FeinCMS Documentation

Release 1.9.5

Feinheit GmbH and contributors

July 22, 2014

Contents

FeinCMS is an extremely stupid content management system. It knows nothing about content – just enough to create an admin interface for your own page content types. It lets you reorder page content blocks using a drag-drop interface, and you can add as many content blocks to a region (f.e. the sidebar, the main content region or something else which I haven't thought of yet). It provides helper functions, which provide ordered lists of page content blocks. That's all.

Adding your own content types is extremely easy. Do you like textile that much, that you'd rather die than using a rich text editor? Then add the following code to your project, and you can go on using the CMS without being forced to use whatever the developers deemed best:

```python
from feincms.module.page.models import Page
from django.contrib.markup.templatetags.markup import textile
from django.db import models


class TextilePageContent(models.Model):
    content = models.TextField()

    class Meta:
        abstract = True

    def render(self, **kwargs):
        return textile(self.content)

Page.create_content_type(TextilePageContent)
```

That's it. Only ten lines of code for your own page content type.

Contents

1.1 Installation instructions

1.1.1 Installation

This document describes the steps needed to install FeinCMS.

FeinCMS requires a working installation of Django version 1.4, 1.5 or 1.6. See the Django documentation for how to install and configure Django.

You can download a stable release of FeinCMS using `pip`. Pip will install feincms and its dependencies. Dependencies which are automatically installed are: feedparser, Pillow and django-mptt.

> $ pip install feincms

In order to install documentation, tests or an example project, install from the Git repository instead:

```
$ git clone git://github.com/feincms/feincms.git
```

If you are looking to implement a blog, check out elephantblog.

You will also need a Javascript WYSIWYG editor of your choice (Not included). TinyMCE works out of the box and is recommended.

1.1.2 Configuration

There isn't much left to do apart from adding a few entries to `INSTALLED_APPS`. Most commonly you'll want to add:

```
feincms,
mptt,
feincms.module.page,
feincms.module.medialibrary
```

The customized administration interface needs some media and javascript libraries which you have to make available to the browser. FeinCMS uses Django's `django.contrib.staticfiles` application for this purpose. The media files will be picked up automatically by the `collectstatic` management command.

If your website is multi-language you have to define `LANGUAGES` in the settings.

Please note that the `feincms` module will not create or need any database tables, but you need to put it into `INSTALLED_APPS` because otherwise the templates in `feincms/templates/` will not be found by the template loader.

The tools contained in FeinCMS can be used for many CMS-related activities. The most common use of a CMS is to manage a hierarchy of pages and this is the most advanced module of FeinCMS too. Please proceed to *The built-in page module* to find out how you can get the page module up and running.

1.2 The built-in page module

FeinCMS is primarily a system to work with lists of content blocks which you can assign to arbitrary other objects. You do not necessarily have to use it with a hierarchical page structure, but that's the most common use case of course. Being able to put content together in small manageable pieces is interesting for other uses too, e.g. for weblog entries where you have rich text content interspersed with images, videos or maybe even galleries.

1.2.1 Activating the page module and creating content types

To activate the page module, you need to follow the instructions in *Installation instructions* and afterwards add `feincms.module.page` to your `INSTALLED_APPS`.

Before proceeding with `manage.py syncdb`, it might be a good idea to take a look at *Page extension modules* – the page module does have the minimum of features in the default configuration and you will probably want to enable several extensions.

You need to create some content models too. No models are created by default, because there is no possibility to unregister models. A sane default might be to create `MediaFileContent` and `RichTextContent` models; you can do this by adding the following lines somewhere into your project, for example in a `models.py` file that will be processed anyway:

```python
from django.utils.translation import ugettext_lazy as _

from feincms.module.page.models import Page
from feincms.content.richtext.models import RichTextContent
from feincms.content.medialibrary.models import MediaFileContent

Page.register_extensions(
    'feincms.module.extensions.datepublisher',
    'feincms.module.extensions.translations'
    ) # Example set of extensions

Page.register_templates({
    'title': _('Standard template'),
    'path': 'base.html',
    'regions': (
        ('main', _('Main content area')),
        ('sidebar', _('Sidebar'), 'inherited'),
        ),
    })

Page.create_content_type(RichTextContent)
Page.create_content_type(MediaFileContent, TYPE_CHOICES=(
    ('default', _('default')),
    ('lightbox', _('lightbox')),
    ))
```

It will be a good idea most of the time to register the `RichTextContent` first, because it's the most used content type for many applications. The content type dropdown will contain content types in the same order as they were registered.

Please note that you should put these statements into a `models.py` file of an app contained in `INSTALLED_APPS`. That file is executed at Django startup time.

1.2.2 Setting up the admin interface

The customized admin interface code is contained inside the `ModelAdmin` subclass, so you do not need to do anything special here.

If you use the `RichTextContent`, you need to download TinyMCE and configure FeinCMS' richtext support:

```
FEINCMS_RICHTEXT_INIT_CONTEXT = {
    'TINYMCE_JS_URL': STATIC_URL + 'your_custom_path/tiny_mce.js',
}
```

1.2.3 Wiring up the views

Just add the following lines to your `urls.py` to get a catch-all URL pattern:

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

If you want to define a page as home page for the whole site, you can give it an `override_url` value of `'/'`.

More information can be found in *Integrating 3rd party apps into your site*

1.2.4 Adding another content type

Imagine you've got a third-party gallery application and you'd like to include excerpts of galleries inside your content. You'd need to write a `GalleryContent` base class and let FeinCMS create a model class for you with some important attributes added.

```python
from django.db import models
from django.template.loader import render_to_string
from feincms.module.page.models import Page
from gallery.models import Gallery


class GalleryContent(models.Model):
    gallery = models.ForeignKey(Gallery)

    class Meta:
        abstract = True # Required by FeinCMS, content types must be abstract

    def render(self, **kwargs):
        return render_to_string('gallery/gallerycontent.html', {
            'content': self, # Not required but a convention followed by
                             # all of FeinCMS' bundled content types
            'images': self.gallery.image_set.order_by('?')[:5],
        })

Page.create_content_type(GalleryContent)
```

The newly created `GalleryContent` for `Page` will live in the database table `page_page_gallerycontent`.

Note: FeinCMS requires your content type model to be abstract.

More information about content types is available in *Content types - what your page content is built of*.

1.2.5 Page extension modules

Extensions are a way to put often-used functionality easily accessible without cluttering up the core page model for those who do not need them. The extensions are standard python modules with a `register()` method which will be called upon registering the extension. The `register()` method receives the `Page` class itself and the model admin class `PageAdmin` as arguments. The extensions can be activated as follows:

```
Page.register_extensions('feincms.module.page.extensions.navigation',
                         'feincms.module.page.extensions.titles',
                         'feincms.module.extensions.translations')
```

The following extensions are available currently:

- `feincms.module.extensions.changedate` — Creation and modification dates

  Adds automatically maintained creation and modification date fields to the page.

- `feincms.module.extensions.ct_tracker` — Content type cache

  Helps reduce database queries if you have three or more content types.

- `feincms.module.extensions.datepublisher` — Date-based publishing

  Adds publication date and end date fields to the page, thereby enabling the administrator to define a date range where a page will be available to website visitors.

- `feincms.module.page.extensions.excerpt` — Page summary

  Add a brief excerpt summarizing the content of this page.

- `feincms.module.extensions.featured` — Simple featured flag for a page

  Lets administrators set a featured flag that lets you treat that page special.

- `feincms.module.page.extensions.navigation` — Navigation extensions

  Adds navigation extensions to the page model. You can define subclasses of `NavigationExtension`, which provide submenus to the navigation generation mechanism. See *Letting 3rd party apps define navigation entries* for more information on how to use this extension.

- `feincms.module.page.extensions.relatedpages` — Links related content

  Add a many-to-many relationship field to relate this page to other pages.

- `feincms.module.extensions.seo` — Search engine optimization

  Adds fields to the page relevant for search engine optimization (SEO), currently only meta keywords and description.

- `feincms.module.page.extensions.sites` — Limit pages to sites

  Allows to limit a page to a certain site and not display it on other sites.

- `feincms.module.page.extensions.symlinks` — Symlinked content extension

  Sometimes you want to reuse all content from a page in another place. This extension lets you do that.

- `feincms.module.page.extensions.titles` — Additional titles

  Adds additional title fields to the page model. You may not only define a single title for the page to be used in the navigation, the <title> tag and inside the content area, you are not only allowed to define different titles for the three uses but also enabled to define titles and subtitles for the content area.

- `feincms.module.extensions.translations` — Page translations

  Adds a language field and a recursive translations many to many field to the page, so that you can define the language the page is in and assign translations. I am currently very unhappy with state of things concerning the definition of translations, so that extension might change somewhat too. This extension also adds new instructions to the setup_request method where the Django i18n tools are initialized with the language given on the page object.

  While it is not required by FeinCMS itself it's still recommended to add `django.middleware.locale.LocaleMiddleware` to the `MIDDLEWARE_CLASSES`; otherwise you will see strange language switching behavior in non-FeinCMS managed views (such as third party apps not integrated using `feincms.content.application.models.ApplicationContent` or Django's own administration tool). You need to have defined `settings.LANGUAGES` as well.

Note: These extension modules add new fields to the `Page` class. If you add or remove page extensions after you've run `syncdb` for the first time you have to change the database schema yourself, or use *Database migration support for FeinCMS with South*.

1.2.6 Using page request processors

A request processor is a function that gets the currently selected page and the request as parameters and returns either None (or nothing) or a HttpResponse. All registered request processors are run before the page is actually rendered. If the request processor indeed returns a `HttpResponse`, further rendering of the page is cut short and this response is returned immediately to the client. It is also possible to raise an exception which will be handled like all exceptions are handled in Django views.

This allows for various actions dependent on page and request, for example a simple user access check can be implemented like this:

```
def authenticated_request_processor(page, request):
    if not request.user.is_authenticated():
        raise django.core.exceptions.PermissionDenied

Page.register_request_processor(authenticated_request_processor)
```

`register_request_processor` has an optional second argument named `key`. If you register a request processor with the same key, the second processor replaces the first. This is especially handy to replace the standard request processors named `path_active` (which checks whether all ancestors of a given page are active too) and `redirect` (which issues HTTP-level redirects if the `redirect_to` page field is filled in).

1.2.7 Using page response processors

Analogous to a request processor, a response processor runs after a page has been rendered. It needs to accept the page, the request and the response as parameters and may change the response (or throw an exception, but try not to).

A response processor is the right place to tweak the returned http response for whatever purposes you have in mind.

```
def set_random_header_response_processor(page, request, response):
    response['X-Random-Number'] = 42

Page.register_response_processor(set_random_header_response_processor)
```

`register_response_processor` has an optional second argument named `key`, exactly like `register_request_processor` above. It behaves in the same way.

1.2.8 WYSIWYG Editors

TinyMCE 3 is configured by default to only allow for minimal formatting. This has proven to be the best compromise between letting the client format text without destroying the page design concept. You can customize the TinyMCE settings by creating your own init_richtext.html that inherits from *admin/content/richtext/init_tinymce.html*. You can even set your own CSS and linklist files like so:

```
FEINCMS_RICHTEXT_INIT_CONTEXT = {
        'TINYMCE_JS_URL': STATIC_URL + 'your_custom_path/tiny_mce.js',
        'TINYMCE_CONTENT_CSS_URL': None,  # add your css path here
        'TINYMCE_LINK_LIST_URL': None  # add your linklist.js path here
}
```

FeinCMS is set up to use TinyMCE 3 but you can use CKEditor instead if you prefer that one. Change the following settings:

```
FEINCMS_RICHTEXT_INIT_TEMPLATE = 'admin/content/richtext/init_ckeditor.html'
FEINCMS_RICHTEXT_INIT_CONTEXT = {
        'CKEDITOR_JS_URL': STATIC_URL + 'path_to_your/ckeditor.js',
}
```

Alternatively, you can also use TinyMCE 4 by changing the following setting:

```
FEINCMS_RICHTEXT_INIT_TEMPLATE = 'admin/content/richtext/init_tinymce4.html'
```

1.2.9 ETag handling

An ETag is a string that is associated with a page – it should change if (and only if) the page content itself has changed. Since a page's content may depend on more than just the raw page data in the database (e.g. it might list its children or a navigation tree or an excerpt from some other place in the CMS alltogether), you are required to write an etag producing method for the page.

```python
# Very stupid etag function, a page is supposed the unchanged as long
# as its id and slug do not change. You definitely want something more
# involved, like including last change dates or whatever.
def my_etag(page, request):
    return 'PAGE-%d-%s' % ( page.id, page.slug )
Page.etag = my_etag

Page.register_request_processors(Page.etag_request_processor)
Page.register_response_processors(Page.etag_response_processor)
```

1.2.10 Sitemaps

To create a sitemap that is automatically populated with all pages in your Feincms site, add the following to your top-level urls.py:

```python
from feincms.module.page.sitemap import PageSitemap
sitemaps = {'pages' : PageSitemap}

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

This will produce a default sitemap at the /sitemap.xml url. A sitemap can be further customised by passing it appropriate parameters, like so:

```python
sitemaps = {'pages': PageSitemap(max_depth=2)}
```

The following parameters can be used to modify the behaviour of the sitemap:

- `navigation_only` – if set to True, only pages that are in_navigation will appear in the site map.
- `max_depth` – if set to a non-negative integer, will limit the sitemap generated to this page hierarchy depth.
- `changefreq` – should be a string or callable specifying the page update frequency, according to the sitemap protocol.
- `queryset` – pass in a query set to restrict the Pages to include in the site map.
- `filter` – pass in a callable that transforms a queryset to filter out the pages you want to include in the site map.
- `extended_navigation` – if set to True, adds pages from any navigation extensions. If using PagePretender, make sure to include title, url, level, in_navigation and optionally modification_date.

1.3 Content types - what your page content is built of

You will learn how to add your own content types and how you can render them in a template.

1.3.1 What is a content type anyway?

In FeinCMS, a content type is something to attach as content to a base model, for example a CMS Page (the base model) may have several rich text components associated to it (those would be RichTextContent content types).

Every content type knows, amongst other things, how to render itself. Think of content types as "snippets" of information to appear on a page.

1.3.2 Rendering contents in your templates

Simple:

```
<div id="content">
    {% block content %}
    {% for content in feincms_page.content.main %}
        {{ content.render }}
    {% endfor %}
    {% endblock %}
</div>

<div id="sidebar">
    {% block sidebar %}
    {% for content in feincms_page.content.sidebar %}
        {{ content.render }}
    {% endfor %}
    {% endblock %}
</div>
```

1.3.3 Implementing your own content types

The minimal content type is an abstract Django model with a `render()` method, nothing else:

```python
class TextileContent(models.Model):
    content = models.TextField()

    class Meta:
        abstract = True

    def render(self, **kwargs):
        return textile(self.content)
```

All content types' `render()` methods must accept `**kwargs`. This allows easily extending the interface with additional parameters. But more on this later.

FeinCMS offers a method on `feincms.models.Base` called `create_content_type()` which will create concrete content types from your abstract content types. Since content types can be used for different CMS base models such as pages and blog entries (implementing a rich text or an image content once and using it for both models makes lots of sense) your implementation needs to be abstract. `create_content_type()` adds a few utility methods and a few model fields to build the concrete type, a foreign key to the base model (f.e. the `Page`) and several properties indicating where the content block will be positioned in the rendered result.

Note: The examples on this page assume that you use the `Page` CMS base model. The principles outlined apply for all other CMS base types.

The complete code required to implement and include a custom textile content type is shown here:

```python
from feincms.module.page.models import Page
from django.contrib.markup.templatetags.markup import textile
from django.db import models
```

```python
class TextilePageContent(models.Model):
    content = models.TextField()

    class Meta:
        abstract = True

    def render(self, **kwargs):
        return textile(self.content)

Page.create_content_type(TextilePageContent)
```

There are three field names you should not use because they are added by `create_content_type`: These are `parent`, `region` and `ordering`. These fields are used to specify the place where the content will be placed in the output.

1.3.4 Customizing the render method for different regions

The default `render` method uses the region key to find a render method in your concrete content type and calls it. This allows you to customize the output depending on the region; you might want to show the same content differently in a sidebar and in the main region for example. If no matching method has been found a `NotImplementedError` is raised.

This `render` method tries to be a sane default, nothing more. You can simply override it and put your own code there if you do not any differentiation, or if you want to do it differently.

All `render` methods should accept `**kwargs`. Some render methods might need the request, for example to determine the correct Google Maps API key depending on the current domain. The two template tags `feincms_render_region` and `feincms_render_content` pass the current rendering context as a keyword argument too.

The example above could be rewritten like this:

```
{% load feincms_tags %}

 <div id="content">
     {% block content %}
     {% for content in feincms_page.content.main %}
         {% feincms_render_content content request %}
     {% endfor %}
     {% endblock %}
 </div>

 <div id="sidebar">
     {% block sidebar %}
     {% for content in feincms_page.content.sidebar %}
         {% feincms_render_content content request %}
     {% endfor %}
     {% endblock %}
 </div>
```

Or even like this:

```
{% load feincms_tags %}

 <div id="content">
     {% block content %}
     {% feincms_render_region feincms_page "main" request %}
     {% endblock %}
 </div>

 <div id="sidebar">
     {% block sidebar %}
```

```
        {% feincms_render_region feincms_page "sidebar" request %}
        {% endblock %}
</div>
```

This does exactly the same, but you do not have to loop over the page content blocks yourself. You need to add the request context processor to your list of context processors for this example to work.

1.3.5 Extra media for content types

Some content types require extra CSS or javascript to work correctly. The content types have a way of individually specifying which CSS and JS files they need. The mechanism in use is almost the same as the one used in form and form widget media.

Include the following code in the *<head>* section of your template to include all JS and CSS media file definitions:

```
{{ feincms_page.content.media }}
```

The individual content types should use a `media` property do define the media files they need:

```python
from django import forms
from django.db import models
from django.template.loader import render_to_string


class MediaUsingContentType(models.Model):
    album = models.ForeignKey('gallery.Album')

    class Meta:
        abstract = True

    @property
    def media(self):
        return forms.Media(
            css={'all': ('gallery/gallery.css',),},
            js=('gallery/gallery.js',),
            )

    def render(self, **kwargs):
        return render_to_string('content/gallery/album.html', {
            'content': self,
            })
```

Please note that you can't define a `Media` inner class (yet). You have to provide the `media` property yourself. As with form and widget media definitions, either `STATIC_URL` or `MEDIA_URL` (in this order) will be prepended to the media file path if it is not an absolute path already.

Alternatively, you can use the `media_property` function from django.forms to implement the functionality, which then also supports inheritance of media files:

```python
from django.forms.widgets import media_property

class MediaUsingContentType(models.Model):
    class Media:
        js = ('whizbang.js',)

MediaUsingContentType.media = media_property(MediaUsingContentType)
```

1.3.6 Influencing request processing through a content type

Since FeinCMS 1.3, content types are not only able to render themselves, they can offer two more entry points which are called before and after the response is rendered. These two entry points are called `process()` and

---

```
finalize().
```

`process()` is called before rendering the template starts. The method always gets the current request as first argument, but should accept `**kwargs` for later extensions of the interface. This method can short-circuit the request-response-cycle simply by returning any response object. If the return value is a `HttpResponse`, the standard FeinCMS view function does not do any further processing and returns the response right away.

As a special case, if a `process()` method returns `True` (for successful processing), `Http404` exceptions raised by any other content type on the current page are ignored. This is especially helpful if you have several `ApplicationContent` content types on a single page.

`finalize()` is called after the response has been rendered. It receives the current request and response objects. This function is normally used to set response headers inside a content type or do some other post-processing. If this function has any return value, the FeinCMS view will return this value instead of the rendered response.

Here's an example form-handling content which uses all of these facilities:

```python
class FormContent(models.Model):
    class Meta:
        abstract = True

    def process(self, request, **kwargs):
        if request.method == 'POST':
            form = FormClass(request.POST)
            if form.is_valid():
                # Do something with form.cleaned_data ...

                return HttpResponseRedirect('?thanks=1')

        else:
            form = FormClass()

        self.rendered_output = render_to_string('content/form.html', {
            'form': form,
            'thanks': request.GET.get('thanks'),
            })
    def render(self, **kwargs):
        return getattr(self, 'rendered_output', u'')

    def finalize(self, request, response):
        # Always disable caches if this content type is used somewhere
        response['Cache-Control'] = 'no-cache, must-revalidate'
```

Note: Please note that the `render` method should not raise an exception if `process` has not been called beforehand.

> **Warning:** The FeinCMS page module views guarantee that `process` is called beforehand, other modules may not do so. `feincms.module.blog` for instance does not.

1.3.7 Bundled content types

Application content

class `feincms.content.application.models.`**`ApplicationContent`**

Used to let the administrator freely integrate 3rd party applications into the CMS. Described in *Integrating 3rd party apps*.

Comments content

class `feincms.content.comments.models.`**`CommentsContent`**

Comment list and form using `django.contrib.comments`.

Contact form content

class `feincms.content.contactform.models.`**`ContactFormContent`**

Simple contact form. Also serves as an example how forms might be used inside content types.

Inline files

class `feincms.content.file.models.`**`FileContent`**

Simple content types holding just a file. You should probably use the MediaFileContent though.

Inline images

class `feincms.content.image.models.`**`ImageContent`**

Simple content types holding just an image with a position. You should probably use the MediaFileContent though.

Additional arguments for `create_content_type()`:

- POSITION_CHOICES
- FORMAT_CHOICES

Media library integration

class `feincms.content.medialibrary.v2.`**`MediaFileContent`**

Mini-framework for arbitrary file types with customizable rendering methods per-filetype. Add 'feincms.module.medialibrary' to INSTALLED_APPS.

Additional arguments for `create_content_type()`:

- TYPE_CHOICES: (mandatory)

  A list of tuples for the type choice radio input fields.

  This field allows the website administrator to select a suitable presentation for a particular media file. For example, images could be shown as thumbnail with a lightbox or offered as downloads. The types should be specified as follows for this use case:

  ```
  ..., TYPE_CHOICES=(('lightbox', _('lightbox')), ('download', _('as download'))),
  ```

  The `MediaFileContent` tries loading the following templates in order for a particular image media file with type `download`:

  - `content/mediafile/image_download.html`
  - `content/mediafile/image.html`
  - `content/mediafile/download.html`
  - `content/mediafile/default.html`

The media file type is stored directly on `MediaFile`.

The file type can also be used to select templates which can be used to further customize the presentation of mediafiles, f.e. `content/mediafile/swf.html` to automatically generate the necessary `<object>` and `<embed>` tags for flash movies.

Raw content

class `feincms.content.raw.models.`**`RawContent`**

Raw HTML code, f.e. for flash movies or javascript code.

Rich text

class `feincms.content.richtext.models.`**`RichTextContent`**

Rich text editor widget, stripped down to the essentials; no media support, only a few styles activated. The necessary javascript files are not included, you need to put them in the right place on your own.

By default, `RichTextContent` expects a TinyMCE activation script at `<MEDIA_URL>js/tiny_mce/tiny_mce.js`. This can be customized by overriding `FEINCMS_RICHTEXT_INIT_TEMPLATE` and `FEINCMS_RICHTEXT_INIT_CONTEXT` in your `settings.py` file.

If you only want to provide a different path to the TinyMCE javascript file, you can do this as follows:

```
FEINCMS_RICHTEXT_INIT_CONTEXT = {
    'TINYMCE_JS_URL': '/your_custom_path/tiny_mce.js',
    }
```

If you pass cleanse=True to the create_content_type invocation for your RichTextContent types, the HTML code will be cleansed right before saving to the database everytime the content is modified.

Additional arguments for `create_content_type()`:

- `cleanse`:

  Whether the HTML code should be cleansed of all tags and attributes which are not explicitly whitelisted. The default is `False`.

RSS feeds

class `feincms.content.rss.models.`**`RSSContent`**

A feed reader widget. This also serves as an example how to build a content type that needs additional processing, in this case from a cron job. If an RSS feed has been added to the CMS, `manage.py update_rsscontent` should be run periodically (either through a cron job or through other means) to keep the shown content up to date. The *feedparser* module is required.

Section content

class `feincms.content.section.models.`**`SectionContent`**

Combined rich text editor, title and media file.

Template content

class `feincms.content.template.models.`**`TemplateContent`**

This is a content type that just includes a snippet from a template. This content type scans all template directories for templates below `content/template/` and allows the user to select one of these templates which are then rendered using the Django template language.

Note that some file extensions are automatically filtered so they will not appear in the list, namely any filenames ending with `.~` or `.tmp` will be ignored.

Also note that a template content is not sandboxed or specially rendered. Whatever a django template can do a TemplateContent snippet can do too, so be careful whom you grant write permissions.

Video inclusion code for youtube, vimeo etc.

class `feincms.content.video.models.`**`VideoContent`**

A easy-to-use content type that automatically generates Flash video inclusion code from a website link. Currently only YouTube and Vimeo links are supported.

1.3.8 Restricting a content type to a subset of regions

Imagine that you have developed a content type which really only makes sense in the sidebar, not in the main content area. It is very simple to restrict a content type to a subset of regions, the only thing you have to do is pass a tuple of region keys to the create_content_type method:

```
Page.create_content_type(SomeSidebarContent, regions=('sidebar',))
```

Note that the restriction only influences the content types shown in the "Add new item"-dropdown in the item editor. The user may still choose to add the SomeSidebarContent to the sidebar, for example, and then proceed to move the content item into the main region.

1.3.9 Design considerations for content types

Because the admin interface is already filled with information, it is sometimes easier to keep the details for certain models outside the CMS content types. Complicated models do not need to be edited directly in the CMS item editor, you can instead use the standard Django administration interface for them, and integrate them into FeinCMS by utilizing foreign keys. Already the bundled FileContent and ImageContent models can be viewed as bad style in this respect, because if you want to use a image or file more than once you need to upload it for every single use instead of being able to reuse the uploaded file. The media library module and MediaFileContent resolve at least this issue nicely by allowing the website administrator to attach metadata to a file and include it in a page by simply selecting the previously uploaded media file.

1.3.10 Configuring and self-checking content types at creation time

So you'd like to check whether Django is properly configured for your content type, or maybe add model/form fields depending on arguments passed at content type creation time? This is very easy to achieve. The only thing you need to do is adding a classmethod named `initialize_type()` to your content type, and pass additional keyword arguments to `create_content_type()`.

If you want to see an example of these two uses, have a look at the `MediaFileContent`.

It is generally recommended to use this hook to configure content types compared to putting the configuration into the site-wide settings file. This is because you might want to configure the content type differently depending on the CMS base model that it is used with.

1.3.11 Obtaining a concrete content type model

The concrete content type models are stored in the same module as the CMS base class, but they do not have a name using which you could import them. Accessing internal attributes is hacky, so what is the best way to get a hold onto the concrete content type?

There are two recommended ways. The example use a `RawContent` content type and the Page CMS base class.

You could take advantage of the fact that `create_content_type` returns the created model:

```python
from feincms.module.page.models import Page
from feincms.content.raw.models import RawContent

PageRawContent = Page.create_content_type(RawContent)
```

Or you could use `content_type_for()`:

```python
from feincms.content.raw.models import RawContent

PageRawContent = Page.content_type_for(RawContent)
```

1.4 Extensions

The extensions mechanism has been refactored to remove the need to make models know about their related model admin classes. The new module `feincms.extensions` contains mixins and base classes - their purpose is as follows:

class feincms.extensions.**ExtensionsMixin**
> This mixin provides the `register_extensions` method which is the place where extensions are registered for a certain model. Extensions can be specified in the following ways:
>
> > •Subclasses of [Extension](#)
> >
> > •Dotted Python module paths pointing to a subclass of the aforementioned extension class
> >
> > •Dotted Python module paths pointing to a module containing either a class named `Extension` or a function named `register` (for legacy extensions)

class feincms.extensions.**Extension**
> This is the base class for your own extension. It has the following methods and properties:
>
> **model**
> > The model class.
>
> **handle_model**(*self*)
> > The method which modifies the Django model class. The model class is available as `self.model`.
>
> **handle_modeladmin**(*self*, *modeladmin*)
> > This method receives the model admin instance bound to the model. This method could be called more than once, especially when using more than one admin site.

class feincms.extensions.**ExtensionModelAdmin**
> This is a model admin subclass which knows about extensions, and lets the extensions do their work modifying the model admin instance after it has been successfully initialized. It has the following methods and properties:
>
> **initialize_extensions**(*self*)
> > This method is automatically called at the end of initialization and loops through all registered extensions and calls their `handle_modeladmin` method.
>
> **add_extension_options**(*self*, **f*)
> > This is a helper to add fields and fieldsets to a model admin instance. Usage is as follows:

```
modeladmin.add_extension_options('field1', 'field2')
```

Or:

```
modeladmin.add_extension_options(_('Fieldset title'), {
    'fields': ('field1', 'field2'),
    })
```

Note: Only model and admin instances which inherit from `ExtensionsMixin` and `ExtensionModelAdmin` can be extended this way.
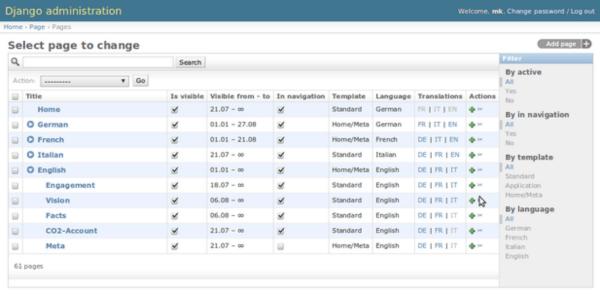
1.5 Administration interfaces

FeinCMS provides two ModelAdmin classes, `ItemEditor`, and `TreeEditor`. Their purpose and their customization hooks are briefly discussed here.

1.5.1 The tree editor

class `feincms.admin.tree_editor.`**`TreeEditor`**

The tree editor replaces the standard change list interface with a collapsible item tree. The model must be registered with [django-mptt](#) for this to work.

Usage is as follows:

```python
from django.db import models
from mptt.fields import TreeForeignKey
from mptt.models import MPTTModel


class YourModel(MPTTModel):
    # model field definitions

    parent = TreeForeignKey('self', null=True, blank=True, related_name='children')

    class Meta:
        ordering = ['tree_id', 'lft']  # The TreeEditor needs this ordering definition
```

And inside your `admin.py` file:

```python
from django.contrib import admin
from feincms.admin import tree_editor
from yourapp.models import YourModel

class YourModelAdmin(tree_editor.TreeEditor):
    pass

admin.site.register(YourModel, YourModelAdmin)
```

All standard `ModelAdmin` attributes such as `ModelAdmin.list_display`, `ModelAdmin.list_editable`, `ModelAdmin.list_filter` work as normally. The only exception to this rule is the column showing the tree structure (the second column in the image). There, we always show the value of `Model.__str__` currently.

AJAX checkboxes

The tree editor allows you to define boolean columns which let the website administrator change the value of the boolean using a simple click on the icon. These boolean columns can be aware of the tree structure. For example, if an object's `active` flag influences the state of its descendants, the tree editor interface is able to show not only the state of the modified element, but also the state of all its descendants without having to reload the page.

Currently, documentation for this feature is not available yet. You can take a look at the implementation of the `is_visible` and `in_navigation` columns of the page editor however.

Usage:

```python
from django.contrib import admin
from feincms.admin import tree_editor
import mptt

class Category(models.Model):
    active = models.BooleanField()
    name = models.CharField(...)
    parent = models.ForeignKey('self', blank=True, null=True)

    # ...
mptt.register(Category)

class CategoryAdmin(tree_editor.TreeEditor):
    list_display = ('__str__', 'active_toggle')
    active_toggle = tree_editor.ajax_editable_boolean('active', _('active'))
```

1.5.2 The item editor

class `feincms.admin.item_editor.`**ItemEditor**

The tabbed interface below is used to edit content and other properties of the edited object. A tab is shown for every region of the template or element, depending on whether templates are activated for the object in question [1].

Here's a screenshot of a content editing pane. The media file content is collapsed currently. New items can be added using the control bar at the bottom, and all content blocks can be reordered using drag and drop:

---

[1] Templates are required for the page module; blog entries managed through the item editor probably won't have a use for them.

Customizing the item editor

New in version 1.2.0.

- The `ItemEditor` now plays nicely with standard Django `fieldsets`; the content-editor is rendered as a replacement for a fieldset with the placeholder name matching `FEINCMS_CONTENT_FIELDSET_NAME`. If no such fieldset is present, one is inserted at the top automatically. If you wish to customise the location of the content-editor, simple include this fieldset at the desired location:

```python
from feincms.admin.item_editor import ItemEditor, FEINCMS_CONTENT_FIELDSET

class MyAdmin(ItemEditor):
    fieldsets = (
        ('Important things', {'fields': ('title', 'slug', 'etc')}),
        FEINCMS_CONTENT_FIELDSET,
        ('Less important things',
            {
                'fields': ('copyright', 'soforth'),
```

```
                'classes': ('collapse',)
            }
        )
    )
```

Customizing the individual content type forms

Customizing the individual content type editors is easily possible through four settings on the content type model itself:

- `feincms_item_editor_context_processors`:

  A list of callables using which you may add additional values to the item editor templates.

- `feincms_item_editor_form`:

  You can specify the base class which should be used for the content type model. The default value is `django.forms.ModelForm`. If you want to customize the form, chances are it is a better idea to set `feincms_item_editor_inline` instead.

- `feincms_item_editor_includes`:

  If you need additional JavaScript or CSS files or need to perform additional initialization on your content type forms, you can specify template fragments which are included in predefined places into the item editor.

  Currently, the only include region available is `head`:

```python
class ContentType(models.Model):
    feincms_item_editor_includes = {
        'head': ['content/init.html'],
        }

    # ...
```

  If you need to execute additional Javascript, for example to add a TinyMCE instance, it is recommended to add the initialization functions to the `contentblock_init_handlers` array, because the initialization needs to be performed not only on page load, but also when adding new content blocks. Please note that these functions *will* be called several times, also several times on the same content types. It is your responsibility to ensure that the handlers aren't attached several times if this would be harmful.

  Additionally, several content types do not support being dragged. Rich text editors such as TinyMCE react badly to being dragged around - they are still visible, but the content disappears and nothing is clickable anymore. Because of this you might want to run routines before and after moving content types around. This is achieved by adding your JavaScript functions to the `contentblock_move_handlers.poorify` array for handlers to be executed before moving and `contentblock_move_handlers.richify` for handlers to be executed after moving. Please note that the item editor executes all handlers on every drag and drop, it is your responsibility to ensure that code is only executed if it has to.

  Take a look at the `richtext` item editor include files to understand how this should be done.

- `feincms_item_editor_inline`:

  New in version 1.4.0.

  This can be used to override the `InlineModelAdmin` class used for the content type. The custom inline should inherit from `FeinCMSInline` or be configured the same way.

  If you override `fieldsets` or `fields` you **must** include `region` and `ordering` even though they aren't shown in the administration interface.

1.5.3 Putting it all together

It is possible to build a limited, but fully functional page CMS administration interface using only the following code (`urls.py` and `views.py` are missing):

`models.py`:

```python
from django.db import models
from mptt.models import MPTTModel
from feincms.models import create_base_model


class Page(create_base_model(MPTTModel)):
    active = models.BooleanField(default=True)
    title = models.CharField(max_length=100)
    slug = models.SlugField()

    parent = models.ForeignKey('self', blank=True, null=True, related_name='children')

    def get_absolute_url(self):
        if self.parent_id:
            return u'%s%s/' % (self.parent.get_absolute_url(), self.slug)
        return u'/%s/' % self.slug
```

`admin.py`:

```python
from django.contrib import admin
from feincms.admin import item_editor, tree_editor
from myapp.models import Page


class PageAdmin(item_editor.ItemEditor, tree_editor.TreeEditor):
    fieldsets = [
        (None, {
            'fields': ['active', 'title', 'slug'],
            }),
        item_editor.FEINCMS_CONTENT_FIELDSET,
        ]
    list_display = ['active', 'title']
    prepopulated_fields = {'slug': ('title',)}
    raw_id_fields = ['parent']
    search_fields = ['title', 'slug']

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

For a more complete (but also more verbose) implementation, have a look at the files inside `feincms/module/page/`.

1.6 Integrating 3rd party apps into your site

With FeinCMS come a set of standard views which you might want to check out before starting to write your own.

1.6.1 Default page handler

The default CMS handler view is `feincms.views.cbv.handler`. You can add the following as last line in your `urls.py` to make a catch-all for any pages which were not matched before:

```python
from feincms.views.cbv.views import Handler
handler = Handler.as_view()

urlpatterns += patterns('',
    url(r'^$', handler, name='feincms_home'),
    url(r'^(.*)/$', handler, name='feincms_handler'),
)
```

Note that this default handler can also take a keyword parameter `path` to specify which url to render. You can use that functionality to implement a default page by adding another entry to your `urls.py`:

```
from feincms.views.cbv.views import Handler
handler = Handler.as_view()

...
    url(r'^$', handler, {'path': '/rootpage'},
        name='feincms_home')
...
```

Please note that it's easier to include `feincms.urls` at the bottom of your own URL patterns like this:

```
# ...

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

The URLconf entry names `feincms_home` and `feincms_handler` must both exist somewhere in your project. The standard `feincms.urls` contains definitions for both. If you want to provide your own view, it's your responsability to create correct URLconf entries.

1.6.2 Generic and custom views

If you use FeinCMS to manage your site, chances are that you still want to use generic and/or custom views for certain parts. You probably still need a `feincms_page` object inside your template to generate the navigation and render regions not managed by the generic views. The best way to ensure the presence of a `feincms_page` instance in the template context is to add `feincms.context_processors.add_page_if_missing` to your `TEMPLATE_CONTEXT_PROCESSORS` setting.

1.6.3 Integrating 3rd party apps

Third party apps such as django-registration can be integrated in the CMS too. `ApplicationContent` lets you delegate a subset of your page tree to a third party application. The only thing you need is specifying a URLconf file which is used to determine which pages exist below the integration point.

Adapting the 3rd party application for FeinCMS

The integration mechanism is very flexible. It allows the website administrator to add the application in multiple places or move the integration point around at will. Obviously, this flexibility puts several constraints on the application developer. It is therefore probable, that you cannot just drop in a 3rd party application and expect it to work. Modifications of `urls.py` and the templates will be required.

The following examples all assume that we want to integrate a news application into FeinCMS. The `ApplicationContent` will be added to the page at `/news/`, but that's not too important really, because the 3rd party app's assumption about where it will be integrated can be too easily violated.

An example `urls.py` follows:

```python
from django.conf.urls import patterns, include, url
from django.views.generic.detail import DetailView
from django.views.generic.list import ListView
from news.models import Entry


urlpatterns = patterns('',
    url(r'^$', ListView.as_view(
        queryset=Entry.objects.all(),
        ), name='entry_list'),
    url(r'^(?P<slug>[^/]+)/$', DetailView.as_view(
        queryset=Entry.objects.all(),
```

```
    ), name='entry_detail'),
)
```

Please note that you should not add the `news/` prefix here. You should *not* reference this `urls.py` file anywhere in a `include` statement.

Registering the 3rd party application with FeinCMS' `ApplicationContent`

It's as simple as that:

```python
from feincms.content.application.models import ApplicationContent
from feincms.module.page.models import Page

Page.create_content_type(ApplicationContent, APPLICATIONS=(
    ('news.urls', 'News application'),
    ))
```

Writing the models

Because the URLconf entries `entry_list` and `entry_detail` aren't reachable through standard means (remember, they aren't `included` anywhere) it's not possible to use standard `reverse` calls to determine the absolute URL of a news entry. FeinCMS provides its own `app_reverse` function (see *More on reversing URLs* for details) and `permalink` decorator mimicking the interface of Django's standard functionality:

```python
from django.db import models
from feincms.content.application import models as app_models


class Entry(models.Model):
    title = models.CharField(max_length=200)
    slug = models.SlugField()
    description = models.TextField(blank=True)

    class Meta:
        ordering = ['-id']

    def __str__(self):
        return self.title

    @app_models.permalink
    def get_absolute_url(self):
        return ('entry_detail', 'news.urls', (), {
            'slug': self.slug,
            })
```

The only difference is that you do not only have to specify the view name (`entry_detail`) but also the URLconf file (`news.urls`) for this specific `permalink` decorator. The URLconf string must correspond to the specification used in the `APPLICATIONS` list in the `create_content_type` call.

Note: Previous FeinCMS versions only provided a monkey patched `reverse` method with a slightly different syntax for reversing URLs. This behavior is still available and as of now (FeinCMS 1.5) still active by default. It is recommended to start using the new way right now and add `FEINCMS_REVERSE_MONKEY_PATCH = False` to your settings file.

Returning content from views

Three different types of return values can be handled by the application content code:

- Unicode data (e.g. the return value of `render_to_string`)

- `HttpResponse` instances

- A tuple consisting of two elements: A template instance, template name or list and a context `dict`. More on this later under *Letting the application content use the full power of Django's template inheritance*

Unicode data is inserted verbatim into the output. `HttpResponse` instances are returned directly to the client under the following circumstances:

- The HTTP status code differs from `200 OK` (Please note that 404 errors may be ignored if more than one content type with a `process` method exists on the current CMS page.)

- The resource was requested by `XmlHttpRequest` (that is, `request.is_ajax` returns `True`)

- The response was explicitly marked as `standalone` by the `feincms.views.decorators.standalone()` view decorator (made easier by mixing-in `feincms.module.mixins.StandaloneView`)

- The mimetype of the response was not `text/plain` or `text/html`

Otherwise, the content of the response is unpacked and inserted into the CMS output as unicode data as if the view returned the content directly, not wrapped into a `HttpResponse` instance.

If you want to customize this behavior, provide your own subclass of `ApplicationContent` with an overridden `send_directly` method. The described behavior is only a sane default and might not fit everyone's use case.

Note: The string or response returned should not contain `<html>` or `<body>` tags because this would invalidate the HTML code returned by FeinCMS.

Letting the application content use the full power of Django's template inheritance

If returning a simple unicode string is not enough and you'd like to modify different blocks in the base template, you have to ensure two things:

- Use the class-based page handler. This is already the default if you include `feincms.urls` or `feincms.views.cbv.urls`.

- Make sure your application views use the third return value type described above: A tuple consisting of a template and a context `dict`.

The news application views would then look as follows. Please note the absence of any template rendering calls:

`views.py`:

```python
from django.shortcuts import get_object_or_404
from news.models import Entry


def entry_list(request):
    # Pagination should probably be added here
    return 'news/entry_list.html', {'object_list': Entry.objects.all()}


def entry_detail(request, slug):
    return 'news/entry_detail', {'object': get_object_or_404(Entry, slug=slug)}
```

`urls.py`:

```python
from django.conf.urls import patterns, include, url

urlpatterns = patterns('news.views',
    url(r'^$', 'entry_list', name='entry_list'),
    url(r'^(?P<slug>[^/]+)/$', 'entry_detail', name='entry_detail'),
)
```

The two templates referenced, `news/entry_list.html` and `news/entry_detail.html`, should now extend a base template. The recommended notation is as follows:

```
{% extends feincms_page.template.path|default:"base.html" %}

{% block ... %}
{# more content snipped #}
```

This ensures that the the selected CMS template is still used when rendering content.

Note: Older versions of FeinCMS only offered fragments for a similar purpose. They are still suported, but it's recommended you switch over to this style instead.

> **Warning:** If you add two application content blocks on the same page and both use this mechanism, the later 'wins'.

More on reversing URLs

Application content-aware URL reversing is available both for Python and Django template code.

The function works almost like Django's own `reverse()` method except that it resolves URLs from application contents. The second argument, `urlconf`, has to correspond to the URLconf parameter passed in the `APPLICATIONS` list to `Page.create_content_type`:

```python
from feincms.content.application.models import app_reverse
app_reverse('mymodel-detail', 'myapp.urls', args=...)
```

or:

```python
app_reverse('mymodel-detail', 'myapp.urls', kwargs=...)
```

The template tag has to be loaded from the `applicationcontent_tags` template tag library first:

```
{% load applicationcontent_tags %}
{% app_reverse "mymodel_detail" "myapp.urls" arg1 arg2 %}
```

or:

```
{% load applicationcontent_tags %}
{% app_reverse "mymodel_detail" "myapp.urls" name1=value1 name2=value2 %}
```

Storing the URL in a context variable is supported too:

```
{% load applicationcontent_tags %}
{% app_reverse "mymodel_detail" "myapp.urls" arg1 arg2 as url %}
```

Inside the app (in this case, inside the views defined in `myapp.urls`), you can also pass the current request instance instead of the URLconf name.

If an application has been added several times to the same page tree, `app_reverse` tries to find the best match. The logic is contained inside `ApplicationContent.closest_match`, and can be overridden by subclassing the application content type. The default implementation only takes the current language into account, which is mostly helpful when you're using the translations page extension.

Additional customization possibilities

The `ApplicationContent` offers additional customization possibilites for those who need them. All of these must be specified in the `APPLICATIONS` argument to `create_content_type`.

- `urls`: Making it easier to swap the URLconf file:

  You might want to use logical names instead of URLconf paths when you create your content types, so that the `ApplicationContent` apps aren't tied to a particular `urls.py` file. This is useful if you

---

want to override a few URLs from a 3rd party application, f.e. replace `registration.urls` with `yourapp.registration_urls`:

```
Page.create_content_type(ApplicationContent, APPLICATIONS=(
   ('registration', 'Account creation and management', {
       'urls': 'yourapp.registration_urls',
       }),
   )
```

- `admin_fields`: Adding more fields to the application content interface:

  Some application contents might require additional configuration parameters which should be modifyable by the website administrator. `admin_fields` to the rescue!

```
def registration_admin_fields(form, *args, **kwargs):
   return {
       'exclusive_subpages': forms.BooleanField(
           label=_('Exclusive subpages'),
           required=False,
           initial=form.instance.parameters.get('exclusive_subpages', True),
           help_text=_('Exclude everything other than the application\'s content when renderin
           ),
       }

Page.create_content_type(ApplicationContent, APPLICATIONS=(
   ('registration', 'Account creation and management', {
       'urls': 'yourapp.registration_urls',
       'admin_fields': registration_admin_fields,
       }),
   )
```

  The form fields will only be visible after saving the `ApplicationContent` for the first time. They are stored inside a JSON-encoded field. The values are added to the template context indirectly when rendering the main template by adding them to `request._feincms_extra_context`.

- `path_mapper`: Customize URL processing by altering the perceived path of the page:

  The applicaton content uses the remainder of the URL to resolve the view inside the 3rd party application by default. This works fine most of the time, sometimes you want to alter the perceived path without modifying the URLconf file itself.

  If provided, the `path_mapper` receives the three arguments, `request.path`, the URL of the current page and all application parameters, and must return a tuple consisting of the path to resolve inside the application content and the path the current page is supposed to have.

  This `path_mapper` function can be used to do things like rewrite the path so you can pretend that an app is anchored deeper than it actually is (e.g. /path/to/page is treated as "/<slug>/" using a parameter value rather than "/" by the embedded app)

- `view_wrapper`: Decorate every view inside the application content:

  If the customization possibilites above aren't sufficient, `view_wrapper` can be used to decorate each and every view inside the application content with your own function. The function specified with `view_wrapper` receives an additional parameters besides the view itself and any arguments or keyword arguments the URLconf contains, `appcontent_parameters` containing the application content configuration.

Letting 3rd party apps define navigation entries

Short answer: You need the `feincms.module.page.extensions.navigation` extension module. Activate it like this:

```
Page.register_extensions('feincms.module.page.extensions.navigation')
```

Please note however, that this call needs to come after all `NavigationExtension` subclasses have been processed, because otherwise they will not be available for selection in the page administration! (Yes, this is lame and yes, this is going to change as soon as we find a better solution. In the meantime, stick your subclass definition before the register_extensions call.)

Because the use cases for extended navigations are so different, FeinCMS does not go to great lengths trying to cover them all. What it does though is to let you execute code to filter, replace or add navigation entries when generating a list of navigation entries.

If you have a blog and you want to display the blog categories as subnavigation entries, you could do it as follows:

1. Create a navigation extension for the blog categories

2. Assign this navigation extension to the CMS page where you want these navigation entries to appear

You don't need to do anything else as long as you use the built-in `feincms_nav` template tag – it knows how to handle extended navigations.

```python
from feincms.module.page.extensions.navigation import NavigationExtension, PagePretender


class BlogCategoriesNavigationExtension(NavigationExtension):
    name = _('blog categories')

    def children(self, page, **kwargs):
        for category in Category.objects.all():
            yield PagePretender(
                title=category.name,
                url=category.get_absolute_url(),
                )


class PassthroughExtension(NavigationExtension):
    name = 'passthrough extension'

    def children(self, page, **kwargs):
        for p in page.children.in_navigation():
            yield p

Page.register_extensions('feincms.module.page.extensions.navigation')
```

Note that the objects returned should at least try to mimic a real page so navigation template tags as `siblings_along_path_to` and friends continue to work, ie. at least the following attributes should exist:

```python
title    = '(whatever)'
url      = '(whatever)'

# Attributes that MPTT assumes to exist
parent_id = page.id
tree_id   = page.tree_id
level     = page.level+1
lft       = page.lft
rght      = page.rght
```

1.7 Media library

The media library module provides a way to store, transform and display files of arbitrary types.

The following instructions assume, that you use the media library together with the page module. However, the media library does not depend on any aspect of the page module – you can use it with any CMS base model.

To activate the media library and use it together with the page module, it is best to first get the page module working with a few content types. Afterwards, add `feincms.module.medialibrary` to your `INSTALLED_APPS` setting, and create a content type for a media file as follows:

```
from feincms.module.page.models import Page
from feincms.content.medialibrary.v2 import MediaFileContent

Page.create_content_type(MediaFileContent, TYPE_CHOICES=(
        ('default', _('default')),
        ('lightbox', _('lightbox')),
        ))
```

`TYPE_CHOICES` has nothing to do with file types – it's about choosing the presentation type for a certain media file, f.e. whether the media file should be presented inline, in a lightbox, floated, or simply as a download link.

1.7.1 Configuration

The location and URL of the media library may be configured either by setting the appropriate variables in your `settings.py` file or in your CMS defining module.

The file system path for all media library files is defined using Django's `MEDIA_ROOT` setting and FeinCMS' `FEINCMS_MEDIALIBRARY_UPLOAD_TO` setting which defaults to `medialibrary/%Y/%m/`.

These settings can also be changed programmatically using `MediaFile.reconfigure(upload_to=..., storage=...)`

1.7.2 Rendering media file contents

A set of recognition functions will be run on the file name to determine the file type. Using combinations of the name and type, the default render method tries to find a template for rendering the `MediaFileContent`.

The default set of pre-defined content types and recognition functions is:

```
MediaFileBase.register_filetypes(
        ('image', _('Image'), lambda f: re.compile(r'\.(bmp|jpe?g|jp2|jxr|gif|png|tiff?)$', re.IG
        ('video', _('Video'), lambda f: re.compile(r'\.(mov|m[14]v|mp4|avi|mpe?g|qt|ogv|wmv)$', re
        ('audio', _('Audio'), lambda f: re.compile(r'\.(au|mp3|m4a|wma|oga|ram|wav)$', re.IGNORECA
        ('pdf', _('PDF document'), lambda f: f.lower().endswith('.pdf')),
        ('swf', _('Flash'), lambda f: f.lower().endswith('.swf')),
        ('txt', _('Text'), lambda f: f.lower().endswith('.txt')),
        ('rtf', _('Rich Text'), lambda f: f.lower().endswith('.rtf')),
        ('zip', _('Zip archive'), lambda f: f.lower().endswith('.zip')),
        ('doc', _('Microsoft Word'), lambda f: re.compile(r'\.docx?$', re.IGNORECASE).search(f)),
        ('xls', _('Microsoft Excel'), lambda f: re.compile(r'\.xlsx?$', re.IGNORECASE).search(f)),
        ('ppt', _('Microsoft PowerPoint'), lambda f: re.compile(r'\.pptx?$', re.IGNORECASE).searc
        ('other', _('Binary'), lambda f: True), # Must be last
    )
```

You can add to that set by calling `MediaFile.register_filetypes()` with your new file types similar to the above.

If we've got an example file `2009/06/foobar.jpg` and a presentation type of `inline`, the templates tried to render the media file are the following:

- `content/mediafile/image_inline.html`
- `content/mediafile/image.html`
- `content/mediafile/inline.html`
- `content/mediafile/default.html`

You are of course free to do with the file what you want inside the template, for example a thumbnail and a lightbox version of the image file, and put everything into an element that's floated to the left.

1.7.3 Media file metadata

Sometimes, just storing media files is not enough. You've got captions and copyrights which you'd like to store alongside the media file. This media library allows that. The caption may even be translated into different languages. This is most often not necessary or does not apply to copyrights, therefore the copyright can only be entered once, not once per language.

The default image template `content/mediafile/image.html` demonstrates how the values of those fields can be retrieved and used.

1.7.4 Using the media library in your own apps and content types

There are a few helpers that allow you to have a nice raw_id selector and thumbnail preview in your own apps and content types that have a ForeignKey to `MediaFile`.

To have a thumbnail preview in your ModelAdmin and Inline class:

```python
from feincms.module.medialibrary.fields import MediaFileForeignKey


class ImageForProject(models.Model):
    project = models.ForeignKey(Project)
    mediafile = MediaFileForeignKey(MediaFile, related_name='+',
                                    limit_choices_to={'type': 'image'})
```

For the maginfying-glass select widget in your content type inherit your inline from FeinCMSInline:

```python
class MyContentInline(FeinCMSInline):
    raw_id_fields = ('mediafile',)


class MyContent(models.Model):
    feincms_item_editor_inline = MyContentInline
```

1.8 Template tags

1.8.1 General template tags

To use the template tags described in this section, you need to load the `feincms_tags` template tag library:

```
{% load feincms_tags %}
```

feincms_render_region:

feincms_render_content:
> Some content types will need the request object to work properly. Contact forms will need to access POSTed data, a Google Map content type needs to use a different API key depending on the current domain etc. This means you should add `django.core.context_processors.request` to your `TEMPLATE_CONTEXT_PROCESSORS`.
>
> These two template tags allow you to pass the request from the template to the content type. `feincms_render_content()` allows you to surround the individual content blocks with custom markup, `feincms_render_region()` simply concatenates the output of all content blocks:
>
> ```
> {% load feincms_tags %}
>
> {% feincms_render_region feincms_page "main" request %}
> ```
>
> or::
>
> ```
> {% load feincms_tags %}
> ```

```
{% for content in feincms_page.content.main %}
    <div class="block">
        {% feincms_render_content content request %}
    </div>
{% endfor %}
```

Both template tags add the current rendering context to the `render` method call too. This means that you can access both the request and the current context inside your content type as follows:

```python
class MyContentType(models.Model):
    # class Meta etc...

    def render(self, **kwargs):
        request = kwargs.get('request')
        context = kwargs.get('context')
```

feincms_frontend_editing:

1.8.2 Page module-specific template tags

All page module-specific template tags are contained in `feincms_page_tags`:

```
{% load feincms_page_tags %}
```

feincms_nav:
    Return a list of pages to be used for the navigation

    level: 1 = toplevel, 2 = sublevel, 3 = sub-sublevel depth: 1 = only one level, 2 = subpages too

    If you set depth to something else than 1, you might want to look into the `tree_info` template tag from the mptt_tags library.

    Example:

```
{% load feincms_page_tags %}

{% feincms_nav feincms_page level=2 depth=1 as sublevel %}
{% for p in sublevel %}
    <a href="{{ p.get_absolute_url }}">{{ p.title }}</a>
{% endfor %}
```

siblings_along_path_to:
    This is a filter designed to work in close conjunction with the `feincms_nav` template tag describe above to build a navigation tree following the path to the current page.

    Example:

```
{% feincms_nav feincms_page level=1 depth=3 as navitems %}
{% with navitems|siblings_along_path_to:feincms_page as navtree %}
    {% recursetree navtree %}
        * {{ node.short_title }} <br>
            {% if children %}
                <div style="margin-left: 20px">{{ children }}</div>
            {% endif %}
    {% endrecursetree %}
{% endwith %}
```

    For helper function converting a tree of pages into an HTML representation please see the mptt_tags library's `tree_info` and `recursetree`.

feincms_parentlink:
    Return a link to an ancestor of the passed page.

    You'd determine the link to the top level ancestor of the current page like this:

```
{% load feincms_page_tags %}
```

```
{% feincms_parentlink of feincms_page level=1 %}
```

Please note that this is not the same as simply getting the URL of the parent of the current page.

feincms_languagelinks:
This template tag needs the translations extension.

Arguments can be any combination of:

- `all` or `existing`: Return all languages or only those where a translation exists

- `excludecurrent`: Excludes the item in the current language from the list

The default behavior is to return an entry for all languages including the current language.

Example:

```
{% load feincms_page_tags %}
```

```
{% feincms_languagelinks for feincms_page as links all,excludecurrent %}
{% for key, name, link in links %}
    <a href="{% if link %}{{ link }}{% else %}/{{ key }}/{% endif %}">{% trans name %}</a>
{% endfor %}
```

feincms_translatedpage:
This template tag needs the translations extension.

Returns the requested translation of the page if it exists. If the language argument is omitted the primary language will be returned (the first language specified in settings.LANGUAGES):

```
{% load feincms_page_tags %}
```

```
{% feincms_translatedpage for feincms_page as feincms_transpage language=en %}
{% feincms_translatedpage for feincms_page as originalpage %}
{% feincms_translatedpage for some_page as translatedpage language=feincms_page.language %}
```

feincms_translatedpage_or_base:
This template tag needs the translations extensions.

Similar in function and arguments to feincms_translatedpage, but if no translation for the requested language exists, the base language page will be returned:

```
{% load feincms_page_tags %}
```

```
{% feincms_translatedpage_or_base for some_page as some_transpage language=gr %}
```

feincms_breadcrumbs:

```
{% load feincms_page_tags %}
```

```
{% feincms_breadcrumbs feincms_page %}
```

is_parent_of:

```
{% load feincms_page_tags %}
```

```
{% if page1|is_parent_of:page2 %}
    page1 is a parent of page2
{% endif %}
```

is_equal_or_parent_of:

```
{% load feincms_page_tags %}

{% feincms_nav feincms_page level=1 as main %}
{% for entry in main %}
    <a {% if entry|is_equal_or_parent_of:feincms_page %}class="mark"{% endif %}
        href="{{ entry.get_absolute_url }}">{{ entry.title }}</a>
{% endfor %}
```

page_is_active:
>   The advantage of `page_is_active` compared to the previous tags is that it also nows how to han-
>   dle page pretenders. If `entry` is a page pretender, the template tag returns `True` if the current
>   path starts with the page pretender's path. If `entry` is a regular page, the logic is the same as in
>   `is_equal_or_parent_of`.

```
{% load feincms_page_tags %}
{% feincms_nav feincms_page level=1 as main %}
{% for entry in main %}
    {% page_is_active entry as is_active %}
    <a {% if is_active %}class="mark"{% endif %}
        href="{{ entry.get_absolute_url }}">{{ entry.title }}</a>
{% endfor %}
```

>   The values of `feincms_page` (the current page) and the current path are pulled out of the context variables
>   `feincms_page` and `request`. They can also be overriden if you so require:

```
{% page_is_active entry feincms_page=something path=request.path %}
```

1.8.3 Application content template tags

app_reverse:
>   Returns an absolute URL for applications integrated with ApplicationContent
>
>   The tag mostly works the same way as Django's own {% url %} tag:

```
{% load applicationcontent_tags %}
{% app_reverse "mymodel_detail" "myapp.urls" arg1 arg2 %}
```

>   or:

```
{% load applicationcontent_tags %}
{% app_reverse "mymodel_detail" "myapp.urls" name1=value1 name2=value2 %}
```

>   The first argument is a path to a view. The second argument is the URLconf under which this app is known
>   to the ApplicationContent.
>
>   Other arguments are space-separated values that will be filled in place of positional and keyword arguments
>   in the URL. Don't mix positional and keyword arguments.
>
>   If you want to store the URL in a variable instead of showing it right away you can do so too:

```
{% app_reverse "mymodel_detail" "myapp.urls" arg1 arg2 as url %}
```

fragment:

get_fragment:
>   Don't use those, read up on *Letting the application content use the full power of Django's template inheri-
>   tance* instead.

1.9 Database migration support for FeinCMS with South

If you don't know what South is you should probably go and read about it right now!

FeinCMS itself does not come with any migrations. It does not have to: Its core models haven't changed for several versions now. This does not mean South isn't supported! You are free to use South to manage FeinCMS' models which is a very useful technique especially if you are using *Page extension modules*.

The following steps should be sufficient to get up and running with South in your project:

- Put a copy of South somewhere on your PYTHONPATH, with pip, hg or whatever pleases you most.

- Add 'south' to INSTALLED_APPS.

- Create a new folder in your app with an empty __init__.py file inside, e.g. yourapp/migrate/.

- Add the following configuration variable to your settings.py:

```
SOUTH_MIGRATION_MODULES = {
    'page': 'yourapp.migrate.page',
    'medialibrary': 'yourapp.migrate.medialibrary', # if you are using the medialibrary
                                                     # which comes with FeinCMS
}
```

- Run ./manage.py convert_to_south page and ./manage.py convert_to_south medialibrary

- That's it!

> **Warning:** You **must not** use migrations as folder name for the FeinCMS migrations, otherwise South **will** get confused.

1.10 Versioning database content with `django-reversion`

The following steps should be followed to integrate the page module with django-reversion:

- Add 'reversion' to the list of installed applications.

- Add 'reversion.middleware.RevisionMiddleware' to MIDDLEWARE_CLASSES.

- Call Page.register_with_reversion() after all content types have been created (after all create_content_type invocations).

Now, you need to create your own model admin subclass inheriting from both FeinCMS' PageAdmin and from reversions VersionAdmin:

```python
from django.contrib import admin
from feincms.module.page.models import Page
from feincms.module.page.modeladmins import PageAdmin
from reversion.admin import VersionAdmin

admin.site.unregister(Page)


class VersionedPageAdmin(PageAdmin, VersionAdmin):
    pass

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

The VersionedPageAdmin does not look like the ItemEditor – it's just raw Django inlines, without any additional JavaScript. Patches are welcome, but the basic functionality needed for versioning page content is there.

Finally, you should ensure that initial revisions are created using django-reversion's createinitialrevisions management command.

> **Note:** You should ensure that you're using a reversion release which is compatible with your installed Django version. The reversion documentation contains an up-to-date list of compatible releases.

The reversion support in FeinCMS requires at least django-reversion 1.6.

1.11 Advanced topics

This section is targeted at more advanced users of FeinCMS. It goes into details which are not relevant for you if you only want to use the `page` module or the media library on your site.

However, if you want to understand the inner workings of the CMS, the design considerations and how to optimize your code, this section is for you.

1.11.1 `feincms.models.Base` — CMS base class

This is the base class which you must inherit if you'd like to use the CMS to manage content with the `ItemEditor`.

Base.**register_templates**(*templates*)

Base.**register_regions**(*regions*)

Base.**content**
> Beware not to name subclass field *content* as this will overshadow *ContentProxy* and you will not be able to reference *ContentProxy*.

Base.**create_content_type**(*model*, *regions=None*[, ***kwargs*])

Base.**content_type_for**(*model*)

Base.**copy_content_from**(*obj*)

Base.**replace_content_with**(*obj*)

Base.**append_content_from**(*obj*)

1.11.2 `feincms.utils` — General utilities

feincms.utils.**get_object**(*path*[, *fail_silently*])
> Helper function which can be used to import a python object. `path` should be the absolute dotted path to the object. You can optionally pass `fail_silently=True` if the function should not raise an `Exception` in case of a failure to import the object:

> ```
> MyClass = get_object('module.MyClass')
> ```

> ```
> myfunc = get_object('anothermodule.module2.my_function', fail_silently=True)
> ```

feincms.utils.**collect_dict_values**(*data*)
> Converts a list of 2-tuples to a dict.

1.11.3 Software design considerations

These are assorted ramblings copy-pasted from various emails.

About rich text editors

We have been struggling with rich text editors for a long time. To be honest, I do not think it was a good idea to add that many features to the rich text editor. Resizing images uploaded into a rich text editor is a real pain, and what if you'd like to reuse these images or display them using a lightbox script or something similar? You have to resort to writing loads of JavaScript code which will only work on one browser. You cannot really filter the

HTML code generated by the user to kick out ugly HTML code generated by copy-pasting from word. The user will upload 10mb JPEGs and resize them to 50x50 pixels in the rich text editor.

All of this convinced me that offering the user a rich text editor with too much capabilities is a really bad idea. The rich text editor in FeinCMS only has bold, italic, bullets, link and headlines activated (and the HTML code button, because that's sort of inevitable – sometimes the rich text editor messes up and you cannot fix it other than going directly into the HTML code. Plus, if someone really knows what he's doing, I'd still like to give them the power to shot their own foot).

If this does not seem convincing you can always add your own rich text content type with a different configuration (or just override the rich text editor initialization template in your own project). We do not want to force our world view on you, it's just that we think that in this case, more choice has the bigger potential to hurt than to help.

Content blocks

Images and other media files are inserted via objects; the user can only select a file and a display mode (f.e. float/block for images or something...). A page's content could look like this:

- Rich Text
- Floated image
- Rich Text
- YouTube Video Link, embedding code is automatically generated from the link
- Rich Text

It's of course easier for the user to start with only a single rich text field, but I think that the user already has too much confusing possibilities with an enhanced rich text editor. Once the user grasps the concept of content blocks which can be freely added, removed and reordered using drag/drop, I'd say it's much easier to administer the content of a webpage. Plus, the content blocks can have their own displaying and updating logic; implementing dynamic content inside the CMS is not hard anymore, on the contrary. Since content blocks are Django models, you can do anything you want inside them.

1.11.4 Performance considerations

While FeinCMS in its raw form is perfectly capable of serving out a medium sized site, more complicated setups quickly lead to death by database load. As the complexity of your pages grows, so do the number of database queries needed to build page content on each and every request.

It is therefore a good idea to keep an eye open for excessive database queries and to try to avoid them.

Denormalization

FeinCMS comes bundled with the "ct_tracker" extension that will reduce the number of database queries needed by keeping some bookkeeping information duplicated in the base type.

Caching

Caching rendered page fragments is probably the most efficient way of reducing database accesses in your FeinCMS site. An important consideration in the design of your site's templates is which areas of your pages depend on which variables. FeinCMS supplies a number of helper methods and variables, ready to be used in your templates.

Here's an (incomplete) list of variables to use in {% cache %} blocks [2]:

---

[2] Please see the django documentation for detailed description of the {% cache %} template tag.

- **feincms_page.cache_key – a string describing the current page.** Depending on the extensions loaded, this varies with the page, the page's modification date, its language, etc. This is always a safe bet to use on page specific fragments.

- **LANGUAGE_CODE – even if two requests are asking for the same page,** the html code rendered might differ in translated elements in the navigation or elsewhere. If the fragment varies on language, include LANGUAGE_CODE in the cache specifier.

- **request.user.id – different users might be allowed to see different** views of the site. Add request.user.id to the cache specifier if this is the case.

1.12 Frequently Asked Questions

This FAQ serves two purposes. Firstly, it does what a FAQ generally does – answer frequently asked questions. Secondly, it is also a place to dump fragments of documentation which haven't matured enough to be moved into their own documentation file.

1.12.1 Should I extend the builtin modules and contents, or should I write my own?

The answer is, as often, the nearly useless "It depends". The built-in modules serve two purposes: On one hand, they should be ready to use and demonstrate the power of FeinCMS. On the other hand, they should be simple enough to serve as examples for you if you want to build your own CMS-like system using the tools provided by FeinCMS.

If a proposed feature greatly enhances the modules' or content types' abilities without adding heaps of code, chances are pretty good that it will be accepted into FeinCMS core. Anyway, the tools included should be so easy to use that you might still want to build your own page CMS, if your needs are very different from those of the original authors. If you don't like monkey patching at all, or if the list of extensions you want to use grows too big, it might be time to reconsider whether you really want to use the extension mechanism or if it might not be easier to start freshly, only using the editor admin classes, feincms.models.Base and maybe parts of the included PageManager...

1.12.2 I run `syncdb` and get a message about missing columns in the page table

You enabled the page module (added `feincms.module.page` to `INSTALLED_APPS`), run syncdb, and afterwards registered a few extensions. The extensions you activated (`datepublisher` and `translations`) add new fields to the page model, but your first `syncdb` did not know about them and therefore did not create the columns for those extensions.

You can either remove the line `Page.register_extensions(...)` from your code or drop the page_page table and re-run `syncdb`. If you want to keep the pages you've already created, you need to figure out the correct ALTER TABLE statements for your database yourself.

1.13 Contributing to the development of FeinCMS

1.13.1 Repository branches

The FeinCMS repository on github has several branches. Their purpose and rewinding policies are described below.

- `maint`: Maintenance branch for the second-newest version of FeinCMS.
- `master`: Stable version of FeinCMS.

`master` and `maint` are never rebased or rewound.

- `next`: Upcoming version of FeinCMS. This branch is rarely rebased if ever, but this might happen. A note will be sent to the official mailing list whenever `next` has been rebased.

- `pu` or feature branches are used for short-lived projects. These branches aren't guaranteed to stay around and are not meant to be deployed into production environments.

1.14 FeinCMS Deprecation Timeline

This document outlines when various pieces of FeinCMS will be removed or altered in backward incompatible way. Before a feature is removed, a warning will be issued for at least two releases.

1.14.1 1.6

- The value of `FEINCMS_REVERSE_MONKEY_PATCH` has been changed to `False`.

- Deprecated page manager methods have been removed (`page_for_path_or_404`, `for_request_or_404`, `best_match_for_request`, `from_request`) - `Page.objects.for_request()`, `Page.objects.page_for_path` and `Page.objects.best_match_for_path` should cover all use cases.

- Deprecated page methods have been removed (`active_children`, `active_children_in_navigation`, `get_siblings_and_self`)

- Request and response processors have to be imported from `feincms.module.page.processors`. Additionally, they must be registered individually by using `register_request_processor` and `register_response_processor`.

- Prefilled attributes have been removed. Use Django's `prefetch_related` or `feincms.utils.queryset_transform` instead.

- `feincms.views.base` has been moved to `feincms.views.legacy`. Use `feincms.views.cbv` instead.

- `FEINCMS_FRONTEND_EDITING`'s default has been changed to `False`.

- The code in `feincms.module.page.models` has been split up. The admin classes are in `feincms.module.page.modeladmin`, the forms in `feincms.module.page.forms` now. Analogous changes have been made to `feincms.module.medialibrary.models`.

1.14.2 1.7

- The monkeypatch to make Django's `django.core.urlresolvers.reverse()` applicationcontent-aware will be removed. Use `feincms.content.application.models.app_reverse()` and the corresponding template tag instead.

- The module `feincms.content.medialibrary.models` will be replaced by the contents of `feincms.content.medialibrary.v2`. The latter uses Django's `raw_id_fields` support instead of reimplementing it badly.

- The legacy views inside `feincms.views.legacy` will be removed.

1.14.3 1.8

- The module `feincms.admin.editor` will be removed. The model admin classes have been available in `feincms.admin.item_editor` and `feincms.admin.tree_editor` since FeinCMS v1.0.

- Cleansing the HTML of a rich text content will still be possible, but the cleansing module `feincms.utils.html.cleanse` will be removed. When creating a rich text content, the `cleanse`

argument must be a callable and cannot be `True` anymore. The cleansing function has been moved into its own package, feincms-cleanse.

- Registering extensions using shorthand notation will be not be possible in FeinCMS v1.8 anymore. Use the following method instead:

```
Page.register_extensions(
    'feincms.module.page.extensions.navigation',
    'feincms.module.extensions.ct_tracker',
    )
```

- `feincms_navigation` and `feincms_navigation_extended` will be removed. Their functionality is provided by `feincms_nav` instead.

- The function-based generic views aren't available in Django after v1.4 anymore. `feincms.views.generic` and `feincms.views.decorators.add_page_to_extra_context()` will be removed as well.

- The module `feincms.content.medialibrary.v2`, which is only an alias for `feincms.content.medialibrary.models` starting with FeinCMS v1.7 will be removed.

- `Page.setup_request()` does not do anything anymore and will be removed.

1.14.4 1.9

- Fields added through page extensions which haven't been explicitly added to the page model admin using `modeladmin.add_extension_options` will disappear from the admin interface. The automatic collection of fields will be removed.

- All extensions should inherit from `feincms.extensions.Extension`. Support for `register(cls, admin_cls)`-style functions will be removed in FeinCMS v1.9.

- The `_feincms_extensions` attribute on the page model and on models inheriting `ExtensionsMixin` is gone.

Releases

2.1 FeinCMS 1.2 release notes

Welcome to the first release notes for FeinCMS!

2.1.1 Overview

FeinCMS 1.2 sports several large changes, including:

- Overhauled item editor. The new item editor uses standard Django administration fieldsets; you can use almost all standard Django configuration mechanisms. `show_on_top` has been deprecated, standard `fieldsets` should be used instead.

- The split pane editor has been removed. It wasn't much more than a proof of concept and was never bug-free.

- The required Django version is now 1.2. Compatibility with older Django versions has been removed.

- The rich text configuration has slightly changed; `CkRichTextContent` has been completely removed in favor of a rich text editor agnostic configuration method. `TINYMCE_JS_URL` should be replaced by an appropriate `FEINCMS_RICHTEXT_INIT_CONTEXT` settings value. See the *Content types - what your page content is built of* documentation for more details.

- A new content type, `TemplateContent` has been added which can be used to render templates residing on the hard disk.

- The `TreeEditor` JavaScript code has been rewritten, reintroducing drag-drop for reordering pages, but this time in a well-performing way not sluggish as before.

- `feincms.models.Base` is still available, `feincms.models.create_base_model` is the more flexible way of creating the aforementioned base model. If `create_base_model` is used the base model can be freely defined.

- Many small improvements and bugfixes all over the place.

2.2 FeinCMS 1.3 release notes

FeinCMS 1.3 includes many bugfixes and cleanups and a number of new features. The cleanups and features caused a few backwards incompatible changes. The upgrade path is outlined below.

2.2.1 Highlights

- FeinCMS pages use the standard Django permalink mechanism inside the `get_absolute_url` implementation. This means that you have to update the URL definition if you did not include `feincms.urls`

directly.

Change this:

```
url(r'^$|^(.*)/$', 'feincms.views.base.handler'),
```

to this:

```
url(r'', include('feincms.urls')),
```

Defining the URL patterns directly is still possible. Have a look at `feincms.urls` to find out how this should be done.

- FeinCMS requires at least Django 1.2 but already has support for Django 1.3 features such as staticfiles. The FeinCMS media file folder has been moved from `feincms/media/feincms` to `feincms/static/feincms` - if you use `django.contrib.staticfiles` with Django 1.3 (and you should!), FeinCMS' media files for the administration interface will automatically be made available without any further work on your part.

- Content types can specify the media files (Javascript and CSS files) they need to work correctly. See *Extra media for content types* for information on how to use this in your own content types.

- The content type loading process has been streamlined and requires much less database queries than before. The performance hit on sites with deep page hierarchies, inheritance and many regions is several times smaller than before.

- The content type interface has been extended with two new methods, available for all content types which need it: `process` is called before rendering pages and is guaranteed to receive the current request instance. Each and every content type (not only application contents as before) has the ability to return full HTTP responses which are returned directly to the user. `finalize` is called after rendering and can be used to set HTTP headers and do other post-processing tasks. See *Influencing request processing through a content type* for more information.

2.2.2 (Backwards incompatible and other) Changes

- The default `ContentProxy` has been rewritten to load all content type instances on initialization. The instances stay around for the full request-response cycle which allows us to remove many quasi-global variables (variables attached to the `request` object). The new initialization is much more efficient in terms of SQL queries needed; the implementation is contained inside the `ContentProxy` class and not distributed all over the place.

- The `ContactFormContent` has been updated to take advantage of the new content type interface where content types can influence the request-response cycle in more ways.

- The `ct_tracker` extension has been rewritten to take advantage of the new `ContentProxy` features. This means that the format of `_ct_inventory` could not be kept backwards compatible and has been changed. The inventory is versioned now, therefore upgrading should not require any action on your part.

- `feincms_site` is not available in the context anymore. It was undocumented, mostly unused and badly named anyway. If you still need this functionality you should use `django.contrib.sites` directly yourself.

- The `_feincms_appcontent_parameters` has been folded into the `_feincms_extra_context` attribute on the current request. The `appcontent_parameters` template tag is not necessary anymore (the content of `_feincms_extra_context` is guaranteed to be available in the template context) and has been removed.

  In your appcontent code, change all references of `_feincms_appcontent_parameters` to `_feincms_extra_context`, e.g.

  params = getattr(request, '_feincms_appcontent_parameters', {})

  becomes

  params = getattr(request, '_feincms_extra_context', {})

- As part of the effort to reduce variables attached to the request object (acting as a replacement for global variables), `request.extra_path` has been removed. The same information can be accessed via `request._feincms_extra_context['extra_path']`.

- The `feincms.views.applicationcontent` module has been removed. The special casing it provided for application content-using pages aren't necessary anymore.

- The page's `get_absolute_url` method uses URL reversion for determining the URL of pages instead of returning `_cached_url`. This means that you need to modify your URLconf entries if you added them to your own `urls.py` instead of including `feincms.urls`. Please make sure that you have two named URL patterns, `feincms_home` and `feincms_handler`:

```python
from feincms.views.base import handler

urlpatterns = patterns('',
    # ... your patterns ...

    url(r'^$', handler, name='feincms_home'),
    url(r'^(.*)/$', handler, name='feincms_handler'),
)
```

  If you want the old behavior back, all you need to do is add the following code to your `settings.py`:

```python
ABSOLUTE_URL_OVERRIDES = {
    'page.page': lambda page: page._cached_url,
    }
```

- The copy/replace and preview mechanisms never worked quite right. They were completely dropped from this release. If you still need the ability to create copies of objects, use the standard Django `ModelAdmin.save_as` feature.

2.3 FeinCMS 1.4 release notes

- FeinCMS supports more than one site from the same database with `django.contrib.sites` now. Thanks to Bojan Mihelac and Stephen Tyler for the work and insistence on this issue.

- It is possible to customize the administration model inline used for content types. This means that it's possible to customize more aspects of content type editing and to reuse more behaviors from Django itself, such as `raw_id_fields`.

- FeinCMS has gained support for `django-reversion`.

- Reusing the media library in your own content types has become much easier than before. When using a `feincms.module.medialibrary.fields.MediaFileForeignKey` instead of the standard `django.db.models.ForeignKey` and adding the media file foreign key to `raw_id_fields`, you get the standard Django behavior supplemented with a thumbnail if the media file is an image. This requires the next feature too, which is...

- Custom `InlineModelAdmin` classes may be used for the content types now by adding a `feincms_item_editor_inline` attribute to the content type specifying the inline class to be used.

- New projects should use `feincms.content.medialibrary.v2.MediaFileContent` instead of `feincms.content.medialibrary.models.MediaFileContent`. The argument `POSITION_CHOICES` and the corresponding field have been renamed to `TYPE_CHOICES` and `type` because that's a more fitting description of the intended use. The old and the new media file content should not be mixed; the hand-woven `raw_id_fields` support of the old media file content was not specific enough and interferes with Django's own `raw_id_fields` support.

- FeinCMS has gained a preview feature for pages which shouldn't be accessible to the general public yet. Just add the following line above the standard FeinCMS handler:

```python
url(r'', include('feincms.contrib.preview.urls')),
```

Another button will be automatically added in the page item editor.

Apart from all these new features a few cleanups have been made:

- FeinCMS 1.2 removed the CKEditor-specific rich text content in favor of a generalized rich text content supporting different rich text editors. Unfortunately the documentation and the available settings only reflected this partially. This has been rectified. Support for `TINYMCE_JS_URL`, `FEINCMS_TINYMCE_INIT_TEMPLATE` and `FEINCMS_TINYMCE_INIT_CONTEXT` has been completely removed. The two settings `FEINCMS_RICHTEXT_INIT_CONTEXT` and `FEINCMS_RICHTEXT_INIT_TEMPLATE` should be used instead. See the *Content types - what your page content is built of* documentation for more details.

- The two settings `FEINCMS_MEDIALIBRARY_ROOT` and `FEINCMS_MEDIALIBRARY_URL` have been removed. Their values always defaulted to `MEDIA_ROOT` and `MEDIA_URL`. The way they were used made it hard to support other storage backends in the media library. If you still need to customize the storage class used in the media library have a look at `MediaFile.reconfigure`.

- Support for the `show_on_top` option for the `ItemEditor` has been completely removed. This functionality has been deprecated since 1.2.

- A few one-line Page manager methods which were too similar to each other have been deprecated. They will be removed in the next release of FeinCMS. This concerns `page_for_path_or_404`, `for_request_or_404`, `best_match_for_request` and `from_request`. The improved `for_request` method should cover all bases.

- A few page methods have been deprecated. This concerns `active_children`, `active_children_in_navigation` and `get_siblings_and_self`. The useful bits are already available through Django's own related managers.

2.4 FeinCMS 1.5 release notes

2.4.1 Explicit reversing of URLs from `ApplicationContent`-embedded apps

URLs from third party apps embedded via `ApplicationContent` have been traditionally made reversable by an ugly monkey patch of `django.core.urlresolvers.reverse`. This mechanism has been deprecated and will be removed in future FeinCMS versions. To reverse an URL of a blog entry, instead of this:

```python
# deprecated
from django.core.urlresolvers import reverse
reverse('blog.urls/blog_detail', kwargs={'year': 2011, 'slug': 'some-slug'})
```

do this:

```python
from feincms.content.application.models import app_reverse
app_reverse('blog_detail', 'blog.urls', kwargs={'year': 2011, 'slug': 'some-slug'})
```

If you do not want to use the monkey patching behavior anymore, set `FEINCMS_REVERSE_MONKEY_PATCH = False` in your settings file.

The new method is accessible inside a template too:

```
{% load applicationcontent_tags %}

{# You have to quote the view name and the URLconf as in Django's future {% url %} tag. #}
{% app_reverse "blog_detail" "blog.urls" year=2011 slug='some-slug' %}
```

2.4.2 Inheritance 2.0

It's possible to use Django's template inheritance from third party applications embedded through `ApplicationContent` too. To use this facility, all you have to do is return a tuple consisting of the template and the context. Instead of:

```
def my_view(request):
    # ...
    return render_to_response('template.html', {'object': ...})
```

simply use:

```
def my_view(request):
    # ...
    return 'template.html', {'object': ...}
```

Note: `template.html` should extend a base template now.

2.4.3 Better support of `InlineModelAdmin` options for content types

The `FeinCMSInline` only differs from a stock StackedInline in differing defaults of `form`, `extra` and `fk_name`. All inline options should be supported now, especially `raw_id_fields` and `fieldsets`.

2.4.4 Minor changes

- The main CMS view is now based on Django's class-based generic views. Inheritance 2.0 will not work with the old views. You don't have to do anything if you use `feincms.urls` (as is recommended).

- Request and response processors have been moved out of the `Page` class into their own module, `feincms.module.page.processors`. They are still accessible for some time at the old place.

- `django.contrib.staticfiles` is now a mandatory dependency for the administration interface.

- The `active` and `in_navigation` booleans on the `Page` class now default to `True`.

- The minimum version requirements have changed. Django versions older than 1.3 aren't supported anymore, django-mptt must be 0.4 upwards.

- The mptt tree rebuilders have been removed; django-mptt offers tree rebuilding functionality itself.

- `django-queryset-transform` has been imported under `feincms.utils` and is used for speeding up various aspects of the media library. The prefilled attributes have been deprecated, because `django-queryset-transform` can be used to do everything they did, and better.

- `PageManager.active_filters` has been converted from a list to a `SortedDict`. This means that replacing or removing individual filters has become much easier than before. If you only used the public methods for registering new filters you don't have to change anything.

- The same has been done with the request and response processors.

- The `TemplateContent` has been changed to use the `filesystem` and the `app_directories` template loaders directly. It can be used together with the cached template loader now.

- The tree editor has received a few usability fixes with (hopefully) more to come.

- `Page.setup_request` can be called repeatedly without harm now. The return value of the first call is cached and returned on subsequent calls which means that request processors are run at most once.

- Extensions such as `translations` and `datepublisher` which were only usable with the page module have been made more generic and are available for other FeinCMS-derived models too.

- Media files from the medialibrary can be exported and imported in bulk.

- When creating a new translation of a page, content is only copied from the original translation when the new page does not have any content yet. Furthermore the user is notified that some content-copying has happened.

- A few bugs have been fixed.

2.5 FeinCMS 1.6 release notes

Welcome to FeinCMS 1.6!

2.5.1 Backwards-incompatible changes

Reversing application content URLs

The default value of `FEINCMS_REVERSE_MONKEY_PATCH` has been changed to `False`. Support for monkey-patching the `reverse()` method to support the old `'urlconf/viewname'` notation will be removed in the 1.7 release.

Improvements to the bundled file and image contents

- `ImageContent`, `FileContent` and `VideoContent` now have pretty icons out-of-the-box.
- `ImageContent` now accepts optional `FORMAT_CHOICES` for use with FeinCMS' bundled thumbnailers, as well as `caption` and `alt_text` fields.

  Note: If you are upgrading from an earlier version of FeinCMS, you'll have to add the new database columns yourself or use a migration tool like South to do it for you. Instructions for MySQL and the page module follow:

  ```
  ALTER TABLE page_page_imagecontent ADD COLUMN `alt_text` varchar(255) NOT NULL;
  ALTER TABLE page_page_imagecontent ADD COLUMN `caption` varchar(255) NOT NULL;
  ```

  If you want to use `FORMAT_CHOICES`:

  ```
  ALTER TABLE page_page_imagecontent ADD COLUMN `format` varchar(64) NOT NULL;
  ```

- `FileContent` now displays the size of the file in the default template, and uses `span` elements to allow styling of the title / size.

Removal of deprecated features

- Deprecated page manager methods have been removed. You should use `Page.objects.for_request` instead of the following manager methods:
  - `Page.objects.page_for_path_or_404()`
  - `Page.objects.for_request_or_404()`
  - `Page.objects.best_match_for_request()`
  - `Page.objects.from_request()`
- Deprecated page methods have been removed:
  - `Page.active_children()`: Use `Page.children.active()` instead.
  - `Page.active_children_in_navigation()`: Use `Page.children.in_navigation()` instead.
  - `Page.get_siblings_and_self()`: You probably wanted `self.parent.children.active()` or `self.get_siblings(include_self=True).active()` anyway.
- The shortcuts `Page.register_request_processors()` and `Page.register_response_processors()` to register several request or response processors at once have been removed in favor of their counterparts which only allow one processor at a time, but allow for replacing FeinCMS' included processors, `require_path_active_request_processor` and `redirect_request_processor`.

- It is not possible anymore to access the request and response processors as methods of the `Page` class. The processors are all in `feincms.module.page.processors` now.

- The deprecated support for prefilled attributes has been removed. Use Django's own `prefetch_related` or `feincms.utils.queryset_transform` instead.

- The deprecated `feincms.views.base` module has been removed. The code has been moved to `feincms.views.legacy` during the FeinCMS v1.5 cycle.

New deprecations

- The view decorator `feincms.views.decorators.add_page_to_extra_context` has been deprecated as it was mostly used with function-based generic views, which have been deprecated in Django as well. Use Django's class-based generic views and the `feincms.context_processors.add_page_if_missing` context processor if you need similar functionality instead.

- The content type `feincms.content.medialibrary.models.MediaFileContent` has been deprecated since FeinCMS v1.4. The whole module has been deprecated now and will be replaced with the contents of `feincms.content.medialibrary.v2` in FeinCMS v1.7. The `v2` module will stay around for another release or two so that code using `v2` will continue working with FeinCMS v1.8 (at least).

- The template tag `feincms_navigation` has been superseded by `feincms_nav` which fixes a few problems with the old code and is generally much more maintainable. The old version will stay around for one more version and will be removed for FeinCMS v1.8. The only difference (apart from the bugfixes and the slightly different syntax) is that `feincms_nav` unconditionally uses navigation extensions. Additionally, `feincms_navigation` uses `feincms_nav`'s implementation behind the scenes, which means that the `extended` argument does not have an effect anymore (it's always active).

- The HTML cleaning support in `feincms.utils.html.cleanse` which could be easily used in the `RichTextContent` by passing `cleanse=True` has been copied into its own Python package, feincms-cleanse. You should start passing a callable to `cleanse` right now. The existing support for cleansing will only be available up to FeinCMS v1.7.

- FeinCMS v1.8 will not support shorthands anymore when registering extensions. Always provide the full python path to the extension file (or pass callables) to `feincms.models.Base.register_extensions`. That is, `Page.register_extensions('feincms.module.extensions.ct_tracker')` should be used instead of `Page.register_extensions('ct_tracker')`. While it is a bit more work it will make it much more explicit what's going on.

Compatibility with Django and other apps

FeinCMS 1.6 requires Django 1.4. If you want to use django-reversion with FeinCMS you have to use django-reversion 1.6 or newer.

2.5.2 Notable features and improvements

- The bundled content types take additional steps to ensure that the main view context is available in content types' templates. If you only use the rendering tags (`feincms_render_region` and `feincms_render_content`) you can take advantage of all variables from your context processors in content types' templates too. Furthermore, those templatetags have been simplified by using Django's `template.Library.simple_tag` method now, which means that filters etc. are supported as template tag arguments now.

- `MediaFile` does no longer auto-rotate images on upload. It really is not a media library's job to magically modify user content; if needed, it should be done in an image filter (like sorl). Also, reading through the image data seems to have a side effect on some external storage engines which then would only save half the image data, see issue #254. Additionally, FeinCMS does not try anymore to detect whether uploaded

files really are images, and only looks at the file extension by default. We did not peek at the contents of other file types either.

- A new model field has been added, `feincms.contrib.richtext.RichTextField`. This is a drop-in replacement for Django's `models.TextField` with the difference that it adds the CSS classes required by rich text fields in the item editor.

- The value of `FEINCMS_FRONTEND_EDITING` defaults to `False` now.

- Frontend editing can now safely be used with caching. This is accomplished by saving state in a cookie instead of creating sessions all the time.

- The `SectionContent` content type has been updated and does properly use `raw_id_fields` for the media files instead of the hack which was used before.

- It is now possible to specify a different function for generating thumbnails in the media library administration. Set the setting `FEINCMS_MEDIALIBRARY_THUMBNAIL` to a function taking a media file instance and returning a URL to a thumbnail image or nothing if the file type cannot be handled by the thumbnailer.

- Thumbnails generated by the bundled `|thumbnail` and `|cropscale` template filters are stored separately from the uploaded files now. This change means that all thumbnails will be automatically regenerated after a FeinCMS update. If you need the old behavior for some reason, set the setting `FEINCMS_THUMBNAIL_DIR` to an empty string. The default setting is `'_thumbs/'`.

- All templates and examples have been converted to the new `{% url %}` syntax.

- Custom comment models are now supported in the `CommentsContent`.

- Media files are now removed from the disk too if a media file entry is removed from the database.

- The modules `feincms.module.page.models` and `feincms.module.medialibrary.models` have been split up. Admin code has been moved into `modeladmin.py` files, form code into `forms.py`.

2.5.3 Bugfixes

- The core page methods support running with `APPEND_SLASH = False` now. Many content types using forms do not, however.

- The MPTT attributes aren't hardcoded in the tree editor anymore. Custom names for the `left`, `right`, `level` and `tree_id` attributes are now supported. Models which do not use `id` as their primary key are supported now as well.

- FeinCMS uses timezone-aware datetimes now.

2.6 FeinCMS 1.7 release notes

Welcome to FeinCMS 1.7!

2.6.1 Extensions-mechanism refactor

The extensions mechanism has been refactored to remove the need to make models know about their related model admin classes. The new module `feincms.extensions` contains mixins and base classes - their purpose is as follows: *Extensions*.

2.6.2 View code refactor

Made views, content type and request / response processors reusable.

The legacy views at `feincms.views.legacy` were considered unhelpful and were removed.

2.6.3 Backwards-incompatible changes

Page manager methods behavior

Previously, the following page manager methods sometimes returned inactive objects or did not raise the appropriate (and asked for) `Http404` exception:

- `Page.objects.page_for_path`
- `Page.objects.best_match_for_path`
- `Page.objects.for_request`

The reason for that was that only the page itself was tested for activity in the manager method, and none of its ancestors. The check whether all ancestors are active was only conducted later in a request processor. This request processor was registered by default and was always run when `Page.objects.for_request` was called with `setup=True`.

However, request processors do not belong into the model layer. The necessity of running code belonging to a request-response cycle to get the correct answer from a manager method was undesirable. This has been rectified, those manager methods check the ancestry directly. The now redundant request processor `require_path_active_request_processor` has been removed.

Reversing application content URLs

The support for monkey-patching applicationcontent-awareness into Django's `django.core.urlresolvers.reverse()` has been removed.

Removal of deprecated features

- The old media library content type module `feincms.content.medialibrary.models` has been replaced with the contents of `feincms.content.medialibrary.v2`. The model field `position` has been renamed to `type`, instead of `POSITION_CHOICES` you should use `TYPE_CHOICES` now. The code has been simplified and hacks to imitate `raw_id_fields` have been replaced by working stock code. The `v2` module will stay around for another release and will be removed in FeinCMS v1.8. The now-unused template `admin/content/mediafile/init.html` has been deleted.

New deprecations

- `Page.setup_request()` does not do anything anymore and will be removed in FeinCMS v1.8.

2.6.4 Notable features and improvements

- A lazy version of `app_reverse()` is now available, `app_reverse_lazy()`.
- Because of the extensions refactor mentioned above, all `register_extension` methods have been removed. Additionally, the model admin classes are not imported inside the `models.py` files anymore.
- The setting `FEINCMS_USE_PAGE_ADMIN` can be set to false to prevent registration of the page model with the administration. This is especially useful if you only want to reuse parts of the page module.
- Various classes in `feincms.module.page` do not hardcode the page class anymore; hooks are provided to use your own models instead. Please refer to the source for additional information.
- `Page.redirect_to` can also contain the primary key of a page now, which means that the redirect target stays correct even if the page URL changes.
- Before, page content was copied automatically when creating a translation of an existing page. This behavior can be deactivated by unchecking a checkbox now.

- Work has begun to make the page forms, model admin classes and managers work with an abstract page model so that it will be easier to work with several page models in a single Django site.

2.6.5 Bugfixes

- It should be possible to store FeinCMS models in a secondary database, as long as the base model and all content types are stored in the same database.

- Changing templates in the item editor where the templates do not share common regions does not result in orphaned content blocks anymore.

- `feincms.utils.get_object()` knows how to import modules, not only objects inside modules now.

- The order and priority values for pages have been fixed when generating sitemaps.

- Various `save` and `delete` methods now come with `alters_data=True` to prevent their use in templates.

- Only one translation is permitted per language when using `feincms.translations`.

- FeinCMS can now be used without `django.contrib.sites`.

- If the fieldset of a content inline has been customized, the fieldset is not processed again to make sure that all form fields are actually shown. If you use dynamically generated fields in a content inline such as the application content does, you must not customize the fieldsets attribute of the `FeinCMSInline`.

2.6.6 Compatibility with Django and other apps

FeinCMS 1.7 requires Django 1.4 or better.

2.7 FeinCMS 1.8 release notes

Welcome to FeinCMS 1.8!

2.7.1 FeinCMS finally got continuous integration

Have a look at the status page here:

Travis CI

2.7.2 Preliminary Python 3.3 support

The testsuite runs through on Python 3.3.

2.7.3 Singleton templates

Templates can be defined to be singletons, which means that those templates can only occur once on a whole site. The page module additionally allows specifying that singleton templates must not have any children, and also that the page cannot be deleted.

2.7.4 Dependencies are automatically installed

Now that distribute and setuptools have merged, `setup.py` has been converted to use setuptools again which means that all dependencies of FeinCMS should be installed automatically.

2.7.5 Backwards-incompatible changes

- The template naming and order used in the section content has been changed to be more similar to the media library. The naming is now `<mediafile type>_<section content type>`, additionally the media file type is considered more important for template resolution than the section content type.

- The mechanism for finding the best application content match has been massively simplified and also made customizable. The default implementation of `ApplicationContent.closest_match` now only takes the current language into account.

Removal of deprecated features

- The module `feincms.admin.editor` has been removed. Import the classes from `feincms.admin.item_editor` or `feincms.admin.tree_editor` directly.

- The HTML cleansing module `feincms.utils.html.cleanse` has been removed. Use the standalone package feincms-cleanse instead.

- Registering extensions using shorthand notation is not possible anymore. Always use the full python path to the extension module.

- The two navigation template tags `feincms_navigation` and `feincms_navigation_extended` have been removed. The improved `feincms_nav` tag has been introduced with FeinCMS v1.6.

- The function-based generic views in `feincms.views.generic` have been removed. The decorator function `feincms.views.decorators.add_page_to_extra_context()` is therefore obsolete and has also been removed.

- The old media library content type module `feincms.content.medialibrary.models` has been replaced with the contents of `feincms.content.medialibrary.v2`. The model field `position` has been renamed to `type`, instead of `POSITION_CHOICES` you should use `TYPE_CHOICES` now. The code has been simplified and hacks to imitate `raw_id_fields` have been replaced by working stock code. The `v2` module will stay around for another release and will be removed in FeinCMS v1.8. The now-unused template `admin/content/mediafile/init.html` has been deleted.

- `Page.setup_request()` has been removed because it has not been doing anything for some time now.

2.7.6 New deprecations

- Page extensions should start explicitly adding their fields to the administration interface using `modeladmin.add_extension_options`. FeinCMS v1.8 will warn about fields collected automatically, the next release will not add unknown fields to the administration interface anymore.

- All extensions should inherit from `feincms.extensions.Extension`. Support for `register(cls, admin_cls)`-style functions will be removed in FeinCMS v1.9.

2.7.7 Notable features and improvements

- The template tags `feincms_render_region` and `feincms_render_content` do not require a request object anymore. If you omit the `request` parameter, the request will not be passed to the `render()` methods.

- The code is mostly flake8 clean.

- The new management command `medialibrary_orphans` can be used to find files which aren't referenced in the media library anymore.

- The test suite has been moved into its own top-level module.

2.7.8 Bugfixes

- The item and tree editor finally take Django permissions into account.

- The datepublisher response processor should not crash during daylight savings time changes anymore.

- The undocumented attribute `PageAdmin.unknown_fields` has been removed because it was modified at class level and not instance level which made reuse harder than necessary.

2.7.9 Compatibility with Django and other apps

FeinCMS 1.8 requires Django 1.4 or better. The testsuite is successfully run against Django 1.4, 1.5 and 1.6. Django 1.7 is not supported.

2.8 FeinCMS 1.9 release notes

Welcome to FeinCMS 1.9!

2.8.1 Extensions in the item editor

Extension fieldsets are now presented using a tabbed interface in the item editor as well to raise their visibility.

2.8.2 A better way to mark pages as active in the navigation

The new template tag `{% page_is_active %}` knows how to handle regular pages and also page pretenders from navigation extensions.

The following template snippet shows the recommended method of marking pages as active:

```
{% feincms_nav feincms_page level=1 as toplevel %}
<ul>
{% for page in toplevel %}
    {% page_is_active page as is_active %}
    <li {% if is_active %}class="active"{% endif %}>
        <a href="{{ page.get_navigation_url }}">{{ page.title }}</a>
    <li>
{% endfor %}
</ul>
```

2.8.3 Backwards-incompatible changes

Removal of deprecated features

- The table content type has been removed. It has open bugs for more than two years which weren't fixed, and now that we've moved on to newer versions of jQuery, the table content didn't even load.

- All extensions should inherit from `feincms.extensions.Extension`. The support for `register(cls, admin_cls)`-style functions has been removed.

- Unknown page model fields (for example those added through page extensions) aren't added to the administration interface anymore. Use `modeladmin.add_extension_options` if you want extension fields to appear.

- The `_feincms_extensions` property on the page model (and on all models inheriting `ExtensionsMixin` has been removed. It has been deprecated since FeinCMS v1.7.

2.8.4 New deprecations

2.8.5 Notable features and improvements

- The bundled versions of jQuery and jQuery UI have been updated to 1.9.1 and 1.10.3 respectively. Custom confirmation boxes have been removed and standard ones are used instead now.

- The diff between 1.8 and 1.9 is large – most of it because of coding style cleanups. flake8 runs should not show any warnings.

- Extension classes can be passed directly to `register_extensions()` now.

2.8.6 Bugfixes

2.8.7 Compatibility with Django and other apps

FeinCMS 1.9 requires Django 1.4 or better. The testsuite is successfully run against Django 1.4, 1.5 and 1.6. Django 1.7 is not supported.

Indices and tables

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

f