Wagtail Documentation

Release 0.7

Torchbox

Apr 21, 2017

Contents

1 Getting started **3**
- 1.1 Installation . . . 3
  - 1.1.1 Before you start . . . 3
  - 1.1.2 The demo site (a.k.a. the no-installation route) . . . 3
  - 1.1.3 A basic Wagtail installation . . . 4
  - 1.1.4 Optional extras . . . 5
  - 1.1.5 Alternative installation methods . . . 6
- 1.2 Creating your project . . . 6
  - 1.2.1 The `wagtail start` command . . . 7
  - 1.2.2 The project . . . 7
  - 1.2.3 Getting it running . . . 8

2 Core components **11**
- 2.1 Sites . . . 11
- 2.2 Pages . . . 11
  - 2.2.1 Theory . . . 11
  - 2.2.2 Creating page models . . . 13
  - 2.2.3 For Front End developers . . . 17
  - 2.2.4 Recipes . . . 22
  - 2.2.5 Displaying fields with the Editing API . . . 25
  - 2.2.6 Page Queryset Methods . . . 37
  - 2.2.7 Private pages . . . 40
  - 2.2.8 Embedding URL configuration in Pages . . . 41
- 2.3 Images . . . 44
  - 2.3.1 Using images in templates . . . 44
  - 2.3.2 Advanced topics . . . 46
- 2.4 Snippets . . . 49
  - 2.4.1 Snippet Models . . . 49
  - 2.4.2 Including Snippets in Template Tags . . . 50
  - 2.4.3 Binding Pages to Snippets . . . 51
- 2.5 Search . . . 52
  - 2.5.1 Indexing . . . 52
  - 2.5.2 Searching . . . 55
  - 2.5.3 Backends . . . 59
  - 2.5.4 Indexing . . . 60
  - 2.5.5 Searching . . . 61

2.5.6 Backends . . . 61
2.6 Form builder . . . 61
2.6.1 Usage . . . 61

3 Contrib components **63**
3.1 Generating a static site . . . 63
3.1.1 Installing django-medusa . . . 63
3.1.2 Rendering . . . 63
3.1.3 Advanced topics . . . 64
3.2 Sitemap generation . . . 64
3.2.1 Basic configuration . . . 65
3.2.2 Customising . . . 65
3.3 Frontend cache invalidation . . . 65
3.3.1 Setting it up . . . 66
3.3.2 Advanced usage . . . 67

4 How to **69**
4.1 Configuring Django for Wagtail . . . 69
4.1.1 Middleware (settings.py) . . . 69
4.1.2 Apps (settings.py) . . . 70
4.1.3 Settings Variables (settings.py) . . . 71
4.1.4 Ready to Use Example Config Files . . . 74
4.2 Deploying Wagtail . . . 79
4.2.1 On your server . . . 79
4.2.2 On Gondor . . . 79
4.2.3 On other PAASs and IAASs . . . 79
4.3 Performance . . . 79
4.3.1 Editor interface . . . 79
4.3.2 Public users . . . 80
4.4 Creating multilingual sites . . . 81
4.4.1 The root page . . . 81
4.4.2 Linking pages together . . . 81
4.5 Contributing to Wagtail . . . 83
4.5.1 Issues . . . 83
4.5.2 Pull requests . . . 83
4.5.3 Coding guidelines . . . 83
4.5.4 Styleguide . . . 84
4.5.5 Translations . . . 84
4.5.6 Other contributions . . . 84

5 Reference **85**
5.1 Management commands . . . 85
5.1.1 publish_scheduled_pages . . . 85
5.1.2 fixtree . . . 85
5.1.3 move_pages . . . 85
5.1.4 update_index . . . 86
5.1.5 search_garbage_collect . . . 86

6 Support **87**
6.1 Mailing list . . . 87
6.2 Issues . . . 87
6.3 Torchbox . . . 87

7 Using Wagtail: an Editor's guide **89**
7.1 Introduction . . . 89

7.2 Getting started . . . 89
7.2.1 The Wagtail demo site . . . 89
7.2.2 Logging in . . . 90
7.3 Finding your way around . . . 90
7.3.1 The Dashboard . . . 90
7.3.2 The Explorer menu . . . 92
7.3.3 Using search . . . 93
7.3.4 The Explorer page . . . 93
7.4 Creating new pages . . . 95
7.4.1 Selecting a page type . . . 96
7.4.2 Creating page body content . . . 96
7.4.3 Inserting images and videos in a page . . . 98
7.4.4 Inserting links in a page . . . 101
7.4.5 Inserting videos into body content . . . 102
7.4.6 Inserting links to documents into body text . . . 103
7.4.7 Adding multiple items . . . 104
7.4.8 Required fields . . . 105
7.4.9 The Promote tab . . . 106
7.4.10 Previewing and submitting pages for moderation . . . 106
7.5 Editing existing pages . . . 107
7.6 Managing documents and images . . . 108
7.6.1 Documents . . . 108
7.6.2 Images . . . 109
7.6.3 Snippets . . . 111

8 Release notes **113**
8.1 Roadmap . . . 113
8.1.1 The story so far . . . 113
8.1.2 What's next . . . 113
8.1.3 You decide . . . 114
8.2 Wagtail 0.7 release notes . . . 114
8.2.1 What's new . . . 114
8.2.2 Upgrade considerations . . . 117
8.3 Wagtail 0.6 release notes . . . 117
8.3.1 What's new . . . 118
8.3.2 Upgrade considerations . . . 118
8.3.3 Deprecated features . . . 119
8.4 Wagtail 0.5 release notes . . . 119
8.4.1 What's new . . . 119
8.4.2 Upgrade considerations . . . 120
8.5 Wagtail 0.4.1 release notes . . . 121
8.5.1 Bug fixes . . . 121
8.6 Wagtail 0.4 release notes . . . 121
8.6.1 What's new . . . 122
8.6.2 Backwards-incompatible changes . . . 124
8.6.3 Deprecated features . . . 124

Python Module Index **127**

Wagtail is a modern, flexible CMS, built on Django.

It supports Django 1.6.2+ and 1.7rc3+ on Python 2.6, 2.7, 3.2, 3.3 and 3.4.

CHAPTER 1

Getting started

Installation

Before you start

A basic Wagtail setup can be installed on your machine with only a few prerequisites - see *A basic Wagtail installation*. However, there are various optional components that will improve the performance and feature set of Wagtail, and our recommended software stack includes the PostgreSQL database, ElasticSearch (for free-text searching), the OpenCV library (for image feature detection), and Redis (as a cache and message queue backend). This would be a lot to install in one go, and for this reason, we provide a virtual machine image, for use with Vagrant, with all of these components ready installed.

Whether you just want to try out the demo site, or you're ready to dive in and create a Wagtail site with all bells and whistles enabled, we strongly recommend the Vagrant approach. Nevertheless, if you're the sort of person who balks at the idea of downloading a whole operating system just to run a web app, we've got you covered too. Start from *A basic Wagtail installation* below.

The demo site (a.k.a. the no-installation route)

We provide a demo site containing a set of standard templates and page types - if you're new to Wagtail, this is the best way to try it out and familiarise yourself with how Wagtail works from the point of view of an editor.

If you're happy to use Vagrant, and you just want to set up the Wagtail demo site, or any other pre-existing Wagtail site that ships with Vagrant support, you don't need to install Wagtail at all. Install Vagrant and VirtualBox, and run:

```
git clone https://github.com/torchbox/wagtaildemo.git
cd wagtaildemo
vagrant up
vagrant ssh
```

Then, within the SSH session:

```
./manage.py createsuperuser
./manage.py runserver 0.0.0.0:8000
```

This will make the demo site available on your host machine at the URL http://localhost:8111/ - you can access the Wagtail admin interface at http://localhost:8111/admin/ . Further instructions can be found at *Using Wagtail: an Editor's guide*.

Once you've experimented with the demo site and are ready to build your own site, it's time to install Wagtail on your host machine. Even if you intend to do all further Wagtail work within Vagrant, installing the Wagtail package on your host machine will provide the `wagtail start` command that sets up the initial file structure for your project.

A basic Wagtail installation

This provides everything you need to create a new Wagtail project from scratch, containing no page definitions or templates other than a basic homepage as a starting point for building your site. (For a gentler introduction to Wagtail, you may wish to try out the demo site first!)

You will need Python's pip package manager. We also recommend virtualenvwrapper so that you can manage multiple independent Python environments for different projects - although this is not strictly necessary if you intend to do all your development under Vagrant.

Wagtail is based on the Django web framework and various other Python libraries. Most of these are pure Python and will install automatically using `pip`, but there are a few native-code components that require further attention:

- libsass-python (for compiling SASS stylesheets) - requires a C++ compiler and the Python development headers.
- Pillow (for image processing) - additionally requires libjpeg and zlib.

On Debian or Ubuntu, these can be installed with the command:

```
sudo apt-get install python-dev python-pip g++ libjpeg62-dev zlib1g-dev
```

With these dependencies installed, Wagtail can then be installed with the command:

```
pip install wagtail
```

(or if you're not using virtualenvwrapper: `sudo pip install wagtail`.)

You will now be able to run the following command to set up an initial file structure for your Wagtail project (replace `myprojectname` with a name of your choice):

```
wagtail start myprojectname
```

Without Vagrant: Run the following steps to complete setup of your project (the `migrate` step will prompt you to set up a superuser account):

```
cd myprojectname
pip install -r requirements.txt
python manage.py migrate
python manage.py runserver
```

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

With Vagrant: Run the following steps to bring up the virtual machine and complete setup of your project (the `createsuperuser` step will prompt you to set up a superuser account):

```
cd myprojectname
vagrant up
vagrant ssh
./manage.py createsuperuser
./manage.py runserver 0.0.0.0:8000
```

Your site is now accessible at http://localhost:8111, with the admin backend available at http://localhost:8111/admin/ .

Optional extras

For the best possible performance and feature set, we recommend setting up the following components. If you're using Vagrant, these are provided as part of the virtual machine image and just need to be enabled in the settings for your project. If you're using Wagtail without Vagrant, this will involve additional installation.

PostgreSQL

PostgreSQL is a mature database engine suitable for production use, and is recommended by the Django development team. Non-Vagrant users will need to install the PostgreSQL development headers in addition to Postgres itself; on Debian or Ubuntu, this can be done with the following command:

```
sudo apt-get install postgresql postgresql-server-dev-all
```

To enable Postgres for your project, uncomment the `psycopg2` line from your project's requirements.txt, and in `myprojectname/settings/base.py`, uncomment the DATABASES section for PostgreSQL, commenting out the SQLite one instead. Then run:

```
pip install -r requirements.txt
createdb -Upostgres myprojectname
./manage.py syncdb
./manage.py migrate
```

This assumes that your PostgreSQL instance is configured to allow you to connect as the 'postgres' user - if not, you'll need to adjust the `createdb` line and the database settings in settings/base.py accordingly.

ElasticSearch

Wagtail integrates with ElasticSearch to provide full-text searching of your content, both within the Wagtail interface and on your site's front-end. If ElasticSearch is not available, Wagtail will fall back to much more basic search functionality using database queries. ElasticSearch is pre-installed as part of the Vagrant virtual machine image; non-Vagrant users can use the debian.sh or ubuntu.sh installation scripts as a guide.

To enable ElasticSearch for your project, uncomment the `elasticsearch` line from your project's requirements.txt, and in `myprojectname/settings/base.py`, uncomment the WAGTAILSEARCH_BACKENDS section. Then run:

```
pip install -r requirements.txt
./manage.py update_index
```

Image feature detection

Wagtail can use the OpenCV computer vision library to detect faces and other features in images, and use this information to select the most appropriate centre point when cropping the image. OpenCV is

pre-installed as part of the Vagrant virtual machine image, and Vagrant users can enable this by setting `WAGTAILIMAGES_FEATURE_DETECTION_ENABLED` to True in `myprojectname/settings/base.py`. For installation outside of Vagrant, see *Feature Detection*.

Alternative installation methods

Ubuntu

If you have a fresh instance of Ubuntu 13.04 or later, you can install Wagtail, along with a demonstration site containing a set of standard templates and page types, in one step. As the root user:

```
curl -O https://wagtail.io/ubuntu.sh; bash ubuntu.sh
```

This script installs all the dependencies for a production-ready Wagtail site, including PostgreSQL, Redis, Elasticsearch, Nginx and uwsgi. We recommend you check through the script before running it, and adapt it according to your deployment preferences. The canonical version is at github.com/torchbox/wagtail/blob/master/scripts/install/ubuntu.sh.

Debian

If you have a fresh instance of Debian 7, you can install Wagtail, along with a demonstration site containing a set of standard templates and page types, in one step. As the root user:

```
curl -O https://wagtail.io/debian.sh; bash debian.sh
```

This script installs all the dependencies for a production-ready Wagtail site, including PostgreSQL, Redis, Elasticsearch, Nginx and uwsgi. We recommend you check through the script before running it, and adapt it according to your deployment preferences. The canonical version is at github.com/torchbox/wagtail/blob/master/scripts/install/debian.sh.

Docker

@oyvindsk has built a Dockerfile for the Wagtail demo. Simply run:

```
docker run -p 8000:8000 -d oyvindsk/wagtail-demo
```

then access the site at http://your-ip:8000 and the admin interface at http://your-ip:8000/admin using admin / test.

See https://index.docker.io/u/oyvindsk/wagtail-demo/ for more details.

Creating your project

Contents

- *The `wagtail start` command*
- *The project*
  - *The "core" app*
  - *Default templates and static files*

- *Vagrant configuration*
- *Django settings*

- *Getting it running*
  - *With Vagrant*

The `wagtail start` command

The easiest way to start a new project with wagtail is to use the `wagtail start` command. This command is installed into your environment when you install Wagtail (see: *Installation*).

The command works the same way as `django-admin.py startproject` except that the produced project is pre-configured for Wagtail. It also contains some useful extras which we will look at in the next section.

To create a project, cd into a directory where you would like to create your project and run the following command:

```
wagtail start mysite
```

The project

Lets look at what `wagtail start` created:

```
mysite/
    core/
        static/
        templates/
            base.html
            404.html
            500.html
    mysite/
        settings/
            base.py
            dev.py
            production.py
    manage.py
    vagrant/
        provision.sh
    Vagrantfile
    readme.rst
    requirements.txt
```

The "core" app

Location: `/mysite/core/`

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/core/templates/` and `/mysite/core/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 sass file. Wagtail uses `django-compressor` for compiling and compressing static files. For more information, see: Django Compressor Documentation

Vagrant configuration

Location: `/Vagrantfile` and `/vagrant/`

If you have Vagrant installed, these files let you easily setup a development environment with PostgreSQL and Elasticsearch inside a virtual machine.

See below section *With Vagrant* for info on how to use Vagrant in development

If you do not want to use Vagrant, you can just delete these files.

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.

Getting it running

With Vagrant

This is the easiest way to get the project running. Vagrant runs your project locally in a virtual machine so you can use PostgreSQL and Elasticsearch in development without having to install them on your host machine. If you haven't yet installed Vagrant, see: Installing Vagrant.

To setup the Vagrant box, run the following commands

```
vagrant up # This may take some time on first run
vagrant ssh
# within the ssh session
dj createsuperuser
djrun
```

If you now visit http://localhost:8111 you should see a very basic "Welcome to your new Wagtail site!" page.

You can browse the Wagtail admin interface at: http://localhost:8111/admin

You can read more about how Vagrant works at: https://docs.vagrantup.com/v2/

The `dj` and `djrun` aliases

When using Vagrant, the Wagtail template provides two aliases: `dj` and `djrun` which can be used in the `vagrant ssh` session.

`dj` This is short for `python manage.py` so you can use it to reduce typing. For example: `python manage.py syncdb` becomes `dj syncdb`.

`djrun` This is short for `python manage.py runserver 0.0.0.0:8000`. This is used to run the testing server which is accessible from `http://localhost:8111` (note that the port number gets changed by Vagrant)

CHAPTER 2

Core components

Sites

Wagtail allows mapping a "site" (hostname or domain) to any node in the tree, using that node as the site's root. See *Theory*.

Access this by going to "Settings" and then "Sites". To try out a development site, add a single site with the hostname `localhost` at port `8000` and map it to one of the pieces of content you have created.

Pages

Note: This documentation is currently being written.

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*.

Creating page models

Wagtail provides a `Page` class to represent types of page (a.k.a Content types). Developers should subclass `Page` for their own page models.

`Page` uses Django's model interface, so you can include any field type and field options that Django allows. Wagtail provides some field types of its own which simplify data entry in the Wagtail admin interface. Wagtail also wraps

Django's field types and widgets with its own concept of "Edit Handlers" and "Panels". These further improve the data entry experience.

An example Wagtail Page Model

This example represents a typical blog post:

```
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.wagtailimages.edit_handlers import ImageChooserPanel
from wagtail.wagtailimages.models import Image

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



BlogPage.content_panels = [
    FieldPanel('title', classname="full title"),
    FieldPanel('date'),
    FieldPanel('body', classname="full"),
]

BlogPage.promote_panels = [
    FieldPanel('slug'),
    FieldPanel('seo_title'),
    FieldPanel('show_in_menus'),
    FieldPanel('search_description'),
    ImageChooserPanel('feed_image'),
]
```

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.

In the example above the `BlogPage` class defines three properties: `body`, `date`, and `feed_image`. These are a mix of basic Django models (`DateField`), Wagtail fields (`RichTextField`), and a pointer to a Wagtail model (`Image`).

Below that the `content_panels` and `promote_panels` lists define the capabilities and layout of the page editing interface in the Wagtail admin. The lists are filled with "panels" and "choosers", which will provide a fine-grain interface for inputting the model's content. The `ImageChooserPanel`, for instance, lets one browse the image library, upload new images and input image metadata. The `RichTextField` is the basic field for creating web-ready website rich text, including text formatting and embedded media like images and video. The Wagtail admin offers other choices for fields, Panels, and Choosers, with the option of creating your own to precisely fit your content

without workarounds or other compromises.

Your models may be even more complex, with methods overriding the built-in functionality of the `Page` to achieve webdev magic. Or, you can keep your models simple and let Wagtail's built-in functionality do the work.

`Page` Class Reference

Default fields

Wagtail provides some fields for the `Page` class by default, which will be common to all your pages. You don't need to add these fields to your own page models however you do need to allocate them to `content_panels`, `promote_panels` or `settings_panels`. See above example and for further information on the panels see *Displaying fields with the Editing API*.

`title` (string, required) Human-readable title of the page - what you'd probably use as your `<h1>` tag.

`slug` (string, required) Machine-readable URL component for this page. The name of the page as it will appear in URLs e.g `http://domain.com/blog/[my-slug]/`

`seo_title` (string) Alternate SEO-crafted title, mainly for use in the page `<title>` tag.

`search_description` (string) SEO-crafted description of the content, used for internal search indexing, suitable for your page's `<meta name="description">` tag.

`show_in_menus` (boolean) Toggles whether the page should be considered for inclusion in any site-wide menus you create.

`go_live_at` (datetime) A datetime on which the page should be automatically made published (made publicly accessible)

`expire_at` (datetime) A datetime on which the page should be unpublished, rendering it inaccessible to the public.

Page Attributes, Properties and Methods Reference

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`**(*id*, *path*, *depth*, *numchild*, *title*, *slug*, *content_type_id*, *live*, *has_unpublished_changes*, *url_path*, *owner_id*, *seo_title*, *show_in_menus*, *search_description*, *go_live_at*, *expire_at*, *expired*, *locked*, *latest_revision_created_at*)

`specific`
Decorator that converts a method with a single self argument into a property cached on the instance.

`specific_class`
Decorator that converts a method with a single self argument into a property cached on the instance.

`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

classmethod get_verbose_name()
Returns the human-readable "verbose name" of this page model e.g "Blog page".

relative_url(*current_site*)
Return the 'most appropriate' URL for this page taking into account the site we're currently on; a local URL if the site matches, or a fully qualified one otherwise. Return None if the page is not routable.

is_navigable()
Return true if it's meaningful to browse subpages of this page - i.e. it currently has subpages, or it's at the top level (this rule necessary for empty out-of-the-box sites to have working navigation)

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*)

classmethod search(*query_string*, *show_unpublished=False*, *search_title_only=False*, *extra_filters={}*, *prefetch_related=[]*, *path=None*)

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']
```

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*

Tips

Friendly model names

Make your model names more friendly to users of Wagtail 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.

The above example also ensures the name of the

Helpful model descriptions

As your site becomes more complex users may require some prompting in deciding which content type to use when creating a new page. Developers can add a description to their Models by extending Django's internal model `Meta` class.

Insert the following once at the top of your `models.py`:

```
import django.db.models.options as options
options.DEFAULT_NAMES = options.DEFAULT_NAMES + ('description',)
```

Then for each model as necessary, add a description option to the model `Meta` class

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

    class Meta:
        description = "The top level homepage for your site"
        verbose_name = "Homepage"
```

(This method can be used to extend the Model Meta class in various ways however Wagtail only supports the addition of a `description` option).

For Front End developers

Contents

- *Overview*
- *Templates*
  - *Page content*
- *Static assets*
  - *User images*
- *Template tags & filters*
  - *Images (tag)*
  - *Rich text (filter)*
    - *Responsive Embeds*
  - *Internal links (tag)*
    - *pageurl*
    - *slugurl*
  - *Static files (tag)*
- *Wagtail User Bar*

Overview

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 `self.`. e.g the page title `{{ self.title }}` or another field `{{ self.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 self.photo width-400 %}

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

See *Using images in templates* for full documentation.

Rich text (filter)

Changed in version 0.4: The 'rich_text' tags library was renamed to 'wagtailcore_tags'

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 %}
...
{{ self.body|richtext }}
```

Note: Note that the template tag loaded differs from the name of the filter.

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)

Changed in version 0.4: The 'pageurl' tags library was renamed to 'wagtailcore_tags'

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 self.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 self.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
}
```

Recipes

Overriding the serve() Method

Wagtail defaults to serving `Page`-derived models by passing `self` 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

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

...

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=[]):
        render(request, self.template, {
            'self': self,
            'echo': ' '.join(path_components),
        })
```

```
Echoer.content_panels = [
    FieldPanel('title', classname="full title"),
]

Echoer.promote_panels = [
    MultiFieldPanel(COMMON_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.tags import ClusterTaggableManager
from taggit.models import Tag, 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, {
        'self': self,
        '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 self.tags.all %}
    <a href="{% pageurl self.blog_index %}?tag={{ tag }}">{{ tag }}</a>
{% endfor %}
```

Iterating through `self.tags.all` will display each tag associated with `self`, 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.

Custom Page Contexts by Overriding get_context()

Load Alternate Templates by Overriding get_template()

Preview Modes

preview_modes serve_preview

Displaying fields with the Editing API

Note: This documentation is currently being written.

Wagtail provides a highly-customizable editing interface consisting of several components:

- **Fields** — built-in content types to augment the basic types provided by Django
- **Panels** — the basic editing blocks for fields, groups of fields, and related object clusters
- **Choosers** — interfaces for finding related objects in a ForeignKey relationship

Configuring your models to use these components will shape the Wagtail editor to your needs. Wagtail also provides an API for injecting custom CSS and JavaScript for further customization, including extending the hallo.js rich text editor.

There is also an Edit Handler API for creating your own Wagtail editor components.

Defining Panels

A "panel" is the basic editing block in Wagtail. Wagtail will automatically pick the appropriate editing widget for most Django field types; implementors just need to add a panel for each field they want to show in the Wagtail page editor, in the order they want them to appear.

There are four basic types of panels:

`FieldPanel( field_name, classname=None )` This is the panel used for basic Django field types. `field_name` is the name of the class property used in your model definition. `classname` 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.

`MultiFieldPanel( children, heading="", classname=None )` This panel condenses several `FieldPanel` s or choosers, from a list or tuple, under a single `heading` string.

`InlinePanel( base_model, relation_name, panels=None, classname=None, label='', help_te` 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 very powerful, but tricky 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( children, classname=None)` This panel is purely aesthetic. It 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*` classnames added to each of the child Panels of the FieldRowPanel. The Wagtail editing interface is layed out using a grid system, in which the maximum width of the editor is 12 columns wide. 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.

(In addition to these four, there are also Chooser Panels, detailed below.)

Wagtail provides a tabbed interface to help organize panels. Three such tabs are provided:

- `content_panels` is the main tab, used for the bulk of your model's fields.
- `promote_panels` is suggested for organizing fields regarding the promotion of the page around the site and the Internet. For example, a field to dictate whether the page should show in site-wide menus, descriptive text that should appear in site search results, SEO-friendly titles, OpenGraph meta tag content and other machine-readable information.
- `settings_panels` is essentially for non-copy fields. By default it contains the page's scheduled publishing fields. Other suggested fields could include a field to switch between one layout/style and another.

Let's look at an example of a panel definition:

```
COMMON_PANELS = (
  FieldPanel('slug'),
  FieldPanel('seo_title'),
  FieldPanel('show_in_menus'),
  FieldPanel('search_description'),
)

...

class ExamplePage( Page ):
  # field definitions omitted
  ...

ExamplePage.content_panels = [
  FieldPanel('title', classname="full title"),
  FieldPanel('body', classname="full"),
  FieldRowPanel([
    FieldPanel('start_date', classname="col3"),
    FieldPanel('end_date', classname="col3"),
  ]),
  ImageChooserPanel('splash_image'),
  DocumentChooserPanel('free_download'),
  PageChooserPanel('related_page'),
]

ExamplePage.promote_panels = [
  MultiFieldPanel(COMMON_PANELS, "Common page configuration"),
]
```

After the `Page`-derived class definition, just add lists of panel definitions to order and organize the Wagtail page editing interface for your model.

Built-in Fields and Choosers

Django's field types are automatically recognized 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.

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()

BookPage.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)*.

Images

One of the features of Wagtail is 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='+'
  )

BookPage.content_panels = [
  ImageChooserPanel('cover'),
  # ...
]
```

Django's default behavior is to "cascade" deletions through a ForeignKey relationship, which is probably not what you want happening. 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*.

Documents

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='+'
  )

BookPage.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.

Documents can be used directly in templates without tags or filters. Its properties are:

`title` The title of the document.

`url` URL to the file.

`created_at` The date and time the document was created (DateTime).

`filename` The filename of the file.

`file_extension` The extension of the file.

`tags` A `TaggableManager` which keeps track of tags associated with the document (uses the `django-taggit` module).

Pages and Page-derived Models

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='+',
  )

BookPage.content_panels = [
  PageChooserPanel('related_page', 'demo.PublisherPage'),
  # ...
]
```

`PageChooserPanel` takes two arguments: a field name and an optional page type. Specifying a page type (in the form of an `"appname.modelname"` string) will filter the chooser to display only pages of that type.

Snippets

Snippets are vanilla Django models you create yourself without a Wagtail-provided base class. So using them as a field in a page requires specifying your own `appname.modelname`. A chooser, `SnippetChooserPanel`, is provided which takes the field name and snippet class.

```
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', Advert),
  # ...
]
```

See *Snippets* for more information.

Field Customization

By adding CSS classnames 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 behavior from Django's admin, so you may find many options for customization 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.

Col*

Fields within a `FieldRowPanel` can have their width dictated in terms of the number of columns it should span. The `FieldRowPanel` is always considered to be 12 columns wide regardless of browser size or the nesting of `FieldRowPanel` in any other type of panel. Specify a number of columns thus: `col3`, `col4`, `col6` etc (up to 12). The resulting width with be *relative* to the full width of the `FieldRowPanel`.

Required Fields

To make input or chooser selection manditory 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)).

MultiFieldPanel

The `MultiFieldPanel` groups a list of child fields into a fieldset, which can also be collapsed into a heading bar to save space.

```
BOOK_FIELD_COLLECTION = [
  ImageChooserPanel('cover'),
  DocumentChooserPanel('book_file'),
  PageChooserPanel('publisher'),
]

BookPage.content_panels = [
  MultiFieldPanel(
    BOOK_FIELD_COLLECTION,
    heading="Collection of Book Fields",
    classname="collapsible collapsed"
  ),
  # ...
]
```

By default, `MultiFieldPanel` s are expanded and not collapsible. Adding the classname `collapsible` 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.

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 ):
  # ...

BookPage.content_panels = [
  # ...
  InlinePanel( BookPage, '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( base_model, relation_name, panels=None, label='', help_text='' )
```

`base_model` is the model you're extending with the cluster. 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. Finally, `label` and `help_text` provide a heading and caption, respectively, for the Wagtail editor.

For another example of using model clusters, see *Tagging*

For more on `django-modelcluster`, visit the django-modelcluster github project page.

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

Edit Handler API

Admin 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.

New in version 0.5: Decorator syntax was added in 0.5; earlier versions only supported `hooks.register` as an ordinary function call.

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 New in version 0.4.

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>")
```

construct_wagtail_edit_bird 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_edit_bird')
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() )
```

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 New in version 0.6.

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.

Also applied as a CSS class to the wrapping `<li>`, as `"menu-{name}"`.

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` .. versionadded:: 0.4

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.

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 customize 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` takes the following init 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.

`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.

Content Index Pages (CRUD)

Custom Choosers

Tests

Page Queryset Methods

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*)

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*, *backend='default'*)
This runs a search query on all the pages in the queryset

See: *Searching Pages*

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()
```

Private pages

New in version 0.4.

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 `self` 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'
```

Embedding URL configuration in Pages

New in version 0.6.

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 `BlogIndex` page.

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, using Django's urlconf scheme. 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*, and configure the `subpage_urls` attribute with your URL configuration.

Here's an example of an `EventPage` with three views:

```
from django.conf.urls import url

from wagtail.contrib.wagtailroutablepage.models import RoutablePageMixin
from wagtail.wagtailcore.models import Page


class EventPage(RoutablePageMixin, Page):
    subpage_urls = (
        url(r'^$', 'current_events', name='current_events'),
        url(r'^past/$', 'past_events', name='past_events'),
        url(r'^year/(\d+)/$', 'events_for_year', name='events_for_year'),
    )

    def current_events(self, request):
        """
        View function for the current events page
        """
        ...

    def past_events(self, request):
        """
        View function for the past events page
        """
        ...

    def events_for_year(self, request):
        """
        View function for the events for year page
        """
        ...
```

The `RoutablePageMixin` class

class `wagtail.contrib.wagtailroutablepage.models.`**`RoutablePageMixin`**

This class can be mixed in to a Page subclass to allow urlconfs to be embedded inside pages.

`subpage_urls` = None

Set this to a tuple of `django.conf.urls.url` objects.

Example:

```
from django.conf.urls import url

from wagtail.wagtailcore.models import Page


class MyPage(RoutablePageMixin, Page):
    subpage_urls = (
        url(r'^$', 'main', name='main'),
        url(r'^archive/$', 'archive', name='archive'),
        url(r'^archive/(?P<year>[0-9]{4})/$', 'archive', name='archive
↪'),
    )

    def main(self, request):
        ...

    def archive(self, request, year=None):
        ...
```

resolve_subpage(*path*)
This finds a view method/function from a URL path.

Example:

```
view, args, kwargs = page.resolve_subpage('/archive/')
response = view(request, *args, **kwargs)
```

reverse_subpage(*name*, *args=None*, *kwargs=None*)
This method does the same job as Djangos' built in "urlresolvers.reverse()" function for subpage urlconfs.

Example:

```
url = page.url + page.reverse_subpage('archive', 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 self "feed" %}
```

```
{% routablepageurl self "archive" 2014 08 14 %}
{% routablepageurl self "food" foo="bar" baz="quux" %}
```

Images

Using images in templates

Changed in version 0.4: The 'image_tags' tags library was renamed to 'wagtailimages_tags'

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 self.photo width-400 %}

<!-- or a square thumbnail: -->
{% image self.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 `self.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 self.photo max-1000x500 %}
```

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.

`min` (takes two dimensions)

```
{% image self.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 it's height and width changed to 500, i.e matching the width required, but greater than the height.

`width` (takes one dimension)

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

Reduces the width of the image to the dimension specified.

height (takes one dimension)

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

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

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

```
{% image self.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.

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 self.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 self.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 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

New in version 0.4.

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

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

<img src="{{ tmp_photo.url }}" width="{{ tmp_photo.width }}"
    height="{{ tmp_photo.height }}" alt="{{ tmp_photo.alt }}" 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 `{{ self.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

New in version 0.4.

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" />
```

Advanced topics

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 = patterns('',
```

```
    ...

    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)">
```

Performance

Currently, Wagtail will regenerate the image every time it is requested. For high volume sites, it is recommended to use a frontend cache to reduce load on the backend servers.

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 image.focal_point is None:
        image.focal_point = image.get_suggested_focal_point()
        image.save()
```

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 are not search-able or order-able in the Wagtail admin, 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 __unicode__(self):
    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 __unicode__(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', Advert),
  # ...
]
```

The snippet could then be accessed within your template as `self.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', Advert),
  ]
```

```
  def __unicode__(self):
    return self.page.title + " -> " + self.advert.text

class BookPage(Page):
  ...

BookPage.content_panels = [
  InlinePanel(BookPage, '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 self.advert_placements.all %}
  <p><a href="{{ advert_placement.advert.url }}">{{ advert_placement.advert.text }}</
↪a></p>
{% endfor %}
```

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 non-page 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

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`.

To register the signal handlers, add the following code somewhere it would be executed at startup. We reccommend adding this to your projects `urls.py`:

Note: If your project was made with the `wagtail start` command, this will already be set up for you.

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

Changed in version 0.4: The `indexed_fields` configuration format was replaced with `search_fields`

Changed in version 0.6: The `wagtail.wagtailsearch.indexed` module was renamed to `wagtail.wagtailsearch.index`

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** (number) - 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*

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'),
    )
```

Indexing non-page 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 Pages

Wagtail provides a `search` method on the QuerySet for all page models:

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

All methods of `PageQuerySet` are supported by wagtailsearch:

```
# 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>]
```

Searching Images, Documents and custom models

You can search these by using the `search` method on the search backend:

```
>>> from wagtail.wagtailimages.models import Image
>>> from wagtail.wagtailsearch.backends import get_search_backend

# Search images
>>> s = get_search_backend()
>>> s.search("Hello", Image)
[<Image: Hello>, <Image: Hello world!>]
```

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

```
>>> from wagtail.wagtailimages.models import Image
>>> from wagtail.wagtailsearch.backends import get_search_backend

# Search images
>>> s = get_search_backend()
>>> s.search("Hello", Image.objects.filter(uploaded_by_user=user))
[<Image: Hello>]
```

This should work the same way for Documents and custom models as well.

Frontend views

Default Page Search

Wagtail provides a default frontend search interface which indexes the `title` field common to all `Page`-derived models. Let's take a look at all the components of the search interface.

The most basic search functionality just needs a search box which submits a request. Since this will be reused throughout the site, let's put it in `mysite/includes/search_box.html` and then use `{% include ... %}` to weave it into templates:

```
<form action="{% url 'wagtailsearch_search' %}" method="get">
  <input type="text" name="q"{% if query_string %} value="{{ query_string }}"{% endif
↪%}>
  <input type="submit" value="Search">
</form>
```

The form is submitted to the url of the `wagtailsearch_search` view, with the search terms variable `q`. The view will use its own basic search results template.

Let's use our own template for the results, though. First, in your project's `settings.py`, define a path to your template:

```
WAGTAILSEARCH_RESULTS_TEMPLATE = 'mysite/search_results.html'
```

Next, let's look at the template itself:

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

{% block title %}Search{% if search_results %} Results{% endif %}{% endblock %}

{% block search_box %}
  {% include "mysite/includes/search_box.html" with query_string=query_string only %}
{% endblock %}

{% block content %}
  <h2>Search Results{% if request.GET.q %} for {{ request.GET.q }}{% endif %}</h2>
  <ul>
    {% for result in search_results %}
      <li>
        <h4><a href="{% pageurl result.specific %}">{{ result.specific }}</a></h4>
        {% if result.specific.search_description %}
          {{ result.specific.search_description|safe }}
        {% endif %}
      </li>
```

```
    {% empty %}
      <li>No results found</li>
    {% endfor %}
  </ul>
{% endblock %}
```

The search view provides a context with a few useful variables.

query_string The terms (string) used to make the search.

search_results A collection of Page objects matching the query. The `specific` property of `Page` will give the most-specific subclassed model object for the Wagtail page. For instance, if an `Event` model derived from the basic Wagtail `Page` were included in the search results, you could use `specific` to access the custom properties of the `Event` model (`result.specific.date_of_event`).

is_ajax Boolean. This returns Django's `request.is_ajax()`.

query A Wagtail `Query` object matching the terms. The `Query` model provides several class methods for viewing the statistics of all queries, but exposes only one property for single objects, `query.hits`, which tracks the number of time the search string has been used over the lifetime of the site. `Query` also joins to the Editor's Picks functionality though `query.editors_picks`. See *Editor's picks*.

Asynchronous Search with JSON and AJAX

Wagtail provides JSON search results when queries are made to the `wagtailsearch_suggest` view. To take advantage of it, we need a way to make that URL available to a static script. Instead of hard-coding it, let's set a global variable in our `base.html`:

```
<script>
  var wagtailJSONSearchURL = "{% url 'wagtailsearch_suggest' %}";
</script>
```

Now add a simple interface for the search with a `<input>` element to gather search terms and a `<div>` to display the results:

```
<div>
  <h3>Search</h3>
  <input id="json-search" type="text">
  <div id="json-results"></div>
</div>
```

Finally, we'll use JQuery to make the asynchronous requests and handle the interactivity:

```
$(function() {

  // cache the elements
  var searchBox = $('#json-search'),
    resultsBox = $('#json-results');
  // when there's something in the input box, make the query
  searchBox.on('input', function() {
    if( searchBox.val() == ''){
  resultsBox.html('');
  return;
    }
    // make the request to the Wagtail JSON search view
```

```
    $.ajax({
  url: wagtailJSONSearchURL + "?q=" +  searchBox.val(),
  dataType: "json"
    })
    .done(function(data) {
  console.log(data);
  if( data == undefined ){
    resultsBox.html('');
    return;
  }
  // we're in business!  let's format the results
  var htmlOutput = '';
  data.forEach(function(element, index, array){
    htmlOutput += '<p><a href="' + element.url + '">' + element.title + '</a></p>';
  });
  // and display them
  resultsBox.html(htmlOutput);
    })
    .error(function(data){
  console.log(data);
    });
  });

});
```

Results are returned as a JSON object with this structure:

```
{
  [
    {
  title: "Lumpy Space Princess",
  url: "/oh-my-glob/"
    },
    {
  title: "Lumpy Space",
  url: "/no-smooth-posers/"
    },
    ...
  ]
}
```

What if you wanted access to the rest of the results context or didn't feel like using JSON? Wagtail also provides a generalized AJAX interface where you can use your own template to serve results asynchronously.

The AJAX interface uses the same view as the normal HTML search, `wagtailsearch_search`, but will serve different results if Django classifies the request as AJAX (`request.is_ajax()`). Another entry in your project settings will let you override the template used to serve this response:

```
WAGTAILSEARCH_RESULTS_TEMPLATE_AJAX = 'myapp/includes/search_listing.html'
```

In this template, you'll have access to the same context variables provided to the HTML template. You could provide a template in JSON format with extra properties, such as `query.hits` and editor's picks, or render an HTML snippet that can go directly into your results `<div>`. If you need more flexibility, such as multiple formats/templates based on differing requests, you can set up a custom search view.

Custom Search Views

This functionality is still under active development to provide a streamlined interface, but take a look at `wagtail/wagtail/wagtailsearch/views/frontend.py` if you are interested in coding custom search views.

Editor's picks

Editor's picks are a way of explicitly linking relevant content to search terms, so results pages can contain curated content in addition to results from the search algorithm. In a template using the search results view, editor's picks can be accessed through the variable `query.editors_picks`. To include editor's picks in your search results template, use the following properties.

`query.editors_picks.all` This gathers all of the editor's picks objects relating to the current query, in order according to their sort order in the Wagtail admin. You can then iterate through them using a `{% for ... %}` loop. Each editor's pick object provides these properties:

`editors_pick.page` The page object associated with the pick. Use `{% pageurl editors_pick.page %}` to generate a URL or provide other properties of the page object.

`editors_pick.description` The description entered when choosing the pick, perhaps explaining why the page is relevant to the search terms.

Putting this all together, a block of your search results template displaying editor's picks might look like this:

```
{% with query.editors_picks.all as editors_picks %}
  {% if editors_picks %}
    <div class="well">
    <h3>Editors picks</h3>
  <ul>
    {% for editors_pick in editors_picks %}
      <li>
        <h4>
      <a href="{% pageurl editors_pick.page %}">
        {{ editors_pick.page.title }}
      </a>
        </h4>
        <p>{{ editors_pick.description|safe }}</p>
      </li>
    {% endfor %}
  </ul>
    </div>
  {% endif %}
{% endwith %}
```

Backends

Wagtail can degrade to a database-backed text search, but we strongly recommend Elasticsearch.

Database Backend

The default DB search backend uses Django's `__icontains` filter.

Elasticsearch Backend

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

```
pip install elasticsearch
```

Note: If you are using Elasticsearch < 1.0, install elasticsearch-py version 0.4.5: `pip install elasticsearch==0.4.5`

The backend is configured in settings:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch.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 non-page 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*

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 implementors can extend to create their own 'Form' page 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 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(FormPage, '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 'self' variable. A very basic template for the form would thus be:

```
{% load pageurl rich_text %}
<html>
    <head>
        <title>{{ self.title }}</title>
    </head>
    <body>
        <h1>{{ self.title }}</h1>
        {{ self.intro|richtext }}
        <form action="{% pageurl self %}" 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.

CHAPTER 3

Contrib components

Generating a static site

This document describes how to render your Wagtail site into static HTML files on your local filesystem, Amazon S3 or Google App Engine, using django medusa and the `wagtail.contrib.wagtailmedusa` module.

Installing django-medusa

First, install django medusa from pip:

```
pip install django-medusa
```

Then add `django_medusa` and `wagtail.contrib.wagtailmedusa` to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'django_medusa',
    'wagtail.contrib.wagtailmedusa',
]
```

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

Replacing GET parameters with custom routing

Pages which require GET parameters (e.g. for pagination) don't generate suitable filenames for generated HTML files so they need to be changed to use custom routing instead.

For example, let's say we have a Blog Index which uses pagination. We can override the `route` method to make it respond on urls like '/page/1', and pass the page number through to the `serve` method:

```
from wagtail.wagtailcore.url_routing import RouteResult

class BlogIndex(Page):
    ...

    def serve(self, request, page=1):
        ...

    def route(self, request, path_components):
        if self.live and len(path_components) == 2 and path_components[0] == 'page':
            try:
                return RouteResult(self, kwargs={'page': int(path_components[1])})
            except (TypeError, ValueError):
                pass

        return super(BlogIndex, self).route(request, path_components)
```

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 generation

New in version 0.4.

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 = patterns('',
    ...

    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.

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 invalidation

New in version 0.4.

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"
  ]
  ```

The `wagtailfrontendcache` module provides a set of signal handlers which will automatically purge the cache whenever a page is published or deleted. You should register these somewhere at the top of your `urls.py` file:

```
# urls.py
from wagtail.contrib.wagtailfrontendcache.signal_handlers import register_signal_
↪handlers

register_signal_handlers()
```

Varnish/Squid

Add an item into the `WAGTAILFRONTENDCACHE` 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.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)


@register(page_published, sender=BlogPage):
def blog_published_handler(instance):
    blog_page_changed(instance)


@register(pre_delete, sender=BlogPage)
def blog_deleted_handler(instance):
    blog_page_changed(instance)
```

Invalidating individual URLs

`wagtail.contrib.wagtailfrontendcache.utils` provides another utils 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')
```

CHAPTER 4

How to

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 urlconfs 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 Config 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. For configuring sites, see *Sites*.

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',

  'south',
  'compressor',
  'taggit',
  'modelcluster',
  'django.contrib.admin',

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

south Used for database migrations. See South Documentation.

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.

django.contrib.admin The Django admin module. While Wagtail will eventually provide a sites-editing interface, the Django admin is included for now to provide that functionality.

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)

Authentication

```
LOGIN_URL = 'wagtailadmin_login'
LOGIN_REDIRECT_URL = 'wagtailadmin_home'
```

These settings variables set the Django authentication system to redirect to the Wagtail admin login. If you plan to use the Django authentication module to log in non-privileged users, you should set these variables to your own login views. See Django User Authentication.

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.ElasticSearch',
    'INDEX': 'myapp'
  }
}
```

The search settings customize 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!
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.

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.

Search Signal Handlers

```
from wagtail.wagtailsearch import register_signal_handlers as wagtailsearch_register_
↪signal_handlers

wagtailsearch_register_signal_handlers()
```

This loads Wagtail's search signal handlers, which need to be loaded very early in the Django life cycle. While not technically a urlconf, this is a convenient place to load them. Calling this function registers signal handlers to watch for when indexed models get saved or deleted. This allows wagtailsearch to update ElasticSearch automatically.

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.urls import frontend as wagtailsearch_frontend_urls

admin.autodiscover()

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

  url(r'^admin/', include(wagtailadmin_urls)),
  url(r'^search/', include(wagtailsearch_frontend_urls)),
```

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

  # Optional urlconf 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 urlconfs (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 urlconf, but it's what's necessary for Wagtail to flourish.

Ready to Use Example Config 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',

    'south',
    'compressor',
    'taggit',
    'modelcluster',
    'django.contrib.admin',

    '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')

# django-compressor settings
COMPRESS_PRECOMPILERS = (
    ('text/x-scss', 'django_libsass.SassCompiler'),
)

# Auth settings
LOGIN_URL = 'wagtailadmin_login'
LOGIN_REDIRECT_URL = 'wagtailadmin_home'

# 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.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!)
# 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.urls import frontend as wagtailsearch_frontend_urls

admin.autodiscover()


# Signal handlers
from wagtail.wagtailsearch import register_signal_handlers as wagtailsearch_register_
↪signal_handlers
wagtailsearch_register_signal_handlers()


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

    url(r'^admin/', include(wagtailadmin_urls)),
    url(r'^search/', include(wagtailsearch_frontend_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. If you are running Debian or Ubuntu, this installation script for our Vagrant box may be useful:

github.com/torchbox/wagtaildemo/blob/master/etc/install/install.sh

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-cache` to your requirements.txt, and enable it as a cache backend:

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

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

Sending emails in the background using Celery

Various actions in the Wagtail admin backend can trigger notification emails - for example, submitting a page for moderation. In Wagtail's default configuration, these are sent as part of the page request/response cycle, which means that web server threads can get tied up for long periods if many emails are being sent. To avoid this, Wagtail can be configured to do this as a background task, using Celery as a task queue. To install Celery, add `django-celery` to your requirements.txt. A sample configuration, using Redis as the queue backend, would look like:

```
import djcelery

djcelery.setup_loader()

CELERY_SEND_TASK_ERROR_EMAILS = True
BROKER_URL = 'redis://'
```

See the Celery documentation for instructions on running the worker process in development or production.

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.

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.

New in version 0.4: Wagtail supports automatic cache invalidation for Varnish/Squid. See *Frontend cache invalidation* for more information.

Creating multilingual sites

This tutorial will show you a method of creating multilingual sites in Wagtail.

Currently, Wagtail doesn't support multiple languages in the same page. The recommended way of creating multilingual websites in Wagtail at the moment is to create one section of your website for each language.

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 Djangos `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 increse the burdon 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.

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

{% if self.french_page and self.get_language != 'fr' %}
    <a href="{{ self.french_page.url }}">{% trans "View in French" %}</a>
{% endif %}

{% if self.spanish_page and self.get_language != 'es' %}
    <a href="{{ self.spanish_page.url }}">{% trans "View in Spanish" %}</a>
{% endif %}
```

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!

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. 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.

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.

Translations

Wagtail has internationalisation support so if you are fluent in a non-English language you can contribute by localising the interface.

Our preferred way to submit or contribute to a language translation is via Transifex.

If you do not want to use Transifex we also welcome pull requests of `django.po` files for any of the Wagtail modules.

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.

CHAPTER 5

Reference

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.

CHAPTER 6

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.

CHAPTER 7

Using Wagtail: an Editor's guide

Note: Documentation currently incomplete and in draft status

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 the Wagtail demo site. However, the instructions are general enough as to be applicable to any Wagtail site. If you want to use the demo site you can find installation and launch instructions on its Github page.

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

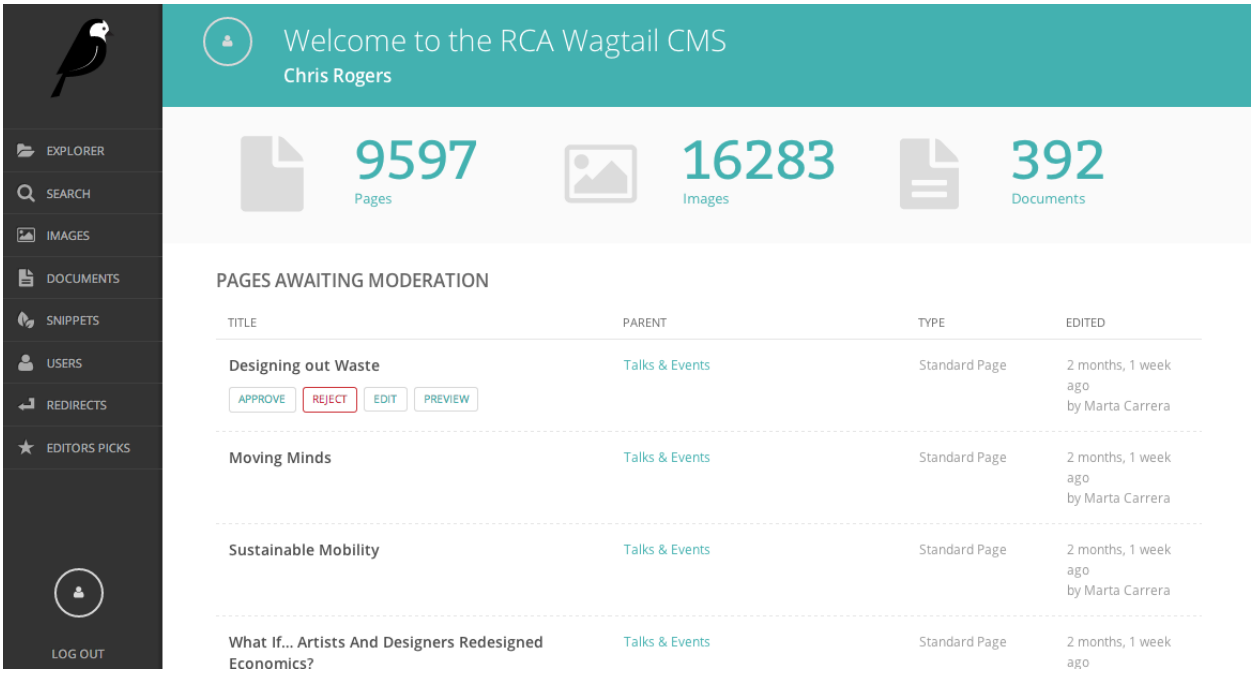
This section describes the different pages that you will see as you navigate around the CMS, and how you can find the content that you are looking for.

The Dashboard

The Dashboard provides information on:

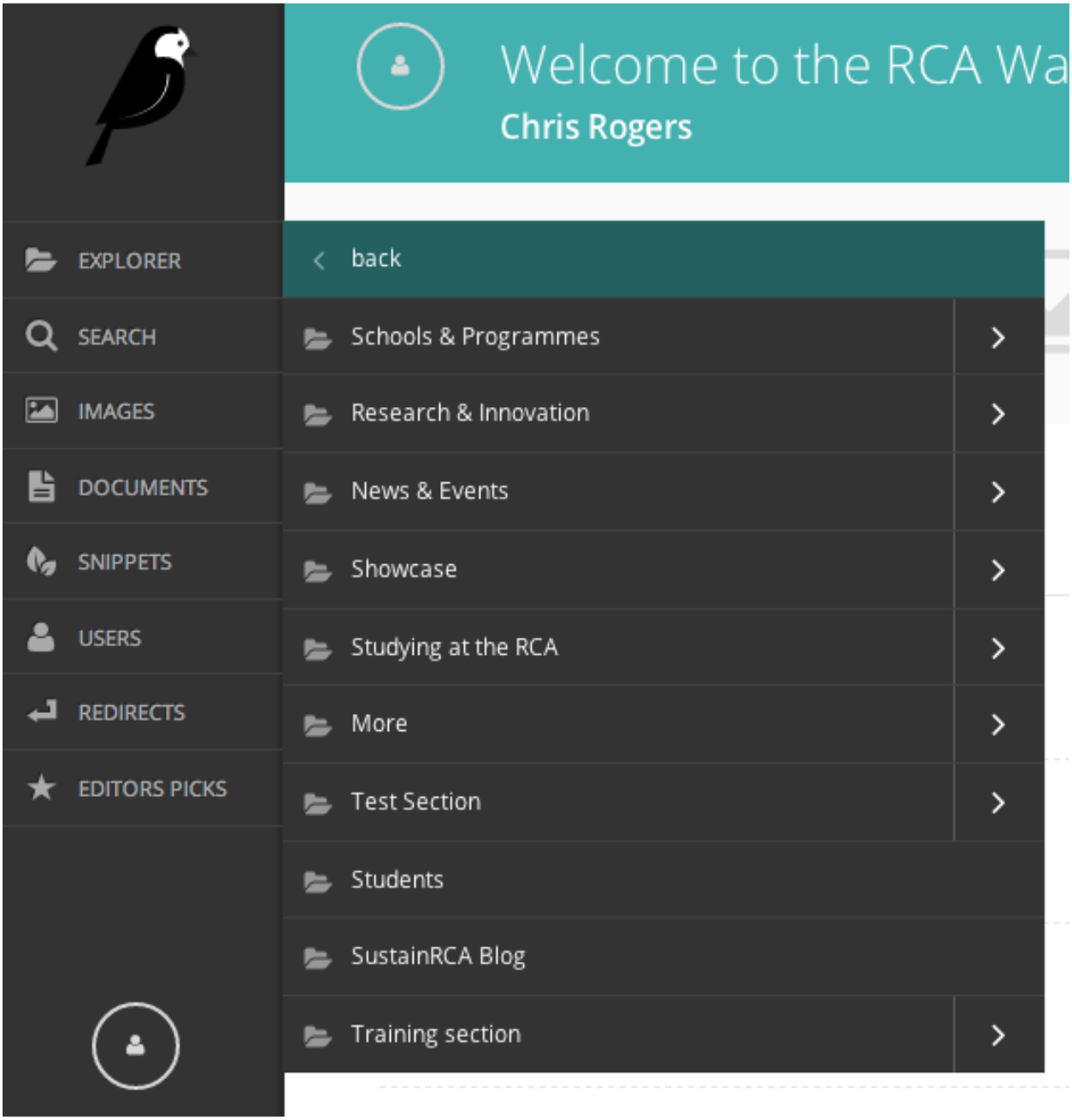
- The number of pages, images, and documents currently held in the Wagtail CMS
- Any pages currently awaiting moderation (if you have these privileges)
- Your most recently edited pages

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.
- Thes 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 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 on one of four 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.

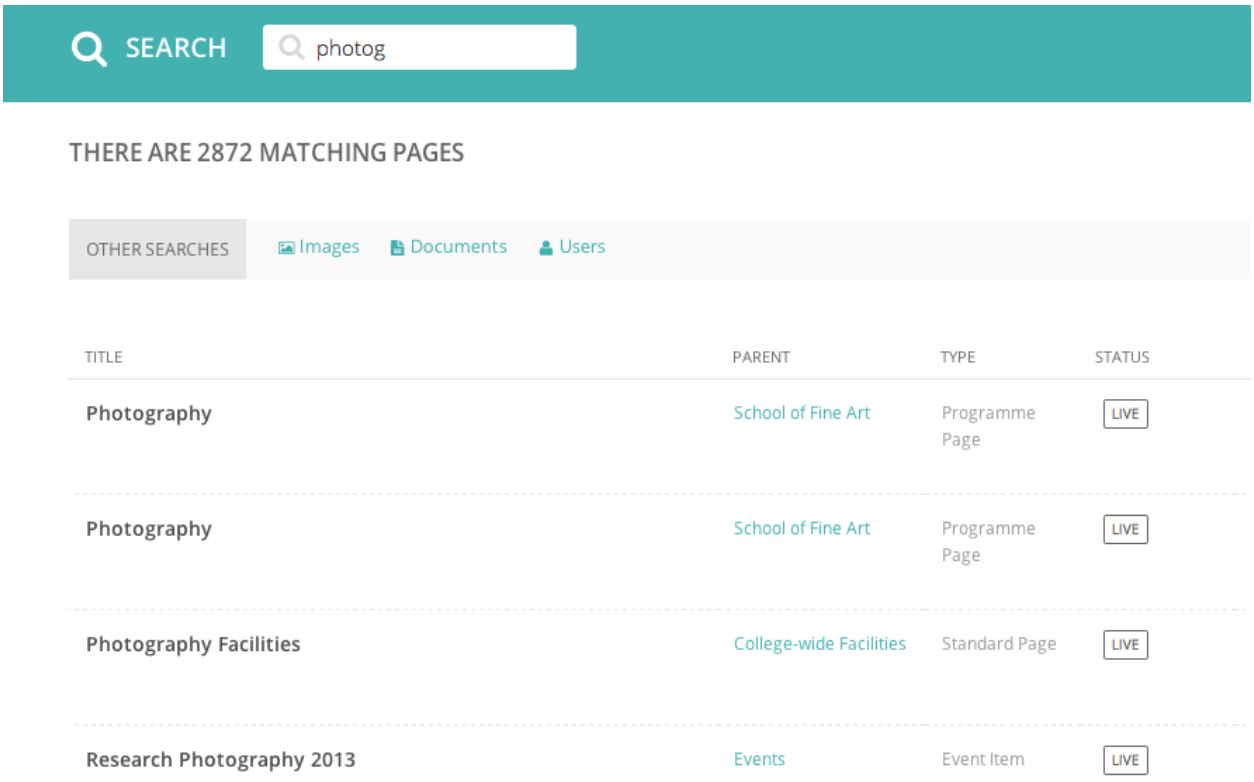
The Explorer menu



- 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.

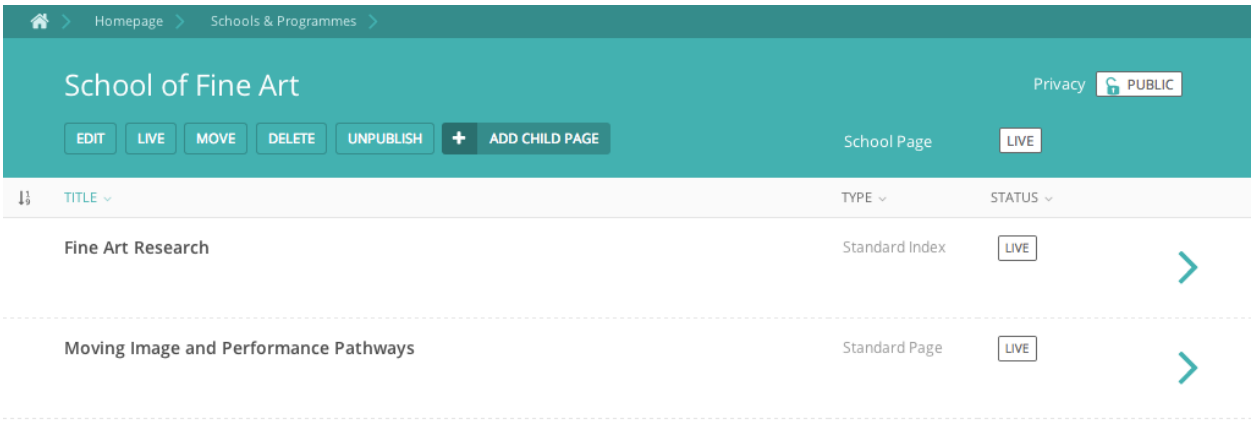
Using search



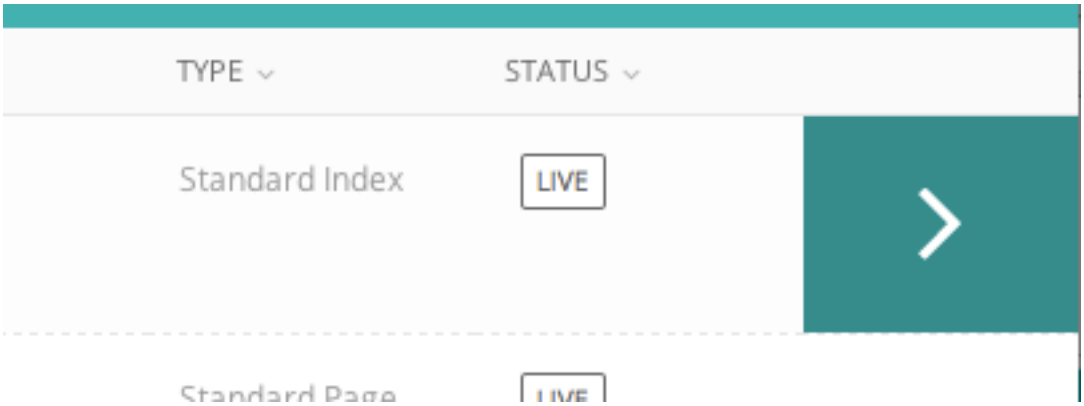
- A very easy way to find the page that you want is to use the main search feature, accessible from the left-hand menu.
- Simply type in part or all of the name of the page you are looking for, and the results below will automatically update as you type.
- Clicking the page title in the results will 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.



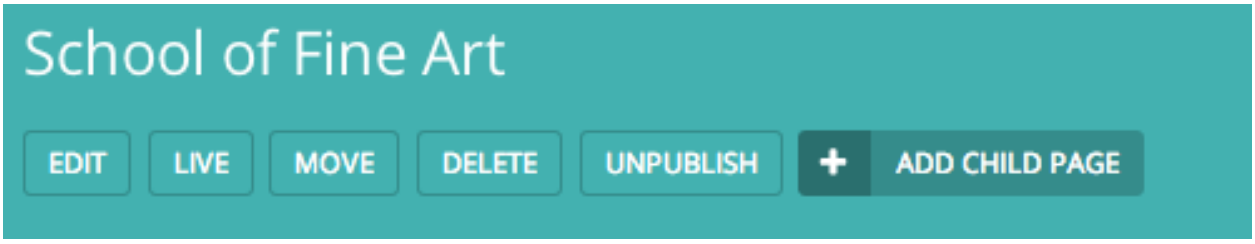
- The name of the section you are looking at is displayed below the breadcrumb (the row of page names beginning with the home icon). Each section is also itself a page (in this case the homepage). Clicking the title of the section takes you to the Edit screen for the section page.
- As the heading suggests, 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.

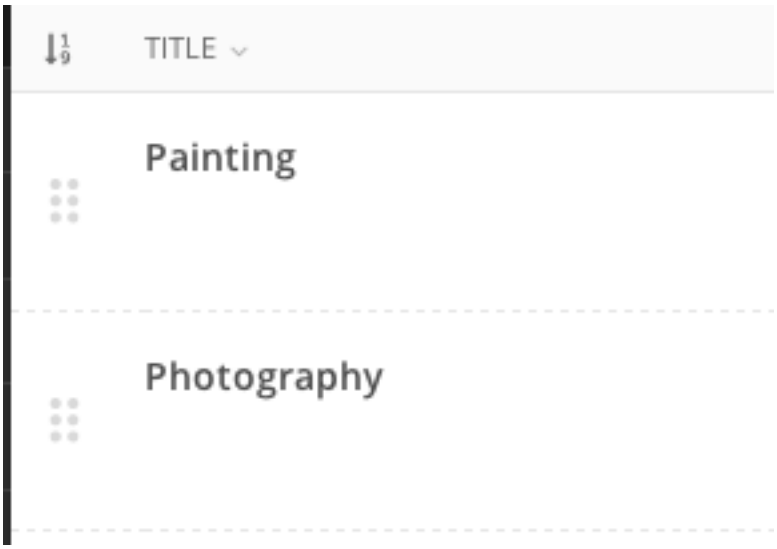


- As you drill down through the site the breadcrumb (the row of pages beginning with the home icon) will display the path you have taken. Clicking on the page titles in the breadcrumb will take you to the Explorer screen for that page.



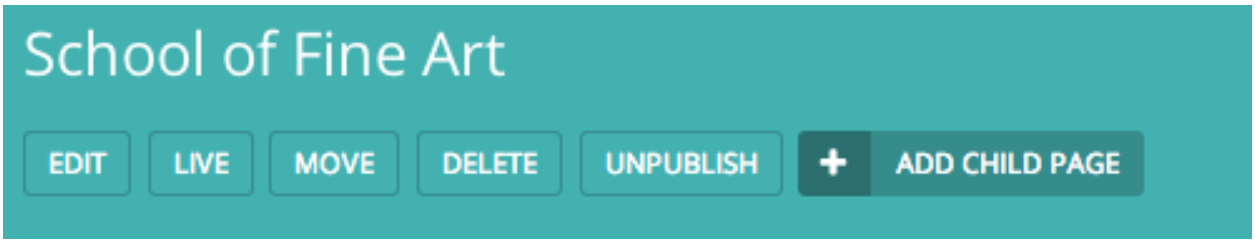
- To add further child pages press the Add child page button below the parent page title. You can view the parent page on the live site by pressing the View live button. The Move button will take you to the Move page screen where you can reposition the page and all its child pages in the site structure.
- Similar buttons are available for each child page. These are made visible on hover.

Reordering pages



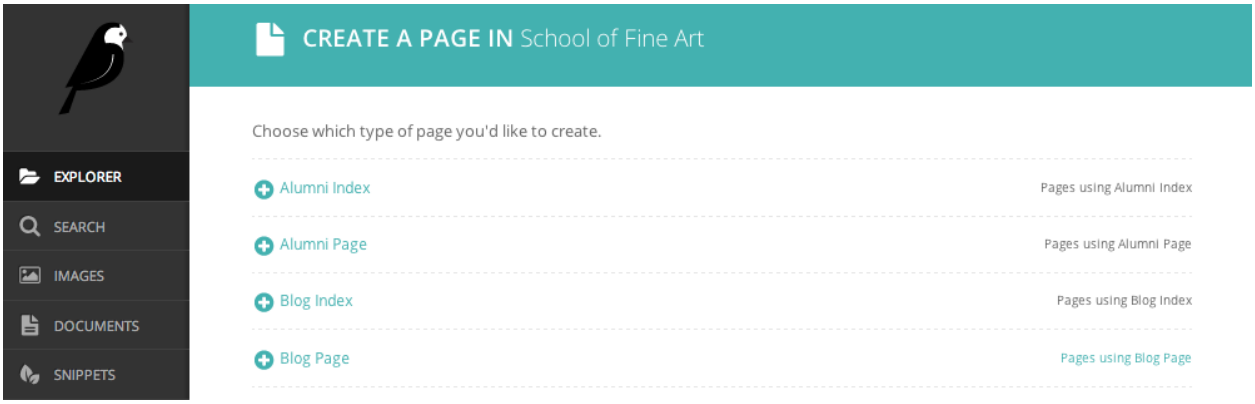
- Clicking the icon to the far left of the child pages table will enable the reordering handles. This allows you to reorder the way that content displays in the main menu of your website.
- Reorder by dragging the pages by the handles on the far left (the icon made up of 8 dots).
- Your new order will be automatically saved each time you drag and drop an item.

Creating new pages

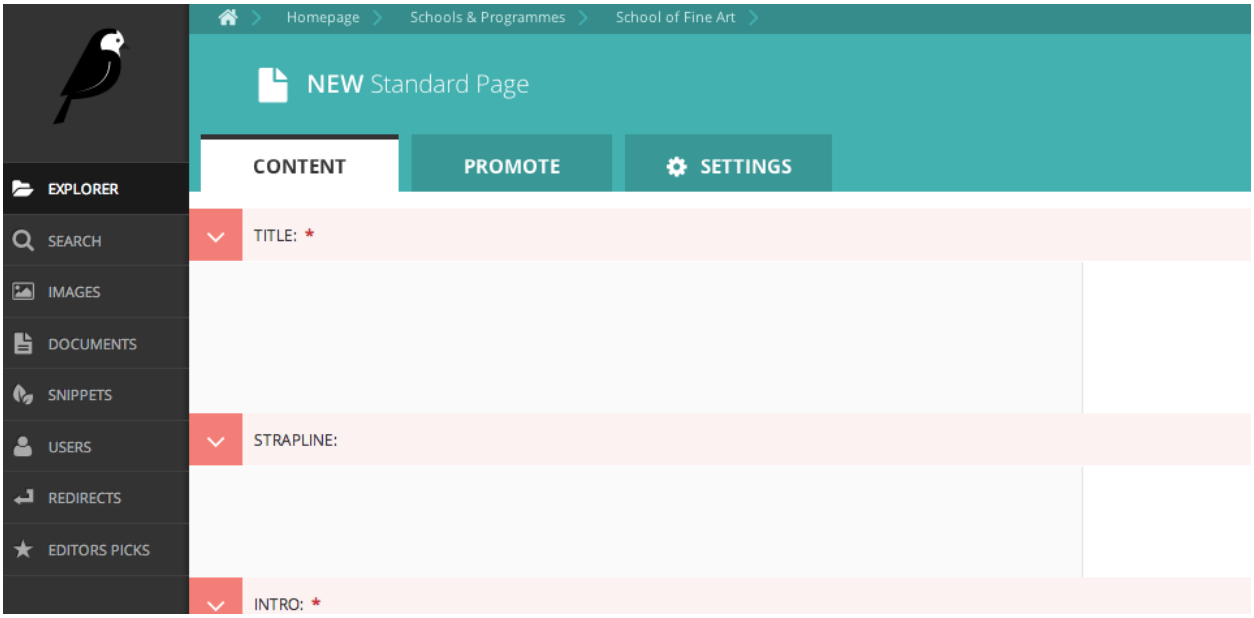


Create new pages by clicking the Add child page. This creates a child page of the section you are currently in. In this case a child page of the Homepage.

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 *View all . . . pages* 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.



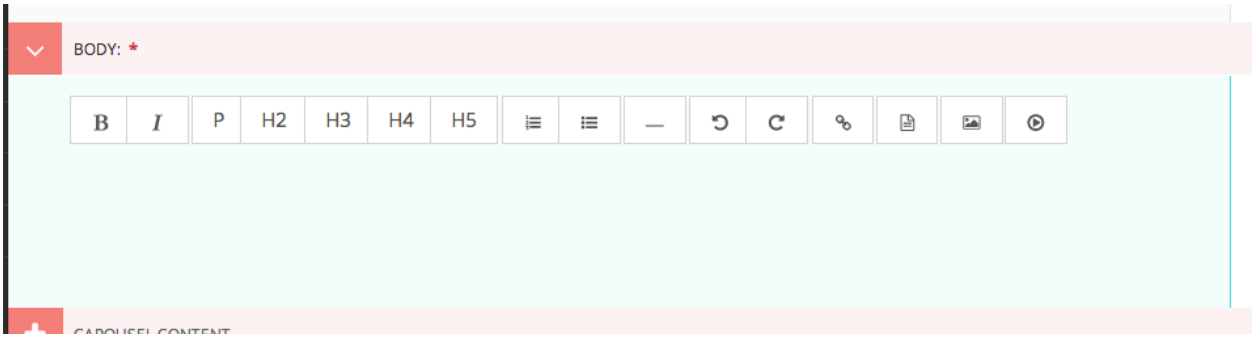
- Once you've selected a page type you will be presented with a blank New page screen.
- Click into the areas below each field's heading to start entering content.

Creating page body content

The majority of the body content within a page will be created within text fields.

There are two types of text entry fields you will see when creating a page. Some, like *Page title* are basic text fields come with no formatting options. This is because all titles display the same across a single page type. Others, *Body* for example, require more formatting, and so a set of tools are provided for this purpose. These are referred to as rich text fields.

So, when you click into certain fields, for example the *Body* field, 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 seperate 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>*.

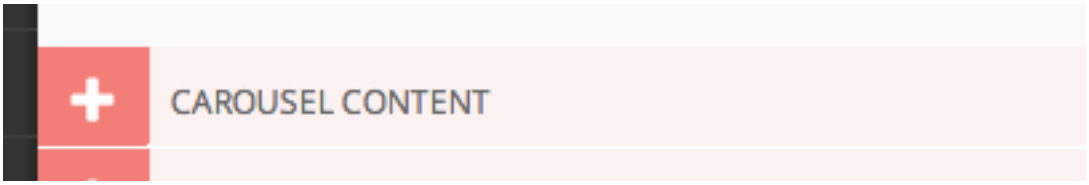
Inserting images and videos in a page

There will obviously be many instances in which you will want to add images to a page. There may be multiple ways in which to add an image to a page, depending on the setup of the site that you work on. For example, the Wagtail demo standard page type has two ways to insert images.

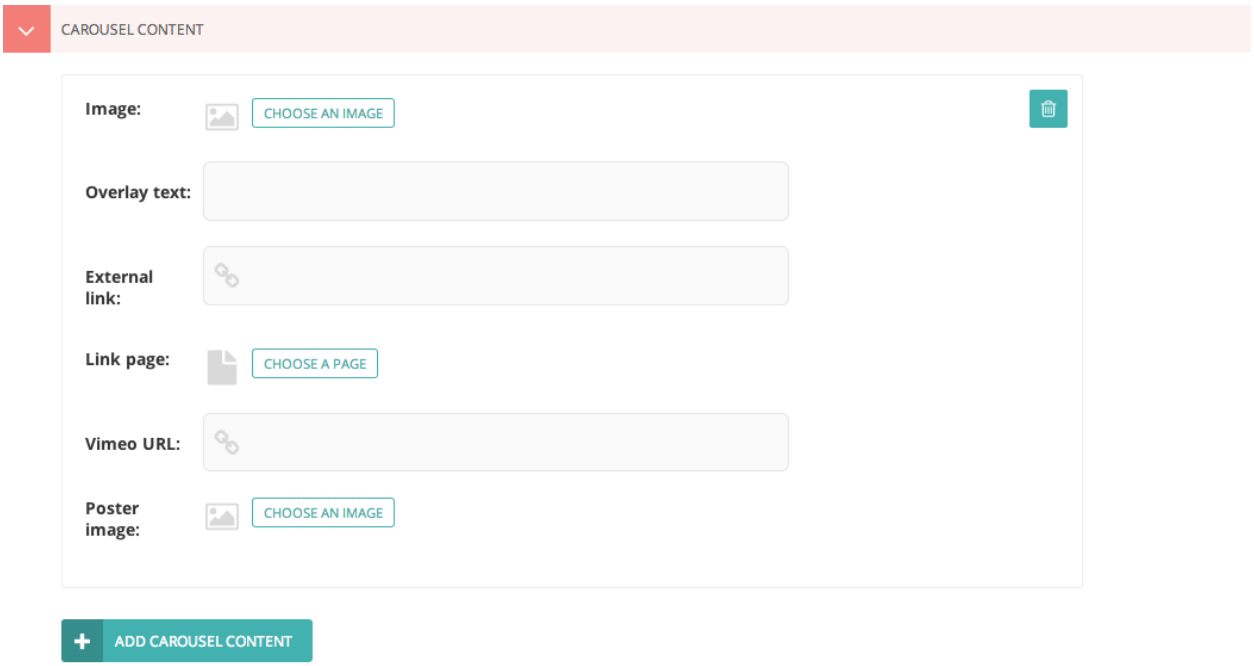
- In the main carousel, or. . .
- Within the body of the page

Inserting images into the carousel

The carousel is where the main, featured images and videos associated with a page should be displayed.



- To insert a carousel item click the Add carousel content link in the Carousel content section.



- You can then insert an image by clicking the *Choose an image* button.
- It is also possible to add videos to a carousel. Simply copy and paste the web address for the video (either YouTube or Vimeo) into the *Embed URL* field and click Insert. A poster image for the video can also be uploaded or selected from the CMS. This is the image displayed before a user has clicked play on the video.
- The *Caption* field allows you to enter a caption to be displayed with your image. This caption also acts as the 'Alternative text' or Alt text for your image. This is a small piece of invisible code that is used by screen readers that enable visually impaired users to hear a description of the image.
- The external link field allows you to enter a web address for pages not within your website.
- Or you can select an internal page using the page chooser (see below for info on the page chooser).
- You can add more items into the same carousel by clicking the Add carousel content link again. Please see Adding multiple items section below for help with removing or ordering carousel items.

Choosing an image to insert

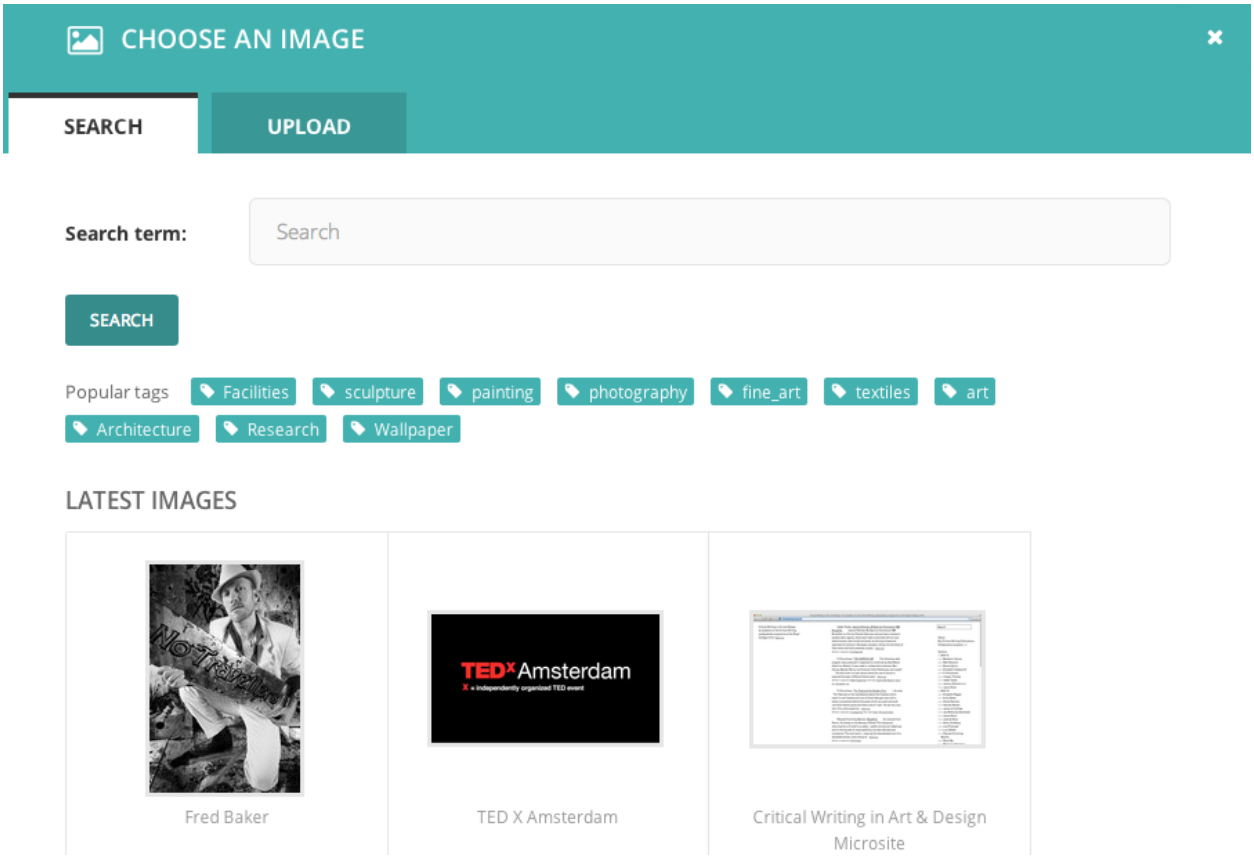
You have two options when selecting an image to insert:

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.

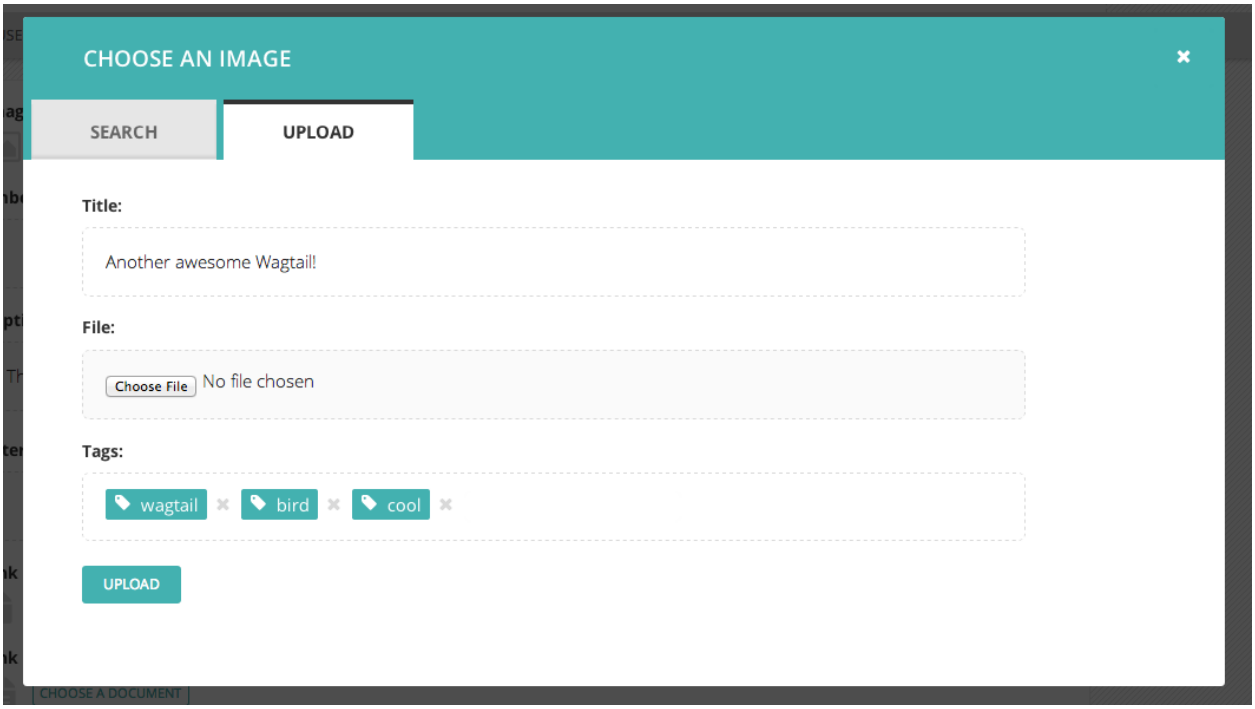
Choosing an image from the image library

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

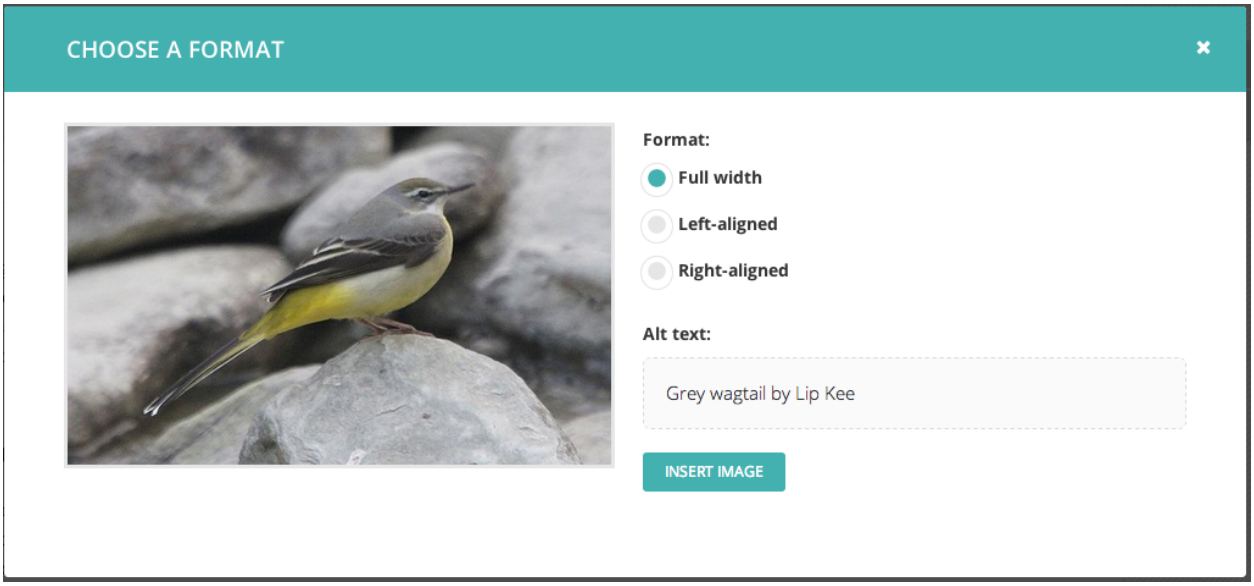


1. You must include an image title for your uploaded image
2. Click the *Choose file* button to choose an image from your computer.
3. This *Tags* allows you to associate tags with the image you are uploading. This allows them to be more easily found when searching. Each tag should be separated by a 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 into the body text

Images can also be inserted into the body text of a page. When editing the Body field of a standard page, click the image illustrated above. You will then be presented with the same options as for inserting images into the main carousel.

In addition, the Wagtail Demo site allows you to chose an aligmnent for you image.



1. You can select how the image is displayed by selecting one of the format options.
2. In the Wagtail Demo site, images inserted into the body text do not have embeded captions (these should be added as regular body text). However you must still provide specific alt text for your image.

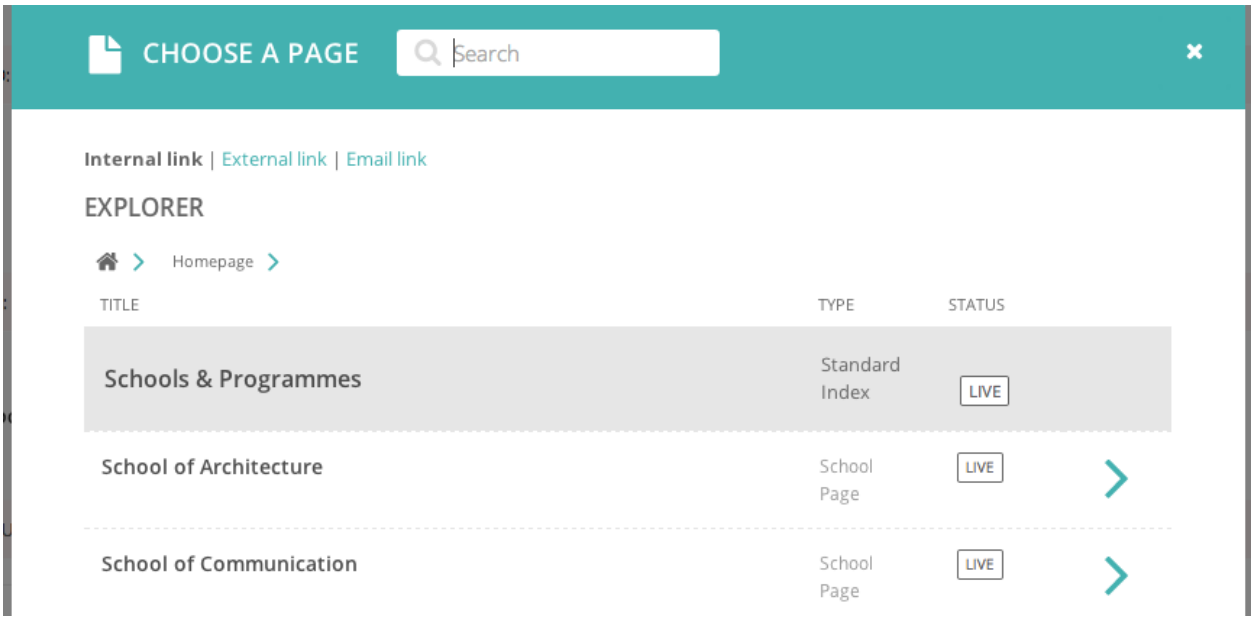
The alignments available are described below:

- **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.

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.

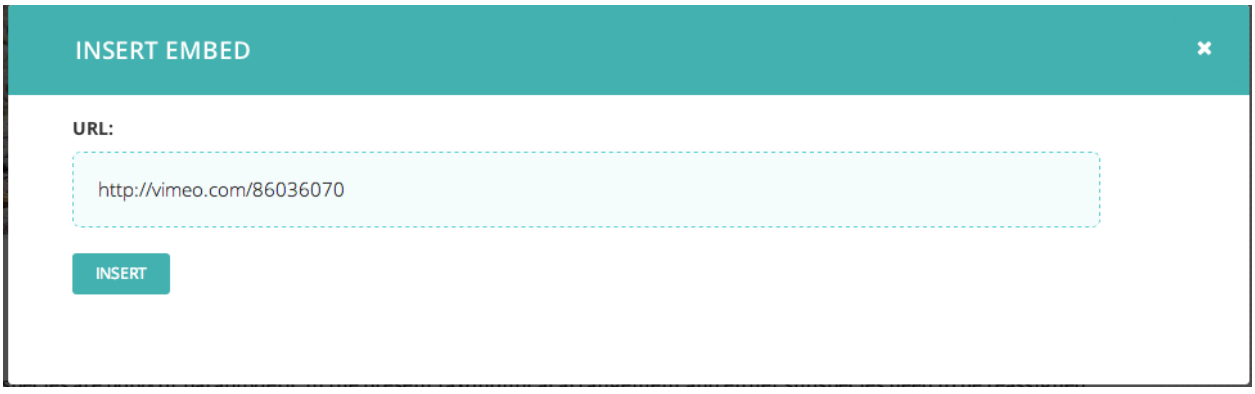
Whichever way you insert a link, you will be presented with the form displayed below.



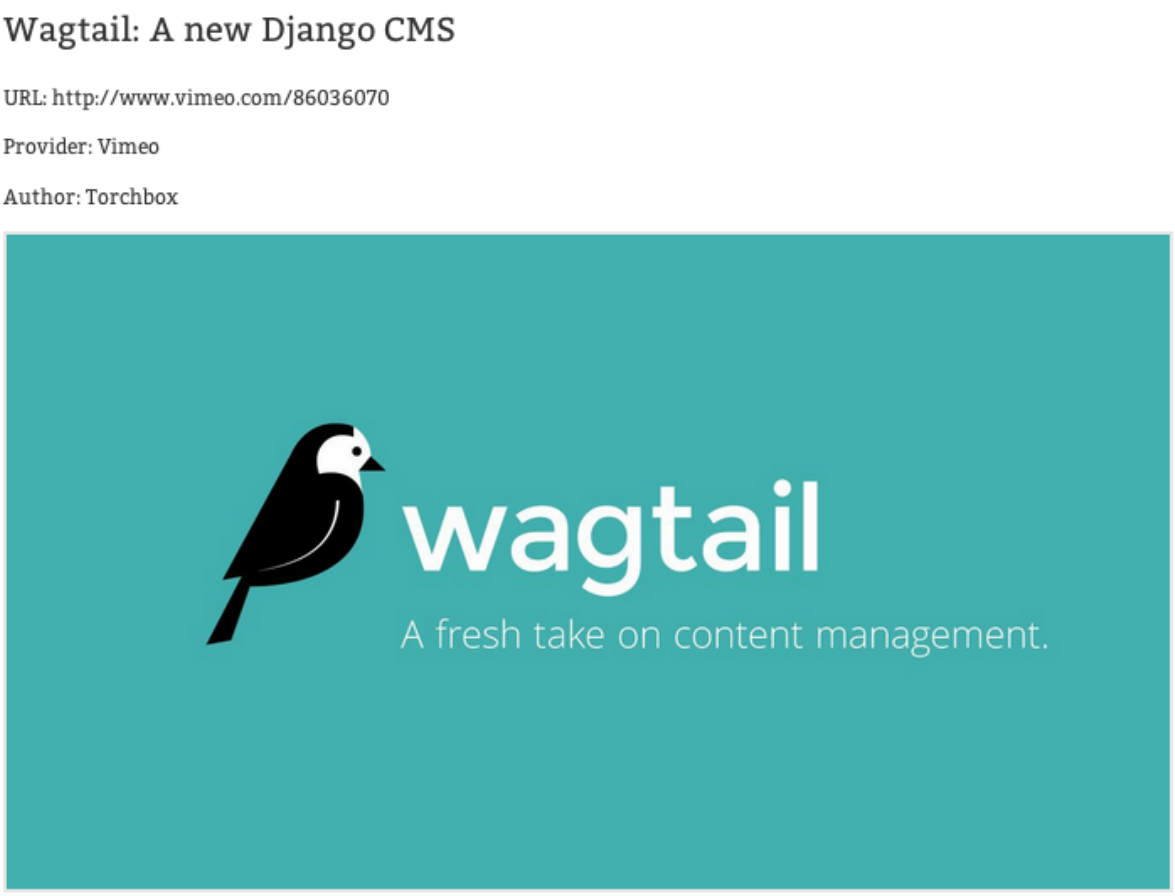
- Search for an existing page to link to using the search bar at the top of the pop-up.
- Below the search bar you can select the type of link you want to insert. The following types are available:
  - Internal link: A link to an existing page within the RCA website.
  - External link: A link to a page on another website.
  - Email link: A link that will open the users 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.



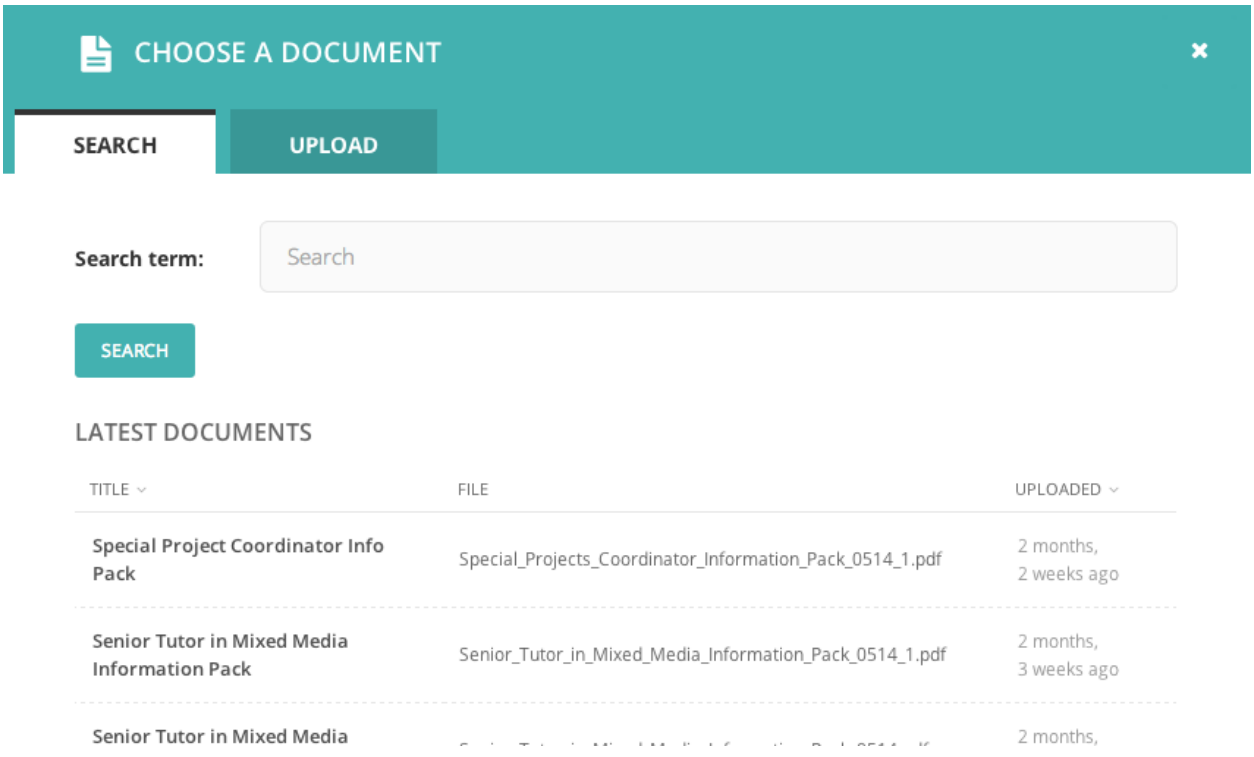
- A placeholder with the name of the video and a screenshot will be inserted into the text area. Clicking the X in the top corner will remove the video.

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 Body 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.

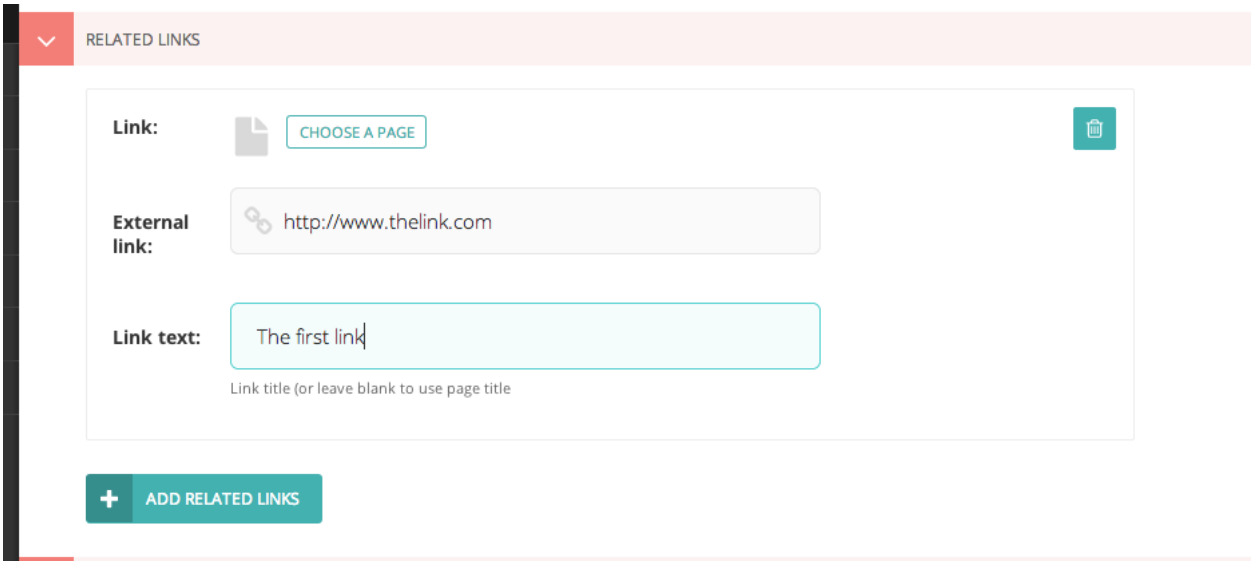


Adding multiple items

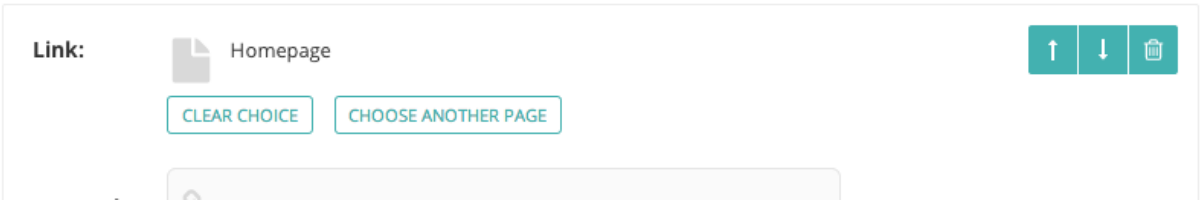
A common feature of Wagtail is the ability to add more than one of a particular type of field or item. For example, you can add as many carousel items or related links as you wish.



- 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.



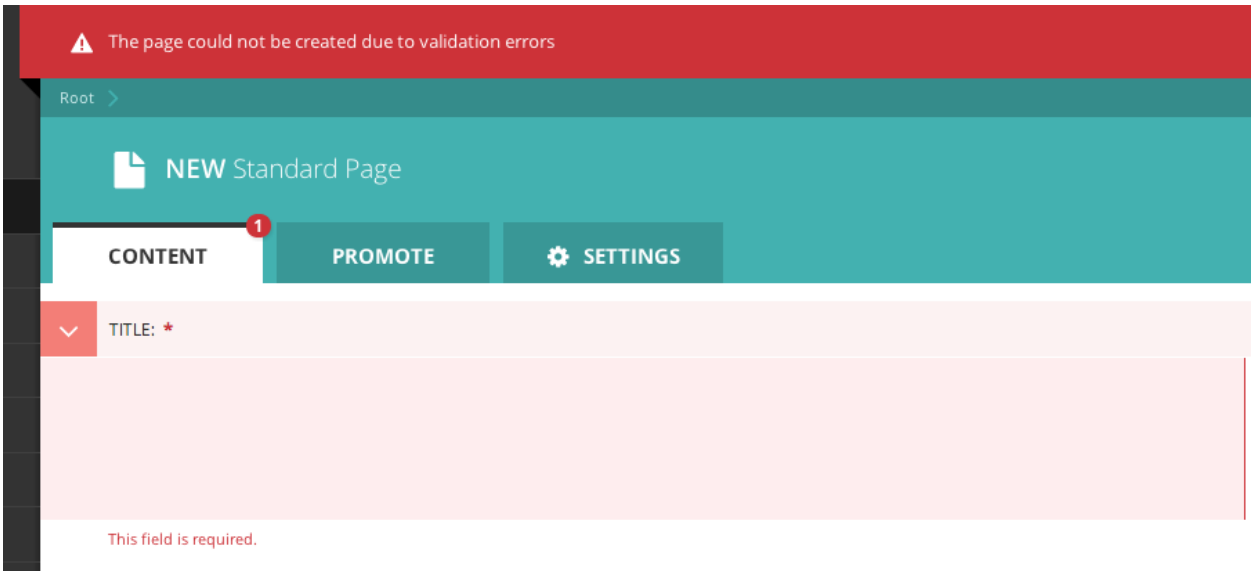
4. You can reorder your multiple items using the up and down arrows. Doing this will affect the order in which they are display on the live page.

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 the possible fields in the promote tab and what they do.

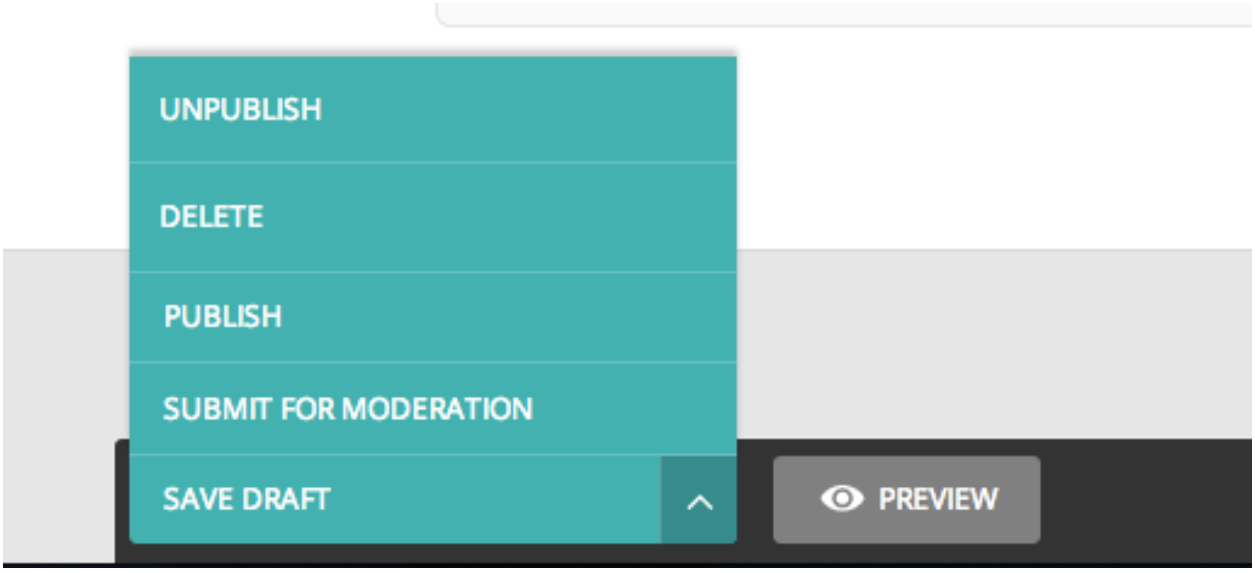
- **Page title:** An optional, search-engine friendly page title. This is the title that appears in the tab of your browser window. It is also the title that would appear in a search engine if the page was returned as part of a set of search results.
- **Slug:** The last part of the web address for the 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 autogenerated 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 (-).
- **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 usefull to distinguish between similarly named pages.
- **Feed image:** This is the image displayed in content feeds, such as the thumbnails on the blog listing page. This optional image is most often used when the primary image on a page is in a format that would not display well in a content feed.

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)

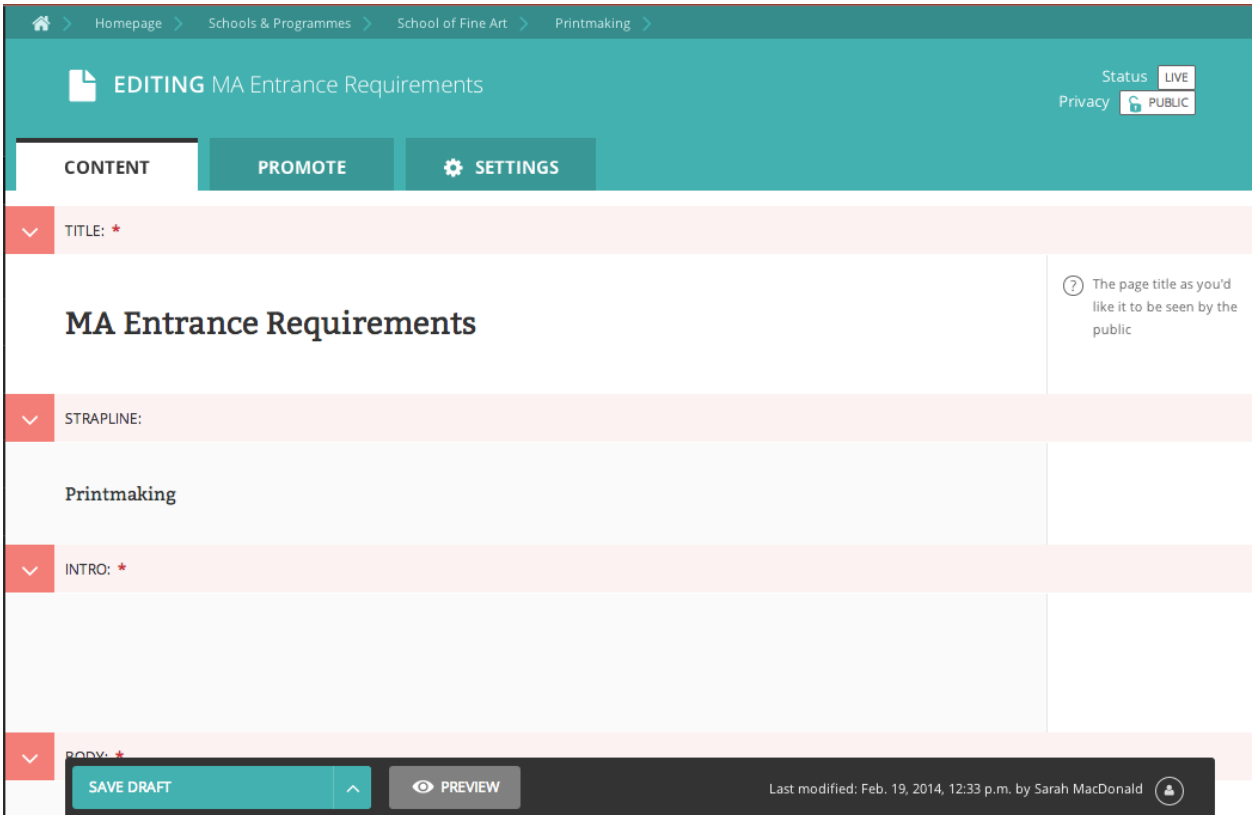
- **Preview:** Opens a new window displaying the page as it would look if published, but does not save your changes or submit the page for moderation. (all roles)
- **Publish/Unpublish:** Clicking either the *Publish* or *Unpublish* buttons will take you to a confirmation screen asking you to confirm 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.



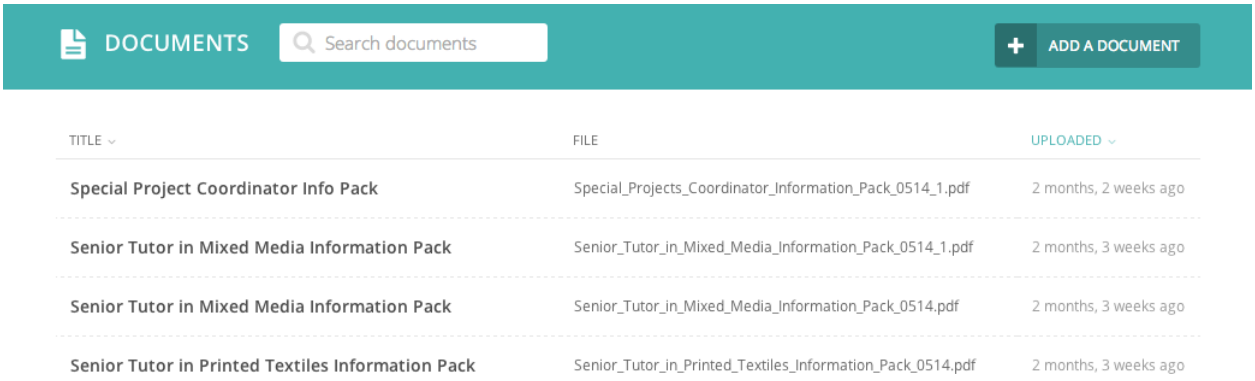
- When editing an existing page the title of the page being edited is displayed at the top of the page.
- The current status of the page is displayed in the top-right.
- You can change the title of the page by clicking into the title field.
- When you are typing into a field, help text is often displayed on the right-hand side of the screen.

Managing documents and images

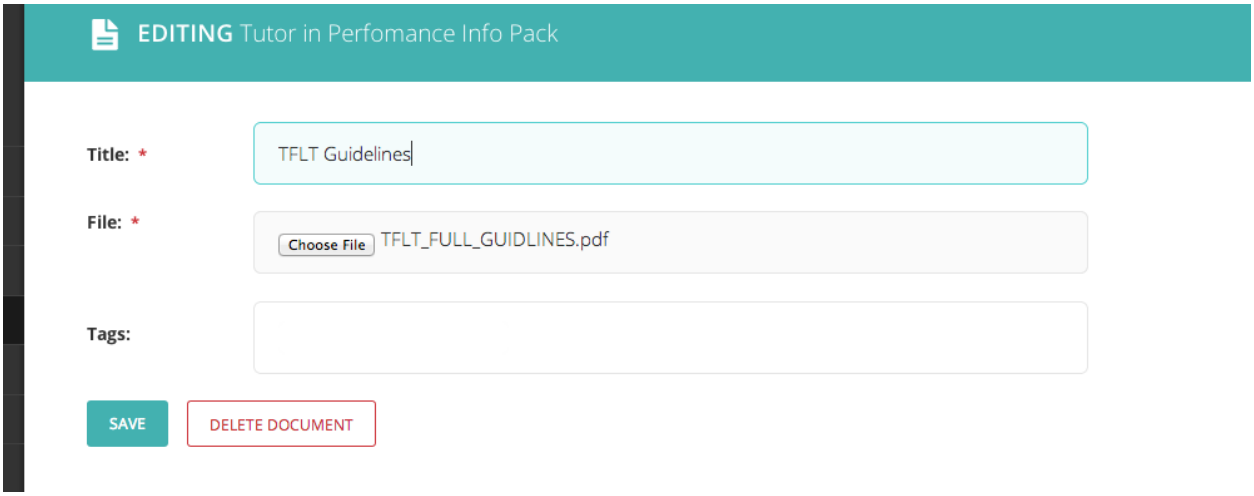
Wagtail allows you to manage all of your documents and images through their own dedicated interfaces. See below for information on each of these elements.

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.



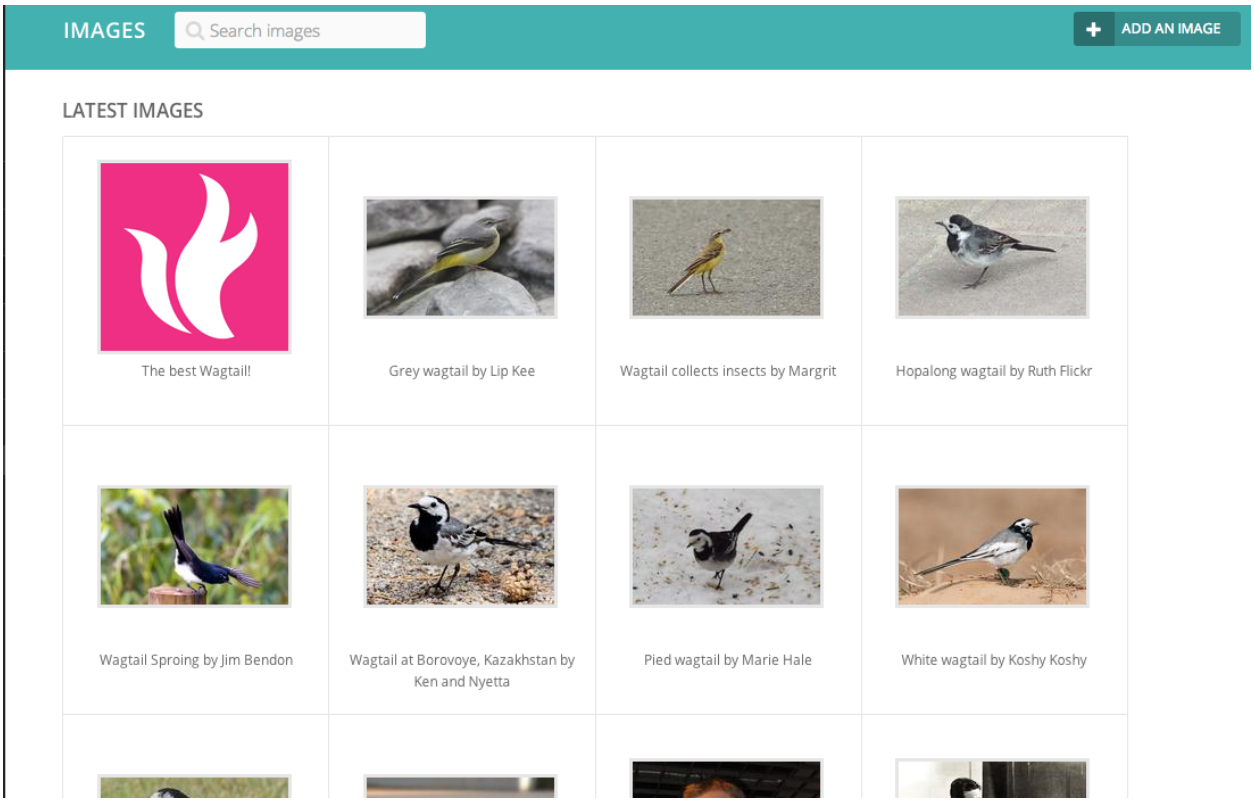
- Add documents by clicking the *Add document* button in the top-right.
- Search for documents in the CMS by entering your search term in the search bar. The results will be automatically updated as you type.
- You can also filter the results by *Popular tags*. Click on a tag to update the search results listing.
- Edit the details of a document by clicking the document title.



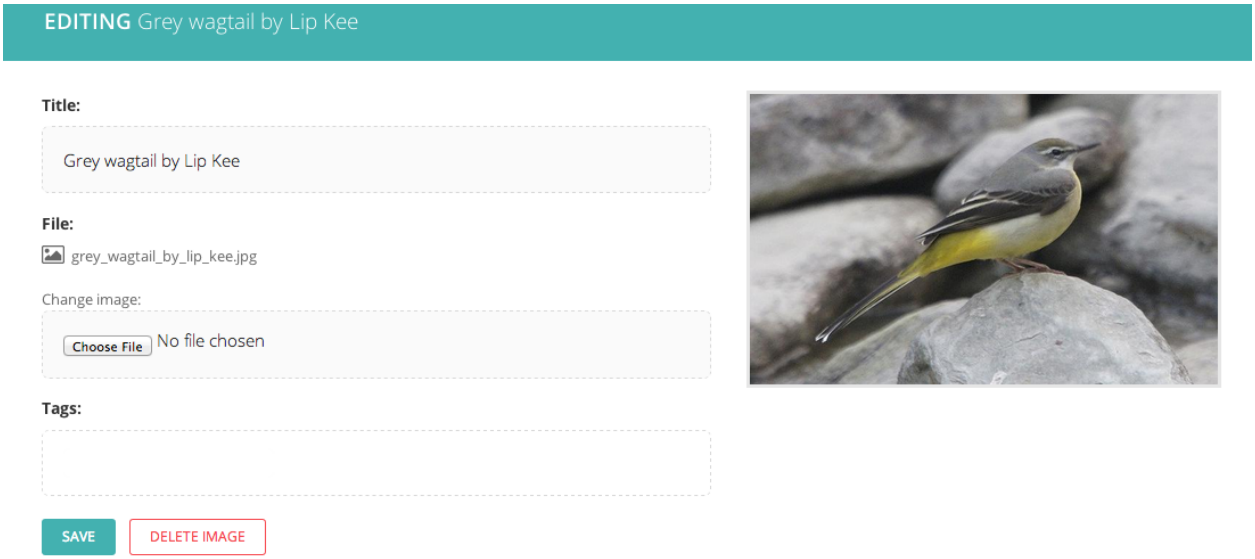
- When editing a document you can replace the file associated with that document record. This means you can update documents without having to update the pages on which they are placed. Changing the file will change it on all pages that use the document.
- Add or remove tags using the Tags field.
- Save or delete documents using the buttons at the bottom of the interface. **NOTE:** 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. **NOTE:** 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.



- 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. Changing the file will change it on all pages that use the image.

Snippets

Note: Documentation currently incomplete and in draft status

When creating a page on a website, it is a common occurance to want to add in a piece of content that already exists on another page. An example of this would be a person's contact details, or an advert that you want to simply show on every page of your site, without having to manually apply it.

Wagtail makes this easy with Snippets.

CHAPTER 8

Release notes

Roadmap

The story so far

Wagtail was developed for the Royal College of Art in 2013, and launched as an open source project in February 2014. The changes since its launch are recorded in the codebase:

https://raw.github.com/torchbox/wagtail/master/CHANGELOG.txt

In summary:

- February 2014: Reduced dependencies, basic documentation, translations, tests

What's next

The issue list gives a detailed view of the immediate tasks, but our current broad priorities are:

- More and better tests (>80% coverage)
- Better documentation: simple setup guides for all levels of user, a manual for editors and administrators, in-depth intstructions for Django developers.
- Move site section permissions out of Django admin
- Improved image handling: intelligent cropping, animated gif support
- Block-level editing UI (see Sir Trevor)
- Site settings management
- Support for an HTML content type
- Simple inline stats

You decide

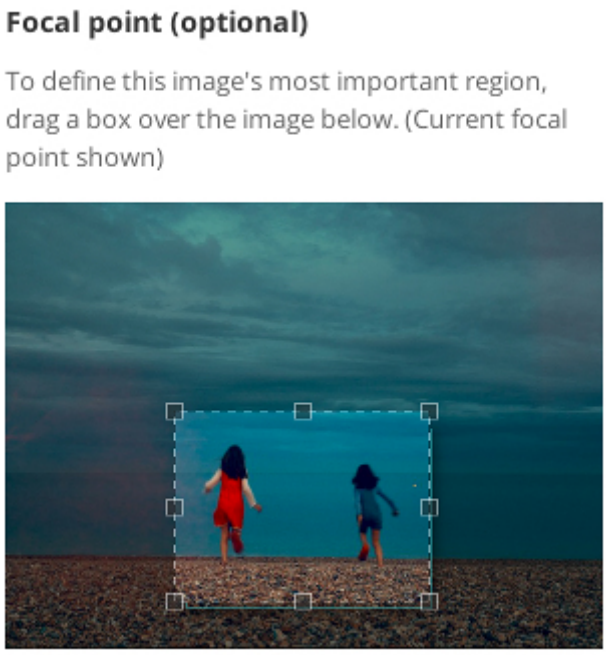
Please help us focus on the right things by raising issues for new features, or joining the discussion on existing issues.

Wagtail 0.7 release notes

- *What's new*
- *Upgrade considerations*

What's new

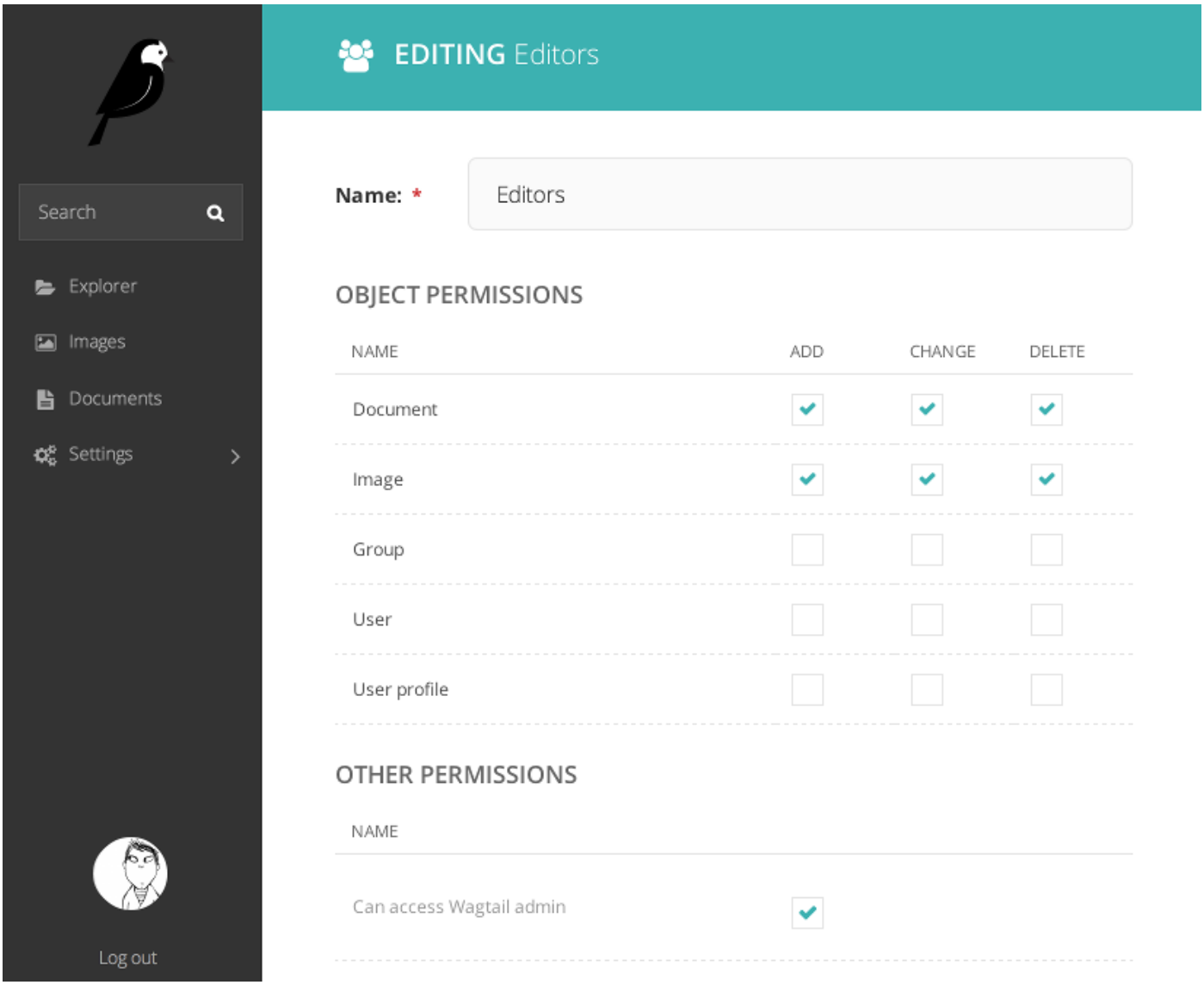
New interface for choosing image focal point



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.


EDITING Editors
Search
Explorer
Images
Documents
Settings
Log out
Name: *
Editors
OBJECT PERMISSIONS
NAME
ADD
CHANGE
DELETE
Document
Image
Group
User
User profile
OTHER PERMISSIONS
NAME
Can access Wagtail admin

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 invalidation*
- 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 *Admin 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.

Embedding URL configuration in Pages

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 generation

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 invalidation

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.models, 42
wagtail.wagtailcore.models, 15
wagtail.wagtailcore.query, 37

Index

A

ancestor_of() (wagtail.wagtailcore.query.PageQuerySet method), 39

B

base.py, **8**

C

child_of() (wagtail.wagtailcore.query.PageQuerySet method), 39
created_at, **29**

D

descendant_of() (wagtail.wagtailcore.query.PageQuerySet method), 38
dev.py, **8**
dj, **9**
djrun, **9**

F

file_extension, **29**
filename, **29**
fill, **45**
full_url (wagtail.wagtailcore.models.Page attribute), 15

G

get_ancestors() (wagtail.wagtailcore.models.Page method), 16
get_context() (wagtail.wagtailcore.models.Page method), 16
get_descendants() (wagtail.wagtailcore.models.Page method), 16
get_siblings() (wagtail.wagtailcore.models.Page method), 16
get_template() (wagtail.wagtailcore.models.Page method), 16
get_verbose_name() (wagtail.wagtailcore.models.Page class method), 15

H

height, **45**

I

in_menu() (wagtail.wagtailcore.query.PageQuerySet method), 38
is_navigable() (wagtail.wagtailcore.models.Page method), 16

L

live() (wagtail.wagtailcore.query.PageQuerySet method), 37
local.py, **8**

M

max, **44**
min, **44**

N

not_ancestor_of() (wagtail.wagtailcore.query.PageQuerySet method), 39
not_descendant_of() (wagtail.wagtailcore.query.PageQuerySet method), 38
not_live() (wagtail.wagtailcore.query.PageQuerySet method), 38
not_page() (wagtail.wagtailcore.query.PageQuerySet method), 38

O

Original, **47**
original, **45**

P

Page (class in wagtail.wagtailcore.models), 15
page() (wagtail.wagtailcore.query.PageQuerySet method), 38
PageQuerySet (class in wagtail.wagtailcore.query), 37

password_required_template (wagtail.wagtailcore.models.Page attribute), 17
preview_modes (wagtail.wagtailcore.models.Page attribute), 16
production.py, **8**
public() (wagtail.wagtailcore.query.PageQuerySet method), 39

R

relative_url() (wagtail.wagtailcore.models.Page method), 16
Resize to fill, **47**
Resize to height, **47**
Resize to max, **47**
Resize to min, **47**
Resize to width, **47**
resolve_subpage() (wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin method), 43
reverse_subpage() (wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin method), 43
RoutablePageMixin (class in wagtail.contrib.wagtailroutablepage.models), 42
routablepageurl() (in module wagtail.contrib.wagtailroutablepage.templatetags.wagtailroutablepage_tags), 43
route() (wagtail.wagtailcore.models.Page method), 16

S

search() (wagtail.wagtailcore.models.Page class method), 16
search() (wagtail.wagtailcore.query.PageQuerySet method), 40
search_fields (wagtail.wagtailcore.models.Page attribute), 16
serve() (wagtail.wagtailcore.models.Page method), 16
serve_preview() (wagtail.wagtailcore.models.Page method), 16
sibling_of() (wagtail.wagtailcore.query.PageQuerySet method), 39
specific (wagtail.wagtailcore.models.Page attribute), 15
specific_class (wagtail.wagtailcore.models.Page attribute), 15
subpage_types (wagtail.wagtailcore.models.Page attribute), 16
subpage_urls (wagtail.contrib.wagtailroutablepage.models.RoutablePageMixin attribute), 42

T

tags, **29**
title, **29**
type() (wagtail.wagtailcore.query.PageQuerySet method), 40

U

unpublish() (wagtail.wagtailcore.query.PageQuerySet method), 40
url, **29**
url (wagtail.wagtailcore.models.Page attribute), 15

W

wagtail.contrib.wagtailroutablepage.models (module), 42
wagtail.wagtailcore.models (module), 15
wagtail.wagtailcore.query (module), 37
width, **44**