Wagtail Documentation

Release 1.2

Torchbox

Apr 21, 2017

Contents

1 Index **3**
- 1.1 Getting started . . . 3
- 1.2 Topics . . . 18
- 1.3 Advanced topics . . . 60
- 1.4 Reference . . . 84
- 1.5 Support . . . 143
- 1.6 Using Wagtail: an Editor's guide . . . 144
- 1.7 Contributing to Wagtail . . . 173
- 1.8 Release notes . . . 180

Python Module Index **213**

Wagtail is an open source CMS written in Python and built on the Django web framework.

Below are some useful links to help you get started with Wagtail.

- **First steps**
  - *Getting started*
  - *Your first Wagtail site*
  - *Demo site*
- **Creating your Wagtail site**
  - *Page models*
  - *Writing templates*
  - *Images*
  - *Search*
  - *Third-party tutorials*
- **Using Wagtail**
  - *Editors guide*

CHAPTER 1

Index

Getting started

Wagtail is built on the Django web framework, so this document assumes you've already got the essentials installed. But if not, those essentials are:

- Python
- pip (Note that pip is included by default with Python 2.7.9 and later and Python 3.4 and later)

We'd also recommend Virtualenv, which provides isolated Python environments:

- Virtualenv

Important: Before installing Wagtail, it is necessary to install the **libjpeg** and **zlib** libraries, which provide support for working with JPEG, PNG and GIF images (via the Python **Pillow** library). The way to do this varies by platform - see Pillow's platform-specific installation instructions.

With the above installed, the quickest way to install Wagtail is:

```
pip install wagtail
```

(`sudo` may be required if installing system-wide or without virtualenv)

Once installed, Wagtail provides a command similar to Django's `django-admin startproject` which stubs out a new site/project:

```
wagtail start mysite
```

This will create a new folder `mysite`, based on a template containing all you need to get started. More information on that template is available *here*.

Inside your `mysite` folder, we now just run the setup steps necessary for any Django project:

```
pip install -r requirements.txt
./manage.py migrate
./manage.py createsuperuser
./manage.py runserver
```

Your site is now accessible at `http://localhost:8000`, with the admin backend available at `http://localhost:8000/admin/`.

This will set you up with a new standalone Wagtail project. If you'd like to add Wagtail to an existing Django project instead, see *Integrating Wagtail into a Django project*.

There are a few optional packages which are not installed by default but are recommended to improve performance or add features to Wagtail, including:

- *Elasticsearch*.
- *Feature Detection*.

Your first Wagtail site

Note: This tutorial covers setting up a brand new Wagtail project. If you'd like to add Wagtail to an existing Django project instead, see *Integrating Wagtail into a Django project*.

1. Install Wagtail and its dependencies:

   ```
   pip install wagtail
   ```

2. Start your site:

   ```
   wagtail start mysite
   cd mysite
   ```

   Wagtail provides a `start` command similar to `django-admin.py startproject`. Running `wagtail start mysite` in your project will generate a new `mysite` folder with a few Wagtail-specific extras, including the required project settings, a "home" app with a blank `HomePage` model and basic templates and a sample "search" app.

3. Install project dependencies:

   ```
   pip install -r requirements.txt
   ```

   This ensures that you have the relevant version of Django for the project you've just created.

4. Create the database:

   ```
   python manage.py migrate
   ```

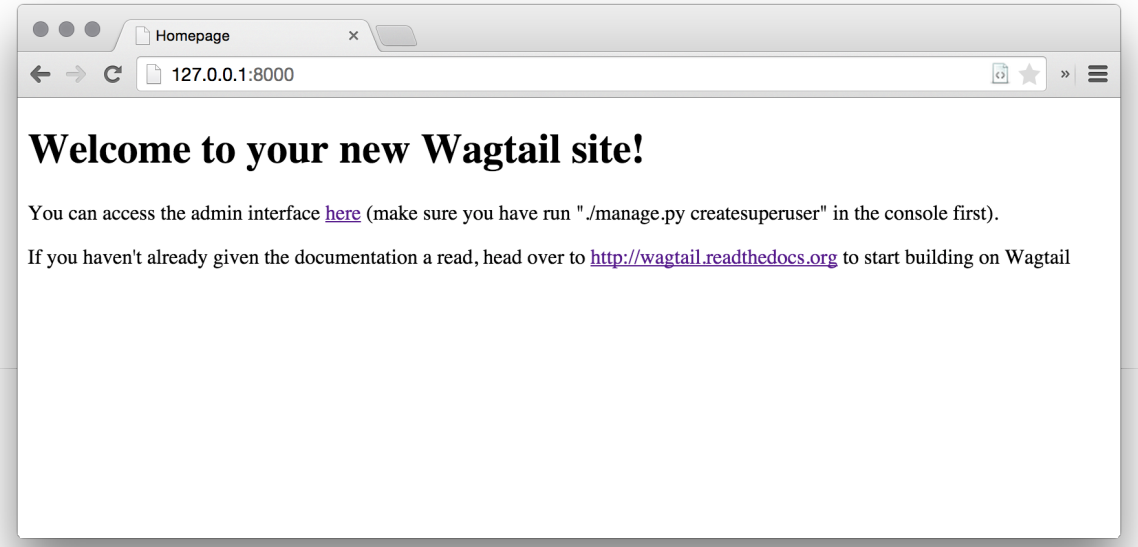
   If you haven't updated the project settings, this will be a SQLite database file in the project directory.

5. Create an admin user:

   ```
   python manage.py createsuperuser
   ```

6. `python manage.py runserver` If everything worked, http://127.0.0.1:8000 will show you a welcome page

   You can now access the administrative area at `/admin`



Extend the HomePage model

Out of the box, the "home" app defines a blank `HomePage` model in `models.py`, along with a migration that creates a homepage and configures Wagtail to use it.

Edit `home/models.py` as follows, to add a `body` field to the model:

```
from __future__ import unicode_literals

from django.db import models

from wagtail.wagtailcore.models import Page
from wagtail.wagtailcore.fields import RichTextField
from wagtail.wagtailadmin.edit_handlers import FieldPanel


class HomePage(Page):
    body = RichTextField(blank=True)

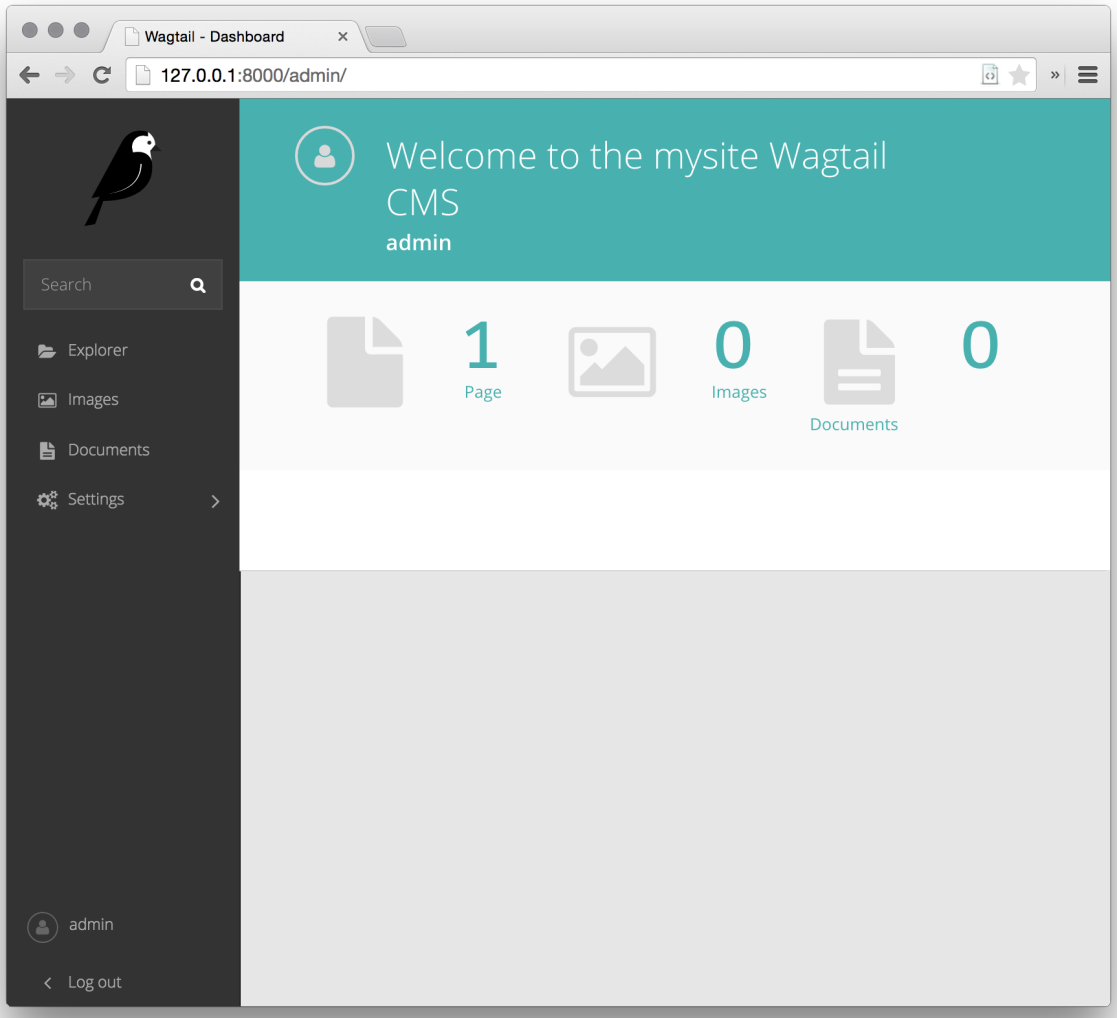
    content_panels = Page.content_panels + [
        FieldPanel('body', classname="full")
    ]
```

`body` is defined as `RichTextField`, a special Wagtail field. You can use any of the Django core fields. `content_panels` define the capabilities and the layout of the editing interface. *More on creating Page models.*

Run `python manage.py makemigrations`, then `python manage.py migrate` to update the database with your model changes. You must run the above commands each time you make changes to the model definition.

You can now edit the homepage within the Wagtail admin area (go to Explorer, Homepage, then Edit) to see the new body field. Enter some text into the body field, and publish the page.

The page template now needs to be updated to reflect the changes made to the model. Wagtail uses normal Django templates to render each page type. It automatically generates a template filename from the model name by separat-


Wagtail - Dashboard
127.0.0.1:8000/admin/
Welcome to the mysite Wagtail CMS
admin
Search
Explorer
Images
Documents
Settings
1
Page
0
Images
0
Documents
admin
Log out

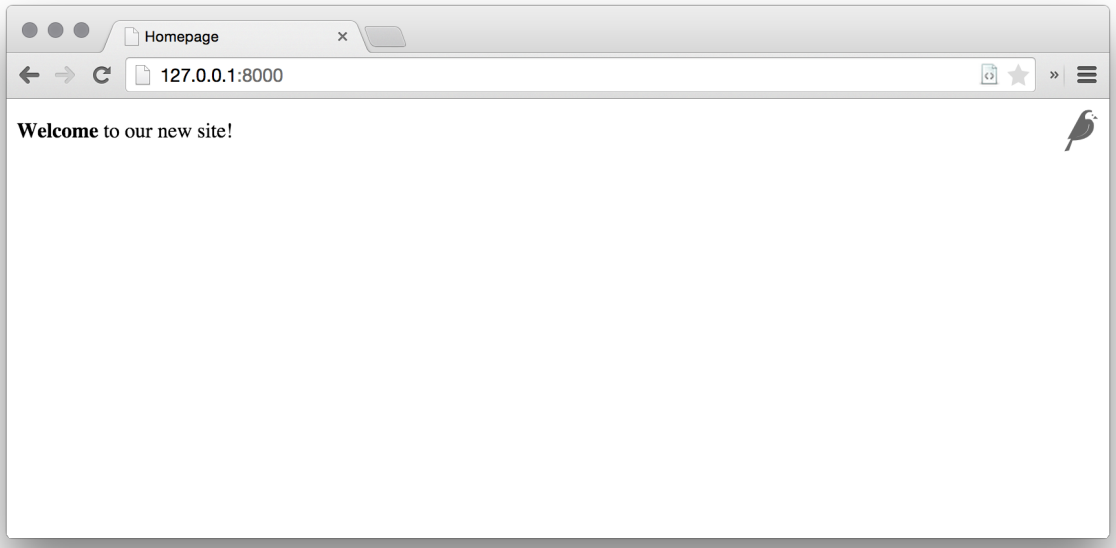
ing capital letters with underscores (e.g. HomePage becomes home_page.html). Edit `home/templates/home/home_page.html` to contain the following:

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

{% load wagtailcore_tags %}

{% block body_class %}template-homepage{% endblock %}

{% block content %}
    {{ page.body|richtext }}
{% endblock %}
```



Wagtail template tags

Wagtail provides a number of template tags & filters which can be loaded by including `{% load wagtailcore_tags %}` at the top of your template file.

In this tutorial, we use the *richtext* filter to escape and print the contents of a `RichTextField`:

```
{% load wagtailcore_tags %}
{{ page.body|richtext }}
```

Produces:

```
<div class="rich-text">
    <p>
        <b>Welcome</b> to our new site!
    </p>
</div>
```

Note: You'll need to include `{% load wagtailcore_tags %}` in each template that uses Wagtail's tags. Django will throw a `TemplateSyntaxError` if the tags aren't loaded.

A basic blog

We are now ready to create a blog. To do so, run `python manage.py startapp blog` to create a new app in your Wagtail site.

Add the new `blog` app to `INSTALLED_APPS` in `mysite/settings/base.py`.

The following example defines a basic blog post model in `blog/models.py`:

```
from django.db import models

from wagtail.wagtailcore.models import Page
from wagtail.wagtailcore.fields import RichTextField
from wagtail.wagtailadmin.edit_handlers import FieldPanel
from wagtail.wagtailsearch import index


class BlogPage(Page):
    date = models.DateField("Post date")
    intro = models.CharField(max_length=250)
    body = RichTextField(blank=True)

    search_fields = Page.search_fields + (
        index.SearchField('intro'),
        index.SearchField('body'),
    )

    content_panels = Page.content_panels + [
        FieldPanel('date'),
        FieldPanel('intro'),
        FieldPanel('body', classname="full")
    ]
```

Create a template at `blog/templates/blog/blog_page.html`:

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

{% load wagtailcore_tags %}

{% block body_class %}template-blogpage{% endblock %}

{% block content %}
    <h1>{{ page.title }}</h1>
    <p class="meta">{{ page.date }}</p>

    <div class="intro">{{ page.intro }}</div>

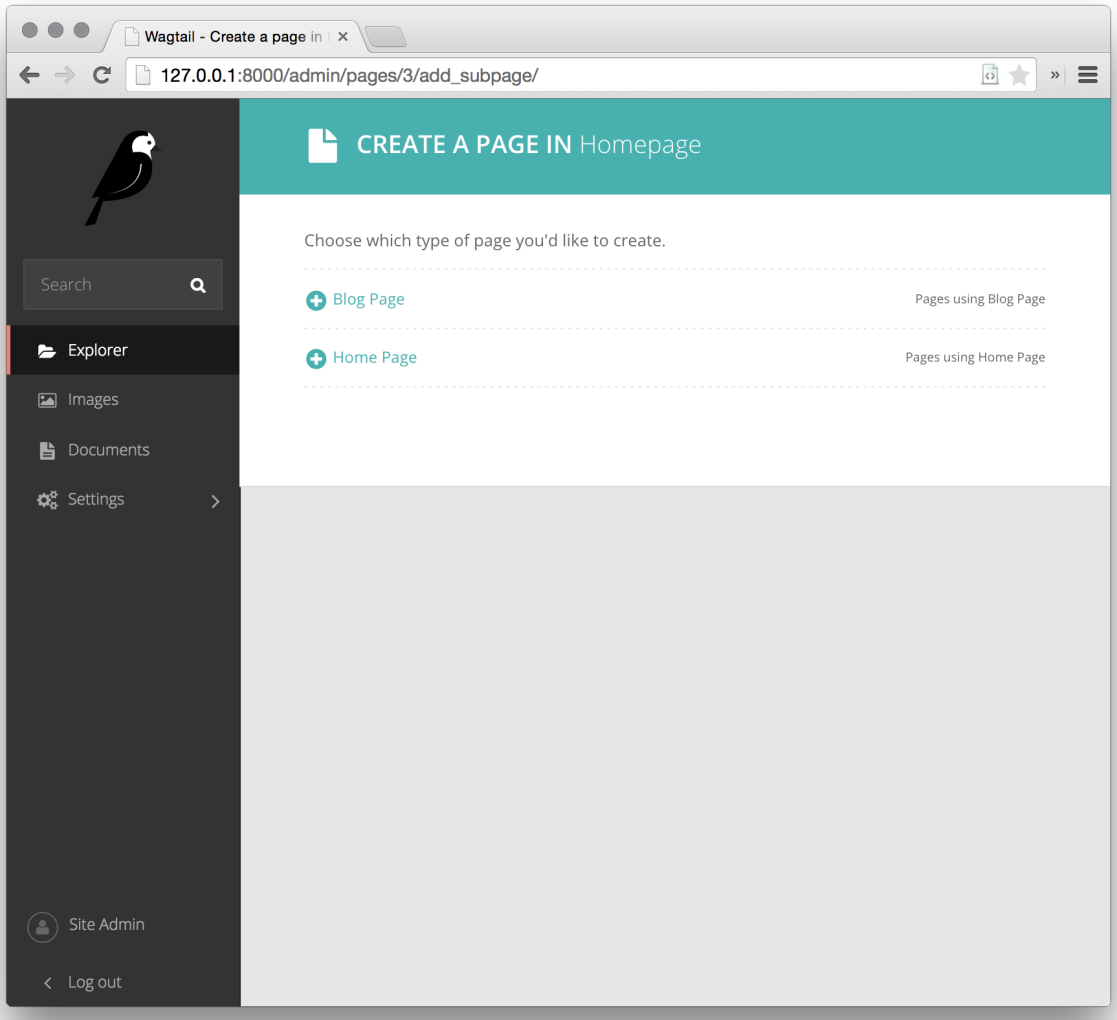
    {{ page.body|richtext }}
{% endblock %}
```

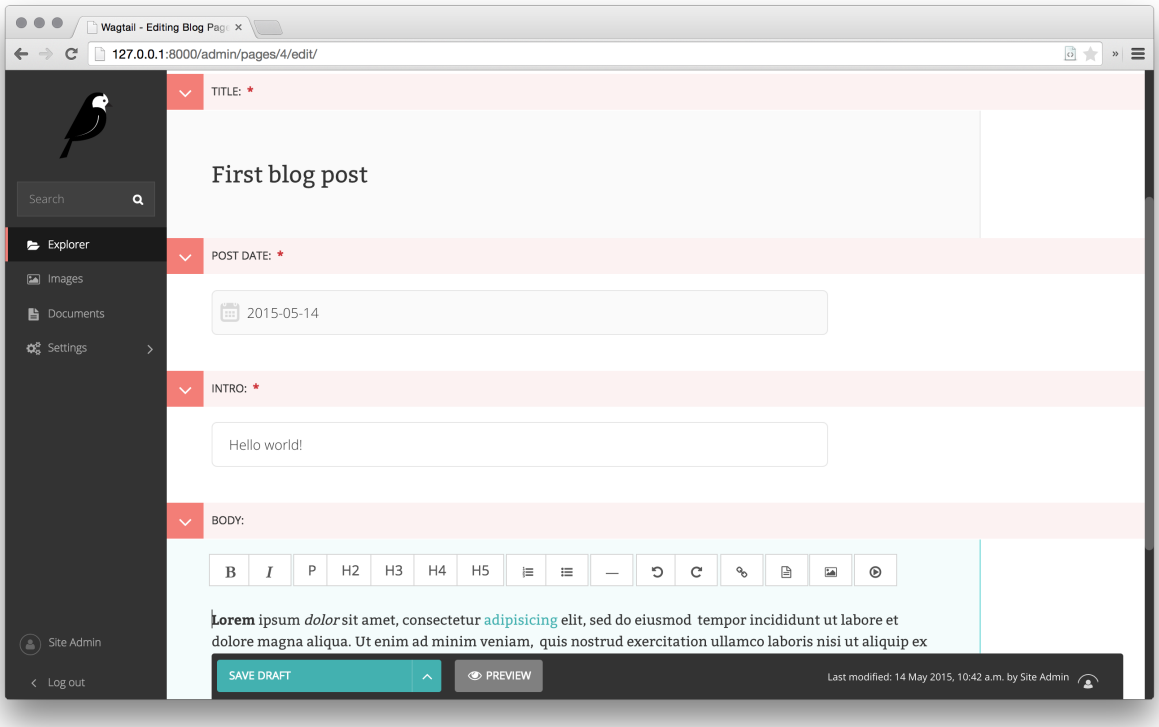
Run `python manage.py makemigrations` and `python manage.py migrate`.

Image support

Wagtail provides support for images out of the box. To add them to your model:

```
from django.db import models
```


Wagtail - Create a page in
127.0.0.1:8000/admin/pages/3/add_subpage/
CREATE A PAGE IN Homepage
Choose which type of page you'd like to create.
Blog Page
Pages using Blog Page
Home Page
Pages using Home Page
Search
Explorer
Images
Documents
Settings
Site Admin
Log out




```
from wagtail.wagtailcore.models import Page
from wagtail.wagtailcore.fields import RichTextField
from wagtail.wagtailadmin.edit_handlers import FieldPanel
from wagtail.wagtailimages.edit_handlers import ImageChooserPanel
from wagtail.wagtailsearch import index


class BlogPage(Page):
    main_image = models.ForeignKey(
        'wagtailimages.Image',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+'
    )
    date = models.DateField("Post date")
    intro = models.CharField(max_length=250)
    body = RichTextField(blank=True)

    search_fields = Page.search_fields + (
        index.SearchField('intro'),
        index.SearchField('body'),
    )

    content_panels = Page.content_panels + [
        FieldPanel('date'),
        ImageChooserPanel('main_image'),
        FieldPanel('intro'),
        FieldPanel('body'),
```

```
    ]
```

Run `python manage.py makemigrations` and `python manage.py migrate`.

Adjust your blog page template to include the image:

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

{% load wagtailcore_tags wagtailimages_tags %}

{% block body_class %}template-blogpage{% endblock %}

{% block content %}
    <h1>{{ page.title }}</h1>
    <p class="meta">{{ page.date }}</p>

    {% if page.main_image %}
      {% image page.main_image width-400 %}
    {% endif %}

    <div class="intro">{{ page.intro }}</div>

    {{ page.body|richtext }}
{% endblock %}
```

You can read more about using images in templates in the *docs*.

Blog Index

Let us extend the Blog app to provide an index.

```
class BlogIndexPage(Page):
    intro = RichTextField(blank=True)

    content_panels = Page.content_panels + [
        FieldPanel('intro', classname="full")
    ]
```

The above creates an index type to collect all our blog posts.

`blog/templates/blog/blog_index_page.html`

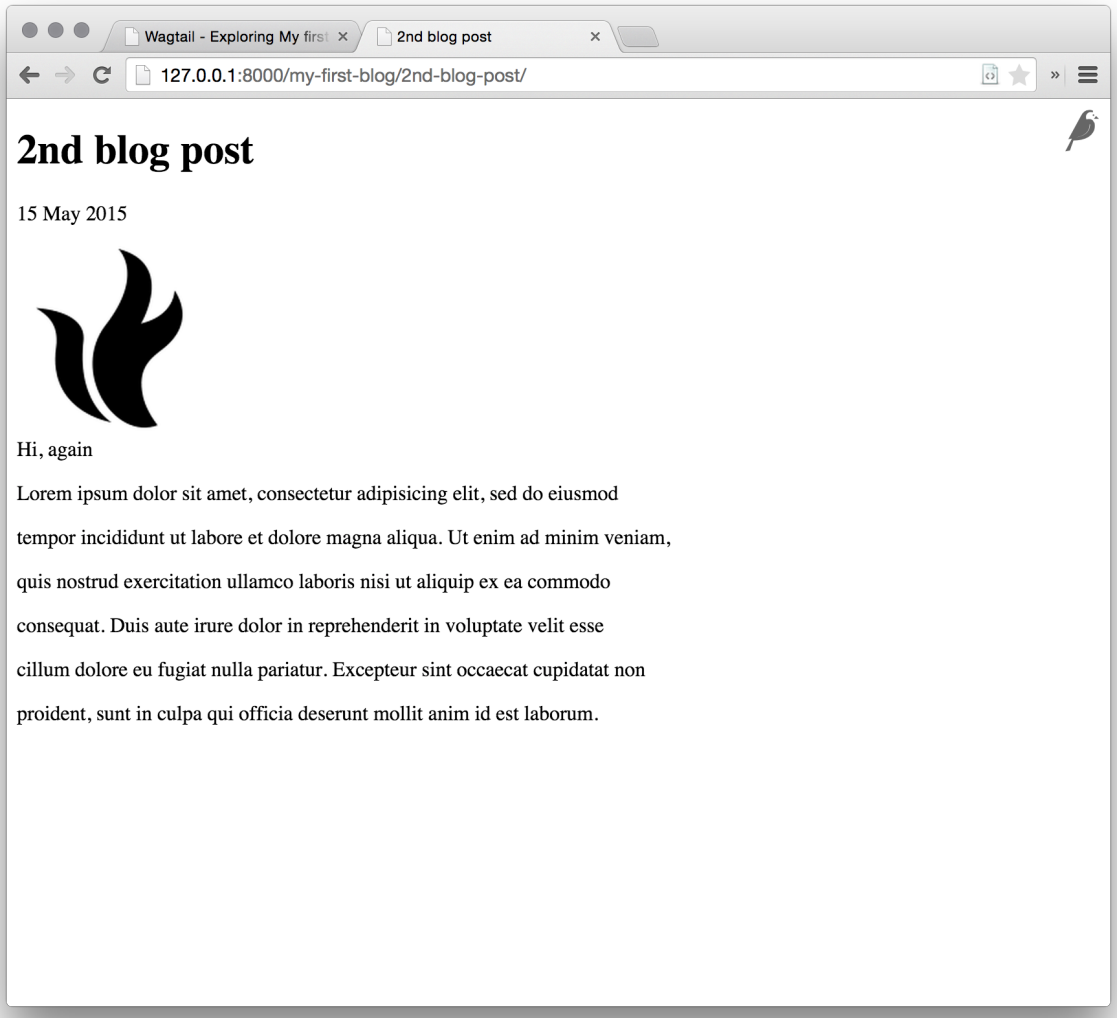
```
{% extends "base.html" %}

{% load wagtailcore_tags %}

{% block body_class %}template-blogindexpage{% endblock %}

{% block content %}
    <h1>{{ page.title }}</h1>

    <div class="intro">{{ page.intro|richtext }}</div>
{% endblock %}
```


Wagtail - Exploring My first
2nd blog post
127.0.0.1:8000/my-first-blog/2nd-blog-post/
2nd blog post
15 May 2015
Hi, again
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Related items

Let's extend the BlogIndexPage to add related links. The related links can be BlogPages or external links. Change `blog/models.py` to

```
from django.db import models

from modelcluster.fields import ParentalKey

from wagtail.wagtailcore.models import Page, Orderable
from wagtail.wagtailcore.fields import RichTextField
from wagtail.wagtailadmin.edit_handlers import (FieldPanel,
                                                InlinePanel,
                                                MultiFieldPanel,
                                                PageChooserPanel)
from wagtail.wagtailimages.edit_handlers import ImageChooserPanel
from wagtail.wagtailsearch import index


# ...

class LinkFields(models.Model):
    link_external = models.URLField("External link", blank=True)
    link_page = models.ForeignKey(
        'wagtailcore.Page',
        null=True,
        blank=True,
        related_name='+'
    )

    @property
    def link(self):
        if self.link_page:
            return self.link_page.url
        else:
            return self.link_external

    panels = [
        FieldPanel('link_external'),
        PageChooserPanel('link_page'),
    ]

    class Meta:
        abstract = True


# Related links
class RelatedLink(LinkFields):
    title = models.CharField(max_length=255, help_text="Link title")

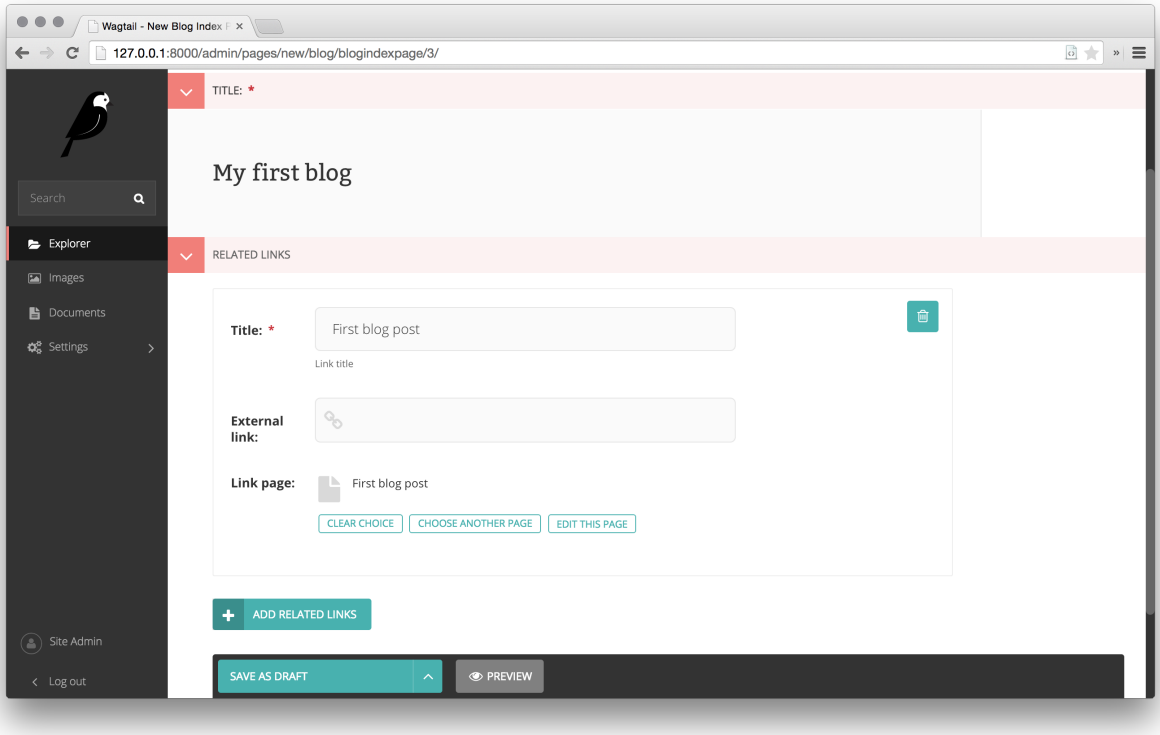
    panels = [
        FieldPanel('title'),
        MultiFieldPanel(LinkFields.panels, "Link"),
    ]

    class Meta:
        abstract = True
```

```
class BlogIndexPage(Page):
    intro = RichTextField(blank=True)

    content_panels = Page.content_panels + [
        FieldPanel('intro', classname="full"),
        InlinePanel('related_links', label="Related links"),
    ]


class BlogIndexRelatedLink(Orderable, RelatedLink):
    page = ParentalKey('BlogIndexPage', related_name='related_links')
```



Extend `blog_index_page.html` to show related items

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

{% load wagtailcore_tags %}

{% block body_class %}template-blogindexpage{% endblock %}

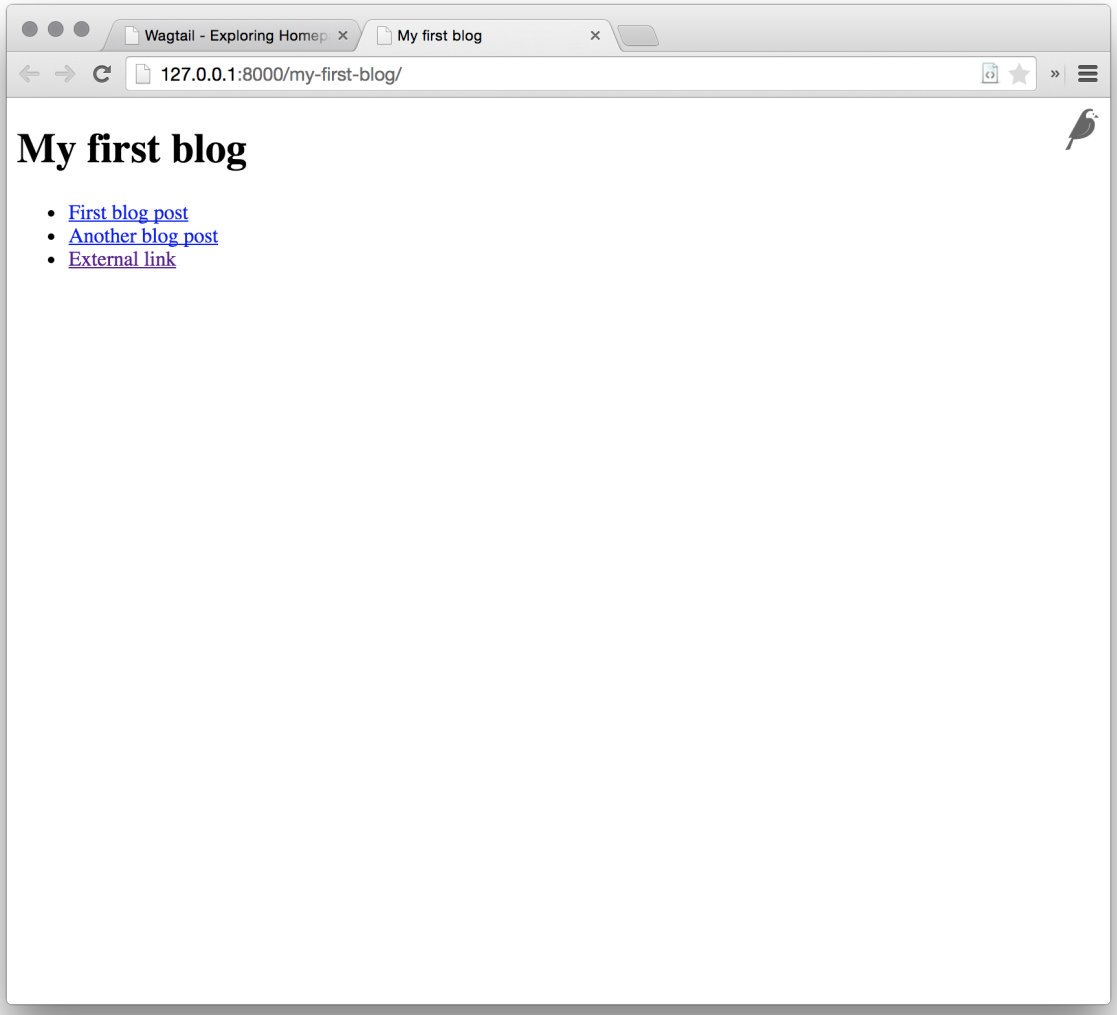
{% block content %}
    <h1>{{ page.title }}</h1>

    <div class="intro">{{ page.intro|richtext }}</div>

    {% if page.related_links.all %}
        <ul>
            {% for item in page.related_links.all %}
```

```
                <li><a href="{{ item.link }}">{{ item.title }}</a></li>
            {% endfor %}
        </ul>
    {% endif %}
{% endblock %}
```

You now have a fully working blog with featured blog posts.



Where next

- Read the Wagtail *topics* and *reference* documentation
- Learn how to implement *StreamField* for freeform page content
- Browse through the *advanced topics* section and read *third-party tutorials*

Demo site

To create a new site on Wagtail we recommend the `wagtail start` command in *Getting started*, however a demo site exists containing example page types and models. We also recommend you use the demo site for testing during development of Wagtail itself.

The repo and installation instructions can be found here: https://github.com/torchbox/wagtaildemo

Integrating Wagtail into a Django project

Wagtail provides the `wagtail start` command and project template to get you started with a new Wagtail project as quickly as possible, but it's easy to integrate Wagtail into an existing Django project too.

Wagtail is currently compatible with Django 1.7 and 1.8. First, install the `wagtail` package from PyPI:

```
pip install wagtail
```

or add the package to your existing requirements file. This will also install the **Pillow** library as a dependency, which requires libjpeg and zlib - see Pillow's platform-specific installation instructions.

Settings

In your settings file, add the following apps to `INSTALLED_APPS`:

```
'wagtail.wagtailforms',
'wagtail.wagtailredirects',
'wagtail.wagtailembeds',
'wagtail.wagtailsites',
'wagtail.wagtailusers',
'wagtail.wagtailsnippets',
'wagtail.wagtaildocs',
'wagtail.wagtailimages',
'wagtail.wagtailsearch',
'wagtail.wagtailadmin',
'wagtail.wagtailcore',

'modelcluster',
'compressor',
'taggit',
```

Add the following entries to `MIDDLEWARE_CLASSES`:

```
'wagtail.wagtailcore.middleware.SiteMiddleware',
'wagtail.wagtailredirects.middleware.RedirectMiddleware',
```

Add a `STATIC_ROOT` setting, if your project does not have one already:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Add a `WAGTAIL_SITE_NAME` - this will be displayed on the main dashboard of the Wagtail admin backend:

```
WAGTAIL_SITE_NAME = 'My Example Site'
```

Various other settings are available to configure Wagtail's behaviour - see *Configuring Django for Wagtail*.

URL configuration

Now make the following additions to your `urls.py` file:

```
from wagtail.wagtailadmin import urls as wagtailadmin_urls
from wagtail.wagtaildocs import urls as wagtaildocs_urls
from wagtail.wagtailcore import urls as wagtail_urls

urlpatterns = [
    ...
    url(r'^cms/', include(wagtailadmin_urls)),
    url(r'^documents/', include(wagtaildocs_urls)),
    url(r'^pages/', include(wagtail_urls)),
    ...
]
```

The URL paths here can be altered as necessary to fit your project's URL scheme.

`wagtailadmin_urls` provides the admin interface for Wagtail. This is separate from the Django admin interface (`django.contrib.admin`); Wagtail-only projects typically host the Wagtail admin at `/admin/`, but if this would clash with your project's existing admin backend then an alternative path can be used, such as `/cms/` here.

`wagtaildocs_urls` is the location from where document files will be served. This can be omitted if you do not intend to use Wagtail's document management features.

`wagtail_urls` is the base location from where the pages of your Wagtail site will be served. In the above example, Wagtail will handle URLs under `/pages/`, leaving the root URL and other paths to be handled as normal by your Django project. If you want Wagtail to handle the entire URL space including the root URL, this can be replaced with:

```
url(r'', include(wagtail_urls)),
```

In this case, this should be placed at the end of the `urlpatterns` list, so that it does not override more specific URL patterns.

With this configuration in place, you are ready to run `./manage.py migrate` to create the database tables used by Wagtail.

User accounts

Superuser accounts receive automatic access to the Wagtail admin interface; use `./manage.py createsuperuser` if you don't already have one. Custom user models are supported, with some restrictions; Wagtail uses an extension of Django's permissions framework, so your user model must at minimum inherit from `AbstractBaseUser` and `PermissionsMixin`.

Start developing

You're now ready to add a new app to your Django project (via `./manage.py startapp` - remember to add it to `INSTALLED_APPS`) and set up page models, as described in *Your first Wagtail site*.

Note that there's one small difference when not using the Wagtail project template: Wagtail creates an initial homepage of the basic type `Page`, which does not include any content fields beyond the title. You'll probably want to replace this with your own `HomePage` class - when you do so, ensure that you set up a site record (under Settings / Sites in the Wagtail admin) to point to the new homepage.

Topics

Page models

Each page type (a.k.a. content type) in Wagtail is represented by a Django model. All page models must inherit from the *wagtail.wagtailcore.models.Page* class.

As all page types are Django models, you can use any field type that Django provides. See Model field reference for a complete list of field types you can use. Wagtail also provides `RichTextField` which provides a WYSIWYG editor for editing rich-text content.

Django models

If you're not yet familiar with Django models, have a quick look at the following links to get you started:

- Creating models
- Model syntax

An example Wagtail page model

This example represents a typical blog post:

```
from django.db import models

from modelcluster.fields import ParentalKey

from wagtail.wagtailcore.models import Page, Orderable
from wagtail.wagtailcore.fields import RichTextField
from wagtail.wagtailadmin.edit_handlers import FieldPanel, MultiFieldPanel,␣
↪InlinePanel
from wagtail.wagtailimages.edit_handlers import ImageChooserPanel
from wagtail.wagtailsearch import index


class BlogPage(Page):

    # Database fields

    body = RichTextField()
    date = models.DateField("Post date")
    feed_image = models.ForeignKey(
        'wagtailimages.Image',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+'
    )


    # Search index configuraiton

    search_fields = Page.search_fields + (
        index.SearchField('body'),
        index.FilterField('date'),
```

```
    )


    # Editor panels configuration

    content_panels = Page.content_panels + [
        FieldPanel('date'),
        FieldPanel('body', classname="full"),
        InlinePanel('related_links', label="Related links"),
    ]

    promote_panels = [
        MultiFieldPanel(Page.promote_panels, "Common page configuration"),
        ImageChooserPanel('feed_image'),
    ]


    # Parent page / subpage type rules

    parent_page_types = ['blog.BlogIndex']
    subpage_types = []


class BlogPageRelatedLink(Orderable):
    page = ParentalKey(BlogPage, related_name='related_links')
    name = models.CharField(max_length=255)
    url = models.URLField()

    panels = [
        FieldPanel('name'),
        FieldPanel('url'),
    ]
```

Tip: To keep track of `Page` models and avoid class name clashes, it can be helpful to suffix model class names with "Page" e.g BlogPage, ListingIndexPage.

Writing page models

Here we'll describe each section of the above example to help you create your own page models.

Database fields

Each Wagtail page type is a Django model, represented in the database as a separate table.

Each page type can have its own set of fields. For example, a news article may have body text and a published date, whereas an event page may need separate fields for venue and start/finish times.

In Wagtail, you can use any Django field class. Most field classes provided by third party apps should work as well.

Wagtail also provides a couple of field classes of its own:

- `RichTextField` - For rich text content
- `StreamField` - A block-based content field (see: *Freeform page content using StreamField*)

For tagging, Wagtail fully supports django-taggit so we recommend using that.

Search

The `search_fields` attribute defines which fields are added to the search index and how they are indexed.

This should be a tuple of `SearchField` and `FilterField` objects. `SearchField` adds a field for full-text search. `FilterField` adds a field for filtering the results. A field can be indexed with both `SearchField` and `FilterField` at the same time (but only one instance of each).

In the above example, we've indexed `body` for full-text search and `date` for filtering.

The arguments that these field types accept are documented here: *Indexing extra fields*

Editor panels

There are a few attributes for defining how the page's fields will be arranged in the page editor interface:

- `content_panels` - For content, such as main body text
- `promote_panels` - For metadata, such as tags, thumbnail image and SEO title
- `settings_panels` - For settings, such as publish date

Each of these attributes is set to a list of `EditHandler` objects, which defines which fields appear on which tabs and how they are structured on each tab.

Here's a summary of the `EditHandler` classes that Wagtail provides out of the box. See *Available panel types* for full descriptions.

Basic

These allow editing of model fields, the `FieldPanel` class will choose the correct widget based on the type of the field. `StreamField` fields need to use a specialised panel class.

- *`FieldPanel`*
- `StreamFieldPanel`

Structural

These are used for structuring fields in the interface.

- *`MultiFieldPanel`* - For grouping similar fields together
- *`InlinePanel`* - For inlining child models
- *`FieldRowPanel`* - For organising multiple fields into a single row

Chooser

`ForeignKey` fields to certain models can use one of the below `ChooserPanel` classes. These add a nice modal-based chooser interface (and the image/document choosers also allow uploading new files without leaving the page editor).

- *`PageChooserPanel`*
- *`ImageChooserPanel`*
- *`DocumentChooserPanel`*
- *`SnippetChooserPanel`*

Note: In order to use one of these choosers, the model being linked to must either be a page, image, document or snippet.

To link to any other model type, you should use `FieldPanel`, which will create a dropdown box.

Customising the page editor interface

The page editor can be customised further. See *Customising the page editing interface*.

Parent page / subpage type rules

These two attributes allow you to control where page types may be used in your site. It allows you to define rules like "blog entries may only be created under a blog index".

Both take a list of model classes or model names. Model names are of the format `app_label.ModelName`. If the `app_label` is omitted, the same app is assumed.

- `parent_page_types` limits which page types this type can be created under
- `subpage_types` limits which page types can be created under this type

By default, any page type can be created under any page type and it is not necessary to set these attributes if that's the desired behaviour.

Setting `parent_page_types` to an empty list is a good way of preventing a particular page type from being created in the editor interface.

Template rendering

Each page model can be given a HTML template which is rendered when a user browses to a page on the site frontend. This is the simplest and most common way to get Wagtail content to end users (but not the only way).

Adding a template for a page model

Wagtail automatically chooses a name for the template based on the app label and model class name.

Format: `<app_label>/<model_name (snake cased)>.html`

For example, the template for the above blog page will be: `blog/blog_page.html`

You just need to create a template in a location where it can be accessed with this name.

Template context

Wagtail renders templates with the `page` variable bound to the page instance being rendered. Use this to access the content of the page. For example, to get the title of the current page, do `{{ page.title }}`. All variables provided by context processors are also available.

Customising template context

All pages have a `get_context` method that is called whenever the template is rendered and returns a dictionary of variables to bind into the template.

To add more variables to the template context, you can override this method:

```
class BlogIndexPage(Page):
    ...

    def get_context(self, request):
        context = super(BlogIndexPage, self).get_context(request)

        # Add extra variables and return the updated context
        context['blog_entries'] = BlogPage.objects.child_of(self).live()
        return context
```

The variables can then be used in the template:

```
{{ page.title }}

{% for entry in blog_entries %}
    {{ entry.title }}
{% endfor %}
```

Changing the template

Set the `template` attribute on the class to use a different template file:

```
class BlogPage(Page):
    ...

    template = 'other_template.html'
```

Dynamically choosing the template

The template can be changed on a per-instance basis by defining a `get_template` method on the page class. This method is called every time the page is rendered:

```
class BlogPage(Page):
    ...

    use_other_template = models.BooleanField()

    def get_template(self, request):
        if self.use_other_template:
            return 'blog/other_blog_page.html'

        return 'blog/blog_page.html'
```

In this example, pages that have the `use_other_template` boolean field set will use the `other_blog_page.html` template. All other pages will use the default `blog/blog_page.html`.

More control over page rendering

All page classes have a `serve()` method, that internally calls the `get_context` and `get_template` methods and renders the template. This method is similar to a Django view function, taking a Django `Request` object and returning a Django `Response` object.

This method can also be overridden for complete control over page rendering.

For example, here's a way you could make a page respond with a JSON representation of itself:

```
from django.http import JsonResponse


class BlogPage(Page):
    ...

    def serve(self, request):
        return JsonResponse({
            'title': self.title,
            'body': self.body,
            'date': self.date,

            # Resizes the image to 300px width and gets a URL to it
            'feed_image': self.feed_image.get_rendition('width-300').url,
        })
```

Inline models

Wagtail can nest the content of other models within the page. This is useful for creating repeated fields, such as related links or items to display in a carousel. Inline model content is also versioned with the rest of the page content.

Each inline model requires the following:

- It must inherit from *`wagtail.wagtailcore.models.Orderable`*
- It must have a `ParentalKey` to the parent model

Note: django-modelcluster and ParentalKey

The model inlining feature is provided by django-modelcluster and the `ParentalKey` field type must be imported from there:

```
from modelcluster.fields import ParentalKey
```

`ParentalKey` is a subclass of Django's `ForeignKey`, and takes the same arguments.

For example, the following inline model can be used to add related links (a list of name, url pairs) to the `BlogPage` model:

```
from django.db import models
from modelcluster.fields import ParentalKey
from wagtail.wagtailcore.models import Orderable


class BlogPageRelatedLink(Orderable):
    page = ParentalKey(BlogPage, related_name='related_links')
    name = models.CharField(max_length=255)
```

```
    url = models.URLField()

    panels = [
        FieldPanel('name'),
        FieldPanel('url'),
    ]
```

To add this to the admin interface, use the *InlinePanel* edit panel class:

```
content_panels = [
    ...

    InlinePanel('related_links', label="Related links"),
]
```

The first argument must match the value of the `related_name` attribute of the `ParentalKey`.

Working with pages

Wagtail uses Django's multi-table inheritance feature to allow multiple page models to be used in the same tree.

Each page is added to both Wagtail's builtin *Page* model as well as its user-defined model (such as the `BlogPage` model created earlier).

Pages can exist in Python code in two forms, an instance of `Page` or an instance of the page model.

> When working with multiple page types together, you will typically use instances of Wagtail's *Page* model, which doesn't give you access to any fields specific to their type.

```
# Get all pages in the database
>>> from wagtail.wagtailcore.models import Page
>>> Page.objects.all()
[<Page: Homepage>, <Page: About us>, <Page: Blog>, <Page: A Blog post>, <Page:␣
↪Another Blog post>]
```

When working with a single page type, you can work with instances of the user-defined model that gives access to all the fields available in `Page` and any user defined fields for that type.

```
# Get all blog entries in the database
>>> BlogPage.objects.all()
[<BlogPage: A Blog post>, <BlogPage: Another Blog post>]
```

You can convert a `Page` object to a specific object using the `.specific` property (this may cause an additional database lookup).

```
>>> page = Page.objects.get(title="A Blog post")
>>> page
<Page: A Blog post>

# Note: the blog post is an instance of Page so we cannot access body, date or feed_
↪image

>>> page.specific
<BlogPage: A Blog post>
```

Tips

Friendly model names

You can make your model names more friendly to users of Wagtail by using Django's internal `Meta` class with a `verbose_name`, e.g.:

```
class HomePage(Page):
    ...

    class Meta:
        verbose_name = "homepage"
```

When users are given a choice of pages to create, the list of page types is generated by splitting your model names on each of their capital letters. Thus a `HomePage` model would be named "Home Page" which is a little clumsy. `verbose_name` as in the example above, would change this to read "Homepage" which is slightly more conventional.

Page QuerySet ordering

`Page`-derived models *cannot* be given a default ordering by using the standard Django approach of adding an `ordering` attribute to the internal `Meta` class.

```
class NewsItemPage(Page):
    publication_date = models.DateField()
    ...

    class Meta:
        ordering = ('-publication_date', )  # will not work
```

This is because `Page` enforces ordering QuerySets by path. Instead you must apply the ordering explicitly when you construct a QuerySet:

```
news_items = NewsItemPage.objects.live().order_by('-publication_date')
```

Page custom managers

`Page` enforces its own 'objects' manager in its `__init__` method, so you cannot add a custom manager at the 'objects' attribute.

```
class EventPageQuerySet(PageQuerySet):

    def future(self):
        return self.filter(
            start_date__gte=timezone.localtime(timezone.now()).date()
        )

class EventPage(Page):
    start_date = models.DateField()

    objects = EventPageQuerySet.as_manager()  # will not work
```

To use a custom manager you must choose a different attribute name. Make sure to subclass `wagtail.wagtailcore.models.PageManager`.

```
from django.db import models
from django.utils import timezone
from wagtail.wagtailcore.models import Page, PageManager


class FutureEventPageManager(PageManager):

    def get_queryset(self):
        return super().get_queryset().filter(
            start_date__gte=timezone.localtime(timezone.now()).date()
        )


class EventPage(Page):
    start_date = models.DateField()

    future_events = FutureEventPageManager()
```

Then you can use `EventPage.future_events` in the manner you might expect.

Writing templates

Wagtail uses Django's templating language. For developers new to Django, start with Django's own template documentation: https://docs.djangoproject.com/en/dev/topics/templates/

Python programmers new to Django/Wagtail may prefer more technical documentation: https://docs.djangoproject.com/en/dev/ref/templates/api/

You should be familiar with Django templating basics before continuing with this documentation.

Templates

Every type of page or "content type" in Wagtail is defined as a "model" in a file called `models.py`. If your site has a blog, you might have a `BlogPage` model and another called `BlogPageListing`. The names of the models are up to the Django developer.

For each page model in `models.py`, Wagtail assumes an HTML template file exists of (almost) the same name. The Front End developer may need to create these templates themselves by refering to `models.py` to infer template names from the models defined therein.

To find a suitable template, Wagtail converts CamelCase names to underscore_case. So for a `BlogPage`, a template `blog_page.html` will be expected. The name of the template file can be overridden per model if necessary.

Template files are assumed to exist here:

```
name_of_project/
    name_of_app/
        templates/
            name_of_app/
                blog_page.html
        models.py
```

For more information, see the Django documentation for the application directories template loader.

Page content

The data/content entered into each page is accessed/output through Django's `{{ double-brace }}` notation. Each field from the model must be accessed by prefixing `page.`. e.g the page title `{{ page.title }}` or another field `{{ page.author }}`.

Additionally `request.` is available and contains Django's request object.

Static assets

Static files e.g CSS, JS and images are typically stored here:

```
name_of_project/
    name_of_app/
        static/
            name_of_app/
                css/
                js/
                images/
        models.py
```

(The names "css", "js" etc aren't important, only their position within the tree.)

Any file within the static folder should be inserted into your HTML using the `{% static %}` tag. More about it: *Static files (tag)*.

User images

Images uploaded to Wagtail by its users (as opposed to a developer's static files, above) go into the image library and from there are added to pages via the *page editor interface*.

Unlike other CMS, adding images to a page does not involve choosing a "version" of the image to use. Wagtail has no predefined image "formats" or "sizes". Instead the template developer defines image manipulation to occur *on the fly* when the image is requested, via a special syntax within the template.

Images from the library must be requested using this syntax, but a developer's static images can be added via conventional means e.g `img` tags. Only images from the library can be manipulated on the fly.

Read more about the image manipulation syntax here *Using images in templates*.

Template tags & filters

In addition to Django's standard tags and filters, Wagtail provides some of its own, which can be `load`-ed as you would any other

Images (tag)

The `image` tag inserts an XHTML-compatible `img` element into the page, setting its `src`, `width`, `height` and `alt`. See also *More control over the img tag*.

The syntax for the tag is thus:

```
{% image [image] [resize-rule] %}
```

For example:

```
{% load wagtailimages_tags %}
...

{% image page.photo width-400 %}

<!-- or a square thumbnail: -->
{% image page.photo fill-80x80 %}
```

See *Using images in templates* for full documentation.

Rich text (filter)

This filter takes a chunk of HTML content and renders it as safe HTML in the page. Importantly it also expands internal shorthand references to embedded images and links made in the Wagtail editor into fully-baked HTML ready for display.

Only fields using `RichTextField` need this applied in the template.

```
{% load wagtailcore_tags %}
...
{{ page.body|richtext }}
```

Responsive Embeds

Wagtail embeds and images are included at their full width, which may overflow the bounds of the content container you've defined in your templates. To make images and embeds responsive – meaning they'll resize to fit their container – include the following CSS.

```
.rich-text img {
    max-width: 100%;
    height: auto;
}

.responsive-object {
    position: relative;
}
    .responsive-object iframe,
    .responsive-object object,
    .responsive-object embed {
        position: absolute;
        top: 0;
        left: 0;
        width: 100%;
        height: 100%;
    }
```

Internal links (tag)

pageurl

Takes a Page object and returns a relative URL (`/foo/bar/`) if within the same site as the current page, or absolute (`http://example.com/foo/bar/`) if not.

```
{% load wagtailcore_tags %}
...
<a href="{% pageurl page.blog_page %}">
```

slugurl

Takes any `slug` as defined in a page's "Promote" tab and returns the URL for the matching Page. Like `pageurl`, will try to provide a relative link if possible, but will default to an absolute link if on a different site. This is most useful when creating shared page furniture e.g top level navigation or site-wide links.

```
{% load wagtailcore_tags %}
...
<a href="{% slugurl page.your_slug %}">
```

Static files (tag)

Used to load anything from your static files directory. Use of this tag avoids rewriting all static paths if hosting arrangements change, as they might between local and a live environments.

```
{% load static %}
...
<img src="{% static "name_of_app/myimage.jpg" %}" alt="My image"/>
```

Notice that the full path name is not required and the path snippet you enter only need begin with the parent app's directory name.

Wagtail User Bar

This tag provides a contextual flyout menu on the top-right of a page for logged-in users. The menu gives editors the ability to edit the current page or add another at the same level. Moderators are also given the ability to accept or reject a page previewed as part of content moderation.

```
{% load wagtailuserbar %}
...
{% wagtailuserbar %}
```

By default the User Bar appears in the top right of the browser window, flush with the edge. If this conflicts with your design it can be moved with a css rule in your own CSS files e.g to move it down from the top:

```
#wagtail-userbar{
   top:200px
}
```

Images

Using images in templates

The `image` tag inserts an XHTML-compatible `img` element into the page, setting its `src`, `width`, `height` and `alt`. See also *More control over the img tag*.

The syntax for the tag is thus:

```
{% image [image] [resize-rule] %}
```

For example:

```
{% load wagtailimages_tags %}
...

{% image page.photo width-400 %}

<!-- or a square thumbnail: -->
{% image page.photo fill-80x80 %}
```

In the above syntax example `[image]` is the Django object refering to the image. If your page model defined a field called "photo" then `[image]` would probably be `page.photo`. The `[resize-rule]` defines how the image is to be resized when inserted into the page; various resizing methods are supported, to cater for different usage cases (e.g. lead images that span the whole width of the page, or thumbnails to be cropped to a fixed size).

Note that a space separates `[image]` and `[resize-rule]`, but the resize rule must not contain spaces.

The available resizing methods are:

`max` (takes two dimensions)

```
{% image page.photo max-1000x500 %}
```

Fit **within** the given dimensions.

The longest edge will be reduced to the equivalent dimension size defined. For example, a portrait image of width 1000, height 2000, treated with the `max` dimensions `1000x500` (landscape) would result in the image shrunk so the *height* was 500 pixels and the width 250.

`min` (takes two dimensions)

```
{% image page.photo min-500x200 %}
```

Cover the given dimensions.

This may result in an image slightly **larger** than the dimensions you specify. e.g A square image of width 2000, height 2000, treated with the `min` dimensions `500x200` (landscape) would have its height and width changed to 500, i.e matching the width required, but greater than the height.

`width` (takes one dimension)

```
{% image page.photo width-640 %}
```

Reduces the width of the image to the dimension specified.

`height` (takes one dimension)

```
{% image page.photo height-480 %}
```

Resize the height of the image to the dimension specified..

`fill` (takes two dimensions and an optional `-c` parameter)

```
{% image page.photo fill-200x200 %}
```

Resize and **crop** to fill the **exact** dimensions.

This can be particularly useful for websites requiring square thumbnails of arbitrary images. For example, a landscape image of width 2000, height 1000, treated with `fill` dimensions `200x200` would have its height reduced to 200, then its width (ordinarily 400) cropped to 200.

This filter will crop to the image's focal point if it has been set. If not, it will crop to the centre of the image.

On images that won't upscale

It's possible to request an image with `fill` dimensions that the image can't support without upscaling. e.g an image 400x200 requested with `fill-400x400`. In this situation the *ratio of the requested fill* will be matched, but the dimension will not. So with that example 400x200 image, the resulting image will be 200x200.

Cropping closer to the focal point

By default, Wagtail will only crop to change the aspect ratio of the image.

In some cases (thumbnails, for example) it may be nice to crop closer to the focal point so the subject of the image is easier to see.

You can do this by appending `-c<percentage>` at the end of the method. For example, if you would like the image to be cropped as closely as possible to its focal point, add `-c100` to the end of the method.

```
{% image page.photo fill-200x200-c100 %}
```

This will crop the image as much as it can, but will never crop into the focal point.

If you find that `-c100` is too close, you can try `-c75` or `-c50` (any whole number from 0 to 100 is accepted).

`original` (takes no dimensions)

```
{% image page.photo original %}
```

Leaves the image at its original size - no resizing is performed.

Note: Wagtail does not allow deforming or stretching images. Image dimension ratios will always be kept. Wagtail also *does not support upscaling*. Small images forced to appear at larger sizes will "max out" at their native dimensions.

More control over the `img` tag

Wagtail provides two shortcuts to give greater control over the `img` element:

1. Adding attributes to the {% image %} tag

Extra attributes can be specified with the syntax `attribute="value"`:

```
{% image page.photo width-400 class="foo" id="bar" %}
```

No validation is performed on attributes added in this way so it's possible to add *src*, *width*, *height* and *alt* of your own that might conflict with those generated by the tag itself.

2. Generating the image "as foo" to access individual properties

Wagtail can assign the image data to another variable using Django's `as` syntax:

```
{% image page.photo width-400 as tmp_photo %}

<img src="{{ tmp_photo.url }}" width="{{ tmp_photo.width }}"
    height="{{ tmp_photo.height }}" alt="{{ page.photo.title }}" class="my-custom-
↪class" />
```

This syntax exposes the underlying image "Rendition" (`tmp_photo`) to the developer. A "Rendition" contains just the information specific to the way you've requested to format the image i.e dimensions and source URL.

If your site defines a custom image model using `AbstractImage`, then any additional fields you add to an image e.g a copyright holder, are **not** part of the image *rendition*, they're part of the image *model*.

Therefore in the above example, if you'd added the field `foo` to your AbstractImage you'd access it using `{{ page.photo.foo }}` not `{{ tmp_photo.foo }}`.

(Due to the links in the database between renditions and their parent image, you could also access it as `{{ tmp_photo.image.foo }}` but this is clearly confusing.)

Note: The image property used for the `src` attribute is actually `image.url`, not `image.src`.

The `attrs` shortcut

You can also use the `attrs` property as a shorthand to output the attributes `src`, `width`, `height` and `alt` in one go:

```
<img {{ tmp_photo.attrs }} class="my-custom-class" />
```

Images embedded in rich text

The information above relates to images defined via image-specific fields in your model, but images can also be embedded arbitrarily in Rich Text fields by the editor (see *Rich Text (HTML)*).

Images embedded in Rich Text fields can't be controlled by the template developer as easily. There are no image objects to work with, so the `{% image %}` template tag can't be used. Instead editors can choose from one of a number of image "Formats" at the point of inserting images into their text.

Wagtail comes with three pre-defined image formats, but more can be defined in Python by the developer. These formats are:

`Full width` Creates an image tag using the filter `width-800` and given the CSS class `full-width`

`Left-aligned` Creates an image tag with the filter `width-500` and given the CSS class `left`

`Right-aligned` Creates an image tag with the filter `width-500` and given the CSS class `right`

Note: The CSS classes added to images do **not** come with any accompanying stylesheets, or inline styles. e.g the `left` class will do nothing, by default. The developer is expected to add these classes to their front end CSS files, to define what exactly `left`, `right` or `full-width` means *to them*.

For more information about image formats, including creating your own, see *Image Formats in the Rich Text Editor*

Using images outside Wagtail

Wagtail provides a way for you to generate external URLs for images in your image library which you can use to display your images on external sites.

Setup

Add an entry in your URLs configuration for `wagtail.wagtailimages.urls`:

```
from wagtail.wagtailimages import urls as wagtailimages_urls


urlpatterns = [
    ...

    url(r'^images/', include(wagtailimages_urls)),
]
```

Generating URLs for images

Once the above setup is done, a button should appear under the image preview on the image edit page. Clicking this button will take you to an interface where you can specify the size you want the image to be, and it will generate a URL and a preview of what the image is going to look like.

The filter box lets you choose how you would like the image to be resized:

`Original` Leaves the image at its original size - no resizing is performed.

`Resize to max` Fit **within** the given dimensions.

The longest edge will be reduced to the equivalent dimension size defined. e.g A portrait image of width 1000, height 2000, treated with the `max` dimensions `1000x500` (landscape) would result in the image shrunk so the *height* was 500 pixels and the width 250.

`Resize to min` **Cover** the given dimensions.

This may result in an image slightly **larger** than the dimensions you specify. e.g A square image of width 2000, height 2000, treated with the `min` dimensions `500x200` (landscape) would have it's height and width changed to 500, i.e matching the width required, but greater than the height.

`Resize to width` Reduces the width of the image to the dimension specified.

`Resize to height` Resize the height of the image to the dimension specified..

`Resize to fill` Resize and **crop** to fill the **exact** dimensions.

This can be particularly useful for websites requiring square thumbnails of arbitrary images. For example, a landscape image of width 2000, height 1000, treated with `fill` dimensions `200x200` would have its height reduced to 200, then its width (ordinarily 400) cropped to 200.

Using the URLs on your website or blog

Once you have generated a URL, you can put it straight into the `src` attribute of an `<img>` tag:

```
<img src="(image url here)">
```

Advanced topics

Custom image model

The `Image` model can be customised, allowing additional fields to be added to images.

To do this, you need to add two models to your project:

- The image model itself that inherits from `wagtail.wagtailimages.models.AbstractImage`. This is where you would add your additional fields
- The renditions model that inherits from `wagtail.wagtailimages.models.AbstractRendition`. This is used to store renditions for the new model.

Here's an example:

```
# models.py
from django.db import models
from django.db.models.signals import pre_delete
from django.dispatch import receiver

from wagtail.wagtailimages.models import Image, AbstractImage, AbstractRendition


class CustomImage(AbstractImage):
    # Add any extra fields to image here

    # eg. To add a caption field:
    # caption = models.CharField(max_length=255)

    admin_form_fields = Image.admin_form_fields + (
        # Then add the field names here to make them appear in the form:
        # 'caption',
    )


class CustomRendition(AbstractRendition):
    image = models.ForeignKey(CustomImage, related_name='renditions')

    class Meta:
        unique_together = (
            ('image', 'filter', 'focal_point_key'),
        )


# Delete the source image file when an image is deleted
@receiver(pre_delete, sender=CustomImage)
def image_delete(sender, instance, **kwargs):
    instance.file.delete(False)


# Delete the rendition image file when a rendition is deleted
@receiver(pre_delete, sender=CustomRendition)
def rendition_delete(sender, instance, **kwargs):
    instance.file.delete(False)
```

Note: If you are using image feature detection, follow these instructions to enable it on your custom image model:

Feature detection and custom image models

Then set the `WAGTAILIMAGES_IMAGE_MODEL` setting to point to it:

```
WAGTAILIMAGES_IMAGE_MODEL = 'images.CustomImage'
```

Migrating from the builtin image model

When changing an existing site to use a custom image model. No images will be copied to the new model automatically. Copying old images to the new model would need to be done manually with a data migration.

Any templates that reference the builtin image model will still continue to work as before but would need to be updated in order to see any new images.

Animated GIF support

Pillow (Wagtail's default image library) doesn't support resizing animated GIFs. If you need animated GIFs in your site, install Wand.

When Wand is installed, Wagtail will automatically start using it for resizing GIF files, and will continue to resize other images with Pillow.

Feature Detection

Wagtail has the ability to automatically detect faces and features inside your images and crop the images to those features.

Feature detection uses OpenCV to detect faces/features in an image when the image is uploaded. The detected features stored internally as a focal point in the `focal_point_{x, y, width, height}` fields on the `Image` model. These fields are used by the `fill` image filter when an image is rendered in a template to crop the image.

Setup

Feature detection requires OpenCV which can be a bit tricky to install as it's not currently pip-installable.

Installing OpenCV on Debian/Ubuntu

Debian and ubuntu provide an apt-get package called `python-opencv`:

```
sudo apt-get install python-opencv python-numpy
```

This will install PyOpenCV into your site packages. If you are using a virtual environment, you need to make sure site packages are enabled or Wagtail will not be able to import PyOpenCV.

Enabling site packages in the virtual environment

If you are not using a virtual envionment, you can skip this step.

Enabling site packages is different depending on whether you are using pyvenv (Python 3.3+ only) or virtualenv to manage your virtual environment.

pyvenv

Go into your pyvenv directory and open the `pyvenv.cfg` file then set `include-system-site-packages` to `true`.

virtualenv

Go into your virtualenv directory and delete a file called `lib/python-x.x/no-global-site-packages.txt`.

Testing the OpenCV installation

You can test that OpenCV can be seen by Wagtail by opening up a python shell (with your virtual environment active) and typing:

```
import cv
```

If you don't see an `ImportError`, it worked. (If you see the error `libdc1394 error: Failed to initialize libdc1394`, this is harmless and can be ignored.)

Switching on feature detection in Wagtail

Once OpenCV is installed, you need to set the `WAGTAILIMAGES_FEATURE_DETECTION_ENABLED` setting to `True`:

```
# settings.py

WAGTAILIMAGES_FEATURE_DETECTION_ENABLED = True
```

Manually running feature detection

Feature detection runs when new images are uploaded in to Wagtail. If you already have images in your Wagtail site and would like to run feature detection on them, you will have to run it manually.

You can manually run feature detection on all images by running the following code in the python shell:

```
from wagtail.wagtailimages.models import Image

for image in Image.objects.all():
    if not image.has_focal_point():
        image.set_focal_point(image.get_suggested_focal_point())
        image.save()
```

Feature detection and custom image models

When using a *Custom image model*, you need to add a signal handler to the model to trigger feature detection whenever a new image is uploaded:

```
# Do feature detection when a user saves an image without a focal point
@receiver(pre_save, sender=CustomImage)
def image_feature_detection(sender, instance, **kwargs):
    # Make sure the image doesn't already have a focal point
    if not instance.has_focal_point():
        # Set the focal point
        instance.set_focal_point(instance.get_suggested_focal_point())
```

Note: This example will always run feature detection regardless of whether the `WAGTAILIMAGES_FEATURE_DETECTION_ENABLED` setting is set.

Add a check for this setting if you still want it to have effect.

Search

Wagtail provides a comprehensive and extensible search interface. In addition, it provides ways to promote search results through "Editor's Picks". Wagtail also collects simple statistics on queries made through the search interface.

Indexing

To make a model searchable, you'll firstly need to add it into the search index. All pages, images and documents are indexed for you and you can start searching them right away.

If you have created some extra fields in a subclass of Page or Image, you may want to add these new fields to the search index too so that a user's search query will match on their content. See *Indexing extra fields* for info on how to do this.

If you have a custom model that you would like to make searchable, see *Indexing custom models*.

Updating the index

If the search index is kept separate from the database (when using Elasticsearch for example), you need to keep them both in sync. There are two ways to do this: using the search signal handlers, or calling the `update_index` command periodically. For best speed and reliability, it's best to use both if possible.

Signal handlers

Changed in version 0.8: Signal handlers are now automatically registered in Django 1.7 and upwards

`wagtailsearch` provides some signal handlers which bind to the save/delete signals of all indexed models. This would automatically add and delete them from all backends you have registered in `WAGTAILSEARCH_BACKENDS`.

If you are using Django version 1.7 or newer, these signal handlers are automatically registered when the `wagtail.wagtailsearch` app is loaded. Otherwise, they must be registered as your application starts up. This can be done by placing the following code in your `urls.py`:

```
# urls.py
from wagtail.wagtailsearch.signal_handlers import register_signal_handlers
register_signal_handlers()
```

The `update_index` command

Wagtail also provides a command for rebuilding the index from scratch.

`./manage.py update_index`

It is recommended to run this command once a week and at the following times:

- whenever any pages have been created through a script (after an import, for example)
- whenever any changes have been made to models or search configuration

The search may not return any results while this command is running, so avoid running it at peak times.

Indexing extra fields

Warning: Indexing extra fields is only supported with ElasticSearch as your backend. If you're using the database backend, any other fields you define via `search_fields` will be ignored.

Fields must be explicitly added to the `search_fields` property of your `Page`-derived model, in order for you to be able to search/filter on them. This is done by overriding `search_fields` to append a list of extra `SearchField`/`FilterField` objects to it.

Example

This creates an `EventPage` model with two fields `description` and `date`. `description` is indexed as a `SearchField` and `date` is indexed as a `FilterField`

```
from wagtail.wagtailsearch import index

class EventPage(Page):
    description = models.TextField()
    date = models.DateField()

    search_fields = Page.search_fields + ( # Inherit search_fields from Page
        index.SearchField('description'),
        index.FilterField('date'),
    )


# Get future events which contain the string "Christmas" in the title or description
>>> EventPage.objects.filter(date__gt=timezone.now()).search("Christmas")
```

`index.SearchField`

These are added to the search index and are used for performing full-text searches on your models. These would usually be text fields.

Options

- **partial_match** (`boolean`) - Setting this to true allows results to be matched on parts of words. For example, this is set on the title field by default so a page titled `Hello World!` will be found if the user only types `Hel` into the search box.
- **boost** (`int/float`) - This allows you to set fields as being more important than others. Setting this to a high number on a field will make pages with matches in that field to be ranked higher. By default, this is set to 2 on the Page title field and 1 on all other fields.
- **es_extra** (`dict`) - This field is to allow the developer to set or override any setting on the field in the ElasticSearch mapping. Use this if you want to make use of any ElasticSearch features that are not yet supported in Wagtail.

index.FilterField

These are added to the search index but are not used for full-text searches. Instead, they allow you to run filters on your search results.

Indexing callables and other attributes

> **Note:** This is not supported in the *Database Backend (default)*

Search/filter fields do not need to be Django fields, they could be any method or attribute on your class.

One use for this is indexing `get_*_display` methods Django creates automatically for fields with choices.

```
from wagtail.wagtailsearch import index

class EventPage(Page):
    IS_PRIVATE_CHOICES = (
        (False, "Public"),
        (True, "Private"),
    )

    is_private = models.BooleanField(choices=IS_PRIVATE_CHOICES)

    search_fields = Page.search_fields + (
        # Index the human-readable string for searching
        index.SearchField('get_is_private_display'),

        # Index the boolean value for filtering
        index.FilterField('is_private'),
    )
```

Callables also provide a way to index fields from related models. In the example from *Inline Panels and Model Clusters*, to index each BookPage by the titles of its related_links:

```
class BookPage(Page):
    # ...
    def get_related_link_titles(self):
        # Get list of titles and concatenate them
        return '\n'.join(self.related_links.all().values_list('title', flat=True))
```

```
    search_fields = Page.search_fields + [
        # ...
        index.SearchField('get_related_link_titles'),
    ]
```

Indexing custom models

Any Django model can be indexed and searched.

To do this, inherit from `index.Indexed` and add some `search_fields` to the model.

```
from wagtail.wagtailsearch import index

class Book(models.Model, index.Indexed):
    title = models.CharField(max_length=255)
    genre = models.CharField(max_length=255, choices=GENRE_CHOICES)
    author = models.ForeignKey(Author)
    published_date = models.DateTimeField()

    search_fields = (
        index.SearchField('title', partial_match=True, boost=10),
        index.SearchField('get_genre_display'),

        index.FilterField('genre'),
        index.FilterField('author'),
        index.FilterField('published_date'),
    )

# As this model doesn't have a search method in its QuerySet, we have to call search␣
↪directly on the backend
>>> from wagtail.wagtailsearch.backends import get_search_backend
>>> s = get_search_backend()

# Run a search for a book by Roald Dahl
>>> roald_dahl = Author.objects.get(name="Roald Dahl")
>>> s.search("chocolate factory", Book.objects.filter(author=roald_dahl))
[<Book: Charlie and the chocolate factory>]
```

Searching

Searching QuerySets

Wagtail search is built on Django's QuerySet API. You should be able to search any Django QuerySet provided the model and the fields being filtered on have been added to the search index.

Searching Pages

Wagtail provides a shortcut for searching pages: the `.search()` QuerySet method. You can call this on any `PageQuerySet`. For example:

```
# Search future EventPages
>>> from wagtail.wagtailcore.models import EventPage
>>> EventPage.objects.filter(date__gt=timezone.now()).search("Hello world!")
```

All other methods of `PageQuerySet` can be used with `search()`. For example:

```
# Search all live EventPages that are under the events index
>>> EventPage.objects.live().descendant_of(events_index).search("Event")
[<EventPage: Event 1>, <EventPage: Event 2>]
```

Note: The `search()` method will convert your `QuerySet` into an instance of one of Wagtail's `SearchResults` classes (depending on backend). This means that you must perform filtering before calling `search()`.

Searching Images, Documents and custom models

Wagtail's document and image models provide a `search` method on their QuerySets, just as pages do:

```
>>> from wagtail.wagtailimages.models import Image

>>> Image.objects.filter(uploaded_by_user=user).search("Hello")
[<Image: Hello>, <Image: Hello world!>]
```

Custom models can be searched by using the `search` method on the search backend directly:

```
>>> from myapp.models import Book
>>> from wagtail.wagtailsearch.backends import get_search_backend

# Search books
>>> s = get_search_backend()
>>> s.search("Great", Book)
[<Book: Great Expectations>, <Book: The Great Gatsby>]
```

You can also pass a QuerySet into the `search` method which allows you to add filters to your search results:

```
>>> from myapp.models import Book
>>> from wagtail.wagtailsearch.backends import get_search_backend

# Search books
>>> s = get_search_backend()
>>> s.search("Great", Book.objects.filter(published_date__year__lt=1900))
[<Book: Great Expectations>]
```

Specifying the fields to search

By default, Wagtail will search all fields that have been indexed using `index.SearchField`.

This can be limited to a certian set of fields using the `fields` keyword argument:

```
# Search just the title field
>>> EventPage.objects.search("Event", fields=["title"])
[<EventPage: Event 1>, <EventPage: Event 2>]
```

Changing search behaviour

Search operator

New in version 1.2.

The search operator specifies how search should behave when the user has typed in multiple search terms. There are two possible values:

- "or" - The results must match at least one term (default for Elasticsearch)
- "and" - The results must match all terms (default for database search)

Both operators have benefits and drawbacks. The "or" operator will return many more results but will likely contain a lot of results that aren't relevent. The "and" operator only returns results that contain all search terms, but require the user to be more precise with their query.

We recommend using the "or" operator when ordering by relevance and the "and" operator when ordering by anything else (note: the database backend doesn't currently support ordering by relevance).

Here's an example of using the "operator" keyword argument:

```
# The database contains a "Thing" model with the following items:
# - Hello world
# - Hello
# - World


# Search with the "or" operator
>>> s = get_search_backend()
>>> s.search("Hello world", Things, operator="or")

# All records returned as they all contain either "hello" or "world"
[<Thing: Hello World>, <Thing: Hello>, <Thing: World>]


# Search with the "and" operator
>>> s = get_search_backend()
>>> s.search("Hello world", Things, operator="and")

# Only "hello world" returned as that's the only item that contains both terms
[<Thing: Hello world>]
```

For page, image and document models, the `operator` keyword argument is also supported on the QuerySet's `search` method:

```
>>> Page.objects.search("Hello world", operator="or")

# All pages containing either "hello" or "world" are returned
[<Page: Hello World>, <Page: Hello>, <Page: World>]
```

Custom ordering

New in version 1.2.

By default, search results are ordered by relevance, if the backend supports it. To preserve the QuerySet's existing ordering, the `order_by_relevance` keyword argument needs to be set to `False` on the `search()` method.

For example:

```
# Get a list of events ordered by date
>>> EventPage.objects.order_by('date').search("Event", order_by_relevance=False)

# Events ordered by date
[<EventPage: Easter>, <EventPage: Halloween>, <EventPage: Christmas>]
```

An example page search view

Here's an example Django view that could be used to add a "search" page to your site:

```
# views.py

from django.shortcuts import render

from wagtail.wagtailcore.models import Page
from wagtail.wagtailsearch.models import Query


def search(request):
    # Search
    search_query = request.GET.get('query', None)
    if search_query:
        search_results = Page.objects.live().search(search_query)

        # Log the query so Wagtail can suggest promoted results
        Query.get(search_query).add_hit()
    else:
        search_results = Page.objects.none()

    # Render template
    return render(request, 'search_results.html', {
        'search_query': search_query,
        'search_results': search_results,
    })
```

And here's a template to go with it:

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

{% block title %}Search{% endblock %}

{% block content %}
    <form action="{% url 'search' %}" method="get">
        <input type="text" name="query" value="{{ search_query }}">
        <input type="submit" value="Search">
    </form>

    {% if search_results %}
        <ul>
            {% for result in search_results %}
                <li>
                    <h4><a href="{% pageurl result %}">{{ result }}</a></h4>
                    {% if result.search_description %}
                        {{ result.search_description|safe }}
                    {% endif %}
```

```
                </li>
            {% endfor %}
        </ul>
    {% elif search_query %}
        No results found
    {% else %}
        Please type something into the search box
    {% endif %}
{% endblock %}
```

Promoted search results

"Promoted search results" allow editors to explicitly link relevant content to search terms, so results pages can contain curated content in addition to results from the search engine.

This functionality is provided by the *wagtailsearchpromotions* contrib module.

Backends

Wagtailsearch has support for multiple backends giving you the choice between using the database for search or an external service such as Elasticsearch.

You can configure which backend to use with the `WAGTAILSEARCH_BACKENDS` setting:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.wagtailsearch.backends.db',
    }
}
```

AUTO_UPDATE

New in version 1.0.

By default, Wagtail will automatically keep all indexes up to date. This could impact performance when editing content, especially if your index is hosted on an external service.

The `AUTO_UPDATE` setting allows you to disable this on a per-index basis:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': ...,
        'AUTO_UPDATE': False,
    }
}
```

If you have disabled auto update, you must run the *update_index* command on a regular basis to keep the index in sync with the database.

ATOMIC_REBUILD

New in version 1.1.

By default (when using the Elasticsearch backend), when the `update_index` command is run, Wagtail deletes the index and rebuilds it from scratch. This causes the search engine to not return results until the rebuild is complete and is also risky as you can't rollback if an error occurs.

Setting the `ATOMIC_REBUILD` setting to `True` makes Wagtail rebuild into a separate index while keep the old index active until the new one is fully built. When the rebuild is finished, the indexes are swapped atomically and the old index is deleted.

> **Warning:** Experimental feature
>
> This feature is currently experimental. Please use it with caution.

BACKEND

Here's a list of backends that Wagtail supports out of the box.

Database Backend (default)

`wagtail.wagtailsearch.backends.db`

Changed in version 1.1: Before 1.1, the full path to the backend class had to be specified: `wagtail.wagtailsearch.backends.db.DBSearch`

The database backend is very basic and is intended only to be used in development and on small sites. It cannot order results by relevance making it not very useful when searching a large amount of pages.

It also doesn't support:

- Searching on fields in subclasses of `Page` (unless the class is being searched directly)
- *Indexing callables and other attributes*
- Converting accented characters to ASCII

If any of these features are important to you, we recommend using Elasticsearch instead.

Elasticsearch Backend

`wagtail.wagtailsearch.backends.elasticsearch`

Changed in version 1.1: Before 1.1, the full path to the backend class had to be specified: `wagtail.wagtailsearch.backends.elasticsearch.ElasticSearch`

Prerequisites are the Elasticsearch service itself and, via pip, the elasticsearch-py package:

```
pip install elasticsearch
```

The backend is configured in settings:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch',
        'URLS': ['http://localhost:9200'],
        'INDEX': 'wagtail',
        'TIMEOUT': 5,
```

```
    }
}
```

Other than `BACKEND` the keys are optional and default to the values shown. In addition, any other keys are passed directly to the Elasticsearch constructor as case-sensitive keyword arguments (e.g. `'max_retries': 1`).

If you prefer not to run an Elasticsearch server in development or production, there are many hosted services available, including Searchly, who offer a free account suitable for testing and development. To use Searchly:

- Sign up for an account at dashboard.searchly.com/users/sign_up
- Use your Searchly dashboard to create a new index, e.g. 'wagtaildemo'
- Note the connection URL from your Searchly dashboard
- Configure `URLS` and `INDEX` in the Elasticsearch entry in `WAGTAILSEARCH_BACKENDS`
- Run `./manage.py update_index`

Rolling Your Own

Wagtail search backends implement the interface defined in `wagtail/wagtail/wagtailsearch/backends/base.py`. At a minimum, the backend's `search()` method must return a collection of objects or `model.objects.none()`. For a fully-featured search backend, examine the Elasticsearch backend code in `elasticsearch.py`.

Indexing

To make objects searchable, they firstly need to be added to the search index. This involves configuring the models/fields that you would like to index (this is done for you for Pages, Images and Documents) and then actually inserting them into the index.

See *Updating the index* for information on how to keep the objects in your search index in sync with the objects in your database.

If you have created some extra fields in a subclass of `Page` or `Image`, you may want to add these new fields to the search index too so a users search query will match on their content. See *Indexing extra fields*.

If you have a custom model which doesn't derive from `Page` or `Image` that you would like to make searchable, see *Indexing custom models*.

Searching

Wagtail provides an API for performing search queries on your models. You can also perform search queries on Django QuerySets.

See *Searching*.

Backends

Wagtail provides two backends for storing the search index and performing search queries: Elasticsearch and the database. It's also possible to roll your own search backend.

See *Backends*

Snippets

Snippets are pieces of content which do not necessitate a full webpage to render. They could be used for making secondary content, such as headers, footers, and sidebars, editable in the Wagtail admin. Snippets are models which do not inherit the `Page` class and are thus not organized into the Wagtail tree, but can still be made editable by assigning panels and identifying the model as a snippet with the `register_snippet` class decorator.

Snippets lack many of the features of pages, such as being orderable in the Wagtail admin or having a defined URL, so decide carefully if the content type you would want to build into a snippet might be more suited to a page.

Snippet Models

Here's an example snippet from the Wagtail demo website:

```
from django.db import models

from wagtail.wagtailadmin.edit_handlers import FieldPanel
from wagtail.wagtailsnippets.models import register_snippet

...

@register_snippet
class Advert(models.Model):
    url = models.URLField(null=True, blank=True)
    text = models.CharField(max_length=255)

    panels = [
        FieldPanel('url'),
        FieldPanel('text'),
    ]

    def __str__(self):              # __unicode__ on Python 2
        return self.text
```

The `Advert` model uses the basic Django model class and defines two properties: text and URL. The editing interface is very close to that provided for `Page`-derived models, with fields assigned in the panels property. Snippets do not use multiple tabs of fields, nor do they provide the "save as draft" or "submit for moderation" features.

`@register_snippet` tells Wagtail to treat the model as a snippet. The `panels` list defines the fields to show on the snippet editing page. It's also important to provide a string representation of the class through `def __str__(self):` so that the snippet objects make sense when listed in the Wagtail admin.

Including Snippets in Template Tags

The simplest way to make your snippets available to templates is with a template tag. This is mostly done with vanilla Django, so perhaps reviewing Django's documentation for django custom template tags will be more helpful. We'll go over the basics, though, and make note of any considerations to make for Wagtail.

First, add a new python file to a `templatetags` folder within your app. The demo website, for instance uses the path `wagtaildemo/demo/templatetags/demo_tags.py`. We'll need to load some Django modules and our app's models and ready the `register` decorator:

```
from django import template
from demo.models import *

register = template.Library()
```

```
...

# Advert snippets
@register.inclusion_tag('demo/tags/adverts.html', takes_context=True)
def adverts(context):
    return {
        'adverts': Advert.objects.all(),
        'request': context['request'],
    }
```

`@register.inclusion_tag()` takes two variables: a template and a boolean on whether that template should be passed a request context. It's a good idea to include request contexts in your custom template tags, since some Wagtail-specific template tags like `pageurl` need the context to work properly. The template tag function could take arguments and filter the adverts to return a specific model, but for brevity we'll just use `Advert.objects.all()`.

Here's what's in the template used by the template tag:

```
{% for advert in adverts %}
  <p>
    <a href="{{ advert.url }}">
      {{ advert.text }}
    </a>
  </p>
{% endfor %}
```

Then in your own page templates, you can include your snippet template tag with:

```
{% block content %}

  ...

  {% adverts %}

{% endblock %}
```

Binding Pages to Snippets

In the above example, the list of adverts is a fixed list, displayed as part of the template independently of the page content. This might be what you want for a common panel in a sidebar, say - but in other scenarios you may wish to refer to a snippet within page content. This can be done by defining a foreign key to the snippet model within your page model, and adding a `SnippetChooserPanel` to the page's `content_panels` definitions. For example, if you wanted to be able to specify an advert to appear on `BookPage`:

```
from wagtail.wagtailsnippets.edit_handlers import SnippetChooserPanel
# ...
class BookPage(Page):
    advert = models.ForeignKey(
        'demo.Advert',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+'
    )
```

```
BookPage.content_panels = [
    SnippetChooserPanel('advert'),
    # ...
]
```

The snippet could then be accessed within your template as `page.advert`.

To attach multiple adverts to a page, the `SnippetChooserPanel` can be placed on an inline child object of `BookPage`, rather than on `BookPage` itself. Here this child model is named `BookPageAdvertPlacement` (so called because there is one such object for each time that an advert is placed on a BookPage):

```
from django.db import models

from wagtail.wagtailcore.models import Page
from wagtail.wagtailsnippets.edit_handlers import SnippetChooserPanel

from modelcluster.fields import ParentalKey

...

class BookPageAdvertPlacement(Orderable, models.Model):
    page = ParentalKey('demo.BookPage', related_name='advert_placements')
    advert = models.ForeignKey('demo.Advert', related_name='+')

    class Meta:
        verbose_name = "Advert Placement"
        verbose_name_plural = "Advert Placements"

    panels = [
        SnippetChooserPanel('advert'),
    ]

    def __str__(self):              # __unicode__ on Python 2
        return self.page.title + " -> " + self.advert.text


class BookPage(Page):
    ...


BookPage.content_panels = [
    InlinePanel('advert_placements', label="Adverts"),
    # ...
]
```

These child objects are now accessible through the page's `advert_placements` property, and from there we can access the linked Advert snippet as `advert`. In the template for `BookPage`, we could include the following:

```
{% for advert_placement in page.advert_placements.all %}
  <p><a href="{{ advert_placement.advert.url }}">{{ advert_placement.advert.text }}</
↪a></p>
{% endfor %}
```

Making Snippets Searchable

If a snippet model inherits from `wagtail.wagtailsearch.index.Indexed`, as described in *Indexing custom models*, Wagtail will automatically add a search box to the chooser interface for that snippet type. For example, the

`Advert` snippet could be made searchable as follows:

```
...

from wagtail.wagtailsearch import index

...

@register_snippet
class Advert(models.Model, index.Indexed):
    url = models.URLField(null=True, blank=True)
    text = models.CharField(max_length=255)

    panels = [
        FieldPanel('url'),
        FieldPanel('text'),
    ]

    search_fields = [
        index.SearchField('text', partial_match=True),
    ]
```

Tagging snippets

Adding tags to snippets is very similar to adding tags to pages. The only difference is that `taggit.manager.TaggableManager` should be used in the place of `ClusterTaggableManager`.

```
from modelcluster.fields import ParentalKey
from taggit.models import TaggedItemBase
from taggit.managers import TaggableManager

class AdvertTag(TaggedItemBase):
    content_object = ParentalKey('demo.Advert', related_name='tagged_items')

@register_snippet
class Advert(models.Model):
    ...
    tags = TaggableManager(through=BlogPageTag, blank=True)

    panels = [
        ...
        FieldPanel('tags'),
    ]
```

The *documentation on tagging pages* has more information on how to use tags in views.

Freeform page content using StreamField

StreamField provides a content editing model suitable for pages that do not follow a fixed structure - such as blog posts or news stories, where the text may be interspersed with subheadings, images, pull quotes and video, and perhaps more specialised content types such as maps and charts (or, for a programming blog, code snippets). In this model, these different content types are represented as a sequence of 'blocks', which can be repeated and arranged in any order.

For further background on StreamField, and why you would use it instead of a rich text field for the article body, see the blog post Rich text fields and faster horses.

StreamField also offers a rich API to define your own block types, ranging from simple collections of sub-blocks (such as a 'person' block consisting of first name, surname and photograph) to completely custom components with their own editing interface. Within the database, the StreamField content is stored as JSON, ensuring that the full informational content of the field is preserved, rather than just an HTML representation of it.

Using StreamField

`StreamField` is a model field that can be defined within your page model like any other field:

```
from django.db import models

from wagtail.wagtailcore.models import Page
from wagtail.wagtailcore.fields import StreamField
from wagtail.wagtailcore import blocks
from wagtail.wagtailadmin.edit_handlers import FieldPanel, StreamFieldPanel
from wagtail.wagtailimages.blocks import ImageChooserBlock

class BlogPage(Page):
    author = models.CharField(max_length=255)
    date = models.DateField("Post date")
    body = StreamField([
        ('heading', blocks.CharBlock(classname="full title")),
        ('paragraph', blocks.RichTextBlock()),
        ('image', ImageChooserBlock()),
    ])

BlogPage.content_panels = [
    FieldPanel('author'),
    FieldPanel('date'),
    StreamFieldPanel('body'),
]
```

Note: StreamField is not backwards compatible with other field types such as RichTextField; if you migrate an existing field to StreamField, the existing data will be lost.

The parameter to `StreamField` is a list of (name, block_type) tuples; 'name' is used to identify the block type within templates and the internal JSON representation (and should follow standard Python conventions for variable names: lower-case and underscores, no spaces) and 'block_type' should be a block definition object as described below. (Alternatively, `StreamField` can be passed a single `StreamBlock` instance - see *Structural block types*.)

This defines the set of available block types that can be used within this field. The author of the page is free to use these blocks as many times as desired, in any order.

Basic block types

All block types accept the following optional keyword arguments:

`default` The default value that a new 'empty' block should receive.

`label` The label to display in the editor interface when referring to this block - defaults to a prettified version of the block name (or, in a context where no name is assigned - such as within a `ListBlock` - the empty string).

`icon` The name of the icon to display for this block type in the menu of available block types. For a list of icon names, see the Wagtail style guide, which can be enabled by adding `wagtail.contrib.wagtailstyleguide` to your project's `INSTALLED_APPS`.

`template` The path to a Django template that will be used to render this block on the front end. See *Template rendering*.

The basic block types provided by Wagtail are as follows:

CharBlock

`wagtail.wagtailcore.blocks.CharBlock`

A single-line text input. The following keyword arguments are accepted:

`required` (default: True) If true, the field cannot be left blank.

`max_length`, `min_length` Ensures that the string is at most or at least the given length.

`help_text` Help text to display alongside the field.

TextBlock

`wagtail.wagtailcore.blocks.TextBlock`

A multi-line text input. As with `CharBlock`, the keyword arguments `required`, `max_length`, `min_length` and `help_text` are accepted.

URLBlock

`wagtail.wagtailcore.blocks.URLBlock`

A single-line text input that validates that the string is a valid URL. The keyword arguments `required`, `max_length`, `min_length` and `help_text` are accepted.

BooleanBlock

`wagtail.wagtailcore.blocks.BooleanBlock`

A checkbox. The keyword arguments `required` and `help_text` are accepted. As with Django's `BooleanField`, a value of `required=True` (the default) indicates that the checkbox must be ticked in order to proceed; for a checkbox that can be ticked or unticked, you must explicitly pass in `required=False`.

DateBlock

`wagtail.wagtailcore.blocks.DateBlock`

A date picker. The keyword arguments `required` and `help_text` are accepted.

TimeBlock

`wagtail.wagtailcore.blocks.TimeBlock`

A time picker. The keyword arguments `required` and `help_text` are accepted.

DateTimeBlock

`wagtail.wagtailcore.blocks.DateTimeBlock`

A combined date / time picker. The keyword arguments `required` and `help_text` are accepted.

RichTextBlock

`wagtail.wagtailcore.blocks.RichTextBlock`

A WYSIWYG editor for creating formatted text including links, bold / italics etc.

RawHTMLBlock

`wagtail.wagtailcore.blocks.RawHTMLBlock`

A text area for entering raw HTML which will be rendered unescaped in the page output. The keyword arguments `required`, `max_length`, `min_length` and `help_text` are accepted.

> **Warning:** When this block is in use, there is nothing to prevent editors from inserting malicious scripts into the page, including scripts that would allow the editor to acquire administrator privileges when another administrator views the page. Do not use this block unless your editors are fully trusted.

ChoiceBlock

`wagtail.wagtailcore.blocks.ChoiceBlock`

A dropdown select box for choosing from a list of choices. The following keyword arguments are accepted:

`choices` A list of choices, in any format accepted by Django's `choices` parameter for model fields: https://docs.djangoproject.com/en/stable/ref/models/fields/#field-choices

`required` (default: True) If true, the field cannot be left blank.

`help_text` Help text to display alongside the field.

`ChoiceBlock` can also be subclassed to produce a reusable block with the same list of choices everywhere it is used. For example, a block definition such as:

```
blocks.ChoiceBlock(choices=[
    ('tea', 'Tea'),
    ('coffee', 'Coffee'),
], icon='cup')
```

could be rewritten as a subclass of ChoiceBlock:

```
class DrinksChoiceBlock(blocks.ChoiceBlock):
    choices = [
        ('tea', 'Tea'),
        ('coffee', 'Coffee'),
    ]

    class Meta:
        icon = 'cup'
```

`StreamField` definitions can then refer to `DrinksChoiceBlock()` in place of the full `ChoiceBlock` definition.

PageChooserBlock

`wagtail.wagtailcore.blocks.PageChooserBlock`

A control for selecting a page object, using Wagtail's page browser. The following keyword arguments are accepted:

`required` (default: True) If true, the field cannot be left blank.

`can_choose_root` (default: False) If true, the editor can choose the tree root as a page. Normally this would be undesirable, since the tree root is never a usable page, but in some specialised cases it may be appropriate; for example, a block providing a feed of related articles could use a PageChooserBlock to select which subsection articles will be taken from, with the root corresponding to 'everywhere'.

DocumentChooserBlock

`wagtail.wagtaildocs.blocks.DocumentChooserBlock`

A control to allow the editor to select an existing document object, or upload a new one. The keyword argument `required` is accepted.

ImageChooserBlock

`wagtail.wagtailimages.blocks.ImageChooserBlock`

A control to allow the editor to select an existing image, or upload a new one. The keyword argument `required` is accepted.

SnippetChooserBlock

`wagtail.wagtailsnippets.blocks.SnippetChooserBlock`

A control to allow the editor to select a snippet object. Requires one positional argument: the snippet class to choose from. The keyword argument `required` is accepted.

EmbedBlock

`wagtail.wagtailembeds.blocks.EmbedBlock`

A field for the editor to enter a URL to a media item (such as a YouTube video) to appear as embedded media on the page. The keyword arguments `required`, `max_length`, `min_length` and `help_text` are accepted.

Structural block types

In addition to the basic block types above, it is possible to define new block types made up of sub-blocks: for example, a 'person' block consisting of sub-blocks for first name, surname and image, or a 'carousel' block consisting of an unlimited number of image blocks. These structures can be nested to any depth, making it possible to have a structure containing a list, or a list of structures.

StructBlock

`wagtail.wagtailcore.blocks.StructBlock`

A block consisting of a fixed group of sub-blocks to be displayed together. Takes a list of (name, block_definition) tuples as its first argument:

```
('person', blocks.StructBlock([
    ('first_name', blocks.CharBlock(required=True)),
    ('surname', blocks.CharBlock(required=True)),
    ('photo', ImageChooserBlock()),
    ('biography', blocks.RichTextBlock()),
], icon='user'))
```

Alternatively, the list of sub-blocks can be provided in a subclass of StructBlock:

```
class PersonBlock(blocks.StructBlock):
    first_name = blocks.CharBlock(required=True)
    surname = blocks.CharBlock(required=True)
    photo = ImageChooserBlock()
    biography = blocks.RichTextBlock()

    class Meta:
        icon = 'user'
```

The `Meta` class supports the properties `default`, `label`, `icon` and `template`; these have the same meanings as when they are passed to the block's constructor.

This defines `PersonBlock()` as a block type that can be re-used as many times as you like within your model definitions:

```
body = StreamField([
    ('heading', blocks.CharBlock(classname="full title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
    ('person', PersonBlock()),
])
```

To customise the styling of the block as it appears in the page editor, your subclass can specify a `form_classname` attribute in `Meta` to override the default value of `struct-block`:

```
class PersonBlock(blocks.StructBlock):
    first_name = blocks.CharBlock(required=True)
    surname = blocks.CharBlock(required=True)
    photo = ImageChooserBlock()
    biography = blocks.RichTextBlock()

    class Meta:
        icon = 'user'
        form_classname = 'person-block struct-block'
```

You can then provide custom CSS for this block, targeted at the specified classname, by using the `insert_editor_css` hook (see *Hooks*). For more extensive customisations that require changes to the HTML markup as well, you can override the `form_template` attribute in `Meta`.

ListBlock

`wagtail.wagtailcore.blocks.ListBlock`

A block consisting of many sub-blocks, all of the same type. The editor can add an unlimited number of sub-blocks, and re-order and delete them. Takes the definition of the sub-block as its first argument:

```
('ingredients_list', blocks.ListBlock(blocks.CharBlock(label="Ingredient")))
```

Any block type is valid as the sub-block type, including structural types:

```
('ingredients_list', blocks.ListBlock(blocks.StructBlock([
    ('ingredient', blocks.CharBlock(required=True)),
    ('amount', blocks.CharBlock()),
])))
```

StreamBlock

`wagtail.wagtailcore.blocks.StreamBlock`

A block consisting of a sequence of sub-blocks of different types, which can be mixed and reordered in any order. Used as the overall mechanism of the StreamField itself, but can also be nested or used within other structural block types. Takes a list of (name, block_definition) tuples as its first argument:

```
('carousel', blocks.StreamBlock(
    [
        ('image', ImageChooserBlock()),
        ('quotation', blocks.StructBlock([
            ('text', blocks.TextBlock()),
            ('author', blocks.CharBlock()),
        ])),
        ('video', EmbedBlock()),
    ],
    icon='cogs'
))
```

As with StructBlock, the list of sub-blocks can also be provided as a subclass of StreamBlock:

```
class CarouselBlock(blocks.StreamBlock):
    image = ImageChooserBlock()
    quotation = blocks.StructBlock([
        ('text', blocks.TextBlock()),
        ('author', blocks.CharBlock()),
    ])
    video = EmbedBlock()

    class Meta:
        icon='cogs'
```

Since `StreamField` accepts an instance of `StreamBlock` as a parameter, in place of a list of block types, this makes it possible to re-use a common set block types without repeating definitions:

```
class HomePage(Page):
    carousel = StreamField(CarouselBlock())
```

Template rendering

The simplest way to render the contents of a StreamField into your template is to output it as a variable, like any other field:

```
{{ page.body }}
```

This will render each block of the stream in turn, wrapped in a `<div class="block-my_block_name">` element (where `my_block_name` is the block name given in the StreamField definition). If you wish to provide your own HTML markup, you can instead iterate over the field's value to access each block in turn:

```
<article>
    {% for block in page.body %}
        <section>{{ block }}</section>
    {% endfor %}
</article>
```

For more control over the rendering of specific block types, each block object provides `block_type` and `value` properties:

```
<article>
    {% for block in page.body %}
        {% if block.block_type == 'heading' %}
            <h1>{{ block.value }}</h1>
        {% else %}
            <section class="block-{{ block.block_type }}">
                {{ block }}
            </section>
        {% endif %}
    {% endfor %}
</article>
```

Each block type provides its own front-end HTML rendering mechanism, and this is used for the output of `{{ block }}`. For most simple block types, such as CharBlock, this will simply output the field's value, but others will provide their own HTML markup; for example, a ListBlock will output the list of child blocks as a `<ul>` element (with each child wrapped in an `<li>` element and rendered using the child block's own HTML rendering).

To override this with your own custom HTML rendering, you can pass a `template` argument to the block, giving the filename of a template file to be rendered. This is particularly useful for custom block types derived from StructBlock, as the default StructBlock rendering is simple and somewhat generic:

```
('person', blocks.StructBlock(
    [
        ('first_name', blocks.CharBlock(required=True)),
        ('surname', blocks.CharBlock(required=True)),
        ('photo', ImageChooserBlock()),
        ('biography', blocks.RichTextBlock()),
    ],
    template='myapp/blocks/person.html',
    icon='user'
))
```

Or, when defined as a subclass of StructBlock:

```
class PersonBlock(blocks.StructBlock):
    first_name = blocks.CharBlock(required=True)
    surname = blocks.CharBlock(required=True)
    photo = ImageChooserBlock()
```

```
    biography = blocks.RichTextBlock()

    class Meta:
        template = 'myapp/blocks/person.html'
        icon = 'user'
```

Within the template, the block value is accessible as the variable `value`:

```
{% load wagtailimages_tags %}

<div class="person">
    {% image value.photo width-400 %}
    <h2>{{ value.first_name }} {{ value.surname }}</h2>
    {{ value.bound_blocks.biography.render }}
</div>
```

The line `value.bound_blocks.biography.render` warrants further explanation. While blocks such as RichTextBlock are aware of their own rendering, the actual block *values* (as returned when accessing properties of a StructBlock, such as `value.biography`), are just plain Python values such as strings. To access the block's proper HTML rendering, you must retrieve the 'bound block' - an object which has access to both the rendering method and the value - via the `bound_blocks` property. To pass additional context variables to the template, block subclasses can override the `get_context` method:

```
import datetime

class EventBlock(blocks.StructBlock):
    title = blocks.CharBlock(required=True)
    date = blocks.DateBlock(required=True)

    def get_context(self, value):
        context = super(EventBlock, self).get_context(value)
        context['is_happening_today'] = (value['date'] == datetime.date.today())
        return context

    class Meta:
        template = 'myapp/blocks/event.html'
```

In this example, the variable `is_happening_today` will be made available within the block template.

Custom block types

If you need to implement a custom UI, or handle a datatype that is not provided by Wagtail's built-in block types (and cannot built up as a structure of existing fields), it is possible to define your own custom block types. For further guidance, refer to the source code of Wagtail's built-in block classes.

For block types that simply wrap an existing Django form field, Wagtail provides an abstract class `wagtail.wagtailcore.blocks.FieldBlock` as a helper. Subclasses just need to set a `field` property that returns the form field object:

```
class IPAddressBlock(FieldBlock):
    def __init__(self, required=True, help_text=None, **kwargs):
        self.field = forms.GenericIPAddressField(required=required, help_text=help_
↪text)
        super(IPAddressBlock, self).__init__(**kwargs)
```

Migrations

StreamField definitions within migrations

As with any model field in Django, any changes to a model definition that affect a StreamField will result in a migration file that contains a 'frozen' copy of that field definition. Since a StreamField definition is more complex than a typical model field, there is an increased likelihood of definitions from your project being imported into the migration - which would cause problems later on if those definitions are moved or deleted.

To mitigate this, StructBlock, StreamBlock and ChoiceBlock implement additional logic to ensure that any subclasses of these blocks are deconstructed to plain instances of StructBlock, StreamBlock and ChoiceBlock - in this way, the migrations avoid having any references to your custom class definitions. This is possible because these block types provide a standard pattern for inheritance, and know how to reconstruct the block definition for any subclass that follows that pattern.

If you subclass any other block class, such as `FieldBlock`, you will need to either keep that class definition in place for the lifetime of your project, or implement a custom deconstruct method that expresses your block entirely in terms of classes that are guaranteed to remain in place. Similarly, if you customise a StructBlock, StreamBlock or ChoiceBlock subclass to the point where it can no longer be expressed as an instance of the basic block type - for example, if you add extra arguments to the constructor - you will need to provide your own `deconstruct` method.

Migrating RichTextFields to StreamField

If you change an existing RichTextField to a StreamField, and create and run migrations as normal, the migration will complete with no errors, since both fields use a text column within the database. However, StreamField uses a JSON representation for its data, and so the existing text needs to be converted with a data migration in order to become accessible again. For this to work, the StreamField needs to include a RichTextBlock as one of the available block types. The field can then be converted by creating a new migration (`./manage.py makemigrations --empty myapp`) and editing it as follows (in this example, the 'body' field of the `demo.BlogPage` model is being converted to a StreamField with a RichTextBlock named `rich_text`):

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models, migrations
from wagtail.wagtailcore.rich_text import RichText


def convert_to_streamfield(apps, schema_editor):
    BlogPage = apps.get_model("demo", "BlogPage")
    for page in BlogPage.objects.all():
        if page.body.raw_text and not page.body:
            page.body = [('rich_text', RichText(page.body.raw_text))]
            page.save()


def convert_to_richtext(apps, schema_editor):
    BlogPage = apps.get_model("demo", "BlogPage")
    for page in BlogPage.objects.all():
        if page.body.raw_text is None:
            raw_text = ''.join([
                child.value.source for child in page.body
                if child.block_type == 'rich_text'
            ])
            page.body = raw_text
            page.save()
```

```
class Migration(migrations.Migration):

    dependencies = [
        # leave the dependency line from the generated migration intact!
        ('demo', '0001_initial'),
    ]

    operations = [
        migrations.RunPython(
            convert_to_streamfield,
            convert_to_richtext,
        ),
    ]
```

Advanced topics

Configuring Django for Wagtail

To install Wagtail completely from scratch, create a new Django project and an app within that project. For instructions on these tasks, see Writing your first Django app. Your project directory will look like the following:

```
myproject/
    myproject/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    myapp/
        __init__.py
        models.py
        tests.py
        admin.py
        views.py
    manage.py
```

From your app directory, you can safely remove `admin.py` and `views.py`, since Wagtail will provide this functionality for your models. Configuring Django to load Wagtail involves adding modules and variables to `settings.py` and URL configuration to `urls.py`. For a more complete view of what's defined in these files, see Django Settings and Django URL Dispatcher.

What follows is a settings reference which skips many boilerplate Django settings. If you just want to get your Wagtail install up quickly without fussing with settings at the moment, see *Ready to Use Example Configuration Files*.

Middleware (`settings.py`)

```
MIDDLEWARE_CLASSES = (
  'django.middleware.common.CommonMiddleware',
  'django.contrib.sessions.middleware.SessionMiddleware',
  'django.middleware.csrf.CsrfViewMiddleware',
  'django.contrib.auth.middleware.AuthenticationMiddleware',
  'django.contrib.messages.middleware.MessageMiddleware',
```

```
  'django.middleware.clickjacking.XFrameOptionsMiddleware',

  'wagtail.wagtailcore.middleware.SiteMiddleware',

  'wagtail.wagtailredirects.middleware.RedirectMiddleware',
)
```

Wagtail requires several common Django middleware modules to work and cover basic security. Wagtail provides its own middleware to cover these tasks:

SiteMiddleware Wagtail routes pre-defined hosts to pages within the Wagtail tree using this middleware.

RedirectMiddleware Wagtail provides a simple interface for adding arbitrary redirects to your site and this module makes it happen.

Apps (settings.py)

```
INSTALLED_APPS = (
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.messages',
  'django.contrib.staticfiles',

  'compressor',
  'taggit',
  'modelcluster',

  'wagtail.wagtailcore',
  'wagtail.wagtailadmin',
  'wagtail.wagtaildocs',
  'wagtail.wagtailsnippets',
  'wagtail.wagtailusers',
  'wagtail.wagtailimages',
  'wagtail.wagtailembeds',
  'wagtail.wagtailsearch',
  'wagtail.wagtailsites',
  'wagtail.wagtailredirects',
  'wagtail.wagtailforms',

  'myapp',  # your own app
)
```

Wagtail requires several Django app modules, third-party apps, and defines several apps of its own. Wagtail was built to be modular, so many Wagtail apps can be omitted to suit your needs. Your own app (here `myapp`) is where you define your models, templates, static assets, template tags, and other custom functionality for your site.

Third-Party Apps

compressor Static asset combiner and minifier for Django. Compressor also enables for the use of preprocessors. See Compressor Documentation.

taggit Tagging framework for Django. This is used internally within Wagtail for image and document tagging and is available for your own models as well. See *Tagging* for a Wagtail model recipe or the Taggit Documentation.

modelcluster Extension of Django ForeignKey relation functionality, which is used in Wagtail pages for on-the-fly related object creation. For more information, see *Inline Panels and Model Clusters* or the django-modelcluster github project page.

Wagtail Apps

wagtailcore The core functionality of Wagtail, such as the `Page` class, the Wagtail tree, and model fields.

wagtailadmin The administration interface for Wagtail, including page edit handlers.

wagtaildocs The Wagtail document content type.

wagtailsnippets Editing interface for non-Page models and objects. See *Snippets*.

wagtailusers User editing interface.

wagtailimages The Wagtail image content type.

wagtailembeds Module governing oEmbed and Embedly content in Wagtail rich text fields. See *Inserting videos into body content*.

wagtailsearch Search framework for Page content. See search.

wagtailredirects Admin interface for creating arbitrary redirects on your site.

wagtailforms Models for creating forms on your pages and viewing submissions. See *Form builder*.

Settings Variables (`settings.py`)

Site Name

```
WAGTAIL_SITE_NAME = 'Stark Industries Skunkworks'
```

This is the human-readable name of your Wagtail install which welcomes users upon login to the Wagtail admin.

Search

```
# Override the search results template for wagtailsearch
WAGTAILSEARCH_RESULTS_TEMPLATE = 'myapp/search_results.html'
WAGTAILSEARCH_RESULTS_TEMPLATE_AJAX = 'myapp/includes/search_listing.html'

# Replace the search backend
WAGTAILSEARCH_BACKENDS = {
  'default': {
    'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch',
    'INDEX': 'myapp'
  }
}
```

The search settings customise the search results templates as well as choosing a custom backend for search. For a full explanation, see search.

Embeds

Wagtail uses the oEmbed standard with a large but not comprehensive number of "providers" (Youtube, Vimeo, etc.). You can also use a different embed backend by providing an Embedly key or replacing the embed backend by writing your own embed finder function.

```
WAGTAILEMBEDS_EMBED_FINDER = 'myapp.embeds.my_embed_finder_function'
```

Use a custom embed finder function, which takes a URL and returns a dict with metadata and embeddable HTML. Refer to the `wagtail.wagtailemebds.embeds` module source for more information and examples.

```
# not a working key, get your own!
WAGTAILEMBEDS_EMBEDLY_KEY = '253e433d59dc4d2xa266e9e0de0cb830'
```

Providing an API key for the Embedly service will use that as a embed backend, with a more extensive list of providers, as well as analytics and other features. For more information, see Embedly.

To use Embedly, you must also install their Python module:

```
pip install embedly
```

Images

```
WAGTAILIMAGES_IMAGE_MODEL = 'myapp.MyImage'
```

This setting lets you provide your own image model for use in Wagtail, which might extend the built-in `AbstractImage` class or replace it entirely.

Email Notifications

```
WAGTAILADMIN_NOTIFICATION_FROM_EMAIL = 'wagtail@myhost.io'
```

Wagtail sends email notifications when content is submitted for moderation, and when the content is accepted or rejected. This setting lets you pick which email address these automatic notifications will come from. If omitted, Django will fall back to using the `DEFAULT_FROM_EMAIL` variable if set, and `webmaster@localhost` if not.

Wagtail update notifications

```
WAGTAIL_ENABLE_UPDATE_CHECK = True
```

For admins only, Wagtail performs a check on the dashboard to see if newer releases are available. This also provides the Wagtail team with the hostname of your Wagtail site. If you'd rather not receive update notifications, or if you'd like your site to remain unknown, you can disable it with this setting.

Private Pages

```
PASSWORD_REQUIRED_TEMPLATE = 'myapp/password_required.html'
```

This is the path to the Django template which will be used to display the "password required" form when a user accesses a private page. For more details, see the *Private pages* documentation.

Other Django Settings Used by Wagtail

```
ALLOWED_HOSTS
APPEND_SLASH
AUTH_USER_MODEL
BASE_URL
CACHES
DEFAULT_FROM_EMAIL
INSTALLED_APPS
MEDIA_ROOT
SESSION_COOKIE_DOMAIN
SESSION_COOKIE_NAME
SESSION_COOKIE_PATH
STATIC_URL
TEMPLATE_CONTEXT_PROCESSORS
USE_I18N
```

For information on what these settings do, see Django Settings.

URL Patterns

```
from django.contrib import admin

from wagtail.wagtailcore import urls as wagtail_urls
from wagtail.wagtailadmin import urls as wagtailadmin_urls
from wagtail.wagtaildocs import urls as wagtaildocs_urls
from wagtail.wagtailsearch import urls as wagtailsearch_urls

urlpatterns = [
  url(r'^django-admin/', include(admin.site.urls)),

  url(r'^admin/', include(wagtailadmin_urls)),
  url(r'^search/', include(wagtailsearch_urls)),
  url(r'^documents/', include(wagtaildocs_urls)),

  # Optional URL for including your own vanilla Django urls/views
  url(r'', include('myapp.urls')),

  # For anything not caught by a more specific rule above, hand over to
  # Wagtail's serving mechanism
  url(r'', include(wagtail_urls)),
]
```

This block of code for your project's `urls.py` does a few things:

- Load the vanilla Django admin interface to `/django-admin/`
- Load the Wagtail admin and its various apps
- Dispatch any vanilla Django apps you're using other than Wagtail which require their own URL configuration (this is optional, since Wagtail might be all you need)
- Lets Wagtail handle any further URL dispatching.

That's not everything you might want to include in your project's URL configuration, but it's what's necessary for Wagtail to flourish.

Ready to Use Example Configuration Files

These two files should reside in your project directory (`myproject/myproject/`).

settings.py

```
import os

PROJECT_ROOT = os.path.join(os.path.dirname(__file__), '..', '..')

DEBUG = True
TEMPLATE_DEBUG = DEBUG

ADMINS = (
    # ('Your Name', 'your_email@example.com'),
)

MANAGERS = ADMINS

# Default to dummy email backend. Configure dev/production/local backend
# as per https://docs.djangoproject.com/en/dev/topics/email/#email-backends
EMAIL_BACKEND = 'django.core.mail.backends.dummy.EmailBackend'

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'myprojectdb',
        'USER': 'postgres',
        'PASSWORD': '',
        'HOST': '',  # Set to empty string for localhost.
        'PORT': '',  # Set to empty string for default.
        'CONN_MAX_AGE': 600,  # number of seconds database connections should persist␣
↪for
    }
}

# Hosts/domain names that are valid for this site; required if DEBUG is False
# See https://docs.djangoproject.com/en/1.5/ref/settings/#allowed-hosts
ALLOWED_HOSTS = []

# Local time zone for this installation. Choices can be found here:
# http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
# although not all choices may be available on all operating systems.
# On Unix systems, a value of None will cause Django to use the same
# timezone as the operating system.
# If running in a Windows environment this must be set to the same as your
# system time zone.
TIME_ZONE = 'Europe/London'

# Language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = 'en-gb'

SITE_ID = 1

# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
```

```
USE_I18N = True

# If you set this to False, Django will not format dates, numbers and
# calendars according to the current locale.
# Note that with this set to True, Wagtail will fall back on using numeric dates
# in date fields, as opposed to 'friendly' dates like "24 Sep 2013", because
# Python's strptime doesn't support localised month names: https://code.djangoproject.
↪com/ticket/13339
USE_L10N = False

# If you set this to False, Django will not use timezone-aware datetimes.
USE_TZ = True

# Absolute filesystem path to the directory that will hold user-uploaded files.
# Example: "/home/media/media.lawrence.com/media/"
MEDIA_ROOT = os.path.join(PROJECT_ROOT, 'media')

# URL that handles the media served from MEDIA_ROOT. Make sure to use a
# trailing slash.
# Examples: "http://media.lawrence.com/media/", "http://example.com/media/"
MEDIA_URL = '/media/'

# Absolute path to the directory static files should be collected to.
# Don't put anything in this directory yourself; store your static files
# in apps' "static/" subdirectories and in STATICFILES_DIRS.
# Example: "/home/media/media.lawrence.com/static/"
STATIC_ROOT = os.path.join(PROJECT_ROOT, 'static')

# URL prefix for static files.
# Example: "http://media.lawrence.com/static/"
STATIC_URL = '/static/'

# List of finder classes that know how to find static files in
# various locations.
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
    'compressor.finders.CompressorFinder',
)

# Make this unique, and don't share it with anybody.
SECRET_KEY = 'change-me'

# List of callables that know how to import templates from various sources.
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
)

MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',

    'wagtail.wagtailcore.middleware.SiteMiddleware',
```

```
    'wagtail.wagtailredirects.middleware.RedirectMiddleware',
)

from django.conf import global_settings
TEMPLATE_CONTEXT_PROCESSORS = global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
    'django.core.context_processors.request',
)

ROOT_URLCONF = 'myproject.urls'

# Python dotted path to the WSGI application used by Django's runserver.
WSGI_APPLICATION = 'wagtaildemo.wsgi.application'

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'compressor',
    'taggit',
    'modelcluster',

    'wagtail.wagtailcore',
    'wagtail.wagtailadmin',
    'wagtail.wagtaildocs',
    'wagtail.wagtailsnippets',
    'wagtail.wagtailusers',
    'wagtail.wagtailimages',
    'wagtail.wagtailembeds',
    'wagtail.wagtailsearch',
    'wagtail.wagtailredirects',
    'wagtail.wagtailforms',

    'myapp',
)

EMAIL_SUBJECT_PREFIX = '[Wagtail] '

INTERNAL_IPS = ('127.0.0.1', '10.0.2.2')

# A sample logging configuration. The only tangible logging
# performed by this configuration is to send an email to
# the site admins on every HTTP 500 error when DEBUG=False.
# See http://docs.djangoproject.com/en/dev/topics/logging for
# more details on how to customize your logging configuration.
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse'
        }
    },
    'handlers': {
        'mail_admins': {
```

```
            'level': 'ERROR',
            'filters': ['require_debug_false'],
            'class': 'django.utils.log.AdminEmailHandler'
        }
    },
    'loggers': {
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': True,
        },
    }
}


# WAGTAIL SETTINGS

# This is the human-readable name of your Wagtail install
# which welcomes users upon login to the Wagtail admin.
WAGTAIL_SITE_NAME = 'My Project'

# Override the search results template for wagtailsearch
# WAGTAILSEARCH_RESULTS_TEMPLATE = 'myapp/search_results.html'
# WAGTAILSEARCH_RESULTS_TEMPLATE_AJAX = 'myapp/includes/search_listing.html'

# Replace the search backend
#WAGTAILSEARCH_BACKENDS = {
#  'default': {
#    'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch',
#    'INDEX': 'myapp'
#  }
#}

# Wagtail email notifications from address
# WAGTAILADMIN_NOTIFICATION_FROM_EMAIL = 'wagtail@myhost.io'

# If you want to use Embedly for embeds, supply a key
# (this key doesn't work, get your own!)
# WAGTAILEMBEDS_EMBEDLY_KEY = '253e433d59dc4d2xa266e9e0de0cb830'
```

urls.py

```
from django.conf.urls import patterns, include, url
from django.conf.urls.static import static
from django.views.generic.base import RedirectView
from django.contrib import admin
from django.conf import settings
import os.path

from wagtail.wagtailcore import urls as wagtail_urls
from wagtail.wagtailadmin import urls as wagtailadmin_urls
from wagtail.wagtaildocs import urls as wagtaildocs_urls
from wagtail.wagtailsearch import urls as wagtailsearch__urls


urlpatterns = patterns('',
```

```
    url(r'^django-admin/', include(admin.site.urls)),

    url(r'^admin/', include(wagtailadmin_urls)),
    url(r'^search/', include(wagtailsearch_urls)),
    url(r'^documents/', include(wagtaildocs_urls)),

    # For anything not caught by a more specific rule above, hand over to
    # Wagtail's serving mechanism
    url(r'', include(wagtail_urls)),
)


if settings.DEBUG:
    from django.contrib.staticfiles.urls import staticfiles_urlpatterns

    urlpatterns += staticfiles_urlpatterns() # tell gunicorn where static files are␣
↪in dev mode
    urlpatterns += static(settings.MEDIA_URL + 'images/', document_root=os.path.
↪join(settings.MEDIA_ROOT, 'images'))
    urlpatterns += patterns('',
        (r'^favicon\.ico$', RedirectView.as_view(url=settings.STATIC_URL + 'myapp/
↪images/favicon.ico'))
    )
```

Deploying Wagtail

On your server

Wagtail is straightforward to deploy on modern Linux-based distributions, but see the section on *performance* for the non-Python services we recommend.

Our current preferences are for Nginx, Gunicorn and supervisor on Debian, but Wagtail should run with any of the combinations detailed in Django's deployment documentation.

On Gondor

Gondor specialise in Python hosting. They provide Redis and Elasticsearch, which are two of the services we recommend for high-performance production sites. Gondor have written a comprehensive tutorial on running your Wagtail site on their platform, at gondor.io/blog/2014/02/14/how-run-wagtail-cms-gondor/.

On other PAASs and IAASs

We know of Wagtail sites running on Heroku, Digital Ocean and elsewhere. If you have successfully installed Wagtail on your platform or infrastructure, please *contribute* your notes to this documentation!

Performance

Wagtail is designed for speed, both in the editor interface and on the front-end, but if you want even better performance or you need to handle very high volumes of traffic, here are some tips on eking out the most from your installation.

Editor interface

We have tried to minimise external dependencies for a working installation of Wagtail, in order to make it as simple as possible to get going. However, a number of default settings can be configured for better performance:

Cache

We recommend Redis as a fast, persistent cache. Install Redis through your package manager (on Debian or Ubuntu: `sudo apt-get install redis-server`), add `django-redis` to your `requirements.txt`, and enable it as a cache backend:

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': '127.0.0.1:6379',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}
```

Without a persistent cache, Wagtail will recreate all compressible assets at each server start, e.g. when any files change under `./manage.py runserver`.

Search

Wagtail has strong support for Elasticsearch - both in the editor interface and for users of your site - but can fall back to a database search if Elasticsearch isn't present. Elasticsearch is faster and more powerful than the Django ORM for text search, so we recommend installing it or using a hosted service like Searchly.

Once the Elasticsearch server is installed and running. Install the `elasticsearch` Python module with:

```
pip install elasticsearch
```

then add the following to your settings:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch.ElasticSearch',
        'INDEX': '{{ project_name }}',
    },
}
```

Once Elasticsearch is configured, you can index any existing content you may have:

```
./manage.py update_index
```

Database

Wagtail is tested on SQLite, and should work on other Django-supported database backends, but we recommend PostgreSQL for production use.

Public users

Caching proxy

To support high volumes of traffic with excellent response times, we recommend a caching proxy. Both Varnish and Squid have been tested in production. Hosted proxies like Cloudflare should also work well.

> Wagtail supports automatic cache invalidation for Varnish/Squid. See *Frontend cache invalidator* for more information.

Internationalisation

This document describes the internationalisation features of Wagtail and how to create multi-lingual sites.

Wagtail uses Django's Internationalisation framework so most of the steps are the same as other Django projects.

Contents

- *Internationalisation*
  - *Wagtail admin translations*
  - *Changing the primary language of your Wagtail installation*
  - *Creating sites with multiple languages*
    - *Enabling multiple language support*
    - *Serving different languages from different URLs*
    - *Translating templates*
    - *Translating content*
    - *Other approaches*

Wagtail admin translations

The Wagtail admin backend has been translated into many different languages. You can find a list of currently available translations on Wagtail's Transifex page. (Note: if you're using an old version of Wagtail, this page may not accurately reflect what languages you have available).

If your language isn't listed on that page, you can easily contribute new languages or correct mistakes. Sign up and submit changes to Transifex. Translation updates are typically merged into an official release within one month of being submitted.

Changing the primary language of your Wagtail installation

The default language of Wagtail is `en-us` (American English). You can change this by tweaking a couple of Django settings:

- Make sure USE_I18N is set to `True`
- Set LANGUAGE_CODE to your websites' primary language

If there is a translation available for your language, the Wagtail admin backend should now be in the language you've chosen.

Creating sites with multiple languages

You can create sites with multiple language support by leveraging Django's translation features.

This section of the documentation will show you how to use Django's translation features with Wagtail and also describe a couple of methods for storing/retrieving translated content using Wagtail pages.

Enabling multiple language support

Firstly, make sure the USE_I18N Django setting is set to `True`.

To enable multi-language support, add `django.middleware.locale.LocaleMiddleware` to your `MIDDLEWARE_CLASSES`:

```
MIDDLEWARE_CLASSES = (
    ...

    'django.middleware.locale.LocaleMiddleware',
)
```

This middleware class looks at the user's browser language and sets the language of the site accordingly.

Serving different languages from different URLs

Just enabling the multi-language support in Django sometimes may not be enough. By default, Django will serve different languages of the same page with the same URL. This has a couple of drawbacks:

- Users cannot change language without changing their browser settings
- It may not work well with various caching setups (as content varies based on browser settings)

Django's `i18n_patterns` feature, when enabled, prefixes the URLs with the language code (eg `/en/about-us`). Users are forwarded to their preferred version, based on browser language, when they first visit the site.

This feature is enabled through the project's root URL configuration. Just put the views you would like to have this enabled for in an `i18n_patterns` list and append that to the other URL patterns:

```
# mysite/urls.py

from django.conf.urls import include, url
from django.conf.urls.i18n import i18n_patterns
from django.conf import settings
from django.contrib import admin

from wagtail.wagtailadmin import urls as wagtailadmin_urls
from wagtail.wagtaildocs import urls as wagtaildocs_urls
from wagtail.wagtailcore import urls as wagtail_urls


urlpatterns = [
    url(r'^django-admin/', include(admin.site.urls)),

    url(r'^admin/', include(wagtailadmin_urls)),
    url(r'^documents/', include(wagtaildocs_urls)),
]
```

```
urlpatterns += i18n_patterns('',
    # These URLs will have /<language_code>/ appended to the beginning

    url(r'^search/$', 'search.views.search', name='search'),

    url(r'', include(wagtail_urls)),
)
```

You can implement switching between languages by changing the part at the beginning of the URL. As each language has its own URL, it also works well with just about any caching setup.

Translating templates

Static text in templates needs to be marked up in a way that allows Django's `makemessages` command to find and export the strings for translators and also allow them to switch to translated versions on the when the template is being served.

As Wagtail uses Django's templates, inserting this markup and the workflow for exporting and translating the strings is the same as any other Django project.

See: https://docs.djangoproject.com/en/1.8/topics/i18n/translation/#internationalization-in-template-code

Translating content

The most common approach for translating content in Wagtail is to duplicate each translatable text field, providing a separate field for each language.

This section will describe how to implement this method manually but there is a third party module you can use, wagtail modeltranslation, which may be quicker if it meets your needs.

Duplicating the fields in your model

For each field you would like to be translatable, duplicate it for every language you support and suffix it with the language code:

```
class BlogPage(Page):

    title_fr = models.CharField(max_length=255)

    body_en = StreamField(...)
    body_fr = StreamField(...)

    # Language-independent fields don't need to be duplicated
    thumbnail_image = models.ForeignKey('wagtailimages.image', ...)
```

Note: We only define the French version of the `title` field as Wagtail already provides the English version for us.

Organising the fields in the admin interface

You can either put all the fields with their translations next to each other on the "content" tab or put the translations for other languages on different tabs.

See *Customising the tabbed interface* for information on how to add more tabs to the admin interface.

Accessing the fields from the template

In order for the translations to be shown on the site frontend, the correct field needs to be used in the template based on what language the client has selected.

Having to add language checks every time you display a field in a template, could make your templates very messy. Here's a little trick that will allow you to implement this while keeping your templates and model code clean.

You can use a snippet like the following to add accessor fields on to your page model. These accessor fields will point at the field that contains the language the user has selected.

Copy this into your project and make sure it's imported in any `models.py` files that contain a `Page` with translated fields. It will require some modification to support different languages.

```
from django.utils import translation

class TranslatedField(object):
    def __init__(self, en_field, fr_field):
        self.en_field = en_field
        self.fr_field = fr_field

    def __get__(self, instance, owner):
        en = getattr(instance, self.en_field)
        fr = getattr(instance, self.fr_field)

        if translation.get_language() == 'fr':
            return fr
        else:
            return en
```

Then, for each translated field, create an instance of `TranslatedField` with a nice name (as this is the name your templates will reference).

For example, here's how we would apply this to the above `BlogPage` model:

```
class BlogPage(Page):
    ...

    translated_title = TranslatedField(
        'title',
        'title_fr',
    )
    body = TranslatedField(
        'body_en',
        'body_fr',
    )
```

Finally, in the template, reference the accessors instead of the underlying database fields:

```
{{ page.translated_title }}

{{ page.body }}
```

Other approaches

Creating a multilingual site (by duplicating the page tree)

This tutorial will show you a method of creating multilingual sites in Wagtail by duplicating the page tree.

For example:

```
/
    en/
        about/
        contact/
    fr/
        about/
        contact/
```

The root page

The root page (`/`) should detect the browsers language and forward them to the correct language homepage (`/en/`, `/fr/`). This page should sit at the site root (where the homepage would normally be).

We must set Django's `LANGUAGES` setting so we don't redirect non English/French users to pages that don't exist.

```
# settings.py
LANGUAGES = (
    ('en', _("English")),
    ('fr', _("French")),
)

# models.py
from django.utils import translation
from django.http import HttpResponseRedirect

from wagtail.wagtailcore.models import Page


class LanguageRedirectionPage(Page):

    def serve(self, request):
        # This will only return a language that is in the LANGUAGES Django setting
        language = translation.get_language_from_request(request)

        return HttpResponseRedirect(self.url + language + '/')
```

Linking pages together

It may be useful to link different versions of the same page together to allow the user to easily switch between languages. But we don't want to increase the burden on the editor too much so ideally, editors should only need to link one of the pages to the other versions and the links between the other versions should be created implicitly.

As this behaviour needs to be added to all page types that would be translated, its best to put this behaviour in a mixin.

Here's an example of how this could be implemented (with English as the main language and French/Spanish as alternative languages):

```
class TranslatablePageMixin(models.Model):
    # One link for each alternative language
    # These should only be used on the main language page (english)
    french_link = models.ForeignKey(Page, null=True, on_delete=models.SET_NULL,␣
↪blank=True, related_name='+')
    spanish_link = models.ForeignKey(Page, null=True, on_delete=models.SET_NULL,␣
↪blank=True, related_name='+')
```

```
    def get_language(self):
        """
        This returns the language code for this page.
        """
        # Look through ancestors of this page for its language homepage
        # The language homepage is located at depth 3
        language_homepage = self.get_ancestors(inclusive=True).get(depth=3)

        # The slug of language homepages should always be set to the language code
        return language_homepage.slug


    # Method to find the main language version of this page
    # This works by reversing the above links

    def english_page(self):
        """
        This finds the english version of this page
        """
        language = self.get_language()

        if language == 'en':
            return self
        elif language == 'fr':
            return type(self).objects.filter(french_link=self).first().specific
        elif language == 'es':
            return type(self).objects.filter(spanish_link=self).first().specific


    # We need a method to find a version of this page for each alternative language.
    # These all work the same way. They firstly find the main version of the page
    # (english), then from there they can just follow the link to the correct page.

    def french_page(self):
        """
        This finds the french version of this page
        """
        english_page = self.english_page()

        if english_page and english_page.french_link:
            return english_page.french_link.specific

    def spanish_page(self):
        """
        This finds the spanish version of this page
        """
        english_page = self.english_page()

        if english_page and english_page.spanish_link:
            return english_page.spanish_link.specific

    class Meta:
        abstract = True


class AboutPage(Page, TranslatablePageMixin):
    ...
```

```
class ContactPage(Page, TranslatablePageMixin):
    ...
```

You can make use of these methods in your template by doing:

```
{% if page.english_page and page.get_language != 'en' %}
    <a href="{{ page.english_page.url }}">{% trans "View in English" %}</a>
{% endif %}

{% if page.french_page and page.get_language != 'fr' %}
    <a href="{{ page.french_page.url }}">{% trans "View in French" %}</a>
{% endif %}

{% if page.spanish_page and page.get_language != 'es' %}
    <a href="{{ page.spanish_page.url }}">{% trans "View in Spanish" %}</a>
{% endif %}
```

Private pages

Users with publish permission on a page can set it to be private by clicking the 'Privacy' control in the top right corner of the page explorer or editing interface, and setting a password. Users visiting this page, or any of its subpages, will be prompted to enter a password before they can view the page.

Private pages work on Wagtail out of the box - the site implementer does not need to do anything to set them up. However, the default "password required" form is only a bare-bones HTML page, and site implementers may wish to replace this with a page customised to their site design.

Setting up a global "password required" page

By setting `PASSWORD_REQUIRED_TEMPLATE` in your Django settings file, you can specify the path of a template which will be used for all "password required" forms on the site (except for page types that specifically override it - see below):

```
PASSWORD_REQUIRED_TEMPLATE = 'myapp/password_required.html'
```

This template will receive the same set of context variables that the blocked page would pass to its own template via `get_context()` - including `page` to refer to the page object itself - plus the following additional variables (which override any of the page's own context variables of the same name):

- **form** - A Django form object for the password prompt; this will contain a field named `password` as its only visible field. A number of hidden fields may also be present, so the page must loop over `form.hidden_fields` if not using one of Django's rendering helpers such as `form.as_p`.
- **action_url** - The URL that the password form should be submitted to, as a POST request.

A basic template suitable for use as `PASSWORD_REQUIRED_TEMPLATE` might look like this:

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Password required</title>
    </head>
    <body>
        <h1>Password required</h1>
        <p>You need a password to access this page.</p>
```

```
        <form action="{{ action_url }}" method="POST">
            {% csrf_token %}

            {{ form.non_field_errors }}

            <div>
                {{ form.password.errors }}
                {{ form.password.label_tag }}
                {{ form.password }}
            </div>

            {% for field in form.hidden_fields %}
                {{ field }}
            {% endfor %}
            <input type="submit" value="Continue" />
        </form>
    </body>
</html>
```

Setting a "password required" page for a specific page type

The attribute `password_required_template` can be defined on a page model to use a custom template for the "password required" view, for that page type only. For example, if a site had a page type for displaying embedded videos along with a description, it might choose to use a custom "password required" template that displays the video description as usual, but shows the password form in place of the video embed.

```
class VideoPage(Page):
    ...
    password_required_template = 'video/password_required.html'
```

Customising Wagtail

Customising the page editing interface

Customising the tabbed interface

New in version 1.0.

As standard, Wagtail organises panels into three tabs: 'Content', 'Promote' and 'Settings'. Depending on the requirements of your site, you may wish to customise this for specific page types - for example, adding an additional tab for sidebar content. This can be done by specifying an `edit_handler` property on the page model. For example:

```
from wagtail.wagtailadmin.edit_handlers import TabbedInterface, ObjectList

class BlogPage(Page):
    # field definitions omitted

    content_panels = [
        FieldPanel('title', classname="full title"),
        FieldPanel('date'),
        FieldPanel('body', classname="full"),
    ]
    sidebar_content_panels = [
        SnippetChooserPanel('advert'),
```

```
        InlinePanel('related_links', label="Related links"),
    ]

    edit_handler = TabbedInterface([
        ObjectList(content_panels, heading='Content'),
        ObjectList(sidebar_content_panels, heading='Sidebar content'),
        ObjectList(Page.promote_panels, heading='Promote'),
        ObjectList(Page.settings_panels, heading='Settings', classname="settings"),
    ])
```

Rich Text (HTML)

Wagtail provides a general-purpose WYSIWYG editor for creating rich text content (HTML) and embedding media such as images, video, and documents. To include this in your models, use the `RichTextField` function when defining a model field:

```
from wagtail.wagtailcore.fields import RichTextField
from wagtail.wagtailadmin.edit_handlers import FieldPanel


class BookPage(Page):
    book_text = RichTextField()

    content_panels = Page.content_panels + [
        FieldPanel('body', classname="full"),
    ]
```

`RichTextField` inherits from Django's basic `TextField` field, so you can pass any field parameters into `RichTextField` as if using a normal Django field. This field does not need a special panel and can be defined with `FieldPanel`.

However, template output from `RichTextField` is special and need to be filtered to preserve embedded content. See *Rich text (filter)*.

If you're interested in extending the capabilities of the Wagtail WYSIWYG editor (`hallo.js`), See *Extending the WYSIWYG Editor (hallo.js)*.

Extending the WYSIWYG Editor (`hallo.js`)

To inject JavaScript into the Wagtail page editor, see the *insert_editor_js* hook. Once you have the hook in place and your `hallo.js` plugin loads into the Wagtail page editor, use the following JavaScript to register the plugin with `hallo.js`.

```
registerHalloPlugin(name, opts);
```

`hallo.js` plugin names are prefixed with the `"IKS."` namespace, but the `name` you pass into `registerHalloPlugin()` should be without the prefix. `opts` is an object passed into the plugin.

For information on developing custom `hallo.js` plugins, see the project's page: https://github.com/bergie/hallo

Image Formats in the Rich Text Editor

On loading, Wagtail will search for any app with the file `image_formats.py` and execute the contents. This provides a way to customise the formatting options shown to the editor when inserting images in the `RichTextField`

editor.

As an example, add a "thumbnail" format:

```
# image_formats.py
from wagtail.wagtailimages.formats import Format, register_image_format

register_image_format(Format('thumbnail', 'Thumbnail', 'richtext-image thumbnail',
↪'max-120x120'))
```

To begin, import the the `Format` class, `register_image_format` function, and optionally `unregister_image_format` function. To register a new `Format`, call the `register_image_format` with the `Format` object as the argument. The `Format` class takes the following constructor arguments:

`name` The unique key used to identify the format. To unregister this format, call `unregister_image_format` with this string as the only argument.

`label` The label used in the chooser form when inserting the image into the `RichTextField`.

`classnames` The string to assign to the `class` attribute of the generated `<img>` tag.

> **Note:** Any class names you provide must have CSS rules matching them written separately, as part of the front end CSS code. Specifying a `classnames` value of `left` will only ensure that class is output in the generated markup, it won't cause the image to align itself left.

`filter_spec` The string specification to create the image rendition. For more, see the *Using images in templates*.

To unregister, call `unregister_image_format` with the string of the `name` of the `Format` as the only argument.

Custom branding

In your projects with Wagtail, you may wish to replace elements such as the Wagtail logo within the admin interface with your own branding. This can be done through Django's template inheritance mechanism, along with the django-overextends package.

Install `django-overextends` with `pip install django-overextends` (or add `django-overextends` to your project's requirements file), and add `'overextends'` to your project's `INSTALLED_APPS`. You now need to create a `templates/wagtailadmin/` folder within one of your apps - this may be an existing one, or a new one created for this purpose, for example, `dashboard`. This app must be registered in `INSTALLED_APPS` before `wagtail.wagtailadmin`:

```
INSTALLED_APPS = (
  # ...

  'overextends',
  'dashboard',

  'wagtail.wagtailcore',
  'wagtail.wagtailadmin',

  # ...
)
```

The template blocks that are available to be overridden are as follows:

branding_logo

To replace the default logo, create a template file `dashboard/templates/wagtailadmin/base.html` that overrides the block `branding_logo`:

```
{% overextends "wagtailadmin/base.html" %}

{% block branding_logo %}
    <img src="{{ STATIC_URL }}images/custom-logo.svg" alt="Custom Project" width="80"␣
↪/>
{% endblock %}
```

branding_favicon

To replace the favicon displayed when viewing admin pages, create a template file `dashboard/templates/wagtailadmin/admin_base.html` that overrides the block `branding_favicon`:

```
{% overextends "wagtailadmin/admin_base.html" %}

{% block branding_favicon %}
    <link rel="shortcut icon" href="{{ STATIC_URL }}images/favicon.ico" />
{% endblock %}
```

branding_login

To replace the login message, create a template file `dashboard/templates/wagtailadmin/login.html` that overrides the block `branding_login`:

```
{% overextends "wagtailadmin/login.html" %}

{% block branding_login %}Sign in to Frank's Site{% endblock %}
```

branding_welcome

To replace the welcome message on the dashboard, create a template file `dashboard/templates/wagtailadmin/home.html` that overrides the block `branding_welcome`:

```
{% overextends "wagtailadmin/home.html" %}

{% block branding_welcome %}Welcome to Frank's Site{% endblock %}
```

Third-party tutorials

Warning: The following list is a collection of tutorials and development notes from third-party developers. Some of the older links may not apply to the latest Wagtail versions.

- Adding a Twitter Widget for Wagtail's new StreamField (2 April 2015)
- Working With Wagtail: Menus (22 January 2015)

- Upgrading Wagtail to use Django 1.7 locally using vagrant (10 December 2014)
- Wagtail redirect page. Can link to page, URL and document (24 September 2014)
- Outputing JSON for a model with properties and db fields in Wagtail/Django (24 September 2014)
- Bi-lingual website using Wagtail CMS (17 September 2014)
- Wagtail CMS – Lesser known features (12 September 2014)
- Wagtail notes: stateful on/off hallo.js plugins (9 August 2014)
- Add some blockquote buttons to Wagtail CMS' WYSIWYG Editor (24 July 2014)
- Adding Bread Crumbs to the front end in Wagtail CMS (1 July 2014)
- Extending hallo.js using Wagtail hooks (9 July 2014)
- Wagtail notes: custom tabs per page type (10 May 2014)
- Wagtail notes: managing redirects as pages (10 May 2014)
- Wagtail notes: dynamic templates per page (10 May 2014)
- Wagtail notes: type-constrained PageChooserPanel (9 May 2014)
- How to Run the Wagtail CMS on Gondor (14 February 2014)
- The first Wagtail tutorial (13 February 2014)

Tip: We are working on a collection of Wagtail tutorials and best practices. Please tweet @WagtailCMS or contact us directly to share your Wagtail HOWTOs, development notes or site launches.

Jinja2 template support

Wagtail supports Jinja2 templating for all front end features. More information on each of the template tags below can be found in the *Writing templates* documentation.

Configuring Django

Django needs to be configured to support Jinja2 templates. As the Wagtail admin is written using regular Django templates, Django has to be configured to use both templating engines. Wagtail supports the Jinja2 backend that ships with Django 1.8 and above. Add the following configuration to the `TEMPLATES` setting for your app:

```
TEMPLATES = [
    # ...
    {
        'BACKEND': 'django.template.backends.jinja2.Jinja2',
        'APP_DIRS': True,
        'OPTIONS': {
            'extensions': [
                'wagtail.wagtailcore.templatetags.jinja2.core',
                'wagtail.wagtailadmin.templatetags.jinja2.userbar',
                'wagtail.wagtailimages.templatetags.jinja2.images',
            ],
        },
    }
]
```

Jinja templates must be placed in a `jinja2/` directory in your app. The template for an `EventPage` model in an `events` app should be created at `events/jinja2/events/event_page.html`.

By default, the Jinja environment does not have any Django functions or filters. The Django documentation has more information on configuring Jinja for Django.

`self` in templates

In Django templates, `self` can be used to refer to the current page, stream block, or field panel. In Jinja, `self` is reserved for internal use. When writing Jinja templates, use `page` to refer to pages, `value` for stream blocks, and `field_panel` for field panels.

Template functions & filters

`pageurl()`

Generate a URL for a Page instance:

```
<a href="{{ pageurl(page.more_information) }}">More information</a>
```

See *pageurl* for more information

`slugurl()`

Generate a URL for a Page with a slug:

```
<a href="{{ slugurl("about") }}">About us</a>
```

See *slugurl* for more information

`image()`

Resize an image, and print an `<img>` tag:

```
{# Print an image tag #}
{{ image(page.header_image, "fill-1024x200", class="header-image") }}

{# Resize an image #}
{% set background=image(page.background_image, "max-1024x1024") %}
<div class="wrapper" style="background-image: url({{ background.url }});">
```

See *Using images in templates* for more information

`|richtext`

Transform Wagtail's internal HTML representation, expanding internal references to pages and images.

```
{{ page.body|richtext }}
```

See *Rich text (filter)* for more information

wagtailuserbar()

Output the Wagtail contextual flyout menu for editing pages from the front end

```
{{ wagtailuserbar() }}
```

See *Wagtail User Bar* for more information

Reference

Pages

Wagtail requires a little careful setup to define the types of content that you want to present through your website. The basic unit of content in Wagtail is the `Page`, and all of your page-level content will inherit basic webpage-related properties from it. But for the most part, you will be defining content yourself, through the construction of Django models using Wagtail's `Page` as a base.

Wagtail organizes content created from your models in a tree, which can have any structure and combination of model objects in it. Wagtail doesn't prescribe ways to organize and interrelate your content, but here we've sketched out some strategies for organizing your models.

The presentation of your content, the actual webpages, includes the normal use of the Django template system. We'll cover additional functionality that Wagtail provides at the template level later on.

Theory

Introduction to Trees

If you're unfamiliar with trees as an abstract data type, you might want to review the concepts involved.

As a web developer, though, you probably already have a good understanding of trees as filesystem directories or paths. Wagtail pages can create the same structure, as each page in the tree has its own URL path, like so:

```
/
    people/
        nien-nunb/
        laura-roslin/
    events/
        captain-picard-day/
        winter-wrap-up/
```

The Wagtail admin interface uses the tree to organize content for editing, letting you navigate up and down levels in the tree through its Explorer menu. This method of organization is a good place to start in thinking about your own Wagtail models.

Nodes and Leaves

It might be handy to think of the `Page`-derived models you want to create as being one of two node types: parents and leaves. Wagtail isn't prescriptive in this approach, but it's a good place to start if you're not experienced in structuring your own content types.

Nodes

Parent nodes on the Wagtail tree probably want to organize and display a browse-able index of their descendants. A blog, for instance, needs a way to show a list of individual posts.

A Parent node could provide its own function returning its descendant objects.

```
class EventPageIndex(Page):
    # ...
    def events(self):
        # Get list of live event pages that are descendants of this page
        events = EventPage.objects.live().descendant_of(self)

        # Filter events list to get ones that are either
        # running now or start in the future
        events = events.filter(date_from__gte=date.today())

        # Order by date
        events = events.order_by('date_from')

        return events
```

This example makes sure to limit the returned objects to pieces of content which make sense, specifically ones which have been published through Wagtail's admin interface (`live()`) and are children of this node (`descendant_of(self)`). By setting a `subpage_types` class property in your model, you can specify which models are allowed to be set as children, and by setting a `parent_page_types` class property, you can specify which models are allowed to be parents of this page model. Wagtail will allow any `Page`-derived model by default. Regardless, it's smart for a parent model to provide an index filtered to make sense.

Leaves

Leaves are the pieces of content itself, a page which is consumable, and might just consist of a bunch of properties. A blog page leaf might have some body text and an image. A person page leaf might have a photo, a name, and an address.

It might be helpful for a leaf to provide a way to back up along the tree to a parent, such as in the case of breadcrumbs navigation. The tree might also be deep enough that a leaf's parent won't be included in general site navigation.

The model for the leaf could provide a function that traverses the tree in the opposite direction and returns an appropriate ancestor:

```
class EventPage(Page):
    # ...
    def event_index(self):
        # Find closest ancestor which is an event index
        return self.get_ancestors().type(EventIndexPage).last()
```

If defined, `subpage_types` and `parent_page_types` will also limit the parent models allowed to contain a leaf. If not, Wagtail will allow any combination of parents and leafs to be associated in the Wagtail tree. Like with index pages, it's a good idea to make sure that the index is actually of the expected model to contain the leaf.

Other Relationships

Your `Page`-derived models might have other interrelationships which extend the basic Wagtail tree or depart from it entirely. You could provide functions to navigate between siblings, such as a "Next Post" link on a blog

page (`post->post->post`). It might make sense for subtrees to interrelate, such as in a discussion forum (`forum->post->replies`) Skipping across the hierarchy might make sense, too, as all objects of a certain model class might interrelate regardless of their ancestors (`events = EventPage.objects.all`). It's largely up to the models to define their interrelations, the possibilities are really endless.

Anatomy of a Wagtail Request

For going beyond the basics of model definition and interrelation, it might help to know how Wagtail handles requests and constructs responses. In short, it goes something like:

1. Django gets a request and routes through Wagtail's URL dispatcher definitions
2. Wagtail checks the hostname of the request to determine which `Site` record will handle this request.
3. Starting from the root page of that site, Wagtail traverses the page tree, calling the `route()` method and letting each page model decide whether it will handle the request itself or pass it on to a child page.
4. The page responsible for handling the request returns a `RouteResult` object from `route()`, which identifies the page along with any additional `args`/`kwargs` to be passed to `serve()`.
5. Wagtail calls `serve()`, which constructs a context using `get_context()`
6. `serve()` finds a template to pass it to using `get_template()`
7. A response object is returned by `serve()` and Django responds to the requester.

You can apply custom behavior to this process by overriding `Page` class methods such as `route()` and `serve()` in your own models. For examples, see *Recipes*.

Recipes

Overriding the `serve()` Method

Wagtail defaults to serving *`Page`*-derived models by passing a reference to the page object to a Django HTML template matching the model's name, but suppose you wanted to serve something other than HTML? You can override the *`serve()`* method provided by the *`Page`* class and handle the Django request and response more directly.

Consider this example from the Wagtail demo site's `models.py`, which serves an `EventPage` object as an iCal file if the `format` variable is set in the request:

```
class EventPage(Page):
    ...
    def serve(self, request):
        if "format" in request.GET:
            if request.GET['format'] == 'ical':
                # Export to ical format
                response = HttpResponse(
                    export_event(self, 'ical'),
                    content_type='text/calendar',
                )
                response['Content-Disposition'] = 'attachment; filename=' + self.slug␣
↪+ '.ics'
                return response
            else:
                # Unrecognised format error
                message = 'Could not export event\n\nUnrecognised format: ' + request.
↪GET['format']
                return HttpResponse(message, content_type='text/plain')
```

```
        else:
            # Display event page as usual
            return super(EventPage, self).serve(request)
```

serve() takes a Django request object and returns a Django response object. Wagtail returns a `TemplateResponse` object with the template and context which it generates, which allows middleware to function as intended, so keep in mind that a simpler response object like a `HttpResponse` will not receive these benefits.

With this strategy, you could use Django or Python utilities to render your model in JSON or XML or any other format you'd like.

Adding Endpoints with Custom `route()` Methods

Note: A much simpler way of adding more endpoints to pages is provided by the *wagtailroutablepage* module.

Wagtail routes requests by iterating over the path components (separated with a forward slash `/`), finding matching objects based on their slug, and delegating further routing to that object's model class. The Wagtail source is very instructive in figuring out what's happening. This is the default `route()` method of the `Page` class:

```
class Page(...):

    ...

    def route(self, request, path_components):
        if path_components:
            # request is for a child of this page
            child_slug = path_components[0]
            remaining_components = path_components[1:]

            # find a matching child or 404
            try:
                subpage = self.get_children().get(slug=child_slug)
            except Page.DoesNotExist:
                raise Http404

            # delegate further routing
            return subpage.specific.route(request, remaining_components)

        else:
            # request is for this very page
            if self.live:
                # Return a RouteResult that will tell Wagtail to call
                # this page's serve() method
                return RouteResult(self)
            else:
                # the page matches the request, but isn't published, so 404
                raise Http404
```

route() takes the current object (`self`), the `request` object, and a list of the remaining `path_components` from the request URL. It either continues delegating routing by calling *route()* again on one of its children in the Wagtail tree, or ends the routing process by returning a `RouteResult` object or raising a 404 error.

The `RouteResult` object (defined in wagtail.wagtailcore.url_routing) encapsulates all the information Wagtail needs to call a page's *serve()* method and return a final response: this information consists of the page object,

and any additional `args`/`kwargs` to be passed to *`serve()`*.

By overriding the *`route()`* method, we could create custom endpoints for each object in the Wagtail tree. One use case might be using an alternate template when encountering the `print/` endpoint in the path. Another might be a REST API which interacts with the current object. Just to see what's involved, lets make a simple model which prints out all of its child path components.

First, `models.py`:

```
from django.shortcuts import render
from wagtail.wagtailcore.url_routing import RouteResult
from django.http.response import Http404
from wagtail.wagtailadmin.edit_handlers import FieldPanel, MultiFieldPanel
from wagtail.wagtailcore.models import Page

...

class Echoer(Page):

    def route(self, request, path_components):
        if path_components:
            # tell Wagtail to call self.serve() with an additional 'path_components'␣
↪kwarg
            return RouteResult(self, kwargs={'path_components': path_components})
        else:
            if self.live:
                # tell Wagtail to call self.serve() with no further args
                return RouteResult(self)
            else:
                raise Http404

    def serve(self, path_components=[]):
        return render(request, self.template, {
            'page': page,
            'echo': ' '.join(path_components),
        })

Echoer.content_panels = [
    FieldPanel('title', classname="full title"),
]

Echoer.promote_panels = [
    MultiFieldPanel(Page.promote_panels, "Common page configuration"),
]
```

This model, `Echoer`, doesn't define any properties, but does subclass `Page` so objects will be able to have a custom title and slug. The template just has to display our `{{ echo }}` property.

Now, once creating a new `Echoer` page in the Wagtail admin titled "Echo Base," requests such as:

```
http://127.0.0.1:8000/echo-base/tauntaun/kennel/bed/and/breakfast/
```

Will return:

```
tauntaun kennel bed and breakfast
```

Be careful if you're introducing new required arguments to the `serve()` method - Wagtail still needs to be able to display a default view of the page for previewing and moderation, and by default will attempt to do this by calling `serve()` with a request object and no further arguments. If your `serve()` method does not accept that as a

method signature, you will need to override the page's `serve_preview()` method to call `serve()` with suitable arguments:

```
def serve_preview(self, request, mode_name):
    return self.serve(request, color='purple')
```

Tagging

Wagtail provides tagging capability through the combination of two django modules, `taggit` and `modelcluster`. `taggit` provides a model for tags which is extended by `modelcluster`, which in turn provides some magical database abstraction which makes drafts and revisions possible in Wagtail. It's a tricky recipe, but the net effect is a many-to-many relationship between your model and a tag class reserved for your model.

Using an example from the Wagtail demo site, here's what the tag model and the relationship field looks like in `models.py`:

```
from modelcluster.fields import ParentalKey
from modelcluster.contrib.taggit import ClusterTaggableManager
from taggit.models import TaggedItemBase

class BlogPageTag(TaggedItemBase):
    content_object = ParentalKey('demo.BlogPage', related_name='tagged_items')

class BlogPage(Page):
    ...
    tags = ClusterTaggableManager(through=BlogPageTag, blank=True)

BlogPage.promote_panels = [
    ...
    FieldPanel('tags'),
]
```

Wagtail's admin provides a nice interface for inputting tags into your content, with typeahead tag completion and friendly tag icons.

Now that we have the many-to-many tag relationship in place, we can fit in a way to render both sides of the relation. Here's more of the Wagtail demo site `models.py`, where the index model for `BlogPage` is extended with logic for filtering the index by tag:

```
class BlogIndexPage(Page):
    ...
    def serve(self, request):
        # Get blogs
        blogs = self.blogs

        # Filter by tag
        tag = request.GET.get('tag')
        if tag:
            blogs = blogs.filter(tags__name=tag)

        return render(request, self.template, {
            'page': page,
            'blogs': blogs,
        })
```

Here, `blogs.filter(tags__name=tag)` invokes a reverse Django queryset filter on the `BlogPageTag` model to optionally limit the `BlogPage` objects sent to the template for rendering. Now, lets render both sides of the relation

by showing the tags associated with an object and a way of showing all of the objects associated with each tag. This could be added to the `blog_page.html` template:

```
{% for tag in page.tags.all %}
    <a href="{% pageurl page.blog_index %}?tag={{ tag }}">{{ tag }}</a>
{% endfor %}
```

Iterating through `page.tags.all` will display each tag associated with `page`, while the link(s) back to the index make use of the filter option added to the `BlogIndexPage` model. A Django query could also use the `tagged_items` related name field to get `BlogPage` objects associated with a tag.

This is just one possible way of creating a taxonomy for Wagtail objects. With all of the components for a taxonomy available through Wagtail, you should be able to fulfill even the most exotic taxonomic schemes.

Available panel types

FieldPanel

class `wagtail.wagtailadmin.edit_handlers.`**`FieldPanel`**(*field_name*, *classname=None*, *widget=None*)

This is the panel used for basic Django field types.

`field_name`
This is the name of the class property used in your model definition.

`classname`
This is a string of optional CSS classes given to the panel which are used in formatting and scripted interactivity. By default, panels are formatted as inset fields.

The CSS class `full` can be used to format the panel so it covers the full width of the Wagtail page editor.

The CSS class `title` can be used to mark a field as the source for auto-generated slug strings.

`widget` (*optional*)
This parameter allows you to specify a Django form widget to use instead of the default widget for this field type.

MultiFieldPanel

class `wagtail.wagtailadmin.edit_handlers.`**`MultiFieldPanel`**(*children*, *heading=""*, *classname=None*)

This panel condenses several *`FieldPanel`* s or choosers, from a `list` or `tuple`, under a single `heading` string.

`children`
A `list` or `tuple` of child panels

`heading`
A heading for the fields

Collapsing MultiFieldPanels to save space

By default, `MultiFieldPanel` s are expanded and not collapsible. Adding `collapsible` to `classname` will enable the collapse control. Adding both `collapsible` and `collapsed` to the `classname` parameter will load the editor page with the `MultiFieldPanel` collapsed under its heading.

```
content_panels = [
    MultiFieldPanel(
        [
            ImageChooserPanel('cover'),
            DocumentChooserPanel('book_file'),
            PageChooserPanel('publisher'),
        ],
        heading="Collection of Book Fields",
        classname="collapsible collapsed"
    ),
]
```

InlinePanel

class wagtail.wagtailadmin.edit_handlers.**InlinePanel**(*relation_name*, *panels=None*, *classname=None*, *label=''*, *help_text=''*, *min_num=None*, *max_num=None*)

This panel allows for the creation of a "cluster" of related objects over a join to a separate model, such as a list of related links or slides to an image carousel.

This is a poweful but complex feature which will take some space to cover, so we'll skip over it for now. For a full explanation on the usage of `InlinePanel`, see *Inline Panels and Model Clusters*.

FieldRowPanel

class wagtail.wagtailadmin.edit_handlers.**FieldRowPanel**(*children*, *classname=None*)

This panel creates a columnar layout in the editing interface, where each of the child Panels appears alongside each other rather than below.

Use of FieldRowPanel particularly helps reduce the "snow-blindness" effect of seeing so many fields on the page, for complex models. It also improves the perceived association between fields of a similar nature. For example if you created a model representing an "Event" which had a starting date and ending date, it may be intuitive to find the start and end date on the same "row".

FieldRowPanel should be used in combination with `col*` class names added to each of the child Panels of the FieldRowPanel. The Wagtail editing interface is laid out using a grid system, in which the maximum width of the editor is 12 columns. Classes `col1`-`col12` can be applied to each child of a FieldRowPanel. The class `col3` will ensure that field appears 3 columns wide or a quarter the width. `col4` would cause the field to be 4 columns wide, or a third the width.

children
A `list` or `tuple` of child panels to display on the row

classname
A class to apply to the FieldRowPanel as a whole

PageChooserPanel

class wagtail.wagtailadmin.edit_handlers.**PageChooserPanel**(*field_name*, *page_type=None*, *can_choose_root=False*)

You can explicitly link *Page*-derived models together using the *Page* model and `PageChooserPanel`.

```
from wagtail.wagtailcore.models import Page
from wagtail.wagtailadmin.edit_handlers import PageChooserPanel


class BookPage(Page):
    publisher = models.ForeignKey(
        'wagtailcore.Page',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+',
    )

    content_panels = Page.content_panels + [
        PageChooserPanel('related_page', 'demo.PublisherPage'),
    ]
```

`PageChooserPanel` takes one required argument, the field name. Optionally, specifying a page type (in the form of an `"appname.modelname"` string) will filter the chooser to display only pages of that type. A list or tuple of page types can also be passed in, to allow choosing a page that matches any of those page types:

```
PageChooserPanel('related_page', ['demo.PublisherPage', 'demo.AuthorPage'])
```

Passing `can_choose_root=True` will allow the editor to choose the tree root as a page. Normally this would be undesirable, since the tree root is never a usable page, but in some specialised cases it may be appropriate; for example, a page with an automatic "related articles" feed could use a PageChooserPanel to select which subsection articles will be taken from, with the root corresponding to 'everywhere'.

ImageChooserPanel

class `wagtail.wagtailimages.edit_handlers.`**`ImageChooserPanel`**(*field_name*)

Wagtail includes a unified image library, which you can access in your models through the `Image` model and the `ImageChooserPanel` chooser. Here's how:

```
from wagtail.wagtailimages.models import Image
from wagtail.wagtailimages.edit_handlers import ImageChooserPanel


class BookPage(Page):
    cover = models.ForeignKey(
        'wagtailimages.Image',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+'
    )

    content_panels = Page.content_panels + [
        ImageChooserPanel('cover'),
    ]
```

Django's default behaviour is to "cascade" deletions through a ForeignKey relationship, which may not be what you want. This is why the `null`, `blank`, and `on_delete` parameters should be set to allow for an empty field. (See Django model field reference (on_delete)). `ImageChooserPanel` takes only one argument: the name of the field.

Displaying `Image` objects in a template requires the use of a template tag. See *Using images in templates*.

DocumentChooserPanel

class `wagtail.wagtaildocs.edit_handlers.`**`DocumentChooserPanel`**(*field_name*)

For files in other formats, Wagtail provides a generic file store through the `Document` model:

```
from wagtail.wagtaildocs.models import Document
from wagtail.wagtaildocs.edit_handlers import DocumentChooserPanel


class BookPage(Page):
    book_file = models.ForeignKey(
        'wagtaildocs.Document',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+'
    )

    content_panels = Page.content_panels + [
        DocumentChooserPanel('book_file'),
    ]
```

As with images, Wagtail documents should also have the appropriate extra parameters to prevent cascade deletions across a ForeignKey relationship. `DocumentChooserPanel` takes only one argument: the name of the field.

SnippetChooserPanel

Changed in version 1.1: Before Wagtail 1.1, it was necessary to pass the snippet model class as a second parameter to `SnippetChooserPanel`. This is now automatically picked up from the field.

class `wagtail.wagtailsnippets.edit_handlers.`**`SnippetChooserPanel`**(*field_name*, *snippet_type=None*)

Snippets are vanilla Django models you create yourself without a Wagtail-provided base class. A chooser, `SnippetChooserPanel`, is provided which takes the field name as an argument.

```
from wagtail.wagtailsnippets.edit_handlers import SnippetChooserPanel

class BookPage(Page):
    advert = models.ForeignKey(
        'demo.Advert',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+'
    )

    content_panels = Page.content_panels + [
        SnippetChooserPanel('advert'),
    ]
```

See *Snippets* for more information.

Built-in Fields and Choosers

Django's field types are automatically recognised and provided with an appropriate widget for input. Just define that field the normal Django way and pass the field name into *FieldPanel* when defining your panels. Wagtail will take care of the rest.

Here are some Wagtail-specific types that you might include as fields in your models.

Field Customisation

By adding CSS classes to your panel definitions or adding extra parameters to your field definitions, you can control much of how your fields will display in the Wagtail page editing interface. Wagtail's page editing interface takes much of its behaviour from Django's admin, so you may find many options for customisation covered there. (See Django model field reference).

Full-Width Input

Use `classname="full"` to make a field (input element) stretch the full width of the Wagtail page editor. This will not work if the field is encapsulated in a *MultiFieldPanel*, which places its child fields into a formset.

Titles

Use `classname="title"` to make Page's built-in title field stand out with more vertical padding.

Required Fields

To make input or chooser selection mandatory for a field, add `blank=False` to its model definition. (See Django model field reference (blank)).

Hiding Fields

Without a panel definition, a default form field (without label) will be used to represent your fields. If you intend to hide a field on the Wagtail page editor, define the field with `editable=False` (See Django model field reference (editable)).

Inline Panels and Model Clusters

The `django-modelcluster` module allows for streamlined relation of extra models to a Wagtail page. For instance, you can create objects related through a `ForeignKey` relationship on the fly and save them to a draft revision of a `Page` object. Normally, your related objects "cluster" would need to be created beforehand (or asynchronously) before linking them to a Page.

Let's look at the example of adding related links to a *Page*-derived model. We want to be able to add as many as we like, assign an order, and do all of this without leaving the page editing screen.

```
from wagtail.wagtailcore.models import Orderable, Page
from modelcluster.fields import ParentalKey

# The abstract model for related links, complete with panels
```

```
class RelatedLink(models.Model):
    title = models.CharField(max_length=255)
    link_external = models.URLField("External link", blank=True)

    panels = [
        FieldPanel('title'),
        FieldPanel('link_external'),
    ]

    class Meta:
        abstract = True

# The real model which combines the abstract model, an
# Orderable helper class, and what amounts to a ForeignKey link
# to the model we want to add related links to (BookPage)
class BookPageRelatedLinks(Orderable, RelatedLink):
    page = ParentalKey('demo.BookPage', related_name='related_links')

class BookPage(Page):
  # ...

  content_panels = Page.content_panels + [
    InlinePanel('related_links', label="Related Links"),
  ]
```

The `RelatedLink` class is a vanilla Django abstract model. The `BookPageRelatedLinks` model extends it with capability for being ordered in the Wagtail interface via the `Orderable` class as well as adding a `page` property which links the model to the `BookPage` model we're adding the related links objects to. Finally, in the panel definitions for `BookPage`, we'll add an *`InlinePanel`* to provide an interface for it all. Let's look again at the parameters that *`InlinePanel`* accepts:

```
InlinePanel( relation_name, panels=None, label='', help_text='', min_num=None, max_
↪num=None )
```

The `relation_name` is the `related_name` label given to the cluster's `ParentalKey` relation. You can add the `panels` manually or make them part of the cluster model. `label` and `help_text` provide a heading and caption, respectively, for the Wagtail editor. Finally, `min_num` and `max_num` allow you to set the minimum/maximum number of forms that the user must submit.

Changed in version 1.0: In previous versions, it was necessary to pass the base model as the first parameter to *`InlinePanel`*; this is no longer required.

For another example of using model clusters, see *Tagging*

For more on `django-modelcluster`, visit the django-modelcluster github project page.

Model Reference

This document contains reference information for the model classes inside the `wagtailcore` module.

Page

Database fields

class `wagtail.wagtailcore.models.`**Page**

title
(text)

Human-readable title of the page.

slug
(text)

This is used for constructing the page's URL.

For example: `http://domain.com/blog/[my-slug]/`

content_type
(foreign key to `django.contrib.contenttypes.models.ContentType`)

A foreign key to the `ContentType` object that represents the specific model of this page.

live
(boolean)

A boolean that is set to `True` if the page is published.

Note: this field defaults to `True` meaning that any pages that are created programmatically will be published by default.

has_unpublished_changes
(boolean)

A boolean that is set to `True` when the page is either in draft or published with draft changes.

owner
(foreign key to user model)

A foreign key to the user that created the page.

first_published_at
(date/time)

The date/time when the page was first published.

seo_title
(text)

Alternate SEO-crafted title, for use in the page's `<title>` HTML tag.

search_description
(text)

SEO-crafted description of the content, used for search indexing. This is also suitable for the page's `<meta name="description">` HTML tag.

show_in_menus
(boolean)

Toggles whether the page should be included in site-wide menus.

This is used by the *in_menu()* QuerySet filter.

Methods and properies

In addition to the model fields provided, `Page` has many properties and methods that you may wish to reference, use, or override in creating your own models. Those listed here are relatively straightforward to use, but consult the Wagtail source code for a full view of what's possible.

class `wagtail.wagtailcore.models.`**`Page`**

`specific`
Return this page in its most specific subclassed form.

`specific_class`
Return the class that this page would be if instantiated in its most specific form

`url`
Return the 'most appropriate' URL for referring to this page from the pages we serve, within the Wagtail backend and actual website templates; this is the local URL (starting with '/') if we're only running a single site (i.e. we know that whatever the current page is being served from, this link will be on the same domain), and the full URL (with domain) if not. Return None if the page is not routable.

`full_url`
Return the full URL (including protocol / domain) to this page, or None if it is not routable

`route`(*request*, *path_components*)

`serve`(*request*, **args*, ***kwargs*)

`get_context`(*request*, **args*, ***kwargs*)

`get_template`(*request*, **args*, ***kwargs*)

`preview_modes`
A list of (internal_name, display_name) tuples for the modes in which this page can be displayed for preview/moderation purposes. Ordinarily a page will only have one display mode, but subclasses of Page can override this - for example, a page containing a form might have a default view of the form, and a post-submission 'thankyou' page

`serve_preview`(*request*, *mode_name*)
Return an HTTP response for use in page previews. Normally this would be equivalent to self.serve(request), since we obviously want the preview to be indicative of how it looks on the live site. However, there are a couple of cases where this is not appropriate, and custom behaviour is required:

1) The page has custom routing logic that derives some additional required args/kwargs to be passed to serve(). The routing mechanism is bypassed when previewing, so there's no way to know what args we should pass. In such a case, the page model needs to implement its own version of serve_preview.

2) The page has several different renderings that we would like to be able to see when previewing - for example, a form page might have one rendering that displays the form, and another rendering to display a landing page when the form is posted. This can be done by setting a custom preview_modes list on the page model - Wagtail will allow the user to specify one of those modes when previewing, and pass the chosen mode_name to serve_preview so that the page model can decide how to render it appropriately. (Page models that do not specify their own preview_modes list will always receive an empty string as mode_name.)

Any templates rendered during this process should use the 'request' object passed here - this ensures that request.user and other properties are set appropriately for the wagtail user bar to be displayed. This request will always be a GET.

`get_ancestors`(*inclusive=False*)

`get_descendants`(*inclusive=False*)

get_siblings(*inclusive=True*)

search_fields
A list of fields to be indexed by the search engine. See Search docs *Indexing extra fields*

subpage_types
A whitelist of page models which can be created as children of this page type e.g a `BlogIndex` page might allow `BlogPage`, but not `JobPage` e.g

```
class BlogIndex(Page):
    subpage_types = ['mysite.BlogPage', 'mysite.BlogArchivePage']
```

The creation of child pages can be blocked altogether for a given page by setting it's subpage_types attribute to an empty array e.g

```
class BlogPage(Page):
    subpage_types = []
```

parent_page_types
A whitelist of page models which are allowed as parent page types e.g a `BlogPage` may only allow itself to be created below the `BlogIndex` page e.g

```
class BlogPage(Page):
    parent_page_types = ['mysite.BlogIndexPage']
```

Pages can block themselves from being created at all by setting parent_page_types to an empty array (this is useful for creating unique pages that should only be created once) e.g

```
class HiddenPage(Page):
    parent_page_types = []
```

password_required_template
Defines which template file should be used to render the login form for Protected pages using this model. This overrides the default, defined using `PASSWORD_REQUIRED_TEMPLATE` in your settings. See *Private pages*

is_creatable
Controls if this page can be created through the Wagtail administration. Defaults to True, and is not inherited by subclasses. This is useful when using multi-table inheritance, to stop the base model from being created as an actual page.

Site

The `Site` model is useful for multi-site installations as it allows an administrator to configure which part of the tree to use for each hostname that the server responds on.

This configuration is used by the `SiteMiddleware` middleware class which checks each request against this configuration and appends the Site object to the Django request object.

Database fields

class `wagtail.wagtailcore.models.`**Site**

hostname
(text)

This is the hostname of the site, excluding the scheme, port and path.

For example: `www.mysite.com`

Note: If you're looking for how to get the root url of a site, use the *root_url* attribute.

port
(number)

This is the port number that the site responds on.

root_page
(foreign key to *Page*)

This is a link to the root page of the site. This page will be what appears at the / URL on the site and would usually be a homepage.

is_default_site
(boolean)

This is set to `True` if the site is the default. Only one site can be the default.

The default site is used as a fallback in situations where a site with the required hostname/port couldn't be found.

Methods and properties

class `wagtail.wagtailcore.models.`**Site**

static find_for_request(*request*)
Find the site object responsible for responding to this HTTP request object. Try:

- unique hostname first
- then hostname and port
- if there is no matching hostname at all, or no matching hostname:port combination, fall back to the unique default site, or raise an exception

NB this means that high-numbered ports on an extant hostname may still be routed to a different hostname which is set as the default

root_url
This returns the URL of the site. It is calculated from the *hostname* and the *port* fields.

The scheme part of the URL is calculated based on value of the *port* field:

- 80 = `http://`
- 443 = `https://`
- Everything else will use the `http://` scheme and the port will be appended to the end of the hostname (eg. `http://mysite.com:8000/`)

static get_site_root_paths()
Return a list of (root_path, root_url) tuples, most specific path first - used to translate url_paths into actual URLs with hostnames

PageRevision

Every time a page is edited a new `PageRevision` is created and saved to the database. It can be used to find the full history of all changes that have been made to a page and it also provides a place for new changes to be kept before going live.

- Revisions can be created from any *`Page`* object by calling its `save_revision()` method
- The content of the page is JSON-serialised and stored in the *`content_json`* field
- You can retrieve a `PageRevision` as a *`Page`* object by calling the *`as_page_object()`* method

Database fields

class `wagtail.wagtailcore.models.`**`PageRevision`**

`page`
(foreign key to *`Page`*)

`submitted_for_moderation`
(boolean)

`True` if this revision is in moderation

`created_at`
(date/time)

This is the time the revision was created

`user`
(foreign key to user model)

This links to the user that created the revision

`content_json`
(text)

This field contains the JSON content for the page at the time the revision was created

Managers

class `wagtail.wagtailcore.models.`**`PageRevision`**

`objects`
This manager is used to retrieve all of the `PageRevision` objects in the database

Example:

```
PageRevision.objects.all()
```

`submitted_revisions`
This manager is used to retrieve all of the `PageRevision` objects that are awaiting moderator approval

Example:

```
PageRevision.submitted_revisions.all()
```

Methods and properties

class wagtail.wagtailcore.models.**PageRevision**

as_page_object()
This method retrieves this revision as an instance of its *Page* subclass.

approve_moderation()
Calling this on a revision that's in moderation will mark it as approved and publish it

reject_moderation()
Calling this on a revision that's in moderation will mark it as rejected

is_latest_revision()
Returns `True` if this revision is its page's latest revision

publish()
Calling this will copy the content of this revision into the live page object. If the page is in draft, it will be published.

GroupPagePermission

Database fields

class wagtail.wagtailcore.models.**GroupPagePermission**

group
(foreign key to `django.contrib.auth.models.Group`)

page
(foreign key to *Page*)

permission_type
(choice list)

PageViewRestriction

Database fields

class wagtail.wagtailcore.models.**PageViewRestriction**

page
(foreign key to *Page*)

password
(text)

Orderable (abstract)

Database fields

class wagtail.wagtailcore.models.**Orderable**

sort_order
(number)

Page QuerySet reference

All models that inherit from *Page* are given some extra QuerySet methods accessible from their `.objects` attribute.

Examples

- Selecting only live pages

```
live_pages = Page.objects.live()
```

- Selecting published EventPages that are descendants of events_index

```
events = EventPage.objects.live().descendant_of(events_index)
```

- Getting a list of menu items

```
# This gets a QuerySet of live children of the homepage with ``show_in_
↪menus`` set
menu_items = homepage.get_children().live().in_menu()
```

Reference

class wagtail.wagtailcore.query.**PageQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

live()
This filters the QuerySet to only contain published pages.

Example:

```
published_pages = Page.objects.live()
```

not_live()
This filters the QuerySet to only contain unpublished pages.

Example:

```
unpublished_pages = Page.objects.not_live()
```

in_menu()
This filters the QuerySet to only contain pages that are in the menus.

Example:

```
# Build a menu from live pages that are children of the homepage
menu_items = homepage.get_children().live().in_menu()
```

Note: To put your page in menus, set the show_in_menus flag to true:

```
# Add 'my_page' to the menu
my_page.show_in_menus = True
```

page (*other*)
This filters the QuerySet so it only contains the specified page.

Example:

```
# Append an extra page to a QuerySet
new_queryset = old_queryset | Page.objects.page(page_to_add)
```

not_page (*other*)
This filters the QuerySet so it doesn't contain the specified page.

Example:

```
# Remove a page from a QuerySet
new_queryset = old_queryset & Page.objects.not_page(page_to_remove)
```

descendant_of (*other*, *inclusive=False*)
This filters the QuerySet to only contain pages that descend from the specified page.

If inclusive is set to True, it will also contain the page itself (instead of just its descendants).

Example:

```
# Get EventPages that are under the special_events Page
special_events = EventPage.objects.descendant_of(special_events_index)

# Alternative way
special_events = special_events_index.get_descendants()
```

not_descendant_of (*other*, *inclusive=False*)
This filters the QuerySet to not contain any pages that descend from the specified page.

If inclusive is set to True, it will also exclude the specified page.

Example:

```
# Get EventPages that are not under the archived_events Page
non_archived_events = EventPage.objects.not_descendant_of(archived_events_
↪index)
```

child_of (*other*)
This filters the QuerySet to only contain pages that are direct children of the specified page.

Example:

```
# Get a list of sections
sections = Page.objects.child_of(homepage)

# Alternative way
sections = homepage.get_children()
```

ancestor_of (*other*, *inclusive=False*)
This filters the QuerySet to only contain pages that are ancestors of the specified page.

If inclusive is set to True, it will also include the specified page.

Example:

```
# Get the current section
current_section = Page.objects.ancestor_of(current_page).child_of(homepage).
↪first()

# Alternative way
current_section = current_page.get_ancestors().child_of(homepage).first()
```

not_ancestor_of (*other*, *inclusive=False*)
This filters the QuerySet to not contain any pages that are ancestors of the specified page.

If inclusive is set to True, it will also exclude the specified page.

Example:

```
# Get the other sections
other_sections = Page.objects.not_ancestor_of(current_page).child_of(homepage)
```

sibling_of (*other*, *inclusive=True*)
This filters the QuerySet to only contain pages that are siblings of the specified page.

By default, inclusive is set to True so it will include the specified page in the results.

If inclusive is set to False, the page will be excluded from the results.

Example:

```
# Get list of siblings
siblings = Page.objects.sibling_of(current_page)

# Alternative way
siblings = current_page.get_siblings()
```

public ()
This filters the QuerySet to only contain pages that are not in a private section

See: *Private pages*

Note: This doesn't filter out unpublished pages. If you want to only have published public pages, use `.live().public()`

Example:

```
# Find all the pages that are viewable by the public
all_pages = Page.objects.live().public()
```

search (*query_string*, *fields=None*, *operator=None*, *order_by_relevance=True*, *backend='default'*)
This runs a search query on all the items in the QuerySet

See: *Searching QuerySets*

Example:

```
# Search future events
results = EventPage.objects.live().filter(date__gt=timezone.now()).search(
↪"Hello")
```

type(*model*)
This filters the QuerySet to only contain pages that are an instance of the specified model (including subclasses).

Example:

```
# Find all pages that are of type AbstractEmailForm, or a descendant of it
form_pages = Page.objects.type(AbstractEmailForm)
```

unpublish()
This unpublishes all pages in the QuerySet

Example:

```
# Unpublish current_page and all of its children
Page.objects.descendant_of(current_page, inclusive=True).unpublish()
```

specific()
This efficiently gets all the specific pages for the queryset, using the minimum number of queries.

Example:

```
# Get the specific instance of all children of the hompage,
# in a minimum number of database queries.
homepage.get_children().specific()
```

See also: *Page.specific*

Contrib modules

Wagtail ships with a variety of extra optional modules.

Site settings

You can define settings for your site that are editable by administrators in the Wagtail admin. These settings can be accessed in code, as well as in templates.

To use these settings, you must add `wagtail.contrib.settings` to your `INSTALLED_APPS`:

```
INSTALLED_APPS += [
    'wagtail.contrib.settings',
]
```

Defining settings

Create a model that inherits from `BaseSetting`, and register it using the `register_setting` decorator:

```
from wagtail.contrib.settings.models import BaseSetting, register_setting

@register_setting
class SocialMediaSettings(BaseSetting):
    facebook = models.URLField(
        help_text='Your Facebook page URL')
    instagram = models.CharField(
        max_length=255, help_text='Your Instagram username, without the @')
```

```
    trip_advisor = models.URLField(
        help_text='Your Trip Advisor page URL')
    youtube = models.URLField(
        help_text='Your YouTube channel or user account URL')
```

A 'Social media settings' link will appear in the Wagtail admin 'Settings' menu.

Edit handlers

Settings use edit handlers much like the rest of Wagtail. Add a `panels` setting to your model defining all the edit handlers required:

```
@register_setting
class ImportantPages(BaseSetting):
    donate_page = models.ForeignKey(
        'wagtailcore.Page', null=True, on_delete=models.SET_NULL)
    sign_up_page = models.ForeignKey(
        'wagtailcore.Page', null=True, on_delete=models.SET_NULL)

    panels = [
        PageChooserPanel('donate_page'),
        PageChooserPanel('sign_up_page'),
    ]
```

Appearance

You can change the label used in the menu by changing the verbose_name of your model.

You can add an icon to the menu by passing an 'icon' argument to the `register_setting` decorator:

```
@register_setting(icon='icon-placeholder')
class SocialMediaSettings(BaseSetting):
    class Meta:
        verbose_name = 'Social media accounts'
    ...
```

For a list of all available icons, please see the *UI Styleguide*.

Using the settings

Settings are designed to be used both in Python code, and in templates.

Using in Python

If access to a setting is required in the code, the `for_site()` method will retrieve the setting for the supplied site:

```
def view(request):
    social_media_settings = SocialMediaSettings.for_site(request.site)
    ...
```

Using in templates

Add the `request` and `settings` context processors to your settings:

```
from django.conf import global_settings
TEMPLATE_CONTEXT_PROCESSORS = global_settings.TEMPLATE_CONTEXT_PROCESSORS + [
    'django.core.context_processors.request',
    'wagtail.contrib.settings.context_processors.settings',
]
```

Then access the settings through `{{ settings }}`:

```
{{ settings.app_label.SocialMediaSettings.instagram }}
```

If you are not in a `RequestContext`, then context processors will not have run, and the `settings` variable will not be availble. To get the `settings`, use the provided `{% get_settings %}` template tag. If a `request` is in the template context, but for some reason it is not a `RequestContext`, just use `{% get_settings %}`:

```
{% load wagtailsettings_tags %}
{% get_settings %}
{{ settings.app_label.SocialMediaSettings.instagram }}
```

If there is no `request` available in the template at all, you can use the settings for the default Wagtail site instead:

```
{% load wagtailsettings_tags %}
{% get_settings use_default_site=True %}
{{ settings.app_label.SocialMediaSettings.instagram }}
```

Note: You can not reliably get the correct settings instance for the current site from this template tag if the request object is not available. This is only relevant for multisite instances of Wagtail.

Form builder

The `wagtailforms` module allows you to set up single-page forms, such as a 'Contact us' form, as pages of a Wagtail site. It provides a set of base models that site implementers can extend to create their own `FormPage` type with their own site-specific templates. Once a page type has been set up in this way, editors can build forms within the usual page editor, consisting of any number of fields. Form submissions are stored for later retrieval through a new 'Forms' section within the Wagtail admin interface; in addition, they can be optionally e-mailed to an address specified by the editor.

Usage

Add `wagtail.wagtailforms` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
   ...
   'wagtail.wagtailforms',
]
```

Within the `models.py` of one of your apps, create a model that extends `wagtailforms.models.AbstractEmailForm`:

```
from modelcluster.fields import ParentalKey
from wagtail.wagtailadmin.edit_handlers import (FieldPanel, InlinePanel,
    MultiFieldPanel)
from wagtail.wagtailcore.fields import RichTextField
from wagtail.wagtailforms.models import AbstractEmailForm, AbstractFormField

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', related_name='form_fields')

class FormPage(AbstractEmailForm):
    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

FormPage.content_panels = [
    FieldPanel('title', classname="full title"),
    FieldPanel('intro', classname="full"),
    InlinePanel('form_fields', label="Form fields"),
    FieldPanel('thank_you_text', classname="full"),
    MultiFieldPanel([
        FieldPanel('to_address', classname="full"),
        FieldPanel('from_address', classname="full"),
        FieldPanel('subject', classname="full"),
    ], "Email")
]
```

`AbstractEmailForm` defines the fields `to_address`, `from_address` and `subject`, and expects `form_fields` to be defined. Any additional fields are treated as ordinary page content - note that `FormPage` is responsible for serving both the form page itself and the landing page after submission, so the model definition should include all necessary content fields for both of those views.

If you do not want your form page type to offer form-to-email functionality, you can inherit from AbstractForm instead of `AbstractEmailForm`, and omit the `to_address`, `from_address` and `subject` fields from the `content_panels` definition.

You now need to create two templates named `form_page.html` and `form_page_landing.html` (where `form_page` is the underscore-formatted version of the class name). `form_page.html` differs from a standard Wagtail template in that it is passed a variable `form`, containing a Django `Form` object, in addition to the usual `page` variable. A very basic template for the form would thus be:

```
{% load wagtailcore_tags %}
<html>
    <head>
        <title>{{ page.title }}</title>
    </head>
    <body>
        <h1>{{ page.title }}</h1>
        {{ page.intro|richtext }}
        <form action="{% pageurl page %}" method="POST">
            {% csrf_token %}
            {{ form.as_p }}
            <input type="submit">
        </form>
    </body>
</html>
```

`form_page_landing.html` is a regular Wagtail template, displayed after the user makes a successful form submission.

Static site generator

This document describes how to render your Wagtail site into static HTML files on your local file system, Amazon S3 or Google App Engine, using django medusa and the `wagtail.contrib.wagtailmedusa` module.

Note: An alternative module based on the django-bakery package is available as a third-party contribution: https://github.com/mhnbcu/wagtailbakery

---

Installing `django-medusa`

First, install `django-medusa` and `django-sendfile` from pip:

```
pip install django-medusa django-sendfile
```

Then add `django_medusa` and `wagtail.contrib.wagtailmedusa` to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'django_medusa',
    'wagtail.contrib.wagtailmedusa',
]
```

Define `MEDUSA_RENDERER_CLASS`, `MEDUSA_DEPLOY_DIR` and `SENDFILE_BACKEND` in settings:

```
MEDUSA_RENDERER_CLASS = 'django_medusa.renderers.DiskStaticSiteRenderer'
MEDUSA_DEPLOY_DIR = os.path.join(BASE_DIR, 'build')
SENDFILE_BACKEND = 'sendfile.backends.simple'
```

Rendering

To render a site, run `./manage.py staticsitegen`. This will render the entire website and place the HTML in a folder called `medusa_output`. The static and media folders need to be copied into this folder manually after the rendering is complete. This feature inherits `django-medusa`'s ability to render your static site to Amazon S3 or Google App Engine; see the medusa docs for configuration details.

To test, open the `medusa_output` folder in a terminal and run `python -m SimpleHTTPServer`.

Advanced topics

GET parameters

Pages which require GET parameters (e.g. for pagination) don't generate a suitable file name for the generated HTML files.

Wagtail provides a mixin (`wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin`) which allows you to embed a Django URL configuration into a page. This allows you to give the subpages a URL like `/page/1/` which work well with static site generation.

Example:

```
from wagtail.contrib.wagtailroutablepage.models import RoutablePageMixin, route


class BlogIndex(Page, RoutablePageMixin):
    ...

    @route(r'^$', name='main')
    @route(r'^page/(?P<page>\d+)/$', name='page')
    def serve_page(self, request, page=1):
        ...
```

Then in the template, you can use the `{% routablepageurl %}` tag to link between the pages:

```
{% load wagtailroutablepage_tags %}

{% if results.has_previous %}
    <a href="{% routablepageurl page 'page' results.previous_page_number %}">Next page
↪</a>
{% else %}

{% if results.has_next %}
    <a href="{% routablepageurl page 'page' results.next_page_number %}">Next page</a>
{% else %}
```

Next, you have to tell the `wagtailmedusa` module about your custom routing...

Rendering pages which use custom routing

For page types that override the `route` method, we need to let `django-medusa` know which URLs it responds on. This is done by overriding the `get_static_site_paths` method to make it yield one string per URL path.

For example, the BlogIndex above would need to yield one URL for each page of results:

```
def get_static_site_paths(self):
    # Get page count
    page_count = ...

    # Yield a path for each page
    for page in range(page_count):
        yield '/%d/' % (page + 1)

    # Yield from superclass
    for path in super(BlogIndex, self).get_static_site_paths():
        yield path
```

Sitemap generator

This document describes how to create XML sitemaps for your Wagtail website using the `wagtail.contrib.wagtailsitemaps` module.

Basic configuration

You firstly need to add `"wagtail.contrib.wagtailsitemaps"` to INSTALLED_APPS in your Django settings file:

```
INSTALLED_APPS = [
    ...

    "wagtail.contrib.wagtailsitemaps",
]
```

Then, in `urls.py`, you need to add a link to the `wagtail.contrib.wagtailsitemaps.views.sitemap` view which generates the sitemap:

```
from wagtail.contrib.wagtailsitemaps.views import sitemap

urlpatterns = [
    ...

    url('^sitemap\.xml$', sitemap),
]
```

You should now be able to browse to `/sitemap.xml` and see the sitemap working. By default, all published pages in your website will be added to the site map.

Setting the hostname

By default, the sitemap uses the hostname defined in the Wagtail Admin's `Sites` area. If your default site is called `localhost`, then URLs in the sitemap will look like:

```
<url>
    <loc>http://localhost/about/</loc>
    <lastmod>2015-09-26</lastmod>
</url>
```

For tools like Google Search Tools to properly index your site, you need to set a valid, crawlable hostname. If you change the site's hostname from `localhost` to `mysite.com`, `sitemap.xml` will contain the correct URLs:

```
<url>
    <loc>http://mysite.com/about/</loc>
    <lastmod>2015-09-26</lastmod>
</url>
```

Find out more about *working with Sites </reference/pages/model_reference.html?highlight=site#site>*.

Customising

URLs

The `Page` class defines a `get_sitemap_urls` method which you can override to customise sitemaps per `Page` instance. This method must return a list of dictionaries, one dictionary per URL entry in the sitemap. You can exclude pages from the sitemap by returning an empty list.

Each dictionary can contain the following:

- **location** (required) - This is the full URL path to add into the sitemap.
- **lastmod** - A python date or datetime set to when the page was last modified.
- **changefreq**

- **priority**

You can add more but you will need to override the `wagtailsitemaps/sitemap.xml` template in order for them to be displayed in the sitemap.

Cache

By default, sitemaps are cached for 100 minutes. You can change this by setting `WAGTAILSITEMAPS_CACHE_TIMEOUT` in your Django settings to the number of seconds you would like the cache to last for.

Frontend cache invalidator

Changed in version 0.7: Multiple backend support added Cloudflare support added

Many websites use a frontend cache such as Varnish, Squid or Cloudflare to gain extra performance. The downside of using a frontend cache though is that they don't respond well to updating content and will often keep an old version of a page cached after it has been updated.

This document describes how to configure Wagtail to purge old versions of pages from a frontend cache whenever a page gets updated.

Setting it up

Firstly, add `"wagtail.contrib.wagtailfrontendcache"` to your INSTALLED_APPS:

- ```
  INSTALLED_APPS = [
      ...

      "wagtail.contrib.wagtailfrontendcache"
  ]
  ```

Changed in version 0.8: Signal handlers are now automatically registered in Django 1.7 and upwards

The `wagtailfrontendcache` module provides a set of signal handlers which will automatically purge the cache whenever a page is published or deleted.

If you are using Django version 1.7 or newer, these signal handlers are automatically registered when the `wagtail.contrib.wagtailfrontendcache` app is loaded. Otherwise, they must be registered as your application starts up. This can be done by placing the following code in your `urls.py`:

```
# urls.py
from wagtail.contrib.wagtailfrontendcache.signal_handlers import register_signal_
↪handlers
register_signal_handlers()
```

Varnish/Squid

Add a new item into the `WAGTAILFRONTENDCACHE` setting and set the `BACKEND` parameter to `wagtail.contrib.wagtailfrontendcache.backends.HTTPBackend`. This backend requires an extra parameter `LOCATION` which points to where the cache is running (this must be a direct connection to the server and cannot go through another proxy).
```

```
# settings.py

WAGTAILFRONTENDCACHE = {
    'varnish': {
        'BACKEND': 'wagtail.contrib.wagtailfrontendcache.backends.HTTPBackend',
        'LOCATION': 'http://localhost:8000',
    },
}
```

Finally, make sure you have configured your frontend cache to accept PURGE requests:

- Varnish
- Squid

Cloudflare

Firstly, you need to register an account with Cloudflare if you haven't already got one. You can do this here: Cloudflare Sign up

Add an item into the `WAGTAILFRONTENDCACHE` and set the `BACKEND` parameter to `wagtail.contrib.wagtailfrontendcache.backends.CloudflareBackend`. This backend requires two extra parameters, `EMAIL` (your Cloudflare account email) and `TOKEN` (your API token from Cloudflare).

```
# settings.py

WAGTAILFRONTENDCACHE = {
    'cloudflare': {
        'BACKEND': 'wagtail.contrib.wagtailfrontendcache.backends.CloudflareBackend',
        'EMAIL': 'your-cloudflare-email-address@example.com',
        'TOKEN': 'your cloudflare api token',
    },
}
```

Advanced usage

Invalidating more than one URL per page

By default, Wagtail will only purge one URL per page. If your page has more than one URL to be purged, you will need to override the `get_cached_paths` method on your page type.

```
class BlogIndexPage(Page):
    def get_blog_items(self):
        # This returns a Django paginator of blog items in this section
        return Paginator(self.get_children().live().type(BlogPage), 10)

    def get_cached_paths(self):
        # Yield the main URL
        yield '/'

        # Yield one URL per page in the paginator to make sure all pages are purged
        for page_number in range(1, self.get_blog_items().num_pages):
            yield '/?page=' + str(page_number)
```

Invalidating index pages

Another problem is pages that list other pages (such as a blog index) will not be purged when a blog entry gets added, changed or deleted. You may want to purge the blog index page so the updates are added into the listing quickly.

This can be solved by using the `purge_page_from_cache` utility function which can be found in the `wagtail.contrib.wagtailfrontendcache.utils` module.

Let's take the the above BlogIndexPage as an example. We need to register a signal handler to run when one of the BlogPages get updated/deleted. This signal handler should call the `purge_page_from_cache` function on all BlogIndexPages that contain the BlogPage being updated/deleted.

```
# models.py
from django.dispatch import receiver
from django.db.models.signals import pre_delete

from wagtail.wagtailcore.signals import page_published
from wagtail.contrib.wagtailfrontendcache.utils import purge_page_from_cache


...


def blog_page_changed(blog_page):
    # Find all the live BlogIndexPages that contain this blog_page
    for blog_index in BlogIndexPage.objects.live():
        if blog_page in blog_index.get_blog_items().object_list:
            # Purge this blog index
            purge_page_from_cache(blog_index)


@receiver(page_published, sender=BlogPage):
def blog_published_handler(instance):
    blog_page_changed(instance)


@receiver(pre_delete, sender=BlogPage)
def blog_deleted_handler(instance):
    blog_page_changed(instance)
```

Invalidating individual URLs

`wagtail.contrib.wagtailfrontendcache.utils` provides another function called `purge_url_from_cache`. As the name suggests, this purges an individual URL from the cache.

For example, this could be useful for purging a single page of blogs:

```
from wagtail.contrib.wagtailfrontendcache.utils import purge_url_from_cache

# Purge the first page of the blog index
purge_url_from_cache(blog_index.url + '?page=1')
```

RoutablePageMixin

The `RoutablePageMixin` mixin provides a convenient way for a page to respond on multiple sub-URLs with different views. For example, a blog section on a site might provide several different types of index page at URLs like `/`

`blog/2013/06/`, `/blog/authors/bob/`, `/blog/tagged/python/`, all served by the same page instance.

A `Page` using `RoutablePageMixin` exists within the page tree like any other page, but URL paths underneath it are checked against a list of patterns. If none of the patterns match, control is passed to subpages as usual (or failing that, a 404 error is thrown).

The basics

To use `RoutablePageMixin`, you need to make your class inherit from both *wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin* and *wagtail.wagtailcore.models.Page*, then define some view methods and decorate them with `wagtail.contrib.wagtailroutablepage.models.route`.

Here's an example of an `EventPage` with three views:

```
from wagtail.wagtailcore.models import Page
from wagtail.contrib.wagtailroutablepage.models import RoutablePageMixin, route


class EventPage(RoutablePageMixin, Page):
    ...

    @route(r'^$')
    def current_events(self, request):
        """
        View function for the current events page
        """
        ...

    @route(r'^past/$')
    def past_events(self, request):
        """
        View function for the past events page
        """
        ...

    # Multiple routes!
    @route(r'^year/(\d+)/$')
    @route(r'^year/current/$')
    def events_for_year(self, request, year=None):
        """
        View function for the events for year page
        """
        ...
```

Reversing URLs

RoutablePageMixin adds a *reverse_subpage()* method to your page model which you can use for reversing URLs. For example:

```
# The URL name defaults to the view method name.
>>> event_page.reverse_subpage('events_for_year', args=(2015, ))
'year/2015/'
```

This method only returns the part of the URL within the page. To get the full URL, you must append it to the values of either the *url* or the *full_url* attribute on your page:

```
>>> event_page.url + event_page.reverse_subpage('events_for_year', args=(2015, ))
'/events/year/2015/'

>>> event_page.full_url + event_page.reverse_subpage('events_for_year', args=(2015, ))
'http://example.com/events/year/2015/'
```

Changing route names

The route name defaults to the name of the view. You can override this name with the `name` keyword argument on `@route`:

```
from wagtail.wagtailcore.models import Page
from wagtail.contrib.wagtailroutablepage.models import RoutablePageMixin, route


class EventPage(RoutablePageMixin, Page):
    ...

    @route(r'^year/(\d+)/$', name='year')
    def events_for_year(self, request, year):
        """
        View function for the events for year page
        """
        ...
```

```
>>> event_page.reverse_subpage('year', args=(2015, ))
'/events/year/2015/'
```

The `RoutablePageMixin` class

class `wagtail.contrib.wagtailroutablepage.models.`**`RoutablePageMixin`**

This class can be mixed in to a Page model, allowing extra routes to be added to it.

classmethod `get_subpage_urls`()

`resolve_subpage`(*path*)

This method takes a URL path and finds the view to call.

Example:

```
view, args, kwargs = page.resolve_subpage('/past/')
response = view(request, *args, **kwargs)
```

`reverse_subpage`(*name*, *args=None*, *kwargs=None*)

This method takes a route name/arguments and returns a URL path.

Example:

```
url = page.url + page.reverse_subpage('events_for_year', kwargs={'year
↪': '2014'})
```

The `routablepageurl` template tag

`wagtail.contrib.wagtailroutablepage.templatetags.wagtailroutablepage_tags.routablepageurl(`

`routablepageurl` is similar to `pageurl`, but works with `RoutablePages`. It behaves like a hybrid between the built-in `reverse`, and `pageurl` from Wagtail.

`page` is the RoutablePage that URLs will be generated from.

`url_name` is a URL name defined in `page.subpage_urls`.

Positional arguments and keyword arguments should be passed as normal positional arguments and keyword arguments.

Example:

```
{% load wagtailroutablepage_tags %}

{% routablepageurl page "feed" %}
{% routablepageurl page "archive" 2014 08 14 %}
{% routablepageurl page "food" foo="bar" baz="quux" %}
```

Wagtail API

The `wagtailapi` module can be used to create a read-only, JSON-based API for public Wagtail content.

There are three endpoints to the API:

- **Pages:** `/api/v1/pages/`
- **Images:** `/api/v1/images/`
- **Documents:** `/api/v1/documents/`

See *Wagtail API Installation* and *Wagtail API Configuration* if you're looking to add this module to your Wagtail site.

See *Wagtail API Usage Guide* for documentation on the API.

Index

Wagtail API Installation

To install, add `wagtail.contrib.wagtailapi` and `rest_framework` to `INSTALLED_APPS` in your Django settings and configure a URL for it in `urls.py`:

```
# settings.py

INSTALLED_APPS = [
    ...
    'wagtail.contrib.wagtailapi',
    'rest_framework',
]

# urls.py
```

```
from wagtail.contrib.wagtailapi import urls as wagtailapi_urls

urlpatterns = [
    ...
    url(r'^api/', include(wagtailapi_urls)),
]
```

Wagtail API Configuration

Settings

`WAGTAILAPI_BASE_URL` (required when using frontend cache invalidation)

This is used in two places, when generating absolute URLs to document files and invalidating the cache.

Generating URLs to documents will fall back the the current request's hostname if this is not set. Cache invalidation cannot do this, however, so this setting must be set when using this module alongside the `wagtailfrontendcache` module.

`WAGTAILAPI_SEARCH_ENABLED` (default: True)

Setting this to false will disable full text search. This applies to all endpoints.

`WAGTAILAPI_MAX_RESULTS` (default: 20)

This allows you to change the maximum number of results a user can get at any time. This applies to all endpoints.

Adding more fields to the pages endpoint

By default, the pages endpoint only includes the `id`, `title` and `type` fields in both the listing and detail views.

You can add more fields to the pages endpoint by setting an attribute called `api_fields` to a `list` or `tuple` of field names:

```
class BlogPage(Page):
    posted_by = models.CharField()
    posted_at = models.DateTimeField()
    content = RichTextField()

    api_fields = ('posted_by', 'posted_at', 'content')
```

This list also supports child relations (which will be nested inside the returned JSON document):

```
class BlogPageRelatedLink(Orderable):
    page = ParentalKey('BlogPage', related_name='related_links')
    link = models.URLField()

    api_fields = ('link', )

class BlogPage(Page):
    posted_by = models.CharField()
    posted_at = models.DateTimeField()
    content = RichTextField()

    api_fields = ('posted_by', 'posted_at', 'content', 'related_links')
```

Frontend cache invalidation

If you have a Varnish, Squid or Cloudflare instance in front of your API, the `wagtailapi` module can automatically invalidate cached responses for you whenever they are updated in the database.

To enable it, firstly configure the `wagtail.contrib.wagtailfrontendcache` module within your project (see [Wagtail frontend cache docs](http://docs.wagtail.io/en/latest/contrib_components/frontendcache.html) for more information).

Then make sure that the `WAGTAILAPI_BASE_URL` setting is set correctly (Example: `WAGTAILAPI_BASE_URL = 'http://api.mysite.com'`).

Then finally, switch it on by setting `WAGTAILAPI_USE_FRONTENDCACHE` to `True`.

Wagtail API Usage Guide

Listing views

Performing a `GET` request against one of the endpoints will get you a listing of objects in that endpoint. The response will look something like this:

```
GET /api/v1/endpoint_name/

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": "total number of results"
    },
    "endpoint_name": [
        {
            "id": 1,
            "meta": {
                "type": "app_name.ModelName",
                "detail_url": "http://api.example.com/api/v1/endpoint_name/1/"
            },
            "field": "value"
        },
        {
            "id": 2,
            "meta": {
                "type": "app_name.ModelName",
                "detail_url": "http://api.example.com/api/v1/endpoint_name/2/"
            },
            "field": "different value"
        }
    ]
}
```

This is the basic structure of all of the listing views. They all have a `meta` section with a `total_count` variable and a listing of things.

Detail views

All of the endpoints also contain a "detail" view which returns information on an individual object. This view is always accessed by appending the id of the object to the URL.

The `pages` endpoint

This endpoint includes all live pages in your site that have not been put in a private section.

The listing view (`/api/v1/pages/`)

This is what a typical response from a `GET` request to this listing would look like:

```
GET /api/v1/pages/

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 2
    },
    "pages": [
        {
            "id": 2,
            "meta": {
                "type": "demo.HomePage",
                "detail_url": "http://api.example.com/api/v1/pages/2/"
            },
            "title": "Homepage"
        },
        {
            "id": 3,
            "meta": {
                "type": "demo.BlogIndexPage",
                "detail_url": "http://api.example.com/api/v1/pages/3/"
            },
            "title": "Blog"
        }
    ]
}
```

Each page object contains the `id`, a `meta` section and the fields with their values.

`meta`

This section is used to hold "metadata" fields which aren't fields in the database. Wagtail API adds two by default:

- `type` - The app label/model name of the object
- `detail_url` - A URL linking to the detail view for this object

Selecting a page type

Most Wagtail sites are made up of multiple different types of page that each have their own specific fields. In order to view/filter/order on fields specific to one page type, you must select that page type using the `type` query parameter.

The `type` query parameter must be set to the Pages model name in the format: `app_label.ModelName`.

```
GET /api/v1/pages/?type=demo.BlogPage

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 3
    },
    "pages": [
        {
            "id": 4,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/4/"
            },
            "title": "My blog 1"
        },
        {
            "id": 5,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/5/"
            },
            "title": "My blog 2"
        },
        {
            "id": 6,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/6/"
            },
            "title": "My blog 3"
        }
    ]
}
```

Specifying a list of fields to return

As you can see, we still only get the `title` field, even though we have selected a type. That's because listing pages require you to explicitly tell it what extra fields you would like to see. You can do this with the `fields` query parameter.

Just set `fields` to a command-separated list of field names that you would like to use.

```
GET /api/v1/pages/?type=demo.BlogPage&fields=title,date_posted,feed_image

HTTP 200 OK
Content-Type: application/json
```

```
{
    "meta": {
        "total_count": 3
    },
    "pages": [
        {
            "id": 4,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/4/"
            },
            "title": "My blog 1",
            "date_posted": "2015-01-23",
            "feed_image": {
                "id": 1,
                "meta": {
                    "type": "wagtailimages.Image",
                    "detail_url": "http://api.example.com/api/v1/images/1/"
                }
            }
        },
        {
            "id": 5,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/5/"
            },
            "title": "My blog 2",
            "date_posted": "2015-01-24",
            "feed_image": {
                "id": 2,
                "meta": {
                    "type": "wagtailimages.Image",
                    "detail_url": "http://api.example.com/api/v1/images/2/"
                }
            }
        },
        {
            "id": 6,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/6/"
            },
            "title": "My blog 3",
            "date_posted": "2015-01-25",
            "feed_image": {
                "id": 3,
                "meta": {
                    "type": "wagtailimages.Image",
                    "detail_url": "http://api.example.com/api/v1/images/3/"
                }
            }
        }
    ]
}
```


We now have enough information to make a basic blog listing with a feed image and date that the blog was posted.

Filtering on fields

Exact matches on field values can be done by using a query parameter with the same name as the field. Any pages with the field that exactly matches the value of this parameter will be returned.

```
GET /api/v1/pages/?type=demo.BlogPage&fields=title,date_posted&date_posted=2015-01-24

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 1
    },
    "pages": [

        {
            "id": 5,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/5/"
            },
            "title": "My blog 2",
            "date_posted": "2015-01-24",
        }
    ]
}
```

Filtering by section of the tree

It is also possible to filter the listing to only include pages with a particular parent or ancestor. This is useful if you have multiple blogs on your site and only want to view the contents of one of them.

child_of

Filters the listing to only include direct children of the specified page.

For example, to get all the pages that are direct children of page 7.

```
GET /api/v1/pages/?child_of=7

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 1
    },
    "pages": [
        {
            "id": 4,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/4/"
            },
            "title": "Other blog 1"
        }
```

```
    ]
}
```

descendant_of

New in version 1.1.

Filters the listing to only include descendants of the specified page.

For example, to get all pages underneath the homepage:

```
GET /api/v1/pages/?descendant_of=2

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 1
    },
    "pages": [
        {
            "id": 3,
            "meta": {
                "type": "demo.BlogIndexPage",
                "detail_url": "http://api.example.com/api/v1/pages/3/"
            },
            "title": "Blog"
        },
        {
            "id": 4,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/4/"
            },
            "title": "My blog 1",
        },
        {
            "id": 5,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/5/"
            },
            "title": "My blog 2",
        },
        {
            "id": 6,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/6/"
            },
            "title": "My blog 3",
        }
    ]
}
```

Ordering

Like filtering, it is also possible to order on database fields. The endpoint accepts a query parameter called `order` which should be set to the field name to order by. Field names can be prefixed with a `-` to reverse the ordering. It is also possible to order randomly by setting this parameter to `random`.

```
GET /api/v1/pages/?type=demo.BlogPage&fields=title,date_posted,feed_image&order=-date_
↪posted

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 3
    },
    "pages": [
        {
            "id": 6,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/6/"
            },
            "title": "My blog 3",
            "date_posted": "2015-01-25",
            "feed_image": {
                "id": 3,
                "meta": {
                    "type": "wagtailimages.Image",
                    "detail_url": "http://api.example.com/api/v1/images/3/"
                }
            }
        },
        {
            "id": 5,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/5/"
            },
            "title": "My blog 2",
            "date_posted": "2015-01-24",
            "feed_image": {
                "id": 2,
                "meta": {
                    "type": "wagtailimages.Image",
                    "detail_url": "http://api.example.com/api/v1/images/2/"
                }
            }
        },
        {
            "id": 4,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/4/"
            },
            "title": "My blog 1",
            "date_posted": "2015-01-23",
            "feed_image": {
```

```
                "id": 1,
                "meta": {
                    "type": "wagtailimages.Image",
                    "detail_url": "http://api.example.com/api/v1/images/1/"
                }
            }
        }
    ]
}
```

Pagination

Pagination is done using two query parameters called `limit` and `offset`. `limit` sets the number of results to return and `offset` is the index of the first result to return. The default value for `limit` is `20` and its maximum value is `100` (which can be changed using the `WAGTAILAPI_MAX_RESULTS` setting).

```
GET /api/v1/pages/?limit=1&offset=1

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 2
    },
    "pages": [
        {
            "id": 3,
            "meta": {
                "type": "demo.BlogIndexPage",
                "detail_url": "http://api.example.com/api/v1/pages/3/"
            },
            "title": "Blog"
        }
    ]
}
```

Pagination will not change the `total_count` value in the meta.

Searching

To perform a full-text search, set the `search` parameter to the query string you would like to search on.

```
GET /api/v1/pages/?search=Blog

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 3
    },
    "pages": [
        {
```

```
            "id": 3,
            "meta": {
                "type": "demo.BlogIndexPage",
                "detail_url": "http://api.example.com/api/v1/pages/3/"
            },
            "title": "Blog"
        },
        {
            "id": 4,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/4/"
            },
            "title": "My blog 1",
        },
        {
            "id": 5,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/5/"
            },
            "title": "My blog 2",
        },
        {
            "id": 6,
            "meta": {
                "type": "demo.BlogPage",
                "detail_url": "http://api.example.com/api/v1/pages/6/"
            },
            "title": "My blog 3",
        }
    ]
}
```

The results are ordered by relevance. It is not possible to use the `order` parameter with a search query.

If your Wagtail site is using Elasticsearch, you do not need to select a type to access specific fields. This will search anything that's defined in the models' `search_fields`.

The detail view (`/api/v1/pages/{id}/`)

This view gives you access to all of the details for a particular page.

```
GET /api/v1/pages/6/

HTTP 200 OK
Content-Type: application/json

{
    "id": 6,
    "meta": {
        "type": "demo.BlogPage",
        "detail_url": "http://api.example.com/api/v1/pages/6/"
    },
    "parent": {
        "id": 3,
        "meta": {
```

```
            "type": "demo.BlogIndexPage",
            "detail_url": "http://api.example.com/api/v1/pages/3/"
        }
    },
    "title": "My blog 3",
    "date_posted": "2015-01-25",
    "feed_image": {
        "id": 3,
        "meta": {
            "type": "wagtailimages.Image",
            "detail_url": "http://api.example.com/api/v1/images/3/"
        }
    },
    "related_links": [
        {
            "title": "Other blog page",
            "page": {
                "id": 5,
                "meta": {
                    "type": "demo.BlogPage",
                    "detail_url": "http://api.example.com/api/v1/pages/5/"
                }
            }
        }
    ]
}
```

The format is the same as that which is returned inside the listing view, with two additions:

- All of the available fields are added to the detail page by default
- A `parent` field has been included that contains information about the parent page

The `images` endpoint

This endpoint gives access to all uploaded images. This will use the custom image model if one was specified. Otherwise, it falls back to `wagtailimages.Image`.

The listing view (`/api/v1/images/`)

This is what a typical response from a `GET` request to this listing would look like:

```
GET /api/v1/images/

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 3
    },
    "images": [
        {
            "id": 4,
            "meta": {
                "type": "wagtailimages.Image",
```

```
                "detail_url": "http://api.example.com/api/v1/images/4/"
            },
            "title": "Wagtail by Mark Harkin"
        },
        {
            "id": 5,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/5/"
            },
            "title": "James Joyce"
        },
        {
            "id": 6,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/6/"
            },
            "title": "David Mitchell"
        }
    ]
}
```

Each image object contains the `id` and `title` of the image.

Getting `width`, `height` and other fields

Like the pages endpoint, the images endpoint supports the `fields` query parameter.

By default, this will allow you to add the `width` and `height` fields to your results. If your Wagtail site uses a custom image model, it is possible to have more.

```
GET /api/v1/images/?fields=title,width,height

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 3
    },
    "images": [
        {
            "id": 4,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/4/"
            },
            "title": "Wagtail by Mark Harkin",
            "width": 640,
            "height": 427
        },
        {
            "id": 5,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/5/"
```

```
            },
            "title": "James Joyce",
            "width": 500,
            "height": 392
        },
        {
            "id": 6,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/6/"
            },
            "title": "David Mitchell",
            "width": 360,
            "height": 282
        }
    ]
}
```

Filtering on fields

Exact matches on field values can be done by using a query parameter with the same name as the field. Any images with the field that exactly matches the value of this parameter will be returned.

```
GET /api/v1/pages/?title=James Joyce

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 3
    },
    "images": [
        {
            "id": 5,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/5/"
            },
            "title": "James Joyce"
        }
    ]
}
```

Ordering

The images endpoint also accepts the `order` parameter which should be set to a field name to order by. Field names can be prefixed with a `-` to reverse the ordering. It is also possible to order randomly by setting this parameter to `random`.

```
GET /api/v1/images/?fields=title,width&order=width

HTTP 200 OK
Content-Type: application/json
```

```
{
    "meta": {
        "total_count": 3
    },
    "images": [
        {
            "id": 6,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/6/"
            },
            "title": "David Mitchell",
            "width": 360
        },
        {
            "id": 5,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/5/"
            },
            "title": "James Joyce",
            "width": 500
        },
        {
            "id": 4,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/4/"
            },
            "title": "Wagtail by Mark Harkin",
            "width": 640
        }
    ]
}
```

Pagination

Pagination is done using two query parameters called `limit` and `offset`. `limit` sets the number of results to return and `offset` is the index of the first result to return. The default value for `limit` is `20` and its maximum value is `100` (which can be changed using the `WAGTAILAPI_MAX_RESULTS` setting).

```
GET /api/v1/images/?limit=1&offset=1

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 3
    },
    "images": [
        {
            "id": 5,
            "meta": {
                "type": "wagtailimages.Image",
```

```
                "detail_url": "http://api.example.com/api/v1/images/5/"
            },
            "title": "James Joyce",
            "width": 500,
            "height": 392
        }
    ]
}
```

Pagination will not change the `total_count` value in the meta.

Searching

To perform a full-text search, set the `search` parameter to the query string you would like to search on.

```
GET /api/v1/images/?search=James

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 1
    },
    "pages": [
        {
            "id": 5,
            "meta": {
                "type": "wagtailimages.Image",
                "detail_url": "http://api.example.com/api/v1/images/5/"
            },
            "title": "James Joyce",
            "width": 500,
            "height": 392
        }
    ]
}
```

Like the pages endpoint, the results are ordered by relevance and it is not possible to use the `order` parameter with a search query.

The detail view (`/api/v1/images/{id}/`)

This view gives you access to all of the details for a particular image.

```
GET /api/v1/images/5/

HTTP 200 OK
Content-Type: application/json

{
    "id": 5,
    "meta": {
        "type": "wagtailimages.Image",
        "detail_url": "http://api.example.com/api/v1/images/5/"
```

```
    },
    "title": "James Joyce",
    "width": 500,
    "height": 392
}
```

The `documents` endpoint

This endpoint gives access to all uploaded documents.

The listing view (`/api/v1/documents/`)

The documents listing supports the same features as the images listing (documented above) but works with Documents instead.

The detail view (`/api/v1/documents/{id}/`)

This view gives you access to all of the details for a particular document.

```
GET /api/v1/documents/1/

HTTP 200 OK
Content-Type: application/json

{
    "id": 1,
    "meta": {
        "type": "wagtaildocs.Document",
        "detail_url": "http://api.example.com/api/v1/documents/1/",
        "download_url": "http://api.example.com/documents/1/usage.md"
    },
    "title": "Wagtail API usage"
}
```

Promoted search results

Changed in version 1.1: Before Wagtail 1.1, promoted search results were implemented in the `wagtail.wagtailsearch` core module and called "editors picks".

The `searchpromotions` module provides the models and user interface for managing "Promoted search results" and displaying them in a search results page.

"Promoted search results" allow editors to explicitly link relevant content to search terms, so results pages can contain curated content in addition to results from the search engine.

Installation

The `searchpromotions` module is not enabled by default. To install it, add `wagtail.contrib.wagtailsearchpromotions` to `INSTALLED_APPS` in your project's Django settings file.

```
INSTALLED_APPS = [
    ...

    'wagtail.contrib.wagtailsearchpromotions',
]
```

This app contains migrations so make sure you run the `migrate` django-admin command after installing.

Usage

Once installed, a new menu item called "Promoted search results" should appear in the "Settings" menu. This is where you can assign pages to popular search terms.

Displaying on a search results page

To retrieve a list of promoted search results for a particular search query, you can use the `{% get_search_promotions %}` template tag from the `wagtailsearchpromotions_tags` templatetag library:

```
{% load wagtailcore_tags wagtailsearchpromotions_tags %}

...

{% get_search_promotions search_query as search_promotions %}

<ul>
    {% for search_promotion in search_promotions %}
        <li>
            <a href="{% pageurl search_promotion.page %}">
                <h2>{{ search_promotion.page.title }}</h2>
                <p>{{ search_promotion.description }}</p>
            </a>
        </li>
    {% endfor %}
</ul>
```

Site settings

Site-wide settings that are editable by administrators in the Wagtail admin.

Form builder

Allows forms to be created by admins and provides an interface for browsing form submissions.

Static site generator

Provides a management command that turns a Wagtail site into a set of static HTML files.

Sitemap generator

Provides a view that generates a Google XML sitemap of your public wagtail content.

Frontend cache invalidator

A module for automatically purging pages from a cache (Varnish, Squid or Cloudflare) when their content is changed.

RoutablePageMixin

Provides a way of embedding Django URLconfs into pages.

Wagtail API

A module for adding a read only, JSON based web API to your Wagtail site

Promoted search results

A module for managing "Promoted Search Results"

Management commands

publish_scheduled_pages

```
./manage.py publish_scheduled_pages
```

This command publishes or unpublishes pages that have had these actions scheduled by an editor. It is recommended to run this command once an hour.

fixtree

```
./manage.py fixtree
```

This command scans for errors in your database and attempts to fix any issues it finds.

move_pages

```
manage.py move_pages from to
```

This command moves a selection of pages from one section of the tree to another.

Options:

- **from** This is the **id** of the page to move pages from. All descendants of this page will be moved to the destination. After the operation is complete, this page will have no children.
- **to** This is the **id** of the page to move pages to.

update_index

`./manage.py update_index [--backend <backend name>]`

This command rebuilds the search index from scratch. It is only required when using Elasticsearch.

It is recommended to run this command once a week and at the following times:

- whenever any pages have been created through a script (after an import, for example)
- whenever any changes have been made to models or search configuration

The search may not return any results while this command is running, so avoid running it at peak times.

Specifying which backend to update

New in version 0.7.

By default, `update_index` will rebuild all the search indexes listed in `WAGTAILSEARCH_BACKENDS`.

If you have multiple backends and would only like to update one of them, you can use the `--backend` option.

For example, to update just the default backend:

```
python manage.py update_index --backend default
```

search_garbage_collect

`./manage.py search_garbage_collect`

Wagtail keeps a log of search queries that are popular on your website. On high traffic websites, this log may get big and you may want to clean out old search queries. This command cleans out all search query logs that are more than one week old.

Hooks

On loading, Wagtail will search for any app with the file `wagtail_hooks.py` and execute the contents. This provides a way to register your own functions to execute at certain points in Wagtail's execution, such as when a `Page` object is saved or when the main menu is constructed.

Registering functions with a Wagtail hook is done through the `@hooks.register` decorator:

```
from wagtail.wagtailcore import hooks

@hooks.register('name_of_hook')
def my_hook_function(arg1, arg2...)
    # your code here
```

Alternatively, `hooks.register` can be called as an ordinary function, passing in the name of the hook and a handler function defined elsewhere:

```
hooks.register('name_of_hook', my_hook_function)
```

The available hooks are: `before_serve_page`

Called when Wagtail is about to serve a page. The callable passed into the hook will receive the page object, the request object, and the `args` and `kwargs` that will be passed to the page's `serve()` method. If the callable returns an `HttpResponse`, that response will be returned immediately to the user, and Wagtail will not proceed to call `serve()` on the page.

```
from wagtail.wagtailcore import hooks

@hooks.register('before_serve_page')
def block_googlebot(page, request, serve_args, serve_kwargs):
    if request.META.get('HTTP_USER_AGENT') == 'GoogleBot':
        return HttpResponse("<h1>bad googlebot no cookie</h1>")
```

Changed in version 1.0: The hook was renamed from `construct_wagtail_edit_bird`

`construct_wagtail_userbar` Add or remove items from the wagtail userbar. Add, edit, and moderation tools are provided by default. The callable passed into the hook must take the `request` object and a list of menu objects, `items`. The menu item objects must have a `render` method which can take a `request` object and return the HTML string representing the menu item. See the userbar templates and menu item classes for more information.

```
from wagtail.wagtailcore import hooks

class UserbarPuppyLinkItem(object):
  def render(self, request):
    return '<li><a href="http://cuteoverload.com/tag/puppehs/" ' \
    + 'target="_parent" class="action icon icon-wagtail">Puppies!</a></li>'

@hooks.register('construct_wagtail_userbar')
def add_puppy_link_item(request, items):
  return items.append( UserbarPuppyLinkItem() )
```

`construct_homepage_panels` Add or remove panels from the Wagtail admin homepage. The callable passed into this hook should take a `request` object and a list of `panels`, objects which have a `render()` method returning a string. The objects also have an `order` property, an integer used for ordering the panels. The default panels use integers between `100` and `300`.

```
from django.utils.safestring import mark_safe

from wagtail.wagtailcore import hooks

class WelcomePanel(object):
  order = 50

  def render(self):
    return mark_safe("""
    <section class="panel summary nice-padding">
      <h3>No, but seriously -- welcome to the admin homepage.</h3>
    </section>
    """)

@hooks.register('construct_homepage_panels')
def add_another_welcome_panel(request, panels):
  return panels.append( WelcomePanel() )
```

`construct_homepage_summary_items` New in version 1.0.

Add or remove items from the 'site summary' bar on the admin homepage (which shows the number of pages and other object that exist on the site). The callable passed into this hook should take a `request` object and

a list of `SummaryItem` objects to be modified as required. These objects have a `render()` method, which returns an HTML string, and an `order` property, which is an integer that specifies the order in which the items will appear.

after_create_page Do something with a `Page` object after it has been saved to the database (as a published page or a revision). The callable passed to this hook should take a `request` object and a `page` object. The function does not have to return anything, but if an object with a `status_code` property is returned, Wagtail will use it as a response object. By default, Wagtail will instead redirect to the Explorer page for the new page's parent.

```
from django.http import HttpResponse

from wagtail.wagtailcore import hooks

@hooks.register('after_create_page')
def do_after_page_create(request, page):
  return HttpResponse("Congrats on making content!", content_type="text/plain")
```

after_edit_page Do something with a `Page` object after it has been updated. Uses the same behavior as `after_create_page`.

after_delete_page Do something after a `Page` object is deleted. Uses the same behavior as `after_create_page`.

register_admin_urls Register additional admin page URLs. The callable fed into this hook should return a list of Django URL patterns which define the structure of the pages and endpoints of your extension to the Wagtail admin. For more about vanilla Django URLconfs and views, see url dispatcher.

```
from django.http import HttpResponse
from django.conf.urls import url

from wagtail.wagtailcore import hooks

def admin_view( request ):
  return HttpResponse( \
    "I have approximate knowledge of many things!", \
    content_type="text/plain")

@hooks.register('register_admin_urls')
def urlconf_time():
  return [
    url(r'^how_did_you_almost_know_my_name/$', admin_view, name='frank' ),
  ]
```

`register_admin_menu_item`

Add an item to the Wagtail admin menu. The callable passed to this hook must return an instance of `wagtail.wagtailadmin.menu.MenuItem`. New items can be constructed from the `MenuItem` class by passing in a `label` which will be the text in the menu item, and the URL of the admin page you want the menu item to link to (usually by calling `reverse()` on the admin view you've set up). Additionally, the following keyword arguments are accepted:

name an internal name used to identify the menu item; defaults to the slugified form of the label.

classnames additional classnames applied to the link, used to give it an icon

attrs additional HTML attributes to apply to the link

order an integer which determines the item's position in the menu

`MenuItem` can be subclassed to customise the HTML output, specify Javascript files required by the menu item, or conditionally show or hide the item for specific requests (for example, to apply permission checks); see the source code (`wagtail/wagtailadmin/menu.py`) for details.

```
from django.core.urlresolvers import reverse

from wagtail.wagtailcore import hooks
from wagtail.wagtailadmin.menu import MenuItem

@hooks.register('register_admin_menu_item')
def register_frank_menu_item():
  return MenuItem('Frank', reverse('frank'), classnames='icon icon-folder-
↪inverse', order=10000)
```

`register_settings_menu_item` New in version 0.7.

As `register_admin_menu_item`, but registers menu items into the 'Settings' sub-menu rather than the top-level menu.

`construct_main_menu` Called just before the Wagtail admin menu is output, to allow the list of menu items to be modified. The callable passed to this hook will receive a `request` object and a list of `menu_items`, and should modify `menu_items` in-place as required. Adding menu items should generally be done through the `register_admin_menu_item` hook instead - items added through `construct_main_menu` will be missing any associated Javascript includes, and their `is_shown` check will not be applied.

```
from wagtail.wagtailcore import hooks

@hooks.register('construct_main_menu')
def hide_explorer_menu_item_from_frank(request, menu_items):
  if request.user.username == 'frank':
    menu_items[:] = [item for item in menu_items if item.name != 'explorer']
```

`insert_editor_js` Add additional Javascript files or code snippets to the page editor. Output must be compatible with `compress`, as local static includes or string.

```
from django.utils.html import format_html, format_html_join
from django.conf import settings

from wagtail.wagtailcore import hooks

@hooks.register('insert_editor_js')
def editor_js():
  js_files = [
    'demo/js/hallo-plugins/hallo-demo-plugin.js',
  ]
  js_includes = format_html_join('\n', '<script src="{0}{1}"></script>',
    ((settings.STATIC_URL, filename) for filename in js_files)
  )
  return js_includes + format_html(
    """
    <script>
      registerHalloPlugin('demoeditor');
    </script>
    """
  )
```

`insert_editor_css` Add additional CSS or SCSS files or snippets to the page editor. Output must be compatible with `compress`, as local static includes or string.

```
from django.utils.html import format_html
from django.conf import settings

from wagtail.wagtailcore import hooks

@hooks.register('insert_editor_css')
def editor_css():
  return format_html('<link rel="stylesheet" href="' \
  + settings.STATIC_URL \
  + 'demo/css/vendor/font-awesome/css/font-awesome.min.css">')
```

`construct_whitelister_element_rules`

Customise the rules that define which HTML elements are allowed in rich text areas. By default only a limited set of HTML elements and attributes are whitelisted - all others are stripped out. The callables passed into this hook must return a dict, which maps element names to handler functions that will perform some kind of manipulation of the element. These handler functions receive the element as a BeautifulSoup Tag object.

The `wagtail.wagtailcore.whitelist` module provides a few helper functions to assist in defining these handlers: `allow_without_attributes`, a handler which preserves the element but strips out all of its attributes, and `attribute_rule` which accepts a dict specifying how to handle each attribute, and returns a handler function. This dict will map attribute names to either True (indicating that the attribute should be kept), False (indicating that it should be dropped), or a callable (which takes the initial attribute value and returns either a final value for the attribute, or None to drop the attribute).

For example, the following hook function will add the `<blockquote>` element to the whitelist, and allow the `target` attribute on `<a>` elements:

```
from wagtail.wagtailcore import hooks
from wagtail.wagtailcore.whitelist import attribute_rule, check_url, allow_
↪without_attributes

@hooks.register('construct_whitelister_element_rules')
def whitelister_element_rules():
    return {
        'blockquote': allow_without_attributes,
        'a': attribute_rule({'href': check_url, 'target': True}),
    }
```

`register_permissions` New in version 0.7.

Return a queryset of Permission objects to be shown in the Groups administration area.

Signals

Wagtail's PageRevision and Page implement Signals from `django.dispatch`. Signals are useful for creating side-effects from page publish/unpublish events.

Primarily, these are used by the Frontend Cache contrib module and the Wagtail API. You could also use signals to send publish notifications to a messaging service, or `POST` messages to another app that's consuming the API, such as a static site generator.

page_published

This signal is emitted from a `PageRevision` when a revision is set to *published*.

sender The page `class`

instance The specific `Page` instance.

revision The `PageRevision` that was published

kwargs Any other arguments passed to `page_published.send()`.

To listen to a signal, implement `page_published.connect(receiver, sender, **kwargs)`. Here's a simple example showing how you might notify your team when something is published:

```
from wagtailcore.signals import page_published
import urllib
import urllib2


# Let everyone know when a new page is published
def send_to_slack(sender, **kwargs)
    instance = kwargs['instance']
    url = 'https://hooks.slack.com/services/T00000000/B00000000/
↪XXXXXXXXXXXXXXXXXXXXXXXX'
    values = {
        "text" : "%s was published by %s " % (instance.title, instance.owner.
↪username),
        "channel": "#publish-notifications",
        "username": "the squid of content",
        "icon_emoji": ":octopus:"
    }

    data = urllib.urlencode(values)
    req = urllib2.Request(url, data)
    response = urllib2.urlopen(req)

# Register a receiver
page_published.connect(send_to_slack)
```

Receiving specific model events

Sometimes you're not interested in receiving signals for every model, or you want to handle signals for specific models in different ways. For instance, you may wish to do something when a new blog post is published:

```
from wagtailcore.signals import page_published
from mysite.models import BlogPostPage

# Do something clever for each model type
def receiver(model, **kwargs):
    # Do something with blog posts
    pass

# Register listeners for each page model class
page_published.connect(receiver, sender=BlogPostPage)
```

Wagtail provides access to a list of registered page types at `wagtail.wagtailcore.models.PAGE_MODEL_CLASSES`.

Read the Django documentation for more information about specifying senders.

page_unpublished

This signal is emitted from a `Page` when the page is unpublished.

sender The page `class`

instance The specific `Page` instance.

kwargs Any other arguments passed to `page_unpublished.send()`

The project template

```
mysite/
    home/
        migrations/
            __init__.py
            0001_initial.py
            0002_create_homepage.py
        templates/
            home/
                home_page.html
        __init__.py
        models.py
    search/
        templates/
            search/
                search.html
        __init__.py
        views.py
    mysite/
        settings/
            __init__.py
            base.py
            dev.py
            production.py
        static/
            css/
                mysite.css
            js/
                mysite.js
        templates/
            404.html
            500.html
            base.html
        __init__.py
        urls.py
        wsgi.py
    manage.py
    requirements.txt
```

The “home” app

Location: `/mysite/home/`

This app is here to help get you started quicker by providing a `HomePage` model with migrations to create one when you first setup your app.

Default templates and static files

Location: `/mysite/mysite/templates/` and `/mysite/mysite/static/`

The templates directory contains `base.html`, `404.html` and `500.html`. These files are very commonly needed on Wagtail sites to they have been added into the template.

The static directory contains an empty JavaScript and CSS file.

Django settings

Location: `/mysite/mysite/settings/`

The Django settings files are split up into `base.py`, `dev.py`, `production.py` and `local.py`.

`base.py` This file is for global settings that will be used in both development and production. Aim to keep most of your configuration in this file.

`dev.py` This file is for settings that will only be used by developers. For example: `DEBUG = True`

`production.py` This file is for settings that will only run on a production server. For example: `DEBUG = False`

`local.py` This file is used for settings local to a particular machine. This file should never be tracked by a version control system.

Tip: On production servers, we recommend that you only store secrets in `local.py` (such as API keys and passwords). This can save you headaches in the future if you are ever trying to debug why a server is behaving badly. If you are using multiple servers which need different settings then we recommend that you create a different `production.py` file for each one.

Support

Mailing list

If you have general questions about Wagtail, or you're looking for help on how to do something that these documents don't cover, join the mailing list at groups.google.com/d/forum/wagtail.

Issues

If you think you've found a bug in Wagtail, or you'd like to suggest a new feature, please check the current list at github.com/torchbox/wagtail/issues. If your bug or suggestion isn't there, raise a new issue, providing as much relevant context as possible.

Torchbox

Finally, if you have a query which isn't relevant for either of the above forums, feel free to contact the Wagtail team at Torchbox directly, on hello@wagtail.io or @wagtailcms.

Using Wagtail: an Editor's guide

This section of the documentation is written for the users of a Wagtail-powered site. That is, the content editors, moderators and administrators who will be running things on a day-to-day basis.

Introduction

Wagtail is a new open source content management system (CMS) developed by Torchbox. It is built on the Django framework and designed to be super easy to use for both developers and editors.

This documentation will explain how to:

- navigate the main user interface of Wagtail
- create pages of all different types
- modify, save, publish and unpublish pages
- how to set up users, and provide them with specific roles to create a publishing workflow
- upload, edit and include images and documents
- ... and more!

Getting started

The Wagtail demo site

This examples in this document are based on Torchbox.com. However, the instructions are general enough as to be applicable to any Wagtail site.

For the purposes of this documentation we will be using the URL, **www.example.com**, to represent the root (home-page) of your website.

Logging in

- The first port of call for an editor is the login page for the administrator interface.
- Access this by adding **/admin** onto the end of your root URL (e.g. `www.example.com/admin`).
- Enter your username and password and click **Sign in**.



Finding your way around

This section describes the different pages that you will see as you navigate around the CMS, and how you can find the content that you are looking for.

The Dashboard

The Dashboard provides information on:

- The number of pages, images, and documents currently held in the Wagtail CMS
- Any pages currently awaiting moderation (if you have these privileges)
- Your most recently edited pages

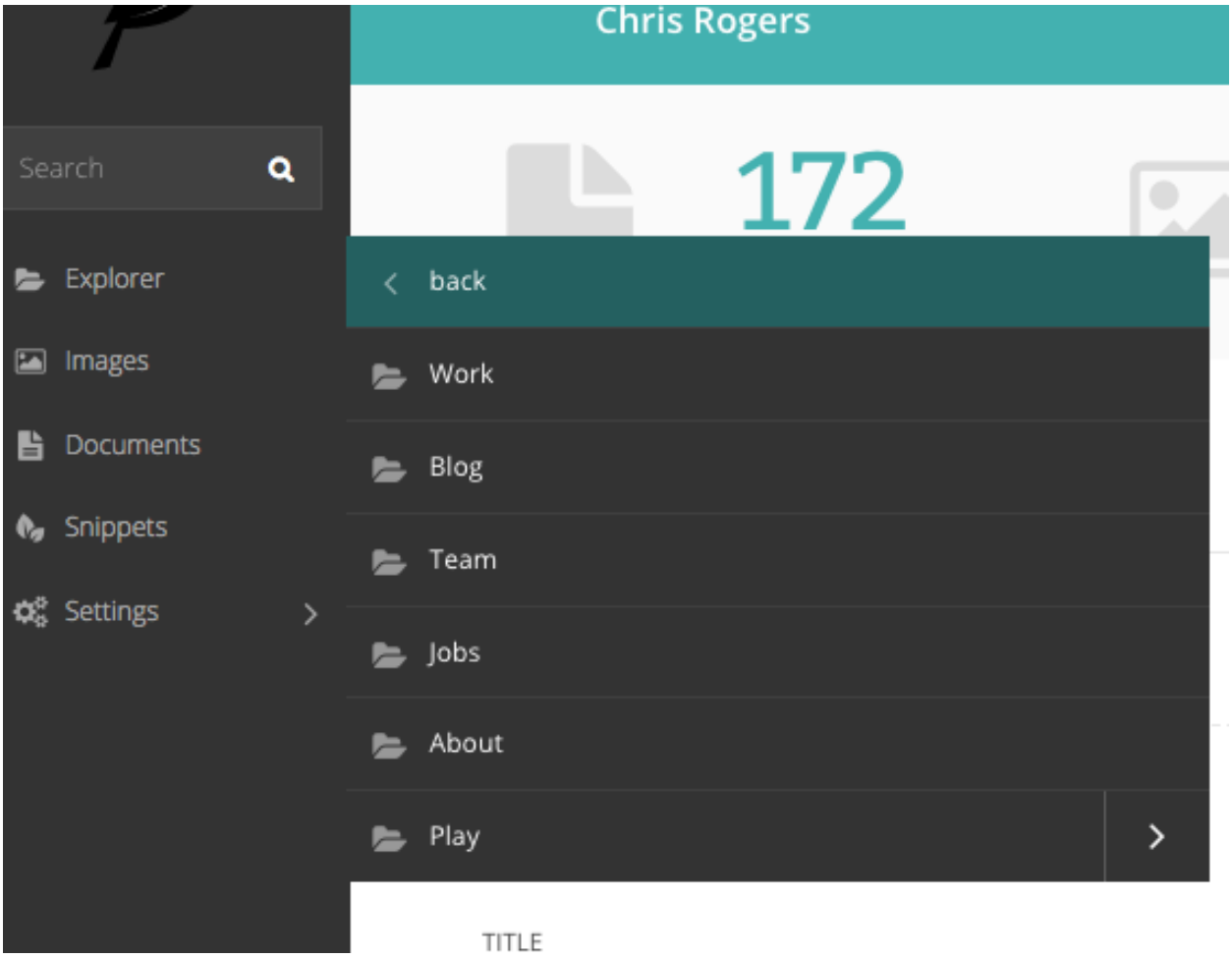
You can return to the Dashboard at any time by clicking the Wagtail log in the top-left of the screen.



- Clicking the logo returns you to your Dashboard.
- The stats at the top of the page describe the total amount of content on the CMS (just for fun!).
- The *Pages awaiting moderation* table will only be displayed if you have moderator or administrator privileges
  - Clicking the name of a page will take you to the 'Edit page' interface for this page.
  - Clicking approve or reject will either change the page status to live or return the page to draft status. An email will be sent to the creator of the page giving the result of moderation either way.
  - The *Parent* column tells you what the parent page of the page awaiting moderation is called. Clicking the parent page name will take you to its Edit page.
- The *Your most recent edits* table displays the five pages that you most recently edited.
- The date column displays the date that you edited the page. Hover your mouse over the date for a more exact time/date.

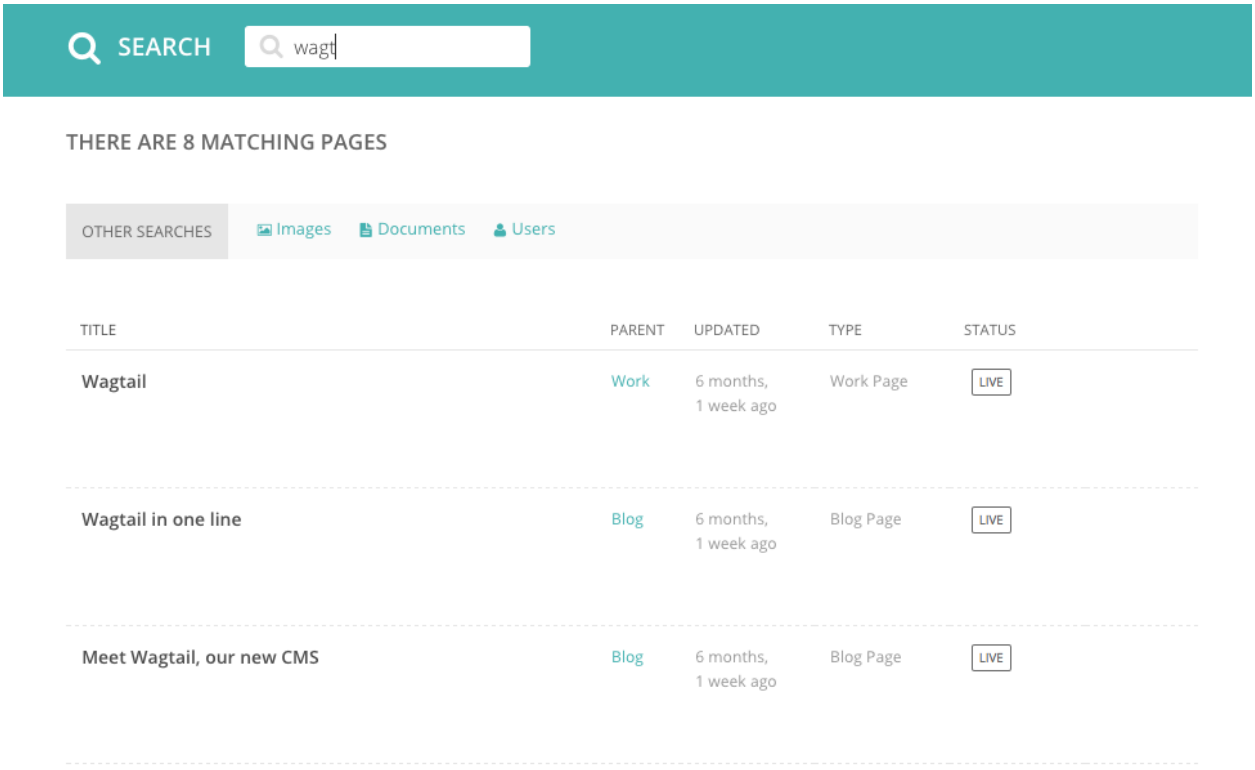
- The status column displays the current status of the page. A page will have one of three statuses:
  - Live: Published and accessible to website visitors
  - Draft: Not live on the website.
  - Live + Draft: A version of the page is live, but a newer version is in draft mode.

The Explorer menu



- Click the Explorer button in the sidebar to open the site explorer. This allows you to navigate through the tree-structure of the site.
- Clicking the name of a page will take you to the Explorer page for that section (see below). NOTE: The site explorer only displays pages which themselves have child pages. To see and edit the child pages you should click the name of the parent page in the site explorer.
- Clicking the green arrow displays the sub-sections (see below).
- Clicking the back button takes you back to the parent section.
- Again, clicking the section title takes you to the Explorer page.
- Clicking further arrows takes you deeper into the tree.

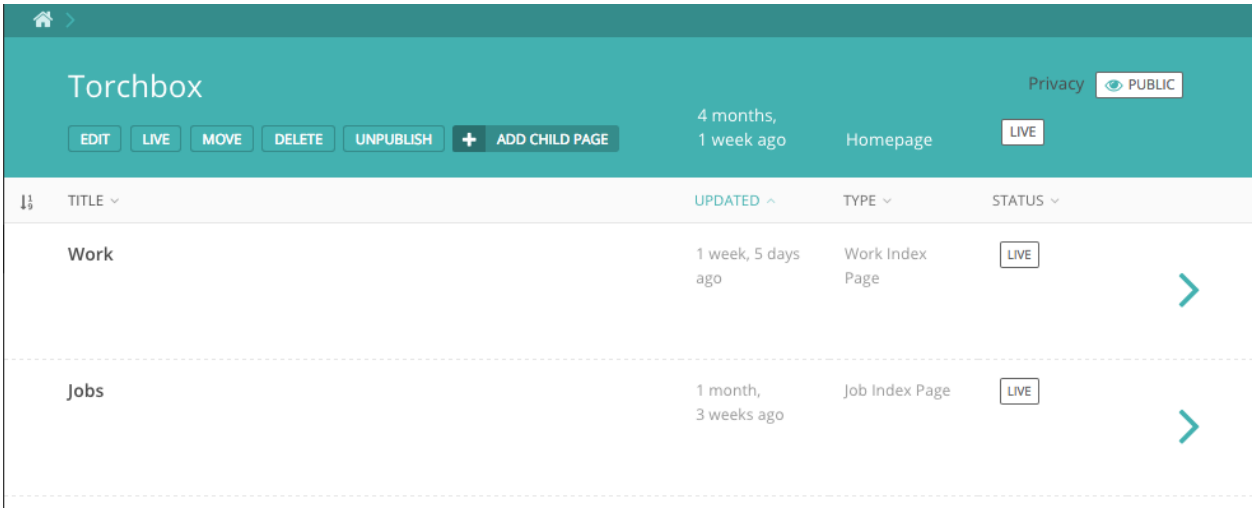
Using search



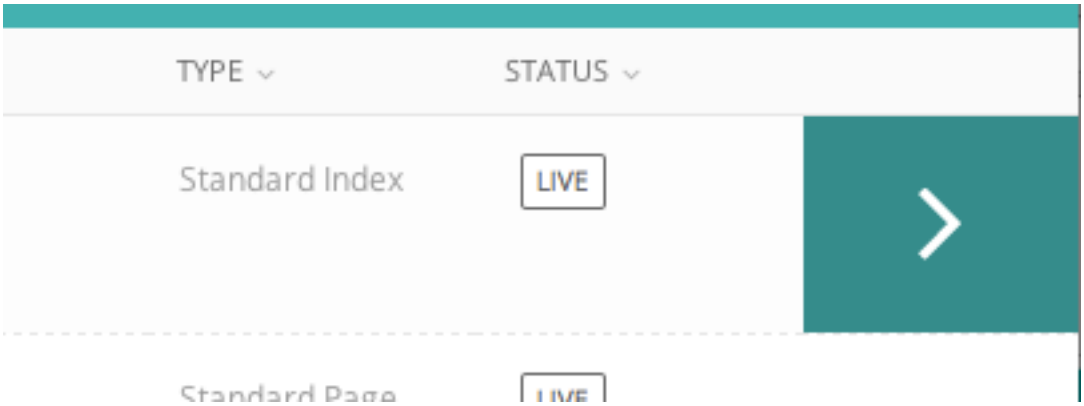
- A very easy way to find the page that you want is to use the main search feature, accessible from the left-hand menu.
- Simply type in part or all of the name of the page you are looking for, and the results below will automatically update as you type.
- Clicking the page title in the results will take you to the Edit page for that result. You can differentiate between similar named pages using the Parent column, which tells you what the parent page of that page is.

The Explorer page

The Explorer page allows you to view the a page's children and perform actions on them. From here you can publish/unpublish pages, move pages to other sections, drill down further into the content tree, or reorder pages under the parent for the purposes of display in menus.



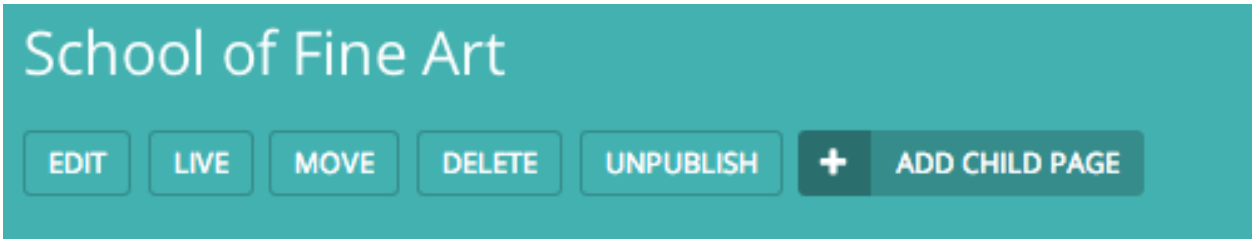
- The name of the section you are looking at is displayed below the breadcrumb (the row of page names beginning with the home icon). Each section is also itself a page (in this case the homepage). Clicking the title of the section takes you to the Edit screen for the section page.
- As the heading suggests, below are the child pages of the section. Clicking the titles of each child page will take you to its Edit screen.



- Clicking the arrows will display a further level of child pages.

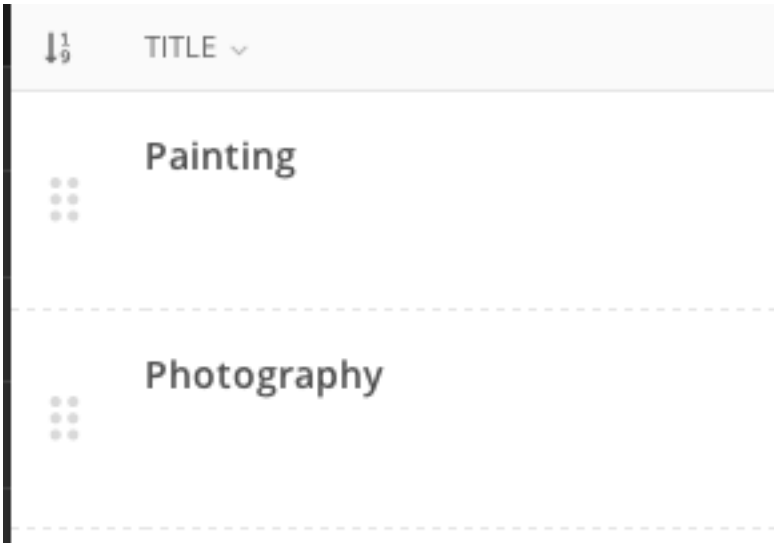


- As you drill down through the site the breadcrumb (the row of pages beginning with the home icon) will display the path you have taken. Clicking on the page titles in the breadcrumb will take you to the Explorer screen for that page.



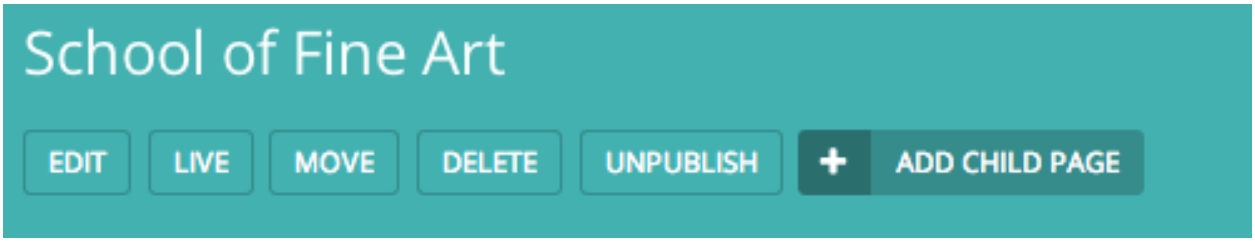
- To add further child pages press the Add child page button below the parent page title. You can view the parent page on the live site by pressing the View live button. The Move button will take you to the Move page screen where you can reposition the page and all its child pages in the site structure.
- Similar buttons are available for each child page. These are made visible on hover.

Reordering pages



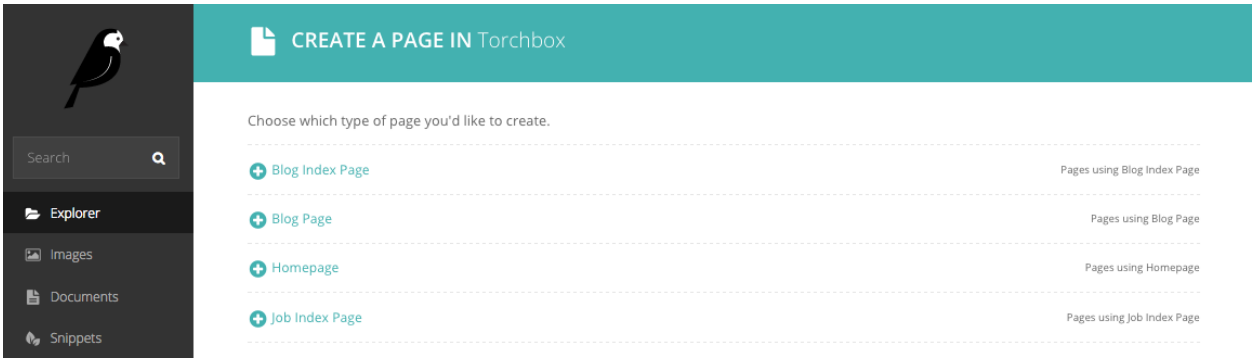
- Clicking the icon to the far left of the child pages table will enable the reordering handles. This allows you to reorder the way that content displays in the main menu of your website.
- Reorder by dragging the pages by the handles on the far left (the icon made up of 6 dots).
- Your new order will be automatically saved each time you drag and drop an item.

Creating new pages

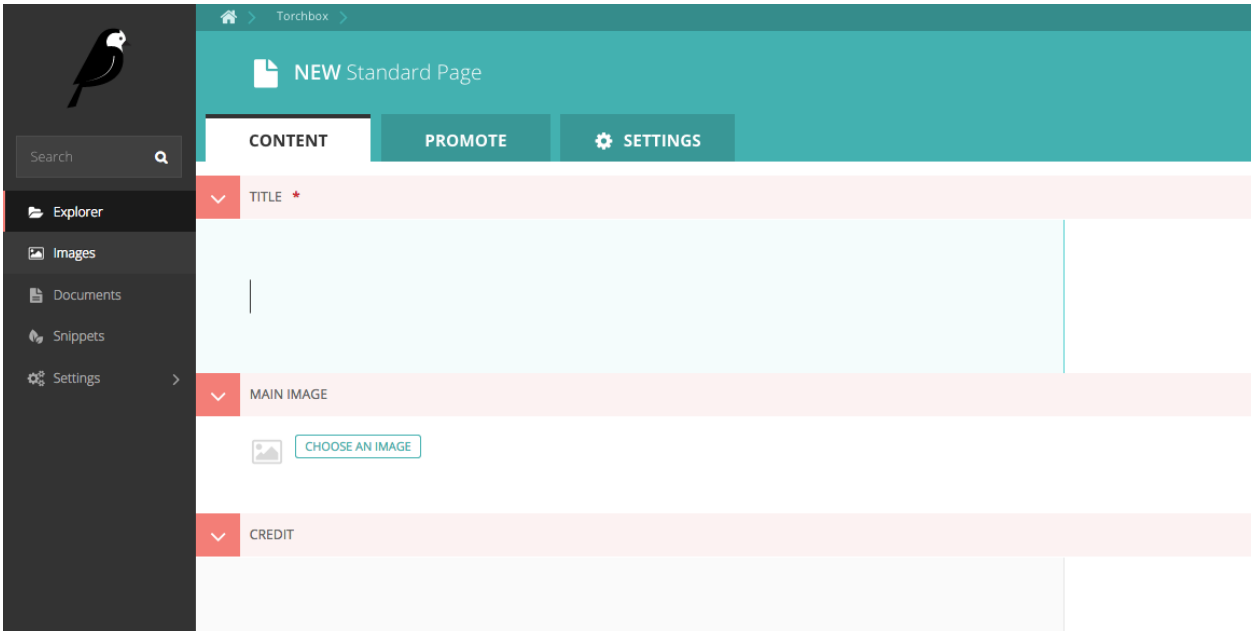


Create new pages by clicking the **Add child page** button. This creates a child page of the section you are currently in. In this case a child page of the 'School of Fine Art' page.

Selecting a page type



- On the left of the page chooser screen are listed all the types of pages that you can create. Clicking the page type name will take you to the Create new page screen for that page type (see below).
- Clicking the *Pages using … Page* links on the right will display all the pages that exist on the website of this type. This is to help you judge what type of page you will need to complete your task.



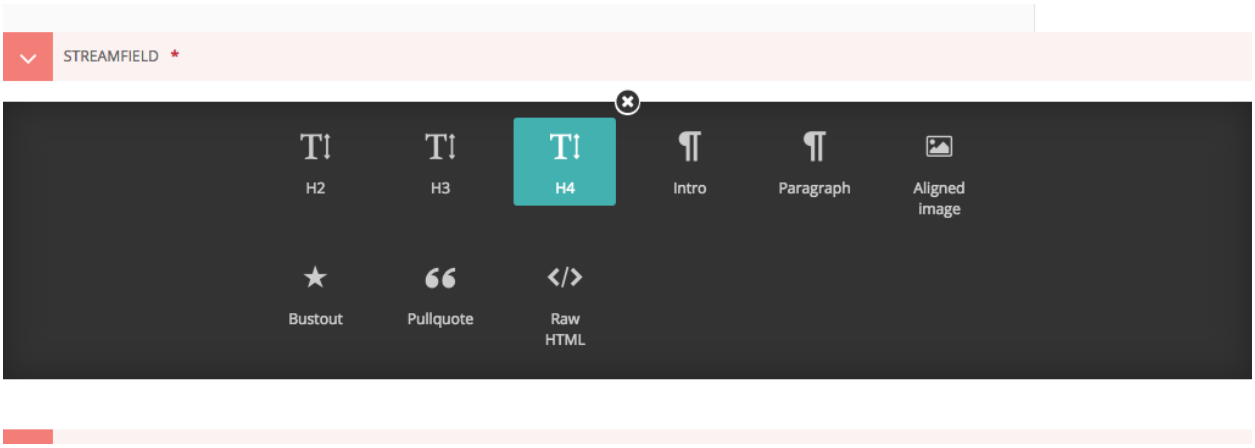
- Once you've selected a page type you will be presented with a blank New page screen.
- Click into the areas below each field's heading to start entering content.

Creating page body content

Wagtail supports a number of basic fields for creating content, as well as our unique StreamField feature which allows you to construct complex layouts by combining these basic fields in any order.

StreamField

StreamField allows you to create complex layouts of content on a page by combining a number of different arrangements of content, 'blocks', in any order.



When you first edit a page, you will be presented with the empty StreamField area, with the option to choose one of several block types. The block types on your website may be different from the screenshot here, but the principles are the same.

Click the block type, and the options will disappear, revealing the entry field for that block.

Depending on the block you chose, the field will display differently, and there might even be more than one field! There are a few common field types though that we will talk about here.

- Basic text field
- Rich text field
- Image field

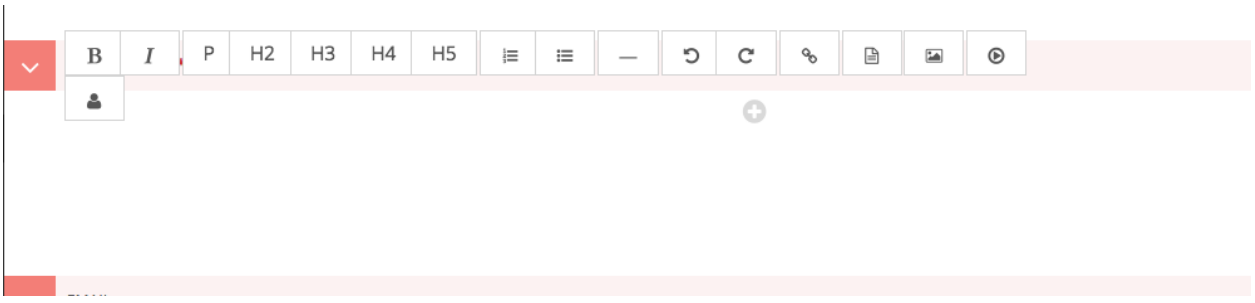
Basic text field

Basic text fields have no formatting options. How these display will be determined by the style of the page in which they are being inserted. Just click into the field and type!

Rich text fields

Most of the time though, you need formatting options to create beautiful looking pages. So some fields, like the fields in the 'Paragraph block' shown in the screenshot, have many of the options you would expect from a word processor. These are referred to as rich text fields.

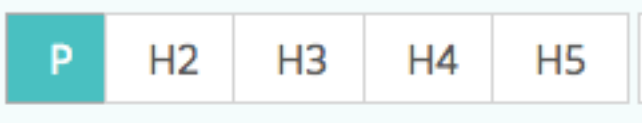
So, when you click into one of these fields, you will be presented with a set of tools which allow you to format and style your text. These tools also allow you to insert links, images, videos clips and links to documents.



Below is a summary of what the different buttons represent:



Bold / Italic: Either click then type for bold or italic, or highlight and select to convert existing text to bold or italic.



Paragraph / heading levels: Clicking into a paragraph and selecting one of these options will change the level of the text. H1 is not included as this is reserved for the page title.



Bulleted and numbered lists



Horizontal rule: Creates a horizontal line at the position of the cursor. If inserted inside a paragraph it will split the paragraph into two separate paragraphs.



Undo / redo: As expected will undo or redo the latest actions. Never use the your browser's back button when attempting to undo changes as this could lead to errors. Either use this undo button, or the usual keyboard shortcut, CTRL+Z.

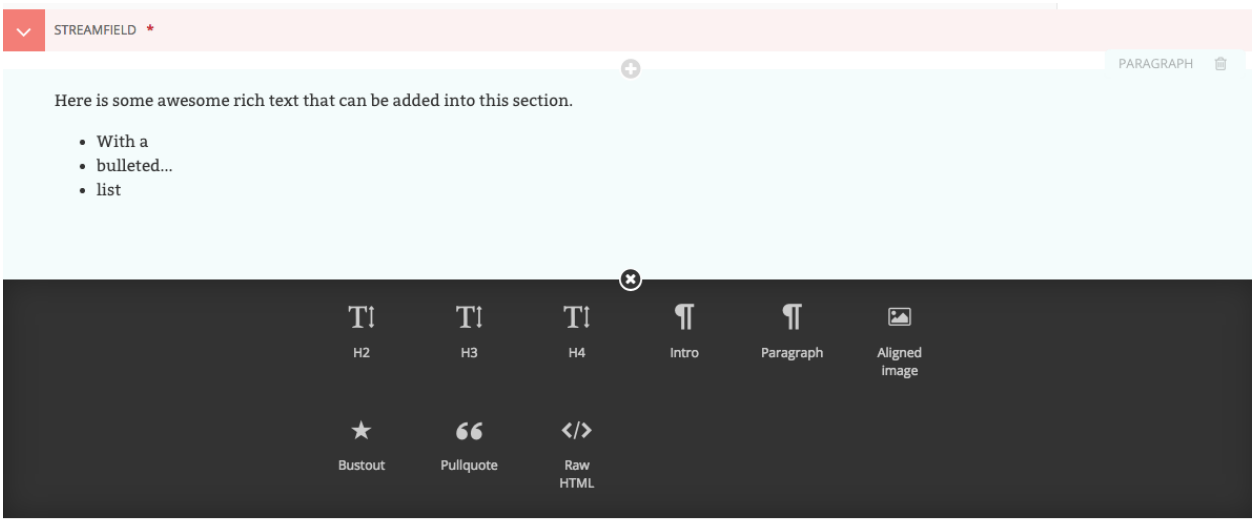


Insert image / video: Allows you to insert an image or video into the rich text field. See Inserting images and videos section for more details. See *Inserting images <inserting_images.html>* and *Inserting videos <inserting_videos.html>* sections.



Insert link / document: Allows you to insert a link or a document into the rich text field. See Inserting links and Inserting documents for more details. See *Inserting links section <inserting_links.html>*.

Adding further blocks in StreamField



- To add new blocks, click the '+' icons above or below the existing blocks.
- You'll then be presented once again with the different blocks from which you may choose.
- You can cancel the addition of a new block by clicking the cross at the top of the block selection interface.

Reordering and deleting content in StreamField



- Click the arrows on the right-hand side of each block to move blocks up and down in the StreamField order of content.
- The blocks will be displayed in the front-end in the order that they are placed in this interface.
- Click the rubbish bin on the far right to delete a field

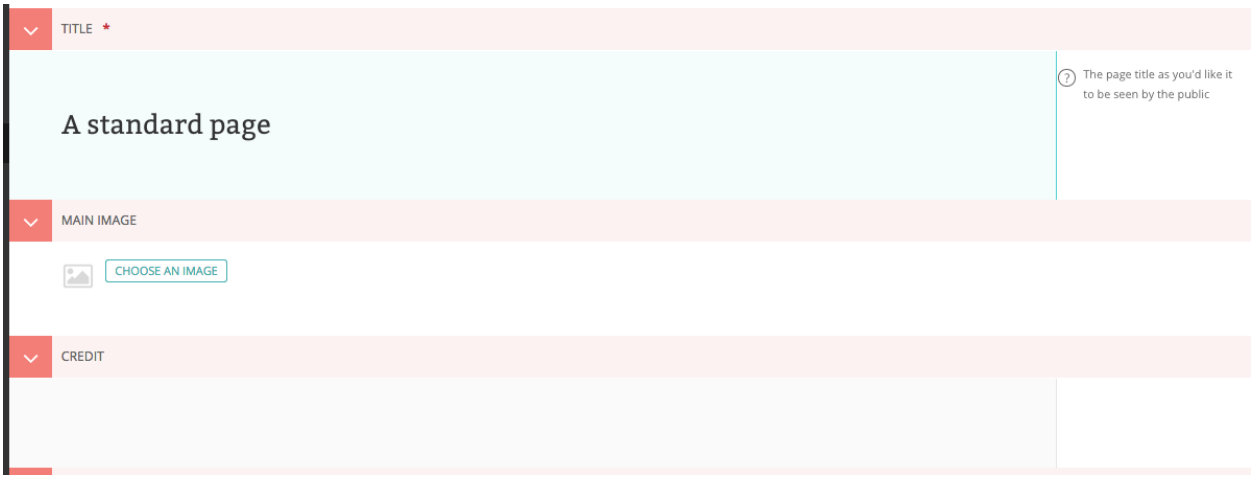
Warning: Once a StreamField field is deleted it cannot be retrieved if the page has not been saved. Save your pages regularly so that if you acidentally delete a field you can reload the page to undo your latest edit.

Inserting images and videos in a page

There will obviously be many instances in which you will want to add images to a page. There are two main ways to add images to pages, either via a specific image chooser field, or via the rich text field image button. Which of these you use will be dependent on the individual setup of your site.

Inserting images using the image chooser field

Often a specific image field will be used for a main image on a page, or for an image to be used when sharing the page on social media. For the standard page on Torchbox.com, the former is used.



- You insert an image by clicking the *Choose an image* button.

Choosing an image to insert

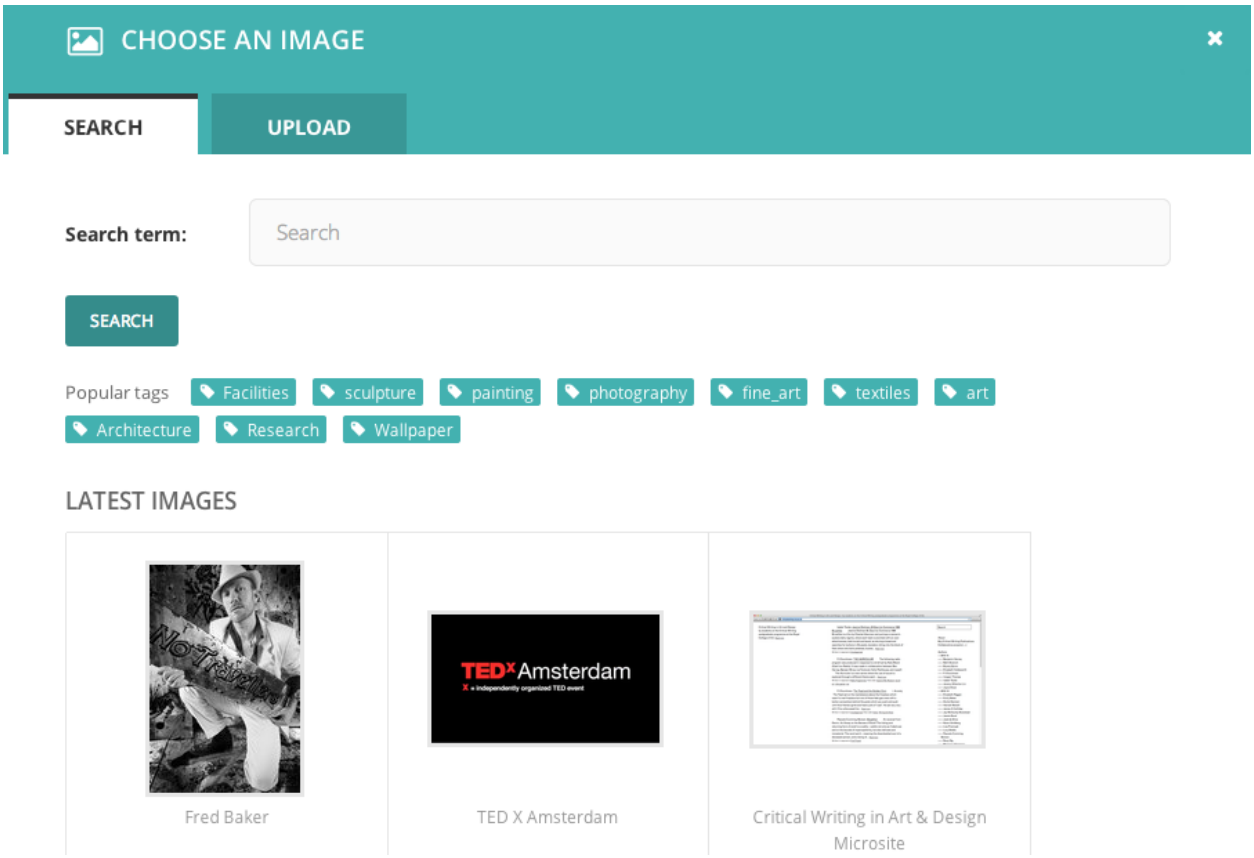
You have two options when selecting an image to insert:

1. Selecting an image from the existing image library, or. . .
2. Uploading a new image to the CMS

When you click the *Choose an image* button you will be presented with a pop-up with two tabs at the top. The first, Search, allows you to search and select from the library. The second, *Upload*, allows you to upload a new image.

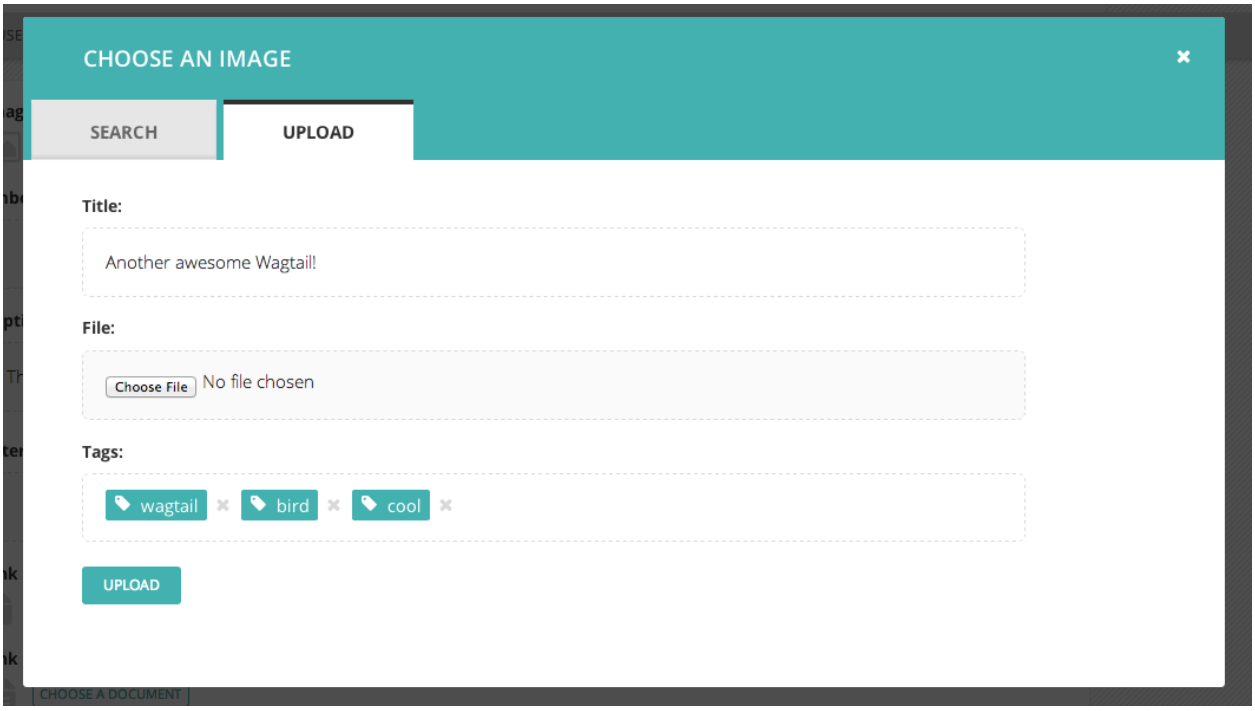
Choosing an image from the image library

The image below demonstrates finding and inserting an image that is already present in the CMS image library.



1. Typing into the search box will automatically display the results below.
2. Clicking one of the Popular tags will filter the search results by that tag.
3. Clicking an image will take you to the Choose a format window (see image below).

Uploading a new image to the CMS



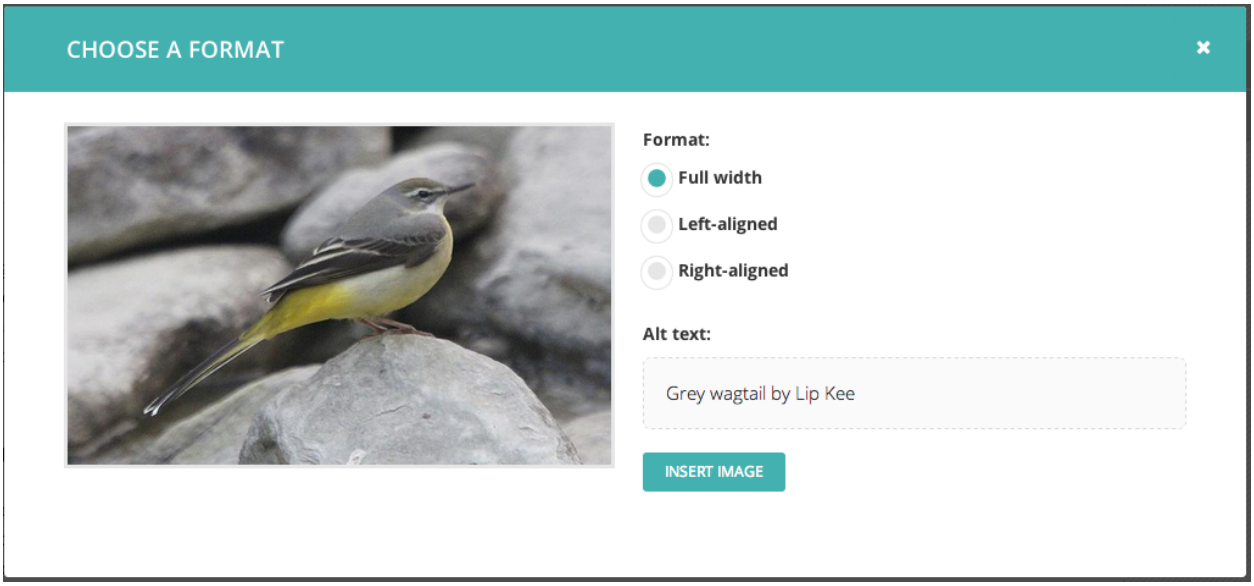
1. You must include an image title for your uploaded image
2. Click the *Choose file* button to choose an image from your computer.
3. *Tags* allows you to associate tags with the image you are uploading. This allows them to be more easily found when searching. Each tag should be separated by a space. Good practice for creating multiple word tags is to use an underscore between each word (e.g. western_yellow_wagtail).
4. Click *Upload* to insert the uploaded image into the carousel. The image will also be added to the main CMS image library for reuse in other content.

Inserting images using the rich text field



Images can also be inserted into the body text of a page via the rich text editor. When working in a rich text field, click the image illustrated above. You will then be presented with the same options as for inserting images into the main carousel.

In addition, Wagtail allows you to choose an alignment for you image.



1. You can select how the image is displayed by selecting one of the format options.
2. You must provide specific alt text for your image.

The alignments available are described below:

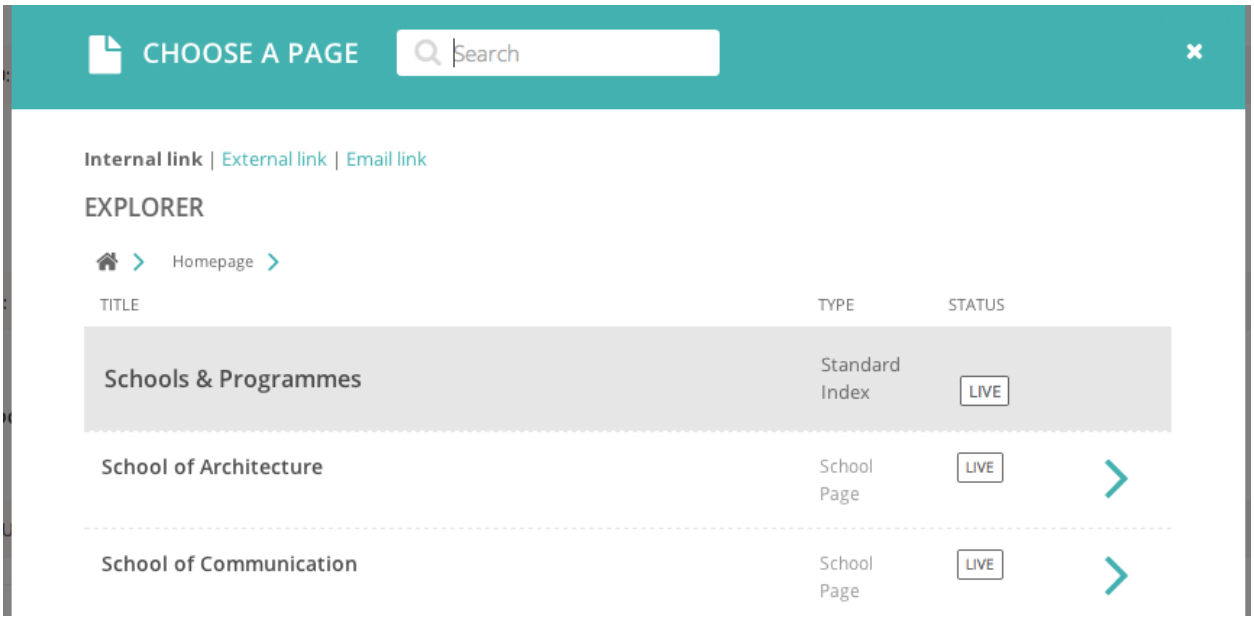
- **Full width:** Image will be inserted using the full width of the text area.
- **Half-width left/right aligned:** Inserts the image at half the width of the text area. If inserted in a block of text the text will wrap around the image. If two half-width images are inserted together they will display next to each other.

Note: The display of images aligned in this way is dependent on your implementation of Wagtail, so you may get slightly different results.

Inserting links in a page

Similar to images, there are a variety of points at which you will want to add links. The most common place to insert a link will be in the body text of a page. You can insert a link into the body text by clicking the **Insert link** button in the rich text toolbar.

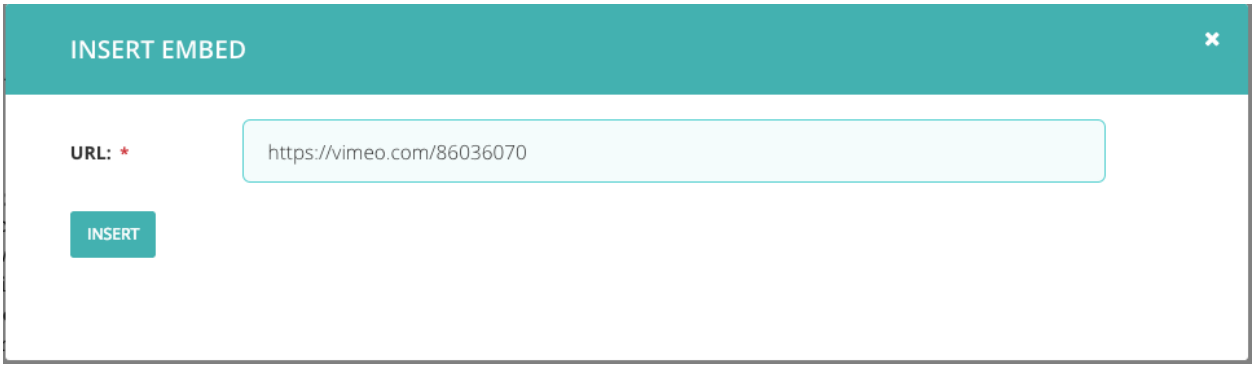
Whichever way you insert a link, you will be presented with the form displayed below.



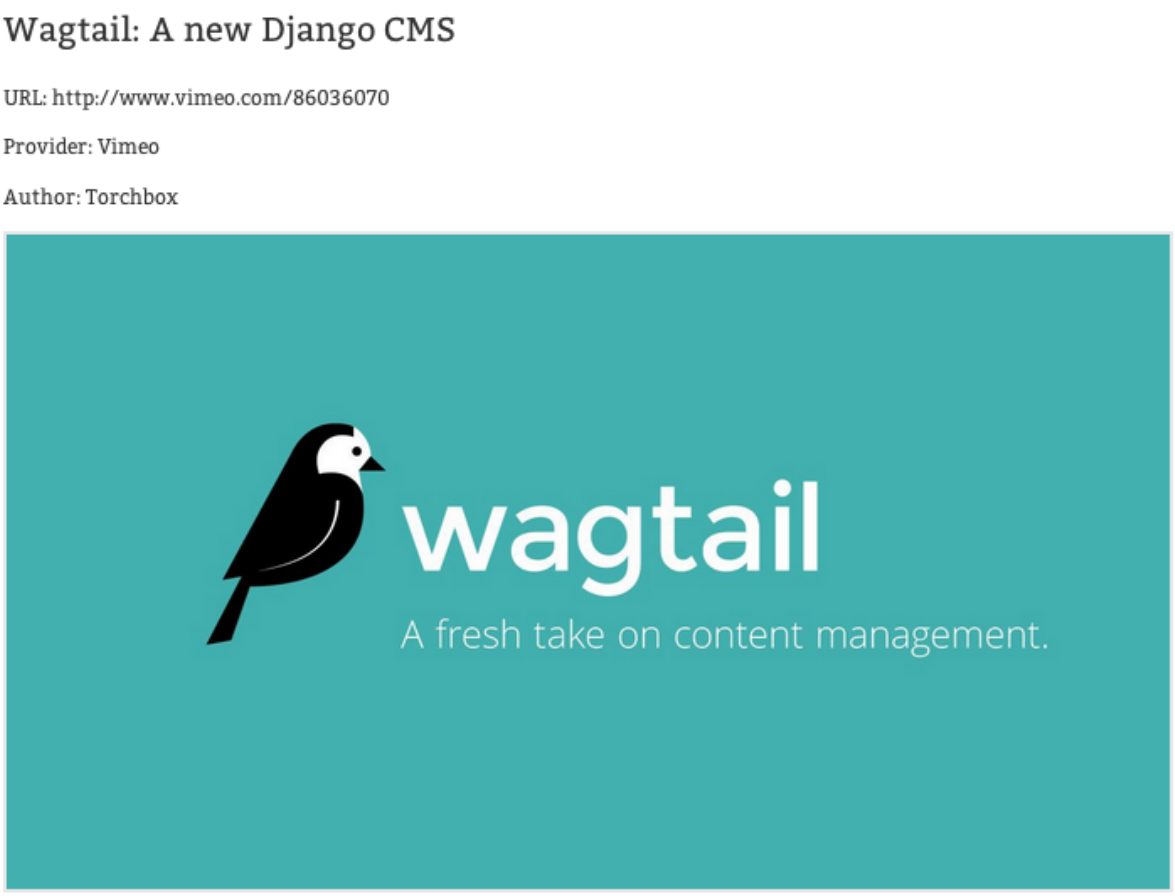
- Search for an existing page to link to using the search bar at the top of the pop-up.
- Below the search bar you can select the type of link you want to insert. The following types are available:
  - Internal link: A link to an existing page within your website.
  - External link: A link to a page on another website.
  - Email link: A link that will open the user's default email client with the email address prepopulated.
- You can also navigate through the website to find an internal link via the explorer.

Inserting videos into body content

As well as inserting videos into a carousel, Wagtail's rich text fields allow you to add videos into the body of a page by clicking the *Add video* button in the toolbar.



- Copy and paste the web address for the video (either YouTube or Vimeo) into the URL field and click Insert.



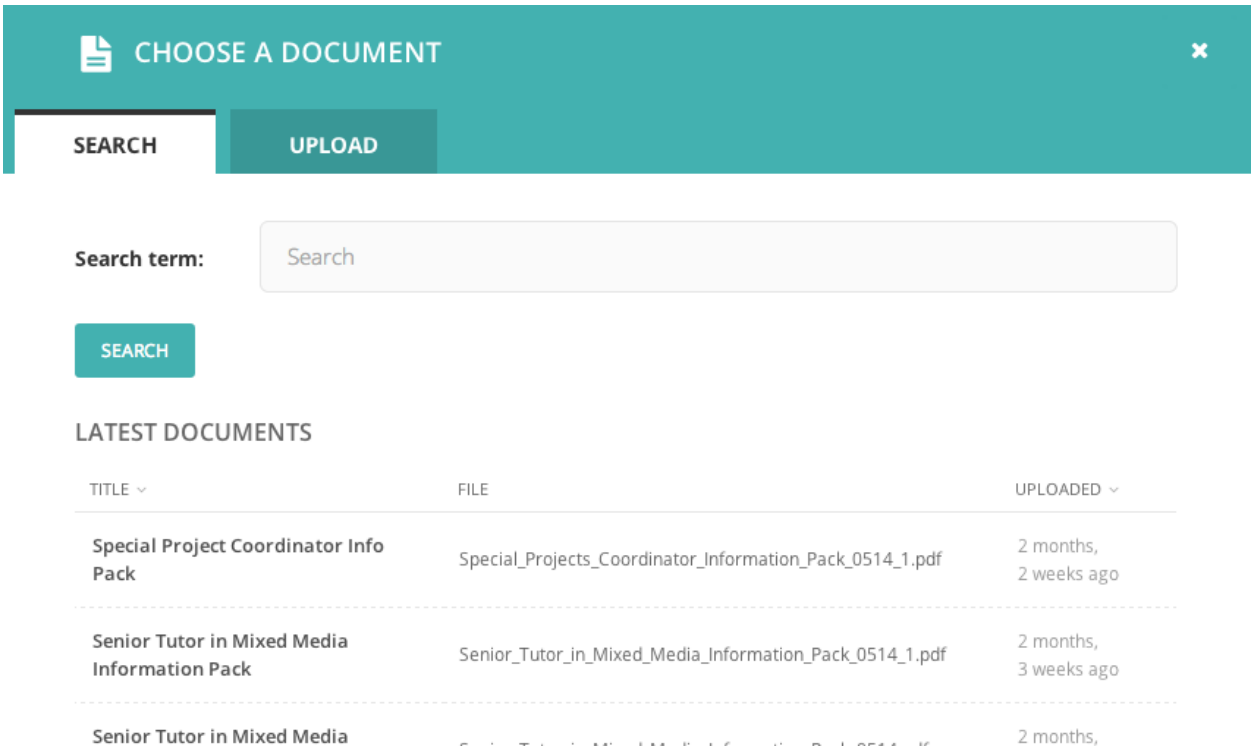
- A placeholder with the name of the video and a screenshot will be inserted into the text area. Clicking the X in the top corner will remove the video.

Inserting links to documents into body text



It is possible to insert links to documents held in the CMS into the body text of a web page by clicking the button above in the rich text field.

The process for doing this is the same as when inserting an image. You are given the choice of either choosing a document from the CMS, or uploading a new document.

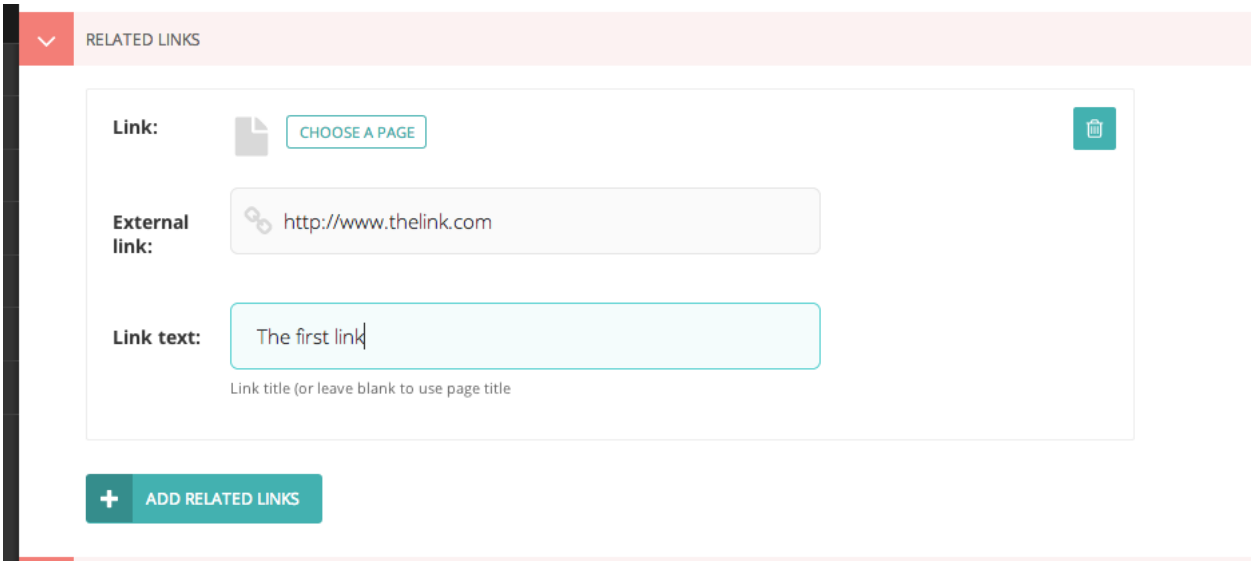


Adding multiple items

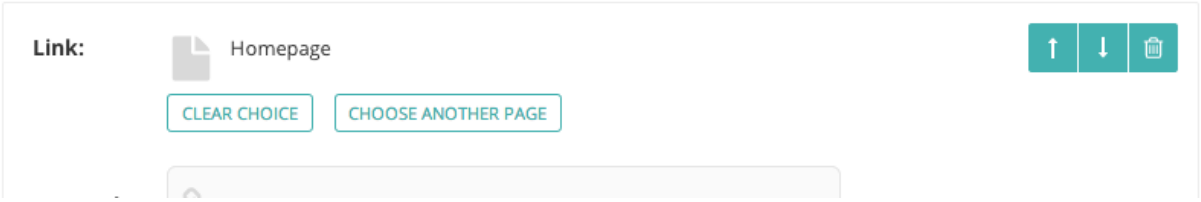
A common feature of Wagtail is the ability to add more than one of a particular type of field or item. For example, you can add as many carousel items or related links as you wish.



- Whenever you see the white cross in the green circle illustrated here it means you can add multiple objects or items to a page. Clicking the icon will display the fields required for that piece of content. The image below demonstrates this with a *Related link* item.



- You can delete an individual item by pressing the trash can in the top-right.
- You can add more items by clicking the link with the white cross again.



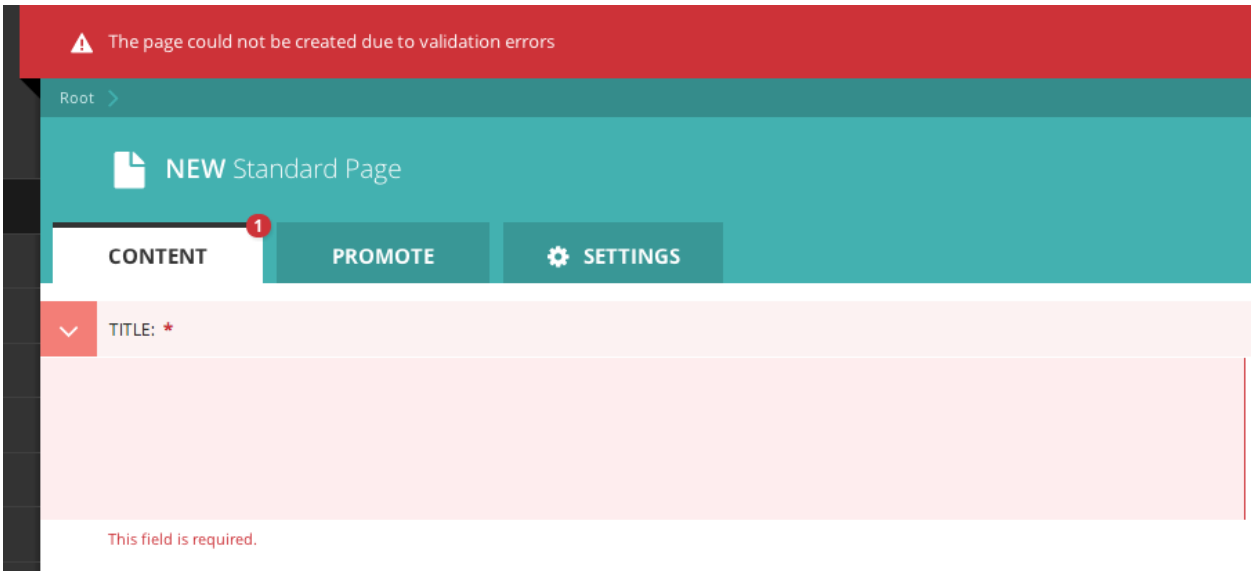
- You can reorder your multiple items using the up and down arrows. Doing this will affect the order in which they are display on the live page.

Required fields

- Fields marked with an asterisk are required. You will not be able to save a draft or submit the page for moderation without these fields being completed.



- If you try to save/submit the page with some required fields not filled out, you will see the error displayed here.
- The number of validation errors for each of the *Promote* and *Content* tabs will appear in a red circle, and the text, 'This field is required', will appear below each field that must be completed.

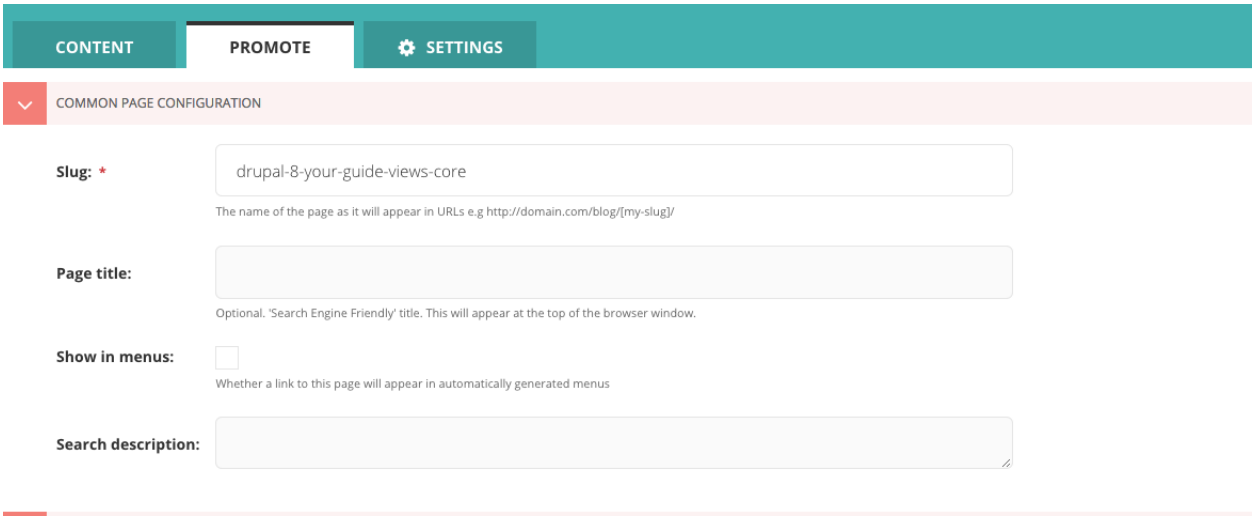


The Promote tab

A common feature of the *Edit* pages for all page types is the two tabs at the top of the screen. The first, Content, is where you build the content of the page itself.

The second, *Promote*, is where you can set all the 'metadata' (data about data!) for the page. Below is a description of all default fields in the promote tab and what they do.

- **Slug:** The last part of the web address for the page. E.g. the slug for a blog page called 'The best things on the web' would be the-best-things-on-the-web (`www.example.com/blog/the-best-things-on-the-web`). This is automatically generated from the main page title set in the Content tab. This can be overridden by adding a new slug into the field. Slugs should be entirely lowercase, with words separated by hyphens (-).
- **Page title:** An optional, search-engine friendly page title. This is the title that appears in the tab of your browser window. It is also the title that would appear in a search engine if the page was returned as part of a set of search results.
- **Show in menus:** Ticking this box will ensure that the page is included in automatically generated menus on your site. Note: Pages will only display in menus if all of its parent pages also have *Show in menus* ticked.
- **Search description:** This field allows you to add text that will be displayed if the page appears in search results. This is especially useful to distinguish between similarly named pages.

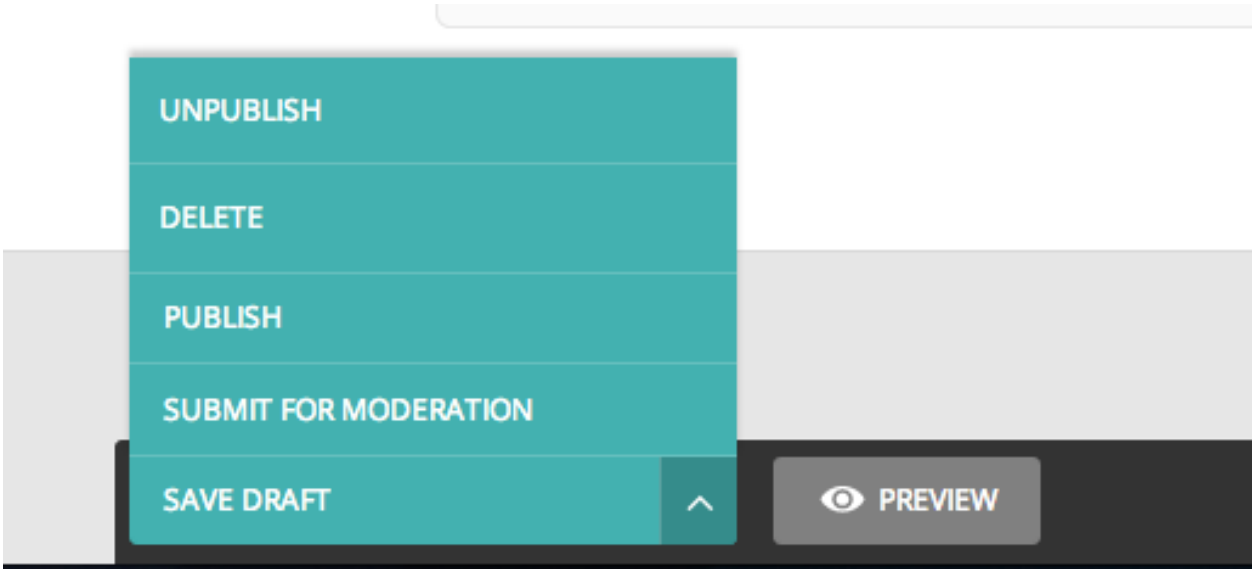


Note: You may see more fields than this in your promote tab. These are just the default fields, but you are free to add other fields to this section as necessary.

Previewing and submitting pages for moderation

The Save/Preview/Submit for moderation menu is always present at the bottom of the page edit/creation screen. The menu allows you to perform the following actions, dependent on whether you are an editor, moderator or administrator:

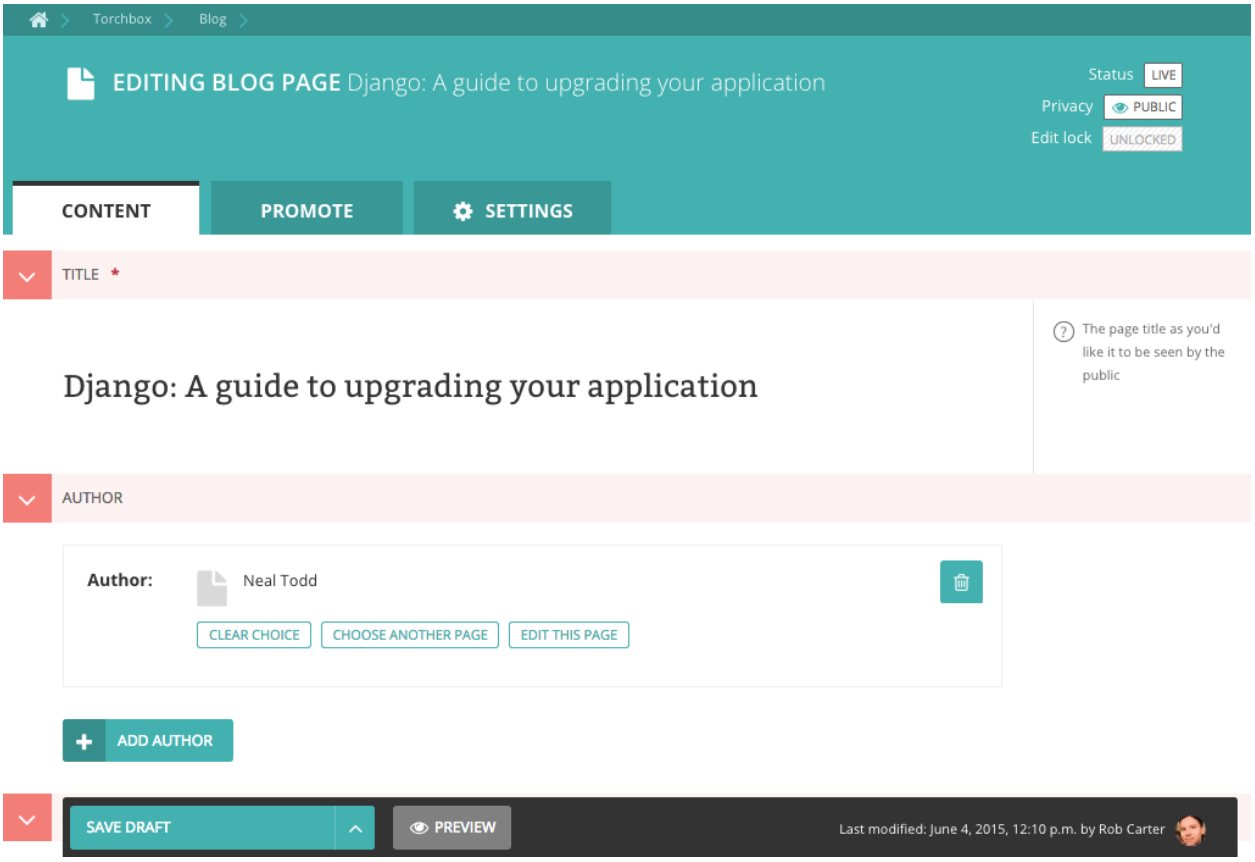
- **Save draft:** Saves your current changes but doesn't submit the page for moderation and so won't be published. (all roles)
- **Submit for moderation:** Saves your current changes and submits the page for moderation. A moderator will be notified and they will then either publish or reject the page. (all roles)
- **Preview:** Opens a new window displaying the page as it would look if published, but does not save your changes or submit the page for moderation. (all roles)
- **Publish/Unpublish:** Clicking either the *Publish* or *Unpublish* buttons will take you to a confirmation screen asking you to confirm that you wish to publish or unpublish this page. If a page is published it will be accessible from its specific URL and will also be displayed in site search results. (moderators and administrators only)
- **Delete:** Clicking this button will take you to a confirmation screen asking you to confirm that you wish to delete the current page. Be sure that this is actually what you want to do, as deleted pages are not recoverable. In many situations simply unpublishing the page will be enough. (moderators and administrators only)



Editing existing pages

There are two ways that you can access the edit screen of an existing page:

- Clicking the title of the page in an *Explorer page* or in *search results*.
- Clicking the *Edit* link below the title in either of the situations above.



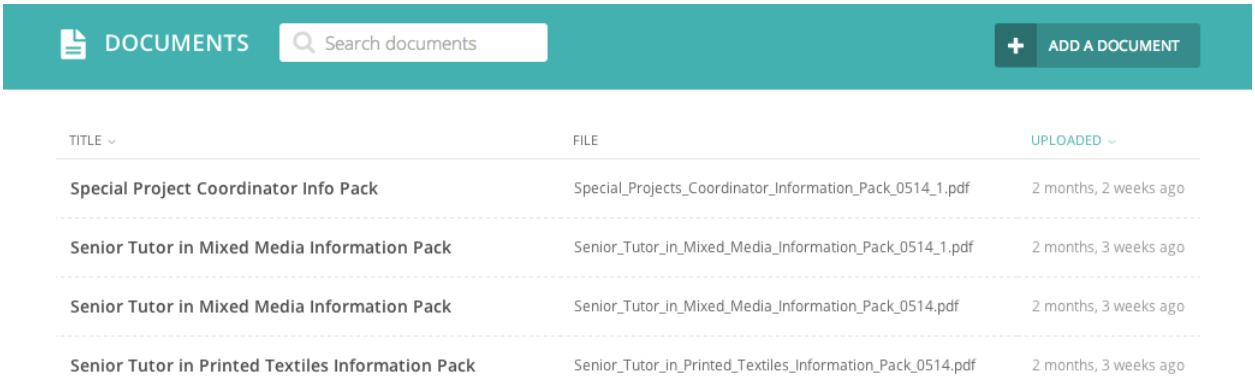
- When editing an existing page the title of the page being edited is displayed at the top of the page.
- The current status of the page is displayed in the top-right.
- You can change the title of the page by clicking into the title field.
- When you are typing into a field, help text is often displayed on the right-hand side of the screen.

Managing documents and images

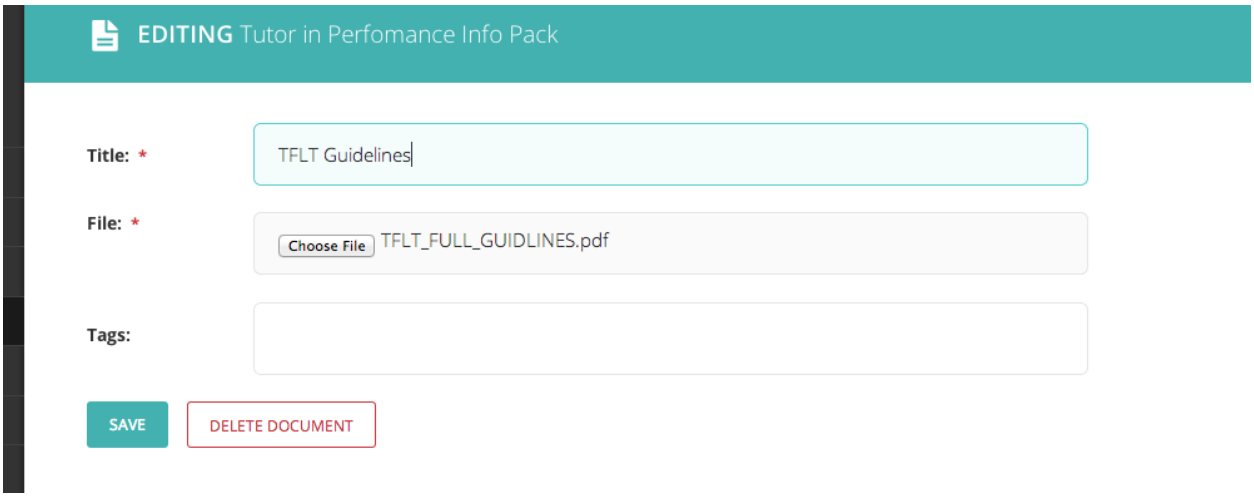
Wagtail allows you to manage all of your documents and images through their own dedicated interfaces. See below for information on each of these elements.

Documents

Documents such as PDFs can be managed from the Documents interface, available in the left-hand menu. This interface allows you to add documents to and remove documents from the CMS.



- Add documents by clicking the *Add document* button in the top-right.
- Search for documents in the CMS by entering your search term in the search bar. The results will be automatically updated as you type.
- You can also filter the results by *Popular tags*. Click on a tag to update the search results listing.
- Edit the details of a document by clicking the document title.

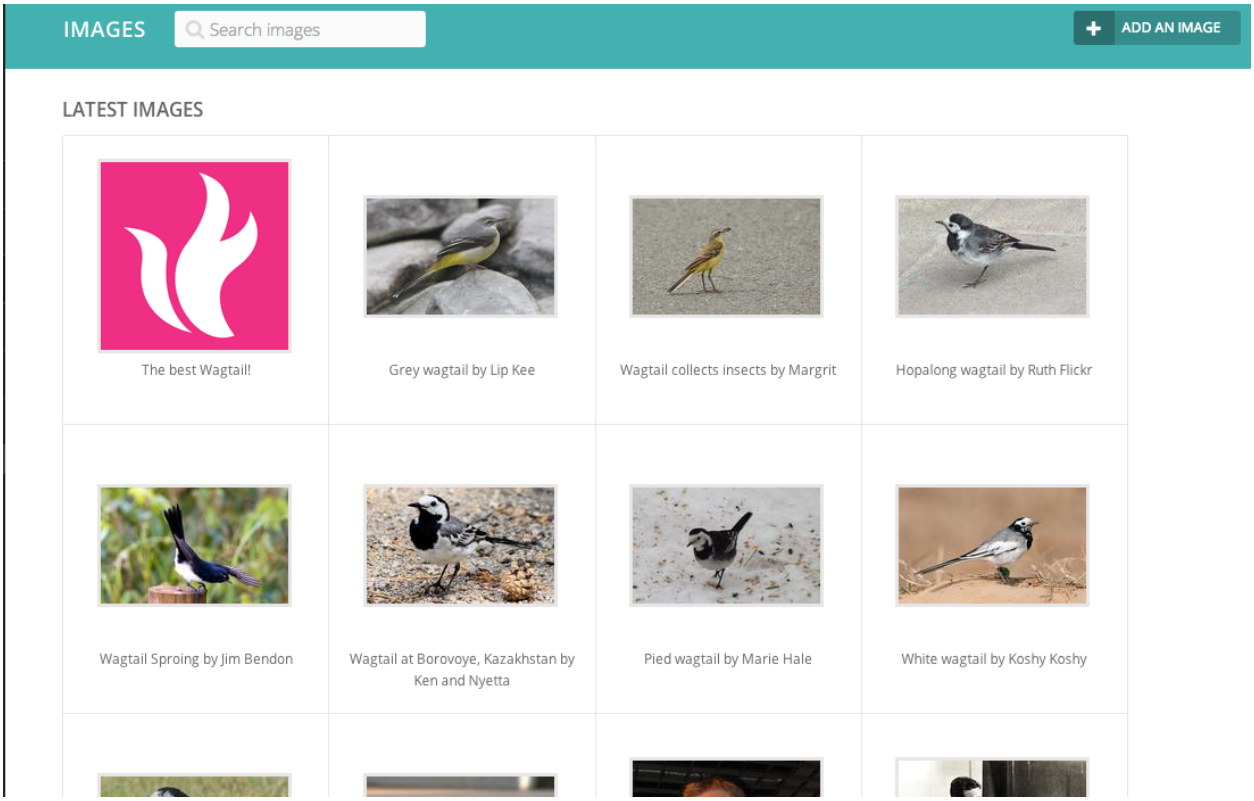


- When editing a document you can replace the file associated with that document record. This means you can update documents without having to update the pages on which they are placed. Changing the file will change it on all pages that use the document.
- Add or remove tags using the Tags field.
- Save or delete documents using the buttons at the bottom of the interface.

Warning: Deleted documents cannot be recovered.

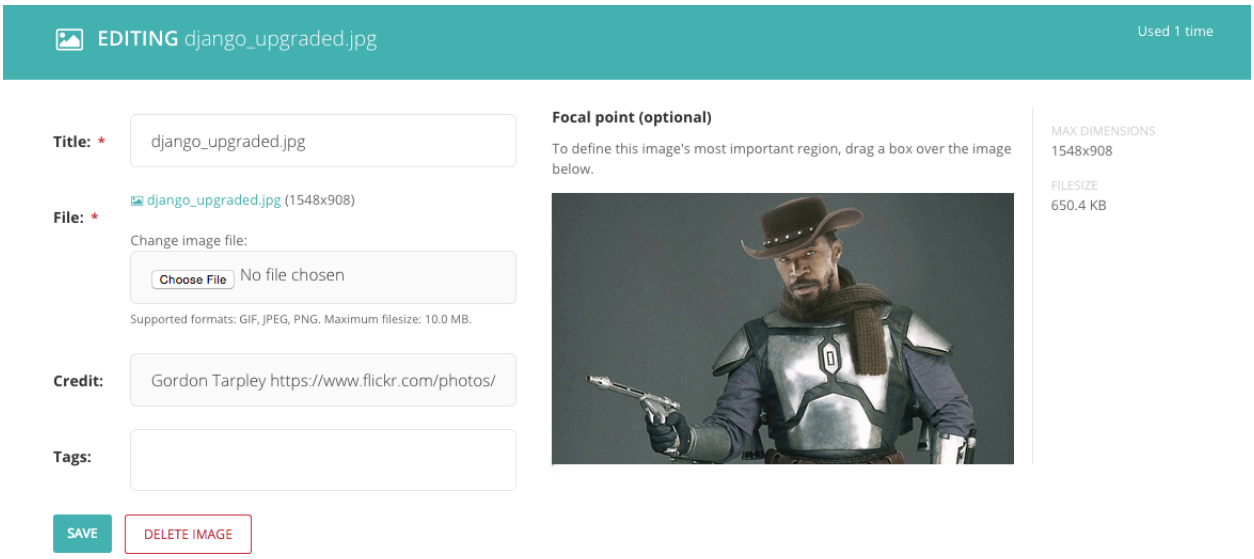
Images

If you want to edit, add or remove images from the CMS outside of the individual pages you can do so from the Images interface. This is accessed from the left-hand menu.



- Clicking an image will allow you to edit the data associated with it. This includes the Alt text, the photographers credit, the medium of the subject matter and much more.

Warning: Changing the alt text here will alter it for all occurrences of the image in carousels, but not in inline images, where the alt text can be set separately.



Changing the image

- When editing an image you can replace the file associated with that image record. This means you can update images without having to update the pages on which they are placed.

Warning: Changing the file will change it on all pages that use the image.

Focal point

- This interface allows you to select a focal point which can effect how your image displays to visitors on the front-end.
- If your images are cropped in some way to make them fit to a specific shape, then the focal point will define the centre point from which the image is cropped.
- To set the focal point, simply drag a marquee around the most important element of the image.
- If the feature is set up in your website, then on the front-end you will see the crop of this image focusing on your selection.

Snippets

Snippets allow you to create elements on a website once and reuse them in multiple places. Then, if you want to change something on the snippet, you only need to change it once, and it will change across all the occurances of the snippet.

How snippets are used can vary widely between websites. Here are a few examples of things Torchbox have used snippets for on our clients' websites:

- For staff contact details, so that they can be added to many pages but managed in one place
- For Adverts, either to be applied sitewide or on individual pages

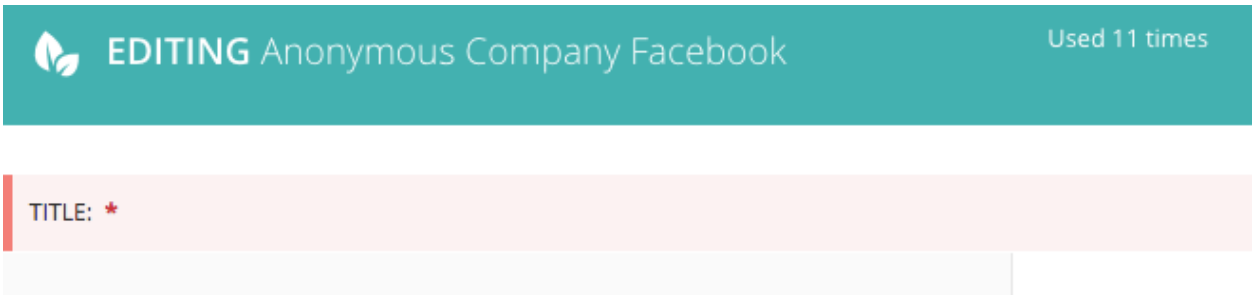
- To manage links in a global area of the site, for example in the footer
- For Calls to Action, such as Newsletter signup blocks, that may be consistent across many different pages

The Snippets menu



- You can access the Snippets menu by clicking on the 'Snippets' link in the left-hand menu bar.
- To add or edit a snippet, click on the snippet type you are interested in (often help text will be included to help you in selecting the right type)
- Click on an individual snippet to edit, or click 'Add ...' in the top right to add a new snippet

Warning: Editing a snippet will change it on all of the pages on which it has been used. In the top-right of the Snippet edit screen you will see a label saying how many times the snippet has been used. Clicking this label will display a listing of all of these pages.



Adding snippets whilst editing a page

If you are editing a page, and you find yourself in need of a new snippet, do not fear! You can create a new one without leaving the page you are editing:

- Whilst editing the page, open the snippets interface in a new tab, either by Ctrl+click (cmd+click on Mac) or by right clicking it and selecting 'Open in new tab' from the resulting menu.
- Add the snippet in this new tab as you normally would.
- Return to your existing tab and reopen the Snippet chooser window.
- You should now see your new snippet, even though you didn't leave the edit page.

Note: Even though this is possible, it is worth saving your page as a draft as often as possible, to avoid your changes being lost by navigating away from the edit page accidentally.

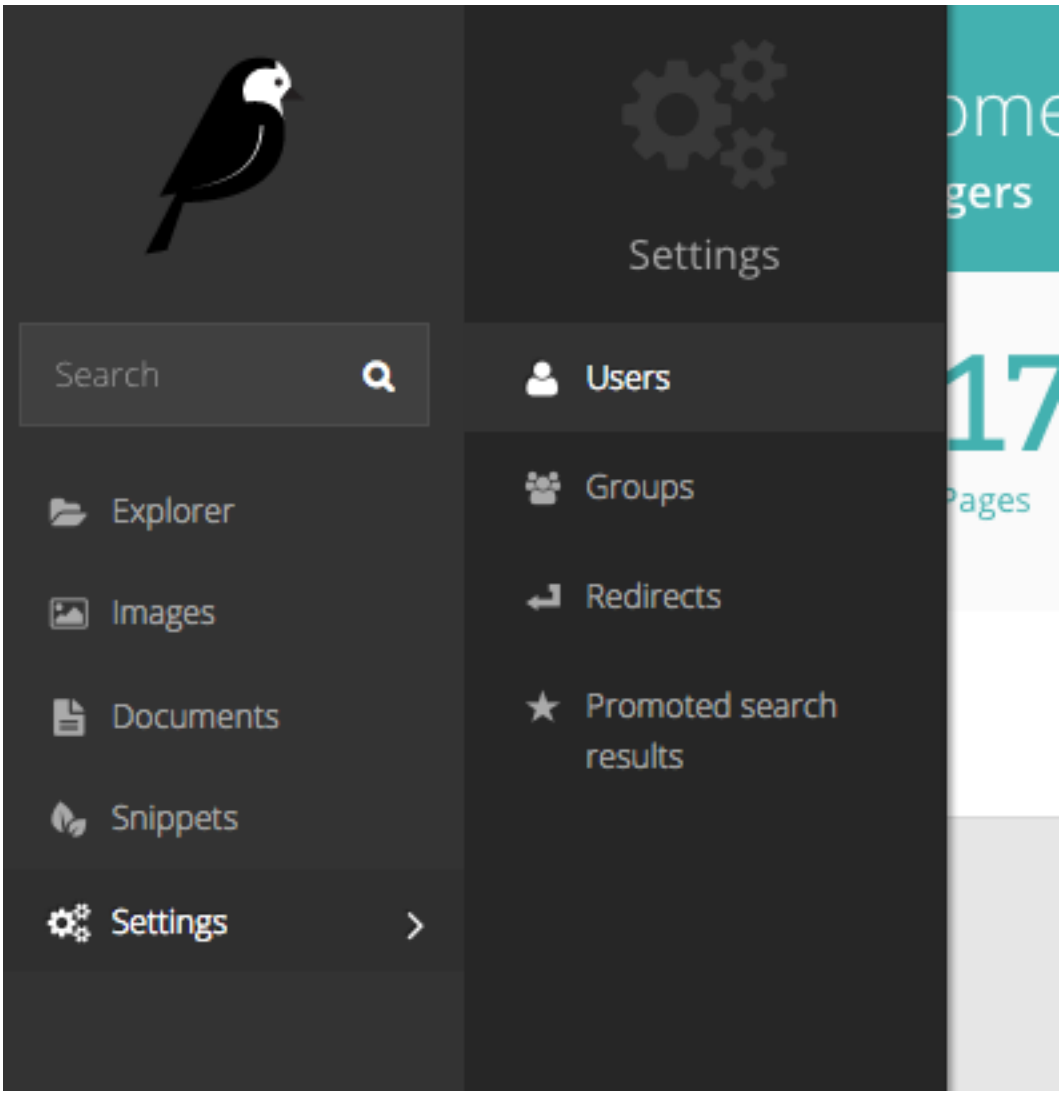
Administrator tasks

This section of the guide documents how to perform common tasks as an administrator of a Wagtail site.

Managing users and roles

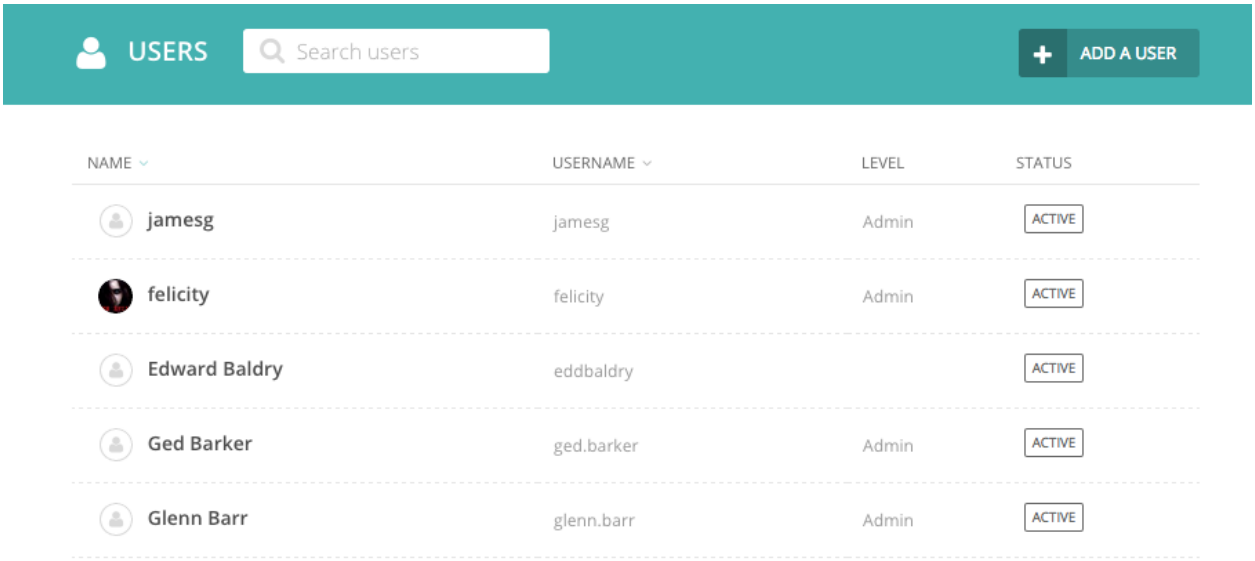
As an administrator, a common task will be adding, modifying or removing user profiles.

This is done via the 'Users' interface, which can be found in the Settings menu, accessible via the left-hand menu bar.



In this interface you can see all of your users, their usernames, their 'level' of access (otherwise known as their 'role'), and their status, either active or inactive.

You can sort this listing either via Name or Username.



Clicking on a user's name will open their profile details. From here you can then edit that users details.

Note: It is possible to change user's passwords in this interface, but it is worth encouraging your users to use the 'Forgotten password' link on the login screen instead. This should save you some time!

Click the 'Roles' tab to edit the level of access your users have. By default there are three roles:

Role	Create drafts	Publish content	Access Settings
Editor	Yes	No	No
Moderator	Yes	Yes	No
Administrator	Yes	Yes	Yes

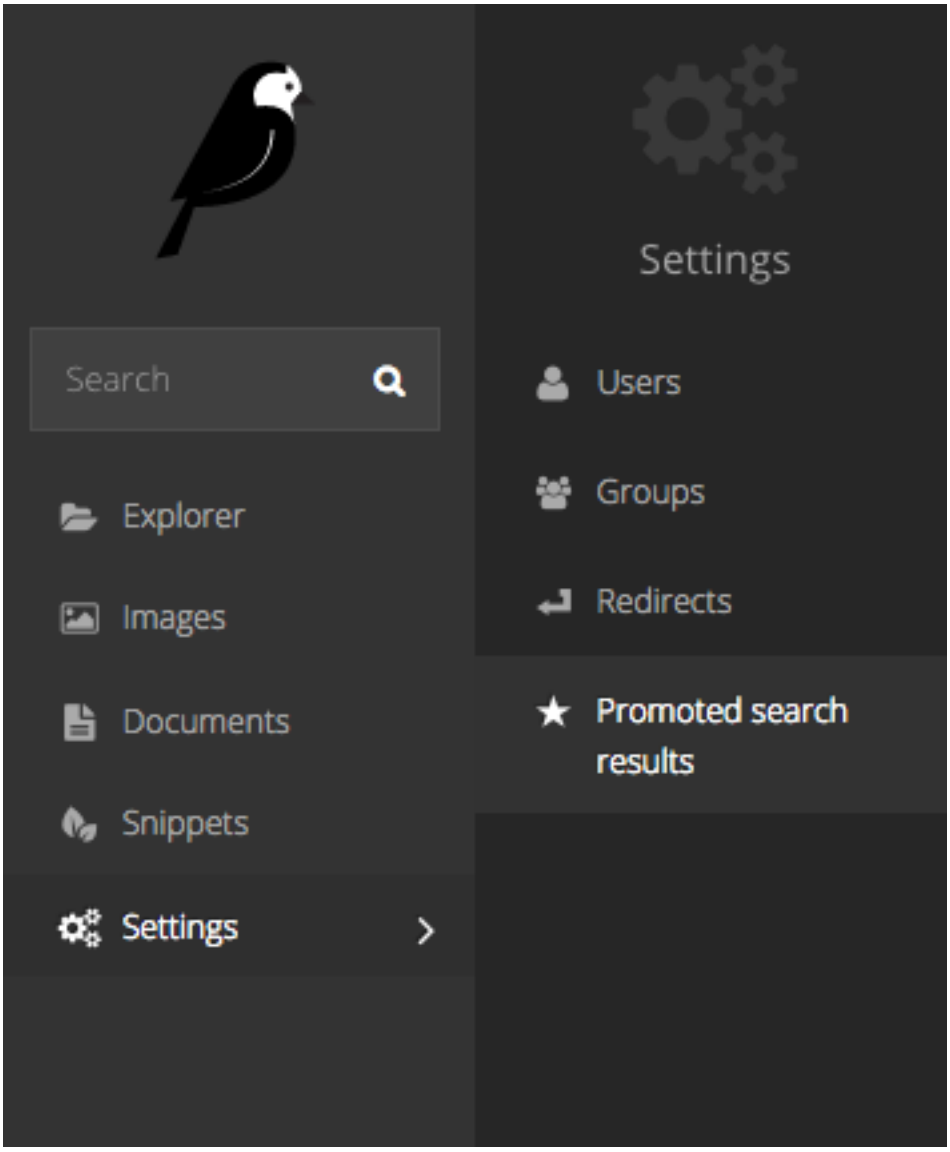
Promoted search results

Wagtail allows you to promote certain search results dependant on the keyword or phrase entered by the user when searching. This can be particularly useful when users commonly refer to parts of your organisation via an acronym that isn't in official use, or if you want to direct users to a page that when they enter a certain term related to the page but not included in the text of the page itself.

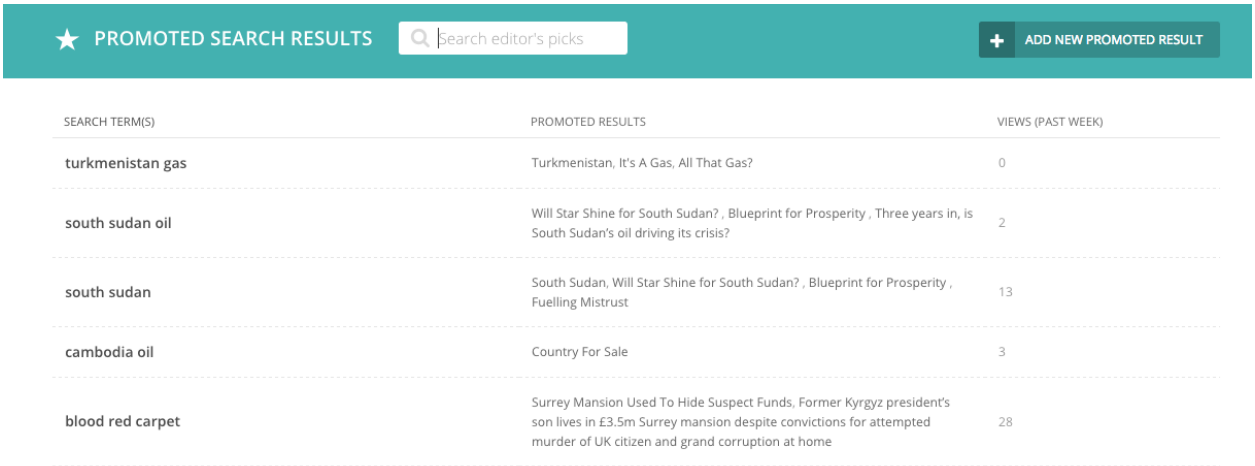
As a concrete example, one of our clients wanted to direct people who searched for 'finances' to their 'Annual budget review' page. The word 'finances' is not mentioned in either the title or the body of the target page. So they created a promoted search result for the word 'finances' that pushed the budget page to the very top of the results.

Note: The promoted result will only work if the user types *exactly* the phrase that you have set it up for. If you have variations of phrase that you want to take into account, then you must create additional promoted results.

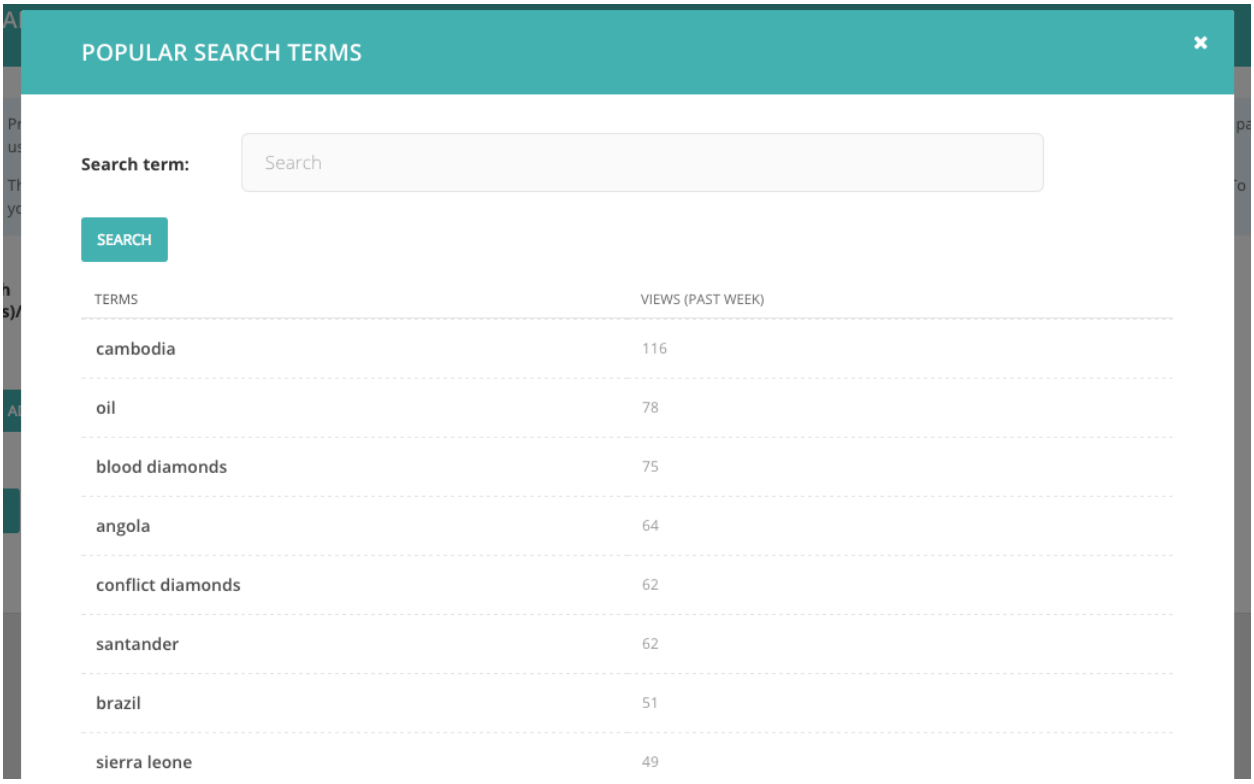
To set up the promoted search results, click on the 'Promoted search results' menu item in the 'Settings' menu.



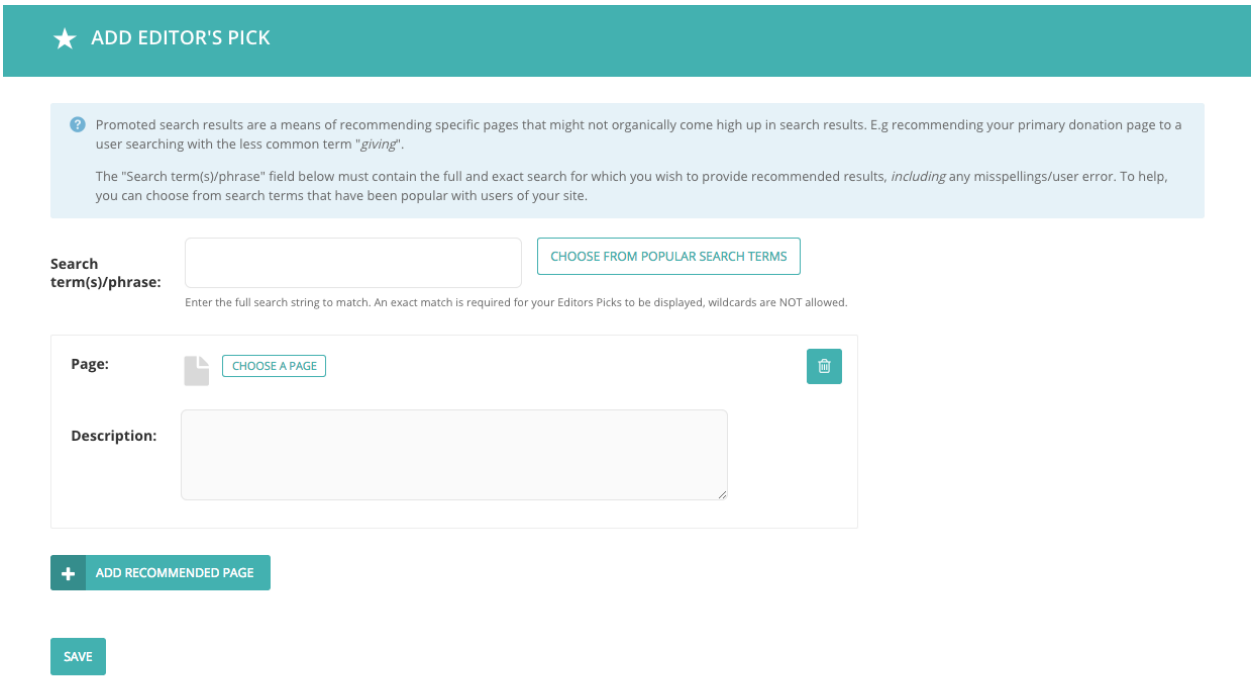
Add a new promoted result from the button in the top right of the resulting screen, or edit an existing promoted result by clicking on it.



When adding a new promoted result, Wagtail provides you with a 'Choose from popular search terms' option. This will show you the most popular terms entered by users into your internal search. You can match this against your existing promoted results to ensure that users are able to find what they are looking for.



You then add a the result itself by clicking 'Add recommended page'. You can add multiple results, but be careful about adding too many, as you may end up hiding all of your organic results with promoted results, which may not be helpful for users who aren't really sure what they are looking for.



Contributing to Wagtail

Issues

The easiest way to contribute to Wagtail is to tell us how to improve it! First, check to see if your bug or feature request has already been submitted at github.com/torchbox/wagtail/issues. If it has, and you have some supporting information which may help us deal with it, comment on the existing issue. If not, please create a new one, providing as much relevant context as possible. For example, if you're experiencing problems with installation, detail your environment and the steps you've already taken. If something isn't displaying correctly, tell us what browser you're using, and include a screenshot if possible.

Pull requests

If you're a Python or Django developer, fork and get stuck in! Send us a useful pull request and we'll post you a t-shirt. We welcome all contributions, whether they solve problems which are specific to you or they address existing issues. If you're stuck for ideas, pick something from the issue list, or email us directly on hello@wagtail.io if you'd like us to suggest something!

Translations

Wagtail has internationalisation support so if you are fluent in a non-English language you can contribute by localising the interface.

Translation work should be submitted through Transifex.

Other contributions

We welcome contributions to all aspects of Wagtail. If you would like to improve the design of the user interface, or extend the documentation, please submit a pull request as above. If you're not familiar with Github or pull requests, contact us directly and we'll work something out.

Development

Setting up a local copy of the Wagtail git repository is slightly more involved than running a release package of Wagtail, as it requires Node.js and NPM for building Javascript and CSS assets. (This is not required when running a release version, as the compiled assets are included in the release package.)

If you're happy to develop on a virtual machine, the vagrant-wagtail-develop setup script is the fastest way to get up and running. This will provide you with a running instance of the Wagtail demo site, with the Wagtail and wagtaildemo codebases available as shared folders for editing on your host machine.

(Build scripts for other platforms would be very much welcomed - if you create one, please let us know via the Wagtail Developers group!)

If you'd prefer to set up all the components manually, read on. These instructions assume that you're familiar with using pip and virtualenv to manage Python packages.

Setting up the Wagtail codebase

Install Node.js, any version between v0.10.x and v0.12.x. Instructions for installing Node.js can be found on the Node.js download page. You will also need to install the **libjpeg** and **zlib** libraries, if you haven't done so already - see Pillow's platform-specific installation instructions.

Clone a copy of the Wagtail codebase:

```
git clone https://github.com/torchbox/wagtail.git
cd wagtail
```

With your preferred virtualenv activated, install the Wagtail package in development mode:

```
python setup.py develop
```

Install the tool chain for building static assets:

```
npm install
```

Compile the assets:

```
npm run build
```

Any Wagtail sites you start up in this virtualenv will now run against this development instance of Wagtail. We recommend using the Wagtail demo site as a basis for developing Wagtail.

Development dependencies

Developing Wagtail requires additional Python modules for testing and documentation.

The list of dependencies is in the Wagtail root directory in `requirements-dev.txt` and can be installed thus, from the Wagtail codebase root directory:

```
pip install -r requirements-dev.txt
```

Testing

From the root of the Wagtail codebase, run the following command to run all the tests:

```
python runtests.py
```

Running only some of the tests

At the time of writing, Wagtail has well over 1000 tests, which takes a while to run. You can run tests for only one part of Wagtail by passing in the path as an argument to `runtests.py`:

```
python runtests.py wagtail.wagtailcore
```

Testing against PostgreSQL

By default, Wagtail tests against SQLite. If you need to test against a different database, set the `DATABASE_ENGINE` environment variable to the name of the Django database backend to test against:

```
DATABASE_ENGINE=django.db.backends.postgresql_psycopg2 python runtests.py
```

This will create a new database called `test_wagtail` in PostgreSQL and run the tests against it.

If you need to use a different user, password or host. Use the `PGUSER`, `PGPASSWORD` and `PGHOST` environment variables.

Testing Elasticsearch

To test Elasticsearch, you need to have the `elasticsearch` package installed.

Once installed, Wagtail will attempt to connect to a local instance of Elasticsearch (`http://localhost:9200`) and use the index `test_wagtail`.

If your Elasticsearch instance is located somewhere else, you can set the `ELASTICSEARCH_URL` environment variable to point to its location:

```
ELASTICSEARCH_URL=http://my-elasticsearch-instance:9200 python runtests.py
```

If you no longer want Wagtail to test against Elasticsearch, uninstall the `elasticsearch` package.

Compiling static assets

All static assets such as JavaScript, CSS, images, and fonts for the Wagtail admin are compiled from their respective sources by `gulp`. The compiled assets are not committed to the repository, and are compiled before packaging each new release. Compiled assets should not be submitted as part of a pull request.

To compile the assets, run:

```
npm run build
```

This must be done after every change to the source files. To watch the source files for changes and then automatically recompile the assets, run:

```
npm start
```

UI Styleguide

Developers working on the Wagtail UI or creating new UI components may wish to test their work against our Styleguide, which is provided as the contrib module "wagtailstyleguide".

To install the styleguide module on your site, add it to the list of `INSTALLED_APPS` in your settings:

```
INSTALLED_APPS = (
    ...
    'wagtail.contrib.wagtailstyleguide',
    ...
)
```

At present the styleguide is static: new UI components must be added to it manually, and there are no hooks into it for other modules to use. We hope to support hooks in the future.

The styleguide doesn't currently provide examples of all the core interface components; notably the Page, Document, Image and Snippet chooser interfaces are not currently represented.

Python coding guidelines

PEP8

We ask that all Python contributions adhere to the PEP8 style guide, apart from the restriction on line length (E501). The pep8 tool makes it easy to check your code, e.g. `pep8 --ignore=E501 your_file.py`.

Python 2 and 3 compatibility

All contributions should support Python 2 and 3 and we recommend using the six compatibility library (use the pip version installed as a dependency, not the version bundled with Django).

Tests

Wagtail has a suite of tests, which we are committed to improving and expanding. See *Testing*.

We run continuous integration at travis-ci.org/torchbox/wagtail to ensure that no commits or pull requests introduce test failures. If your contributions add functionality to Wagtail, please include the additional tests to cover it; if your contributions alter existing functionality, please update the relevant tests accordingly.

CSS coding guidelines

Our CSS is written in Sass, using the SCSS syntax.

Compiling

The SCSS source files are compiled to CSS using the [gulp](http://gulpjs.com/) build system. This requires [node.js](http://nodejs.org) to run. To install the libraries required for compiling the SCSS, run the following from the Wagtail repository root:

```
$ npm install
```

To compile the assets, run:

$ npm run build

Alternatively, the SCSS files can be monitored, automatically recompiling when any changes are observed, by running:

$ npm start

Spacing

- Use soft-tabs with a four space indent. Spaces are the only way to guarantee code renders the same in any person's environment.
- Put spaces after `:` in property declarations.
- Put spaces before `{` in rule declarations.
- Put line breaks between rulesets.
- When grouping selectors, keep individual selectors to a single line.
- Place closing braces of declaration blocks on a new line.
- Each declaration should appear on its own line for more accurate error reporting.
- Add a newline at the end of your `.scss` files.
- Strip trailing whitespace from your rules.

Formatting

- Use hex color codes `#000` unless using `rgba()` in raw CSS (SCSS' `rgba()` function is overloaded to accept hex colors as a param, e.g., `rgba(#000, .5)`).
- Use `//` for comment blocks (instead of `/* */`).
- Use single quotes for string values `background: url('my/image.png')` or `content: 'moose'`
- Avoid specifying units for zero values, e.g., `margin: 0;` instead of `margin: 0px;`.
- Strive to limit use of shorthand declarations to instances where you must explicitly set all the available values.

Sass imports

Leave off underscores and file extensions in includes:

```
// Bad
@import 'components/_widget.scss'

// Better
@import 'components/widget'
```

Pixels vs. ems

Use `rems` for `font-size`, because they offer absolute control over text. Additionally, unit-less `line-height` is preferred because it does not inherit a percentage value of its parent element, but instead is based on a multiplier of the `font-size`.

Specificity (classes vs. ids)

Always use classes instead of IDs in CSS code. IDs are overly specific and lead to duplication of CSS.

When styling a component, start with an element + class namespace, prefer direct descendant selectors by default, and use as little specificity as possible. Here is a good example:

```
<ul class="category-list">
    <li class="item">Category 1</li>
    <li class="item">Category 2</li>
    <li class="item">Category 3</li>
</ul>
```

```
.category-list { // element + class namespace

    // Direct descendant selector > for list items
    > li {
        list-style-type: disc;
    }

    // Minimal specificity for all links
    a {
        color: #f00;
    }
}
```

Class naming conventions

Never reference `js-` prefixed class names from CSS files. `js-` are used exclusively from JS files.

Use the SMACSS `is-` prefix for state rules that are shared between CSS and JS.

Misc

As a rule of thumb, avoid unnecessary nesting in SCSS. At most, aim for three levels. If you cannot help it, step back and rethink your overall strategy (either the specificity needed, or the layout of the nesting).

Examples

Here are some good examples that apply the above guidelines:

```
// Example of good basic formatting practices
.styleguide-format {
    color: #000;
    background-color: rgba(0, 0, 0, .5);
    border: 1px solid #0f0;
}

// Example of individual selectors getting their own lines (for error reporting)
.multiple,
.classes,
.get-new-lines {
    display: block;
```

```
}

// Avoid unnecessary shorthand declarations
.not-so-good {
    margin: 0 0 20px;
}
.good {
    margin-bottom: 20px;
}
```

Vendor prefixes

Line up your vendor prefixes.

```
// Example of good prefix formatting practices
.styleguide-format {
    -webkit-transition: opacity 0.2s ease-out;
       -moz-transition: opacity 0.2s ease-out;
        -ms-transition: opacity 0.2s ease-out;
         -o-transition: opacity 0.2s ease-out;
            transition: opacity 0.2s ease-out;
}
```

Don't write vendor prefixes for `border-radius`, it's pretty well supported.

If you're unsure, you can always check support at caniuse

Linting SCSS

The guidelines are included in a `.scss-lint.yml` file so that you can check that your code conforms to the style guide.

Run the linter with `scss-lint .` from the wagtail project root. You'll need to have the linter installed to do this. You can get it by running:

```
gem install scss-lint
```

JavaScript coding guidelines

Write JavaScript according to the Airbnb Styleguide, with some exceptions:

- Use soft-tabs with a four space indent. Spaces are the only way to guarantee code renders the same in any person's environment.
- We accept `snake_case` in object properties, such as `ajaxResponse.page_title`, however camelCase or UPPER_CASE should be used everywhere else.

Linting and formatting code

Wagtail provides some tooling configuration to help check your code meets the styleguide. You'll need node.js and npm on your development machine. Ensure project dependencies are installed by running `npm install`

Linting code

```
npm run lint:js
```

This will lint all the JS in the wagtail project, excluding vendor files and compiled libraries.

Some of the modals are generated via server-side scripts. These include template tags that upset the linter, so modal workflow JavaScript is excluded from the linter.

Formatting code

```
npm run format:js
```

This will perform safe edits to conform your JS code to the styleguide. It won't touch the line-length, or convert quotemarks from double to single.

Run the linter after you've formatted the code to see what manual fixes you need to make to the codebase.

Changing the linter configuration

Under the hood, the tasks use the JavaScript Code Style library.

To edit the settings for ignored files, or to change the linting rules, edit the `.jscsrc` file in the wagtail project root.

A complete list of the possible linting rules can be found here: JSCS Rules

Release notes

Wagtail 1.2 release notes

- *What's new*
- *Upgrade considerations*

What's new

Site settings module

Wagtail now includes a contrib module (previously available as the wagtailsettings package) to allow administrators to edit site-specific settings.

See: *Site settings*

Jinja2 support

The core templatetags (`pageurl`, `slugurl`, `image`, `richtext` and `wagtailuserbar`) are now compatible with Jinja2 so it's now possible to use Jinja2 as the template engine for your Wagtail site.

Note that the variable name `self` is reserved in Jinja2, and so Wagtail now provides alternative variable names where `self` was previously used: `page` to refer to page objects, and `value` to refer to StreamField blocks. All code examples in this documentation have now been updated to use the new variable names, for compatibility with Jinja2; however, users of the default Django template engine can continue to use `self`.

See: *Jinja2 template support*

Site-specific redirects

You can now create redirects for a particular site using the admin interface.

Search API improvements

Wagtail's image and document models now provide a `search` method on their QuerySets, making it easy to perform searches on filtered data sets. In addition, search methods now accept two new keyword arguments:

- `operator`, to determine whether multiple search terms will be treated as 'or' (any term may match) or 'and' (all terms must match);
- `order_by_relevance`, set to True (the default) to order by relevance or False to preserve the QuerySet's original ordering.

See: *Searching*

`max_num` and `min_num` parameters on inline panels

Inline panels now accept the optional parameters `max_num` and `min_num`, to specify the maximum / minimum number of child items that must exist in order for the page to be valid.

See: *Inline Panels and Model Clusters*

`get_context` on StreamField blocks

StreamField blocks now *provide a get_context method* that can be overridden to pass additional variables to the block's template.

Browsable API

The Wagtail API now incorporates the browsable front-end provided by Django REST Framework. Note that this must be enabled by adding `'rest_framework'` to your project's `INSTALLED_APPS` setting.

Python 3.5 support

Wagtail now supports Python 3.5 when run in conjunction with Django 1.8.6 or later.

Minor features

- WagtailRedirectMiddleware can now ignore the query string if there is no redirect that exactly matches it
- Order of URL parameters now ignored by redirect middleware
- Added SQL Server compatibility to image migration
- Added classnames to Wagtail rich text editor buttons to aid custom styling
- Simplified body_class in default homepage template
- page_published signal now called with the revision object that was published

- Added a favicon to the admin interface, customisable by overriding the `branding_favicon` block (see *Custom branding*).
- Added spinner animations to long-running form submissions
- The EMBEDLY_KEY setting has been renamed to WAGTAILEMBEDS_EMBEDLY_KEY
- StreamField blocks are now added automatically, without showing the block types menu, if only one block type exists (Alex Gleason)
- The `first_published_at` and `latest_revision_created_at` fields on page models are now available as filter fields on search queries
- Wagtail admin now standardises on a single thumbnail image size, to reduce the overhead of creating multiple renditions
- Rich text fields now strip out HTML comments
- Page editor form now sets `enctype="multipart/form-data"` as appropriate, allowing FileField to be used on page models (Petr Vacha)
- Explorer navigation menu on a completely empty page tree now takes you to the root level, rather than doing nothing
- Added animation and fixed display issues when focusing a rich text field (Alex Gleason)
- Added a system check to warn if Pillow is compiled without JPEG / PNG support
- Page chooser now prevents users from selecting the root node where this would be invalid
- New translations for Dutch (Netherlands), Georgian, Swedish and Turkish (Turkey)

Bug fixes

- Page slugs are no longer auto-updated from the page title if the page is already published
- Deleting a page permission from the groups admin UI does not immediately submit the form
- Wagtail userbar is shown on pages that do not pass a `page` variable to the template (e.g. because they override the `serve` method)
- `request.site` now set correctly on page preview when the page is not in the default site
- Project template no longer raises a deprecation warning (Maximilian Stauss)
- `PageManager.sibling_of(page)` and `PageManager.not_sibling_of(page)` now default to inclusive (i.e. `page` is considered a sibling of itself), for consistency with other sibling methods
- The "view live" button displayed after publishing a page now correctly reflects any changes made to the page slug (Ryan Pineo)
- API endpoints now accept and ignore the _ query parameter used by jQuery for cache-busting
- Page slugs are no longer cut off when Unicode characters are expanded into multiple characters (Sævar Öfjörð Magnússon)
- Searching a specific page model while filtering it by either ID or tree position no longer raises an error (Ashia Zawaduk)
- Scrolling an over-long explorer menu no longer causes white background to show through (Alex Gleason)
- Removed jitter when hovering over StreamField blocks (Alex Gleason)
- Non-ASCII email addresses no longer throw errors when generating Gravatar URLs (Denis Voskvitsov, Kyle Stratis)

- Dropdown for `ForeignKey` s are now styled consistently (Ashia Zawaduk)
- Date choosers now appear on top of StreamField menus (Sergey Nikitin)
- Fixed a migration error that was raised when block-updating from 0.8 to 1.1+
- `Page.copy()` no longer breaks on models with a `ClusterTaggableManager` or `ManyToManyField`
- Validation errors when inserting an embed into a rich text area are now reported back to the editor

Upgrade considerations

`PageManager.sibling_of(page)` and `PageManager.not_sibling_of(page)` have changed behaviour

In previous versions of Wagtail, the `sibling_of` and `not_sibling_of` methods behaved inconsistently depending on whether they were called on a manager (e.g. `Page.objects.sibling_of(some_page)` or `EventPage.objects.sibling_of(some_page)`) or a QuerySet (e.g. `Page.objects.all().sibling_of(some_page)` or `EventPage.objects.live().sibling_of(some_page)`).

Previously, the manager methods behaved as *exclusive* by default; that is, they did not count the passed-in page object as a sibling of itself:

```
>>> event_1 = EventPage.objects.get(title='Event 1')
>>> EventPage.objects.sibling_of(event_1)
[<EventPage: Event 2>]  # OLD behaviour: Event 1 is not considered a sibling of itself
```

This has now been changed to be *inclusive* by default; that is, the page is counted as a sibling of itself:

```
>>> event_1 = EventPage.objects.get(title='Event 1')
>>> EventPage.objects.sibling_of(event_1)
[<EventPage: Event 1>, <EventPage: Event 2>]  # NEW behaviour: Event 1 is considered␣
↪a sibling of itself
```

If the call to `sibling_of` or `not_sibling_of` is chained after another QuerySet method - such as `all()`, `filter()` or `live()` - behaviour is unchanged; this behaves as *inclusive*, as it did in previous versions:

```
>>> event_1 = EventPage.objects.get(title='Event 1')
>>> EventPage.objects.all().sibling_of(event_1)
[<EventPage: Event 1>, <EventPage: Event 2>]  # OLD and NEW behaviour
```

If your project includes queries that rely on the old (exclusive) behaviour, this behaviour can be restored by adding the keyword argument `inclusive=False`:

```
>>> event_1 = EventPage.objects.get(title='Event 1')
>>> EventPage.objects.sibling_of(event_1, inclusive=False)
[<EventPage: Event 2>]  # passing inclusive=False restores the OLD behaviour
```

`Image.search` and `Document.search` methods are deprecated

The `Image.search` and `Document.search` methods have been deprecated in favour of the new QuerySet-based search mechanism - see *Searching Images, Documents and custom models*. Code using the old `search` methods should be updated to search on QuerySets instead; for example:

```
Image.search("Hello", filters={'uploaded_by_user': user})
```

can be rewritten as:

```
Image.objects.filter(uploaded_by_user=user).search("Hello")
```

Wagtail API requires adding `rest_framework` to INSTALLED_APPS

If you have the Wagtail API (`wagtail.contrib.wagtailapi`) enabled, you must now add `'rest_framework'` to your project's `INSTALLED_APPS` setting. In the current version the API will continue to function without this app, but the browsable front-end will not be available; this ability will be dropped in a future release.

`Page.get_latest_revision_as_page()` now returns live page object when there are no draft changes

If you have any application code that makes direct updates to page data, at the model or database level, be aware that the way these edits are reflected in the page editor has changed.

Previously, the `get_latest_revision_as_page` method - used by the page editor to return the current page revision for editing - always retrieved data from the page's revision history. Now, it will only do so if the page has unpublished changes (i.e. the page is in `live + draft` state) - pages which have received no draft edits since being published will return the page's live data instead.

As a result, any changes made directly to a live page object will be immediately reflected in the editor without needing to update the latest revision record (but note, the old behaviour is still used for pages in `live + draft` state).

Wagtail 1.1 release notes

- *What's new*
- *Upgrade considerations*

What's new

`specific()` method on PageQuerySet

Usually, an operation that retrieves a queryset of pages (such as `homepage.get_children()`) will return them as basic Page instances, which only include the core page data such as title. The `specific()` method (e.g. `homepage.get_children().specific()`) now allows them to be retrieved as their most specific type, using the minimum number of queries.

"Promoted search results" has moved into its own module

Previously, this was implemented in `wagtailsearch` but now has been moved into a separate module: *`wagtail.contrib.wagtailsearchpromotions`*

Atomic rebuilding of Elasticsearch indexes

The Elasticsearch search backend now accepts an experimental `ATOMIC_REBUILD` flag which ensures that the existing search index continues to be available while the `update_index` task is running. See *ATOMIC_REBUILD*.

The `wagtailapi` module now uses Django REST Framework

The `wagtailapi` module is now built on Django REST Framework and it now also has a library of serialisers that you can use in your own REST Framework based APIs. No user-facing changes have been made.

We hope to support more REST framework features, such as a browsable API, in future releases.

Permissions fixes in the admin interface

A number of inconsistencies around permissions in the admin interface were fixed in this release:

- Removed all permissions for "User profile" (not used)
- Removed "delete" permission for Images and documents (not used)
- Users can now access images and documents when they only have the "change" permission (previously required "add" permission as well)
- Permissions for Users now taken from custom user model, if set (previously always used permissions on Djangos builtin User model)
- Groups and Users now respond consistently to their respective "add", "change" and "delete" permissions

Searchable snippets

Snippets that inherit from `wagtail.wagtailsearch.index.Indexed` are now given a search box on the snippet chooser and listing pages. See *Making Snippets Searchable*.

Minor features

- Implemented deletion of form submissions
- Implemented pagination in the page chooser modal
- Changed INSTALLED_APPS in project template to list apps in precedence order
- The `{% image %}` tag now supports filters on the image variable, e.g. `{% image primary_img|default:secondary_img width-500 %}`
- Moved the style guide menu item into the Settings sub-menu
- Search backends can now be specified by module (e.g. `wagtail.wagtailsearch.backends.elasticsearch`), rather than a specific class (`wagtail.wagtailsearch.backends.elasticsearch.ElasticSearch`)
- Added `descendant_of` filter to the API
- Added optional directory argument to "wagtail start" command
- Non-superusers can now view/edit/delete sites if they have the correct permissions
- Image file size is now stored in the database, to avoid unnecessary filesystem lookups

- Page URL lookups hit the cache/database less often
- Updated URLs within the admin backend to use namespaces
- The `update_index` task now indexes objects in batches of 1000, to indicate progress and avoid excessive memory use
- Added database indexes on PageRevision and Image to improve performance on large sites
- Search in page chooser now uses Wagtail's search framework, to order results by relevance
- `PageChooserPanel` now supports passing a list (or tuple) of accepted page types
- The snippet type parameter of `SnippetChooserPanel` can now be omitted, or passed as a model name string rather than a model class
- Added aliases for the `self` template variable to accommodate Jinja as a templating engine: `page` for pages, `field_panel` for field panels / edit handlers, and `value` for blocks
- Added signposting text to the explorer to steer editors away from creating pages at the root level unless they are setting up new sites
- "Clear choice" and "Edit this page" buttons are no longer shown on the page field of the group page permissions form
- Altered styling of stream controls to be more like all other buttons
- Added ability to mark page models as not available for creation using the flag `is_creatable`; pages that are abstract Django models are automatically made non-creatable
- New translations for Norwegian Bokmål and Icelandic

Bug fixes

- Text areas in the non-default tab of the page editor now resize to the correct height
- Tabs in "insert link" modal in the rich text editor no longer disappear (Tim Heap)
- H2 elements in rich text fields were accidentally given a click() binding when put insite a collapsible multi field panel
- The `wagtailimages` module is now compatible with remote storage backends that do not allow reopening closed files
- Search no longer crashes when auto-indexing a model that doesn't have an `id` field
- The `wagtailfrontendcache` module's HTTP backend has been rewritten to reliably direct requests to the configured cache hostname
- Resizing single pixel images with the "fill" filter no longer raises "ZeroDivisionError" or "tile cannot extend outside image"
- The queryset returned from `search` operations when using the database search backend now correctly preserves additional properties of the original query, such as `prefetch_related` / `select_related`
- Responses from the external image URL generator are correctly marked as streaming and will no longer fail when used with Django's cache middleware
- Page copy now works with pages that use multiple inheritance
- Form builder pages now pick up template variables defined in the `get_context` method
- When copying a page, IDs of child objects within page revision records were not remapped to the new objects; this would cause those objects to be lost from the original page when editing the new one

- Newly added redirects now take effect on all sites, rather than just the site that the Wagtail admin backend was accessed through
- Add user form no longer throws a hard error on validation failure

Upgrade considerations

"Promoted search results" no longer in `wagtailsearch`

This feature has moved into a contrib module so is no longer enabled by default.

To re-enable it, add *`wagtail.contrib.wagtailsearchpromotions`* to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...

    'wagtail.contrib.wagtailsearchpromotions',

    ...
```

If you have references to the `wagtail.wagtailsearch.models.EditorsPick` model in your project, you will need to update these to point to the `wagtail.contrib.wagtailsearchpromotions.models.SearchPromotion` model instead.

If you created your project using the `wagtail start` command with Wagtail 1.0, you will probably have references to this model in the `search/views.py` file.

`is_abstract` flag on page models has been replaced by `is_creatable`

Previous versions of Wagtail provided an undocumented `is_abstract` flag on page models - not to be confused with Django's `abstract` Meta flag - to indicate that it should not be included in the list of available page types for creation. (Typically this would be used on model classes that were designed to be subclassed to create new page types, rather than used directly.) To avoid confusion with Django's distinct concept of abstract models, this has now been replaced by a new flag, `is_creatable`.

If you have used `is_abstract = True` on any of your models, you should now change this to `is_creatable = False`.

It is not necessary to include this flag if the model is abstract in the Django sense (i.e. it has `abstract = True` in the model's `Meta` class), since it would never be valid to create pages of that type.

Wagtail 1.0 release notes

- *What's changed*
- *Upgrade considerations*

What's changed

StreamField - a field type for freeform content

StreamField provides an editing model for freeform content such as blog posts and news stories, allowing diverse content types such as text, images, headings, video and more specialised types such as maps and charts to be mixed in any order. See *Freeform page content using StreamField*.

Wagtail API - A RESTful API for your Wagtail site

When installed, the new Wagtail API module provides a RESTful web API to your Wagtail site. You can use this for accessing your raw field content for your sites pages, images and documents in JSON format. See *Wagtail API*

MySQL support

Wagtail now officially supports MySQL as a database backend.

Django 1.8 support

Wagtail now officially supports running under Django 1.8.

Vanilla project template

The built-in project template is more like the Django built-in one with several Wagtail-specific additions. It includes bare minimum settings and two apps (home and search).

Minor changes

- Dropped Django 1.6 support
- Dropped Python 2.6 and 3.2 support
- Dropped Elasticsearch 0.90.x support
- Removed dependency on `libsass`
- Users without usernames can now be created and edited in the admin interface
- Added new translations for Croatian and Finnish

Core

- The Page model now records the date/time that a page was first published, as the field `first_published_at`
- Increased the maximum length of a page slug from 50 to 255 characters
- Added hooks `register_rich_text_embed_handler` and `register_rich_text_link_handler` for customising link / embed handling within rich text fields
- Page URL paths can now be longer than 255 characters

Admin

UI

- Improvements to the layout of the left-hand menu footer
- Menu items of custom apps are now highlighted when being used
- Added thousands separator for counters on dashboard
- Added contextual links to admin notification messages
- When copying pages, it is now possible to specify a place to copy to
- Added pagination to the snippets listing and chooser
- Page / document / image / snippet choosers now include a link to edit the chosen item
- Plain text fields in the page editor now use auto-expanding text areas
- Added "Add child page" button to admin userbar
- Added update notifications (See: *Wagtail update notifications*)

Page editor

- JavaScript includes in the admin backend have been moved to the HTML header, to accommodate form widgets that render inline scripts that depend on libraries such as jQuery
- The external link chooser in rich text areas now accepts URLs of the form '/some/local/path', to allow linking to non-Wagtail-controlled URLs within the local site
- Bare text entered in rich text areas is now automatically wrapped in a paragraph element

Edit handlers API

- `FieldPanel` now accepts an optional `widget` parameter to override the field's default form widget
- Page model fields without a `FieldPanel` are no longer displayed in the form
- No longer need to specify the base model on `InlinePanel` definitions
- Page classes can specify an edit_handler property to override the default Content / Promote / Settings tabbed interface. See *Customising the tabbed interface*.

Other admin changes

- SCSS files in wagtailadmin now use absolute imports, to permit overriding by user stylesheets
- Removed the dependency on `LOGIN_URL` and `LOGIN_REDIRECT_URL` settings
- Password reset view names namespaced to wagtailadmin
- Removed the need to add permission check on admin views (now automated)
- Reversing `django.contrib.auth.admin.login` will no longer lead to Wagtails login view (making it easier to have frontend login views)
- Added cache-control headers to all admin views. This allows Varnish/Squid/CDN to run on vanilla settings in front of a Wagtail site
- Date / time pickers now consistently use times without seconds, to prevent JavasSript behaviour glitches when focusing / unfocusing fields
- Added hook `construct_homepage_summary_items` for customising the site summary panel on the admin homepage
- Renamed the `construct_wagtail_edit_bird` hook to `construct_wagtail_userbar`

- 'static' template tags are now used throughout the admin templates, in place of `STATIC_URL`

Docs

- Support for `django-sendfile` added
- Documents now served with correct mime-type
- Support for `If-Modified-Since` HTTP header

Search

- Search view accepts "page" GET parameter in line with pagination
- Added *AUTO_UPDATE* flag to search backend settings to enable/disable automatically updating the search index on model changes

Routable pages

- Added a new decorator-based syntax for RoutablePage, compatible with Django 1.8

Bug fixes

- The `document_served` signal now correctly passes the Document class as `sender` and the document as `instance`
- Image edit page no longer throws `OSError` when the original image is missing
- Collapsible blocks stay open on any form error
- Document upload modal no longer switches tabs on form errors
- `with_metaclass` is now imported from Django's bundled copy of the `six` library, to avoid errors on Mac OS X from an outdated system copy of the library being imported

Upgrade considerations

Support for older Django/Python/Elasticsearch versions dropped

This release drops support for Django 1.6, Python 2.6/3.2 and Elasticsearch 0.90.x. Please make sure these are updated before upgrading.

If you are upgrading from Elasticsearch 0.90.x, you may also need to update the `elasticsearch` pip package to a version greater than `1.0` as well.

Wagtail version upgrade notifications are enabled by default

Starting from Wagtail 1.0, the admin dashboard will (for admin users only) perform a check to see if newer releases are available. This also provides the Wagtail team with the hostname of your Wagtail site. If you'd rather not receive update notifications, or if you'd like your site to remain unknown, you can disable it by adding this line to your settings file:

```
WAGTAIL_ENABLE_UPDATE_CHECK = False
```

`InlinePanel` definitions no longer need to specify the base model

In previous versions of Wagtail, inline child blocks on a page or snippet were defined using a declaration like:

```
InlinePanel(HomePage, 'carousel_items', label="Carousel items")
```

It is no longer necessary to pass the base model as a parameter, so this declaration should be changed to:

```
InlinePanel('carousel_items', label="Carousel items")
```

The old format is now deprecated; all existing `InlinePanel` declarations should be updated to the new format.

Custom image models should now set the `admin_form_fields` attribute

Django 1.8 now requires that all the fields in a `ModelForm` must be defined in its `Meta.fields` attribute.

As Wagtail uses Django's `ModelForm` for creating image model forms, we've added a new attribute called `admin_form_fields` that should be set to a `tuple` of field names on the image model.

See *Custom image model* for an example.

You no longer need `LOGIN_URL` and `LOGIN_REDIRECT_URL` to point to Wagtail admin.

If you are upgrading from an older version of Wagtail, you probably want to remove these from your project settings.

Prevously, these two settings needed to be set to `wagtailadmin_login` and `wagtailadmin_dashboard` respectively or Wagtail would become very tricky to log in to. This is no longer the case and Wagtail should work fine without them.

`RoutablePage` now uses decorator syntax for defining views

In previous versions of Wagtail, page types that used the *`RoutablePageMixin`* had endpoints configured by setting their `subpage_urls` attribute to a list of urls with view names. This will not work on Django 1.8 as view names can no longer be passed into a url (see: https://docs.djangoproject.com/en/1.8/releases/1.8/#django-conf-urls-patterns).

Wagtail 1.0 introduces a new syntax where each view function is annotated with a `@route` decorator - see *RoutablePageMixin*.

The old `subpage_urls` convention will continue to work on Django versions prior to 1.8, but this is now deprecated; all existing `RoutablePage` definitions should be updated to the decorator-based convention.

Upgrading from the external `wagtailapi` module.

If you were previously using the external `wagtailapi` module (which has now become `wagtail.contrib.wagtailapi`). Please be aware of the following backwards-incompatible changes:

1. Representation of foreign keys has changed

Foreign keys were previously represented by just the value of their primary key. For example:

```
"feed_image": 1
```

This has now been changed to add some `meta` information:

```
"feed_image": {
    "id": 1,
    "meta": {
        "type": "wagtailimages.Image",
        "detail_url": "http://api.example.com/api/v1/images/1/"
    }
}
```

2. On the page detail view, the "parent" field has been moved out of meta

Previously, there was a "parent" field in the "meta" section on the page detail view:

```
{
    "id": 10,
    "meta": {
         "type": "demo.BlogPage",
         "parent": 2
    },

    ...
}
```

This has now been moved to the top level. Also, the above change to how foreign keys are represented applies to this field too:

```
{
    "id": 10,
    "meta": {
         "type": "demo.BlogPage"
    },
    "parent": {
         "id": 2,
         "meta": {
             "type": "demo.BlogIndexPage"
         }
    }

    ...
}
```

Celery no longer automatically used for sending notification emails

Previously, Wagtail would try to use Celery whenever the `djcelery` module was installed, even if Celery wasn't actually set up. This could cause a very hard to track down problem where notification emails would not be sent so this functionality has now been removed.

If you would like to keep using Celery for sending notification emails, have a look at: django-celery-email

Login/Password reset views renamed

It was previously possible to reverse the Wagtail login view using `django.contrib.auth.views.login`. This is no longer possible. Update any references to `wagtailadmin_login`.

Password reset view name has changed from `password_reset` to `wagtailadmin_password_reset`.

JavaScript includes in admin backend have been moved

To improve compatibility with third-party form widgets, pages within the Wagtail admin backend now output their JavaScript includes in the HTML header, rather than at the end of the page. If your project extends the admin backend (through the `register_admin_menu_item` hook, for example) you will need to ensure that all associated JavaScript code runs correctly from the new location. In particular, any code that accesses HTML elements will need to be contained in an 'onload' handler (e.g. jQuery's `$(document).ready()`).

EditHandler internal API has changed

While it is not an official Wagtail API, it has been possible for Wagtail site implementers to define their own `EditHandler` subclasses for use in panel definitions, to customise the behaviour of the page / snippet editing forms. If you have made use of this facility, you will need to update your custom EditHandlers, as this mechanism has been refactored (to allow EditHandler classes to keep a persistent reference to their corresponding model). If you have only used Wagtail's built-in panel types (`FieldPanel`, `InlinePanel`, `PageChooserPanel` and so on), you are unaffected by this change.

Previously, functions like `FieldPanel` acted as 'factory' functions, where a call such as `FieldPanel('title')` constructed and returned an `EditHandler` subclass tailored to work on a 'title' field. These functions now return an object with a `bind_to_model` method instead; the EditHandler subclass can be obtained by calling this with the model class as a parameter. As a guide to updating your custom EditHandler code, you may wish to refer to the relevant change to the Wagtail codebase.

chooser_panel templates are obsolete

If you have added your own custom admin views to the Wagtail admin (e.g. through the `register_admin_urls` hook), you may have used one of the following template includes to incorporate a chooser element for pages, documents, images or snippets into your forms:

- `wagtailadmin/edit_handlers/chooser_panel.html`
- `wagtailadmin/edit_handlers/page_chooser_panel.html`
- `wagtaildocs/edit_handlers/document_chooser_panel.html`
- `wagtailimages/edit_handlers/image_chooser_panel.html`
- `wagtailsnippets/edit_handlers/snippet_chooser_panel.html`

All of these templates are now deprecated. Wagtail now provides a set of Django form widgets for this purpose - `AdminPageChooser`, `AdminDocumentChooser`, `AdminImageChooser` and `AdminSnippetChooser` - which can be used in place of the `HiddenInput` widget that these form fields were previously using. The field can then be rendered using the regular `wagtailadmin/shared/field.html` or `wagtailadmin/shared/field_as_li.html` template.

`document_served` signal arguments have changed

Previously, the `document_served` signal (which is fired whenever a user downloads a document) passed the document instance as the `sender`. This has now been changed to correspond the behaviour of Django's built-in signals; `sender` is now the `Document` class, and the document instance is passed as the argument `instance`. Any existing signal listeners that expect to receive the document instance in `sender` must now be updated to check the `instance` argument instead.

Custom image models must specify an `admin_form_fields` list

Previously, the forms for creating and editing images followed Django's default behaviour of showing all fields defined on the model; this would include any custom fields specific to your project that you defined by subclassing `AbstractImage` and setting `WAGTAILIMAGES_IMAGE_MODEL`. This behaviour is risky as it may lead to fields being unintentionally exposed to the user, and so Django has deprecated this, for removal in Django 1.8. Accordingly, if you create your own custom subclass of `AbstractImage`, you must now provide an `admin_form_fields` property, listing the fields that should appear on the image creation / editing form - for example:

```
from wagtail.wagtailimages.models import AbstractImage, Image

class MyImage(AbstractImage):
    photographer = models.CharField(max_length=255)
    has_legal_approval = models.BooleanField()

    admin_form_fields = Image.admin_form_fields + ['photographer']
```

`construct_wagtail_edit_bird` hook has been renamed

Previously you could customize the Wagtail userbar using the `construct_wagtail_edit_bird` hook. The hook has been renamed to `construct_wagtail_userbar`.

The old hook is now deprecated; all existing `construct_wagtail_edit_bird` declarations should be updated to the new hook.

`IMAGE_COMPRESSION_QUALITY` setting has been renamed

The `IMAGE_COMPRESSION_QUALITY` setting, which determines the quality of saved JPEG images as a value from 1 to 100, has been renamed to `WAGTAILIMAGES_JPEG_QUALITY`. If you have used this setting, please update your settings file accordingly.

Wagtail 0.8.10 release notes

- *What's changed*

What's changed

Bug fixes

- When copying a page, IDs of child objects within page revision records were not remapped to the new objects; this would cause those objects to be lost from the original page when editing the new one
- Search no longer crashes when auto-indexing a model that doesn't have an id field (Scot Hacker)
- Resizing single pixel images with the "fill" filter no longer raises "ZeroDivisionError" or "tile cannot extend outside image"

Wagtail 0.8.8 release notes

- *What's changed*

What's changed

Bug fixes

- Form builder no longer raises a `TypeError` when submitting unchecked boolean field
- Image upload form no longer breaks when using i10n thousand separators
- Multiple image uploader now escapes HTML in filenames
- Retrieving an individual item from a sliced `BaseSearchResults` object now properly takes the slice offset into account
- Removed dependency on unicodecsv which fixes a crash on Python 3
- Submitting unicode text in form builder form no longer crashes with `UnicodeEncodeError` on Python 2
- Creating a proxy model from a Page class no longer crashes in the system check
- Unrecognised embed URLs passed to the `|embed` filter no longer cause the whole page to crash with an `EmbedNotFoundException`
- Underscores no longer get stripped from page slugs

Wagtail 0.8.7 release notes

- *What's changed*

What's changed

Bug fixes

- `wagtailfrontendcache` no longer tries to purge pages that are not in a site
- The contents of `<div>` elements in the rich text editor were not being whitelisted

- Due to the above issue, embeds/images in a rich text field would sometimes be saved into the database in their editor representation
- RoutablePage now prevents `subpage_urls` from being defined as a property, which would cause a memory leak
- Added validation to prevent pages being created with only whitespace characters in their title fields
- Users are no longer logged out on changing password when SessionAuthenticationMiddleware (added in Django 1.7) is in use
- Added a workaround for a Python / Django issue that prevented documents with certain non-ASCII filenames from being served

Wagtail 0.8.6 release notes

- *What's new*
- *Upgrade considerations*

What's new

Minor features

- Translations updated, including new translations for Czech, Italian and Japanese
- The "fixtree" command can now delete orphaned pages

Bug fixes

- django-taggit library updated to 0.12.3, to fix a bug with migrations on SQLite on Django 1.7.2 and above (https://github.com/alex/django-taggit/issues/285)
- Fixed a bug that caused children of a deleted page to not be deleted if they had a different type

Upgrade considerations

Orphaned pages may need deleting

This release fixes a bug with page deletion introduced in 0.8, where deleting a page with child pages will result in those child pages being left behind in the database (unless the child pages are of the same type as the parent). This may cause errors later on when creating new pages in the same position. To identify and delete these orphaned pages, it is recommended that you run the following command (from the project root) after upgrading to 0.8.6:

```
./manage.py fixtree
```

This will output a list of any orphaned pages found, and request confirmation before deleting them.

Since this now makes `fixtree` an interactive command, a `./manage.py fixtree --noinput` option has been added to restore the previous non-interactive behaviour. With this option enabled, deleting orphaned pages is always skipped.

Wagtail 0.8.5 release notes

- *What's new*

What's new

Bug fixes

- On adding a new page, the available page types are ordered by the displayed verbose name
- Active admin submenus were not properly closed when activating another
- `get_sitemap_urls` is now called on the specific page class so it can now be overridden
- (Firefox and IE) Fixed preview window hanging and not refocusing when "Preview" button is clicked again
- Storage backends that return raw ContentFile objects are now handled correctly when resizing images
- Punctuation characters are no longer stripped when performing search queries
- When adding tags where there were none before, it is now possible to save a single tag with multiple words in it
- `richtext` template tag no longer raises `TypeError` if `None` is passed into it
- Serving documents now uses a streaming HTTP response and will no longer break Django's cache middleware
- User admin area no longer fails in the presence of negative user IDs (as used by `django-guardian`'s default settings)
- Password reset emails now use the `BASE_URL` setting for the reset URL
- `BASE_URL` is now included in the project template's default settings file

Wagtail 0.8.4 release notes

- *What's new*

What's new

Bug fixes

- It is no longer possible to have the explorer and settings menu open at the same time
- Page IDs in page revisions were not updated on page copy, causing subsequent edits to be committed to the original page instead
- Copying a page now creates a new page revision, ensuring that changes to the title/slug are correctly reflected in the editor (and also ensuring that the user performing the copy is logged)
- Prevent a race condition when creating Filter objects
- On adding a new page, the available page types are ordered by the displayed verbose name

Wagtail 0.8.3 release notes

- *What's new*
- *Upgrade considerations*

What's new

Bug fixes

- Added missing jQuery UI sprite files, causing collectstatic to throw errors (most reported on Heroku)
- Page system check for on_delete actions of ForeignKeys was throwing false positives when page class decends from an abstract class (Alejandro Giacometti)
- Page system check for on_delete actions of ForeignKeys now only raises warnings, not errors
- Fixed a regression where form builder submissions containing a number field would fail with a JSON serialisation error
- Resizing an image with a focal point equal to the image size would result in a divide-by-zero error
- Focal point indicator would sometimes be positioned incorrectly for small or thin images
- Fix: Focal point chooser background colour changed to grey to make working with transparent images easier
- Elasticsearch configuration now supports specifying HTTP authentication parameters as part of the URL, and defaults to ports 80 (HTTP) and 443 (HTTPS) if port number not specified
- Fixed a TypeError when previewing pages that use RoutablePageMixin
- Rendering image with missing file in rich text no longer crashes the entire page
- IOErrors thrown by underlying image libraries that are not reporting a missing image file are no longer caught
- Fix: Minimum Pillow version bumped to 2.6.1 to work around a crash when using images with transparency
- Fix: Images with transparency are now handled better when being used in feature detection

Upgrade considerations

Port number must be specified when running Elasticsearch on port 9200

In previous versions, an Elasticsearch connection URL in `WAGTAILSEARCH_BACKENDS` without an explicit port number (e.g. `http://localhost/`) would be treated as port 9200 (the Elasticsearch default) whereas the correct behaviour would be to use the default http/https port of 80/443. This behaviour has now been fixed, so sites running Elasticsearch on port 9200 must now specify this explicitly - e.g. `http://localhost:9200`. (Projects using the default settings, or the settings given in the Wagtail documentation, are unaffected.)

Wagtail 0.8.1 release notes

- *What's new*

What's new

Bug fixes

- Fixed a regression where images would fail to save when feature detection is active

Wagtail 0.8 release notes

- *What's new*
- *Upgrade considerations*

What's new

Minor features

- Page operations (creation, publishing, copying etc) are now logged via Python's `logging` framework; to configure this, add a logger entry for `'wagtail'` or `'wagtail.core'` to the `LOGGING` setup in your settings file.
- The save button on the page edit page now redirects the user back to the edit page instead of the explorer
- Signal handlers for `wagtail.wagtailsearch` and `wagtail.contrib.wagtailfrontendcache` are now automatically registered when using Django 1.7 or above.
- Added a Django 1.7 system check to ensure that foreign keys from Page models are set to `on_delete=SET_NULL`, to prevent inadvertent (and tree-breaking) page deletions
- Improved error reporting on image upload, including ability to set a maximum file size via a new setting `WAGTAILIMAGES_MAX_UPLOAD_SIZE`
- The external image URL generator now keeps persistent image renditions, rather than regenerating them on each request, so it no longer requires a front-end cache.
- Added Dutch translation

Bug fixes

- Replaced references of .username with .get_username() on users for better custom user model support
- Unpinned dependency versions for six and requests to help prevent dependency conflicts
- Fixed TypeError when getting embed HTML with oembed on Python 3
- Made HTML whitelisting in rich text fields more robust at catching disallowed URL schemes such as `jav\tascript:`
- `created_at` timestamps on page revisions were not being preserved on page copy, causing revisions to get out of sequence
- When copying pages recursively, revisions of sub-pages were being copied regardless of the `copy_revisions` flag
- Updated the migration dependencies within the project template to ensure that Wagtail's own migrations consistently apply first

- The cache of site root paths is now cleared when a site is deleted
- Search indexing now prevents pages from being indexed multiple times, as both the base Page model and the specific subclass
- Search indexing now avoids trying to index abstract models
- Fixed references to "username" in login form help text for better custom user model support
- Later items in a model's search_field list now consistently override earlier items, allowing subclasses to redefine rules from the parent
- Image uploader now accepts JPEG images that PIL reports as being in MPO format
- Multiple checkbox fields on form-builder forms did not correctly save multiple values
- Editing a page's slug and saving it without publishing could sometimes cause the URL paths of child pages to be corrupted
- `latest_revision_created_at` was being cleared on page publish, causing the page to drop to the bottom of explorer listings
- Searches on partial_match fields were wrongly applying prefix analysis to the search query as well as the document (causing e.g. a query for "water" to match against "wagtail")

Upgrade considerations

Corrupted URL paths may need fixing

This release fixes a bug in Wagtail 0.7 where editing a parent page's slug could cause the URL paths of child pages to become corrupted. To ensure that your database does not contain any corrupted URL paths, it is recommended that you run `./manage.py set_url_paths` after upgrading.

Automatic registration of signal handlers (Django 1.7+)

Signal handlers for the `wagtailsearch` core app and `wagtailfrontendcache` contrib app are automatically registered when using Django 1.7. Calls to `register_signal_handlers` from your `urls.py` can be removed.

Change to search API when using database backend

When using the database backend, calling search (either through `Page.objects.search()` or on the backend directly) will now return a `SearchResults` object rather than a Django `QuerySet` to make the database backend work more like the Elasticsearch backend.

This change shouldn't affect most people as `SearchResults` behaves very similarly to `QuerySet`. But it may cause issues if you are calling `QuerySet` specific methods after calling `.search()`. Eg: `Page.objects.search("Hello").filter(foo="Bar")` (in this case, `.filter()` should be moved before `.search()` and it would work as before).

Removal of validate_image_format from custom image model migrations (Django 1.7+)

If your project is running on Django 1.7, and you have defined a custom image model (by extending the `wagtailimages.AbstractImage` class), the migration that creates this model will probably have a reference to `wagtail.wagtailimages.utils.validators.validate_image_format`. This module has now

been removed, which will cause `manage.py migrate` to fail with an `ImportError` (even if the migration has already been applied). You will need to edit the migration file to remove the line:

```
import wagtail.wagtailimages.utils.validators
```

and the `validators` attribute of the 'file' field - that is, the line:

```
('file', models.ImageField(upload_to=wagtail.wagtailimages.models.get_upload_to,
    width_field='width', height_field='height',
    validators=[wagtail.wagtailimages.utils.validators.validate_image_format],
    verbose_name='File')),
```

should become:

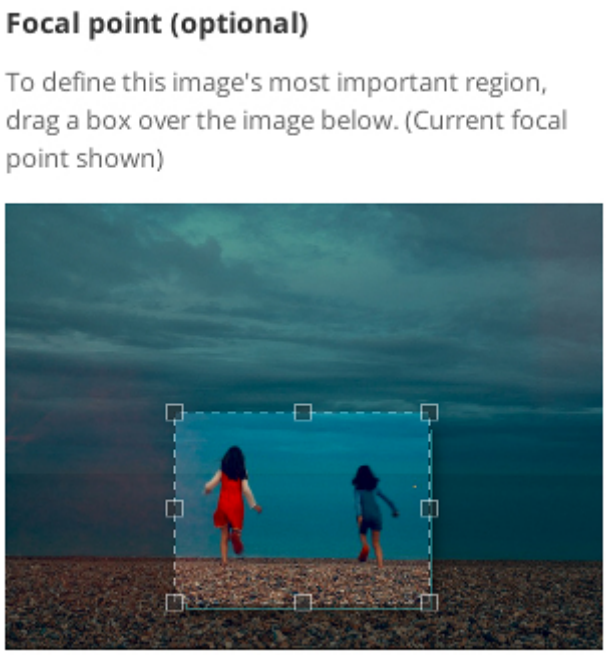
```
('file', models.ImageField(upload_to=wagtail.wagtailimages.models.get_upload_to,
    width_field='width', height_field='height', verbose_name='File')),
```

Wagtail 0.7 release notes

- *What's new*
- *Upgrade considerations*

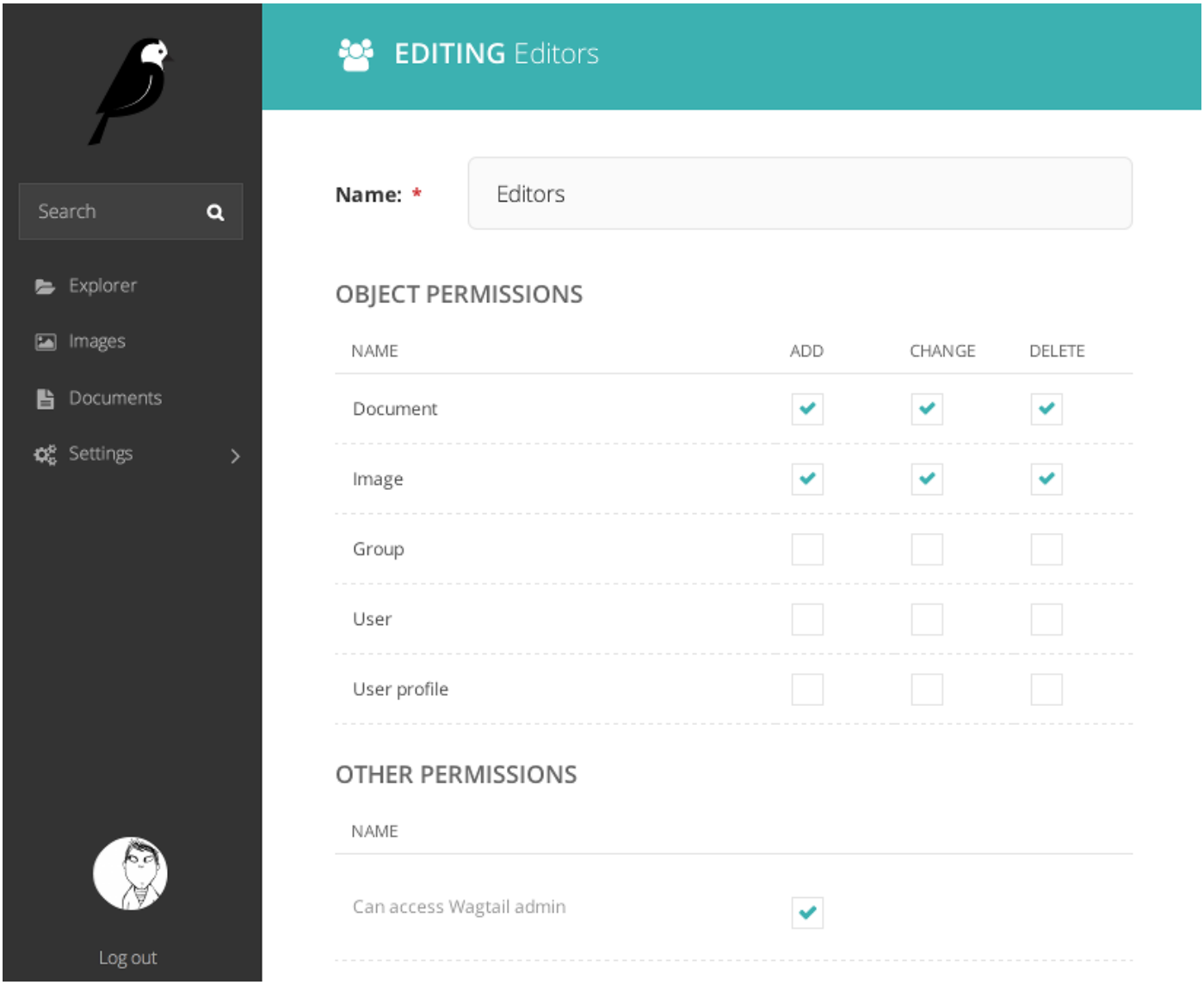
What's new

New interface for choosing image focal point



When editing images, users can now specify a 'focal point' region that cropped versions of the image will be centred on. Previously the focal point could only be set automatically, through image feature detection.

Groups and Sites administration interfaces



The main navigation menu has been reorganised, placing site configuration options in a 'Settings' submenu. This includes two new items, which were previously only available through the Django admin backend: 'Groups', for setting up user groups with a specific set of permissions, and 'Sites', for managing the list of sites served by this Wagtail instance.

Page locking

Moderators and administrators now have the ability to lock a page, preventing further edits from being made to that page until it is unlocked again.



Minor features

- The `content_type` template filter has been removed from the project template, as the same thing can be accomplished with `self.get_verbose_name|slugify`.
- Page copy operations now also copy the page revision history.
- Page models now support a `parent_page_types` property in addition to `subpage types`, to restrict the types of page they can be created under.
- `register_snippet` can now be invoked as a decorator.
- The project template (used when running `wagtail start`) has been updated to Django 1.7.
- The 'boost' applied to the title field on searches has been reduced from 100 to 2.
- The `type` method of `PageQuerySet` (used to filter the queryset to a specific page type) now includes subclasses of the given page type.
- The `update_index` management command now updates all backends listed in `WAGTAILSEARCH_BACKENDS`, or a specific one passed on the command line, rather than just the default backend.
- The 'fill' image resize method now supports an additional parameter defining the closeness of the crop. See *Using images in templates*
- Added support for invalidating Cloudflare caches. See *Frontend cache invalidator*
- Pages in the explorer can now be ordered by last updated time.

Bug fixes

- The 'wagtail start' command now works on Windows and other environments where the `django-admin.py` executable is not readily accessible.
- The external image URL generator no longer stores generated images in Django's cache; this was an unintentional side-effect of setting cache control headers.
- The Elasticsearch backend can now search querysets that have been filtered with an 'in' clause of a non-list type (such as a `ValuesListQuerySet`).
- Logic around the `has_unpublished_changes` flag has been fixed, to prevent issues with the 'View draft' button failing to show in some cases.
- It is now easier to move pages to the beginning and end of their section
- Image rendering no longer creates erroneous duplicate Rendition records when the focal point is blank.

Upgrade considerations

Addition of `wagtailsites` app

The Sites administration interface is contained within a new app, `wagtailsites`. To enable this on an existing Wagtail project, add the line:

```
'wagtail.wagtailsites',
```

to the `INSTALLED_APPS` list in your project's settings file.

Title boost on search reduced to 2

Wagtail's search interface applies a 'boost' value to give extra weighting to matches on the title field. The original boost value of 100 was found to be excessive, and in Wagtail 0.7 this has been reduced to 2. If you have used comparable boost values on other fields, to give them similar weighting to title, you may now wish to reduce these accordingly. See *Indexing*.

Addition of `locked` field to Page model

The page locking mechanism adds a `locked` field to wagtailcore.Page, defaulting to False. Any application code working with Page objects should be unaffected, but any code that creates page records using direct SQL, or within existing South migrations using South's frozen ORM, will fail as this code will be unaware of the new database column. To fix a South migration that fails in this way, add the following line to the `'wagtailcore.page'` entry at the bottom of the migration file:

```
'locked': ('django.db.models.fields.BooleanField', [], {'default': 'False'}),
```

Update to `focal_point_key` field on custom Rendition models

The `focal_point_key` field on wagtailimages.Rendition has been changed to `null=False`, to fix an issue with duplicate renditions being created. If you have defined a custom Rendition model in your project (by extending the `wagtailimages.AbstractRendition` class), you will need to apply a migration to make the corresponding change on your custom model. Unfortunately neither South nor Django 1.7's migration system are able to generate this automatically - you will need to customise the migration produced by `./manage.py schemamigration` / `./manage.py makemigrations`, using the wagtailimages migration as a guide:

- https://github.com/torchbox/wagtail/blob/master/wagtail/wagtailimages/south_migrations/0004_auto__chg_field_rendition_focal_point_key.py (for South / Django 1.6)
- https://github.com/torchbox/wagtail/blob/master/wagtail/wagtailimages/migrations/0004_make_focal_point_key_not_nullable.py (for Django 1.7)

Wagtail 0.6 release notes

- *What's new*
- *Upgrade considerations*
- *Deprecated features*

What's new

Project template and start project command

Wagtail now has a basic project template built in to make starting new projects much easier.

To use it, install `wagtail` onto your machine and run `wagtail start project_name`.

Django 1.7 support

Wagtail can now be used with Django 1.7.

Minor features

- A new template tag has been added for reversing URLs inside routable pages. See *The routablepageurl template tag*.
- RoutablePage can now be used as a mixin. See *wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin*.
- MenuItems can now have bundled JavaScript
- Added the `register_admin_menu_item` hook for registering menu items at startup. See *Hooks*
- Added a version indicator into the admin interface (hover over the wagtail to see it)
- Added Russian translation

Bug fixes

- Page URL generation now returns correct URLs for sites that have the main 'serve' view rooted somewhere other than '/'.
- Search results in the page chooser now respect the page_type parameter on PageChooserPanel.
- Rendition filenames are now prevented from going over 60 chars, even with a large focal_point_key.
- Child relations that are defined on a model's superclass (such as the base Page model) are now picked up correctly by the page editing form, page copy operations and the replace_text management command.
- Tags on images and documents are now committed to the search index immediately on saving.

Upgrade considerations

All features deprecated in 0.4 have been removed

See: *Deprecated features*

Search signal handlers have been moved

If you have an import in your `urls.py` file like `from wagtail.wagtailsearch import register_signal_handlers`, this must now be changed to `from wagtail.wagtailsearch.signal_handlers import register_signal_handlers`

Deprecated features

- The `wagtail.wagtailsearch.indexed` module has been renamed to `wagtail.wagtailsearch.index`

Wagtail 0.5 release notes

- *What's new*
- *Upgrade considerations*

What's new

Multiple image uploader

The image uploader UI has been improved to allow multiple images to be uploaded at once.

Image feature detection

Wagtail can now apply face and feature detection on images using OpenCV, and use this to intelligently crop images.

Feature Detection

Using images outside Wagtail

In normal use, Wagtail will generate resized versions of images at the point that they are referenced on a template, which means that those images are not easily accessible for use outside of Wagtail, such as displaying them on external sites. Wagtail now provides a way to obtain URLs to your images, at any size.

Using images outside Wagtail

RoutablePage

A `RoutablePage` model has been added to allow embedding Django-style URL routing within a page.

RoutablePageMixin

Usage stats for images, documents and snippets

It's now easier to find where a particular image, document or snippet is being used on your site.

Set the `WAGTAIL_USAGE_COUNT_ENABLED` setting to `True` and an icon will appear on the edit page showing you which pages they have been used on.

Copy Page action

The explorer interface now offers the ability to copy pages, with or without subpages.

Minor features

Core

- Hooks can now be defined using decorator syntax:

```
@hooks.register('construct_main_menu')
def construct_main_menu(request, menu_items):
    menu_items.append(
        MenuItem('Kittens!', '/kittens/', classnames='icon icon-
↪folder-inverse', order=1000)
    )
```

- The lxml library (used for whitelisting and rewriting of rich text fields) has been replaced with the pure-python html5lib library, to simplify installation.
- A `page_unpublished` signal has been added.

Admin

- Explorer nav now rendered separately and fetched with AJAX when needed.

  This improves the general performance of the admin interface for large sites.

Bug fixes

- Updates to tag fields are now properly committed to the database when publishing directly from the page edit interface.

Upgrade considerations

Urlconf entries for `/admin/images/`, `/admin/embeds/` etc need to be removed

If you created a Wagtail project prior to the release of Wagtail 0.3, it is likely to contain the following entries in its `urls.py`:

```
# TODO: some way of getting wagtailimages to register itself within␣
↪wagtailadmin so that we
# don't have to define it separately here
url(r'^admin/images/', include(wagtailimages_urls)),
url(r'^admin/embeds/', include(wagtailembeds_urls)),
url(r'^admin/documents/', include(wagtaildocs_admin_urls)),
url(r'^admin/snippets/', include(wagtailsnippets_urls)),
url(r'^admin/search/', include(wagtailsearch_admin_urls)),
url(r'^admin/users/', include(wagtailusers_urls)),
url(r'^admin/redirects/', include(wagtailredirects_urls)),
```

These entries (and the corresponding `from wagtail.wagtail* import ...` lines) need to be removed from `urls.py`. (The entry for `/admin/` should be left in, however.)

Since Wagtail 0.3, the wagtailadmin module automatically takes care of registering these URL subpaths, so these entries are redundant, and these urlconf modules are not guaranteed to remain stable and backwards-compatible in future. Leaving these entries in place will now cause an `ImproperlyConfigured` exception to be thrown.

New fields on Image and Rendition models

Several new fields have been added to the Image and Rendition models to support *Feature Detection*. These will be added to the database when you run `./manage.py migrate`. If you have defined a custom image model (by extending the `wagtailimages.AbstractImage` and `wagtailimages.AbstractRendition` classes and specifying `WAGTAILIMAGES_IMAGE_MODEL` in settings), the change needs to be applied to that model's database table too. Running the command:

```
./manage.py schemamigration myapp --auto add_image_focal_point_fields
```

(with 'myapp' replaced with your app name) will generate the necessary migration file.

South upgraded to 1.0

In preparation for Django 1.7 support in a future release, Wagtail now depends on South 1.0, and its migration files have been moved from `migrations` to `south_migrations`. Older versions of South will fail to find the migrations in the new location.

If your project's requirements file (most commonly requirements.txt or requirements/base.txt) references a specific older version of South, this must be updated to South 1.0.

Wagtail 0.4.1 release notes

Bug fixes

- ElasticSearch backend now respects the backward-compatible URLS configuration setting, in addition to HOSTS
- Documentation fixes

Wagtail 0.4 release notes

- *What's new*
- *Backwards-incompatible changes*
- *Deprecated features*

What's new

Private Pages

Wagtail now supports password protecting pages on the frontend, allowing sections of your website to be made private.

Private pages

Python 3 support

Wagtail now supports Python 3.2, 3.3 and 3.4.

Scheduled publishing

Editors can now schedule pages to be published or unpublished at specified times.

A new management command has been added (*publish_scheduled_pages*) to publish pages that have been scheduled by an editor.

Search on QuerySet with Elasticsearch

It's now possible to perform searches with Elasticsearch on `PageQuerySet` objects:

```
>>> from wagtail.wagtailcore.models import Page
>>> Page.objects.live().descendant_of(events_index).search("Hello")
[<Page: Event 1>, <Page: Event 2>]
```

Sitemap generation

A new module has been added (`wagtail.contrib.wagtailsitemaps`) which produces XML sitemaps for Wagtail sites.

Sitemap generator

Front-end cache invalidation

A new module has been added (`wagtail.contrib.wagtailfrontendcache`) which invalidates pages in a frontend cache when they are updated or deleted in Wagtail.

Frontend cache invalidator

Notification preferences

Users can now decide which notifications they receive from Wagtail using a new "Notification preferences" section located in the account settings.

Minor features

Core

- Any extra arguments given to `Page.serve` are now passed through to `get_context` and `get_template`
- Added `in_menu` and `not_in_menu` methods to `PageQuerySet`
- Added `search` method to `PageQuerySet`
- Added `get_next_siblings` and `get_prev_siblings` to `Page`
- Added `page_published` signal
- Added `copy` method to `Page` to allow copying of pages
- Added `construct_whitelister_element_rules` hook for customising the HTML whitelist used when saving `RichText` fields

- Support for setting a `subpage_types` property on `Page` models, to define which page types are allowed as subpages

Admin

- Removed the "More" section from the menu
- Added pagination to page listings
- Added a new datetime picker widget
- Updated `hallo.js` to version 1.0.4
- Aesthetic improvements to preview experience
- Login screen redirects to dashboard if user is already logged in
- Snippets are now ordered alphabetically
- Added `init_new_page` signal

Search

- Added a new way to configure searchable/filterable fields on models
- Added `get_indexed_objects` allowing developers to customise which objects get added to the search index
- Major refactor of Elasticsearch backend
- Use `match` instead of `query_string` queries
- Fields are now indexed in Elasticsearch with their correct type
- Filter fields are no longer included in `_all`
- Fields with partial matching are now indexed together into `_partials`

Images

- Added `original` as a resizing rule supported by the `{% image %}` tag
- `image` tag now accepts extra keyword arguments to be output as attributes on the `img` tag
- Added an `attrs` property to image rendition objects to output `src`, `width`, `height` and `alt` attributes all in one go

Other

- Added styleguide, for Wagtail developers

Bug fixes

- Animated GIFs are now coalesced before resizing
- The Wand backend clones images before modifying them

- The admin breadcrumb is now positioned correctly on mobile
- The page chooser breadcrumb now updates the chooser modal instead of linking to Explorer
- Embeds - fixed crash when no HTML field is sent back from the embed provider
- Multiple sites with same hostname but different ports are now allowed
- It is no longer possible to create multiple sites with `is_default_site = True`

Backwards-incompatible changes

ElasticUtils replaced with elasticsearch-py

If you are using the elasticsearch backend, you must install the `elasticsearch` module into your environment.

Note: If you are using an older version of Elasticsearch (< 1.0) you must install `elasticsearch` version 0.4.x.

Addition of `expired` column may break old data migrations involving pages

The scheduled publishing mechanism adds an `expired` field to wagtailcore.Page, defaulting to False. Any application code working with Page objects should be unaffected, but any code that creates page records using direct SQL, or within existing South migrations using South's frozen ORM, will fail as this code will be unaware of the `expired` database column. To fix a South migration that fails in this way, add the following line to the `'wagtailcore.page'` entry at the bottom of the migration file:

```
'expired': ('django.db.models.fields.BooleanField', [], {'default': 'False'}),
```

Deprecated features

Template tag libraries renamed

The following template tag libraries have been renamed:

- `pageurl` => `wagtailcore_tags`
- `rich_text` => `wagtailcore_tags`
- `embed_filters` => `wagtailembeds_tags`
- `image_tags` => `wagtailimages_tags`

The old names will continue to work, but output a `DeprecationWarning` - you are advised to update any `{% load %}` tags in your templates to refer to the new names.

New search field configuration format

`indexed_fields` is now deprecated and has been replaced by a new search field configuration format called `search_fields`. See *Indexing* for how to define a `search_fields` property on your models.

`Page.route` method should now return a `RouteResult`

Previously, the `route` method called `serve` and returned an `HttpResponse` object. This has now been split up so `serve` is called separately and `route` must now return a RouteResult object.

If you are overriding `Page.route` on any of your page models, you will need to update the method to return a `RouteResult` object. The old method of returning an `HttpResponse` will continue to work, but this will throw a `DeprecationWarning` and bypass the `before_serve_page` hook, which means in particular that *Private pages* will not work on those page types. See *Adding Endpoints with Custom route() Methods*.

Wagtailadmins `hooks` module has moved to wagtailcore

If you use any `wagtail_hooks.py` files in your project, you may have an import like: `from wagtail.wagtailadmin import hooks`

Change this to: `from wagtail.wagtailcore import hooks`

Miscellaneous

- `Page.show_as_mode` replaced with `Page.serve_preview`
- `Page.get_page_modes` method replaced with `Page.preview_modes` property
- `Page.get_other_siblings` replaced with `Page.get_siblings(inclusive=False)`

Python Module Index

W

wagtail.contrib.wagtailroutablepage, 114
wagtail.contrib.wagtailroutablepage.models, 116
wagtail.contrib.wagtailsearchpromotions, 133
wagtail.wagtailadmin.edit_handlers, 90
wagtail.wagtailcore.models, 95
wagtail.wagtailcore.query, 102
wagtail.wagtaildocs.edit_handlers, 93
wagtail.wagtailimages.edit_handlers, 92
wagtail.wagtailsnippets.edit_handlers, 93

Index

A

ancestor_of() (wagtail.wagtailcore.query.PageQuerySet method), 103
approve_moderation() (wagtail.wagtailcore.models.PageRevision method), 101
as_page_object() (wagtail.wagtailcore.models.PageRevision method), 101

B

base.py, **143**

C

child_of() (wagtail.wagtailcore.query.PageQuerySet method), 103
children (wagtail.wagtailadmin.edit_handlers.FieldRowPanel attribute), 91
children (wagtail.wagtailadmin.edit_handlers.MultiFieldPanel attribute), 90
classname (wagtail.wagtailadmin.edit_handlers.FieldPanel attribute), 90
classname (wagtail.wagtailadmin.edit_handlers.FieldRowPanel attribute), 91
content_json (wagtail.wagtailcore.models.PageRevision attribute), 100
content_type (wagtail.wagtailcore.models.Page attribute), 96
created_at (wagtail.wagtailcore.models.PageRevision attribute), 100

D

descendant_of() (wagtail.wagtailcore.query.PageQuerySet method), 103
dev.py, **143**
DocumentChooserPanel (class in wagtail.wagtaildocs.edit_handlers), 93

F

field_name (wagtail.wagtailadmin.edit_handlers.FieldPanel attribute), 90
FieldPanel (class in wagtail.wagtailadmin.edit_handlers), 90
FieldRowPanel (class in wagtail.wagtailadmin.edit_handlers), 91
fill, **30**
find_for_request() (wagtail.wagtailcore.models.Site static method), 99
first_published_at (wagtail.wagtailcore.models.Page attribute), 96
Full width, **32**
full_url (wagtail.wagtailcore.models.Page attribute), 97

G

get_ancestors() (wagtail.wagtailcore.models.Page method), 97
get_context() (wagtail.wagtailcore.models.Page method), 97
get_descendants() (wagtail.wagtailcore.models.Page method), 97
get_siblings() (wagtail.wagtailcore.models.Page method), 98
get_site_root_paths() (wagtail.wagtailcore.models.Site static method), 99
get_subpage_urls() (wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin class method), 116
get_template() (wagtail.wagtailcore.models.Page method), 97
group (wagtail.wagtailcore.models.GroupPagePermission attribute), 101
GroupPagePermission (class in wagtail.wagtailcore.models), 101

H

has_unpublished_changes (wagtail.wagtailcore.models.Page attribute), 96

heading (wagtail.wagtailadmin.edit_handlers.MultiFieldPanel attribute), 90
height, **30**
hostname (wagtail.wagtailcore.models.Site attribute), 98

I

ImageChooserPanel (class in wagtail.wagtailimages.edit_handlers), 92
in_menu() (wagtail.wagtailcore.query.PageQuerySet method), 102
InlinePanel (class in wagtail.wagtailadmin.edit_handlers), 91
is_creatable (wagtail.wagtailcore.models.Page attribute), 98
is_default_site (wagtail.wagtailcore.models.Site attribute), 99
is_latest_revision() (wagtail.wagtailcore.models.PageRevision method), 101

L

Left-aligned, **32**
live (wagtail.wagtailcore.models.Page attribute), 96
live() (wagtail.wagtailcore.query.PageQuerySet method), 102
local.py, **143**

M

max, **30**
min, **30**
MultiFieldPanel (class in wagtail.wagtailadmin.edit_handlers), 90

N

not_ancestor_of() (wagtail.wagtailcore.query.PageQuerySet method), 104
not_descendant_of() (wagtail.wagtailcore.query.PageQuerySet method), 103
not_live() (wagtail.wagtailcore.query.PageQuerySet method), 102
not_page() (wagtail.wagtailcore.query.PageQuerySet method), 103

O

objects (wagtail.wagtailcore.models.PageRevision attribute), 100
Orderable (class in wagtail.wagtailcore.models), 102
Original, **33**
original, **31**
owner (wagtail.wagtailcore.models.Page attribute), 96

P

Page (class in wagtail.wagtailcore.models), 96, 97
page (wagtail.wagtailcore.models.GroupPagePermission attribute), 101
page (wagtail.wagtailcore.models.PageRevision attribute), 100
page (wagtail.wagtailcore.models.PageViewRestriction attribute), 101
page() (wagtail.wagtailcore.query.PageQuerySet method), 103
PageChooserPanel (class in wagtail.wagtailadmin.edit_handlers), 91
PageQuerySet (class in wagtail.wagtailcore.query), 102
PageRevision (class in wagtail.wagtailcore.models), 100, 101
PageViewRestriction (class in wagtail.wagtailcore.models), 101
parent_page_types (wagtail.wagtailcore.models.Page attribute), 98
password (wagtail.wagtailcore.models.PageViewRestriction attribute), 101
password_required_template (wagtail.wagtailcore.models.Page attribute), 98
permission_type (wagtail.wagtailcore.models.GroupPagePermission attribute), 101
port (wagtail.wagtailcore.models.Site attribute), 99
preview_modes (wagtail.wagtailcore.models.Page attribute), 97
production.py, **143**
public() (wagtail.wagtailcore.query.PageQuerySet method), 104
publish() (wagtail.wagtailcore.models.PageRevision method), 101

R

reject_moderation() (wagtail.wagtailcore.models.PageRevision method), 101
Resize to fill, **33**
Resize to height, **33**
Resize to max, **33**
Resize to min, **33**
Resize to width, **33**
resolve_subpage() (wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin method), 116
reverse_subpage() (wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin method), 116
Right-aligned, **32**
root_page (wagtail.wagtailcore.models.Site attribute), 99
root_url (wagtail.wagtailcore.models.Site attribute), 99
RoutablePageMixin (class in wagtail.contrib.wagtailroutablepage.models),

116
routablepageurl() (in module wagtail.contrib.wagtailroutablepage.templatetags.wagtailroutablepage_tags), 117
route() (wagtail.wagtailcore.models.Page method), 97

S

search() (wagtail.wagtailcore.query.PageQuerySet method), 104
search_description (wagtail.wagtailcore.models.Page attribute), 96
search_fields (wagtail.wagtailcore.models.Page attribute), 98
seo_title (wagtail.wagtailcore.models.Page attribute), 96
serve() (wagtail.wagtailcore.models.Page method), 97
serve_preview() (wagtail.wagtailcore.models.Page method), 97
show_in_menus (wagtail.wagtailcore.models.Page attribute), 96
sibling_of() (wagtail.wagtailcore.query.PageQuerySet method), 104
Site (class in wagtail.wagtailcore.models), 98, 99
slug (wagtail.wagtailcore.models.Page attribute), 96
SnippetChooserPanel (class in wagtail.wagtailsnippets.edit_handlers), 93
sort_order (wagtail.wagtailcore.models.Orderable attribute), 102
specific (wagtail.wagtailcore.models.Page attribute), 97
specific() (wagtail.wagtailcore.query.PageQuerySet method), 105
specific_class (wagtail.wagtailcore.models.Page attribute), 97
submitted_for_moderation (wagtail.wagtailcore.models.PageRevision attribute), 100
submitted_revisions (wagtail.wagtailcore.models.PageRevision attribute), 100
subpage_types (wagtail.wagtailcore.models.Page attribute), 98

T

title (wagtail.wagtailcore.models.Page attribute), 96
type() (wagtail.wagtailcore.query.PageQuerySet method), 104

U

unpublish() (wagtail.wagtailcore.query.PageQuerySet method), 105
url (wagtail.wagtailcore.models.Page attribute), 97
user (wagtail.wagtailcore.models.PageRevision attribute), 100

W

wagtail.contrib.wagtailroutablepage (module), 114
wagtail.contrib.wagtailroutablepage.models (module), 116
wagtail.contrib.wagtailsearchpromotions (module), 133
wagtail.wagtailadmin.edit_handlers (module), 90
wagtail.wagtailcore.models (module), 95
wagtail.wagtailcore.query (module), 102
wagtail.wagtaildocs.edit_handlers (module), 93
wagtail.wagtailimages.edit_handlers (module), 92
wagtail.wagtailsnippets.edit_handlers (module), 93
widget (wagtail.wagtailadmin.edit_handlers.FieldPanel attribute), 90
width, **30**