wasn't shown with Django 2.1.
- Made overriding the `Regions` type used in `TemplatePluginRenderer` less verbose.
- Modified the documentation to produce several pages. Completed the guide for building your own CMS and added a section about customizing rendering using `Regions` subclasses.

5.1.9 0.23 (2018-07-30)

- Switched the preferred quote to `"` and started using black to automatically format Python code.

Switched to a new library for recursive common table expressions

django-tree-queries supports more database engines, which means that the PostgreSQL-only days of feincms3 are gone.

Incompatible differences are few:

- The attributes on page objects are named `tree_depth` and `tree_path` now instead of `depth` and `cte_path`. If you're using `WHERE` clauses on your querysets change `depth` to `__tree.tree_depth` (or only `tree_depth`). Properties for backward compatibility have been added to the `AbstractPage` class, but of course those cannot be used in database queries.

- django-tree-queries uses the correct definition of node depth where root nodes have a depth of `0`, not `1`.
- django-tree-queries does not add the CTE by default to all queries, instead, users are expected to call `.with_tree_fields()` themselves if they want to use the CTE attributes. For the time being, the `AbstractPageManager` always returns querysets with tree fields.

5.1.10 0.22 (2018-05-04)

- Fixed a problem in `MoveForm` where invalid move targets would crash because of missing form fields to attach the error to instead of showing the underlying problem.
- Made it possible to override the list of apps processed in `apps_urlconf`.
- Converted the apps middleware into a function, now named `apps_middleware`. The old name `AppsMiddleware` will stay available for some undefined time.
- Made the path clash check less expensive by running less SQL queries.
- Made page saving a bit less expensive by only saving descendants when `is_active` or `path` changed.

5.1.11 0.21 (2018-03-28)

- Added a template tag for `reverse_app`.
- **(At least a bit) BACKWARDS INCOMPATIBLE** Switched the preferred image field from django-versatileimagefield to django-imagefield. The transition should mostly require replacing `versatileimagefield` with `imagefield` in your settings etc., adding the appropriate `IMAGEFIELD_FORMATS` setting and running `./manage.py process_imagefields` once. Switch from `feincms3[all]` to `feincms3[versatileimagefield]` to stay with django-versatileimagefield for the moment.

5.1.12 0.20 (2018-03-21)

- Changed `render_list` and `render_detail` to return `TemplateResponse` instances instead of pre-rendered instances to increase the shortcuts' flexibility.
- Factored the JSON fetching from `oembed_html` into a new `oembed_json` helper.
- Added Django 2.0 to the Travis CI build (nothing had to be changed, 0.19 was already compatible)
- Changed the `TemplatePluginRenderer` to also work when used standalone, not from inside a template.
- Dropped compatibility with Django versions older than 1.11.
- Changed `AppsMixin.clean_fields` to use `_default_manager` instead of `_base_manager` to search for already existing app instances.
- Changed the page move view to suppress the "Save and add another" button with great force.

5.1.13 0.19 (2017-08-17)

The diff for this release is big, but there are almost no changes in functionality.

- Minor documentation edits, added a form builder example app to the documentation.
- Made `reverse_fallback` catch `NoReverseMatch` exceptions only, and fixed a related test which didn't reverse anything at all.

- Switch to tox for building docs, code style checking and local test running.
- Made the `forms.Media` CSS a list, not a set.

5.1.14 0.18 (2017-05-10)

- Slight improvements to `TreeAdmin`'s alignment of box drawing characters.
- Allow overriding the outer namespace name used in `feincms3.apps` by setting the `LANGUAGE_CODES_NAMESPACE` class attribute of the pages class. The default value of `language-codes` has been changed to `apps`. Also, the outer instance namespaces of apps are now of the form `<LANGUAGE_CODES_NAMESPACE>-<language_code>` (example: `apps-en` for english), not only `<language_code>`. This makes namespace collisions less of a concern.

5.1.15 0.17.1 (2017-05-02)

- Minor documentation edits.
- Added the `AncestorFilter` for filtering the admin changelist by ancestor. The default setting is to allow filtering by the first two tree levels.
- Switched from feincms-cleanse to html-sanitizer which allows configuring the allowed tags and attributes using a `HTML_SANITIZERS` setting.

5.1.16 0.16 (2017-04-24)

- Fixed the releasing-via-PyPI configuration.
- Removed strikethrough from our recommended rich text configuration, since feincms-cleanse would remove the tag anyway.
- Made `TemplatePluginRenderer.regions` and the `Regions` class into documented API.
- Made `register_template_renderer`'s `context` argument default to `default_context` instead of `None`, so please stop passing `None` and expecting the default context to work as before.
- Before adding Python 2 compatibility, a few methods and functions had keyword-only arguments. Python 2-compatible keyword-only enforcement has been added back to make it straightforward to transition back to keyword-only arguments later.

5.1.17 0.15 (2017-04-05)

- Dropped the `is_descendant_of` template tag. It was probably never used without `include_self=True`, and this particular use case is better covered by checking whether a given primary key is a member of `page.cte_path`.
- Dropped the `menu` template tag, and with it also the `group_by_tree` filter. Its arguments were interpreted according to the long-gone django-mptt and it promoted bad database querying patterns.
- Dropped the now-empty `feincms3_pages` template tag library.
- Added a default manager implementing `active()` to `AbstractPage`.

5.1.18 0.14 (2017-03-14)

- Removed Django from `install_requires` so that updating feincms3 without updating Django is easier.
- Allowed overriding the Page queryset used in `page_for_app_request` (for example for adding `select_related`).
- Moved validation logic in varous model mixins from `clean()` to `clean_fields(exclude)` to be able to attach errors to individual form fields (if they are available on the given form).
- Added Django 1.11 to the build matrix on Travis CI.
- Fixed an "interesting" bug where the `TreeAdmin` would crash with an `AttributeError` if no query has been run on the model before.

5.1.19 0.13 (2016-11-07)

- Fixed oEmbed read timeouts to not crash but retry after 60 seconds instead.
- Added the `TemplatePluginRenderer.regions` helper and the `{% render_region %}` template tag which support caching of plugins.
- Disallowed empty static paths for pages. `Page.get_absolute_url()` fails with the recommended URL pattern when `path` equals `''`.
- Added flake8 and isort style checking.
- Made the dependency on feincms-cleanse, requests and django-versatileimagefield less strong than before. Plugins depending on those apps simply will not be available in the `feincms3.plugins` namespace, but you have to be careful yourself to not import the actual modules yourself.
- Added Django, django-content-editor and django-cte-forest to `install_requires` so that they are automatically installed, and added an extra with dependencies for all included plugins, so if you want that simply install `feincms3[all]`.

5.1.20 0.12 (2016-10-23)

- Made `reverse_any` mention all viewnames in the `NoReverseMatch` exception instead of bubbling the last viewname's exception.
- Added a `RedirectMixin` to `feincms3.mixins` for redirecting pages to other pages or arbitrary URLs.
- Added a footgun plugin (raw HTML code).
- Reinstate Python 2 compatibility because Python 2 still seems to be in wide use.

5.1.21 0.11 (2016-09-19)

- Changed the implementation of the `is_descendant_of` template tag to not depend on django-mptt's API anymore, and removed the compatibility shims from `AbstractPage`.
- Made the documentation build again and added some documentation for the new `feincms3.admin` module.
- Made `TreeAdmin.move_view` run transactions on the correct database in multi-DB setups.
- Removed the unused `NoCommitException` class.
- Fixed a crash in the `MoveForm` validation.
- Made `AppsMiddleware` work with Django's `MIDDLEWARE` setting.

- Made the `{% menu %}` template tag not depend on a `page` variable in context.

5.1.22 0.10 (2016-09-13)

- **BACKWARDS INCOMPATIBLE** Switched from django-mptt to django-cte-forest which means that feincms3 is for the moment PostgreSQL-only. By switching we completely avoid the MPTT attribute corruption which plagued projects for years. The *lft* attribute is directly reusable as *position*, and should be renamed in a migration instead of created from scratch to avoid losing the ordering of nodes within a branch.
- Added a `feincms3.admin.TreeAdmin` which shows the tree hierarchy and has facilities for moving nodes around.
- Avoided a deprecation warning on Django 1.10 regarding `django.core.urlresolvers`.
- Started rolling releases using Travis CI's PyPI deployment provider.
- Made `{% is_descendant_of %}` return `False` if either of the variables passed is no page instance instead of crashing.

5.1.23 0.9 (2016-08-17)

- Dropped compatibility with Python 2.
- Fixed `AbstractPage.save()` to actually detect page moves correctly again. Calling `save()` in a transaction was a bad idea because it messed with MPTT's bookkeeping information. Depending on the transaction isolation level going back to a clean slate *after* `clean()` proved much harder than expected.

5.1.24 0.8 (2016-08-05)

- Added `feincms3.apps.reverse_fallback` to streamline reversing with fallback values in case of crashes.
- The default template renderer context (`TemplatePluginRenderer.register_template_renderer`) contains now the plugin instance as `plugin` instead of nothing.
- Make django-mptt-nomagic a required dependency, by depending on the fact that nomagic always calls `Page.save()` (django-mptt does not do that when nodes are moved using `TreeManager.node_move`, which is used in the draggable mptt admin interface. Use a `node_moved` signal listener which calls `save()` if the `node_moved` call includes a `position` keyword argument if you can't switch to django-mptt-nomagic for some reason.

5.1.25 0.7 (2016-07-21)

- Removed all dependencies from `install_requires` to make it easier to replace individual items.
- Enabled the use of `i18n_patterns` in `ROOT_URLCONF` by importing and adding the urlpatterns contained instead of `include()`-ing the module in `apps_urlconf`.
- Modified the cleansing configuration to allow empty `<a>` tags (mostly useful for internal anchors).
- Fixed crash when adding a page with a path that exists already (when not using a statich path).

5.1.26 0.6 (2016-07-11)

- Updated the translation files.
- Fixed crashes when path of pages would not be unique when moving subtrees.

5.1.27 0.5 (2016-07-07)

- Fixed a crash where apps without `required_fields` could not be saved.
- Added a template snippet based renderer for plugins.
- Prevented adding the exact same application (that is, the same `app_instance_namespace`) more than once.

5.1.28 0.4 (2016-07-04)

- Made application instances (`feincms3.apps`) more flexible by allowing programmatically generated instance namespace specifiers.

5.1.29 0.3 (2016-07-02)

- Lots of work on the documentation.
- Moved all signal receivers into their classes as staticmethods.
- Fixed a crash on an attempted save of an `External` plugin instance with an empty URL.
- Added an incomplete testsuite, and add the Travis CI badge to the README.
- Removed the requirement of passing a context to `render_list` and `render_detail`.

5.1.30 0.2 (2016-06-28)

- The external plugin admin form now checks whether the URL can be embedded using OEmbed or not.
- Added the `plugin_ckeditor.js` file required for the rich text editor.
- Added a `SnippetInline` for consistency.
- Ensured that choice fields have a `get_*_display` method by setting dummy choices in advance (menus, snippets and templates).
- Added automatically built documentation on readthedocs.io.

5.1.31 0.1 (2016-06-25)

- Plugins (apps, external, richtext, snippet and versatileimage) for use with django-content-editor.
- HTML editing and cleansing using django-ckeditor and feincms-cleanse.
- Shortcuts (`render_list` and `render_detail` – the most useful parts of Django's class based generic views)
- An abstract page base model building on django-mptt with mixins for handling templates, menus and language codes.

- Template tags for fetching and grouping menu entries inside templates.
- A german translation.

5.2 Contributing

This isn't a Jazzband project, but by contributing you agree to abide to the same Contributor Code of Conduct as if it was one.

5.2.1 Bug reports and feature requests

You can report bugs and request features in the bug tracker.

5.2.2 Code

The code is available on GitHub.

To work on the code I strongly recommend installing tox. I use tox as a glorified virtualenv-builder and task runner for local development.

Available tasks are:

- `tox -e style`: Reformats the code using black and runs flake8.
- `tox -e docs`: Builds the HTML docs into `build/docs/html/`
- `tox -e tests`: Runs tests using a local PostgreSQL server.
- `tox -e tests-sqlite3`: Runs tests using sqlite3.

Both testing tasks also generate HTML-based code coverage output into the `htmlcov/` folder.

5.2.3 Style

Python code for the feincms3 project may be automatically formatted and checked using `tox -e style`. The coding style is also checked when building pull requests on Travis CI.

5.2.4 Patches and translations

Please submit pull requests!

I am not using a centralized tool for translations right now, I'll happily accept them as a patch too.

5.2.5 Mailing list

If you wish to discuss a topic, please open an issue on Github. Alternatively, the django-feincms Google Group may also be used for discussing this project.

CHAPTER 6

Related projects

- feincms3-example: Example project demonstrating some of feincms3's capabilities.
- feincms3-sites: Multisite support for feincms3. Allows running a feincms3 site on several domains with separate page trees.
- feincms3-downloads: A downloads plugin which also supports thumbnailing e.g. PDFs using ImageMagick.
- feincms3-meta: Helpers and feincms3 mixins for making Open Graph tags and meta tags less annoying.
- django-cabinet: A media library for Django which works well with feincms3 and follows the same software design guidelines.
- django-content-editor: The admin interface for editing structured heterogenous content.
- django-imagefield: An image field with in-depth image file validation and thumbnailing support which does not depend on a cache to be and stay fast.
- django-sitemaps: Sitemaps generation using a real XML library and support for alternates.
- django-tree-queries: The library feincms3's pages use for querying tree-shaped data.
- html-sanitizer: Allowlist-based HTML sanitizer used for feincms3' rich text plugin.
- FeinCMS: First version.

Python Module Index

f

Index