
Wagtail Documentation

Release 4.1a0

Torchbox

Sep 22, 2022

CONTENTS

1 Index	3
1.1 Getting started	3
1.2 Usage guide	29
1.3 Advanced topics	94
1.4 Extending Wagtail	199
1.5 Reference	254
1.6 Support	455
1.7 Using Wagtail: an Editor’s guide	456
1.8 Contributing to Wagtail	510
1.9 Release notes	539
Python Module Index	747
Index	749

Wagtail is an open source CMS written in [Python](#) and built on the [Django](#) web framework.

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

If you'd like to get a quick feel for Wagtail, try spinning up a [temporary developer environment](#) in your browser (running on Gitpod - here's [how it works](#)).

- **First steps**

- [*Getting started*](#)
- [*Your first Wagtail site*](#)
- [*Demo site*](#)

- **Using Wagtail**

- [*Page models*](#)
- [*Writing templates*](#)
- [*How to use images in templates*](#)
- [*Search*](#)
- [*Third-party tutorials*](#)

- **For editors**

- [*Editors guide*](#)

1.1 Getting started

Note: These instructions assume familiarity with virtual environments and the [Django web framework](#). For more detailed instructions, see [Your first Wagtail site](#). To add Wagtail to an existing Django project, see [Integrating Wagtail into a Django project](#).

1.1.1 Dependencies needed for installation

- Python 3
- **libjpeg** and **zlib**, libraries required for Django’s **Pillow** library. See Pillow’s platform-specific installation instructions.

1.1.2 Quick install

Run the following in a virtual environment of your choice:

```
$ pip install wagtail
```

(Installing outside a virtual environment may require `sudo`.)

Once installed, Wagtail provides a command similar to Django’s `django-admin startproject` to generate a new site/project:

```
$ wagtail start mysite
```

This will create a new folder `mysite`, based on a template containing everything you need to get started. More information on that template is available in [the project template reference](#).

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

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

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

This will set you up with a new stand-alone Wagtail project. If you'd like to add Wagtail to an existing Django project instead, see [Integrating Wagtail into a Django project](#).

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

- [Elasticsearch](#).
- [Feature Detection](#).

Your first Wagtail site

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

Install and run Wagtail

Install dependencies

Wagtail supports Python 3.7, 3.8, 3.9 and 3.10.

To check whether you have an appropriate version of Python 3:

```
$ python3 --version
```

If this does not return a version number or returns a version lower than 3.7, you will need to [install Python 3](#).

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

Create and activate a virtual environment

We recommend using a virtual environment, which isolates installed dependencies from other projects. This tutorial uses **venv**, which is packaged with Python 3.

On Windows (cmd.exe):

```
> python3 -m venv mysite\env  
> mysite\env\Scripts\activate.bat
```

On GNU/Linux or MacOS (bash):

```
$ python3 -m venv mysite/env  
$ source mysite/env/bin/activate
```

For other shells see the [venv](#) documentation.

Note: If you’re using version control (such as git), `mysite` will be the directory for your project. The `env` directory inside of it should be excluded from any version control.

Install Wagtail

Use pip, which is packaged with Python, to install Wagtail and its dependencies:

```
$ pip install wagtail
```

Generate your site

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

Because the folder `mysite` was already created by `venv`, run `wagtail start` with an additional argument to specify the destination directory:

```
$ wagtail start mysite mysite
```

Note: Generally, in Wagtail, each page type, or content type, is represented by a single app. However, different apps can be aware of each other and access each other’s data. All of the apps need to be registered within the `INSTALLED_APPS` section of the `settings` file. Look at this file to see how the `start` command has listed them in there.

Install project dependencies

```
$ cd mysite
$ pip install -r requirements.txt
```

This ensures that you have the relevant versions of Wagtail, Django, and any other dependencies for the project you have just created.

Create the database

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

```
$ python manage.py migrate
```

This command ensures that the tables in your database are matched to the models in your project. Every time you alter your model (for example you may add a field to a model) you will need to run this command in order to update the database.

Create an admin user

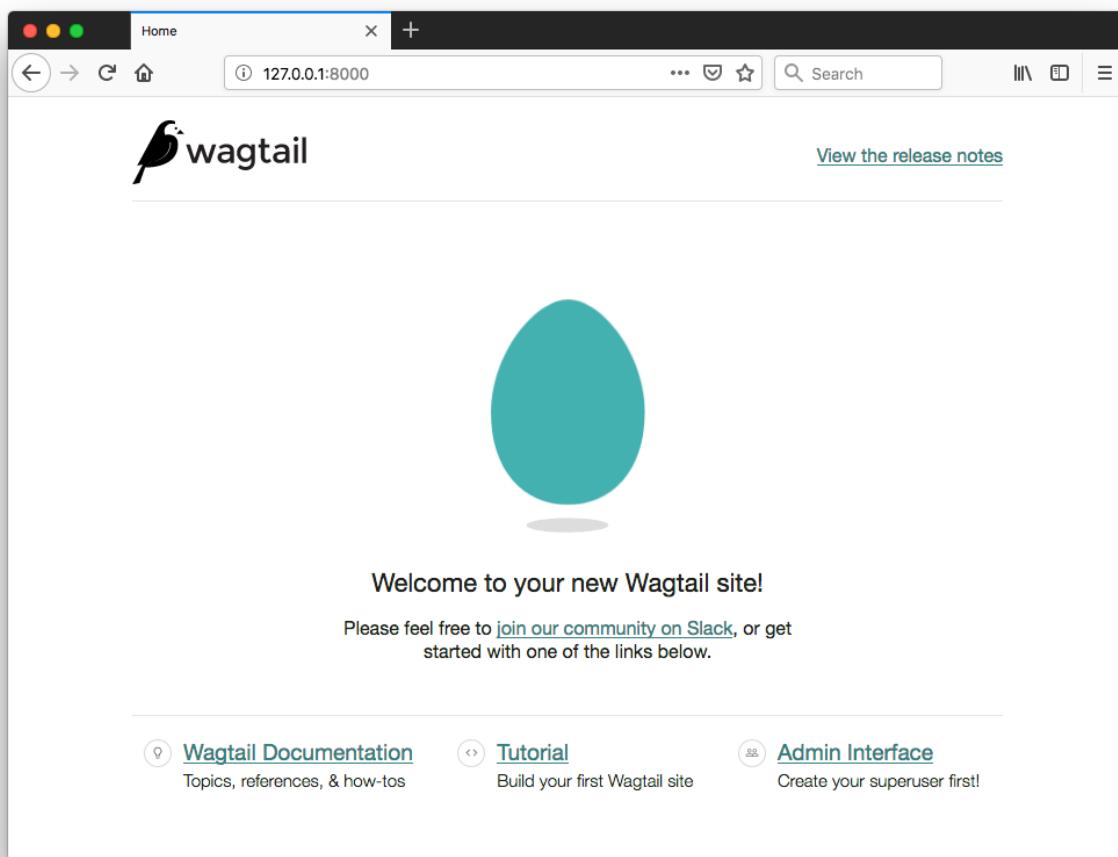
```
$ python manage.py createsuperuser
```

When logged into the admin site, a superuser has full permissions and is able to view/create/manage the database.

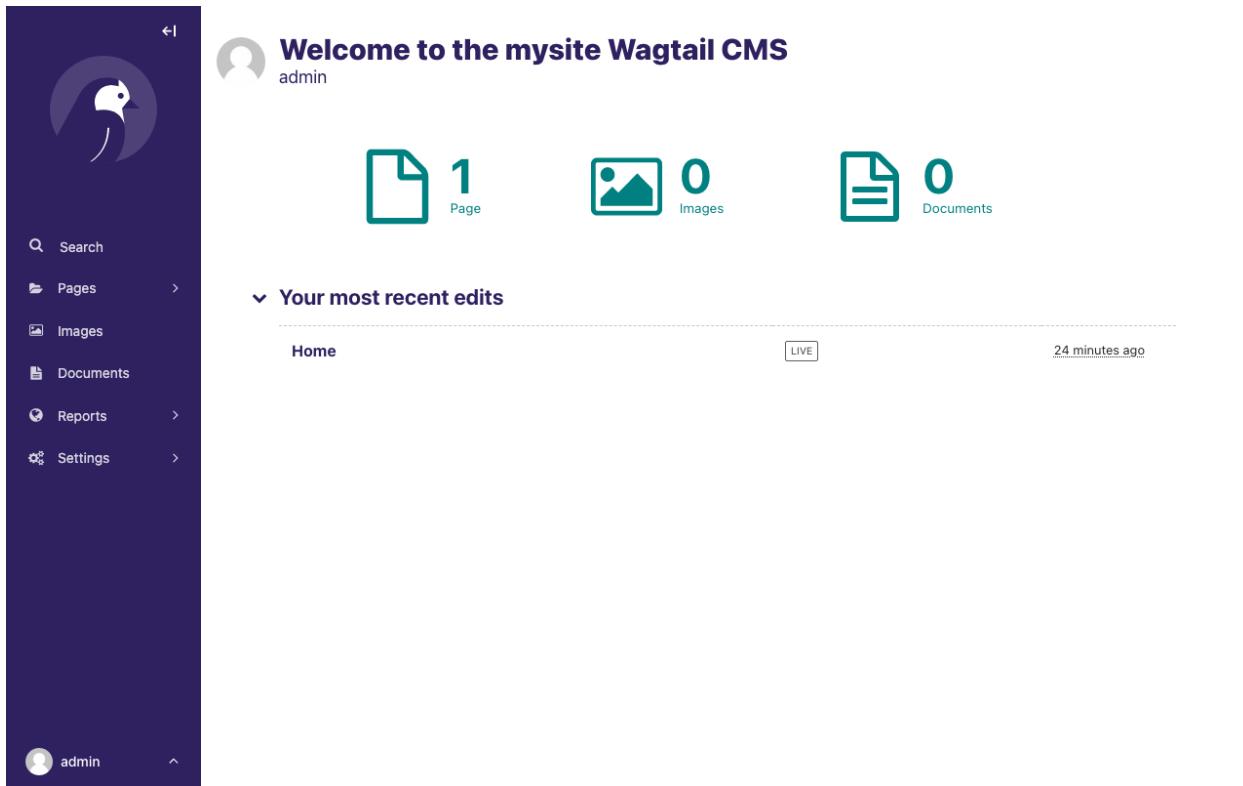
Start the server

```
$ python manage.py runserver
```

If everything worked, <http://127.0.0.1:8000> will show you a welcome page:



You can now access the administrative area at <http://127.0.0.1:8000/admin>



Extend the HomePage model

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

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

```
from django.db import models

from wagtail.models import Page
from wagtail.fields import RichTextField
from wagtail.admin.panels import FieldPanel


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

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

`body` is defined as `RichTextField`, a special Wagtail field. When `blank=True`, it means that this field is not required and can be empty. You can use any of the [Django core fields](#). `content_panels` define the capabilities and the layout of the editing interface. When you add fields to `content_panels`, it enables them to be edited on the Wagtail interface. [More on creating Page models](#).

Run `python manage.py makemigrations` (this will create the migrations file), then `python manage.py migrate` (this executes the migrations and updates the database with your model changes). You must run the above commands each time you make changes to the model definition.

You can now edit the homepage within the Wagtail admin area (go to Pages, Homepage, then Edit) to see the new body field. Enter some text into the body field, and publish the page by selecting *Publish* at the bottom of the page editor, rather than *Save Draft*.

The page template now needs to be updated to reflect the changes made to the model. Wagtail uses normal Django templates to render each page type. By default, it will look for a template filename formed from the app and model name, separating capital letters with underscores (for example HomePage within the ‘home’ app becomes home/home_page.html). This template file can exist in any location recognised by [Django’s template rules](#); conventionally it is placed under a `templates` folder within the app.

Edit `home/templates/home/home_page.html` to contain the following:

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

{% load wagtailcore_tags %}

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

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

`base.html` refers to a parent template and must always be the first template tag used in a template. Extending from this template saves you from rewriting code and allows pages across your app to share a similar frame (by using block tags in the child template, you are able to override specific content within the parent template).

`wagtailcore_tags` must also be loaded at the top of the template and provide additional tags to those provided by Django.

Welcome to our new site!



Wagtail template tags

In addition to Django's template tags and filters, Wagtail provides a number of its own *template tags & filters* which can be loaded by including `{% load wagtailcore_tags %}` at the top of your template file.

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

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

Produces:

```
<p><b>Welcome</b> to our new site!</p>
```

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

A basic blog

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

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

Blog Index and Posts

Lets start with a simple index page for our blog. In `blog/models.py`:

```
from wagtail.models import Page
from wagtail.fields import RichTextField
from wagtail.admin.panels import FieldPanel

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

    content_panels = Page.content_panels + [
        FieldPanel('intro')
    ]
```

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

Since the model is called `BlogIndexPage`, the default template name (unless we override it) will be `blog/templates/blog/blog_index_page.html`. Create this file with the following content:

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

{% load wagtailcore_tags %}

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

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

(continues on next page)

(continued from previous page)

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

{% for post in page.get_children %}
    <h2><a href="{% pageurl post %}">{{ post.title }}</a></h2>
    {{ post.specific.intro }}
    {{ post.specific.body|richtext }}
{% endfor %}

{% endblock %}
```

Most of this should be familiar, but we'll explain `get_children` a bit later. Note the `pageurl` tag, which is similar to Django's `url` tag but takes a Wagtail Page object as an argument.

In the Wagtail admin, create a `BlogIndexPage` as a child of the Homepage, make sure it has the slug "blog" on the Promote tab, and publish it. You should now be able to access the url `/blog` on your site (note how the slug from the Promote tab defines the page URL).

Now we need a model and template for our blog posts. In `blog/models.py`:

```
from django.db import models

from wagtail.models import Page
from wagtail.fields import RichTextField
from wagtail.admin.panels import FieldPanel
from wagtail.search import index

# Keep the definition of BlogIndexPage, and add:

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

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

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

In the model above, we import `index` as this makes the model searchable. You can then list fields that you want to be searchable for the user.

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

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

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

{% load wagtailcore_tags %}

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

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

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

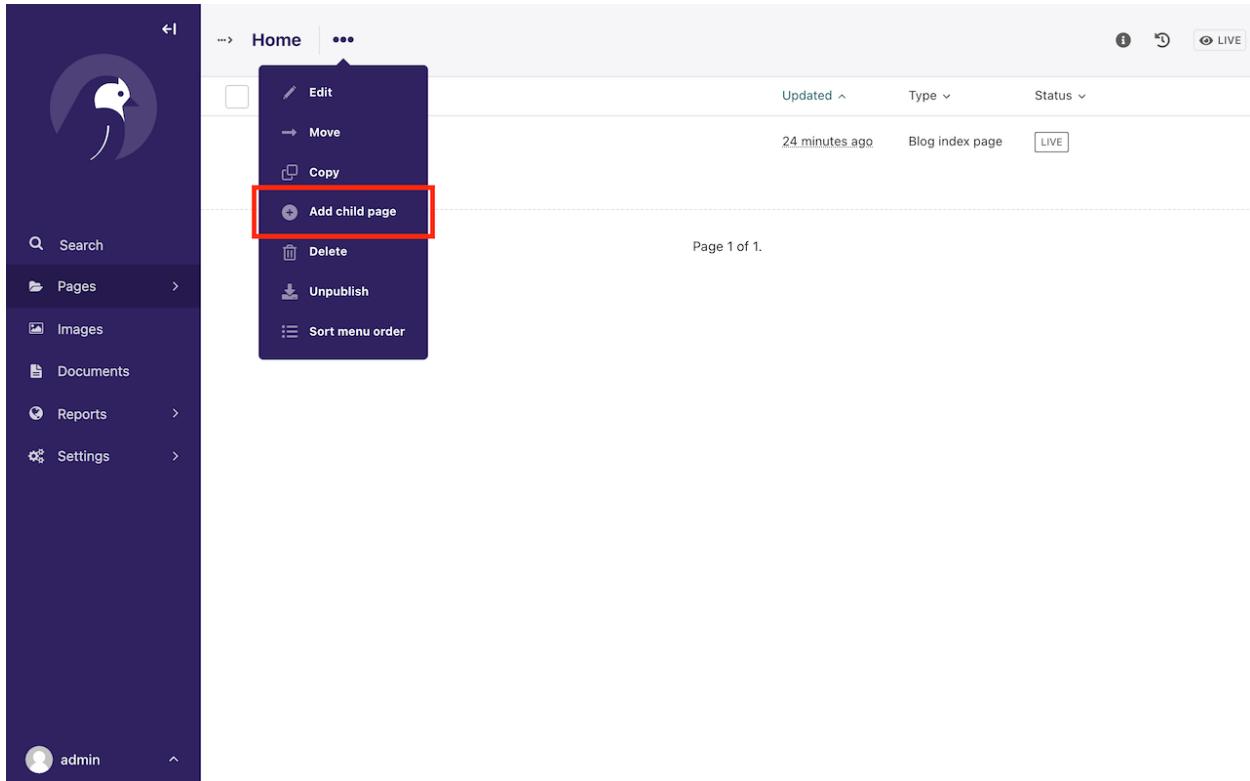
{{ page.body|richtext }}

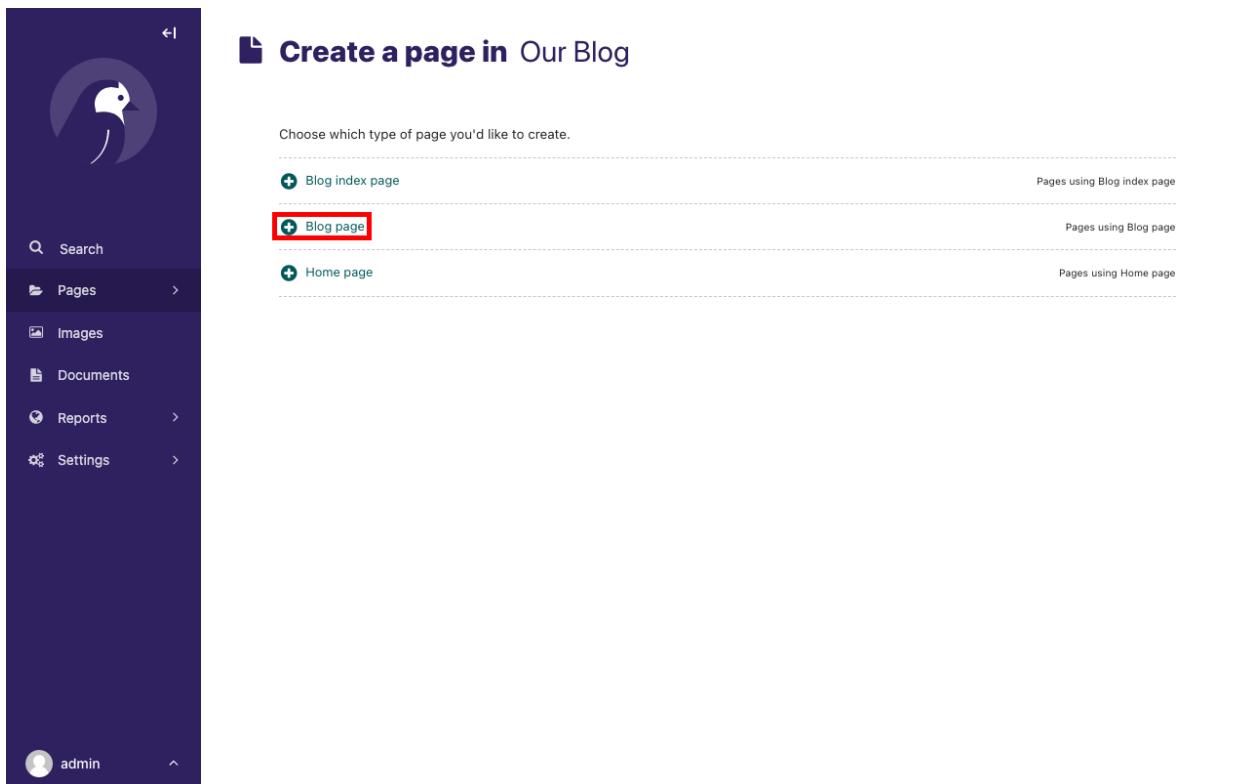
<p><a href="{{ page.get_parent.url }}">Return to blog</a></p>

{% endblock %}
```

Note the use of Wagtail's built-in `get_parent()` method to obtain the URL of the blog this post is a part of.

Now create a few blog posts as children of `BlogIndexPage`. Be sure to select type “Blog Page” when creating your posts.





Wagtail gives you full control over what kinds of content can be created under various parent content types. By default, any page type can be a child of any other page type.

The screenshot shows the Wagtail admin interface for creating a 'First blog post'. The sidebar is identical to the previous screenshot. The main content area shows a title 'First blog post' and a sub-header 'The page title as you'd like it to be seen by the public'. There are three expandable sections: 'Post date *' containing a date input field with '2022-05-14'; 'Intro *' containing a text input field with 'Hello world!'; and 'Body' containing a rich text editor with the following placeholder text:
There's a lot that we can say about our ~~bread~~ bread but frankly we think you need to taste it to believe it. Crafted from lines of Python, Django and the old staples of HTML, CSS and JS we think you'll love what we are baking. With our demonstrations across our fair Island you're never far from a treat. Come visit us whenever you're nearby.
Proportions of types of flour and other ingredients vary widely, as do modes of preparation. As a result, types, shapes, sizes, and textures of breads differ around the world. Bread may
The rich text editor also includes a note: 'Save draft' and a note about sourdough leavening.

Publish each blog post when you are done editing.

You should now have the very beginnings of a working blog. Access the `/blog` URL and you should see something like this:

Our Blog

First blog post

Hello world!

There's a *lot* that we can say about our [bread](#) but frankly we think you need to taste it to believe it. Crafted from lines of Python, Django and the old staples of HTML, CSS and JS we think you'll love what we are baking. With our demonstrations across our fair Island you're never far from a treat. Come visit us whenever you're nearby.

Proportions of types of flour and other ingredients vary widely, as do modes of preparation. As a result, types, shapes, sizes, and textures of breads differ around the world. Bread may be leavened by processes such as reliance on naturally occurring sourdough microbes, chemicals, industrially produced yeast, or high-pressure aeration. Some bread is cooked before it can leaven, including for traditional or religious reasons. Non-cereal ingredients such as fruits, nuts and fats may be included. Commercial bread commonly contains additives to improve flavor, texture, color, shelf life, and ease of manufacturing.

Second Post



Our most excellent bread selection

Bread is a [staple food](#) prepared from a [dough](#) of [flour](#) and [water](#), usually by [baking](#). Throughout recorded history it has been popular around the world and is one of the oldest artificial foods, having been of importance since the dawn of [agriculture](#).

Third Post

If you read a lot you're well read / If you eat a lot you're well bread

Proportions of types of flour and other ingredients vary widely, as do modes of preparation. As a result, types, shapes, sizes, and textures of breads differ around the world. Bread may be [leavened](#) by processes such as reliance on naturally occurring [sourdough](#) microbes, chemicals, industrially produced yeast, or high-pressure aeration. Some bread is cooked before it can leaven, including for traditional or religious reasons. Non-cereal ingredients such as fruits, nuts and fats may be included. Commercial bread commonly contains additives to improve flavor, texture, color, shelf life, and ease of manufacturing.



Titles should link to post pages, and a link back to the blog's homepage should appear in the footer of each post page.

Parents and Children

Much of the work you'll be doing in Wagtail revolves around the concept of hierarchical “tree” structures consisting of nodes and leaves (see [Theory](#)). In this case, the `BlogIndexPage` is a “node” and individual `BlogPage` instances are the “leaves”.

Take another look at the guts of `blog_index_page.html`:

```
{% for post in page.get_children %}
    <h2><a href="{% pageurl post %}">{{ post.title }}</a></h2>
    {{ post.specific.intro }}
    {{ post.specific.body|richtext }}
{% endfor %}
```

Every “page” in Wagtail can call out to its parent or children from its own position in the hierarchy. But why do we have to specify `post.specific.intro` rather than `post.intro`? This has to do with the way we defined our model:

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

The `get_children()` method gets us a list of instances of the `Page` base class. When we want to reference properties of the instances that inherit from the base class, Wagtail provides the `specific` method that retrieves the actual `BlogPage` record. While the “title” field is present on the base `Page` model, “intro” is only present on the `BlogPage` model, so we need `.specific` to access it.

To tighten up template code like this, we could use Django’s `with` tag:

```
{% for post in page.get_children %}
  {% with post=post.specific %}
    <h2><a href="{% pageurl post %}">{{ post.title }}</a></h2>
    <p>{{ post.intro }}</p>
    {{ post.body|richtext }}
  {% endwith %}
{% endfor %}
```

When you start writing more customised Wagtail code, you'll find a whole set of QuerySet modifiers to help you navigate the hierarchy.

```
# Given a page object 'somepage':
MyModel.objects.descendant_of(somepage)
child_of(page) / not_child_of(somepage)
ancestor_of(somepage) / not_ancestor_of(somepage)
parent_of(somepage) / not_parent_of(somepage)
sibling_of(somepage) / not_sibling_of(somepage)
# ... and ...
somepage.get_children()
somepage.get_ancestors()
somepage.get_descendants()
somepage.get_siblings()
```

For more information, see: [Page QuerySet reference](#)

Overriding Context

There are a couple of problems with our blog index view:

1. Blogs generally display content in *reverse* chronological order
2. We want to make sure we're only displaying *published* content.

To accomplish these things, we need to do more than just grab the index page's children in the template. Instead, we'll want to modify the QuerySet in the model definition. Wagtail makes this possible via the overridable `get_context()` method. Modify your `BlogIndexPage` model like this:

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

    def get_context(self, request):
        # Update context to include only published posts, ordered by reverse-chron
        context = super().get_context(request)
        blogpages = self.get_children().live().order_by('-first_published_at')
        context['blogpages'] = blogpages
        return context
```

All we've done here is retrieve the original context, create a custom QuerySet, add it to the retrieved context, and return the modified context back to the view. You'll also need to modify your `blog_index_page.html` template slightly. Change:

```
{% for post in page.get_children %} to {% for post in blogpages %}
```

Now try unpublishing one of your posts - it should disappear from the blog index page. The remaining posts should now be sorted with the most recently published posts first.

Images

Let's add the ability to attach an image gallery to our blog posts. While it's possible to simply insert images into the body rich text field, there are several advantages to setting up our gallery images as a new dedicated object type within the database - this way, you have full control of the layout and styling of the images on the template, rather than having to lay them out in a particular way within the rich text field. It also makes it possible for the images to be used elsewhere, independently of the blog text - for example, displaying a thumbnail on the blog index page.

Add a new `BlogPageGalleryImage` model to `models.py`:

```
from django.db import models

# New imports added for ParentalKey, Orderable, InlinePanel

from modelcluster.fields import ParentalKey

from wagtail.models import Page, Orderable
from wagtail.fields import RichTextField
from wagtail.admin.panels import FieldPanel, InlinePanel
from wagtail.search import index


# ... (Keep the definition of BlogIndexPage, and update BlogPage:)

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

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

    content_panels = Page.content_panels + [
        FieldPanel('date'),
        FieldPanel('intro'),
        FieldPanel('body'),
        InlinePanel('gallery_images', label="Gallery images"),
    ]


class BlogPageGalleryImage(Orderable):
    page = ParentalKey(BlogPage, on_delete=models.CASCADE, related_name='gallery_images')
    image = models.ForeignKey(
        'wagtailimages.Image', on_delete=models.CASCADE, related_name='+'
    )
    caption = models.CharField(blank=True, max_length=250)

    panels = [
        FieldPanel('image'),
        FieldPanel('caption'),
    ]
```

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

There are a few new concepts here, so let's take them one at a time:

Inheriting from `Orderable` adds a `sort_order` field to the model, to keep track of the ordering of images in the gallery.

The `ParentalKey` to `BlogPage` is what attaches the gallery images to a specific page. A `ParentalKey` works similarly to a `ForeignKey`, but also defines `BlogPageGalleryImage` as a “child” of the `BlogPage` model, so that it’s treated as a fundamental part of the page in operations like submitting for moderation, and tracking revision history.

`image` is a `ForeignKey` to Wagtail’s built-in `Image` model, where the images themselves are stored. This appears in the page editor as a pop-up interface for choosing an existing image or uploading a new one. This way, we allow an image to exist in multiple galleries - effectively, we’ve created a many-to-many relationship between pages and images.

Specifying `on_delete=models.CASCADE` on the foreign key means that if the image is deleted from the system, the gallery entry is deleted as well. (In other situations, it might be appropriate to leave the entry in place - for example, if an “our staff” page included a list of people with headshots, and one of those photos was deleted, we’d rather leave the person in place on the page without a photo. In this case, we’d set the foreign key to `blank=True`, `null=True`, `on_delete=models.SET_NULL`.)

Finally, adding the `InlinePanel` to `BlogPage.content_panels` makes the gallery images available on the editing interface for `BlogPage`.

Adjust your blog page template to include the images:

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

{% load wagtailcore_tags wagtailimages_tags %}

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

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

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

    {{ page.body|richtext }}

    {% for item in page.gallery_images.all %}
        <div style="float: left; margin: 10px">
            {% image item.image fill-320x240 %}
            <p>{{ item.caption }}</p>
        </div>
    {% endfor %}

    <p><a href="{{ page.get_parent.url }}">Return to blog</a></p>
{% endblock %}
```

Here we use the `{% image %}` tag (which exists in the `wagtailimages_tags` library, imported at the top of the template) to insert an `` element, with a `fill-320x240` parameter to indicate that the image should be resized and cropped to fill a 320x240 rectangle. You can read more about using images in templates in the [docs](#).

Second Post

Sept. 3, 2022

Our most excellent bread selection

Bread is a [staple food](#) prepared from a [dough](#) of [flour](#) and [water](#), usually by [baking](#). Throughout recorded history it has been popular around the world and is one of the oldest artificial foods, having been of importance since the dawn of [agriculture](#).



Anadama bread getting sliced



A baguette



Close-up of yeast

[Return to blog](#)



Since our gallery images are database objects in their own right, we can now query and re-use them independently of the blog post body. Let's define a `main_image` method, which returns the image from the first gallery item (or `None` if no gallery items exist):

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

    def main_image(self):
        gallery_item = self.gallery_images.first()
        if gallery_item:
            return gallery_item.image
        else:
            return None

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

    content_panels = Page.content_panels + [
        FieldPanel('date'),
        FieldPanel('intro'),
        FieldPanel('body'),
        InlinePanel('gallery_images', label="Gallery images"),
    ]
```

This method is now available from our templates. Update `blog_index_page.html` to include the main image as a

thumbnail alongside each post:

```
{% load wagtailcore_tags wagtailimages_tags %}

...
{% for post in blogpages %}
  {% with post=post.specific %}
    <h2><a href="{% pageurl post %}">{{ post.title }}</a></h2>

    {% with post.main_image as main_image %}
      {% if main_image %}{% image main_image fill-160x100 %}{% endif %}
      {% endwith %}

      <p>{{ post.intro }}</p>
      {{ post.body|richtext }}
    {% endwith %}
  {% endfor %}
```

Tagging Posts

Let's say we want to let editors "tag" their posts, so that readers can, for example, view all bicycle-related content together. For this, we'll need to invoke the tagging system bundled with Wagtail, attach it to the `BlogPage` model and content panels, and render linked tags on the blog post template. Of course, we'll need a working tag-specific URL view as well.

First, alter `models.py` once more:

```
from django.db import models

# New imports added for ClusterTaggableManager, TaggedItemBase, MultiFieldPanel

from modelcluster.fields import ParentalKey
from modelcluster.contrib.taggit import ClusterTaggableManager
from taggit.models import TaggedItemBase

from wagtail.models import Page, Orderable
from wagtail.fields import RichTextField
from wagtail.admin.panels import FieldPanel, InlinePanel, MultiFieldPanel
from wagtail.search import index

# ... (Keep the definition of BlogIndexPage)

class BlogPageTag(TaggedItemBase):
  content_object = ParentalKey(
    'BlogPage',
    related_name='tagged_items',
    on_delete=models.CASCADE
  )
```

(continues on next page)

(continued from previous page)

```
class BlogPage(Page):
    date = models.DateField("Post date")
    intro = models.CharField(max_length=250)
    body = RichTextField(blank=True)
    tags = ClusterTaggableManager(through=BlogPageTag, blank=True)

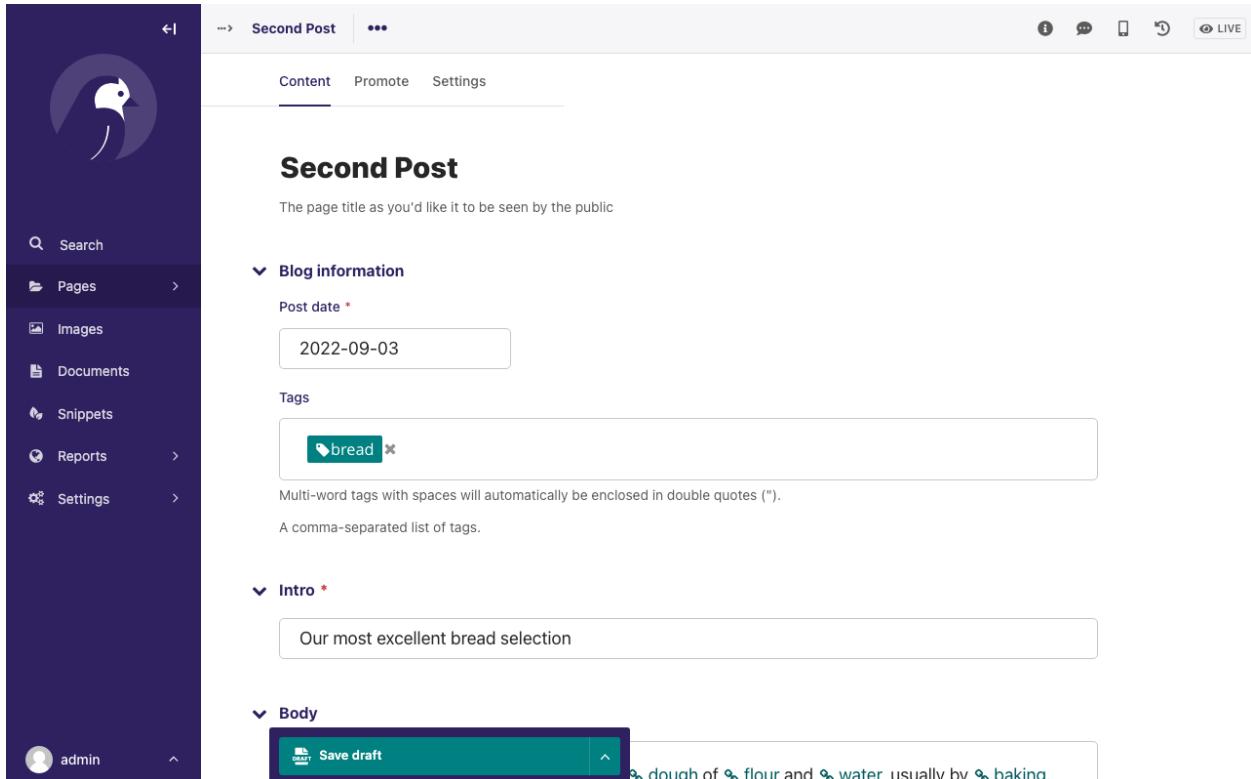
    # ... (Keep the main_image method and search_fields definition)

    content_panels = Page.content_panels + [
        MultiFieldPanel([
            FieldPanel('date'),
            FieldPanel('tags'),
        ], heading="Blog information"),
        FieldPanel('intro'),
        FieldPanel('body'),
        InlinePanel('gallery_images', label="Gallery images"),
    ]
```

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

Note the new `modelcluster` and `taggit` imports, the addition of a new `BlogPageTag` model, and the addition of a `tags` field on `BlogPage`. We've also taken the opportunity to use a `MultiFieldPanel` in `content_panels` to group the date and tags fields together for readability.

Edit one of your `BlogPage` instances, and you should now be able to tag posts:



To render tags on a `BlogPage`, add this to `blog_page.html`:

```
{% if page.tags.all.count %}
    <div class="tags">
        <h3>Tags</h3>
        {% for tag in page.tags.all %}
            <a href="{% slugurl 'tags' %}?tag={{ tag }}"><button type="button">{{ tag }}</button></a>
        {% endfor %}
    </div>
{% endif %}
```

Notice that we're linking to pages here with the builtin `slugurl` tag rather than `pageurl`, which we used earlier. The difference is that `slugurl` takes a Page slug (from the Promote tab) as an argument. `pageurl` is more commonly used because it is unambiguous and avoids extra database lookups. But in the case of this loop, the Page object isn't readily available, so we fall back on the less-preferred `slugurl` tag.

Visiting a blog post with tags should now show a set of linked buttons at the bottom - one for each tag. However, clicking a button will get you a 404, since we haven't yet defined a "tags" view. Add to `models.py`:

```
class BlogTagIndexPage(Page):

    def get_context(self, request):

        # Filter by tag
        tag = request.GET.get('tag')
        blogpages = BlogPage.objects.filter(tags__name=tag)

        # Update template context
        context = super().get_context(request)
        context['blogpages'] = blogpages
        return context
```

Note that this Page-based model defines no fields of its own. Even without fields, subclassing `Page` makes it a part of the Wagtail ecosystem, so that you can give it a title and URL in the admin, and so that you can manipulate its contents by returning a `QuerySet` from its `get_context()` method.

Migrate this in, then create a new `BlogTagIndexPage` in the admin. You'll probably want to create the new page/view as a child of `Homepage`, parallel to your `Blog` index. Give it the slug "tags" on the Promote tab.

Access `/tags` and Django will tell you what you probably already knew: you need to create a template `blog/blog_tag_index_page.html`:

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

{% block content %}

    {% if request.GET.tag %}
        <h4>Showing pages tagged "{{ request.GET.tag }}"{</h4>
    {% endif %}

    {% for blogpage in blogpages %}

        <p>
            <strong><a href="{{ pageurl blogpage }}">{{ blogpage.title }}</a></strong>
        <br />
    {% endfor %}
```

(continues on next page)

(continued from previous page)

```
<small>Revised: {{ blogpage.latest_revision_created_at }}</small><br />
{% if blogpage.author %}
    <p>By {{ blogpage.author.profile }}</p>
{% endif %}
</p>

{% empty %}
    No pages found with that tag.
{% endfor %}

{% endblock %}
```

We're calling the built-in `latest_revision_created_at` field on the `Page` model - handy to know this is always available.

We haven't yet added an "author" field to our `BlogPage` model, nor do we have a `Profile` model for authors - we'll leave those as an exercise for the reader.

Clicking the tag button at the bottom of a `BlogPost` should now render a page something like this:

Showing pages tagged "bread"

[First blog post](#)
Revised: Sept. 5, 2022, 2:24 p.m.

[Second Post](#)
Revised: Sept. 5, 2022, 2:29 p.m.



Categories

Let's add a category system to our blog. Unlike tags, where a page author can bring a tag into existence simply by using it on a page, our categories will be a fixed list, managed by the site owner through a separate area of the admin interface.

First, we define a `BlogCategory` model. A category is not a page in its own right, and so we define it as a standard Django `models.Model` rather than inheriting from `Page`. Wagtail introduces the concept of “snippets” for reusable pieces of content that need to be managed through the admin interface, but do not exist as part of the page tree themselves; a model can be registered as a snippet by adding the `@register_snippet` decorator. All the field types we've used so far on pages can be used on snippets too - here we'll give each category an icon image as well as a name. Add to `blog/models.py`:

```
from wagtail.snippets.models import register_snippet

@register_snippet
class BlogCategory(models.Model):
    name = models.CharField(max_length=255)
    icon = models.ForeignKey(
        'wagtailimages.Image', null=True, blank=True,
        on_delete=models.SET_NULL, related_name='+'
    )

    panels = [
        FieldPanel('name'),
        FieldPanel('icon'),
    ]

    def __str__(self):
        return self.name

    class Meta:
        verbose_name_plural = 'blog categories'
```

Note: Note that we are using `panels` rather than `content_panels` here - since snippets generally have no need for fields such as slug or publish date, the editing interface for them is not split into separate ‘content’ / ‘promote’ / ‘settings’ tabs as standard, and so there is no need to distinguish between ‘content panels’ and ‘promote panels’.

Migrate this change in, and create a few categories through the Snippets area which now appears in the admin menu.

We can now add categories to the `BlogPage` model, as a many-to-many field. The field type we use for this is `ParentalManyToManyField` - this is a variant of the standard Django `ManyToManyField` which ensures that the chosen objects are correctly stored against the page record in the revision history, in much the same way that `ParentalKey` replaces `ForeignKey` for one-to-many relations.

```
# New imports added for forms and ParentalManyToManyField
from django import forms
from django.db import models

from modelcluster.fields import ParentalKey, ParentalManyToManyField
from modelcluster.contrib.taggit import ClusterTaggableManager
from taggit.models import TaggedItemBase
```

(continues on next page)

(continued from previous page)

```
# ...

class BlogPage(Page):
    date = models.DateField("Post date")
    intro = models.CharField(max_length=250)
    body = RichTextField(blank=True)
    tags = ClusterTaggableManager(through=BlogPageTag, blank=True)
    categories = ParentalManyToManyField('blog.BlogCategory', blank=True)

    # ... (Keep the main_image method and search_fields definition)

    content_panels = Page.content_panels + [
        MultiFieldPanel([
            FieldPanel('date'),
            FieldPanel('tags'),
            FieldPanel('categories', widget=forms.CheckboxSelectMultiple),
        ], heading="Blog information"),
        FieldPanel('intro'),
        FieldPanel('body'),
        InlinePanel('gallery_images', label="Gallery images"),
    ]
```

Here we're making use of the `widget` keyword argument on the `FieldPanel` definition to specify a checkbox-based widget instead of the default multiple select box, as this is often considered more user-friendly.

Finally, we can update the `blog_page.html` template to display the categories:

```
<h1>{{ page.title }}</h1>
<p class="meta">{{ page.date }}</p>

{% with categories=page.categories.all %}
  {% if categories %}
    <h3>Posted in:</h3>
    <ul>
      {% for category in categories %}
        <li style="display: inline">
          {% image category.icon fill-32x32 style="vertical-align: middle" %}
          {{ category.name }}
        </li>
      {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

Second Post

Sept. 3, 2022

Posted in:



Our most excellent bread selection

Bread is a [staple food](#) prepared from a [dough](#) of [flour](#) and [water](#), usually by [baking](#). Throughout recorded history it has been popular around the world and is one of the oldest artificial foods, having been of importance since the dawn of [agriculture](#).



Anadama bread getting sliced



A baguette



Close-up of yeast

[Return to blog](#)

Tags

[bread](#)



Where next

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

Demo site

To create a new site on Wagtail we recommend the `wagtail start` command in [Getting started](#). We also have a demo site, The Wagtail Bakery, which contains example page types and models. We recommend you use the demo site for testing during development of Wagtail itself.

The source code and installation instructions can be found at <https://github.com/wagtail/bakerydemo>

Integrating Wagtail into a Django project

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

Wagtail is currently compatible with Django 3.2, 4.0 and 4.1. First, install the `wagtail` package from PyPI:

```
$ pip install wagtail
```

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

Settings

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

```
'wagtail.contrib.forms',
'wagtail.contrib.redirects',
'wagtail.embeds',
'wagtail.sites',
'wagtail.users',
'wagtail.snippets',
'wagtail.documents',
'wagtail.images',
'wagtail.search',
'wagtail.admin',
'wagtail',

'modelcluster',
'taggit',
```

Add the following entry to MIDDLEWARE:

```
'wagtail.contrib.redirects.middleware.RedirectMiddleware',
```

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

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

Add MEDIA_ROOT and MEDIA_URL settings, if your project does not have these already:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

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

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

Various other settings are available to configure Wagtail's behaviour - see [Settings](#).

URL configuration

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

```
from django.urls import path, include

from wagtail.admin import urls as wagtailadmin_urls
from wagtail import urls as wagtail_urls
from wagtail.documents import urls as wagtaildocs_urls

urlpatterns = [
    ...
    path('cms/', include(wagtailadmin_urls)),
    path('documents/', include(wagtaildocs_urls)),
    path('pages/', include(wagtail_urls)),
```

(continues on next page)

(continued from previous page)

```
[ ... ]
```

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

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

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

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

```
path('', include(wagtail_urls)),
```

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

Finally, your project needs to be set up to serve user-uploaded files from `MEDIA_ROOT`. Your Django project may already have this in place, but if not, add the following snippet to `urls.py`:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ... the rest of your URLconf goes here ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Note that this only works in development mode (`DEBUG = True`); in production, you will need to configure your web server to serve files from `MEDIA_ROOT`. For further details, see the Django documentation: [Serving files uploaded by a user during development](#) and [Deploying static files](#).

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

User accounts

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

Start developing

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

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

The Zen of Wagtail

Wagtail has been born out of many years of experience building websites, learning approaches that work and ones that don't, and striking a balance between power and simplicity, structure and flexibility. We hope you'll find that Wagtail is in that sweet spot. However, as a piece of software, Wagtail can only take that mission so far - it's now up to you to create a site that's beautiful and a joy to work with. So, while it's tempting to rush ahead and start building, it's worth taking a moment to understand the design principles that Wagtail is built on.

In the spirit of “[The Zen of Python](#)”, The Zen of Wagtail is a set of guiding principles, both for building websites in Wagtail, and for the ongoing development of Wagtail itself.

Wagtail is not an instant website in a box.

You can't make a beautiful website by plugging off-the-shelf modules together - expect to write code.

Always wear the right hat.

The key to using Wagtail effectively is to recognise that there are multiple roles involved in creating a website: the content author, site administrator, developer and designer. These may well be different people, but they don't have to be - if you're using Wagtail to build your personal blog, you'll probably find yourself hopping between those different roles. Either way, it's important to be aware of which of those hats you're wearing at any moment, and to use the right tools for that job. A content author or site administrator will do the bulk of their work through the Wagtail admin interface; a developer or designer will spend most of their time writing Python, HTML or CSS code. This is a good thing: Wagtail isn't designed to replace the job of programming. Maybe one day someone will come up with a drag-and-drop UI for building websites that's as powerful as writing code, but Wagtail is not that tool, and does not try to be.

A common mistake is to push too much power and responsibility into the hands of the content author and site administrator - indeed, if those people are your clients, they'll probably be loudly clamouring for exactly that. The success of your site depends on your ability to say no. The real power of content management comes not from handing control over to CMS users, but from setting clear boundaries between the different roles. Amongst other things, this means not having editors doing design and layout within the content editing interface, and not having site administrators building complex interaction workflows that would be better achieved in code.

A CMS should get information out of an editor's head and into a database, as efficiently and directly as possible.

Whether your site is about cars, cats, cakes or conveyancing, your content authors will be arriving at the Wagtail admin interface with some domain-specific information they want to put up on the website. Your aim as a site builder is to extract and store this information in its raw form - not one particular author's idea of how that information should look.

Keeping design concerns out of page content has numerous advantages. It ensures that the design remains consistent across the whole site, not subject to the whims of editors from one day to the next. It allows you to make full use of the informational content of the pages - for example, if your pages are about events, then having a dedicated "Event" page type with data fields for the event date and location will let you present the events in a calendar view or filtered listing, which wouldn't be possible if those were just implemented as different styles of heading on a generic page. Finally, if you redesign the site at some point in the future, or move it to a different platform entirely, you can be confident that the site content will work in its new setting, and not be reliant on being formatted a particular way.

Suppose a content author comes to you with a request: "We need this text to be in bright pink Comic Sans". Your question to them should be "Why? What's special about this particular bit of text?" If the reply is "I just like the look of it", then you'll have to gently persuade them that it's not up to them to make design choices. (Sorry.) But if the answer is "it's for our Children's section", then that gives you a way to divide the editorial and design concerns: give your editors the ability to designate certain pages as being "the Children's section" (through tagging, different page models, or the site hierarchy) and let designers decide how to apply styles based on that.

The best user interface for a programmer is usually a programming language.

A common sight in content management systems is a point-and-click interface to let you define the data model that makes up a page:

The screenshot shows the Wagtail Admin interface for managing fields of a 'News item' model. The top navigation bar includes links for Home, Administration, Structure, Content types, and News item. Below the navigation is a toolbar with tabs: EDIT, ACCESS CONTROL, MANAGE FIELDS (which is selected), MANAGE DISPLAY, COMMENT FIELDS, and COMMENT DISPLAY. A 'Show row weights' link is located in the top right corner of the table header. The table has columns for LABEL, MACHINE NAME, FIELD TYPE, WIDGET, and OPERATIONS. The first row, 'Content', is expanded to show its configuration: it uses a 'Vertical tab' field type with 'tab closed' and 'classes group-content field-group-tab required_fields yes'. Other rows include 'Title' (Node module element), 'Publication date' (Date ISO format), 'News section' (List text), 'News type' (Term reference), 'Introduction' (Long text), and 'Body' (Long text and summary). Each row also includes edit and delete buttons in the OPERATIONS column.

Show row weights				
LABEL	MACHINE NAME	FIELD TYPE	WIDGET	OPERATIONS
Content	group_content	Vertical tab	tab closed classes group-content field-group-tab required_fields yes	edit delete
Title	title	Node module element		edit delete
Publication date	field_media_date_published	Date (ISO format)	Pop-up calendar	edit delete
News section	field_news_section	List (text)	Check boxes/radio buttons	edit delete
News type	field_news_type	Term reference	Select list	edit delete
Introduction	field_intro	Long text	Text area (multiple rows)	edit delete
Body	field_body	Long text and summary	Text area with a summary	edit delete

It looks nice in the sales pitch, but in reality, no CMS end-user can realistically make that kind of fundamental change - on a live site, no less - unless they have a programmer's insight into how the site is built, and what impact the change will have. As such, it will always be the programmer's job to negotiate that point-and-click interface - all you've done is taken them away from the comfortable world of writing code, where they have a whole ecosystem of tools, from text editors to version control systems, to help them develop, test and deploy their code changes.

Wagtail recognises that most programming tasks are best done by writing code, and does not try to turn them into box-filling exercises when there's no good reason to. Likewise, when building functionality for your site, you should keep in mind that some features are destined to be maintained by the programmer rather than a content editor, and consider whether making them configurable through the Wagtail admin is going to be more of a hindrance than a convenience. For example, Wagtail provides a form builder to allow content authors to create general-purpose data collection forms. You might be tempted to use this as the basis for more complex forms that integrate with (for example) a CRM system or payment processor - however, in this case there's no way to edit the form fields without rewriting the backend logic, so making them editable through Wagtail has limited value. More likely, you'd be better off building these using Django's form framework, where the form fields are defined entirely in code.

1.2 Usage guide

1.2.1 Page models

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

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

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

- [Creating models](#)
 - [Model syntax](#)
-

An example Wagtail page model

This example represents a typical blog post:

```
from django.db import models

from modelcluster.fields import ParentalKey

from wagtail.models import Page, Orderable
from wagtail.fields import RichTextField
from wagtail.admin.panels import FieldPanel, MultiFieldPanel, InlinePanel
from wagtail.search import index


class BlogPage(Page):

    # Database fields

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

(continues on next page)

(continued from previous page)

```
        on_delete=models.SET_NULL,
        related_name='+'  
    )  
  
    # Search index configuration  
  
    search_fields = Page.search_fields + [  
        index.SearchField('body'),  
        index.FilterField('date'),  
    ]  
  
    # Editor panels configuration  
  
    content_panels = Page.content_panels + [  
        FieldPanel('date'),  
        FieldPanel('body'),  
        InlinePanel('related_links', heading="Related links", label="Related link"),  
    ]  
  
    promote_panels = [  
        MultiFieldPanel(Page.promote_panels, "Common page configuration"),  
        FieldPanel('feed_image'),  
    ]  
  
    # Parent page / subpage type rules  
  
    parent_page_types = ['blog.BlogIndex']  
    subpage_types = []  
  
class BlogPageRelatedLink(Orderable):  
    page = ParentalKey(BlogPage, on_delete=models.CASCADE, related_name='related_links')  
    name = models.CharField(max_length=255)  
    url = models.URLField()  
  
    panels = [  
        FieldPanel('name'),  
        FieldPanel('url'),  
    ]
```

Note: Ensure that none of your field names are the same as your class names. This will cause errors due to the way Django handles relations ([read more](#)). In our examples we have avoided this by appending “Page” to each model name.

Writing page models

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

Database fields

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

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

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

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

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

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

Search

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

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

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

The arguments that these field types accept are documented in [indexing extra fields](#).

Editor panels

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

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

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

Here's a summary of the `Panel` classes that Wagtail provides out of the box. See [Panel types](#) for full descriptions.

Basic

These allow editing of model fields. The `FieldPanel` class will choose the correct widget based on the type of the field, such as a rich text editor for `RichTextField`, or an image chooser for a `ForeignKey` to an image model. `FieldPanel` also provides a page chooser interface for `ForeignKeys` to page models, but for more fine-grained control over which page types can be chosen, `PageChooserPanel` provides additional configuration options.

- [FieldPanel](#)
- [PageChooserPanel](#)

Changed in version 3.0: Previously, certain field types required special-purpose panels: `StreamFieldPanel`, `ImageChooserPanel`, `DocumentChooserPanel` and `SnippetChooserPanel`. These are now all handled by `FieldPanel`.

Structural

These are used for structuring fields in the interface.

- `MultiFieldPanel`
- `InlinePanel`
- `FieldRowPanel`

Customising the page editor interface

The page editor can be customised further. See [Customising the editing interface](#).

Parent page / subpage type rules

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

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

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

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

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

Page descriptions

With every Wagtail Page you are able to add a helpful description text, similar to a `help_text` model attribute. By adding `page_description` to your Page model you’ll be adding a short description that can be seen when you create a new page, edit an existing page or when you’re prompted to select a child page type.

```
class LandingPage(Page):  
  
    page_description = "Use this page for converting users"
```

Page URLs

The most common method of retrieving page URLs is by using the `{% pageurl %}` template tag. Since it's called from a template, `pageurl` automatically includes the optimizations mentioned below. For more information, see [pageurl](#).

Page models also include several low-level methods for overriding or accessing page URLs.

Customising URL patterns for a page model

The `Page.get_url_parts(request)` method will not typically be called directly, but may be overridden to define custom URL routing for a given page model. It should return a tuple of (`site_id`, `root_url`, `page_path`), which are used by `get_url` and `get_full_url` (see below) to construct the given type of page URL.

When overriding `get_url_parts()`, you should accept `*args`, `**kwargs`:

```
def get_url_parts(self, *args, **kwargs):
```

and pass those through at the point where you are calling `get_url_parts` on `super` (if applicable), for example:

```
super().get_url_parts(*args, **kwargs)
```

While you could pass only the `request` keyword argument, passing all arguments as-is ensures compatibility with any future changes to these method signatures.

For more information, please see [wagtail.models.Page.get_url_parts\(\)](#).

Obtaining URLs for page instances

The `Page.get_url(request)` method can be called whenever a page URL is needed. It defaults to returning local URLs (not including the protocol or domain) if it determines that the page is on the current site (via the hostname in `request`); otherwise, a full URL including the protocol and domain is returned. Whenever possible, the optional `request` argument should be included to enable per-request caching of site-level URL information and facilitate the generation of local URLs.

A common use case for `get_url(request)` is in any custom template tag your project may include for generating navigation menus. When writing such a custom template tag, ensure that it includes `takes_context=True` and use `context.get('request')` to safely pass the request or `None` if no request exists in the context.

For more information, please see [wagtail.models.Page.get_url\(\)](#).

In the event a full URL (including the protocol and domain) is needed, `Page.get_full_url(request)` can be used instead. Whenever possible, the optional `request` argument should be included to enable per-request caching of site-level URL information.

For more information, please see [wagtail.models.Page.get_full_url\(\)](#).

Template rendering

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

Adding a template for a page model

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

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

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

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

Template context

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

Customising template context

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

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

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

    def get_context(self, request, *args, **kwargs):
        context = super().get_context(request, *args, **kwargs)

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

The variables can then be used in the template:

```
{{ page.title }}

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

Changing the template

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

```
class BlogPage(Page):
    ...
    template = 'other_template.html'
```

Dynamically choosing the template

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

```
class BlogPage(Page):
    ...
    use_other_template = models.BooleanField()

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

        return 'blog/blog_page.html'
```

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

Ajax Templates

If you want to add AJAX functionality to a page, such as a paginated listing that updates in-place on the page rather than triggering a full page reload, you can set the `ajax_template` attribute to specify an alternative template to be used when the page is requested via an AJAX call (as indicated by the `X-Requested-With: XMLHttpRequest` HTTP header):

```
class BlogPage(Page):
    ...
    ajax_template = 'other_template_fragment.html'
    template = 'other_template.html'
```

More control over page rendering

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

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

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

```
from django.http import JsonResponse

class BlogPage(Page):
    ...

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

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

```

Inline models

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

Each inline model requires the following:

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

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

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

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

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

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

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

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

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

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

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

Working with pages

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

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

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

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

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

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

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

You can convert a `Page` object to its more specific user-defined equivalent using the `.specific` property. This may cause an additional database lookup.

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

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

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

Tips

Friendly model names

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

```
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. Defining `verbose_name` as in the example above would change this to read “Homepage”, which is slightly more conventional.

Page QuerySet ordering

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

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

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

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

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

Custom Page managers

You can add a custom Manager to your Page class. Any custom Managers should inherit from `wagtail.models.PageManager`:

```
from django.db import models
from wagtail.models import Page, PageManager

class EventPageManager(PageManager):
    """Custom manager for Event pages"""

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

    objects = EventPageManager()
```

Alternately, if you only need to add extra QuerySet methods, you can inherit from `wagtail.models.PageQuerySet` to build a custom Manager:

```

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

class EventPageQuerySet(PageQuerySet):
    def future(self):
        today = timezone.localtime(timezone.now()).date()
        return self.filter(start_date__gte=today)

EventPageManager = PageManager.from_queryset(EventPageQuerySet)

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

    objects = EventPageManager()

```

1.2.2 Writing templates

Wagtail uses Django’s templating language. For developers new to Django, start with Django’s own template documentation: [Templates](#)

Python programmers new to Django/Wagtail may prefer more technical documentation: [The Django template language: for Python programmers](#)

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 (almost) the same name. The Front End developer may need to create these templates themselves by referring to `models.py` to infer template names from the models defined therein.

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

Template files are assumed to exist here:

```

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

```

For more information, see the Django documentation for the [application directories template loader](#).

Page content

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

A custom variable name can be configured on the page model `wagtail.models.Page.context_object_name`. If a custom name is defined, `page` is still available for use in shared templates.

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

Static assets

Static files (such as 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 a Wagtail site by its users (as opposed to a developer’s static files, mentioned above) go into the image library and from there are added to pages via the [page editor interface](#).

Unlike other CMSs, 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 like `img` tags. Only images from the library can be manipulated on the fly.

Read more about the image manipulation syntax here: [How to use 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` just like 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 `image` tag is thus:

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

For example:

```
{% load wagtailimages_tags %}  
...  
  
{% image page.photo width-400 %}  
  
<!-- or a square thumbnail: -->  
{% image page.photo fill-80x80 %}
```

See [How to use images in templates](#) for full documentation.

Rich text (filter)

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

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

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

Responsive Embeds

As Wagtail does not impose any styling of its own on templates, images and embedded media will be displayed at a fixed width as determined by the HTML. Images can be made to resize to fit their container using a CSS rule such as the following:

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

where `body` is a container element in your template surrounding the images.

Making embedded media resizable is also possible, but typically requires custom style rules matching the media's aspect ratio. To assist in this, Wagtail provides built-in support for responsive embeds, which can be enabled by setting `WAGTAILEMBEDS_RESPONSIVE_HTML = True` in your project settings. This adds a CSS class of `responsive-object` and an inline `padding-bottom` style to the embed, to be used in conjunction with the following CSS:

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

Internal links (tag)

pageurl

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

```
{% load wagtailcore_tags %}  
...  
<a href="{% pageurl page.get_parent %}">Back to index</a>
```

A `fallback` keyword argument can be provided - this can be a URL string, a named URL route that can be resolved with no parameters, or an object with a `get_absolute_url` method, and will be used as a substitute URL when the passed page is `None`.

```
{% load wagtailcore_tags %}  
  
{% for publication in page.related_publications.all %}  
    <li>  
        <a href="{% pageurl publication.detail_page fallback='coming_soon' %}">  
            {{ publication.title }}  
        </a>  
    </li>  
{% endfor %}
```

slugurl

Takes any `slug` as defined in a page’s “Promote” tab and returns the URL for the matching Page. If multiple pages exist with the same slug, the page chosen is undetermined.

Like `pageurl`, this will try to provide a relative link if possible, but will default to an absolute link if the Page is on a different Site. This is most useful when creating shared page furniture, for example top level navigation or site-wide links.

```
{% load wagtailcore_tags %}  
...  
<a href="{% slugurl 'news' %}">News index</a>
```

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 development and live environments.

```
{% load static %}

...

```

Notice that the full path is not required - the path given here is relative to the app's `static` directory. To avoid clashes with static files from other apps (including Wagtail itself), it's recommended to place static files in a subdirectory of `static` with the same name as the app.

Multi-site support

wagtail_site

Returns the Site object corresponding to the current request.

```
{% load wagtailcore_tags %}

{% wagtail_site as current_site %}
```

Wagtail User Bar

This tag provides a contextual flyout menu for logged-in users. The menu gives editors the ability to edit the current page or add a child page, besides the options to show the page in the Wagtail page explorer or jump to the Wagtail admin dashboard. Moderators are also given the ability to accept or reject a page being previewed as part of content moderation.

This tag may be used on standard Django views, without page object. The user bar will contain one item pointing to the admin.

We recommend putting the tag near the top of the `<body>` element so keyboard users can reach it. You should consider putting the tag after any `skip links` but before the navigation and main content of your page.

```
{% load wagtailuserbar %}

...
<body>
    <a id="#content">Skip to content</a>
    {% wagtailuserbar %} # This is a good place for the userbar #
    <nav>
        ...
    </nav>
    <main id="content">
        ...
    </main>
</body>
```

By default the User Bar appears in the bottom right of the browser window, inset from the edge. If this conflicts with your design it can be moved by passing a parameter to the template tag. These examples show you how to position the userbar in each corner of the screen:

```
...
{% wagtailuserbar 'top-left' %}
{% wagtailuserbar 'top-right' %}
{% wagtailuserbar 'bottom-left' %}
{% wagtailuserbar 'bottom-right' %}
...
```

The userbar can be positioned where it works best with your design. Alternatively, you can position it with a CSS rule in your own CSS files, for example:

```
.wagtail-userbar {
    top: 200px !important;
    left: 10px !important;
}
```

Varying output between preview and live

Sometimes you may wish to vary the template output depending on whether the page is being previewed or viewed live. For example, if you have visitor tracking code such as Google Analytics in place on your site, it's a good idea to leave this out when previewing, so that editor activity doesn't appear in your analytics reports. Wagtail provides a `request.is_preview` variable to distinguish between preview and live:

```
{% if not request.is_preview %}
<script>
    (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function() {
        ...
    }
</script>
{% endif %}
```

If the page is being previewed, `request.preview_mode` can be used to determine the specific preview mode being used, if the page supports *multiple preview modes*.

1.2.3 How to use images in templates

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

The syntax for the tag is thus:

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

Both the image and resize rule must be passed to the template tag.

For example:

```
{% load wagtailimages_tags %}
...
<!-- Display the image scaled to a width of 400 pixels: -->
{% image page.photo width-400 %}

<!-- Display it again, but this time as a square thumbnail: -->
{% image page.photo fill-80x80 %}
```

In the above syntax example `[image]` is the Django object referring to the image. If your page model defined a field called “photo” then `[image]` would probably be `page.photo`. The `[resize-rule]` defines how the image is to be resized when inserted into the page. Various resizing methods are supported, to cater to different use cases (for example 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 width is always specified before the height. Resized images will maintain their original aspect ratio unless the `fill` rule is used, which may result in some pixels being cropped.

Available resizing methods

The available resizing methods are as follows:

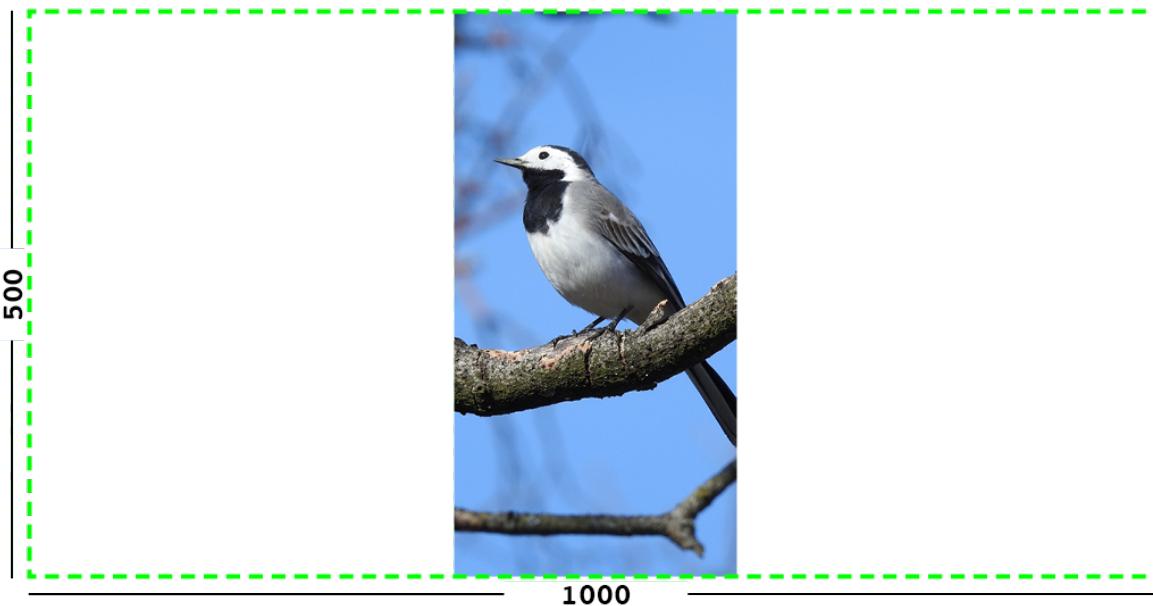
`max`

(takes two dimensions)

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

Fit **within** the given dimensions.

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



Example: The image will keep its proportions but fit within the max (green line) dimensions provided.

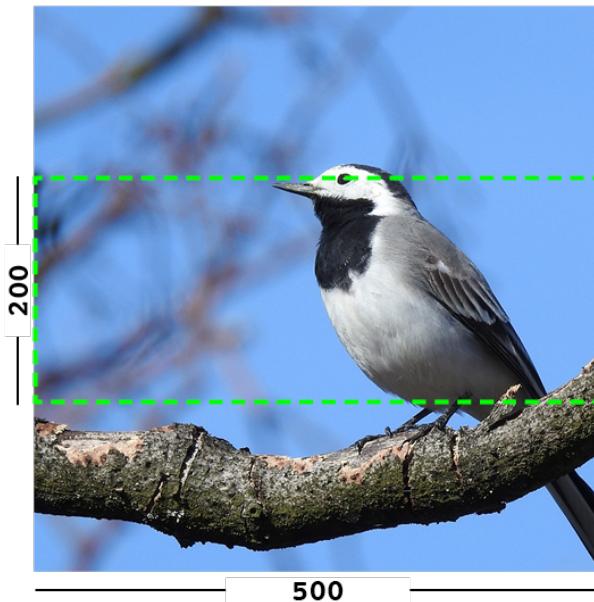
min

(takes two dimensions)

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

Cover the given dimensions.

This may result in an image slightly **larger** than the dimensions you specify. A square image of width 2000 and height 2000, treated with the `min-500x200` rule would have its height and width changed to 500, that is matching the *width* of the resize-rule, but greater than the height.



Example: The image will keep its proportions while filling at least the min (green line) dimensions provided.

width

(takes one dimension)

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

Reduces the width of the image to the dimension specified.

height

(takes one dimension)

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

Reduces the height of the image to the dimension specified.

scale

(takes percentage)

```
{% image page.photo scale-50 %}
```

Resize the image to the percentage specified.

fill

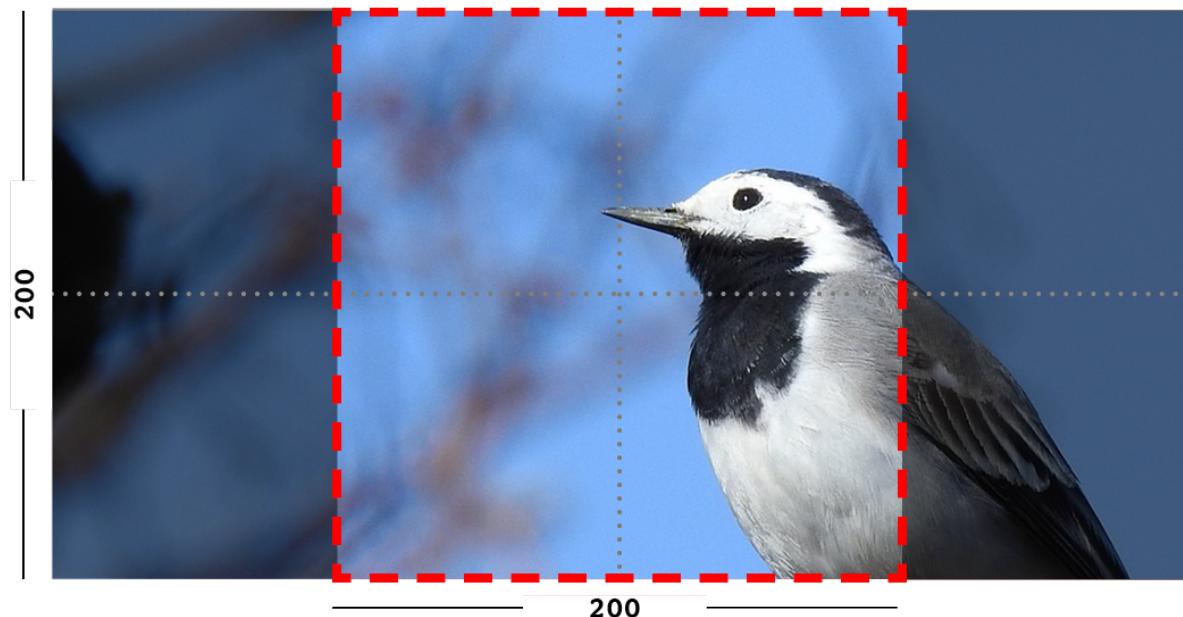
(takes two dimensions and an optional -c parameter)

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

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

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

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



Example: The image is scaled and also cropped (red line) to fit as much of the image as possible within the provided dimensions.

On images that won't upscale

It's possible to request an image with `fill` dimensions that the image can't support without upscaling. For example an image of width 400 and height 200 requested with `fill-400x400`. In this situation the *ratio of the requested fill* will be matched, but the dimension will not. So that example 400x200 image (a 2:1 ratio) could become 200x200 (a 1:1 ratio, matching the resize-rule).

Cropping closer to the focal point

By default, Wagtail will only crop enough to change the aspect ratio of the image to match the ratio in the resize-rule.

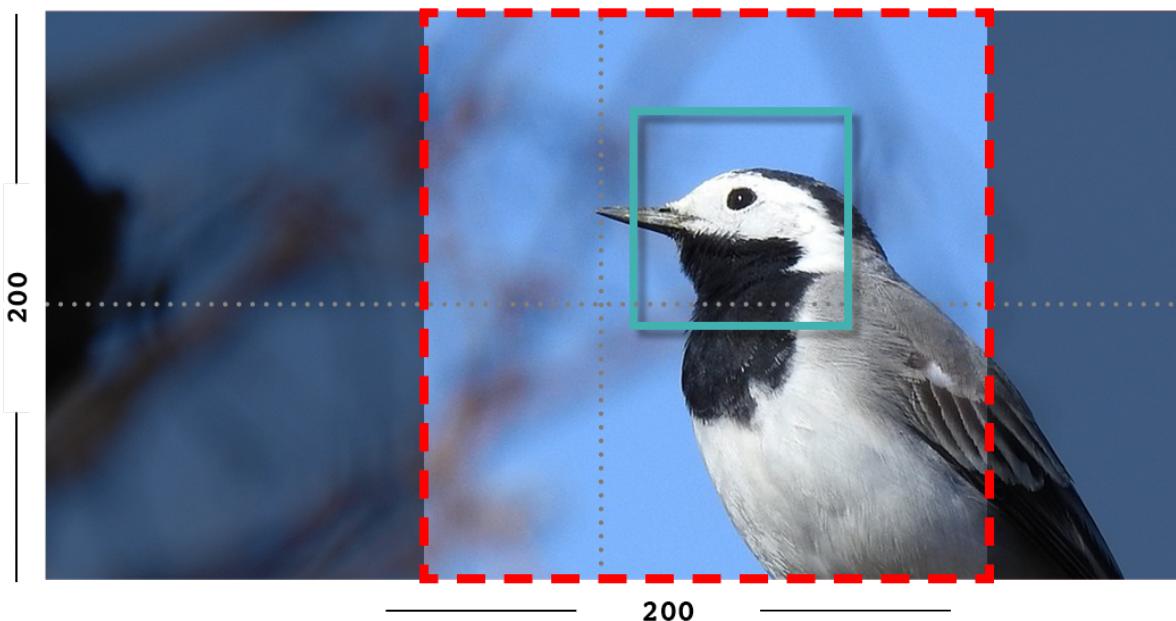
In some cases (for example thumbnails), it may be preferable to crop closer to the focal point, so that the subject of the image is more prominent.

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

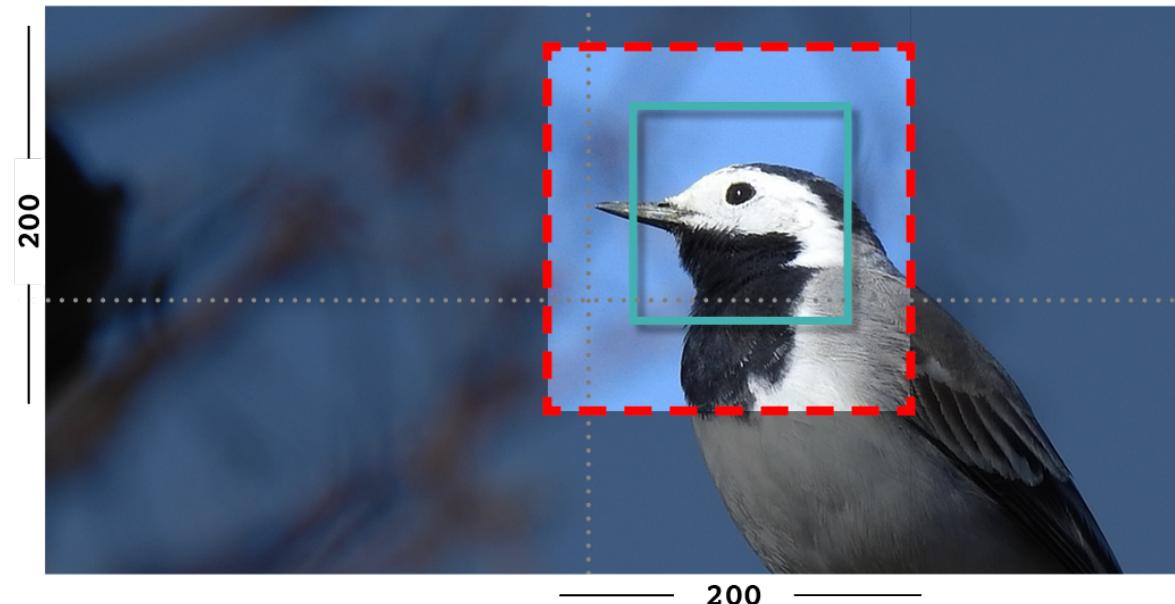
```
{% image page.photo fill-200x200-c100 %}
```

This will crop the image as much as it can, without cropping 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.



Example: The focal point is set off centre so the image is scaled and also cropped like `fill`, however the center point of the crop is positioned closer to the focal point.



Example: With `-c75` set, the final crop will be closer to the focal point.

original

(takes no dimensions)

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

Renders the image at its original size.

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

More control over the `img` tag

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

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

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

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

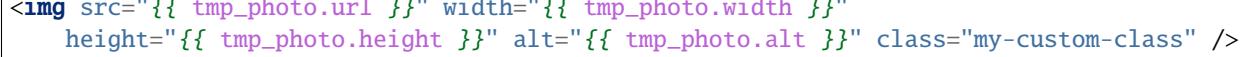
You can set a more relevant `alt` attribute this way, overriding the one automatically generated from the title of the image. The `src`, `width`, and `height` attributes can also be overridden, if necessary.

You can also add default attributes to all images (a default class or data attribute for example) - see [Adding default attributes to all images](#).

2. Generating the image “as foo” to access individual properties

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

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


    width="{{ tmp_photo.width }}"
    height="{{ tmp_photo.height }}"
    src="{{ tmp_photo.url }}"/>
```

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

This syntax exposes the underlying image Rendition (`tmp_photo`) to the developer. A “Rendition” contains the information specific to the way you’ve requested to format the image using the resize-rule, dimensions and source URL. The following properties are available:

url

URL to the resized version of the image. This may be a local URL (such as `/static/images/example.jpg`) or a full URL (such as `https://assets.example.com/images/example.jpg`), depending on how static files are configured.

width

Image width after resizing.

height

Image height after resizing.

alt

Alternative text for the image, typically taken from the image title.

attrs

A shorthand for outputting the attributes `src`, `width`, `height` and `alt` in one go:

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

full_url

Same as `url`, but always returns a full absolute URL. This requires `WAGTAILADMIN_BASE_URL` to be set in the project settings.

This is useful for images that will be re-used outside of the current site, such as social share images:

```
<meta name="twitter:image" content="{{ tmp_photo.full_url }}>
```

If your site defines a custom image model using `AbstractImage`, any additional fields you add to an image (such as a copyright holder) are **not** included in the rendition.

Therefore, if you'd added the field `author` to your `AbstractImage` in the above example, you'd access it using `{{ page.photo.author }}` rather than `{{ tmp_photo.author }}`.

(Due to the links in the database between renditions and their parent image, you *could* access it as `{{ tmp_photo.image.author }}`, but that has reduced readability.)

Adding default attributes to all images

We can configure the `wagtail.images` application to specify additional attributes to add to images. This is done by setting up a custom `AppConfig` class within your project folder (i.e. the package containing the top-level `settings` and `urls` modules).

To do this, create or update your existing `apps.py` file with the following:

```
from wagtail.images.apps import WagtailImagesAppConfig

class CustomImagesAppConfig(WagtailImagesAppConfig):
    default_attrs = {"decoding": "async", "loading": "lazy"}
```

Then, replace `wagtail.images` in `settings.INSTALLED_APPS` with the path to `CustomUsersAppConfig`:

```
INSTALLED_APPS = [
    ...,
    "myapplication.apps.CustomImagesAppConfig",
    # "wagtail.images",
    ...
]
```

Now, images created with `{% image %}` will additionally have `decoding="async"` `loading="lazy"` attributes. This also goes for images added to Rich Text and `ImageBlock` blocks.

Alternative HTML tags

The `as` keyword allows alternative HTML image tags (such as `<picture>` or ``) to be used. For example, to use the `<picture>` tag:

```
<picture>
    {% image page.photo width-800 as wide_photo %}
    <source srcset="{{ wide_photo.url }}" media="(min-width: 800px)">
    {% image page.photo width-400 %}
</picture>
```

And to use the `<amp-img>` tag (based on the Mountains example from the AMP docs):

```
{% image image width-550 format-webp as webp_image %}  
{% image image width-550 format-jpeg as jpeg_image %}  
  
<amp-img alt="{{ image.alt }}"  
    width="{{ webp_image.width }}"  
    height="{{ webp_image.height }}"  
    src="{{ webp_image.url }}>  
    <amp-img alt="{{ image.alt }}"  
        fallback  
        width="{{ jpeg_image.width }}"  
        height="{{ jpeg_image.height }}"  
        src="{{ jpeg_image.url }}></amp-img>  
</amp-img>
```

Images embedded in rich text

The information above relates to images defined via image-specific fields in your model. However, images can also be embedded arbitrarily in Rich Text fields by the page editor (see [Rich Text \(HTML\)](#)).

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

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

Full width

Creates an image rendition using `width-800`, giving the tag the CSS class `full-width`.

Left-aligned

Creates an image rendition using `width-500`, giving the tag the CSS class `left`.

Right-aligned

Creates an image rendition using `width-500`, giving the tag the CSS class `right`.

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

For more information about image formats, including creating your own, see [Image Formats in the Rich Text Editor](#).

Output image format

Wagtail may automatically change the format of some images when they are resized:

- PNG and JPEG images don't change format
- GIF images without animation are converted to PNGs
- BMP images are converted to PNGs
- WebP images are converted to PNGs

It is also possible to override the output format on a per-tag basis by using the `format` filter after the resize rule.

For example, to make the tag always convert the image to a JPEG, use `format-jpeg`:

```
{% image page.photo width-400 format-jpeg %}
```

You may also use `format-png` or `format-gif`.

Lossless WebP

You can encode the image into lossless WebP format by using the `format-webp-lossless` filter:

```
{% image page.photo width-400 format-webp-lossless %}
```

Background colour

The PNG and GIF image formats both support transparency, but if you want to convert images to JPEG format, the transparency will need to be replaced with a solid background colour.

By default, Wagtail will set the background to white. But if a white background doesn't fit your design, you can specify a colour using the `bgcolor` filter.

This filter takes a single argument, which is a CSS 3 or 6 digit hex code representing the colour you would like to use:

```
{# Sets the image background to black #}
{% image page.photo width-400 bgcolor-000 format-jpeg %}
```

Image quality

Wagtail's JPEG and WebP image quality settings default to 85 (which is quite high). This can be changed either globally or on a per-tag basis.

Changing globally

Use the `WAGTAILIMAGES_JPEG_QUALITY` and `WAGTAILIMAGES_WEBP_QUALITY` settings to change the global defaults of JPEG and WebP quality:

```
# settings.py

# Make low-quality but small images
WAGTAILIMAGES_JPEG_QUALITY = 40
WAGTAILIMAGES_WEBP_QUALITY = 45
```

Note that this won't affect any previously generated images so you may want to delete all renditions so they can regenerate with the new setting. This can be done from the Django shell:

```
# Replace this with your custom rendition model if you use one
>>> from wagtail.images.models import Rendition
>>> Rendition.objects.all().delete()
```

You can also directly use the image management command from the console for regenerating the renditions:

```
$ ./manage.py wagtail_update_image_renditions --purge
```

You can read more about this command from [*wagtail_update_image_renditions*](#)

Changing per-tag

It's also possible to have different JPEG and WebP qualities on individual tags by using `jpegquality` and `webpquality` filters. This will always override the default setting:

```
{% image page.photo_jpeg width-400 jpegquality-40 %}
{% image page.photo_webp width-400 webpquality-50 %}
```

Note that this will have no effect on PNG or GIF files. If you want all images to be low quality, you can use this filter with `format-jpeg` or `format-webp` (which forces all images to output in JPEG or WebP format):

```
{% image page.photo width-400 format-jpeg jpegquality-40 %}
{% image page.photo width-400 format-webp webpquality-50 %}
```

Generating image renditions in Python

All of the image transformations mentioned above can also be used directly in Python code. See [*Generating renditions in Python*](#).

1.2.4 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 need to add it into the search index. All pages, images and documents are indexed for you, so you can start searching them right away.

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

If you have a custom model that you would like to make searchable, see [*Indexing custom models*](#).

Updating the index

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

Signal handlers

`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`. These signal handlers are automatically registered when the `wagtail.search` app is loaded.

In some cases, you may not want your content to be automatically reindexed and instead rely on the `update_index` command for indexing. If you need to disable these signal handlers, use one of the following methods:

Disabling auto update signal handlers for a model

You can disable the signal handlers for an individual model by adding `search_auto_update = False` as an attribute on the model class.

Disabling auto update signal handlers for a search backend/whole site

You can disable the signal handlers for a whole search backend by setting the `AUTO_UPDATE` setting on the backend to `False`.

If all search backends have `AUTO_UPDATE` set to `False`, the signal handlers will be completely disabled for the whole site.

For documentation on the `AUTO_UPDATE` setting, see [AUTO_UPDATE](#).

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.

Note: The `update_index` command is also aliased as `wagtail_update_index`, for use when another installed package (such as `Haystack`) provides a conflicting `update_index` command. In this case, the other package's entry in `INSTALLED_APPS` should appear above `wagtail.search` so that its `update_index` command takes precedence over Wagtail's.

Indexing extra fields

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.search import index
from django.utils import timezone

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 used for performing full-text searches on your models, usually for text fields.

Options

- **partial_match** (boolean) - Setting this to true allows results to be matched on parts of words. For example, this is set on the title field by default, so a page titled `Hello World!` will be found if the user only types `Hel` into the search box.
- **boost** (int/float) - This allows you to set fields as being more important than others. Setting this to a high number on a field will cause 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.

Note: The PostgreSQL full text search only supports four weight levels (`A`, `B`, `C`, `D`). When the database search backend `wagtail.search.backends.database` is used on a PostgreSQL database, it will take all boost values in the project into consideration and group them into the four available weights.

This means that in this configuration there are effectively only four boost levels used for ranking the search results, even if more boost values have been used.

You can find out roughly which boost thresholds map to which weight in PostgreSQL by starting an new Django shell with `./manage.py shell` and inspecting `wagtail.search.backends.database.postgres.weights.BOOST_WEIGHTS`. You should see something like `[(10.0, 'A'), (7.166666666666666, 'B'),`

(4.33333333333333, 'C'), (1.5, 'D')]. Boost values above each threshold will be treated with the respective weight.

- **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.AutocompleteField`

These are used for autocomplete queries which match partial words. For example, a page titled Hello World! will be found if the user only types Hel into the search box.

This takes the exact same options as `index.SearchField` (with the exception of `partial_match`, which has no effect).

Note: Only index fields that are displayed in the search results with `index.AutocompleteField`. This allows users to see any words that were partial-matched on.

`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.

`index.RelatedFields`

This allows you to index fields from related objects. It works on all types of related fields, including their reverse accessors.

For example, if we have a book that has a `ForeignKey` to its author, we can nest the author's `name` and `date_of_birth` fields inside the book:

```
from wagtail.search import index

class Book(models.Model, index.Indexed):
    ...

    search_fields = [
        index.SearchField('title'),
        index.FilterField('published_date'),

        index.RelatedFields('author', [
            index.SearchField('name'),
            index.FilterField('date_of_birth'),
        ]),
    ]
```

This will allow you to search for books by their author's name.

It works the other way around as well. You can index an author's books, allowing an author to be searched for by the titles of books they've published:

```
from wagtail.search import index

class Author(models.Model, index.Indexed):
    ...

    search_fields = [
        index.SearchField('name'),
        index.FilterField('date_of_birth'),

        index.RelatedFields('books', [
            index.SearchField('title'),
            index.FilterField('published_date'),
        ]),
    ]
]
```

Filtering on `index.RelatedFields`

It's not possible to filter on any `index.FilterFields` within `index.RelatedFields` using the QuerySet API. However, the fields are indexed, so it should be possible to use them by querying Elasticsearch manually.

Filtering on `index.RelatedFields` with the QuerySet API is planned for a future release of Wagtail.

Indexing callables and other attributes

Search/filter fields do not need to be Django model fields. They can also be any method or attribute on your model class.

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

```
from wagtail.search import index

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

    is_private = models.BooleanField(choices=IS_PRIVATE_CHOICES)

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

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

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

```
class BookPage(Page):
    # ...
```

(continues on next page)

(continued from previous page)

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

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

Indexing custom models

Any Django model can be indexed and searched.

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

```
from wagtail.search import index

class Book(index.Indexed, models.Model):
    title = models.CharField(max_length=255)
    genre = models.CharField(max_length=255, choices=GENRE_CHOICES)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    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.search.backends import get_search_backend
>>> s = get_search_backend()

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

Searching

Searching QuerySets

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

Searching Pages

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

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

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

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

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

Before the `autocomplete()` method was introduced, the `search` method also did partial matching. This behaviour is will be deprecated and you should either switch to the new `autocomplete()` method or pass `partial_match=False` into the `search` method to opt-in to the new behaviour. The partial matching in `search()` will be completely removed in a future release.

Autocomplete searches

Wagtail provides a separate method which performs partial matching on specific autocomplete fields. This is useful for suggesting pages to the user in real-time as they type their query.

```
>>> EventPage.objects.live().autocomplete("Eve")
[<EventPage: Event 1>, <EventPage: Event 2>]
```

Note: This method should only be used for real-time autocomplete and actual search requests should always use the `search()` method.

Searching Images, Documents and custom models

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

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

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

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

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

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

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

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

Specifying the fields to search

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

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

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

Faceted search

Wagtail supports faceted search which is a kind of filtering based on a taxonomy field (such as category or page type).

The `.facet(field_name)` method returns an `OrderedDict`. The keys are the IDs of the related objects that have been referenced by the specified field, and the values are the number of references found for each ID. The results are ordered by number of references descending.

For example, to find the most common page types in the search results:

```
>>> Page.objects.search("Test").facet("content_type_id")

# Note: The keys correspond to the ID of a ContentType object; the values are the
```

(continues on next page)

(continued from previous page)

```
# number of pages returned for that type
OrderedDict([
    ('2', 4), # 4 pages have content_type_id == 2
    ('1', 2), # 2 pages have content_type_id == 1
])
```

Changing search behaviour

Search operator

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

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

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

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

Here’s an example of using the `operator` keyword argument:

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

Phrase searching

Phrase searching is used for finding whole sentence or phrase rather than individual terms. The terms must appear together and in the same order.

For example:

```
>>> from wagtail.search.query import Phrase

>>> Page.objects.search(Phrase("Hello world"))
[<Page: Hello World>]

>>> Page.objects.search(Phrase("World hello"))
[<Page: World Hello day>]
```

If you are looking to implement phrase queries using the double-quote syntax, see [Query string parsing](#).

Fuzzy matching

New in version 4.0.

Fuzzy matching will return documents which contain terms similar to the search term, as measured by a [Levenshtein edit distance](#).

A maximum of one edit (transposition, insertion, or removal of a character) is permitted for three to five letter terms, two edits for longer terms, and shorter terms must match exactly.

For example:

```
>>> from wagtail.search.query import Fuzzy

>>> Page.objects.search(Fuzzy("Hallo"))
[<Page: Hello World>]
```

Fuzzy matching is supported by the Elasticsearch search backend only.

Complex search queries

Through the use of search query classes, Wagtail also supports building search queries as Python objects which can be wrapped by and combined with other search queries. The following classes are available:

`PlainText(query_string, operator=None, boost=1.0)`

This class wraps a string of separate terms. This is the same as searching without query classes.

It takes a query string, operator and boost.

For example:

```
>>> from wagtail.search.query import PlainText
>>> Page.objects.search(PlainText("Hello world"))

# Multiple plain text queries can be combined. This example will match both "hello world"
# and "Hello earth"
>>> Page.objects.search(PlainText("Hello") & (PlainText("world") | PlainText("earth")))
```

Phrase(query_string)

This class wraps a string containing a phrase. See previous section for how this works.

For example:

```
# This example will match both the phrases "hello world" and "Hello earth"
>>> Page.objects.search(Phrase("Hello world") | Phrase("Hello earth"))
```

Boost(query, boost)

This class boosts the score of another query.

For example:

```
>>> from wagtail.search.query import PlainText, Boost

# This example will match both the phrases "hello world" and "Hello earth" but matches
# for "hello world" will be ranked higher
>>> Page.objects.search(Boost(Phrase("Hello world"), 10.0) | Phrase("Hello earth"))
```

Note that this isn't supported by the PostgreSQL or database search backends.

Query string parsing

The previous sections show how to construct a phrase search query manually, but a lot of search engines (Wagtail admin included, try it!) support writing phrase queries by wrapping the phrase with double-quotes. In addition to phrases, you might also want to allow users to add filters into the query using the colon syntax (`hello world published:yes`).

These two features can be implemented using the `parse_query_string` utility function. This function takes a query string that a user typed and returns a query object and dictionary of filters:

For example:

```
>>> from wagtail.search.utils import parse_query_string
>>> filters, query = parse_query_string('my query string "this is a phrase" this-is-
# a:filter', operator='and')

>>> filters
{
    'this-is-a': 'filter',
}

>>> query
And([
    PlainText("my query string", operator='and'),
    Phrase("this is a phrase"),
])
```

Here's an example of how this function can be used in a search view:

```
from wagtail.search.utils import parse_query_string

def search(request):
    query_string = request.GET['query']

    # Parse query
    filters, query = parse_query_string(query_string, operator='and')

    # Published filter
    # An example filter that accepts either `published:yes` or `published:no` and filters
    # the pages accordingly
    published_filter = filters.get('published')
    published_filter = published_filter and published_filter.lower()
    if published_filter in ['yes', 'true']:
        pages = pages.filter(live=True)
    elif published_filter in ['no', 'false']:
        pages = pages.filter(live=False)

    # Search
    pages = pages.search(query)

    return render(request, 'search_results.html', {'pages': pages})
```

Custom ordering

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

For example:

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

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

Annotating results with score

For each matched result, Elasticsearch calculates a “score”, which is a number that represents how relevant the result is based on the user's query. The results are usually ordered based on the score.

There are some cases where having access to the score is useful (such as programmatically combining two queries for different models). You can add the score to each result by calling the `.annotate_score(field)` method on the `SearchQuerySet`.

For example:

```
>>> events = EventPage.objects.search("Event").annotate_score("_score")
>>> for event in events:
...     print(event.title, event._score)
```

(continues on next page)

(continued from previous page)

```
...
("Easter", 2.5),
("Halloween", 1.7),
("Christmas", 1.5),
```

Note that the score itself is arbitrary and it is only useful for comparison of results for the same query.

An example page search view

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

```
# views.py

from django.shortcuts import render

from wagtail.models import Page
from wagtail.search.models import Query


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

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

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

And here's a template to go with it:

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

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

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

    {% if search_results %}
        <ul>
```

(continues on next page)

(continued from previous page)

```

{% for result in search_results %}
    <li>
        <h4><a href="{% pageurl result %}">{{ result }}</a></h4>
        {% if result.search_description %}
            {{ result.search_description|safe }}
        {% endif %}
    </li>
{% endfor %}
</ul>
{% elif search_query %}
    No results found
{% else %}
    Please type something into the search box
{% endif %}
{% endblock %}

```

Promoted search results

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

This functionality is provided by the `search_promotions` contrib module.

Backends

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

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

```

WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.search.backends.database',
    }
}

```

AUTO_UPDATE

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

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

```

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

```

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

ATOMIC_REBUILD

Warning: This option may not work on Elasticsearch version 5.4.x, due to a bug in the handling of aliases - please upgrade to 5.5 or later.

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

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

BACKEND

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

Database Backend (default)

`wagtail.search.backends.database`

The database search backend searches content in the database using the full text search features of the database backend in use (such as PostgreSQL FTS, SQLite FTS5). This backend is intended to be used for development and also should be good enough to use in production on sites that don't require any Elasticsearch specific features.

Elasticsearch Backend

Elasticsearch versions 5, 6 and 7 are supported. Use the appropriate backend for your version:

- `wagtail.search.backends.elasticsearch5` (Elasticsearch 5.x)
- `wagtail.search.backends.elasticsearch6` (Elasticsearch 6.x)
- `wagtail.search.backends.elasticsearch7` (Elasticsearch 7.x)

Prerequisites are the `Elasticsearch` service itself and, via pip, the `elasticsearch-py` package. The major version of the package must match the installed version of Elasticsearch:

```
pip install "elasticsearch>=5.0.0,<6.0.0" # for Elasticsearch 5.x
```

```
pip install "elasticsearch>=6.4.0,<7.0.0" # for Elasticsearch 6.x
```

```
pip install "elasticsearch>=7.0.0,<8.0.0" # for Elasticsearch 7.x
```

Warning: Version 6.3.1 of the Elasticsearch client library is incompatible with Wagtail. Use 6.4.0 or above.

The backend is configured in settings:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.search.backends.elasticsearch5',
        'URLS': ['http://localhost:9200'],
        'INDEX': 'wagtail',
        'TIMEOUT': 5,
        'OPTIONS': {},
        'INDEX_SETTINGS': {}
    }
}
```

Other than BACKEND, the keys are optional and default to the values shown. Any defined key in OPTIONS is passed directly to the Elasticsearch constructor as case-sensitive keyword argument (for example 'max_retries': 1).

A username and password may be optionally be supplied to the URL field to provide authentication credentials for the Elasticsearch service:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        ...
        'URLS': ['http://username:password@localhost:9200'],
        ...
    }
}
```

INDEX_SETTINGS is a dictionary used to override the default settings to create the index. The default settings are defined inside the `ElasticsearchSearchBackend` class in the module `wagtail/wagtail/search/backends/elasticsearch7.py`. Any new key is added, any existing key, if not a dictionary, is replaced with the new value. Here's a sample on how to configure the number of shards and setting the Italian LanguageAnalyzer as the default analyzer:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        ...
        'INDEX_SETTINGS': {
            'settings': {
                'index': {
                    'number_of_shards': 1,
                },
                'analysis': {
                    'analyzer': {
                        'default': {
                            'type': 'italian'
                        }
                    }
                }
            }
        }
    }
}
```

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

- Sign up for an account at Bonsai

- Use your Bonsai dashboard to create a Cluster.
- Configure URLs in the Elasticsearch entry in `WAGTAILSEARCH_BACKENDS` using the Cluster URL from your Bonsai dashboard
- Run `./manage.py update_index`

Amazon AWS Elasticsearch

The Elasticsearch backend is compatible with Amazon Elasticsearch Service, but requires additional configuration to handle IAM based authentication. This can be done with the `requests-aws4auth` package along with the following configuration:

```
from elasticsearch import RequestsHttpConnection
from requests_aws4auth import AWS4Auth

WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.search.backends.elasticsearch5',
        'INDEX': 'wagtail',
        'TIMEOUT': 5,
        'HOSTS': [{
            'host': 'YOURCLUSTER.REGION.es.amazonaws.com',
            'port': 443,
            'use_ssl': True,
            'verify_certs': True,
            'http_auth': AWS4Auth('ACCESS_KEY', 'SECRET_KEY', 'REGION', 'es'),
        }],
        'OPTIONS': {
            'connection_class': RequestsHttpConnection,
        },
    }
}
```

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 must first be added to the search index. This involves configuring the models and fields that you would like to index (which 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, so a user's search query can match the `Page` or `Image`'s extra content. See [Indexing extra fields](#).

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

Searching

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

See [Searching](#).

Backends

Wagtail provides two backends for storing the search index and performing search queries: one using the database's full-text search capabilities, and another using Elasticsearch. It's also possible to roll your own search backend.

See [Backends](#).

1.2.5 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 Django models which do not inherit the [Page](#) class and are thus not organised into the Wagtail tree. However, they can still be made editable by assigning panels and identifying the model as a snippet with the `register_snippet` class decorator.

Snippets lack many of the features of pages, such as being orderable in the Wagtail admin or having a defined URL. 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 model:

```
from django.db import models

from wagtail.admin.panels import FieldPanel
from wagtail.snippets.models import register_snippet

# ...

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

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

    def __str__(self):
        return self.text
```

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

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

Including snippets in template tags

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

First, add a new python file to a `templatetags` folder within your app - for example, `myproject/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 Advert

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 instance of the model, but for brevity we'll just use `Advert.objects.all()`.

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

```
{% load wagtailcore_tags demo_tags %}

...
{% block content %}

...
```

(continues on next page)

(continued from previous page)

```
{% adverts %}

{% endblock %}
```

Binding pages to snippets

In the above example, the list of adverts is a fixed list that is displayed via the custom template tag independent of any other content on the page. This might be what you want for a common panel in a sidebar, but, in another scenario, you might wish to display just one specific instance of a snippet on a particular page. This can be accomplished by defining a foreign key to the snippet model within your page model and adding a [FieldPanel](#) to the page's `content_panels` list. For example, if you wanted to display a specific advert on a BookPage instance:

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

    content_panels = Page.content_panels + [
        FieldPanel('advert'),
        # ...
    ]
```

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

To attach multiple adverts to a page, the `FieldPanel` 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.models import Page, Orderable

from modelcluster.fields import ParentalKey

# ...

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

    class Meta(Orderable.Meta):
        verbose_name = "advert placement"
        verbose_name_plural = "advert placements"

    panels = [
        FieldPanel('advert'),
```

(continues on next page)

(continued from previous page)

```

]

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

class BookPage(Page):
    # ...

    content_panels = Page.content_panels + [
        InlinePanel('advert_placements', label="Adverts"),
        # ...
    ]

```

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

```

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

```

Making snippets previewable

New in version 4.0: The `PreviewableMixin` class was introduced.

If a snippet model inherits from `PreviewableMixin`, Wagtail will automatically add a live preview panel in the editor. In addition to inheriting the mixin, the model must also override `get_preview_template()` or `serve_preview()`. For example, the `Advert` snippet could be made previewable as follows:

```

# ...

from wagtail.models import PreviewableMixin

# ...

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

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

    def get_preview_template(self, request, mode_name):
        return "demo/previews/advert.html"

```

With the following `demo/previews/advert.html` template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{ object.text }}</title>
  </head>
  <body>
    <a href="{{ object.url }}">{{ object.text }}</a>
  </body>
</html>
```

The variables available in the default context are `request` (a fake `HttpRequest` object) and `object` (the snippet instance). To customise the context, you can override the `get_preview_context()` method.

By default, the `serve_preview` method returns a `TemplateResponse` that is rendered using the `request` object, the template returned by `get_preview_template`, and the context object returned by `get_preview_context`. You can override the `serve_preview` method to customise the rendering and/or routing logic.

Similar to pages, you can define multiple preview modes by overriding the `preview_modes` property. For example, the following `Advert` snippet has two preview modes:

```
# ...

from wagtail.models import PreviewableMixin

# ...

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

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

    @property
    def preview_modes(self):
        return PreviewableMixin.DEFAULT_PREVIEW_MODES + [("alt", "Alternate")]

    def get_preview_template(self, request, mode_name):
        templates = {
            "": "demo/previews/advert.html", # Default preview mode
            "alt": "demo/previews/advert_alt.html", # Alternate preview mode
        }
        return templates.get(mode_name, templates[""])

    def get_preview_context(self, request, mode_name):
        context = super().get_preview_context(request, mode_name)
        if mode_name == "alt":
            context["extra_context"] = "Alternate preview mode"
        return context
```

Making snippets searchable

If a snippet model inherits from `wagtail.search.index.Indexed`, as described in [Indexing custom models](#), Wagtail will automatically add a search box to the chooser interface for that snippet type. For example, the `Advert` snippet could be made searchable as follows:

```
# ...

from wagtail.search import index

# ...

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

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

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

Saving revisions of snippets

New in version 4.0: The `RevisionMixin` class was introduced.

If a snippet model inherits from `RevisionMixin`, Wagtail will automatically save revisions when you save any changes in the snippets admin. In addition to inheriting the mixin, it is recommended to define a `GenericRelation` to the `Revision` model and override the `revisions` property to return the `GenericRelation`. For example, the `Advert` snippet could be made revisable as follows:

```
# ...

from django.contrib.contenttypes.fields import GenericRelation
from wagtail.models import RevisionMixin

# ...

@register_snippet
class Advert(RevisionMixin, models.Model):
    url = models.URLField(null=True, blank=True)
    text = models.CharField(max_length=255)
    _revisions = GenericRelation("wagtailcore.Revision", related_query_name="advert")

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

(continues on next page)

(continued from previous page)

```
@property
def revisions(self):
    return self._revisions
```

The `RevisionMixin` includes a `latest_revision` field that needs to be added to your database table. Make sure to run the `makemigrations` and `migrate` management commands after making the above changes to apply the changes to your database.

With the `RevisionMixin` applied, any changes made from the snippets admin will create an instance of the `Revision` model that contains the state of the snippet instance. The revision instance is attached to the `audit log` entry of the edit action, allowing you to revert to a previous revision or compare the changes between revisions from the snippet history page.

You can also save revisions programmatically by calling the `save_revision()` method. After applying the mixin, it is recommended to call this method (or save the snippet in the admin) at least once for each instance of the snippet that already exists (if any), so that the `latest_revision` field is populated in the database table.

Saving draft changes of snippets

New in version 4.0: The `DraftStateMixin` class was introduced.

If a snippet model inherits from `DraftStateMixin`, Wagtail will automatically change the “Save” action menu in the snippets admin to “Save draft” and add a new “Publish” action menu. Any changes you save in the snippets admin will be saved as revisions and will not be reflected to the “live” snippet instance until you publish the changes. For example, the `Advert` snippet could save draft changes by defining it as follows:

```
# ...

from django.contrib.contenttypes.fields import GenericRelation
from wagtail.models import DraftStateMixin, RevisionMixin

# ...

@register_snippet
class Advert(DraftStateMixin, RevisionMixin, models.Model):
    url = models.URLField(null=True, blank=True)
    text = models.CharField(max_length=255)
    _revisions = GenericRelation("wagtailcore.Revision", related_query_name="advert")

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

    @property
    def revisions(self):
        return self._revisions
```

You can publish revisions programmatically by calling `instance.publish(revision)` or by calling `revision.publish()`. After applying the mixin, it is recommended to publish at least one revision for each instance of the snippet that already exists (if any), so that the `latest_revision` and `live_revision` fields are populated in the database table.

Warning: Wagtail does not yet have a mechanism to prevent editors from including unpublished (“draft”) snippets in pages. When including a `DraftStateMixin`-enabled snippet in pages, make sure that you add necessary checks to handle how a draft snippet should be rendered (e.g. by checking its `live` field). We are planning to improve this in the future.

Note: The `DraftStateMixin` includes fields used by Wagtail’s publishing mechanism that may currently be inapplicable for snippets. For example, the scheduled publishing fields (i.e. `go_live_at`, `expire_at`, and `expired`) are added to snippet models that inherit from the mixin, but the scheduled publishing feature itself is not yet officially supported for snippets.

We are introducing these fields early to make adding new features easier in the future. Until the features become available and officially supported, we recommend explicitly defining the `panels` in your snippets with only your relevant model fields.

Tagging snippets

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

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

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

@register_snippet
class Advert(ClusterableModel):
    # ...
    tags = TaggableManager(through=AdvertTag, blank=True)

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

The [documentation on tagging pages](#) has more information on how to use tags in views.

1.2.6 How to use StreamField for mixed content

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

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

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

Using StreamField

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

```
from django.db import models

from wagtail.models import Page
from wagtail.fields import StreamField
from wagtail import blocks
from wagtail.admin.panels import FieldPanel
from wagtail.images.blocks import ImageChooserBlock

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

    content_panels = Page.content_panels + [
        FieldPanel('author'),
        FieldPanel('date'),
        FieldPanel('body'),
    ]
```

In this example, the body field of `BlogPage` is defined as a `StreamField` where authors can compose content from three different block types: headings, paragraphs, and images, which can be used and repeated in any order. The block types available to authors are defined as a list of `(name, block_type)` tuples: ‘name’ is used to identify the block type within templates, and should follow the standard Python conventions for variable names: lower-case and underscores, no spaces.

You can find the complete list of available block types in the [StreamField block reference](#).

Note: StreamField is not a direct replacement for other field types such as RichTextField. If you need to migrate an existing field to StreamField, refer to [Migrating RichTextFields to StreamField](#).

Changed in version 3.0: The `use_json_field=True` argument was added. This indicates that the database’s native JSONField support should be used for this field, and is a temporary measure to assist in migrating StreamFields created on earlier Wagtail versions; it will become the default in a future release.

Template rendering

StreamField provides an HTML representation for the stream content as a whole, as well as for each individual block. To include this HTML into your page, use the `{% include_block %}` tag:

```
{% load wagtailcore_tags %}

...
{% include_block page.body %}
```

In the default rendering, each block of the stream is wrapped in a `<div class="block-my_block_name">` element (where `my_block_name` is the block name given in the StreamField definition). If you wish to provide your own HTML markup, you can instead iterate over the field's value, and invoke `{% include_block %}` on each block in turn:

```
{% load wagtailcore_tags %}

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

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

```
{% load wagtailcore_tags %}

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

Combining blocks

In addition to using the built-in block types directly within StreamField, it's possible to construct new block types by combining sub-blocks in various ways. Examples of this could include:

- An “image with caption” block consisting of an image chooser and a text field
- A “related links” section, where an author can provide any number of links to other pages
- A slideshow block, where each slide may be an image, text or video, arranged in any order

Once a new block type has been built up in this way, you can use it anywhere where a built-in block type would be used - including using it as a component for yet another block type. For example, you could define an image gallery block where each item is an “image with caption” block.

StructBlock

StructBlock allows you to group several ‘child’ blocks together to be presented as a single block. The child blocks are passed to StructBlock as a list of `(name, block_type)` tuples:

```
body = StreamField([
    ('person', blocks.StructBlock([
        ('first_name', blocks.CharBlock()),
        ('surname', blocks.CharBlock()),
        ('photo', ImageChooserBlock(required=False)),
        ('biography', blocks.RichTextBlock()),
    ])),
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
], use_json_field=True)
```

When reading back the content of a StreamField (such as when rendering a template), the value of a StructBlock is a dict-like object with keys corresponding to the block names given in the definition:

```
<article>
    {% for block in page.body %}
        {% if block.block_type == 'person' %}
            <div class="person">
                {% image block.value.photo width-400 %}
                <h2>{{ block.value.first_name }} {{ block.value.surname }}</h2>
                {{ block.value.biography }}
            </div>
        {% else %}
            (rendering for other block types)
        {% endif %}
    {% endfor %}
</article>
```

Subclassing StructBlock

Placing a StructBlock's list of child blocks inside a StreamField definition can often be hard to read, and makes it difficult for the same block to be reused in multiple places. As an alternative, StructBlock can be subclassed, with the child blocks defined as attributes on the subclass. The 'person' block in the above example could be rewritten as:

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

PersonBlock can then be used in a StreamField definition in the same way as the built-in block types:

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

Block icons

In the menu that content authors use to add new blocks to a StreamField, each block type has an associated icon. For StructBlock and other structural block types, a placeholder icon is used, since the purpose of these blocks is specific to your project. To set a custom icon, pass the option `icon` as either a keyword argument to StructBlock, or an attribute on a Meta class:

```
body = StreamField([
    ('person', blocks.StructBlock([
        ('first_name', blocks.CharBlock()),
        ('surname', blocks.CharBlock()),
        ('photo', ImageChooserBlock(required=False)),
        ('biography', blocks.RichTextBlock()),
    ], icon='user')),
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
], use_json_field=True)
```

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

    class Meta:
        icon = 'user'
```

For a list of the recognised icon identifiers, see the [UI Styleguide](#).

ListBlock

ListBlock defines a repeating block, allowing content authors to insert as many instances of a particular block type as they like. For example, a ‘gallery’ block consisting of multiple images can be defined as follows:

```
body = StreamField([
    ('gallery', blocks.ListBlock(ImageChooserBlock())),
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
], use_json_field=True)
```

When reading back the content of a StreamField (such as when rendering a template), the value of a ListBlock is a list of child values:

```
<article>
    {% for block in page.body %}
        {% if block.block_type == 'gallery' %}
            <ul class="gallery">
                {% for img in block.value %}
                    <li>{% image img width=400 %}</li>
                {% endfor %}
            </ul>
        {% else %}
            (rendering for other block types)
        {% endif %}
    {% endfor %}
</article>
```

StreamBlock

StreamBlock defines a set of child block types that can be mixed and repeated in any sequence, via the same mechanism as StreamField itself. For example, a carousel that supports both image and video slides could be defined as follows:

```
body = StreamField([
    ('carousel', blocks.StreamBlock([
        ('image', ImageChooserBlock()),
        ('video', EmbedBlock()),
    ])),
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
], use_json_field=True)
```

StreamBlock can also be subclassed in the same way as StructBlock, with the child blocks being specified as attributes on the class:

```
class CarouselBlock(blocks.StreamBlock):
    image = ImageChooserBlock()
    video = EmbedBlock()

    class Meta:
        icon = 'image'
```

A StreamBlock subclass defined in this way can also be passed to a StreamField definition, instead of passing a list of block types. This allows setting up a common set of block types to be used on multiple page types:

```
class CommonContentBlock(blocks.StreamBlock):
    heading = blocks.CharBlock(form_classname="title")
    paragraph = blocks.RichTextBlock()
    image = ImageChooserBlock()

class BlogPage(Page):
    body = StreamField(CommonContentBlock(), use_json_field=True)
```

When reading back the content of a StreamField, the value of a StreamBlock is a sequence of block objects with block_type and value properties, just like the top-level value of the StreamField itself.

```
<article>
    {% for block in page.body %}
        {% if block.block_type == 'carousel' %}
            <ul class="carousel">
                {% for slide in block.value %}
                    {% if slide.block_type == 'image' %}
                        <li class="image">{% image slide.value width-200 %}</li>
                    {% else %}
                        <li class="video">{% include_block slide %}</li>
                    {% endif %}
                {% endfor %}
            </ul>
        {% else %}
            (rendering for other block types)
        {% endif %}
    {% endfor %}
</article>
```

Limits

By default, a StreamField can contain an unlimited number of blocks. The min_num and max_num options on StreamField or StreamBlock allow you to set a minimum or maximum number of blocks:

```
body = StreamField([
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
], min_num=2, max_num=5, use_json_field=True)
```

Or equivalently:

```
class CommonContentBlock(blocks.StreamBlock):
    heading = blocks.CharBlock(form_classname="title")
    paragraph = blocks.RichTextBlock()
    image = ImageChooserBlock()

class Meta:
```

(continues on next page)

(continued from previous page)

```
min_num = 2
max_num = 5
```

The `block_counts` option can be used to set a minimum or maximum count for specific block types. This accepts a dict, mapping block names to a dict containing either or both of `min_num` and `max_num`. For example, to permit between 1 and 3 ‘heading’ blocks:

```
body = StreamField([
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
], block_counts={
    'heading': {'min_num': 1, 'max_num': 3},
}, use_json_field=True)
```

Or equivalently:

```
class CommonContentBlock(blocks.StreamBlock):
    heading = blocks.CharBlock(form_classname="title")
    paragraph = blocks.RichTextBlock()
    image = ImageChooserBlock()

    class Meta:
        block_counts = {
            'heading': {'min_num': 1, 'max_num': 3},
        }
```

Per-block templates

By default, each block is rendered using simple, minimal HTML markup, or no markup at all. For example, a `CharBlock` value is rendered as plain text, while a `ListBlock` outputs its child blocks in a `` wrapper. To override this with your own custom HTML rendering, you can pass a `template` argument to the block, giving the filename of a template file to be rendered. This is particularly useful for custom block types derived from `StructBlock`:

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

Or, when defined as a subclass of `StructBlock`:

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

(continues on next page)

(continued from previous page)

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

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

```
{% load wagtailimages_tags %}

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

Since `first_name`, `surname`, `photo` and `biography` are defined as blocks in their own right, this could also be written as:

```
{% load wagtailcore_tags wagtailimages_tags %}

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

Writing `{{ my_block }}` is roughly equivalent to `% include_block my_block %`, but the short form is more restrictive, as it does not pass variables from the calling template such as `request` or `page`; for this reason, it is recommended that you only use it for simple values that do not render HTML of their own. For example, if our `PersonBlock` used the template:

```
{% load wagtailimages_tags %}

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

    {% if request.user.is_authenticated %}
        <a href="#">Contact this person</a>
    {% endif %}

    {{ value.biography }}
</div>
```

then the `request.user.is_authenticated` test would not work correctly when rendering the block through a `{{ ... }}` tag:

```
{# Incorrect: #}

{% for block in page.body %}
    {% if block.block_type == 'person' %}
        <div>
            {{ block }}
```

(continues on next page)

(continued from previous page)

```

</div>
{%
  endif %}
{%
  endfor %}

{# Correct: #-}

{% for block in page.body %}
  {% if block.block_type == 'person' %}
    <div>
      {% include_block block %}
    </div>
  {% endif %}
{%
  endfor %}

```

Like Django's `{% include %}` tag, `{% include_block %}` also allows passing additional variables to the included template, through the syntax `{% include_block my_block with foo="bar" %}`:

```

{# In page template: #-}

{% for block in page.body %}
  {% if block.block_type == 'person' %}
    {% include_block block with classname="important" %}
  {% endif %}
{%
  endfor %}

{# In PersonBlock template: #-}

<div class="{{ classname }}>
  ...
</div>

```

The syntax `{% include_block my_block with foo="bar" only %}` is also supported, to specify that no variables from the parent template other than `foo` will be passed to the child template.

As well as passing variables from the parent template, block subclasses can pass additional template variables of their own by overriding the `get_context` method:

```

import datetime

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

    def get_context(self, value, parent_context=None):
        context = super().get_context(value, parent_context=parent_context)
        context['is_happening_today'] = (value['date'] == datetime.date.today())
        return context

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

```

In this example, the variable `is_happening_today` will be made available within the block template. The `parent_context` keyword argument is available when the block is rendered through an `{% include_block %}` tag, and is a dict of variables passed from the calling template.

All block types, not just `StructBlock`, support the `template` property. However, for blocks that handle basic Python data types, such as `CharBlock` and `IntegerField`, there are some limitations on where the template will take effect. For further details, see [About StreamField BoundBlocks and values](#).

Customisations

All block types implement a common API for rendering their front-end and form representations, and storing and retrieving values to and from the database. By subclassing the various block classes and overriding these methods, all kinds of customisations are possible, from modifying the layout of `StructBlock` form fields to implementing completely new ways of combining blocks. For further details, see [How to build custom StreamField blocks](#).

Modifying StreamField data

A StreamField's value behaves as a list, and blocks can be inserted, overwritten and deleted before saving the instance back to the database. A new item can be written to the list as a tuple of `(block_type, value)` - when read back, it will be returned as a `BoundBlock` object.

```
# Replace the first block with a new block of type 'heading'
my_page.body[0] = ('heading', "My story")

# Delete the last block
del my_page.body[-1]

# Append a rich text block to the stream
from wagtail.rich_text import RichText
my_page.body.append(('paragraph', RichText("<p>And they all lived happily ever after.</p>\n")))

# Save the updated data back to the database
my_page.save()
```

Retrieving blocks by name

New in version 4.0: The `blocks_by_name` and `first_block_by_name` methods were added.

StreamField values provide a `blocks_by_name` method for retrieving all blocks of a given name:

```
my_page.body.blocks_by_name('heading') # returns a list of 'heading' blocks
```

Calling `blocks_by_name` with no arguments returns a dict-like object, mapping block names to the list of blocks of that name. This is particularly useful in template code, where passing arguments isn't possible:

```
<h2>Table of contents</h2>
<ol>
    {% for heading_block in page.body.blocks_by_name.heading %}
        <li>{{ heading_block.value }}</li>
    {% endfor %}
</ol>
```

The `first_block_by_name` method returns the first block of the given name in the stream, or `None` if no matching block is found:

```
hero_image = my_page.body.first_block_by_name('image')
```

`first_block_by_name` can also be called without arguments to return a dict-like mapping:

```
<div class="hero-image">{{ page.body.first_block_by_name.image }}</div>
```

Migrating RichTextFields to StreamField

If you change an existing RichTextField to a StreamField, the database migration will complete with no errors, since both fields use a text column within the database. However, StreamField uses a JSON representation for its data, so the existing text requires an extra conversion step in order to become accessible again. For this to work, the StreamField needs to include a RichTextBlock as one of the available block types. Create the migration as normal using `./manage.py makemigrations`, then edit it as follows (in this example, the ‘body’ field of the `demo.BlogPage` model is being converted to a StreamField with a RichTextBlock named `rich_text`):

Note: This migration cannot be used if the StreamField has the `use_json_field` argument set to True. To migrate, set the `use_json_field` argument to False first, migrate the data, then set it back to True.

```
# -*- coding: utf-8 -*-
from django.db import models, migrations
from wagtail.rich_text import RichText

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

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

class Migration(migrations.Migration):

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

(continues on next page)

(continued from previous page)

```

operations = [
    # leave the generated AlterField intact!
    migrations.AlterField(
        model_name='BlogPage',
        name='body',
        field=wagtail.fields.StreamField([
            ('rich_text', wagtail.blocks.
RichTextBlock()),
        ],
        migrations.RunPython(
            convert_to_streamfield,
            convert_to_richtext,
        ),
    ],
]

```

Note that the above migration will work on published Page objects only. If you also need to migrate draft pages and page revisions, then edit the migration as in the following example instead:

```

# -*- coding: utf-8 -*-
import json

from django.core.serializers.json import DjangoJSONEncoder
from django.db import migrations, models

from wagtail.rich_text import RichText


def page_to_streamfield(page):
    changed = False
    if page.body.raw_text and not page.body:
        page.body = [('rich_text', {'rich_text': RichText(page.body.raw_text)})]
        changed = True
    return page, changed


def pagerevision_to_streamfield(revision_data):
    changed = False
    body = revision_data.get('body')
    if body:
        try:
            json.loads(body)
        except ValueError:
            revision_data['body'] = json.dumps(
                [
                    {
                        "value": {"rich_text": body},
                        "type": "rich_text"
                    },
                ],
                cls=DjangoJSONEncoder)
            changed = True
    else:
        # It's already valid JSON. Leave it.
        pass

```

(continues on next page)

(continued from previous page)

```

    return revision_data, changed

def page_to_richtext(page):
    changed = False
    if page.body.raw_text is None:
        raw_text = ''.join([
            child.value['rich_text'].source for child in page.body
            if child.block_type == 'rich_text'
        ])
        page.body = raw_text
        changed = True
    return page, changed

def pagerevision_to_richtext(revision_data):
    changed = False
    body = revision_data.get('body', 'definitely non-JSON string')
    if body:
        try:
            body_data = json.loads(body)
        except ValueError:
            # It's not apparently a StreamField. Leave it.
            pass
    else:
        raw_text = ''.join([
            child['value']['rich_text'] for child in body_data
            if child['type'] == 'rich_text'
        ])
        revision_data['body'] = raw_text
        changed = True
    return revision_data, changed

def convert(apps, schema_editor, page_converter, pagerevision_converter):
    BlogPage = apps.get_model("demo", "BlogPage")
    for page in BlogPage.objects.all():

        page, changed = page_converter(page)
        if changed:
            page.save()

        for revision in page.revisions.all():
            revision_data = revision.content
            revision_data, changed = pagerevision_converter(revision_data)
            if changed:
                revision.content = revision_data
                revision.save()

def convert_to_streamfield(apps, schema_editor):
    return convert(apps, schema_editor, page_to_streamfield, pagerevision_to_streamfield)

```

(continues on next page)

(continued from previous page)

```
def convert_to_richtext(apps, schema_editor):
    return convert(apps, schema_editor, page_to_richtext, pagerevision_to_richtext)

class Migration(migrations.Migration):

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

    operations = [
        # leave the generated AlterField intact!
        migrations.AlterField(
            model_name='BlogPage',
            name='body',
            field=wagtail.fields.StreamField([('rich_text', wagtail.blocks.RichTextBlock())]),
        ),
        migrations.RunPython(
            convert_to_streamfield,
            convert_to_richtext,
        ),
    ]
```

1.2.7 Permissions

Wagtail adapts and extends the [Django permission system](#) to cater for the needs of website content creation, such as moderation workflows, and multiple teams working on different areas of a site (or multiple sites within the same Wagtail installation). Permissions can be configured through the ‘Groups’ area of the Wagtail admin interface, under ‘Settings’.

Page permissions

Permissions can be attached at any point in the page tree, and propagate down the tree. For example, if a site had the page tree:

```
MegaCorp/
  About us
  Offices/
    UK
    France
    Germany
```

then a group with ‘edit’ permissions on the ‘Offices’ page would automatically receive the ability to edit the ‘UK’, ‘France’ and ‘Germany’ pages. Permissions can be set globally for the entire tree by assigning them on the ‘root’ page - since all pages must exist underneath the root node, and the root cannot be deleted, this permission will cover all pages that exist now and in future.

Whenever a user creates a page through the Wagtail admin, that user is designated as the owner of that page. Any user with ‘add’ permission has the ability to edit pages they own, as well as adding new ones. This is in recognition of the fact that creating pages is typically an iterative process involving creating a number of draft versions - giving a user the ability to create a draft but not letting them subsequently edit it would not be very useful. Ability to edit a page also implies the ability to delete it; unlike Django’s standard permission model, there is no distinct ‘delete’ permission.

The full set of available permission types is as follows:

- **Add** - grants the ability to create new subpages underneath this page (provided the page model permits this - see [Parent page / subpage type rules](#)), and to edit and delete pages owned by the current user. Published pages cannot be deleted unless the user also has ‘publish’ permission.
- **Edit** - grants the ability to edit and delete this page, and any pages underneath it, regardless of ownership. A user with only ‘edit’ permission may not create new pages, only edit existing ones. Published pages cannot be deleted unless the user also has ‘publish’ permission.
- **Publish** - grants the ability to publish and unpublish this page and/or its children. A user without publish permission cannot directly make changes that are visible to visitors of the website; instead, they must submit their changes for moderation. Publish permission is independent of edit permission; a user with only publish permission will not be able to make any edits of their own.
- **Bulk delete** - allows a user to delete pages that have descendants, in a single operation. Without this permission, a user has to delete the descendant pages individually before deleting the parent. This is a safeguard against accidental deletion. This permission must be used in conjunction with ‘add’ / ‘edit’ permission, as it does not provide any deletion rights of its own; it only provides a ‘shortcut’ for the permissions the user has already. For example, a user with just ‘add’ and ‘bulk delete’ permissions will only be able to bulk-delete if all the affected pages are owned by that user, and are unpublished.
- **Lock** - grants the ability to lock or unlock this page (and any pages underneath it) for editing, preventing users from making any further edits to it.

Drafts can be viewed only if the user has either Edit or Publish permission.

Image / document permissions

The permission rules for images and documents work on a similar basis to pages. Images and documents are considered to be ‘owned’ by the user who uploaded them; a user with ‘add’ permission also has the ability to edit items they own; and deletion is considered equivalent to editing rather than having a specific permission type.

Access to specific sets of images and documents can be controlled by setting up *collections*. By default all images and documents belong to the ‘root’ collection, but users with appropriate permissions can create new collections the Settings -> Collections area of the admin interface. Permissions set on ‘root’ apply to all collections, so a user with ‘edit’ permission for images in the root collection can edit all images; permissions set on other collections only apply to that collection and any of its sub-collections.

The ‘choose’ permission for images and documents determines which collections are visible within the chooser interface used to select images and document links for insertion into pages (and other models, such as snippets). Typically, all users are granted choose permission for all collections, allowing them to use any uploaded image or document on pages they create, but this permission can be limited to allow creating collections that are only visible to specific groups.

Collection management permissions

Permission for managing collections themselves can be attached at any point in the collection tree. The available collection management permissions are as follows:

- **Add** - grants the ability to create new collections underneath this collection.
- **Edit** - grants the ability to edit the name of the collection, change its location in the collection tree, and to change the privacy settings for documents within this collection.
- **Delete** - grants the ability to delete collections that were added below this collection. *Note:* a collection must have no subcollections under it and the collection itself must be empty before it can be deleted.

Note: Users are not allowed to move or delete the collection that is used to assign them permission to manage collections.

Displaying custom permissions in the admin

Most permissions will automatically show up in the wagtail admin Group edit form, however, you can also add them using the `register_permissions` hook (see [register_permissions](#)).

FieldPanel permissions

Permissions can be used to restrict access to fields within the editor interface. See [permission on FieldPanel](#).

1.3 Advanced topics

1.3.1 Images

Generating renditions in Python

Rendered versions of original images generated by the Wagtail `{% image %}` template tag are called “renditions”, and are stored as new image files in the site’s `[media]/images` directory on the first invocation.

Image renditions can also be generated dynamically from Python via the native `get_rendition()` method, for example:

```
newimage = myimage.get_rendition('fill-300x150|jpegquality-60')
```

If `myimage` had a filename of `foo.jpg`, a new rendition of the image file called `foo.fill-300x150.jpegquality-60.jpg` would be generated and saved into the site’s `[media]/images` directory. Argument options are identical to the `{% image %}` template tag’s filter spec, and should be separated with `|`.

The generated Rendition object will have properties specific to that version of the image, such as `url`, `width` and `height`, so something like this could be used in an API generator, for example:

```
url = myimage.get_rendition('fill-300x186|jpegquality-60').url
```

Properties belonging to the original image from which the generated Rendition was created, such as `title`, can be accessed through the Rendition’s `image` property:

```
>>> newimage.image.title
'Blue Sky'
>>> newimage.image.is_landscape()
True
```

See also: [How to use images in templates](#)

Prefetching image renditions

When using a queryset to render a list of images or objects with images, you can prefetch the renditions needed with a single additional query. For long lists of items, or where multiple renditions are used for each item, this can provide a significant boost to performance.

New in version 4.0: The `prefetch_renditions` method is only applicable in Wagtail versions 4.0 and above.

Image QuerySets

When working with an Image QuerySet, you can make use of Wagtail's built-in `prefetch_renditions` queryset method to prefetch the renditions needed.

For example, say you were rendering a list of all the images uploaded by a certain user:

```
def get_images_uploaded_by_user(user):
    return ImageModel.objects.filter(uploaded_by_user=user)
```

The above can be modified slightly to prefetch the renditions of the images returned:

```
def get_images_uploaded_by_user(user):
    return ImageModel.objects.filter(uploaded_by_user=user).prefetch_renditions()
```

The above will prefetch all renditions even if we may not need them.

If images in your project tend to have very large numbers of renditions, and you know in advance the ones you need, you might want to consider specifying a set of filters to the `prefetch_renditions` method and only select the renditions you need for rendering. For example:

```
def get_images_uploaded_by_user(user):
    # Only specify the renditions required for rendering
    return ImageModel.objects.filter(uploaded_by_user=user).prefetch_renditions(
        "fill-700x586", "min-600x400", "max-940x680"
    )
```

Non Image Querysets

If you're working with a non Image Model, you can make use of Django's built-in `prefetch_related()` queryset method to prefetch renditions.

For example, say you were rendering a list of events (with thumbnail images for each). Your code might look something like this:

```
def get_events():
    return EventPage.objects.live().select_related("listing_image")
```

The above can be modified slightly to prefetch the renditions for listing images:

```
def get_events():
    return EventPage.objects.live().select_related("listing_image").prefetch_related(
        "listing_image_renditions")
```

If you know in advance the renditions you'll need, you can filter the renditions queryset to use:

```
from django.db.models import Prefetch
from wagtail.images import get_image_model


def get_events():
    Image = get_image_model()
    filters = ["fill-300x186", "fill-600x400", "fill-940x680"]

    # `Prefetch` is used to fetch only the required renditions
    prefetch_images_and_renditions = Prefetch(
        "listing_image",
        queryset=Image.objects.prefetch_renditions(*filters)
    )
    return EventPage.objects.live().prefetch_related(prefetch_images_and_renditions)
```

Model methods involved in rendition generation

New in version 3.0: The following method references are only applicable to Wagtail versions 3.0 and above.

The following `AbstractImage` model methods are involved in finding and generating a renditions. If using a custom image model, you can customise the behaviour of either of these methods by overriding them on your model:

`class wagtail.images.models.AbstractImage`

`get_rendition(filter: Union[wagtail.images.models.Filter, str]) → wagtail.images.models.AbstractRendition`

Returns a `Rendition` instance with a `file` field value (an image) reflecting the supplied `filter` value and focal point values from this object.

Note: If using custom image models, an instance of the custom rendition model will be returned.

`find_existing_rendition(filter: wagtail.images.models.Filter) → wagtail.images.models.AbstractRendition`

Returns an existing `Rendition` instance with a `file` field value (an image) reflecting the supplied `filter` value and focal point values from this object.

If no such rendition exists, a `DoesNotExist` error is raised for the relevant model.

Note: If using custom image models, an instance of the custom rendition model will be returned.

`create_rendition(filter: wagtail.images.models.Filter) → wagtail.images.models.AbstractRendition`

Creates and returns a `Rendition` instance with a `file` field value (an image) reflecting the supplied `filter` value and focal point values from this object.

This method is usually called by `Image.get_rendition()`, after first checking that a suitable rendition does not already exist.

Note: If using custom image models, an instance of the custom rendition model will be returned.

`generate_rendition_file(filter: wagtail.images.models.Filter) → django.core.files.base.File`

Generates an in-memory image matching the supplied `filter` value and focal point value from this object, wraps it in a `File` object with a suitable filename, and returns it. The return value is used as the `file` field value for rendition objects saved by `AbstractImage.create_rendition()`.

NOTE: The responsibility of generating the new image from the original falls to the supplied `filter` object. If you want to do anything custom with rendition images (for example, to preserve metadata from the original image), you might want to consider swapping out `filter` for an instance of a custom Filter subclass of your design.

Animated GIF support

Pillow, Wagtail’s default image library, doesn’t support animated GIFs.

To get animated GIF support, you will have to [install Wand](#). Wand is a binding to ImageMagick so make sure that has been installed as well.

When installed, Wagtail will automatically use Wand for resizing GIF files but continue to resize other images with Pillow.

Image file formats

Using the picture element

The `picture` element can be used with the `format-<type>` image operation to specify different image formats and let the browser choose the one it prefers. For example:

```
{% load wagtailimages_tags %}

<picture>
    {% image myimage width-1000 format-webp as image_webp %}
    <source srcset="{{ image_webp.url }}" type="image/webp">

    {% image myimage width-1000 format-png as image_png %}
    <source srcset="{{ image_png.url }}" type="image/png">

    {% image myimage width-1000 format-png %}
</picture>
```

Customising output formats

By default all `bmp` and `webp` images are converted to the `png` format when no image output format is given.

The default conversion mapping can be changed by setting the `WAGTAILIMAGES_FORMAT_CONVERSIONS` to a dictionary which maps the input type to an output type.

For example:

```
WAGTAILIMAGES_FORMAT_CONVERSIONS = {
    'bmp': 'jpeg',
    'webp': 'webp',
}
```

will convert `bmp` images to `jpeg` and disable the default `webp` to `png` conversion.

Custom image models

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

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

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

Here's an example:

```
# models.py
from django.db import models

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

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

    # To add a caption field:
    # caption = models.CharField(max_length=255, blank=True)

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

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

    class Meta:
        unique_together = (
            ('image', 'filter_spec', 'focal_point_key'),
        )
```

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

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

Migrating from the builtin image model

When changing an existing site to use a custom image model, no images will be copied to the new model automatically. Copying old images to the new model would need to be done manually with a [data migration](#).

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

Referring to the image model

`wagtail.images.get_image_model()`

Get the image model from the `WAGTAILIMAGES_IMAGE_MODEL` setting. Useful for developers making Wagtail plugins that need the image model. Defaults to the standard `Image` model if no custom model is defined.

`wagtail.images.get_image_model_string()`

Get the dotted app.Model name for the image model as a string. Useful for developers making Wagtail plugins that need to refer to the image model, such as in foreign keys, but the model itself is not required.

Changing rich text representation

The HTML representation of an image in rich text can be customised - for example, to display captions or custom fields.

To do this requires subclassing `Format` (see [Image Formats in the Rich Text Editor](#)), and overriding its `image_to_html` method.

You may then register formats of your subclass using `register_image_format` as usual.

```
# image_formats.py
from wagtail.images.formats import Format, register_image_format

class SubclassedImageFormat(Format):

    def image_to_html(self, image, alt_text, extra_attributes=None):
        custom_html = # the custom HTML representation of your image here
                      # in Format, the image's rendition.img_tag(extra_attributes) is
        ↪used to generate the HTML
                      # representation

        return custom_html

register_image_format(
    SubclassedImageFormat('subclassed_format', 'Subclassed Format', classnames, filter_
    ↪spec)
)
```

As an example, let's say you want the alt text to be displayed as a caption for the image as well:

```
# image_formats.py
from django.utils.html import format_html
from wagtail.images.formats import Format, register_image_format

class CaptionedImageFormat(Format):

    def image_to_html(self, image, alt_text, extra_attributes=None):
        default_html = super().image_to_html(image, alt_text, extra_attributes)

        return format_html("{}<figcaption>{}</figcaption>", default_html, alt_text)
```

(continues on next page)

(continued from previous page)

```
register_image_format(
    CaptionedImageFormat('captioned_fullwidth', 'Full width captioned', 'bodytext-image',
    ↵ 'width-750')
)
```

Note: Any custom HTML image features will not be displayed in the Draftail editor, only on the published page.

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 third-party tools to detect faces/features in an image when the image is uploaded. The detected features are 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.

Installation

Two third-party tools are known to work with Wagtail: One based on [OpenCV](#) for general feature detection and one based on [Rustface](#) for face detection.

OpenCV on Debian/Ubuntu

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

There is more than one way to install these components, but in each case you will need to test that both OpenCV itself *and* the Python interface have been correctly installed.

Install opencv-python

`opencv-python` is available on PyPI. It includes a Python interface to OpenCV, as well as the statically-built OpenCV binaries themselves.

To install:

```
$ pip install opencv-python
```

Depending on what else is installed on your system, this may be all that is required. On lighter-weight Linux systems, you may need to identify and install missing system libraries (for example, a slim version of Debian Stretch requires `libsdl2 libxrender1 libxext6` to be installed with `apt`).

Install a system-level package

A system-level package can take care of all of the required components. Check what is available for your operating system. For example, `python-opencv` is available for Debian; it installs OpenCV itself, and sets up Python bindings.

However, it may make incorrect assumptions about how you're using Python (for example, which version you're using) - test as described below.

Testing the installation

Test the installation:

```
python3
>>> import cv2
```

An error such as:

```
ImportError: libSM.so.6: cannot open shared object file: No such file or directory
```

indicates that a required system library (in this case `libsm6`) has not been installed.

On the other hand,

```
ModuleNotFoundError: No module named 'cv2'
```

means that the Python components have not been set up correctly in your Python environment.

If you don't get an import error, installation has probably been successful.

Rustface

`Rustface` is Python library with prebuilt wheel files provided for Linux and macOS. Although implemented in Rust it is pip-installable:

```
$ pip install wheel
$ pip install rustface
```

Registering with Willow

Rustface provides a plug-in that needs to be registered with `Willow`.

This should be done somewhere that gets run on application startup:

```
from willow.registry import registry
import rustface.willow

registry.register_plugin(rustface.willow)
```

For example, in an app's `AppConfig.ready`.

Cropping

The face detection algorithm produces a focal area that is tightly cropped to the face rather than the whole head.

For images with a single face this can be okay in some cases (thumbnails for example), it might be overly tight for “headshots”. Image renditions can encompass more of the head by reducing the crop percentage (-c<percentage>), at the end of the resize-rule, down to as low as 0%:

```
{% image page.photo fill-200x200-c0 %}
```

Switching on feature detection in Wagtail

Once installed, you need to set the `WAGTAILIMAGES_FEATURE_DETECTION_ENABLED` setting to `True` to automatically detect faces/features whenever a new image is uploaded in to Wagtail or when an image without a focal point is saved (this is done via a pre-save signal handler):

```
# settings.py

WAGTAILIMAGES_FEATURE_DETECTION_ENABLED = True
```

Manually running feature detection

If you already have images in your Wagtail site and would like to run feature detection on them, or you want to apply feature detection selectively when the `WAGTAILIMAGES_FEATURE_DETECTION_ENABLED` is set to `False` you can run it manually using the `get_suggested_focal_point()` method on the `Image` model.

For example, you can manually run feature detection on all images by running the following code in the python shell:

```
from wagtail.images import get_image_model

Image = get_image_model()

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

Dynamic image serve view

In most cases, developers wanting to generate image renditions in Python should use the `get_rendition()` method. See [Generating renditions in Python](#).

If you need to be able to generate image versions for an *external* system such as a blog or mobile app, Wagtail provides a view for dynamically generating renditions of images by calling a unique URL.

The view takes an image id, filter spec and security signature in the URL. If these parameters are valid, it serves an image file matching that criteria.

Like the `{% image %}` tag, the rendition is generated on the first call and subsequent calls are served from a cache.

Setup

Add an entry for the view into your URLs configuration:

```
from wagtail.images.views.serve import ServeView

urlpatterns = [
    ...
    re_path(r'^images/([^/]*)/(\d*)/([^/]*)/[^/]*$', ServeView.as_view(), name='wagtailimages_serve'),
    ...
    # Ensure that the wagtailimages_serve line appears above the default Wagtail page serving route
    re_path(r'', include(wagtail_urls)),
]
```

Usage

Image URL generator UI

When the dynamic serve view is enabled, an image URL generator in the admin interface becomes available automatically. This can be accessed through the edit page of any image by clicking the “URL generator” button on the right hand side.

This interface allows editors to generate URLs to cropped versions of the image.

Generating dynamic image URLs in Python

Dynamic image URLs can also be generated using Python code and served to a client over an API or used directly in the template.

One advantage of using dynamic image URLs in the template is that they do not block the initial response while rendering like the `{% image %}` tag does.

The `generate_image_url` function in `wagtail.images.views.serve` is a convenience method to generate a dynamic image URL.

Here's an example of this being used in a view:

```
def display_image(request, image_id):
    image = get_object_or_404(Image, id=image_id)

    return render(request, 'display_image.html', {
        'image_url': generate_image_url(image, 'fill-100x100')
    })
```

Image operations can be chained by joining them with a `|` character:

```
return render(request, 'display_image.html', {
    'image_url': generate_image_url(image, 'fill-100x100|jpegquality-40')
})
```

In your templates:

```
{% load wagtailimages_tags %}
...
<!-- Get the url for the image scaled to a width of 400 pixels: -->
{% image_url page.photo "width-400" %}

<!-- Again, but this time as a square thumbnail: -->
{% image_url page.photo "fill-100x100|jpegquality-40" %}

<!-- This time using our custom image serve view: -->
{% image_url page.photo "width-400" "mycustomview_serve" %}
```

You can pass an optional view name that will be used to serve the image through. The default is `wagtailimages_serve`

Advanced configuration

Making the view redirect instead of serve

By default, the view will serve the image file directly. This behaviour can be changed to a 301 redirect instead which may be useful if you host your images externally.

To enable this, pass `action='redirect'` into the `ServeView.as_view()` method in your urls configuration:

```
from wagtail.images.views.serve import ServeView

urlpatterns = [
    ...
    re_path(r'^images/([^/]*)/(\d*)/([^/]*)/[^/]*$', ServeView.as_view(action='redirect'),
    name='wagtailimages_serve'),
]
```

Integration with django-sendfile

`django-sendfile` offloads the job of transferring the image data to the web server instead of serving it directly from the Django application. This could greatly reduce server load in situations where your site has many images being downloaded but you're unable to use a [Caching proxy](#) or a CDN.

You firstly need to install and configure `django-sendfile` and configure your web server to use it. If you haven't done this already, please refer to the [installation docs](#).

To serve images with `django-sendfile`, you can use the `SendFileView` class. This view can be used out of the box:

```
from wagtail.images.views.serve import SendFileView
```

(continues on next page)

(continued from previous page)

```
urlpatterns = [
    ...
    re_path(r'^images/([^/]*)/(\d*)/([^/]*)/[^/]*$', SendFileView.as_view(), name='wagtailimages_serve'),
]
```

You can customise it to override the backend defined in the SENDFILE_BACKEND setting:

```
from wagtail.images.views.serve import SendFileView
from project.sendfile_backends import MyCustomBackend

class MySendFileView(SendFileView):
    backend = MyCustomBackend
```

You can also customise it to serve private files. For example, if the only need is to be authenticated (Django >= 1.9):

```
from django.contrib.auth.mixins import LoginRequiredMixin
from wagtail.images.views.serve import SendFileView

class PrivateSendFileView(LoginRequiredMixin, SendFileView):
    raise_exception = True
```

Focal points

Focal points are used to indicate to Wagtail the area of an image that contains the subject. This is used by the fill filter to focus the cropping on the subject, and avoid cropping into it.

Focal points can be defined manually by a Wagtail user, or automatically by using face or feature detection.

Setting the background-position inline style based on the focal point

When using a Wagtail image as the background of an element, you can use the .background_position_style attribute on the rendition to position the rendition based on the focal point in the image:

```
{% image page.image width-1024 as image %}

<div style="background-image: url('{{ image.url }}'); {{ image.background_position_style }}">
</div>
```

Accessing the focal point in templates

You can access the focal point in the template by accessing the `.focal_point` attribute of a rendition:

```
{% load wagtailimages %}

{% image myimage width-800 as myrendition %}

![{{ myimage.title }}]({{ myrendition.url }})>
```

Title generation on upload

When uploading an image, Wagtail takes the filename, removes the file extension, and populates the title field. This section is about how to customise this filename to title conversion.

The filename to title conversion is used on the single file widget, multiple upload widget, and within chooser modals.

You can also customise this [same behaviour for documents](#).

You can customise the resolved value of this title using a JavaScript `event listener` which will listen to the '`wagtail:images-upload`' event.

The simplest way to add JavaScript to the editor is via the [`insert_global_admin_js` hook](#), however any JavaScript that adds the event listener will work.

DOM Event

The event name to listen for is '`wagtail:images-upload`'. It will be dispatched on the image upload form. The event's `detail` attribute will contain:

- `data` - An object which includes the `title` to be used. It is the filename with the extension removed.
- `maxTitleLength` - An integer (or `null`) which is the maximum length of the Image model title field.
- `filename` - The original filename without the extension removed.

To modify the generated Image title, access and update `event.detail.data.title`, no return value is needed.

For single image uploads, the custom event will only run if the title does not already have a value so that we do not overwrite whatever the user has typed.

You can prevent the default behaviour by calling `event.preventDefault()`. For the single upload page or modals, this will not pre-fill any value into the title. For multiple upload, this will avoid any title submission and use the filename title only (with file extension) as a title is required to save the image.

The event will ‘bubble’ up so that you can simply add a global `document` listener to capture all of these events, or you can scope your listener or handler logic as needed to ensure you only adjust titles in some specific scenarios.

See MDN for more information about [custom JavaScript events](#).

Code Examples

Removing any url unsafe characters from the title

```
# wagtail_hooks.py
from django.utils.safestring import mark_safe

from wagtail import hooks

@hooks.register("insert_global_admin_js")
def get_global_admin_js():
    return mark_safe(
        """
<script>
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:images-upload', function(event) {
        var newTitle = (event.detail.data.title || "").replace(/[^a-zA-Z0-9\s-]/g, " ");
        event.detail.data.title = newTitle;
    });
});
</script>
"""
    )
)
```

Changing generated titles on the page editor only to remove dashes/underscores

Using the `insert_editor_js hook` instead so that this script will not run on the Image upload page, only on page editors.

```
# wagtail_hooks.py
from django.utils.safestring import mark_safe

from wagtail import hooks

@hooks.register("insert_editor_js")
def get_global_admin_js():
    return mark_safe(
        """
<script>
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:images-upload', function(event) {
        // replace dashes/underscores with a space
        var newTitle = (event.detail.data.title || "").replace(/\s/_/-/g, " ");
        event.detail.data.title = newTitle;
    });
});
</script>
"""
    )
)
```

(continues on next page)

(continued from previous page)

)

Stopping pre-filling of title based on filename

```
# wagtail_hooks.py
from django.utils.safestring import mark_safe

from wagtail import hooks

@hooks.register("insert_global_admin_js")
def get_global_admin_js():
    return mark_safe(
        """
<script>
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:images-upload', function(event) {
        // will stop title pre-fill on single file uploads
        // will set the multiple upload title to the filename (with extension)
        event.preventDefault();
    });
});
</script>
"""
    )

```

1.3.2 Documents

Custom document model

An alternate Document model can be used to add custom behaviour and additional fields.

You need to complete the following steps in your project to do this:

- Create a new document model that inherits from `wagtail.documents.models.AbstractDocument`. This is where you would add additional fields.
- Point `WAGTAILDOCS_DOCUMENT_MODEL` to the new model.

Here's an example:

```
# models.py
from django.db import models

from wagtail.documents.models import Document, AbstractDocument

class CustomDocument(AbstractDocument):
    # Custom field example:
    source = models.CharField(
        max_length=255,
        blank=True,
```

(continues on next page)

(continued from previous page)

```

    null=True
)

admin_form_fields = Document.admin_form_fields + (
    # Add all custom fields names to make them appear in the form:
    'source',
)

```

Then in your settings module:

```

# Ensure that you replace app_label with the app you placed your custom
# model in.
WAGTAILDOCS_DOCUMENT_MODEL = 'app_label.CustomDocument'

```

Note: Migrating from the builtin document model

When changing an existing site to use a custom document model, no documents will be copied to the new model automatically. Copying old documents to the new model would need to be done manually with a [data migration](#).

Any templates that reference the builtin document model will still continue to work as before.

Referring to the document model

wagtail.documents.get_document_model()

Get the document model from the WAGTAILDOCS_DOCUMENT_MODEL setting. Defaults to the standard Document model if no custom model is defined.

wagtail.documents.get_document_model_string()

Get the dotted app.Model name for the document model as a string. Useful for developers making Wagtail plugins that need to refer to the document model, such as in foreign keys, but the model itself is not required.

Title generation on upload

When uploading a file (document), Wagtail takes the filename, removes the file extension, and populates the title field. This section is about how to customise this filename to title conversion.

The filename to title conversion is used on the single file widget, multiple upload widget, and within chooser modals.

You can also customise this [same behaviour for images](#).

You can customise the resolved value of this title using a JavaScript [event listener](#) which will listen to the 'wagtail:documents-upload' event.

The simplest way to add JavaScript to the editor is via the [insert_global_admin_js hook](#), however any JavaScript that adds the event listener will work.

DOM Event

The event name to listen for is 'wagtail:documents-upload'. It will be dispatched on the document upload form. The event's `detail` attribute will contain:

- `data` - An object which includes the `title` to be used. It is the filename with the extension removed.
- `maxTitleLength` - An integer (or null) which is the maximum length of the `Document` model title field.
- `filename` - The original filename without the extension removed.

To modify the generated `Document` title, access and update `event.detail.data.title`, no return value is needed.

For single document uploads, the custom event will only run if the title does not already have a value so that we do not overwrite whatever the user has typed.

You can prevent the default behaviour by calling `event.preventDefault()`. For the single upload page or modals, this will not pre-fill any value into the title. For multiple uploads, this will avoid any title submission and use the filename title only (with file extension) as a title is required to save the document.

The event will ‘bubble’ up so that you can simply add a global `document` listener to capture all of these events, or you can scope your listener or handler logic as needed to ensure you only adjust titles in some specific scenarios.

See MDN for more information about [custom JavaScript events](#).

Code Examples

Adding the file extension to the start of the title

```
# wagtail_hooks.py
from django.utils.safestring import mark_safe

from wagtail import hooks

@hooks.register("insert_global_admin_js")
def get_global_admin_js():
    return mark_safe(
        """
<script>
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:documents-upload', function(event) {
        var extension = (event.detail.filename.match(/^.([^.]*?)\?=\?|#/|$/) || [
            ''
        ])[1];
        var newTitle = '(' + extension.toUpperCase() + ')' + (event.detail.data.title +
        '');
        event.detail.data.title = newTitle;
    });
});
</script>
"""
    )
}
```

Changing generated titles on the page editor only to remove dashes/underscores

Using the `insert_editor_js` hook instead so that this script will not run on the Document upload page, only on page editors.

```
# wagtail_hooks.py
from django.utils.safestring import mark_safe

from wagtail import hooks

@hooks.register("insert_editor_js")
def get_global_admin_js():
    return mark_safe(
        """
<script>
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:documents-upload', function(event) {
        // replace dashes/underscores with a space
        var newTitle = (event.detail.data.title || "").replace(/(\s|_|-)/g, " ");
        event.detail.data.title = newTitle;
    });
});
</script>
"""
    )
)
```

Stopping pre-filling of title based on filename

```
# wagtail_hooks.py
from django.utils.safestring import mark_safe

from wagtail import hooks

@hooks.register("insert_global_admin_js")
def get_global_admin_js():
    return mark_safe(
        """
<script>
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:documents-upload', function(event) {
        // will stop title pre-fill on single file uploads
        // will set the multiple upload title to the filename (with extension)
        event.preventDefault();
    });
});
</script>
"""
    )
)
```

1.3.3 Embedded content

Wagtail supports generating embed code from URLs to content on external providers such as Youtube or Twitter. By default, Wagtail will fetch the embed code directly from the relevant provider's site using the oEmbed protocol.

Wagtail has a built-in list of the most common providers and this list can be changed *with a setting*. Wagtail also supports fetching embed code using *Embedby* and *custom embed finders*.

Embedding content on your site

Wagtail's embeds module should work straight out of the box for most providers. You can use any of the following methods to call the module:

Rich text

Wagtail's default rich text editor has a “media” icon that allows embeds to be placed into rich text. You don't have to do anything to enable this; just make sure the rich text field's content is being passed through the `|richtext` filter in the template as this is what calls the embeds module to fetch and nest the embed code.

EmbedBlock StreamField block type

The `EmbedBlock` block type allows embeds to be placed into a `StreamField`.

The `max_width` and `max_height` arguments are sent to the provider when fetching the embed code.

For example:

```
from wagtail.embeds.blocks import EmbedBlock

class MyStreamField(blocks.StreamBlock):
    ...

    embed = EmbedBlock(max_width=800, max_height=400)
```

{% embed %} tag

Syntax: `{% embed <url> [max_width=<max width>] %}`

You can nest embeds into a template by passing the URL and an optional `max_width` argument to the `{% embed %}` tag.

The `max_width` argument is sent to the provider when fetching the embed code.

```
{% load wagtailembeds_tags %}

{# Embed a YouTube video #}
{% embed 'https://www.youtube.com/watch?v=Ffu-2jEdLPw' %}

{# This tag can also take the URL from a variable #}
{% embed page.video_url %}
```

From Python

You can also call the internal `get_embed` function that takes a URL string and returns an `Embed` object (see model documentation below). This also takes a `max_width` keyword argument that is sent to the provider when fetching the embed code.

```
from wagtail.embeds.embeds import get_embed
from wagtail.embeds.exceptions import EmbedException

try:
    embed = get_embed('https://www.youtube.com/watch?v=Ffu-2jEdLPw')

    print(embed.html)
except EmbedException:
    # Cannot find embed
    pass
```

Configuring embed “finders”

Embed finders are the modules within Wagtail that are responsible for producing embed code from a URL.

Embed finders are configured using the `WAGTAILEMBEDS_FINDERS` setting. This is a list of finder configurations that are each run in order until one of them successfully returns an embed:

The default configuration is:

```
WAGTAILEMBEDS_FINDERS = [
    {
        'class': 'wagtail.embeds.finders.oembed'
    }
]
```

oEmbed (default)

The default embed finder fetches the embed code directly from the content provider using the oEmbed protocol. Wagtail has a built-in list of providers which are all enabled by default. You can find that provider list at the following link:

https://github.com/wagtail/wagtail/blob/main/wagtail/embeds/oembed_providers.py

Customising the provider list

You can limit which providers may be used by specifying the list of providers in the finder configuration.

For example, this configuration will only allow content to be nested from Vimeo and Youtube. It also adds a custom provider:

```
from wagtail.embeds.oembed_providers import youtube, vimeo

# Add a custom provider
# Your custom provider must support oEmbed for this to work. You should be
# able to find these details in the provider's documentation.
# - 'endpoint' is the URL of the oEmbed endpoint that Wagtail will call
```

(continues on next page)

(continued from previous page)

```
# - 'urls' specifies which patterns
my_custom_provider = {
    'endpoint': 'https://customvideosite.com/oembed',
    'urls': [
        '^http(?:s)?://(?:www\\.)?customvideosite\\.com/[^#?/]+/videos/.+$',
    ]
}

WAGTAILEMBEDS_FINDERS = [
{
    'class': 'wagtail.embeds.finders.oembed',
    'providers': [youtube, vimeo, my_custom_provider],
}
]
```

Customising an individual provider

Multiple finders can be chained together. This can be used for customising the configuration for one provider without affecting the others.

For example, this is how you can instruct YouTube to return videos in HTTPS (which must be done explicitly for YouTube):

```
from wagtail.embeds.oembed_providers import youtube

WAGTAILEMBEDS_FINDERS = [
    # Fetches YouTube videos but puts ``?scheme=https`` in the GET parameters
    # when calling YouTube's oEmbed endpoint
    {
        'class': 'wagtail.embeds.finders.oembed',
        'providers': [youtube],
        'options': {'scheme': 'https'}
    },
    # Handles all other oEmbed providers the default way
    {
        'class': 'wagtail.embeds.finders.oembed',
    }
]
```

How Wagtail uses multiple finders

If multiple providers can handle a URL (for example, a YouTube video was requested using the configuration above), the topmost finder is chosen to perform the request.

Wagtail will not try to run any other finder, even if the chosen one didn't return an embed.

Facebook and Instagram

As of October 2020, Facebook deprecated their public oEmbed APIs. If you would like to embed Facebook or Instagram posts in your site, you will need to use the new authenticated APIs. This requires you to set up a Facebook Developer Account and create a Facebook App that includes the *oEmbed Product*. Instructions for creating the necessary app are in the requirements sections of the [Facebook](#) and [Instagram](#) documentation.

As of June 2021, the *oEmbed Product* has been replaced with the *oEmbed Read* feature. In order to embed Facebook and Instagram posts your app must activate the *oEmbed Read* feature. Furthermore the app must be reviewed and accepted by Facebook. You can find the announcement in the [API changelog](#).

Apps that activated the oEmbed Product before June 8, 2021 need to activate the oEmbed Read feature and review their app before September 7, 2021.

Once you have your app access tokens (App ID and App Secret), add the Facebook and/or Instagram finders to your `WAGTAILEMBEDS_FINDERS` setting and configure them with the App ID and App Secret from your app:

```
WAGTAILEMBEDS_FINDERS = [
    {
        'class': 'wagtail.embeds.finders.facebook',
        'app_id': 'YOUR FACEBOOK APP_ID HERE',
        'app_secret': 'YOUR FACEBOOK APP_SECRET HERE',
    },
    {
        'class': 'wagtail.embeds.finders.instagram',
        'app_id': 'YOUR INSTAGRAM APP_ID HERE',
        'app_secret': 'YOUR INSTAGRAM APP_SECRET HERE',
    },
    # Handles all other oEmbed providers the default way
    {
        'class': 'wagtail.embeds.finders.oembed',
    }
]
```

By default, Facebook and Instagram embeds include some JavaScript that is necessary to fully render the embed. In certain cases, this might not be something you want - for example, if you have multiple Facebook embeds, this would result in multiple script tags. By passing `'omitscript': True` in the configuration, you can indicate that these script tags should be omitted from the embed HTML. Note that you will then have to take care of loading this script yourself.

Embed.ly

[Embed.ly](#) is a paid-for service that can also provide embeds for sites that do not implement the oEmbed protocol.

They also provide some helpful features such as giving embeds a consistent look and a common video playback API which is useful if your site allows videos to be hosted on different providers and you need to implement custom controls for them.

Wagtail has built in support for fetching embeds from Embed.ly. To use it, first pip install the [Embedly python package](#).

Now add an embed finder to your `WAGTAILEMBEDS_FINDERS` setting that uses the `wagtail.embeds.finders.oembed` class and pass it your API key:

```
WAGTAILEMBEDS_FINDERS = [
    {
        'class': 'wagtail.embeds.finders.embedly',
        'key': 'YOUR EMBED.LY KEY HERE'
    }
]
```

Custom embed finder classes

For complete control, you can create a custom finder class.

Here's a stub finder class that could be used as a skeleton; please read the docstrings for details of what each method does:

```
from wagtail.embeds.finders.base import EmbedFinder

class ExampleFinder(EmbedFinder):
    def __init__(self, **options):
        pass

    def accept(self, url):
        """
        Returns True if this finder knows how to fetch an embed for the URL.

        This should not have any side effects (no requests to external servers)
        """
        pass

    def find_embed(self, url, max_width=None):
        """
        Takes a URL and max width and returns a dictionary of information about the
        content to be used for embedding it on the site.

        This is the part that may make requests to external APIs.
        """
        # TODO: Perform the request

        return {
            'title': "Title of the content",
            'author_name': "Author name",
            'provider_name': "Provider name (such as YouTube, Vimeo, etc)",
            'type': "Either 'photo', 'video', 'link' or 'rich'",
            'thumbnail_url': "URL to thumbnail image",
            'width': width_in_pixels,
            'height': height_in_pixels,
            'html': "<h2>The Embed HTML</h2>",
        }
```

Once you've implemented all of those methods, you just need to add it to your `WAGTAILEMBEDS_FINDERS` setting:

```
WAGTAILEMBEDS_FINDERS = [
    {
        'class': 'path.to.your.finder.class.here',
        # Any other options will be passed as kwargs to the __init__ method
    }
]
```

The Embed model

class wagtail.embeds.models.Embed

Embeds are fetched only once and stored in the database so subsequent requests for an individual embed do not hit the embed finders again.

url

(text)

The URL of the original content of this embed.

max_width

(integer, nullable)

The max width that was requested.

type

(text)

The type of the embed. This can be either ‘video’, ‘photo’, ‘link’ or ‘rich’.

html

(text)

The HTML content of the embed that should be placed on the page

title

(text)

The title of the content that is being embedded.

author_name

(text)

The author name of the content that is being embedded.

provider_name

(text)

The provider name of the content that is being embedded.

For example: YouTube, Vimeo

thumbnail_url

(text)

a URL to a thumbnail image of the content that is being embedded.

width

(integer, nullable)

The width of the embed (images and videos only).

height

(integer, nullable)

The height of the embed (images and videos only).

last_updated

(datetime)

The Date/time when this embed was last fetched.

Deleting embeds

As long as your embeds configuration is not broken, deleting items in the `Embed` model should be perfectly safe to do. Wagtail will automatically repopulate the records that are being used on the site.

You may want to do this if you've changed from oEmbed to Embedly or vice-versa as the embed code they generate may be slightly different and lead to inconsistency on your site.

1.3.4 How to add Wagtail into an existing Django project

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

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

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

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

Middleware (`settings.py`)

```
MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',

    'wagtail.contrib.redirects.middleware.RedirectMiddleware',
]
```

Wagtail depends on the default set of Django middleware modules, to cover basic security and functionality such as login sessions. One additional middleware module is provided:

RedirectMiddleware

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

Apps (`settings.py`)

```
INSTALLED_APPS = [

    'myapp', # your own app

    'wagtail.contrib.forms',
    'wagtail.contrib.redirects',
    'wagtail.embeds',
    'wagtail.sites',
    'wagtail.users',
    'wagtail.snippets',
    'wagtail.documents',
    'wagtail.images',
    'wagtail.search',
    'wagtail.admin',
    'wagtail',

    'taggit',
    'modelcluster',

    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

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.

Wagtail Apps

wagtail

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

wagtail.admin

The administration interface for Wagtail, including page edit handlers.

wagtail.documents

The Wagtail document content type.

wagtail.snippets

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

wagtail.users

User editing interface.

wagtail.images

The Wagtail image content type.

wagtail.embeds

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

wagtail.search

Search framework for Page content. See *Search*.

wagtail.sites

Management UI for Wagtail sites.

wagtail.contrib.redirects

Admin interface for creating arbitrary redirects on your site.

wagtail.contrib.forms

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

Third-Party Apps

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](#).

URL Patterns

```
from django.contrib import admin

from wagtail import urls as wagtail_urls
from wagtail.admin import urls as wagtailadmin_urls
from wagtail.documents import urls as wagtaildocs_urls

urlpatterns = [
    path('django-admin/', admin.site.urls),
```

(continues on next page)

(continued from previous page)

```

path('admin/', include(wagtailadmin_urls)),
path('documents/', include(wagtailedocs_urls)),

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

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

```

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

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

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

Ready to Use Example Configuration Files

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

`settings.py`

```

import os

PROJECT_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
BASE_DIR = os.path.dirname(PROJECT_DIR)

DEBUG = True

# Application definition

INSTALLED_APPS = [
    'myapp',

    'wagtail.contrib.forms',
    'wagtail.contrib.redirects',
    'wagtail.embeds',
    'wagtail.sites',
    'wagtail.users',
    'wagtail.snippets',
    'wagtail.documents',
    'wagtail.images',
    'wagtail.search',
]

```

(continues on next page)

(continued from previous page)

```
'wagtail.admin',
'wagtail',

'taggit',
'modelcluster',

'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
]

MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',

    'wagtail.contrib.redirects.middleware.RedirectMiddleware',
]

ROOT_URLCONF = 'myproject.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(PROJECT_DIR, 'templates'),
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
WSGI_APPLICATION = 'myproject.wsgi.application'

# Database

DATABASES = {
    'default': {

```

(continues on next page)

(continued from previous page)

```

'ENGINE': 'django.db.backends.postgresql',
'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
}
}

# Internationalization

LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
USE_I18N = True
USE_L10N = True
USE_TZ = True

# Static files (CSS, JavaScript, Images)

STATICFILES_FINDERS = [
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
]

STATICFILES_DIRS = [
    os.path.join(PROJECT_DIR, 'static'),
]

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

MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'

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/stable/topics/email/#email-backends
EMAIL_BACKEND = 'django.core.mail.backends.dummy.EmailBackend'

# Hosts/domain names that are valid for this site; required if DEBUG is False
ALLOWED_HOSTS = []

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

```

(continues on next page)

(continued from previous page)

```
EMAIL SUBJECT PREFIX = '[Wagtail] '

INTERNAL_IPS = ('127.0.0.1', '10.0.2.2')

# A sample logging configuration. The only tangible logging
# performed by this configuration is to send an email to
# the site admins on every HTTP 500 error when DEBUG=False.
# See https://docs.djangoproject.com/en/stable/topics/logging for
# more details on how to customise 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'

# Replace the search backend
#WAGTAILSEARCH_BACKENDS = {
#    'default': {
#        'BACKEND': 'wagtail.search.backends.elasticsearch5',
#        'INDEX': 'myapp'
#    }
#}

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

# Wagtail email notification format
```

(continues on next page)

(continued from previous page)

```
# WAGTAILADMIN_NOTIFICATION_USE_HTML = True

# Reverse the default case-sensitive handling of tags
TAGGIT_CASE_INSENSITIVE = True
```

urls.py

```
from django.urls import include, path, re_path
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 import urls as wagtail_urls
from wagtail.admin import urls as wagtailadmin_urls
from wagtail.documents import urls as wagtaildocs_urls

urlpatterns = [
    path('django-admin/', admin.site.urls),

    path('admin/', include(wagtailadmin_urls)),
    path('documents/', include(wagtaildocs_urls)),

    # For anything not caught by a more specific rule above, hand over to
    # Wagtail's serving mechanism
    re_path(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 += [
        path('favicon.ico', RedirectView.as_view(url=settings.STATIC_URL + 'myapp/images/
    ↪ favicon.ico'))
    ]
```

1.3.5 Deploying Wagtail

On your server

Wagtail is straightforward to deploy on modern Linux-based distributions, and should run with any of the combinations detailed in Django's [deployment documentation](#). See the section on *performance* for the non-Python services we recommend.

On Divio Cloud

Divio Cloud is a Dockerised cloud hosting platform for Python/Django that allows you to launch and deploy Wagtail projects in minutes. With a free account, you can create a Wagtail project. Choose from a:

- site based on the [Wagtail Bakery project](#), or
- brand new Wagtail project (see the [how to get started notes](#)).

Divio Cloud also hosts a live [Wagtail Bakery demo](#) (no account required).

On PythonAnywhere

[PythonAnywhere](#) is a Platform-as-a-Service (PaaS) focused on Python hosting and development. It allows developers to quickly develop, host, and scale applications in a cloud environment. Starting with a free plan they also provide MySQL and PostgreSQL databases as well as very flexible and affordable paid plans, so there's all you need to host a Wagtail site. To get quickly up and running you may use the [wagtail-pythonanywhere-quickstart](#).

On Google Cloud

[Google Cloud](#) is an Infrastructure-as-a-Service (IaaS) that offers multiple managed products, supported by Python client libraries, to help you build, deploy, and monitor your applications. You can deploy Wagtail, or any Django application, in a number of ways, including on [App Engine](#) or [Cloud Run](#).

On alwaysdata

[alwaysdata](#) is a Platform-as-a-Service (PaaS) providing Public and Private Cloud offers. Starting with a free plan they provide MySQL/PostgreSQL databases, emails, free SSL certificates, included backups, etc.

To get your Wagtail application running you may:

- Install Wagtail from [alwaysdata Marketplace](#)
- Configure a Django application

On other PaaS and IaaS

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!

Deployment tips

Static files

As with all Django projects, static files are only served by the Django application server during development, when running through the `manage.py runserver` command. In production, these need to be handled separately at the web server level. See [Django's documentation on deploying static files](#).

The JavaScript and CSS files used by the Wagtail admin frequently change between releases of Wagtail - it's important to avoid serving outdated versions of these files due to browser or server-side caching, as this can cause hard-to-diagnose issues. We recommend enabling `ManifestStaticFilesStorage` in the `STATICFILES_STORAGE` setting - this ensures that different versions of files are assigned distinct URLs.

User Uploaded Files

Wagtail follows [Django's conventions for managing uploaded files](#). So by default, Wagtail uses Django's built-in `FileSystemStorage` class which stores files on your site's server, in the directory specified by the `MEDIA_ROOT` setting. Alternatively, Wagtail can be configured to store uploaded images and documents on a cloud storage service such as Amazon S3; this is done through the `DEFAULT_FILE_STORAGE` setting in conjunction with an add-on package such as [django-storages](#).

When using `FileSystemStorage`, image urls are constructed starting from the path specified by the `MEDIA_URL`. In most cases, you should configure your web server to serve image files directly (without passing through Django/Wagtail). When using one of the cloud storage backends, images urls go directly to the cloud storage file url. If you would like to serve your images from a separate asset server or CDN, you can [*configure the image serve view*](#) to redirect instead.

Document serving is controlled by the `WAGTAILDOCS_SERVE_METHOD` method. When using `FileSystemStorage`, documents are stored in a `documents` subdirectory within your site's `MEDIA_ROOT`. If all your documents are public, you can set the `WAGTAILDOCS_SERVE_METHOD` to `direct` and configure your web server to serve the files itself. However, if you use Wagtail's [*Collection Privacy settings*](#) to restrict access to some or all of your documents, you may or may not want to configure your web server to serve the documents directly. The default setting is `redirect` which allows Wagtail to perform any configured privacy checks before offloading serving the actual document to your web server or CDN. This means that Wagtail constructs document links that pass through Wagtail, but the final url in the user's browser is served directly by your web server. If a user bookmarks this url, they will be able to access the file without passing through Wagtail's privacy checks. If this is not acceptable, you may want to set the `WAGTAILDOCS_SERVE_METHOD` to `serve_view` and configure your web server so it will not serve document files itself. If you are serving documents from the cloud and need to enforce privacy settings, you should make sure the documents are not publicly accessible using the cloud service's file url.

Cloud storage

Be aware that setting up remote storage will not entirely offload file handling tasks from the application server - some Wagtail functionality requires files to be read back by the application server. In particular, original image files need to be read back whenever a new resized rendition is created, and documents may be configured to be served through a Django view in order to enforce permission checks (see [`WAGTAILDOCS_SERVE_METHOD`](#)).

Note that the `django-storages` Amazon S3 backends (`storages.backends.s3boto.S3BotoStorage` and `storages.backends.s3boto3.S3Boto3Storage`) **do not correctly handle duplicate filenames** in their default configuration. When using these backends, `AWS_S3_FILE_OVERWRITE` must be set to `False`.

If you are also serving Wagtail's static files from remote storage (using Django's `STATICFILES_STORAGE` setting), you'll need to ensure that it is configured to serve [`CORS HTTP headers`](#), as current browsers will reject remotely-hosted font files that lack a valid header. For Amazon S3, refer to the documentation [Setting Bucket and Object Access](#)

Permissions, or (for the `storages.backends.s3boto.S3Boto3Storage` backend only) add the following to your Django settings:

```
AWS_S3_OBJECT_PARAMETERS = {  
    "ACL": "public-read"  
}
```

The `ACL` parameter accepts a list of predefined configurations for Amazon S3. For more information, refer to the documentation [Canned ACL](#).

For Google Cloud Storage, create a `cors.json` configuration:

```
[  
  {  
    "origin": ["*"],  
    "responseHeader": ["Content-Type"],  
    "method": ["GET"],  
    "maxAgeSeconds": 3600  
  }  
]
```

Then, apply this CORS configuration to the storage bucket:

```
gsutil cors set cors.json gs://$GS_BUCKET_NAME
```

For other storage services, refer to your provider's documentation, or the documentation for the Django storage backend library you're using.

1.3.6 Performance

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

Editor interface

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

Cache

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

```
CACHES = {  
    'default': {  
        'BACKEND': 'django_redis.cache.RedisCache',  
        'LOCATION': 'redis://127.0.0.1:6379/dbname',  
        # for django-redis < 3.8.0, use:  
        # 'LOCATION': '127.0.0.1:6379',  
        'OPTIONS': {  
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
```

(continues on next page)

(continued from previous page)

```

        }
    }
}
```

Caching image renditions

If you define a cache named ‘renditions’ (typically alongside your ‘default’ cache), Wagtail will cache image rendition lookups, which may improve the performance of pages which include many images.

```
CACHES = {
    'default': {...},
    'renditions': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
        'TIMEOUT': 600,
        'OPTIONS': {
            'MAX_ENTRIES': 1000
        }
    }
}
```

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](#).

For details on configuring Wagtail for Elasticsearch, see [Elasticsearch Backend](#).

Database

Wagtail is tested on PostgreSQL, SQLite and MySQL. It may work on some third-party database backends as well, but this is not guaranteed. We recommend PostgreSQL for production use.

Templates

The overhead from reading and compiling templates adds up. Django wraps its default loaders with [cached template loader](#) which stores the compiled Template in memory and returns it for subsequent requests. The cached loader is automatically enabled when DEBUG is False. If you are using custom loaders, update your settings to use it:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'OPTIONS': {
            'loaders': [
                ('django.template.loaders.cached.Loader', [
                    'django.template.loaders.filesystem.Loader',
                    'django.template.loaders.app_directories.Loader',

```

(continues on next page)

(continued from previous page)

```
        'path.to.custom.Loader',
    ],
},
[]
```

Public users

Caching proxy

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

Wagtail supports automatic cache invalidation for Varnish/Squid. See [Frontend cache invalidator](#) for more information.

Image attributes

For some images, it may be beneficial to lazy load images, so the rest of the page can continue to load. It can be configured site-wide [Adding default attributes to all images](#) or per-image [More control over the img tag](#). For more details you can read about the `loading='lazy'` attribute and the `'decoding='async'` attribute or this [web.dev](#) article on lazy loading images.

This optimisation is already handled for you for images in the admin site.

1.3.7 Internationalisation

- *Multi-language content*
 - *Overview*
 - *Wagtail's approach to multi-lingual content*
 - * *Page structure*
 - * *How locales and translations are recorded in the database*
 - * *Translated homepages*
 - * *Language detection and routing*
 - * *Locales*
 - *Configuration*
 - * *Enabling internationalisation*
 - * *Configuring available languages*
 - * *Enabling the locale management UI (optional)*
 - * *Adding a language prefix to URLs*
 - * *User language auto-detection*
 - * *Custom routing/language detection*

- *Recipes for internationalised sites*
 - * *Language/region selector*
 - * *API filters for headless sites*
 - * *Translatable snippets*
- *Translation workflow*
 - * *Wagtail Localize*
- *Alternative internationalisation plugins*
- *Wagtail admin translations*
- *Change Wagtail admin language on a per-user basis*
- *Changing the primary language of your Wagtail installation*

Multi-language content

Overview

Out of the box, Wagtail assumes all content will be authored in a single language. This document describes how to configure Wagtail for authoring content in multiple languages.

Note: Wagtail provides the infrastructure for creating and serving content in multiple languages. There are two options for managing translations across different languages in the admin interface: [`wagtail.contrib.simple_translation`](#) or the more advanced [`wagtail-localize`](#) (third-party package).

This document only covers the internationalisation of content managed by Wagtail. For information on how to translate static content in template files, JavaScript code, etc, refer to the [Django internationalisation docs](#). Or, if you are building a headless site, refer to the docs of the frontend framework you are using.

Wagtail's approach to multi-lingual content

This section provides an explanation of Wagtail's internationalisation approach. If you're in a hurry, you can skip to [Configuration](#).

In summary:

- Wagtail stores content in a separate page tree for each locale
- It has a built-in `Locale` model and all pages are linked to a `Locale` with the `locale` foreign key field
- It records which pages are translations of each other using a shared UUID stored in the `translation_key` field
- It automatically routes requests through translations of the site's homepage
- It uses Django's `i18n_patterns` and `LocaleMiddleware` for language detection

Page structure

Wagtail stores content in a separate page tree for each locale.

For example, if you have two sites in two locales, then you will see four homepages at the top level of the page hierarchy in the explorer.

This approach has some advantages for the editor experience as well:

- There is no default language for editing, so content can be authored in any language and then translated to any other.
- Translations of a page are separate pages so they can be published at different times.
- Editors can be given permission to edit content in one locale and not others.

How locales and translations are recorded in the database

All pages (and any snippets that have translation enabled) have a `locale` and `translation_key` field:

- `locale` is a foreign key to the `Locale` model
- `translation_key` is a UUID that's used to find translations of a piece of content. Translations of the same page/snippet share the same value in this field

These two fields have a ‘unique together’ constraint so you can’t have more than one translation in the same locale.

Translated homepages

When you set up a site in Wagtail, you select the site’s homepage in the ‘root page’ field and all requests to that site’s root URL will be routed to that page.

Multi-lingual sites have a separate homepage for each locale that exist as siblings in the page tree. Wagtail finds the other homepages by looking for translations of the site’s ‘root page’.

This means that to make a site available in another locale, you just need to translate and publish its homepage in that new locale.

If Wagtail can’t find a homepage that matches the user’s language, it will fall back to the page that is selected as the ‘root page’ on the site record, so you can use this field to specify the default language of your site.

Language detection and routing

For detecting the user’s language and adding a prefix to the URLs (`/en/`, `/fr-fr/`, for example), Wagtail is designed to work with Django’s built-in internationalisation utilities such as `i18n_patterns` and `LocaleMiddleware`. This means that Wagtail should work seamlessly with any other internationalised Django applications on your site.

Locales

The locales that are enabled on a site are recorded in the `Locale` model in `wagtailcore`. This model has just two fields: `ID` and `language_code` which stores the [BCP-47 language tag](#) that represents this locale.

The locale records can be set up with an [optional management UI](#) or created in the shell. The possible values of the `language_code` field are controlled by the `WAGTAIL_CONTENT_LANGUAGES` setting.

Note: Read this if you've changed `LANGUAGE_CODE` before enabling internationalisation

On initial migration, Wagtail creates a `Locale` record for the language that was set in the `LANGUAGE_CODE` setting at the time the migration was run. All pages will be assigned to this `Locale` when Wagtail's internationalisation is disabled.

If you have changed the `LANGUAGE_CODE` setting since updating to Wagtail 2.11, you will need to manually update the record in the `Locale` model too before enabling internationalisation, as your existing content will be assigned to the old code.

Configuration

In this section, we will go through the minimum configuration required to enable content to be authored in multiple languages.

- [Enabling internationalisation](#)
- [Configuring available languages](#)
- [Enabling the locale management UI \(optional\)](#)
- [Adding a language prefix to URLs](#)
- [User language auto-detection](#)
- [Custom routing/language detection](#)

Enabling internationalisation

To enable internationalisation in both Django and Wagtail, set the following settings to `True`:

```
# my_project/settings.py

USE_I18N = True
WAGTAIL_I18N_ENABLED = True
```

In addition, you might also want to enable Django's localisation support. This will make dates and numbers display in the user's local format:

```
# my_project/settings.py

USE_L10N = True
```

Configuring available languages

Next we need to configure the available languages. There are two settings for this that are each used for different purposes:

- `LANGUAGES` - This sets which languages are available on the frontend of the site.
- `WAGTAIL_CONTENT_LANGUAGES` - This sets which the languages Wagtail content can be authored in.

You can set both of these settings to the exact same value. For example, to enable English, French, and Spanish:

```
# my_project/settings.py

WAGTAIL_CONTENT_LANGUAGES = LANGUAGES = [
    ('en', "English"),
    ('fr', "French"),
    ('es', "Spanish"),
]
```

Note: Whenever `WAGTAIL_CONTENT_LANGUAGES` is changed, the `Locale` model needs to be updated as well to match. This can either be done with a data migration or with the optional locale management UI described in the next section.

You can also set these to different values. You might want to do this if you want to have some programmatic localisation (like date formatting or currency, for example) but use the same Wagtail content in multiple regions:

```
# my_project/settings.py

LANGUAGES = [
    ('en-GB', "English (Great Britain)"),
    ('en-US', "English (United States)"),
    ('en-CA', "English (Canada)"),
    ('fr-FR', "French (France)"),
    ('fr-CA', "French (Canada)"),
]

WAGTAIL_CONTENT_LANGUAGES = [
    ('en-GB', "English"),
    ('fr-FR', "French"),
]
```

When configured like this, the site will be available in all the different locales in the first list, but there will only be two language trees in Wagtail.

All the `en-` locales will use the “English” language tree, and the `fr-` locales will use the “French” language tree. The differences between each locale in a language would be programmatic. For example: which date/number format to use, and what currency to display prices in.

Enabling the locale management UI (optional)

An optional locale management app exists to allow a Wagtail administrator to set up the locales from the Wagtail admin interface.

To enable it, add `wagtail.locales` into `INSTALLED_APPS`:

```
# my_project/settings.py

INSTALLED_APPS = [
    # ...
    'wagtail.locales',
    # ...
]
```

Adding a language prefix to URLs

To allow all of the page trees to be served at the same domain, we need to add a URL prefix for each language.

To implement this, we can use Django's built-in `i18n_patterns` function, which adds a language prefix to all of the URL patterns passed into it. This activates the language code specified in the URL and Wagtail takes this into account when it decides how to route the request.

In your project's `urls.py` add Wagtail's core URLs (and any other URLs you want to be translated) into an `i18n_patterns` block:

```
# /my_project/urls.py

# ...

from django.conf.urls.i18n import i18n_patterns

# Non-translatable URLs
# Note: if you are using the Wagtail API or sitemaps,
# these should not be added to `i18n_patterns` either
urlpatterns = [
    path('django-admin/', admin.site.urls),

    path('admin/', include(wagtailadmin_urls)),
    path('documents/', include(wagtailedocs_urls)),
]

# Translatable URLs
# These will be available under a language code prefix. For example /en/search/
urlpatterns += i18n_patterns(
    path('search/', search_views.search, name='search'),
    path("", include(wagtail_urls)),
)
```

Bypass language prefix for the default language

If you want your default language to have URLs that resolve normally without a language prefix, you can set the `prefix_default_language` parameter of `i18n_patterns` to `False`. For example, if you have your languages configured like this:

```
# myproject/settings.py

# ...

LANGUAGE_CODE = 'en'
WAGTAIL_CONTENT_LANGUAGES = LANGUAGES = [
    ('en', "English"),
    ('fr', "French"),
]
# ...
```

And your `urls.py` configured like this:

```
# myproject/urls.py
# ...

# These URLs will be available under a language code prefix only for languages that
# are not set as default in LANGUAGE_CODE.

urlpatterns += i18n_patterns(
    path('search/', search_views.search, name='search'),
    path("", include(wagtail_urls)),
    prefix_default_language=False,
)
```

Your URLs will now be prefixed only for the French version of your website, for example:

```
- /search/
- /fr/search/
```

User language auto-detection

After wrapping your URL patterns with `i18n_patterns`, your site will now respond on URL prefixes. But now it won't respond on the root path.

To fix this, we need to detect the user's browser language and redirect them to the best language prefix. The recommended approach to do this is with Django's `LocaleMiddleware`:

```
# my_project/settings.py

MIDDLEWARE = [
    # ...
    'django.middleware.locale.LocaleMiddleware',
    # ...
]
```

Custom routing/language detection

You don't strictly have to use `i18n_patterns` or `LocaleMiddleware` for this and you can write your own logic if you need to.

All Wagtail needs is the language to be activated (using Django's `django.utils.translation.activate` function) before the `wagtail.views.serve` view is called.

Recipes for internationalised sites

Language/region selector

Perhaps the most important bit of internationalisation-related UI you can add to your site is a selector to allow users to switch between different languages.

If you're not convinced that you need this, have a look at <https://www.w3.org/International/questions/qa-site-conneg#yyshortcomings> for some rationale.

Basic example

Here is a basic example of how to add links between translations of a page.

This example, however, will only include languages defined in `WAGTAIL_CONTENT_LANGUAGES` and not any extra languages that might be defined in `LANGUAGES`. For more information on what both of these settings mean, see [Configuring available languages](#).

If both settings are set to the same value, this example should work well for you, otherwise skip to the next section that has a more complicated example which takes this into account.

```
{# make sure these are at the top of the file #}
{% load i18n wagtailcore_tags %}

{% if page %}
    {% for translation in page.get_translations.live %}
        {% get_language_info for translation.locale.language_code as lang %}
        <a href="{% pageurl translation %}" rel="alternate" hreflang="{{ language_code }}"
        ↪>
            {{ lang.name_local }}
        </a>
        {% endfor %}
    {% endif %}
```

Let's break this down:

```
{% if page %}
...
{% endif %}
```

If this is part of a shared base template it may be used in situations where no page object is available, such as 404 error responses, so check that we have a page before proceeding.

```
{% for translation in page.get_translations.live %}
...
{% endfor %}
```

This `for` block iterates through all published translations of the current page.

```
{% get_language_info for translation.locale.language_code as lang %}
```

This is a Django built-in tag that gets info about the language of the translation. For more information, see `get_language_info()` in the Django docs.

```
<a href="{% pageurl translation %}" rel="alternate" hreflang="{{ lang.language_code }}>
  {{ lang.name_local }}
</a>
```

This adds a link to the translation. We use `{{ lang.name_local }}` to display the name of the locale in its own language. We also add `rel` and `hreflang` attributes to the `<a>` tag for SEO.

Handling locales that share content

Rather than iterating over pages, this example iterates over all of the configured languages and finds the page for each one. This works better than the [Basic example](#) above on sites that have extra Django LANGUAGES that share the same Wagtail content.

For this example to work, you firstly need to add Django's `django.template.context_processors.i18n` context processor to your `TEMPLATES` setting:

```
# myproject/settings.py

TEMPLATES = [
    {
        # ...
        'OPTIONS': {
            'context_processors': [
                # ...
                'django.template.context_processors.i18n',
            ],
        },
    },
]
```

Now for the example itself:

```
{% for language_code, language_name in LANGUAGES %}
  {% get_language_info for language_code as lang %}

  {% language language_code %}
    <a href="{% pageurl page.localized %}" rel="alternate" hreflang="{{ language_
→code }}>
      {{ lang.name_local }}
    </a>
  {% endlanguage %}
{% endfor %}
```

Let's break this down too:

```
{% for language_code, language_name in LANGUAGES %}
...
{% endfor %}
```

This `for` block iterates through all of the configured languages on the site. The `LANGUAGES` variable comes from the `django.template.context_processors.i18n` context processor.

```
{% get_language_info for language_code as lang %}
```

Does exactly the same as the previous example.

```
{% language language_code %}
...
{% endlanguage %}
```

This `language` tag comes from Django's `i18n` tag library. It changes the active language for just the code contained within it.

```
<a href="{% pageurl page.localized %}" rel="alternate" hreflang="{{ language_code }}"
{{ lang.name_local }}>
</a>
```

The only difference with the `<a>` tag here from the `<a>` tag in the previous example is how we're getting the page's URL: `{% pageurl page.localized %}`.

All page instances in Wagtail have a `.localized` attribute which fetches the translation of the page in the current active language. This is why we activated the language previously.

Another difference here is that if the same translated page is shared in two locales, Wagtail will generate the correct URL for the page based on the current active locale. This is the key difference between this example and the previous one as the previous one can only get the URL of the page in its default locale.

API filters for headless sites

For headless sites, the Wagtail API supports two extra filters for internationalised sites:

- `?locale=` Filters pages by the given locale
- `?translation_of=` Filters pages to only include translations of the given page ID

For more information, see [Special filters for internationalized sites](#).

Translatable snippets

You can make a snippet translatable by making it inherit from `wagtail.models.TranslatableMixin`. For example:

```
# myapp/models.py

from django.db import models

from wagtail.models import TranslatableMixin
from wagtail.snippets.models import register_snippet
```

(continues on next page)

(continued from previous page)

```
@register_snippet
class Advert(TranslatableMixin, models.Model):
    name = models.CharField(max_length=255)
```

The TranslatableMixin model adds the `locale` and `translation_key` fields to the model.

Making snippets with existing data translatable

For snippets with existing data, it's not possible to just add `TranslatableMixin`, make a migration, and run it. This is because the `locale` and `translation_key` fields are both required and `translation_key` needs a unique value for each instance.

To migrate the existing data properly, we firstly need to use `BootstrapTranslatableMixin`, which excludes these constraints, then add a data migration to set the two fields, then switch to `TranslatableMixin`.

This is only needed if there are records in the database. So if the model is empty, you can go straight to adding `TranslatableMixin` and skip this.

Step 1: Add `BootstrapTranslatableMixin` to the model

This will add the two fields without any constraints:

```
# myapp/models.py

from django.db import models

from wagtail.models import BootstrapTranslatableMixin
from wagtail.snippets.models import register_snippet


@register_snippet
class Advert(BootstrapTranslatableMixin, models.Model):
    name = models.CharField(max_length=255)

    # if the model has a Meta class, ensure it inherits from
    # BootstrapTranslatableMixin.Meta too
    class Meta(BootstrapTranslatableMixin.Meta):
        verbose_name = 'adverts'
```

Run `python manage.py makemigrations myapp` to generate the schema migration.

Step 2: Create a data migration

Create a data migration with the following command:

```
python manage.py makemigrations myapp --empty
```

This will generate a new empty migration in the app's `migrations` folder. Edit that migration and add a `BootstrapTranslatableModel` for each model to bootstrap in that app:

```

from django.db import migrations
from wagtail.models import BootstrapTranslatableModel

class Migration(migrations.Migration):
    dependencies = [
        ('myapp', '0002_bootstraptranslations'),
    ]

    # Add one operation for each model to bootstrap here
    # Note: Only include models that are in the same app!
    operations = [
        BootstrapTranslatableModel('myapp.Advert'),
    ]

```

Repeat this for any other apps that contain a model to be bootstrapped.

Step 3: Change `BootstrapTranslatableMixin` to `TranslatableMixin`

Now that we have a migration that fills in the required fields, we can swap out `BootstrapTranslatableMixin` for `TranslatableMixin` that has all the constraints:

```

# myapp/models.py

from wagtail.models import TranslatableMixin # Change this line

@register_snippet
class Advert(TranslatableMixin, models.Model): # Change this line
    name = models.CharField(max_length=255)

    class Meta(TranslatableMixin.Meta): # Change this line, if present
        verbose_name = 'adverts'

```

Step 4: Run `makemigrations` to generate schema migrations, then migrate!

Run `makemigrations` to generate the schema migration that adds the constraints into the database, then run `migrate` to run all of the migrations:

```

python manage.py makemigrations myapp
python manage.py migrate

```

When prompted to select a fix for the nullable field ‘`locale`’ being changed to non-nullable, select the option “Ignore for now” (as this has been handled by the data migration).

Translation workflow

As mentioned at the beginning, Wagtail does supply `wagtail.contrib.simple_translation`.

The `simple_translation` module provides a user interface that allows users to copy pages and translatable snippets into another language.

- Copies are created in the source language (not translated)
- Copies of pages are in draft status

Content editors need to translate the content and publish the pages.

To enable add "`wagtail.contrib.simple_translation`" to `INSTALLED_APPS` and run `python manage.py migrate` to create the `submit_translation` permissions. In the Wagtail admin, go to settings and give some users or groups the “Can submit translations” permission.

Note: Simple Translation is optional. It can be switched out by third-party packages. Like the more advanced `wagtail-localize`.

Wagtail Localize

As part of the initial work on implementing internationalisation for Wagtail core, we also created a translation package called `wagtail-localize`. This supports translating pages within Wagtail, using PO files, machine translation, and external integration with translation services.

Github: <https://github.com/wagtail/wagtail-localize>

Alternative internationalisation plugins

Before official multi-language support was added into Wagtail, site implementors had to use external plugins. These have not been replaced by Wagtail’s own implementation as they use slightly different approaches, one of them might fit your use case better:

- `Wagtailtrans`
- `wagtail-modeltranslation`

For a comparison of these options, see AccordBox’s blog post [How to support multi-language in Wagtail CMS](#).

Wagtail admin translations

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

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

Change Wagtail admin language on a per-user basis

Logged-in users can set their preferred language from `/admin/account/`. By default, Wagtail provides a list of languages that have a $\geq 90\%$ translation coverage. It is possible to override this list via the `WAGTAILADMIN_PERMITTED_LANGUAGES` setting.

In case there is zero or one language permitted, the form will be hidden.

If there is no language selected by the user, the `LANGUAGE_CODE` will be used.

Changing the primary language of your Wagtail installation

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

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

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

1.3.8 Private pages

Users with publish permission on a page can set it to be private by clicking the ‘Privacy’ control in the top right corner of the page explorer or editing interface. This sets a restriction on who is allowed to view the page and its sub-pages. Several different kinds of restriction are available:

- **Accessible to logged-in users:** The user must log in to view the page. All user accounts are granted access, regardless of permission level.
- **Accessible with the following password:** The user must enter the given password to view the page. This is appropriate for situations where you want to share a page with a trusted group of people, but giving them individual user accounts would be overkill. The same password is shared between all users, and this works independently of any user accounts that exist on the site.
- **Accessible to users in specific groups:** The user must be logged in, and a member of one or more of the specified groups, in order to view the page.

Similarly, documents can be made private by placing them in a collection with appropriate privacy settings (see: [Image / document permissions](#)).

Private pages and documents work on Wagtail out of the box - the site implementer does not need to do anything to set them up. However, the default “log in” and “password required” forms are only bare-bones HTML pages, and site implementers may wish to replace them with a page customised to their site design.

Setting up a login page

The basic login page can be customised by setting `WAGTAIL_FRONTEND_LOGIN_TEMPLATE` to the path of a template you wish to use:

```
WAGTAIL_FRONTEND_LOGIN_TEMPLATE = 'myapp/login.html'
```

Wagtail uses Django's standard `django.contrib.auth.views.LoginView` view here, and so the context variables available on the template are as detailed in [Django's login view documentation](#).

If the stock Django login view is not suitable - for example, you wish to use an external authentication system, or you are integrating Wagtail into an existing Django site that already has a working login view - you can specify the URL of the login view via the `WAGTAIL_FRONTEND_LOGIN_URL` setting:

```
WAGTAIL_FRONTEND_LOGIN_URL = '/accounts/login/'
```

To integrate Wagtail into a Django site with an existing login mechanism, setting `WAGTAIL_FRONTEND_LOGIN_URL = LOGIN_URL` will usually be sufficient.

Setting up a global “password required” page

By setting `PASSWORD_REQUIRED_TEMPLATE` in your Django settings file, you can specify the path of a template which will be used for all “password required” forms on the site (except for page types that specifically override it - see below):

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

This template will receive the same set of context variables that the blocked page would pass to its own template via `get_context()` - including `page` to refer to the page object itself - plus the following additional variables (which override any of the page’s own context variables of the same name):

- `form` - A Django form object for the password prompt; this will contain a field named `password` as its only visible field. A number of hidden fields may also be present, so the page must loop over `form.hidden_fields` if not using one of Django’s rendering helpers such as `form.as_p`.
- `action_url` - The URL that the password form should be submitted to, as a POST request.

A basic template suitable for use as `PASSWORD_REQUIRED_TEMPLATE` might look like this:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Password required</title>
  </head>
  <body>
    <h1>Password required</h1>
    <p>You need a password to access this page.</p>
    <form action="{{ action_url }}" method="POST">
      {% csrf_token %}

      {{ form.non_field_errors }}

      <div>
        {{ form.password.errors }}
        {{ form.password.label_tag }}
        {{ form.password }}
      </div>

      {% for field in form.hidden_fields %}
        {{ field }}
      {% endfor %}
      <input type="submit" value="Continue" />
    </form>
  </body>
</html>
```

Password restrictions on documents use a separate template, specified through the setting DOCUMENT_PASSWORD_REQUIRED_TEMPLATE; this template also receives the context variables `form` and `action_url` as described above.

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'
```

1.3.9 Customising Wagtail

Customising the editing interface

Customising the tabbed interface

As standard, Wagtail organises panels for pages into three tabs: ‘Content’, ‘Promote’ and ‘Settings’. For snippets Wagtail puts all panels into one page. Depending on the requirements of your site, you may wish to customise this for specific page types or snippets - for example, adding an additional tab for sidebar content. This can be done by specifying an `edit_handler` attribute on the page or snippet model. For example:

```
from wagtail.admin.panels import TabbedInterface, ObjectList

class BlogPage(Page):
    # field definitions omitted

    content_panels = [
        FieldPanel('title', classname="title"),
        FieldPanel('date'),
        FieldPanel('body'),
    ]
    sidebar_content_panels = [
        FieldPanel('advert'),
        InlinePanel('related_links', heading="Related links", label="Related link"),
    ]

    edit_handler = TabbedInterface([
        ObjectList(content_panels, heading='Content'),
        ObjectList(sidebar_content_panels, heading='Sidebar content'),
        ObjectList(Page.promote_panels, heading='Promote'),
        ObjectList(Page.settings_panels, heading='Settings'),
    ])
```

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.fields import RichTextField
from wagtail.admin.panels import FieldPanel

class BookPage(Page):
    body = RichTextField()

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

`RichTextField` inherits from Django's basic `TextField` field, so you can pass any field parameters into `RichTextField` as if using a normal Django field. Its `max_length` will ignore any rich text formatting. This field does not need a special panel and can be defined with `FieldPanel`.

However, template output from `RichTextField` is special and needs to be filtered in order to preserve embedded content. See [Rich text \(filter\)](#).

Limiting features in a rich text field

By default, the rich text editor provides users with a wide variety of options for text formatting and inserting embedded content such as images. However, we may wish to restrict a rich text field to a more limited set of features - for example:

- The field might be intended for a short text snippet, such as a summary to be pulled out on index pages, where embedded images or videos would be inappropriate;
- When page content is defined using `StreamField`, elements such as headings, images and videos are usually given their own block types, alongside a rich text block type used for ordinary paragraph text; in this case, allowing headings and images to also exist within the rich text content is redundant (and liable to result in inconsistent designs).

This can be achieved by passing a `features` keyword argument to `RichTextField`, with a list of identifiers for the features you wish to allow:

```
body = RichTextField(features=['h2', 'h3', 'bold', 'italic', 'link'])
```

The feature identifiers provided on a default Wagtail installation are as follows:

- `h1, h2, h3, h4, h5, h6` - heading elements
- `bold, italic` - bold / italic text
- `ol, ul` - ordered / unordered lists
- `hr` - horizontal rules
- `link` - page, external and email links
- `document-link` - links to documents
- `image` - embedded images
- `embed` - embedded media (see [Embedded content](#))

We have few additional feature identifiers as well. They are not enabled by default, but you can use them in your list of identifiers. These are as follows:

- `code` - inline code
- `superscript`, `subscript`, `strikethrough` - text formatting
- `blockquote` - blockquote

The process for creating new features is described in the following pages:

- *Rich text internals*
- *Extending the Draftail Editor*

You can also provide a setting for naming a group of rich text features. See [WAGTAILADMIN_RICH_TEXT_EDITORS](#).

Image Formats in the Rich Text Editor

On loading, Wagtail will search for any app with the file `image_formats.py` and execute the contents. This provides a way to customise the formatting options shown to the editor when inserting images in the `RichTextField` editor.

As an example, add a “thumbnail” format:

```
# image_formats.py
from wagtail.images.formats import Format, register_image_format

register_image_format(Format('thumbnail', 'Thumbnail', 'richtext-image thumbnail', 'max-'
    ↴120x120'))
```

To begin, import the `Format` class, `register_image_format` function, and optionally `unregister_image_format` function. To register a new `Format`, call the `register_image_format` with the `Format` object as the argument. The `Format` class takes the following constructor arguments:

`name`

The unique key used to identify the format. To unregister this format, call `unregister_image_format` with this string as the only argument.

`label`

The label used in the chooser form when inserting the image into the `RichTextField`.

`classnames`

The string to assign to the `class` attribute of the generated `` tag.

Note: Any class names you provide must have CSS rules matching them written separately, as part of the front end CSS code. Specifying a `classnames` value of `left` will only ensure that class is output in the generated markup, it won’t cause the image to align itself left.

`filter_spec`

The string specification to create the image rendition. For more, see [How to use images in templates](#).

To unregister, call `unregister_image_format` with the string of the name of the `Format` as the only argument.

Warning: Unregistering `Format` objects will cause errors viewing or editing pages that reference them.

Customising generated forms

```
class wagtail.admin.forms.WagtailAdminModelForm
```

```
class wagtail.admin.forms.WagtailAdminPageForm
```

Wagtail automatically generates forms using the panels configured on the model. By default, this form subclasses `WagtailAdminModelForm`, or `WagtailAdminPageForm`, for pages. A custom base form class can be configured by setting the `base_form_class` attribute on any model. Custom forms for snippets must subclass `WagtailAdminModelForm`, and custom forms for pages must subclass `WagtailAdminPageForm`.

This can be used to add non-model fields to the form, to automatically generate field content, or to add custom validation logic for your models:

```
from django import forms
from django.db import models
import geocoder # not in Wagtail, for example only - https://geocoder.readthedocs.io/
from wagtail.admin.panels import FieldPanel
from wagtail.admin.forms import WagtailAdminPageForm
from wagtail.models import Page

class EventPageForm(WagtailAdminPageForm):
    address = forms.CharField()

    def clean(self):
        cleaned_data = super().clean()

        # Make sure that the event starts before it ends
        start_date = cleaned_data['start_date']
        end_date = cleaned_data['end_date']
        if start_date and end_date and start_date > end_date:
            self.add_error('end_date', 'The end date must be after the start date')

        return cleaned_data

    def save(self, commit=True):
        page = super().save(commit=False)

        # Update the duration field from the submitted dates
        page.duration = (page.end_date - page.start_date).days

        # Fetch the location by geocoding the address
        page.location = geocoder.arcgis(self.cleaned_data['address'])

        if commit:
            page.save()
        return page

class EventPage(Page):
    start_date = models.DateField()
    end_date = models.DateField()
    duration = models.IntegerField()
```

(continues on next page)

(continued from previous page)

```

location = models.CharField(max_length=255)

content_panels = [
    FieldPanel('title'),
    FieldPanel('start_date'),
    FieldPanel('end_date'),
    FieldPanel('address'),
]
base_form_class = EventPageForm

```

Wagtail will generate a new subclass of this form for the model, adding any fields defined in `panels` or `content_panels`. Any fields already defined on the model will not be overridden by these automatically added fields, so the form field for a model field can be overridden by adding it to the custom form.

Customising admin templates

In your projects with Wagtail, you may wish to replace elements such as the Wagtail logo within the admin interface with your own branding. This can be done through Django's template inheritance mechanism.

You need to create a `templates/wagtailadmin/` folder within one of your apps - this may be an existing one, or a new one created for this purpose, for example, `dashboard`. This app must be registered in `INSTALLED_APPS` before `wagtail.admin`:

```

INSTALLED_APPS = (
    # ...

    'dashboard',

    'wagtail',
    'wagtail.admin',

    # ...
)

```

Custom branding

The template blocks that are available to customise the branding in the admin interface are as follows:

`branding_logo`

To replace the default logo, create a template file `dashboard/templates/wagtailadmin/base.html` that overrides the block `branding_logo`:

```

{% extends "wagtailadmin/base.html" %}
{% load static %}

{% block branding_logo %}
    
{% endblock %}

```

The logo also appears in the following pages and can be replaced with its template file:

- **login page** - create a template file `dashboard/templates/wagtailadmin/login.html` that overwrites the `branding_logo` block.
- **404 error page** - create a template file `dashboard/templates/wagtailadmin/404.html` that overrides the `branding_logo` block.
- **wagtail userbar** - create a template file `dashboard/templates/wagtailadmin/userbar/base.html` that overwrites the `branding_logo` block.

branding_favicon

To replace the favicon displayed when viewing admin pages, create a template file `dashboard/templates/wagtailadmin/admin_base.html` that overrides the block `branding_favicon`:

```
{% extends "wagtailadmin/admin_base.html" %}  
{% load static %}  
  
{% block branding_favicon %}  
    <link rel="shortcut icon" href="{% static 'images/favicon.ico' %}" />  
{% endblock %}
```

branding_title

To replace the title prefix (which is ‘Wagtail’ by default), create a template file `dashboard/templates/wagtailadmin/admin_base.html` that overrides the block `branding_title`:

```
{% extends "wagtailadmin/admin_base.html" %}  
  
{% block branding_title %}Frank's CMS{% endblock %}
```

branding_login

To replace the login message, create a template file `dashboard/templates/wagtailadmin/login.html` that overrides the block `branding_login`:

```
{% extends "wagtailadmin/login.html" %}  
  
{% block branding_login %}Sign in to Frank's Site{% endblock %}
```

branding_welcome

To replace the welcome message on the dashboard, create a template file `dashboard/templates/wagtailadmin/home.html` that overrides the block `branding_welcome`:

```
{% extends "wagtailadmin/home.html" %}  
  
{% block branding_welcome %}Welcome to Frank's Site{% endblock %}
```

Custom user interface fonts

To customise the font families used in the admin user interface, inject a CSS file using the hook `insert_global_admin_css` and override the variables within the `:root` selector:

```
:root {
    --w-font-sans: Papyrus;
    --w-font-mono: Courier;
}
```

Custom user interface colours

Warning: The default Wagtail colours conform to the WCAG2.1 AA level colour contrast requirements. When customising the admin colours you should test the contrast using tools like [Axe](#).

To customise the colours used in the admin user interface, inject a CSS file using the hook `insert_global_admin_css` and set the desired variables within the `:root` selector. There are two ways to customisation options: either set each colour separately (for example `--w-color-primary: #2E1F5E;`); or separately set `HSL` (`--w-color-primary-hue`, `--w-color-primary-saturation`, `--w-color-primary-lightness`) variables so all shades are customised at once. For example, setting `--w-color-secondary-hue: 180;` will customise all of the secondary shades at once.

Specifying a site or page in the branding

The admin interface has a number of variables available to the renderer context that can be used to customise the branding in the admin page. These can be useful for customising the dashboard on a multitenanted Wagtail installation:

`root_page`

Returns the highest explorable page object for the currently logged in user. If the user has no explore rights, this will default to `None`.

`root_site`

Returns the name on the site record for the above root page.

`site_name`

Returns the value of `root_site`, unless it evaluates to `None`. In that case, it will return the value of `settings.WAGTAIL_SITE_NAME`.

To use these variables, create a template file `dashboard/templates/wagtailadmin/home.html`, just as if you were overriding one of the template blocks in the dashboard, and use them as you would any other Django template variable:

```
{% extends "wagtailadmin/home.html" %}

{% block branding_welcome %}Welcome to the Admin Homepage for {{ root_site }}{% endblock %}
```

Extending the login form

To add extra controls to the login form, create a template file `dashboard/templates/wagtailadmin/login.html`.

above_login and below_login

To add content above or below the login form, override these blocks:

```
{% extends "wagtailadmin/login.html" %}

{% block above_login %} If you are not Frank you should not be here! {% endblock %}
```

fields

To add extra fields to the login form, override the `fields` block. You will need to add `{{ block.super }}` somewhere in your block to include the username and password fields:

```
{% extends "wagtailadmin/login.html" %}

{% block fields %}
    {{ block.super }}
    <li>
        <div>
            <label for="id_two-factor-auth">Two factor auth token</label>
            <input type="text" name="two-factor-auth" id="id_two-factor-auth">
        </div>
    </li>
{% endblock %}
```

submit_buttons

To add extra buttons to the login form, override the `submit_buttons` block. You will need to add `{{ block.super }}` somewhere in your block to include the sign in button:

```
{% extends "wagtailadmin/login.html" %}

{% block submit_buttons %}
    {{ block.super }}
    <a href="{% url 'signup' %}"><button type="button" class="button">{% trans 'Sign up' %}</button></a>
{% endblock %}
```

login_form

To completely customise the login form, override the `login_form` block. This block wraps the whole contents of the `<form>` element:

```
{% extends "wagtailadmin/login.html" %}

{% block login_form %}
    <p>Some extra form content</p>
    {{ block.super }}
{% endblock %}
```

Extending the password reset request form

To add extra controls to the password reset form, create a template file `dashboard/templates/wagtailadmin/account/password_reset/form.html`.

above_form and below_form

To add content above or below the password reset form, override these blocks:

```
{% extends "wagtailadmin/account/password_reset/form.html" %}

{% block above_login %} If you have not received your email within 7 days, call us. {% endblock %}
```

submit_buttons

To add extra buttons to the password reset form, override the `submit_buttons` block. You will need to add `{{ block.super }}` somewhere in your block if you want to include the original submit button:

```
{% extends "wagtailadmin/account/password_reset/form.html" %}

{% block submit_buttons %}
    <a href="{% url 'helpdesk' %}">Contact the helpdesk</a>
    {{ block.super }}
```

Extending client-side components

Some of Wagtail's admin interface is written as client-side JavaScript with [React](#). In order to customise or extend those components, you may need to use React too, as well as other related libraries. To make this easier, Wagtail exposes its React-related dependencies as global variables within the admin. Here are the available packages:

```
// 'focus-trap-react'
window.FocusTrapReact;
// 'react'
window.React;
// 'react-dom'
```

(continues on next page)

(continued from previous page)

```
window.ReactDOM;
// 'react-transition-group/CSSTransitionGroup'
window.CSSTransitionGroup;
```

Wagtail also exposes some of its own React components. You can reuse:

```
window.wagtail.components.Icon;
window.wagtail.components.Portal;
```

Pages containing rich text editors also have access to:

```
// 'draft-js'
window.DraftJS;
// 'draftail'
window.Draftail;

// Wagtail's Draftail-related APIs and components.
window.draftail;
window.draftail.DraftUtils;
window.draftail.ModalWorkflowSource;
window.draftail.ImageModalWorkflowSource;
window.draftail.EmbedModalWorkflowSource;
window.draftail.LinkModalWorkflowSource;
window.draftail.DocumentModalWorkflowSource;
window.draftail.Tooltip;
window.draftail.TooltipEntity;
```

Custom user models

Custom user forms example

This example shows how to add a text field and foreign key field to a custom user model and configure Wagtail user forms to allow the fields values to be updated.

Create a custom user model. This must at minimum inherit from `AbstractBaseUser` and `PermissionsMixin`. In this case we extend the `AbstractUser` class and add two fields. The foreign key references another model (not shown).

```
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    country = models.CharField(verbose_name='country', max_length=255)
    status = models.ForeignKey(MembershipStatus, on_delete=models.SET_NULL, null=True, default=1)
```

Add the app containing your user model to `INSTALLED_APPS` - it must be above the '`wagtail.users`' line, in order to override Wagtail's built-in templates - and set `AUTH_USER_MODEL` to reference your model. In this example the app is called `users` and the model is `User`

```
AUTH_USER_MODEL = 'users.User'
```

Create your custom user 'create' and 'edit' forms in your app:

```

from django import forms
from django.utils.translation import gettext_lazy as _

from wagtail.users.forms import UserEditForm, UserCreationForm

from users.models import MembershipStatus

class CustomUserEditForm(UserEditForm):
    country = forms.CharField(required=True, label=_('Country'))
    status = forms.ModelChoiceField(queryset=MembershipStatus.objects, required=True, ↴
        label=_('Status'))

class CustomUserCreationForm(UserCreationForm):
    country = forms.CharField(required=True, label=_('Country'))
    status = forms.ModelChoiceField(queryset=MembershipStatus.objects, required=True, ↴
        label=_('Status'))

```

Extend the Wagtail user ‘create’ and ‘edit’ templates. These extended templates should be placed in a template directory `wagtailusers/users`.

Template `create.html`:

```

{% extends "wagtailusers/users/create.html" %}

{% block extra_fields %}
    <li>{% include "wagtailadmin/shared/field.html" with field=form.country %}</li>
    <li>{% include "wagtailadmin/shared/field.html" with field=form.status %}</li>
{% endblock extra_fields %}

```

Template `edit.html`:

```

{% extends "wagtailusers/users/edit.html" %}

{% block extra_fields %}
    <li>{% include "wagtailadmin/shared/field.html" with field=form.country %}</li>
    <li>{% include "wagtailadmin/shared/field.html" with field=form.status %}</li>
{% endblock extra_fields %}

```

The `extra_fields` block allows fields to be inserted below the `last_name` field in the default templates. Other block overriding options exist to allow appending fields to the end or beginning of the existing fields, or to allow all the fields to be redefined.

Add the wagtail settings to your project to reference the user form additions:

```

WAGTAIL_USER_EDIT_FORM = 'users.forms.CustomUserEditForm'
WAGTAIL_USER_CREATION_FORM = 'users.forms.CustomUserCreationForm'
WAGTAIL_USER_CUSTOM_FIELDS = ['country', 'status']

```

How to build custom StreamField blocks

Custom editing interfaces for StructBlock

To customise the styling of a StructBlock as it appears in the page editor, you can specify a `form_classname` attribute (either as a keyword argument to the StructBlock constructor, or in a subclass's `Meta`) to override the default value of `struct-block`:

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

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

You can then provide custom CSS for this block, targeted at the specified classname, by using the `insert_editor_css` hook.

Note: Wagtail's editor styling has some built in styling for the `struct-block` class and other related elements. If you specify a value for `form_classname`, it will overwrite the classes that are already applied to StructBlock, so you must remember to specify the `struct-block` as well.

For more extensive customisations that require changes to the HTML markup as well, you can override the `form_template` attribute in `Meta` to specify your own template path. The following variables are available on this template:

`children`

An `OrderedDict` of `BoundBlocks` for all of the child blocks making up this StructBlock.

`help_text`

The help text for this block, if specified.

`classname` The class name passed as `form_classname` (defaults to `struct-block`).

`block_definition` The StructBlock instance that defines this block.

`prefix` The prefix used on form fields for this block instance, guaranteed to be unique across the form.

To add additional variables, you can override the block's `get_form_context` method:

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

    def get_form_context(self, value, prefix='', errors=None):
        context = super().get_form_context(value, prefix=prefix, errors=errors)
        context['suggested_first_names'] = ['John', 'Paul', 'George', 'Ringo']
        return context

    class Meta:
```

(continues on next page)

(continued from previous page)

```
icon = 'user'
form_template = 'myapp/block_forms/person.html'
```

A form template for a StructBlock must include the output of `render_form` for each child block in the `children` dict, inside a container element with a `data-contentpath` attribute equal to the block's name. This attribute is used by the commenting framework to attach comments to the correct fields. The StructBlock's form template is also responsible for rendering labels for each field, but this (and all other HTML markup) can be customised as you see fit. The template below replicates the default StructBlock form rendering:

```
{% load wagtailadmin_tags %}



{% if help_text %}
        <span>
            <div class="help">
                {% icon name="help" class_name="default" %}
                {{ help_text }}
            </div>
        </span>
    {% endif %}

    {% for child in children.values %}
        <div class="w-field" data-field data-contentpath="{{ child.block.name }}">
            {% if child.block.label %}
                <label class="w-field__label" {% if child.id_for_label %}for="{{ child.id_for_label }}"{% endif %}>{{ child.block.label }}{% if child.block.required %}<span class="w-required-mark">*</span>{% endif %}</label>
                {% endif %}
                {{ child.render_form }}
            </div>
        {% endfor %}
    </div>


```

Additional JavaScript on StructBlock forms

Often it may be desirable to attach custom JavaScript behaviour to a StructBlock form. For example, given a block such as:

```
class AddressBlock(StructBlock):
    street = CharBlock()
    town = CharBlock()
    state = CharBlock(required=False)
    country = ChoiceBlock(choices=[
        ('us', 'United States'),
        ('ca', 'Canada'),
        ('mx', 'Mexico'),
    ])
```

we may wish to disable the 'state' field when a country other than United States is selected. Since new blocks can be added dynamically, we need to integrate with StreamField's own front-end logic to ensure that our custom JavaScript code is executed when a new block is initialised.

StreamField uses the `telepath` library to map Python block classes such as `StructBlock` to a corresponding JavaScript implementation. These JavaScript implementations can be accessed through the `window.wagtailStreamField.blocks` namespace, as the following classes:

- `FieldBlockDefinition`
- `ListBlockDefinition`
- `StaticBlockDefinition`
- `StreamBlockDefinition`
- `StructBlockDefinition`

First, we define a telepath adapter for `AddressBlock`, so that it uses our own JavaScript class in place of the default `StructBlockDefinition`. This can be done in the same module as the `AddressBlock` definition:

```
from wagtail.blocks.struct_block import StructBlockAdapter
from wagtail.telepath import register
from django import forms
from django.utils.functional import cached_property

class AddressBlockAdapter(StructBlockAdapter):
    js_constructor = 'myapp.blocks.AddressBlock'

    @cached_property
    def media(self):
        structblock_media = super().media
        return forms.Media(
            js=structblock_media._js + ['js/address-block.js'],
            css=structblock_media._css
        )

register(AddressBlockAdapter(), AddressBlock)
```

Here `'myapp.blocks.AddressBlock'` is the identifier for our JavaScript class that will be registered with the telepath client-side code, and `'js/address-block.js'` is the file that defines it (as a path within any static file location recognised by Django). This implementation subclasses `StructBlockDefinition` and adds our custom code to the `render` method:

```
class AddressBlockDefinition extends window.wagtailStreamField.blocks
    .StructBlockDefinition {
    render(placeholder, prefix, initialState, initialError) {
        const block = super.render(
            placeholder,
            prefix,
            initialState,
            initialError,
        );

        const stateField = document.getElementById(prefix + '-state');
        const countryField = document.getElementById(prefix + '-country');
        const updateStateInput = () => {
            if (countryField.value == 'us') {
                stateField.removeAttribute('disabled');
            } else {
                stateField.setAttribute('disabled', true);
            }
        };
    }
}
```

(continues on next page)

(continued from previous page)

```

        }
    };
    updateStateInput();
    countryField.addEventListener('change', updateStateInput);

    return block;
}
}

window.telepath.register('myapp.blocks.AddressBlock', AddressBlockDefinition);

```

Additional methods and properties on StructBlock values

When rendering StreamField content on a template, StructBlock values are represented as dict-like objects where the keys correspond to the names of the child blocks. Specifically, these values are instances of the class `wagtail.blocks.StructValue`.

Sometimes, it's desirable to make additional methods or properties available on this object. For example, given a StructBlock that represents either an internal or external link:

```

class LinkBlock(StructBlock):
    text = CharBlock(label="link text", required=True)
    page = PageChooserBlock(label="page", required=False)
    external_url = URLBlock(label="external URL", required=False)

```

you may want to make a `url` property available, that returns either the page URL or external URL depending which one was filled in. A common mistake is to define this property on the block class itself:

```

class LinkBlock(StructBlock):
    text = CharBlock(label="link text", required=True)
    page = PageChooserBlock(label="page", required=False)
    external_url = URLBlock(label="external URL", required=False)

    @property
    def url(self): # INCORRECT - will not work
        return self.external_url or self.page.url

```

This does not work because the value as seen in the template is not an instance of `LinkBlock`. StructBlock instances only serve as specifications for the block's behaviour, and do not hold block data in their internal state - in this respect, they are similar to Django's form widget objects (which provide methods for rendering a given value as a form field, but do not hold on to the value itself).

Instead, you should define a subclass of `StructValue` that implements your custom property or method. Within this method, the block's data can be accessed as `self['page']` or `self.get('page')`, since `StructValue` is a dict-like object.

```

from wagtail.blocks import StructValue

class LinkStructValue(StructValue):
    def url(self):
        external_url = self.get('external_url')

```

(continues on next page)

(continued from previous page)

```
page = self.get('page')
return external_url or page.url
```

Once this is defined, set the block's `value_class` option to instruct it to use this class rather than a plain `StructValue`:

```
class LinkBlock(StructBlock):
    text = CharBlock(label="link text", required=True)
    page = PageChooserBlock(label="page", required=False)
    external_url = URLBlock(label="external URL", required=False)

    class Meta:
        value_class = LinkStructValue
```

Your extended value class methods will now be available in your template:

```
{% for block in page.body %}
  {% if block.block_type == 'link' %}
    <a href="{{ link.value.url }}>{{ link.value.text }}</a>
  {% endif %}
{% endfor %}
```

Custom block types

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

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

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

Since the StreamField editing interface needs to create blocks dynamically, certain complex widget types will need additional JavaScript code to define how to render and populate them on the client-side. If a field uses a widget type that does not inherit from one of the classes inheriting from `django.forms.widgets.Input`, `django.forms.Textarea`, `django.forms.Select` or `django.forms.RadioSelect`, or has customised client-side behaviour to the extent where it is not possible to read or write its data simply by accessing the form element's `value` property, you will need to provide a JavaScript handler object, implementing the methods detailed on [Form widget client-side API](#).

Handling block definitions within migrations

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

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

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

1.3.10 Third-party tutorials

Warning: The following list is a collection of tutorials and development notes from third-party developers. Some of the older links may not apply to the latest Wagtail versions.

- [Hosting a Wagtail site on Digital Ocean with CapRover](#) (21 July 2022)
- [Add Heading Blocks with Bookmarks in Wagtail](#) (5 July 2022)
- [Import files into Wagtail](#) (2 July 2022)
- [Adding MapBox Blocks to Wagtail Stream Fields](#) (19 June 2022)
- [5 Tips to Streamline Your Wagtail CMS Development](#) (14 June 2022)
- [Wagtail 3 Upgrade: Per Site Features](#) (2 June 2022)
- [Wagtail 3 Upgrade: Per User FieldPanel Permissions](#) (1 June 2022)
- [Upgrading to Wagtail 3.0](#) (3 May 2022)
- [Django for E-Commerce: A Developers Guide \(with Wagtail CMS Tutorial\) - Updated](#) (21 March 2022)
- [How to install Wagtail on Ubuntu 20.04|22.04](#) (1 March 2022)
- [Building a blog with Wagtail \(tutorial part 1 of 2\)](#) (27 February 2022); [part 2 of 2](#) (6 March 2022)
- [Creating a schematic editor within Wagtail CMS with StimulusJS](#) (20 February 2022)
- [Adding Placeholder Text to Wagtail Forms](#) (11 February 2022)
- [Deploying a Wagtail 2.16 website to Heroku](#) (9 February 2022)
- [Build an E-Commerce Site with Wagtail CMS, Bootstrap & Django Framework](#) (7 February 2022)
- [Complex Custom Field Pagination in Django \(Wagtail\)](#) (3 February 2022)
- [How to Connect Wagtail and React](#) (31 January 2022)
- [Wagtail: Dynamically Adding Admin Menu Items](#) (25 January 2022)
- [Headless Wagtail, what are the pain points? \(with solutions\)](#) (24 January 2022)

- A collection of UIKit components that can be used as a Wagtail StreamField block (14 January 2022)
- Introduction to Wagtail CMS (1 January 2022)
- How to make Wagtail project have good coding style (18 December 2021)
- Wagtail: The Django newcomer - German (13 December 2021)
- Create a Developer Portfolio with Wagtail Part 10: Dynamic Site Settings (3 December 2021)
- Dockerize Wagtail CMS for your development environment (29 November 2021)
- How To Add an Email Newsletter to Wagtail (25 November 2021)
- Dealing with UNIQUE Fields on a Multi-lingual Site (6 November 2021)
- General Wagtail Tips & Ticks (26 October 2021)
- Branching workflows in Wagtail (12 October 2021)
- Wagtail is the best python CMS in our galaxy - Russian (12 October 2021)
- Adding Tasks with a Checklist to Wagtail Workflows (22 September 2021)
- How to create a Zen (Focused) mode for the Wagtail CMS admin (5 September 2021)
- How to Install Wagtail on Shared Hosting without Root (CPPanel) (26 August 2021)
- Django for E-Commerce: A Developers Guide (with Wagtail CMS Tutorial) (26 August 2021)
- How to create a Kanban (Trello style) view of your ModelAdmin data in Wagtail (20 August 2021)
- eBook: The Definitive Guide to Next.js and Wagtail (19 August 2021)
- How to build an interactive guide for users in the Wagtail CMS admin (19 August 2021)
- Add Custom User Model (with custom fields like phone no, profile picture) to django or wagtail sites (16 August 2021)
- File size limits in Nginx and animated GIF support (14 August 2021)
- Deploying Wagtail site on Digital Ocean (11 August 2021)
- Multi-language Wagtail websites with XLIFF (21 June 2021)
- Add & Configure Mail in Django (or Wagtail) using Sendgrid (28 May 2021)
- Advanced Django Development: How to build a professional CMS for any business? (3 part tutorial) (2 April 2021)
- Matomo Analytics with WagtailCMS (31 March 2021)
- Dockerizing a Wagtail App (16 March 2021)
- Deploying Wagtail on CentOS8 with MariaDB/Nginx/Gunicorn (7 March 2021)
- How to add a List of Related Fields to a Page (6 March 2021)
- Wagtail - get_absolute_url, without domain (3 March 2021)
- How To Alternate Blocks in Your Django & Wagtail Templates (19 February 2021)
- Build a Blog With Wagtail CMS (second version) (13 January 2021)
- Migrate your Wagtail Website from wagtailtrans to the new wagtail-localize (10 January 2021)
- How to Use the Wagtail CMS for Django: An Overview (21 December 2020)
- Wagtail modeladmin and a dynamic panels list (14 December 2020)
- Install and Deploy Wagtail CMS on pythonanywhere.com (14 December 2020)

- Overriding the admin CSS in Wagtail (4 December 2020)
- Migrating your Wagtail site to a different database engine (3 December 2020)
- Wagtail for Django Devs: Create a Developer Portfolio (30 November 2020)
- Create a Developer Portfolio with Wagtail Tutorial Series (11 November 2020)
- Wagtail Instagram New oEmbed API (5 November 2020)
- Image upload in wagtail forms (21 October 2020)
- Adding a timeline of your wagtail Posts (18 September 2020)
- How to create amazing SSR website with Wagtail 2 + Vue 3 (1 September 2020)
- Migrate Wagtail Application Database from SQLite to PostgreSQL (5 June 2020)
- How to Build Scalable Websites with Wagtail and Nuxt (14 May 2020)
- Wagtail multi-language and internationalization (8 April 2020)
- Wagtail SEO Guide (30 March 2020)
- Adding a latest-changes list to your Wagtail site (27 March 2020)
- How to support multi-language in Wagtail CMS (22 February 2020)
- Deploying my Wagtail blog to Digital Ocean Part 1 of a 2 part series (29 January 2020)
- How to Create and Manage Menus of Wagtail application - An updated overview of implementing menus (22 February 2020)
- Adding a React component in Wagtail Admin - Shows how to render an interactive timeline of published Pages (30 December 2019)
- Wagtail API - how to customise the detail URL (19 December 2020)
- How to Add Django Models to the Wagtail Admin (27 August 2019)
- How do I Wagtail - An Editor's Guide for Mozilla's usage of Wagtail (25 April 2019)
- Learn Wagtail - Regular video tutorials about all aspects of Wagtail (1 March 2019)
- How to add buttons to ModelAdmin Index View in Wagtail CMS (23 January 2019)
- Wagtail Tutorial Series (20 January 2019)
- How to Deploy Wagtail to Google App Engine PaaS (Video) (18 December 2018)
- How To Prevent Users From Creating Pages by Page Type (25 October 2018)
- How to Deploy Wagtail to Jelastic PaaS (11 October 2018)
- Basic Introduction to Setting Up Wagtail (15 August 2018)
- E-Commerce for Django developers (with Wagtail shop tutorial) (5 July 2018)
- Supporting StreamFields, Snippets and Images in a Wagtail GraphQL API (14 June 2018)
- Wagtail and GraphQL (19 April 2018)
- Wagtail and Azure storage blob containers (29 November 2017)
- Building TwilioQuest with Twilio Sync, Django [incl. Wagtail], and Vue.js (6 November 2017)
- Upgrading from Wagtail 1.0 to Wagtail 1.11 (19 July 2017)
- Wagtail-Multilingual: a simple project to demonstrate how multilingual is implemented (31 January 2017)
- Wagtail: 2 Steps for Adding Pages Outside of the CMS (15 February 2016)

- Adding a Twitter Widget for Wagtail’s new StreamField (2 April 2015)
- Working With Wagtail: Menus (22 January 2015)
- Upgrading Wagtail to use Django 1.7 locally using vagrant (10 December 2014)
- Wagtail redirect page. Can link to page, URL and document (24 September 2014)
- Outputting JSON for a model with properties and db fields in Wagtail/Django (24 September 2014)
- Bi-lingual website using Wagtail CMS (17 September 2014)
- Wagtail CMS – Lesser known features (12 September 2014)
- Wagtail notes: stateful on/off hallo.js plugins (9 August 2014)
- Add some blockquote buttons to Wagtail CMS’ WYSIWYG Editor (24 July 2014)
- Adding Bread Crumbs to the front end in Wagtail CMS (1 July 2014)
- Extending hallo.js using Wagtail hooks (9 July 2014)
- Wagtail notes: custom tabs per page type (10 May 2014)
- Wagtail notes: managing redirects as pages (10 May 2014)
- Wagtail notes: dynamic templates per page (10 May 2014)
- Wagtail notes: type-constrained PageChooserPanel (9 May 2014)

You can also find more resources from the community on [Awesome Wagtail](#).

Tip

We are working on a collection of Wagtail tutorials and best practices. Please tweet @WagtailCMS or contact us directly to share your Wagtail HOWTOs, development notes or site launches.

1.3.11 Testing your Wagtail site

Wagtail comes with some utilities that simplify writing tests for your site.

WagtailPageTests

`class wagtail.test.utils.WagtailPageTests` `WagtailPageTests` extends `django.test.TestCase`, adding a few new assert methods. You should extend this class to make use of its methods:

```
from wagtail.test.utils import WagtailPageTests
from myapp.models import MyPage

class MyPageTests(WagtailPageTests):
    def test_can_create_a_page(self):
        ...
```

`assertCanCreateAt(parent_model, child_model, msg=None)` Assert a particular child Page type can be created under a parent Page type. `parent_model` and `child_model` should be the Page classes being tested.

```
def test_can_create_under_home_page(self):
    # You can create a ContentPage under a HomePage
    self.assertCanCreateAt(HomePage, ContentPage)
```

assertCannotCreateAt(*parent_model*, *child_model*, *msg=None*) Assert a particular child Page type can not be created under a parent Page type. *parent_model* and *child_model* should be the Page classes being tested.

```
def test_cant_create_under_event_page(self):
    # You can not create a ContentPage under an EventPage
    self.assertCannotCreateAt(EventPage, ContentPage)
```

assertCanCreate(*parent*, *child_model*, *data*, *msg=None*) Assert that a child of the given Page type can be created under the parent, using the supplied POST data.

parent should be a Page instance, and *child_model* should be a Page subclass. *data* should be a dict that will be POSTed at the Wagtail admin Page creation method.

```
from wagtail.test.utils.form_data import nested_form_data, streamfield

def test_can_create_content_page(self):
    # Get the HomePage
    root_page = HomePage.objects.get(pk=2)

    # Assert that a ContentPage can be made here, with this POST data
    self.assertCanCreate(root_page, ContentPage, nested_form_data({
        'title': 'About us',
        'body': streamfield([
            ('text', 'Lorem ipsum dolor sit amet'),
        ])
    }))
```

See [Form data helpers](#) for a set of functions useful for constructing POST data.

assertAllowedParentPageTypes(*child_model*, *parent_models*, *msg=None*) Test that the only page types that *child_model* can be created under are *parent_models*.

The list of allowed parent models may differ from those set in Page.parent_page_types, if the parent models have set Page.subpage_types.

```
def test_content_page_parent_pages(self):
    # A ContentPage can only be created under a HomePage
    # or another ContentPage
    self.assertAllowedParentPageTypes(
        ContentPage, {HomePage, ContentPage})

    # An EventPage can only be created under an EventIndex
    self.assertAllowedParentPageTypes(
        EventPage, {EventIndex})
```

assertAllowedSubpageTypes(*parent_model*, *child_models*, *msg=None*) Test that the only page types that can be created under *parent_model* are *child_models*.

The list of allowed child models may differ from those set in Page.subpage_types, if the child models have set Page.parent_page_types.

```
def test_content_page_subpages(self):
    # A ContentPage can only have other ContentPage children
    self.assertAllowedSubpageTypes(
        ContentPage, {ContentPage})
```

(continues on next page)

(continued from previous page)

```
# A HomePage can have ContentPage and EventIndex children
self.assertAllowedSubpageTypes(
    HomePage, {ContentPage, EventIndex})
```

Form data helpers

The `assertCanCreate` method requires page data to be passed in the same format that the page edit form would submit. For complex page types, it can be difficult to construct this data structure by hand; the `wagtail.test.utils.form_data` module provides a set of helper functions to assist with this.

`wagtail.test.utils.form_data.nested_form_data(data)`

Translates a nested dict structure into a flat form data dict with hyphen-separated keys.

```
nested_form_data({
    'foo': 'bar',
    'parent': {
        'child': 'field',
    },
})
# Returns: {'foo': 'bar', 'parent-child': 'field'}
```

`wagtail.test.utils.form_data.rich_text(value, editor='default', features=None)`

Converts an HTML-like rich text string to the data format required by the currently active rich text editor.

Parameters

- **editor** – An alternative editor name as defined in `WAGTAILADMIN_RICH_TEXT_EDITORS`
- **features** – A list of features allowed in the rich text content (see [Limiting features in a rich text field](#))

```
self.assertCanCreate(root_page, ContentPage, nested_form_data({
    'title': 'About us',
    'body': rich_text('<p>Lorem ipsum dolor sit amet</p>'),
}))
```

`wagtail.test.utils.form_data.streamfield(items)`

Takes a list of (block_type, value) tuples and turns it in to StreamField form data. Use this within a `nested_form_data()` call, with the field name as the key.

```
nested_form_data({'content': streamfield([
    ('text', 'Hello, world'),
])})
# Returns:
# {
#     'content-count': '1',
#     'content-0-type': 'text',
#     'content-0-value': 'Hello, world',
#     'content-0-order': '0',
#     'content-0-deleted': '',
# }
```

```
wagtail.test.utils.form_data.inline_formset(items, initial=0, min=0, max=1000)
```

Takes a list of form data for an InlineFormset and translates it in to valid POST data. Use this within a `nested_form_data()` call, with the formset relation name as the key.

```
nested_form_data({'lines': inline_formset([
    {'text': 'Hello'},
    {'text': 'World'},
])})
# Returns:
# {
#     'lines-TOTAL_FORMS': '2',
#     'lines-INITIAL_FORMS': '0',
#     'lines-MIN_NUM_FORMS': '0',
#     'lines-MAX_NUM_FORMS': '1000',
#     'lines-0-text': 'Hello',
#     'lines-0-ORDER': '0',
#     'lines-0-DELETE': '',
#     'lines-1-text': 'World',
#     'lines-1-ORDER': '1',
#     'lines-1-DELETE': '',
# }
```

Fixtures

Using dumpdata

Creating `fixtures` for tests is best done by creating content in a development environment, and using Django's `dumpdata` command.

Note that by default `dumpdata` will represent `content_type` by the primary key; this may cause consistency issues when adding / removing models, as content types are populated separately from fixtures. To prevent this, use the `--natural-foreign` switch, which represents content types by `["app", "model"]` instead.

Manual modification

You could modify the dumped fixtures manually, or even write them all by hand. Here are a few things to be wary of.

Custom Page models

When creating customised Page models in fixtures, you will need to add both a `wagtailcore.page` entry, and one for your custom Page model.

Let's say you have a `website` module which defines a `Homepage(Page)` class. You could create such a homepage in a fixture with:

```
[{
    "model": "wagtailcore.page",
    "pk": 3,
    "fields": {
        "title": "My Customer's Homepage",
```

(continues on next page)

(continued from previous page)

```
        "content_type": ["website", "homepage"],
        "depth": 2
    },
    {
        "model": "website.homepage",
        "pk": 3,
        "fields": {}
    }
]
```

Treebeard fields

Filling in the `path` / `numchild` / `depth` fields is necessary in order for tree operations like `get_parent()` to work correctly. `url_path` is another field that can cause errors in some uncommon cases if it isn't filled in.

The [Treebeard docs](#) might help in understanding how this works.

1.3.12 Wagtail API

The API module provides a public-facing, JSON-formatted API to allow retrieving content as raw field data. This is useful for cases like serving content to non-web clients (such as a mobile phone app) or pulling content out of Wagtail for use in another site.

See [RFC 8: Wagtail API](#) for full details on our stabilisation policy.

Wagtail API v2 Configuration Guide

This section of the docs will show you how to set up a public API for your Wagtail site.

Even though the API is built on Django REST Framework, you do not need to install this manually as it is already a dependency of Wagtail.

Basic configuration

Enable the app

Firstly, you need to enable Wagtail's API app so Django can see it. Add `wagtail.api.v2` to `INSTALLED_APPS` in your Django project settings:

```
# settings.py

INSTALLED_APPS = [
    ...
    'wagtail.api.v2',
    ...
]
```

Optionally, you may also want to add `rest_framework` to `INSTALLED_APPS`. This would make the API browsable when viewed from a web browser but is not required for basic JSON-formatted output.

Configure endpoints

Next, it's time to configure which content will be exposed on the API. Each content type (such as pages, images and documents) has its own endpoint. Endpoints are combined by a router, which provides the url configuration you can hook into the rest of your project.

Wagtail provides three endpoint classes you can use:

- Pages `wagtail.api.v2.views.PagesAPIViewSet`
- Images `wagtail.images.api.v2.views.ImagesAPIViewSet`
- Documents `wagtail.documents.api.v2.views.DocumentsAPIViewSet`

You can subclass any of these endpoint classes to customize their functionality. Additionally, there is a base endpoint class you can use for adding different content types to the API: `wagtail.api.v2.views.BaseAPIViewSet`

For this example, we will create an API that includes all three builtin content types in their default configuration:

```
# api.py

from wagtail.api.v2.views import PagesAPIViewSet
from wagtail.api.v2.router import WagtailAPIRouter
from wagtail.images.api.v2.views import ImagesAPIViewSet
from wagtail.documents.api.v2.views import DocumentsAPIViewSet

# Create the router. "wagtailapi" is the URL namespace
api_router = WagtailAPIRouter('wagtailapi')

# Add the three endpoints using the "register_endpoint" method.
# The first parameter is the name of the endpoint (such as pages, images). This
# is used in the URL of the endpoint
# The second parameter is the endpoint class that handles the requests
api_router.register_endpoint('pages', PagesAPIViewSet)
api_router.register_endpoint('images', ImagesAPIViewSet)
api_router.register_endpoint('documents', DocumentsAPIViewSet)
```

Next, register the URLs so Django can route requests into the API:

```
# urls.py

from .api import api_router

urlpatterns = [
    ...
    path('api/v2/', api_router.urls),
    ...
    # Ensure that the api_router line appears above the default Wagtail page serving
    # route
```

(continues on next page)

(continued from previous page)

```
    re_path(r'^', include(wagtail_urls)),  
]
```

With this configuration, pages will be available at `/api/v2/pages/`, images at `/api/v2/images/` and documents at `/api/v2/documents/`

Adding custom page fields

It's likely that you would need to export some custom fields over the API. This can be done by adding a list of fields to be exported into the `api_fields` attribute for each page model.

For example:

```
# blog/models.py  
  
from wagtail.api import APIField  
  
class BlogPageAuthor(Orderable):  
    page = models.ForeignKey('blog.BlogPage', on_delete=models.CASCADE, related_name='authors')  
    name = models.CharField(max_length=255)  
  
    api_fields = [  
        APIField('name'),  
    ]  
  
class BlogPage(Page):  
    published_date = models.DateTimeField()  
    body = RichTextField()  
    feed_image = models.ForeignKey('wagtailimages.Image', on_delete=models.SET_NULL, null=True, ...)  
    private_field = models.CharField(max_length=255)  
  
    # export fields over the api  
    api_fields = [  
        APIField('published_date'),  
        APIField('body'),  
        APIField('feed_image'),  
        APIField('authors'), # this will nest the relevant BlogPageAuthor objects in the api response  
    ]
```

This will make `published_date`, `body`, `feed_image` and a list of `authors` with the `name` field available in the API. But to access these fields, you must select the `blog.BlogPage` type using the `?type parameter in the API itself`.

Custom serializers

Serializers are used to convert the database representation of a model into JSON format. You can override the serializer for any field using the `serializer` keyword argument:

```
from rest_framework.fields import DateField

class BlogPage(Page):
    ...

    api_fields = [
        # Change the format of the published_date field to "Thursday 06 April 2017"
        APIField('published_date', serializer=DateField(format='%A %d %B %Y')),
        ...
    ]
```

Django REST framework's serializers can all take a `source` argument allowing you to add API fields that have a different field name or no underlying field at all:

```
from rest_framework.fields import DateField

class BlogPage(Page):
    ...

    api_fields = [
        # Date in ISO8601 format (the default)
        APIField('published_date'),

        # A separate published_date_display field with a different format
        APIField('published_date_display', serializer=DateField(format='%A %d %B %Y', ↴
        source='published_date')),

        ...
    ]
```

This adds two fields to the API (other fields omitted for brevity):

```
{
    "published_date": "2017-04-06",
    "published_date_display": "Thursday 06 April 2017"
}
```

Images in the API

The `ImageRenditionField` serializer allows you to add renditions of images into your API. It requires an image filter string specifying the resize operations to perform on the image. It can also take the `source` keyword argument described above.

For example:

```
from wagtail.api import APIField
from wagtail.images.api.fields import ImageRenditionField

class BlogPage(Page):
```

(continues on next page)

(continued from previous page)

```
...  
  
api_fields = [  
    # Adds information about the source image (eg, title) into the API  
    APIMField('feed_image'),  
  
    # Adds a URL to a rendered thumbnail of the image to the API  
    APIMField('feed_image_thumbnail', serializer=ImageRenditionField('fill-100x100',  
        source='feed_image')),  
    ...  
]
```

This would add the following to the JSON:

```
{  
    "feed_image": {  
        "id": 45529,  
        "meta": {  
            "type": "wagtailimages.Image",  
            "detail_url": "http://www.example.com/api/v2/images/12/",  
            "download_url": "/media/images/a_test_image.jpg",  
            "tags": []  
        },  
        "title": "A test image",  
        "width": 2000,  
        "height": 1125  
    },  
    "feed_image_thumbnail": {  
        "url": "/media/images/a_test_image.fill-100x100.jpg",  
        "full_url": "http://www.example.com/media/images/a_test_image.fill-100x100.jpg",  
        "width": 100,  
        "height": 100,  
        "alt": "image alt text"  
    }  
}
```

Note: `download_url` is the original uploaded file path, whereas `feed_image_thumbnail['url']` is the url of the rendered image. When you are using another storage backend, such as S3, `download_url` will return a URL to the image if your media files are properly configured.

Additional settings

`WAGTAILAPI_BASE_URL`

(required when using frontend cache invalidation)

This is used in two places, when generating absolute URLs to document files and invalidating the cache.

Generating URLs to documents will fall back to the current request's hostname if this is not set. Cache invalidation cannot do this, however, so this setting must be set when using this module alongside the `wagtailfrontendcache` module.

WAGTAILAPI_SEARCH_ENABLED

(default: True)

Setting this to false will disable full text search. This applies to all endpoints.

WAGTAILAPI_LIMIT_MAX

(default: 20)

This allows you to change the maximum number of results a user can request at a time. This applies to all endpoints. Set to None for no limit.

Wagtail API v2 Usage Guide

The Wagtail API module exposes a public, read only, JSON-formatted API which can be used by external clients (such as a mobile app) or the site's frontend.

This document is intended for developers using the API exposed by Wagtail. For documentation on how to enable the API module in your Wagtail site, see [Wagtail API v2 Configuration Guide](#)

Contents

- *Fetching content*
 - *Example response*
 - *Custom page fields in the API*
 - *Pagination*
 - *Ordering*
 - * *Random ordering*
 - *Filtering*
 - *Filtering by tree position (pages only)*
 - *Filtering pages by site*
 - *Search*
 - * *Search operator*
 - *Special filters for internationalized sites*
 - * *Filtering pages by locale*
 - * *Getting translations of a page*
 - *Fields*
 - * *Additional fields*
 - * *All fields*
 - * *Removing fields*
 - * *Removing all default fields*
 - *Detail views*

- [Finding pages by HTML path](#)
- [Default endpoint fields](#)
 - [Common fields](#)
 - [Pages](#)
 - [Images](#)
 - [Documents](#)
- [Changes since v1](#)
 - [Breaking changes](#)
 - [Major features](#)
 - [Minor features](#)

Fetching content

To fetch content over the API, perform a GET request against one of the following endpoints:

- Pages </api/v2/pages/>
- Images </api/v2/images/>
- Documents </api/v2/documents/>

Note: The available endpoints and their URLs may vary from site to site, depending on how the API has been configured.

Example response

Each response contains the list of items (`items`) and the total count (`meta.total_count`). The total count is irrespective of pagination.

```
GET /api/v2/endpoint_name/  
  
HTTP 200 OK  
Content-Type: application/json  
  
{  
    "meta": {  
        "total_count": "total number of results"  
    },  
    "items": [  
        {  
            "id": 1,  
            "meta": {  
                "type": "app_name.ModelName",  
                "detail_url": "http://api.example.com/api/v2/endpoint_name/1/"  
            },  
            "field": "value"  
        }  
    ]  
}
```

(continues on next page)

(continued from previous page)

```

},
{
    "id": 2,
    "meta": {
        "type": "app_name.ModelName",
        "detail_url": "http://api.example.com/api/v2/endpoint_name/2/"
    },
    "field": "different value"
}
]
}

```

Custom page fields in the API

Wagtail sites contain many page types, each with their own set of fields. The `pages` endpoint will only expose the common fields by default (such as `title` and `slug`).

To access custom page fields with the API, select the page type with the `?type` parameter. This will filter the results to only include pages of that type but will also make all the exported custom fields for that type available in the API.

For example, to access the `published_date`, `body` and `authors` fields on the `blog.BlogPage` model in the [configuration docs](#):

```
GET /api/v2/pages/?type=blog.BlogPage&fields=published_date,body,authors(name)
```

HTTP 200 OK

Content-Type: application/json

```
{
    "meta": {
        "total_count": 10
    },
    "items": [
        {
            "id": 1,
            "meta": {
                "type": "blog.BlogPage",
                "detail_url": "http://api.example.com/api/v2/pages/1/",
                "html_url": "http://www.example.com/blog/my-blog-post/",
                "slug": "my-blog-post",
                "first_published_at": "2016-08-30T16:52:00Z"
            },
            "title": "Test blog post",
            "published_date": "2016-08-30",
            "authors": [
                {
                    "id": 1,
                    "meta": {
                        "type": "blog.BlogPageAuthor",
                    },
                    "name": "Karl Hobley"
                }
            ]
        }
    ]
}
```

(continues on next page)

(continued from previous page)

```
        ],
    },
    ...
]
```

Note: Only fields that have been explicitly exported by the developer may be used in the API. This is done by adding a `api_fields` attribute to the page model. You can read about configuration [here](#).

This doesn't apply to images/documents as there is only one model exposed in those endpoints. But for projects that have customized image/document models, the `api_fields` attribute can be used to export any custom fields into the API.

Pagination

The number of items in the response can be changed by using the `?limit` parameter (default: 20) and the number of items to skip can be changed by using the `?offset` parameter.

For example:

```
GET /api/v2/pages/?offset=20&limit=20

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 50
    },
    "items": [
        pages 20 - 40 will be listed here.
    ]
}
```

Note: There may be a maximum value for the `?limit` parameter. This can be modified in your project settings by setting `WAGTAILAPI_LIMIT_MAX` to either a number (the new maximum value) or `None` (which disables maximum value check).

Ordering

The results can be ordered by any field by setting the `?order` parameter to the name of the field to order by.

```
GET /api/v2/pages/?order=title

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 50
    },
    "items": [
        pages will be listed here in ascending title order (a-z)
    ]
}
```

The results will be ordered in ascending order by default. This can be changed to descending order by prefixing the field name with a `-` sign.

```
GET /api/v2/pages/?order=-title

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 50
    },
    "items": [
        pages will be listed here in descending title order (z-a)
    ]
}
```

Note: Ordering is case-sensitive so lowercase letters are always ordered after uppercase letters when in ascending order.

Random ordering

Passing `random` into the `?order` parameter will make results return in a random order. If there is no caching, each request will return results in a different order.

```
GET /api/v2/pages/?order=random

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
```

(continues on next page)

(continued from previous page)

```
        "total_count": 50
    },
    "items": [
        pages will be listed here in random order
    ]
}
```

Note: It's not possible to use `?offset` while ordering randomly because consistent random ordering cannot be guaranteed over multiple requests (so requests for subsequent pages may return results that also appeared in previous pages).

Filtering

Any field may be used in an exact match filter. Use the filter name as the parameter and the value to match against.

For example, to find a page with the slug “about”:

```
GET /api/v2/pages/?slug=about

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 1
    },
    "items": [
        {
            "id": 10,
            "meta": {
                "type": "standard.StandardPage",
                "detail_url": "http://api.example.com/api/v2/pages/10/",
                "html_url": "http://www.example.com/about/",
                "slug": "about",
                "first_published_at": "2016-08-30T16:52:00Z"
            },
            "title": "About"
        },
    ]
}
```

Filtering by tree position (pages only)

Pages can additionally be filtered by their relation to other pages in the tree.

The `?child_of` filter takes the id of a page and filters the list of results to contain only direct children of that page.

For example, this can be useful for constructing the main menu, by passing the id of the homepage to the filter:

```
GET /api/v2/pages/?child_of=2&show_in_menus=true

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 5
    },
    "items": [
        {
            "id": 3,
            "meta": {
                "type": "blog.BlogIndexPage",
                "detail_url": "http://api.example.com/api/v2/pages/3/",
                "html_url": "http://www.example.com/blog/",
                "slug": "blog",
                "first_published_at": "2016-09-21T13:54:00Z"
            },
            "title": "About"
        },
        {
            "id": 10,
            "meta": {
                "type": "standard.StandardPage",
                "detail_url": "http://api.example.com/api/v2/pages/10/",
                "html_url": "http://www.example.com/about/",
                "slug": "about",
                "first_published_at": "2016-08-30T16:52:00Z"
            },
            "title": "About"
        },
        ...
    ]
}
```

The `?ancestor_of` filter takes the id of a page and filters the list to only include ancestors of that page (parent, grandparent etc.) all the way down to the site's root page.

For example, when combined with the `type` filter it can be used to find the particular `blog.BlogIndexPage` a `blog.BlogPage` belongs to. By itself, it can be used to construct a breadcrumb trail from the current page back to the site's root page.

The `?descendant_of` filter takes the id of a page and filter the list to only include descendants of that page (children, grandchildren etc.).

Filtering pages by site

New in version 4.0.

By default, the API will look for the site based on the hostname of the request. In some cases, you might want to query pages belonging to a different site. The `?site=` filter is used to filter the listing to only include pages that belong to a specific site. The filter requires the configured hostname of the site. If you have multiple sites using the same hostname but a different port number, it's possible to filter by port number using the format `hostname:port`. For example:

```
GET /api/v2/pages/?site=demo-site.local  
GET /api/v2/pages/?site=demo-site.local:8080
```

Search

Passing a query to the `?search` parameter will perform a full-text search on the results.

The query is split into “terms” (by word boundary), then each term is normalized (lowercased and unaccented).

For example: `?search=James+Joyce`

Search operator

The `search_operator` specifies how multiple terms in the query should be handled. There are two possible values:

- `and` - All terms in the search query (excluding stop words) must exist in each result
- `or` - At least one term in the search query must exist in each result

The `or` operator is generally better than `and` as it allows the user to be inexact with their query and the ranking algorithm will make sure that irrelevant results are not returned at the top of the page.

The default search operator depends on whether the search engine being used by the site supports ranking. If it does (Elasticsearch), the operator will default to `or`. Otherwise (database), it will default to `and`.

For the same reason, it's also recommended to use the `and` operator when using `?search` in conjunction with `?order` (as this disables ranking).

For example: `?search=James+Joyce&order=-first_published_at&search_operator=and`

Special filters for internationalized sites

When `WAGTAIL_I18N_ENABLED` is set to `True` (see [Enabling internationalisation](#) for more details) two new filters are made available on the `pages` endpoint.

Filtering pages by locale

The `?locale=` filter is used to filter the listing to only include pages in the specified locale. For example:

```
GET /api/v2/pages/?locale=en-us  
  
HTTP 200 OK  
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
{
    "meta": {
        "total_count": 5
    },
    "items": [
        {
            "id": 10,
            "meta": {
                "type": "standard.StandardPage",
                "detail_url": "http://api.example.com/api/v2/pages/10/",
                "html_url": "http://www.example.com/usa-page/",
                "slug": "usa-page",
                "first_published_at": "2016-08-30T16:52:00Z",
                "locale": "en-us"
            },
            "title": "American page"
        },
        ...
    ]
}
```

Getting translations of a page

The `?translation_of` filter is used to filter the listing to only include pages that are a translation of the specified page ID. For example:

```
GET /api/v2/pages/?translation_of=10

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 2
    },
    "items": [
        {
            "id": 11,
            "meta": {
                "type": "standard.StandardPage",
                "detail_url": "http://api.example.com/api/v2/pages/11/",
                "html_url": "http://www.example.com/gb-page/",
                "slug": "gb-page",
                "first_published_at": "2016-08-30T16:52:00Z",
                "locale": "en-gb"
            },
            "title": "British page"
        },
        {
            "id": 12,
```

(continues on next page)

(continued from previous page)

```
"meta": [
    "type": "standard.StandardPage",
    "detail_url": "http://api.example.com/api/v2/pages/12/",
    "html_url": "http://www.example.com/fr-page/",
    "slug": "fr-page",
    "first_published_at": "2016-08-30T16:52:00Z",
    "locale": "fr"
],
"title": "French page"
],
}
```

Fields

By default, only a subset of the available fields are returned in the response. The `?fields` parameter can be used to both add additional fields to the response and remove default fields that you know you won't need.

Additional fields

Additional fields can be added to the response by setting `?fields` to a comma-separated list of field names you want to add.

For example, `?fields=body,feed_image` will add the `body` and `feed_image` fields to the response.

This can also be used across relationships. For example, `?fields=body,feed_image(width,height)` will nest the `width` and `height` of the image in the response.

All fields

Setting `?fields` to an asterisk (*) will add all available fields to the response. This is useful for discovering what fields have been exported.

For example: `?fields=*`

Removing fields

Fields you know that you do not need can be removed by prefixing the name with a - and adding it to `?fields`.

For example, `?fields=-title,body` will remove `title` and add `body`.

This can also be used with the asterisk. For example, `?fields=*, -body` adds all fields except for `body`.

Removing all default fields

To specify exactly the fields you need, you can set the first item in `fields` to an underscore (`_`) which removes all default fields.

For example, `?fields=_`,`title` will only return the `title` field.

Detail views

You can retrieve a single object from the API by appending its id to the end of the URL. For example:

- Pages `/api/v2/pages/1/`
- Images `/api/v2/images/1/`
- Documents `/api/v2/documents/1/`

All exported fields will be returned in the response by default. You can use the `?fields` parameter to customize which fields are shown.

For example: `/api/v2/pages/1/?fields=_`,`title`,`body` will return just the `title` and `body` of the page with the id of 1.

Finding pages by HTML path

You can find an individual page by its HTML path using the `/api/v2/pages/find/?html_path=<path>` view.

This will return either a `302` redirect response to that page's detail view, or a `404` not found response.

For example: `/api/v2/pages/find/?html_path=/` always redirects to the homepage of the site

Default endpoint fields

Common fields

These fields are returned by every endpoint.

id (number) The unique ID of the object

Note: Except for page types, every other content type has its own id space so you must combine this with the `type` field in order to get a unique identifier for an object.

type (string) The type of the object in `app_label.ModelName` format

detail_url (string) The URL of the detail view for the object

Pages

title (string) **meta.slug** (string) **meta.show_in_menus** (boolean) **meta.seo_title** (string) **meta.search_description** (string) **meta.first_published_at** (date/time) These values are taken from their corresponding fields on the page

meta.html_url (string) If the site has an HTML frontend that's generated by Wagtail, this field will be set to the URL of this page

meta.parent Nests some information about the parent page (only available on detail views)

meta.alias_of (dictionary) If the page marked as an alias return original page id and full url

Images

title (string) The value of the image's title field. Within Wagtail, this is used in the image's alt HTML attribute.

width (number) **height** (number) The size of the original image file

meta.tags (list of strings) A list of tags associated with the image

Documents

title (string) The value of the document's title field

meta.tags (list of strings) A list of tags associated with the document

meta.download_url (string) A URL to the document file

Changes since v1

Breaking changes

- The results list in listing responses has been renamed to `items` (was previously either `pages`, `images` or `documents`)

Major features

- The `fields` parameter has been improved to allow removing fields, adding all fields and customising nested fields

Minor features

- `html_url`, `slug`, `first_published_at`, `expires_at` and `show_in_menus` fields have been added to the `pages` endpoint
- `download_url` field has been added to the `documents` endpoint
- Multiple page types can be specified in `type` parameter on `pages` endpoint
- `true` and `false` may now be used when filtering boolean fields
- `order` can now be used in conjunction with `search`

- `search_operator` parameter was added

1.3.13 How to build a site with AMP support

This recipe document describes a method for creating an [AMP](#) version of a Wagtail site and hosting it separately to the rest of the site on a URL prefix. It also describes how to make Wagtail render images with the `<amp-img>` tag when a user is visiting a page on the AMP version of the site.

Overview

In the next section, we will add a new URL entry that points at Wagtail’s internal `serve()` view which will have the effect of rendering the whole site again under the `/amp` prefix.

Then, we will add some utilities that will allow us to track whether the current request is in the `/amp` prefixed version of the site without needing a request object.

After that, we will add a template context processor to allow us to check from within templates which version of the site is being rendered.

Then, finally, we will modify the behaviour of the `{% image %}` tag to make it render `<amp-img>` tags when rendering the AMP version of the site.

Creating the second page tree

We can render the whole site at a different prefix by duplicating the Wagtail URL in the project `urls.py` file and giving it a prefix. This must be before the default URL from Wagtail, or it will try to find `/amp` as a page:

```
# <project>/urls.py

urlpatterns += [
    # Add this line just before the default ``include(wagtail_urls)`` line
    path('amp/', include(wagtail_urls)),
    path('', include(wagtail_urls)),
]
```

If you now open `http://localhost:8000/amp/` in your browser, you should see the homepage.

Making pages aware of “AMP mode”

All the pages will now render under the `/amp` prefix, but right now there isn’t any difference between the AMP version and the normal version.

To make changes, we need to add a way to detect which URL was used to render the page. To do this, we will have to wrap Wagtail’s `serve()` view and set a thread-local to indicate to all downstream code that AMP mode is active.

Note: Why a thread-local?

(feel free to skip this part if you’re not interested)

Modifying the `request` object would be the most common way to do this. However, the image tag rendering is performed in a part of Wagtail that does not have access to the `request`.

Threadlocals are global variables that can have a different value for each running thread. As each thread only handles one request at a time, we can use it as a way to pass around data that is specific to that request without having to pass the request object everywhere.

Django uses threadlocals internally to track the currently active language for the request.

Python implements thread-local data through the `threading.local` class, but as of Django 3.x, multiple requests can be handled in a single thread and so threadlocals will no longer be unique to a single request. Django therefore provides `asgiref.Local` as a drop-in replacement.

Now let's create that thread-local and some utility functions to interact with it, save this module as `amp_utils.py` in an app in your project:

```
# <app>/amp_utils.py

from contextlib import contextmanager
from asgiref.local import Local

_amp_mode_active = Local()

@contextmanager
def activate_amp_mode():
    """
    A context manager used to activate AMP mode
    """
    _amp_mode_active.value = True
    try:
        yield
    finally:
        del _amp_mode_active.value

def amp_mode_active():
    """
    Returns True if AMP mode is currently active
    """
    return hasattr(_amp_mode_active, 'value')
```

This module defines two functions:

- `activate_amp_mode` is a context manager which can be invoked using Python's `with` syntax. In the body of the `with` statement, AMP mode would be active.
- `amp_mode_active` is a function that returns `True` when AMP mode is active.

Next, we need to define a view that wraps Wagtail's builtin `serve` view and invokes the `activate_amp_mode` context manager:

```
# <app>/amp_views.py

from django.template.response import SimpleTemplateResponse
from wagtail.views import serve as wagtail_serve

from .amp_utils import activate_amp_mode

def serve(request, path):
    with activate_amp_mode():
```

(continues on next page)

(continued from previous page)

```

response = wagtail_serve(request, path)

# Render template responses now while AMP mode is still active
if isinstance(response, SimpleTemplateResponse):
    response.render()

return response

```

Then we need to create a `amp_urls.py` file in the same app:

```

# <app>/amp_urls.py

from django.urls import re_path
from wagtail.urls import serve_pattern

from . import amp_views

urlpatterns = [
    re_path(serve_pattern, amp_views.serve, name='wagtail_amp_serve')
]

```

Finally, we need to update the project's main `urls.py` to use this new URLs file for the `/amp` prefix:

```

# <project>/urls.py

from myapp import amp_urls as wagtail_amp_urls

urlpatterns += [
    # Change this line to point at your amp_urls instead of Wagtail's urls
    path('amp/', include(wagtail_amp_urls)),

    re_path(r'', include(wagtail_urls)),
]

```

After this, there shouldn't be any noticeable difference to the AMP version of the site.

Write a template context processor so that AMP state can be checked in templates

This is optional, but worth doing so we can confirm that everything is working so far.

Add a `amp_context_processors.py` file into your app that contains the following:

```

# <app>/amp_context_processors.py

from .amp_utils import amp_mode_active

def amp(request):
    return {
        'amp_mode_active': amp_mode_active(),
    }

```

Now add the path to this context processor to the `['OPTIONS']['context_processors']` key of the `TEMPLATES` setting:

```
# Either <project>/settings.py or <project>/settings/base.py

TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                # Add this after other context processors
                'myapp.amp_context_processors.amp',
            ],
        },
    },
]
```

You should now be able to use the `amp_mode_active` variable in templates. For example:

```
{% if amp_mode_active %}
    AMP MODE IS ACTIVE!
{% endif %}
```

Using a different page template when AMP mode is active

You're probably not going to want to use the same templates on the AMP site as you do on the normal web site. Let's add some logic in to make Wagtail use a separate template whenever a page is served with AMP enabled.

We can use a mixin, which allows us to re-use the logic on different page types. Add the following to the bottom of the `amp_utils.py` file that you created earlier:

```
# <app>/amp_utils.py

import os.path
...
class PageAMPTemplateMixin:

    @property
    def amp_template(self):
        # Get the default template name and insert `'_amp` before the extension
        name, ext = os.path.splitext(self.template)
        return name + '_amp' + ext

    def get_template(self, request):
        if amp_mode_active():
            return self.amp_template

    return super().get_template(request)
```

Now add this mixin to any page model, for example:

```
# <app>/models.py

from .amp_utils import PageAMPTemplateMixin

class MyPageModel(PageAMPTemplateMixin, Page):
    ...
```

When AMP mode is active, the template at `app_label/mypagemodel_amp.html` will be used instead of the default one.

If you have a different naming convention, you can override the `amp_template` attribute on the model. For example:

```
# <app>/models.py

from .amp_utils import PageAMPTemplateMixin

class MyPageModel(PageAMPTemplateMixin, Page):
    amp_template = 'my_custom_amp_template.html'
```

Overriding the `{% image %}` tag to output `<amp-img>` tags

Finally, let's change Wagtail's `{% image %}` tag, so it renders an `<amp-img>` tags when rendering pages with AMP enabled. We'll make the change on the Rendition model itself so it applies to both images rendered with the `{% image %}` tag and images rendered in rich text fields as well.

Doing this with a *Custom image model* is easier, as you can override the `img_tag` method on your custom Rendition model to return a different tag.

For example:

```
from django.forms.utils import flatatt
from django.utils.safestring import mark_safe

from wagtail.images.models import AbstractRendition

...

class CustomRendition(AbstractRendition):
    def img_tag(self, extra_attributes):
        attrs = self.attrs_dict.copy()
        attrs.update(extra_attributes)

        if amp_mode_active():
            return mark_safe('<amp-img{}>'.format(flatatt(attrs)))
        else:
            return mark_safe('<img{}>'.format(flatatt(attrs)))

...
```

Without a custom image model, you will have to monkey-patch the builtin Rendition model. Add this anywhere in your project where it would be imported on start:

```
from django.forms.utils import flatatt
from django.utils.safestring import mark_safe
```

(continues on next page)

(continued from previous page)

```
from wagtail.images.models import Rendition

def img_tag(rendition, extra_attributes={}):
    """
    Replacement implementation for Rendition.img_tag

    When AMP mode is on, this returns an <amp-img> tag instead of an <img> tag
    """
    attrs = rendition.attrs_dict.copy()
    attrs.update(extra_attributes)

    if amp_mode_active():
        return mark_safe('<amp-img{}>'.format(flatatt(attrs)))
    else:
        return mark_safe('<img{}>'.format(flatatt(attrs)))

Rendition.img_tag = img_tag
```

1.3.14 Accessibility considerations

Accessibility for CMS-driven websites is a matter of *modeling content appropriately, creating accessible templates, and authoring accessible content* with readability and accessibility guidelines in mind.

Wagtail generally puts developers in control of content modeling and front-end markup, but there are a few areas to be aware of nonetheless, and ways to help authors be aware of readability best practices. Note there is much more to building accessible websites than we cover here – see our list of *accessibility resources* for more information.

- *Content modeling*
- *Accessibility in templates*
- *Authoring accessible content*
- *Accessibility resources*

Content modeling

As part of defining your site’s models, here are areas to pay special attention to:

Alt text for images

The default behaviour for Wagtail images is to use the `title` field as the alt text ([#4945](#)). This is inappropriate, as it’s not communicated in the CMS interface, and the image upload form uses the image’s filename as the title by default.

Ideally, always add an optional “alt text” field wherever an image is used, alongside the image field:

- For normal fields, add an alt text field to your image’s panel.
- For StreamField, add an extra field to your image block.
- For rich text – Wagtail already makes it possible to customise alt text for rich text images.

When defining the alt text fields, make sure they are optional so editors can choose to not write any alt text for decorative images. Take the time to provide `help_text` with appropriate guidance. For example, linking to established resources on alt text.

Note: Should I add an alt text field on the Image model for my site?

It's better than nothing to have a dedicated alt field on the Image model ([#5789](#)), and may be appropriate for some websites, but we recommend to have it inline with the content because ideally alt text should be written for the context the image is used in:

- If the alt text's content is already part of the rest of the page, ideally the image should not repeat the same content.
 - Ideally, the alt text should be written based on the context the image is displayed in.
 - An image might be decorative in some cases but not in others. For example, thumbnails in page listings can often be considered decorative.
-

See [RFC 51: Contextual alt text](#) for a long-term solution to this problem.

Embeds title

Missing embed titles are common failures in accessibility audits of Wagtail websites. In some cases, Wagtail embeds' iframe doesn't have a `title` attribute set. This is generally a problem with OEmbed providers like YouTube ([#5982](#)). This is very problematic for screen reader users, who rely on the title to understand what the embed is, and whether to interact with it or not.

If your website relies on embeds that have are missing titles, make sure to either:

- Add the OEmbed `title` field as a `title` on the `iframe`.
- Add a custom mandatory Title field to your embeds, and add it as the `iframe`'s `title`.

Available heading levels

Wagtail makes it very easy for developers to control which heading levels should be available for any given content, via [rich text features](#) or custom StreamField blocks. In both cases, take the time to restrict what heading levels are available so the pages' document outline is more likely to be logical and sequential. Consider using the following restrictions:

- Disallow `h1` in rich text. There should only be one `h1` tag per page, which generally maps to the page's `title`.
- Limit heading levels to `h2` for the main content of a page. Add `h3` only if deemed necessary. Avoid other levels as a general rule.
- For content that is displayed in a specific section of the page, limit heading levels to those directly below the section's main heading.

If managing headings via StreamField, make sure to apply the same restrictions there.

Bold and italic formatting in rich text

By default, Wagtail stores its bold formatting as a `b` tag, and italic as `i` (#4665). While those tags don't necessarily always have correct semantics (`strong` and `em` are more ubiquitous), there isn't much consequence for screen reader users, as by default screen readers do not announce content differently based on emphasis.

If this is a concern to you, you can change which tags are used when saving content with *rich text format converters*. In the future, *rich text rewrite handlers* should also support this being done without altering the storage format (#4223).

TableBlock

The `TableBlock` default implementation makes it too easy for end-users to miss they need either row or column headers (#5989). Make sure to always have either row headers or column headers set. Always add a Caption, so screen reader users navigating the site's tables know where they are.

Accessibility in templates

Here are common gotchas to be aware of to make the site's templates as accessible as possible,

Alt text in templates

See the *content modelling* section above. Additionally, make sure to *customise images' alt text*, either setting it to the relevant field, or to an empty string for decorative images, or images where the alt text would be a repeat of other content. Even when your images have alt text coming directly from the image model, you still need to decide whether there should be alt text for the particular context the image is used in. For example, avoid alt text in listings where the alt text just repeats the listing items' title.

Empty heading tags

In both rich text and custom StreamField blocks, it's sometimes easy for editors to create a heading block but not add any content to it. If this is a problem for your site,

- Add validation rules to those fields, making sure the page can't be saved with the empty headings, for example by using the `StreamField CharBlock` which is required by default.
- Consider adding similar validation rules for rich text fields (#6526).

Additionally, you can hide empty heading blocks with CSS:

```
h1:empty,  
h2:empty,  
h3:empty,  
h4:empty,  
h5:empty,  
h6:empty {  
    display: none;  
}
```

Forms

The [Form builder](#) uses Django's forms API. Here are considerations specific to forms in templates:

- Avoid rendering helpers such as `as_table`, `as_ul`, `as_p`, which can make forms harder to navigate for screen reader users or cause HTML validation issues (see Django ticket [#32339](#)).
- Make sure to visually distinguish required and optional fields.
- Take the time to group related fields together in `fieldset`, with an appropriate legend, in particular for radios and checkboxes (see Django ticket [#32338](#)).
- If relevant, use the appropriate `autocomplete` and `autocapitalize` attributes.
- For Date and Datetime fields, make sure to display the expected format or an example value (see Django ticket [#32340](#)). Or use `input type="date"`.
- For Number fields, consider whether `input type="number"` really is appropriate, or whether there may be better alternatives such as `inputmode`.

Make sure to test your forms' implementation with assistive technologies, and review [official W3C guidance on accessible forms development](#) for further information.

Authoring accessible content

Here are things you can do to help authors create accessible content.

wagtail-accessibility

[wagtail-accessibility](#) is a third-party package which adds `totally` to Wagtail previews. This makes it easy for authors to run basic accessibility checks – validating the page's heading outline, or link text.

help_text and HelpPanel

Occasional Wagtail users may not be aware of your site's content guidelines, or best practices of writing for the web. Use fields' `help_text` and `HelpPanel` (see [Panel types](#)).

Readability

Readability is fundamental to accessibility. One of the ways to improve text content is to have a clear target for reading level / reading age, which can be assessed with [wagtail-readinglevel](#) as a score displayed in rich text fields.

Accessibility resources

We focus on considerations specific to Wagtail websites, but there is much more to accessibility. Here are valuable resources to learn more, for developers but also designers and authors:

- [W3C Accessibility Fundamentals](#)
- [The A11Y Project](#)
- [US GSA – Accessibility for Teams](#)
- [UK GDS – Dos and don'ts on designing for accessibility](#)

- Accessibility Developer Guide

1.3.15 About StreamField BoundBlocks and values

All StreamField block types accept a `template` parameter to determine how they will be rendered on a page. However, for blocks that handle basic Python data types, such as `CharBlock` and `IntegerBlock`, there are some limitations on where the template will take effect, since those built-in types (`str`, `int` and so on) cannot be ‘taught’ about their template rendering. As an example of this, consider the following block definition:

```
class HeadingBlock(blocks.CharBlock):  
    class Meta:  
        template = 'blocks/heading.html'
```

where `blocks/heading.html` consists of:

```
<h1>{{ value }}</h1>
```

This gives us a block that behaves as an ordinary text field, but wraps its output in `<h1>` tags whenever it is rendered:

```
class BlogPage(Page):  
    body = StreamField([  
        # ...  
        ('heading', HeadingBlock()),  
        # ...  
    ], use_json_field=True)
```

```
{% load wagtailcore_tags %}  
  
{% for block in page.body %}  
    {% if block.block_type == 'heading' %}  
        {% include_block block %}  {# This block will output its own <h1>...</h1> tags.  
    #}  
    {% endif %}  
{% endfor %}
```

This kind of arrangement - a value that supposedly represents a plain text string, but has its own custom HTML representation when output on a template - would normally be a very messy thing to achieve in Python, but it works here because the items you get when iterating over a StreamField are not actually the ‘native’ values of the blocks. Instead, each item is returned as an instance of `BoundBlock` - an object that represents the pairing of a value and its block definition. By keeping track of the block definition, a `BoundBlock` always knows which template to render. To get to the underlying value - in this case, the text content of the heading - you would need to access `block.value`. Indeed, if you were to output `{% include_block block.value %}` on the page, you would find that it renders as plain text, without the `<h1>` tags.

(More precisely, the items returned when iterating over a StreamField are instances of a class `StreamChild`, which provides the `block_type` property as well as `value`.)

Experienced Django developers may find it helpful to compare this to the `BoundField` class in Django’s forms framework, which represents the pairing of a form field value with its corresponding form field definition, and therefore knows how to render the value as an HTML form field.

Most of the time, you won’t need to worry about these internal details; Wagtail will use the template rendering wherever you would expect it to. However, there are certain cases where the illusion isn’t quite complete - namely, when accessing children of a `ListBlock` or `StructBlock`. In these cases, there is no `BoundBlock` wrapper, and so the item cannot be

relied upon to know its own template rendering. For example, consider the following setup, where our `HeadingBlock` is a child of a `StructBlock`:

```
class EventBlock(blocks.StructBlock):
    heading = HeadingBlock()
    description = blocks.TextBlock()
    # ...

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

In `blocks/event.html`:

```
{% load wagtailcore_tags %}

<div class="event {% if value.heading == 'Party!' %}lots-of-balloons{% endif %}">
    {% include_block value.heading %}
    - {% include_block value.description %}
</div>
```

In this case, `value.heading` returns the plain string value rather than a `BoundBlock`; this is necessary because otherwise the comparison in `{% if value.heading == 'Party!' %}` would never succeed. This in turn means that `{% include_block value.heading %}` renders as the plain string, without the `<h1>` tags. To get the HTML rendering, you need to explicitly access the `BoundBlock` instance through `value.bound_blocks.heading`:

```
{% load wagtailcore_tags %}

<div class="event {% if value.heading == 'Party!' %}lots-of-balloons{% endif %}">
    {% include_block value.bound_blocks.heading %}
    - {% include_block value.description %}
</div>
```

In practice, it would probably be more natural and readable to make the `<h1>` tag explicit in the `EventBlock`'s template:

```
{% load wagtailcore_tags %}

<div class="event {% if value.heading == 'Party!' %}lots-of-balloons{% endif %}">
    <h1>{{ value.heading }}</h1>
    - {% include_block value.description %}
</div>
```

This limitation does not apply to `StructBlock` and `StreamBlock` values as children of a `StructBlock`, because Wagtail implements these as complex objects that know their own template rendering, even when not wrapped in a `BoundBlock`. For example, if a `StructBlock` is nested in another `StructBlock`, as in:

```
class EventBlock(blocks.StructBlock):
    heading = HeadingBlock()
    description = blocks.TextBlock()
    guest_speaker = blocks.StructBlock([
        ('first_name', blocks.CharBlock()),
        ('surname', blocks.CharBlock()),
        ('photo', ImageChooserBlock()),
    ], template='blocks/speaker.html')
```

then `{% include_block value.guest_speaker %}` within the `EventBlock`'s template will pick up the template

rendering from `blocks/speaker.html` as intended.

In summary, interactions between BoundBlocks and plain values work according to the following rules:

1. When iterating over the value of a StreamField or StreamBlock (as in `{% for block in page.body %}`), you will get back a sequence of BoundBlocks.
2. If you have a BoundBlock instance, you can access the plain value as `block.value`.
3. Accessing a child of a StructBlock (as in `value.heading`) will return a plain value; to retrieve the BoundBlock instead, use `value.bound_blocks.heading`.
4. Likewise, accessing children of a ListBlock (for example `for item in value`) will return plain values; to retrieve BoundBlocks instead, use `value.bound_blocks`.
5. StructBlock and StreamBlock values always know how to render their own templates, even if you only have the plain value rather than the BoundBlock.

Changed in version 2.16: The value of a ListBlock now provides a `bound_blocks` property; previously it was a plain Python list of child values.

1.3.16 Multi-site, multi-instance and multi-tenancy

This page gives background information on how to run multiple Wagtail sites (with the same source code).

- [Multi-site](#)
- [Multi-instance](#)
- [Multi-tenancy](#)

Multi-site

Multi-site is a Wagtail project configuration where content creators go into a single admin interface and manage the content of multiple websites. Permission to manage specific content, and restricting access to other content, is possible to some extent.

Multi-site configuration is a single code base, on a single server, connecting to a single database. Media is stored in a single media root directory. Content can be shared between sites.

Wagtail supports multi-site out of the box: Wagtail comes with a [*site model*](#). The site model contains a hostname, port, and root page field. When a URL is requested, the request comes in, the domain name and port are taken from the request object to look up the correct site object. The root page is used as starting point to resolve the URL and serve the correct page.

Wagtail also comes with [*site settings*](#). *Site settings* are ‘singletons’ that let you store additional information on a site. For example, social media settings, a field to upload a logo, or a choice field to select a theme.

Model objects can be linked to a site by placing a foreign key field on the model pointing to the site object. A request object can be used to look up the current site. This way, content belonging to a specific site can be served.

User, groups, and permissions can be configured in such a way that content creators can only manage the pages, images, and documents of a specific site. Wagtail can have multiple *site objects* and multiple *page trees*. Permissions can be linked to a specific page tree or a subsection thereof. Collections are used to categorize images and documents. A collection can be restricted to users who are in a specific group.

Some projects require content editors to have permissions on specific sites and restrict access to other sites. Splitting *all* content per site and guaranteeing that no content ‘leaks’ is difficult to realize in a multi-site project. If you require full separation of content, then multi-instance might be a better fit...

Multi-instance

Multi-instance is a Wagtail project configuration where a single set of project files is used by multiple websites. Each website has its own settings file, and a dedicated database and media directory. Each website runs in its own server process. This guarantees the *total separation of all content*.

Assume the domains a.com and b.com. Settings files can be `base.py`, `a.com.py`, and `b.com.py`. The base settings will contain all settings like normal. The contents of site-specific settings override the base settings:

```
# settings/a.com.py

from base import *

ALLOWED_HOSTS = ['a.com']
DATABASES["NAME"] = "a.com"
DATABASES["PASSWORD"] = "password-for-a.com"
MEDIA_DIR = BASE_DIR / "a.com-media"
```

Each site can be started with its own settings file. In development `./manage.py runserver --settings settings.a.com`. In production, for example with uWSGI, specify the correct settings with `env = DJANGO_SETTINGS_MODULE=settings.a.com`.

Because each site has its own database and media folder, nothing can ‘leak’ to another site. But this also means that content cannot be shared between sites as one can do when using the multi-site option.

In this configuration, multiple sites share the same, single set of project files. Deployment would update the single set of project files and reload each instance.

This multi-instance configuration isn’t that different from deploying the project code several times. However, having a single set of project files, and only differentiate with settings files, is the closest Wagtail can get to true multi-tenancy. Every site is identical, content is separated, including user management. ‘Adding a new tenant’ is adding a new settings file and running a new instance.

In a multi-instance configuration, each instance requires a certain amount of server resources (CPU and memory). That means adding sites will increase server load. This only scales up to a certain point.

Multi-tenancy

Multi-tenancy is a project configuration in which a single instance of the software serves multiple tenants. A tenant is a group of users who have access and permission to a single site. Multitenant software is designed to provide every tenant its configuration, data, and user management.

Wagtail supports *multi-site*, where user management and content are shared. Wagtail can run *multi-instance* where there is full separation of content at the cost of running multiple instances. Multi-tenancy combines the best of both worlds: a single instance, and the full separation of content per site and user management.

Wagtail does not support full multi-tenancy at this moment. But it is on our radar, we would like to improve Wagtail to add multi-tenancy - while still supporting the existing multi-site option. If you have ideas or like to contribute, join us on [Slack](#) in the multi-tenancy channel.

Wagtail currently has the following features to support multi-tenancy:

- A Site model mapping a hostname to a root page

- Permissions to allow groups of users to manage:
 - arbitrary sections of the page tree
 - sections of the collection tree (coming soon)
 - one or more collections of documents and images
- The page API is automatically scoped to the host used for the request

But several features do not currently support multi-tenancy:

- Snippets are global pieces of content so not suitable for multi-tenancy but any model that can be registered as a snippet can also be managed via the Wagtail model admin. You can add a `site_id` to the model and then use the model admin `get_queryset` method to determine which site can manage each object. The built-in snippet choosers can be replaced by `modelchooser` that allows filtering the queryset to restrict which sites may display which objects.
- Site, site setting, user, and group management. At the moment, your best bet is to only allow superusers to manage these objects.
- Workflows and workflow tasks
- Site history
- Redirects

Permission configuration for built-in models like Sites, Site settings and Users is not site-specific, so any user with permission to edit a single entry can edit them all. This limitation can be mostly circumvented by only allowing superusers to manage these models.

Python, Django, and Wagtail allow you to override, extend and customise functionality. Here are some ideas that may help you create a multi-tenancy solution for your site:

- Django allows to override templates, this also works in the Wagtail admin.
- A custom user model can be used to link users to a specific site.
- Custom admin views can provide more restrictive user management.

We welcome interested members of the Wagtail community to contribute code and ideas.

1.3.17 How to use a redirect with Form builder to prevent double submission

It is common for form submission HTTP responses to be a `302 Found` temporary redirection to a new page. By default `wagtail.contrib.forms.models.FormPage` success responses don't do this, meaning there is a risk that users will refresh the success page and re-submit their information.

Instead of rendering the `render_landing_page` content in the POST response, we will redirect to a `route` of the `FormPage` instance at a child URL path. The content will still be managed within the same form page's admin. This approach uses the additional contrib module `wagtail.contrib.routable_page`.

An alternative approach is to redirect to an entirely different page, which does not require the `routable_page` module. See [Custom landing page redirect](#).

Make sure "`wagtail.contrib.routable_page`" is added to `INSTALLED_APPS`, see [RoutablePageMixin](#) documentation.

```
from django.shortcuts import redirect

from wagtail.contrib.forms.models import AbstractEmailForm
from wagtail.contrib.routable_page.models import RoutablePageMixin, path
```

(continues on next page)

(continued from previous page)

```
class FormPage(RoutablePageMixin, AbstractEmailForm):  
  
    # fields, content_panels, ...  
  
    @path("")  
    def index_route(self, request, *args, **kwargs):  
        """Serve the form, and validate it on POST"""  
        return super(AbstractEmailForm, self).serve(request, *args, **kwargs)  
  
    def render_landing_page(self, request, form_submission, *args, **kwargs):  
        """Redirect instead to self.thank_you route"""  
        url = self.reverse_subpage("thank_you")  
        # If a form_submission instance is available, append the ID to URL.  
        if form_submission:  
            url += "?id=%s" % form_submission.id  
        return redirect(self.url + url, permanent=False)  
  
    @path("thank-you/")  
    def thank_you(self, request):  
        """Return the superclass's landing page, after redirect."""  
        form_submission = None  
        try:  
            submission_id = int(request.GET["id"])  
        except (KeyError, TypeError):  
            pass  
        else:  
            submission_class = self.get_submission_class()  
            try:  
                form_submission = submission_class.objects.get(id=submission_id)  
            except submission_class.DoesNotExist:  
                pass  
  
        return super().render_landing_page(request, form_submission)
```

1.4 Extending Wagtail

The Wagtail admin interface is a suite of Django apps, and so the familiar concepts from Django development - views, templates, URL routes and so on - can be used to add new functionality to Wagtail. Numerous [third-party packages](#) can be installed to extend Wagtail's capabilities.

This section describes the various mechanisms that can be used to integrate your own code into Wagtail's admin interface.

1.4.1 Creating admin views

The most common use for adding custom views to the Wagtail admin is to provide an interface for managing a Django model. The [Index](#) app makes this simple, providing ready-made views for listing, creating and editing objects with minimal configuration.

For other kinds of admin view that don't fit this pattern, you can write your own Django views and register them as part of the Wagtail admin through [hooks](#). In this example, we'll implement a view that displays a calendar for the current year, using the [calendar module](#) from Python's standard library.

Defining a view

Within a Wagtail project, create a new `wagtailcalendar` app with `./manage.py startapp wagtailcalendar` and add it to your project's `INSTALLED_APPS`. (In this case we're using the name '`wagtailcalendar`' to avoid clashing with the standard library's `calendar` module - in general there is no need to use a '`wagtail`' prefix.)

Edit `views.py` as follows - note that this is a plain Django view with no Wagtail-specific code.

```
import calendar

from django.http import HttpResponse
from django.utils import timezone

def index(request):
    current_year = timezone.now().year
    calendar_html = calendar.HTMLCalendar().formatyear(current_year)

    return HttpResponse(calendar_html)
```

Registering a URL route

At this point, the standard practice for a Django project would be to add a URL route for this view to your project's top-level URL config module. However, in this case we want the view to only be available to logged-in users, and to appear within the `/admin/` URL namespace which is managed by Wagtail. This is done through the [Register Admin URLs](#) hook.

On startup, Wagtail looks for a `wagtail_hooks` submodule within each installed app. In this submodule, you can define functions to be run at various points in Wagtail's operation, such as building the URL config for the admin, and constructing the main menu.

Create a `wagtail_hooks.py` file within the `wagtailcalendar` app containing the following:

```
from django.urls import path
from wagtail import hooks

from .views import index

@hooks.register('register_admin_urls')
def register_calendar_url():
    return [
        path('calendar/', index, name='calendar'),
    ]
```

The calendar will now be visible at the URL `/admin/calendar/`.

January							February							March							
Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
4	5	6	7	8	9	10	1	2	3	4	5	6	7	1	2	3	4	5	6	7	
11	12	13	14	15	16	17	8	9	10	11	12	13	14	8	9	10	11	12	13	14	
18	19	20	21	22	23	24	15	16	17	18	19	20	21	15	16	17	18	19	20	21	
25	26	27	28	29	30	31	22	23	24	25	26	27	28	29	30	31	22	23	24	25	26
May							June							July							
Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
12	13	14	15	16	17	18	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
19	20	21	22	23	24	25	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
26	27	28	29	30		31	24	25	26	27	28	29	30	28	29	30	29	30	31		
August							September							October							
Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
12	13	14	15	16	17	18	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
19	20	21	22	23	24	25	16	17	18	19	20	21	22	20	21	22	23	24	25	26	
26	27	28	29	30	31		23	24	25	26	27	28	29	27	28	29	30	31			
November							December														
Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
1	2	3	4	5	6	7	1	2	3	4	5	6	7	1	2	3	4	5	6	7	
4	5	6	7	8	9	10	8	9	10	11	12	13	14	6	7	8	9	10	11	12	
11	12	13	14	15	16	17	15	16	17	18	19	20	21	13	14	15	16	17	18	19	
18	19	20	21	22	23	24	22	23	24	25	26	27	28	20	21	22	23	24	25	26	
25	26	27	28	29	30	31	29	30	31		30			27	28	29	30	31			

Adding a template

Currently this view is outputting a plain HTML fragment. Let's insert this into the usual Wagtail admin page furniture, by creating a template that extends Wagtail's base template `"wagtailadmin/base.html"`.

Note: The base template and HTML structure are not considered astable part of Wagtail's API, and may change in futurereleases.

Update `views.py` as follows:

```
import calendar
from django.shortcuts import render
from django.utils import timezone

def index(request):
    current_year = timezone.now().year
    calendar_html = calendar.HTMLCalendar().formatyear(current_year)

    return render(request, 'wagtailcalendar/index.html', {
        'current_year': current_year,
        'calendar_html': calendar_html,
    })
```

Now create a `templates/wagtailcalendar/` folder within the `wagtailcalendar` app, containing `index.html` as follows:

```
{% extends "wagtailadmin/base.html" %}
{% block titletag %}{{ current_year }} calendar{% endblock %}
```

(continues on next page)

(continued from previous page)

```

{% block extra_css %}
    {{ block.super }}
<style>
    table.month {
        margin: 20px;
    }
    table.month td, table.month th {
        padding: 5px;
    }
</style>
{% endblock %}

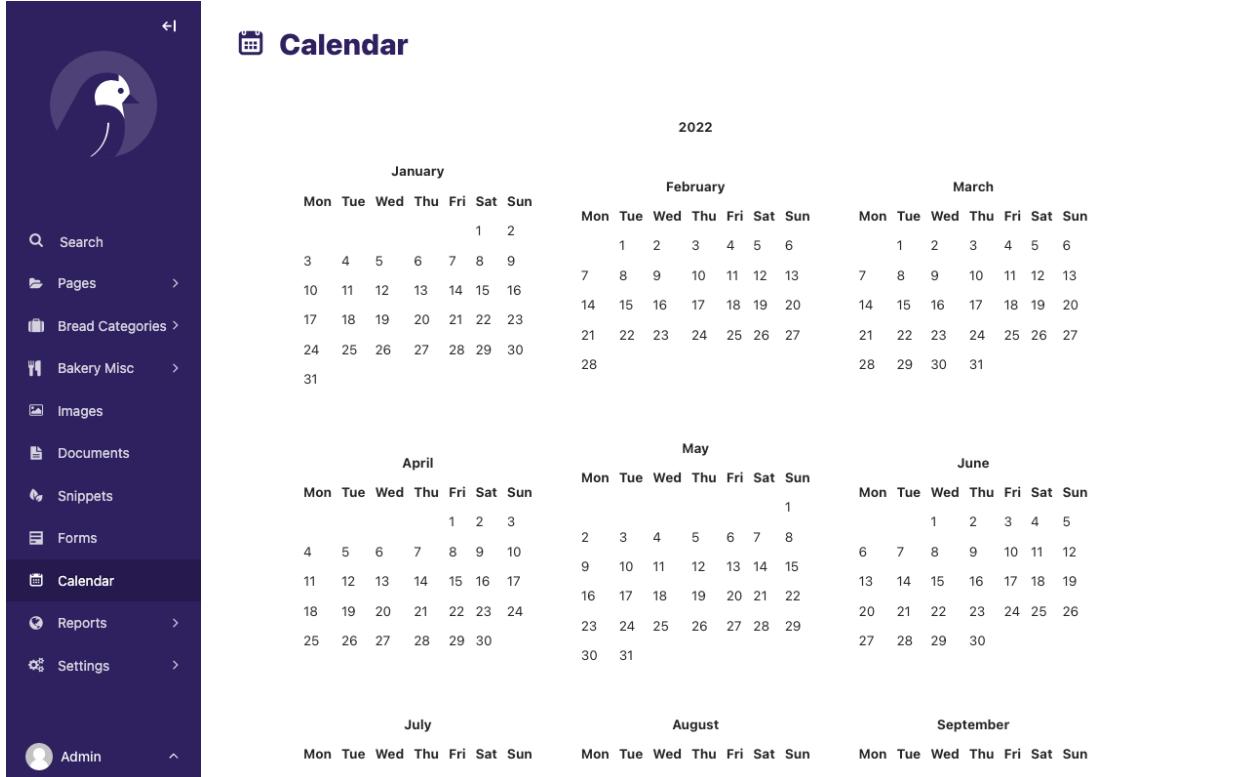
{% block content %}
    {% include "wagtailadmin/shared/header.html" with title="Calendar" icon="date" %}

    <div class="nice-padding">
        {{ calendar_html|safe }}
    </div>
{% endblock %}

```

Here we are overriding three of the blocks defined in the base template: `titletag` (which sets the content of the HTML `<title>` tag), `extra_css` (which allows us to provide additional CSS styles specific to this page), and `content` (for the main content area of the page). We're also including the standard header bar component, and setting a title and icon. For a list of the recognised icon identifiers, see the [style guide](#).

Revisiting `/admin/calendar/` will now show the calendar within the Wagtail admin page furniture.



Adding a menu item

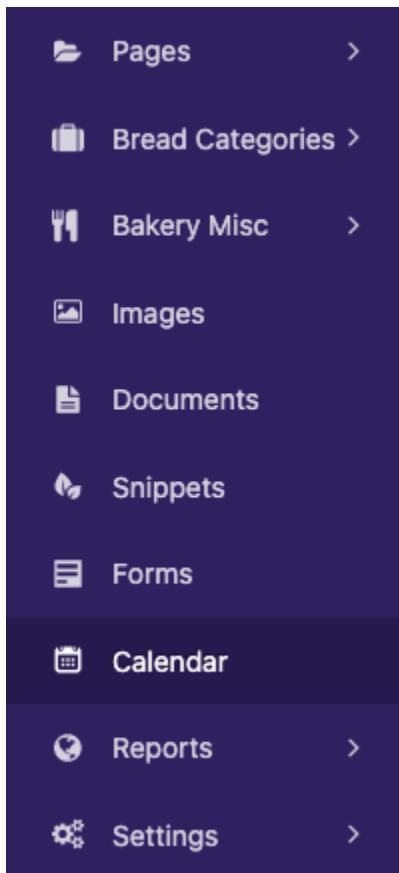
Our calendar view is now complete, but there's no way to reach it from the rest of the admin backend. To add an item to the sidebar menu, we'll use another hook, [Register Admin Menu Item](#). Update `wagtail_hooks.py` as follows:

```
from django.urls import path, reverse
from wagtail.admin.menu import MenuItem
from wagtail import hooks
from .views import index

@hooks.register('register_admin_urls')
def register_calendar_url():
    return [
        path('calendar/', index, name='calendar'),
    ]

@hooks.register('register_admin_menu_item')
def register_calendar_menu_item():
    return MenuItem('Calendar', reverse('calendar'), icon_name='date')
```

A ‘Calendar’ item will now appear in the menu.



Adding a group of menu items

Sometimes you want to group custom views together in a single menu item in the sidebar. Let's create another view to display only the current calendar month:

```
import calendar
from django.shortcuts import render
from django.utils import timezone

def index(request):
    current_year = timezone.now().year
    calendar_html = calendar.HTMLCalendar().formatyear(current_year)

    return render(request, 'wagtailcalendar/index.html', {
        'current_year': current_year,
        'calendar_html': calendar_html,
    })

def month(request):
    current_year = timezone.now().year
    current_month = timezone.now().month
    calendar_html = calendar.HTMLCalendar().formatmonth(current_year, current_month)

    return render(request, 'wagtailcalendar/index.html', {
        'current_year': current_year,
        'calendar_html': calendar_html,
    })
```

We also need to update `wagtail_hooks.py` to register our URL in the admin interface:

```
from django.urls import path
from wagtail import hooks

from .views import index, month

@hooks.register('register_admin_urls')
def register_calendar_url():
    return [
        path('calendar/', index, name='calendar'),
        path('calendar/month/', month, name='calendar-month'),
    ]
```

The calendar will now be visible at the URL `/admin/calendar/month/`.

Calendar

September 2022						
Mon	Tue	Wed	Thu	Fri	Sat	Sun
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

Finally we can alter our `wagtail_hooks.py` to include a group of custom menu items. This is similar to adding a single item but involves importing two more classes, `Menu` and `SubmenuItem`.

```
from django.urls import path, reverse

from wagtail.admin.menu import Menu, MenuItem, SubmenuItem
from wagtail import hooks

from .views import index, month

@hooks.register('register_admin_urls')
def register_calendar_url():
    return [
        path('calendar/', index, name='calendar'),
        path('calendar/month/', month, name='calendar-month'),
    ]

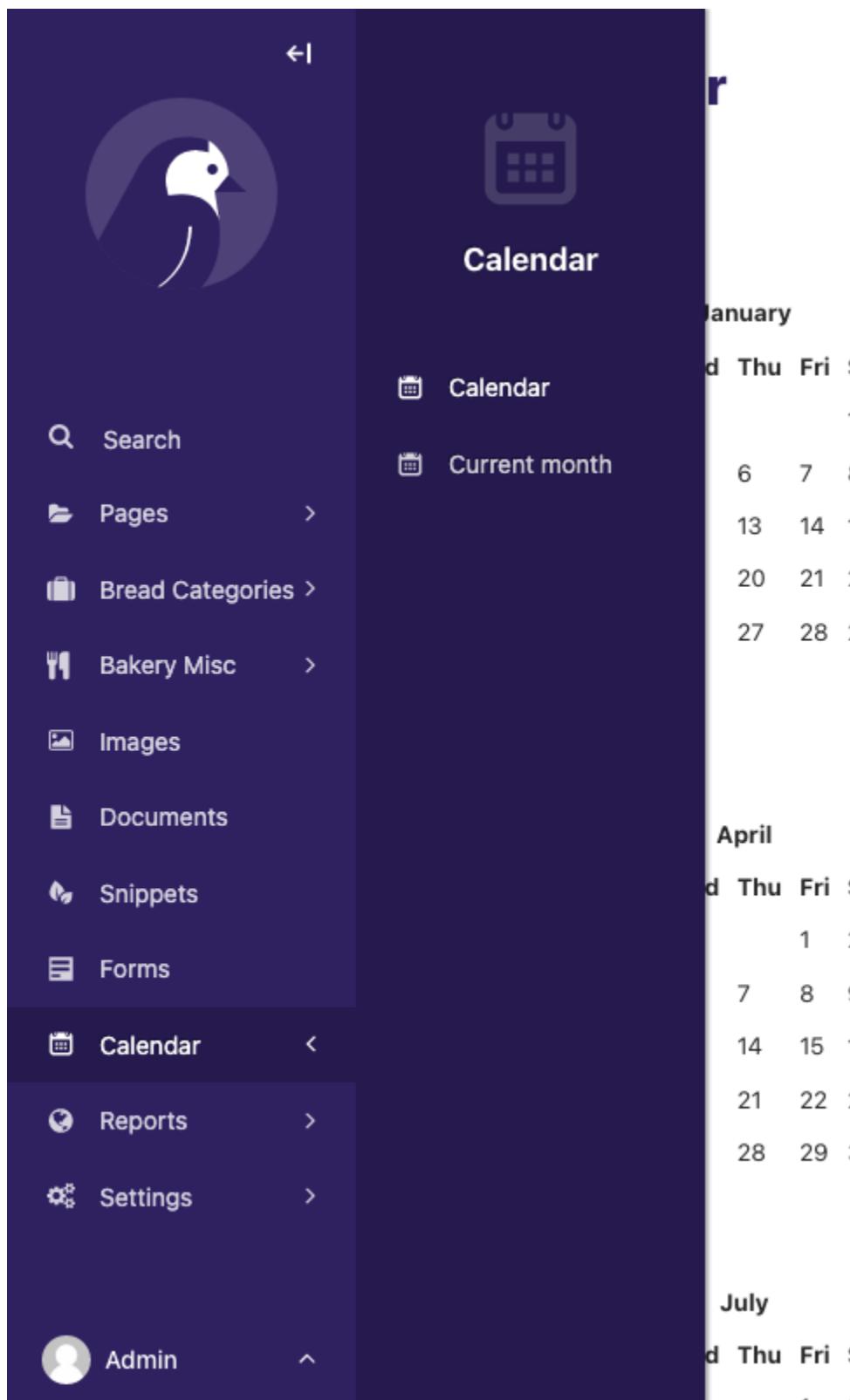
@hooks.register('register_admin_menu_item')
def register_calendar_menu_item():
    submenu = Menu(items=[
        MenuItem('Calendar', reverse('calendar'), icon_name='date'),
    ])
```

(continues on next page)

(continued from previous page)

```
MenuItem('Current month', reverse('calendar-month'), icon_name='date'),  
])  
  
return SubmenuItem('Calendar', submenu, icon_name='date')
```

The ‘Calendar’ item will now appear as a group of menu items. When expanded, the ‘Calendar’ item will now show our two custom menu items.



1.4.2 Generic views

Wagtail provides a number of generic views for handling common tasks such as creating / editing model instances, and chooser modals. Since these often involve several related views with shared properties (such as the model that we're working with, and its associated icon) Wagtail also implements the concept of a *viewset*, which allows a bundle of views to be defined collectively, and their URLs to be registered with the admin app as a single operation through the `register_admin_viewset` hook.

ModelViewSet

The `wagtail.admin.viewsets.model.ModelViewSet` class provides the views for listing, creating, editing and deleting model instances. For example, if we have the following model:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)

    def __str__(self):
        return "%s %s" % (self.first_name, self.last_name)
```

The following definition (to be placed in the same app's `views.py`) will generate a set of views for managing Person instances:

```
from wagtail.admin.viewsets.model import ModelViewSet
from .models import Person

class PersonViewSet(ModelViewSet):
    model = Person
    form_fields = ["first_name", "last_name"]
    icon = "user"

person_viewset = PersonViewSet("person") # defines /admin/person/ as the base URL
```

This viewset can then be registered with the Wagtail admin to make it available under the URL `/admin/person/`, by adding the following to `wagtail_hooks.py`:

```
from wagtail import hooks

from .views import person_viewset

@hooks.register("register_admin_viewset")
def register_viewset():
    return person_viewset
```

Various additional attributes are available to customise the viewset - see [Viewsets](#).

ChooserViewSet

The `wagtail.admin.viewsets.chooser.ChooserViewSet` class provides the views that make up a modal chooser interface, allowing users to select from a list of model instances to populate a ForeignKey field. Using the same Person model, the following definition (to be placed in `views.py`) will generate the views for a person chooser modal:

```
from wagtail.admin.viewsets.chooser import ChooserViewSet

class PersonChooserViewSet(ChooserViewSet):
    # The model can be specified as either the model class or an "app_label.model_name" string;
    # using a string avoids circular imports when accessing the StreamField block class (see below)
    model = "myapp.Person"

    icon = "user"
    choose_one_text = "Choose a person"
    choose_another_text = "Choose another person"
    edit_item_text = "Edit this person"
    form_fields = ["first_name", "last_name"] # fields to show in the "Create" tab

person_chooser_viewset = PersonChooserViewSet("person_chooser")
```

Again this can be registered with the `register_admin_viewset` hook:

```
from wagtail import hooks

from .views import person_chooser_viewset

@hooks.register("register_admin_viewset")
def register_viewset():
    return person_chooser_viewset
```

Registering a chooser viewset will also set up a chooser widget to be used whenever a ForeignKey field to that model appears in a `WagtailAdminModelForm` - see [Using forms in admin views](#). In particular, this means that a panel definition such as `FieldPanel("author")`, where `author` is a foreign key to the Person model, will automatically use this chooser interface. The chooser widget class can also be retrieved directly (for use in ordinary Django forms, for example) as the `widget_class` property on the viewset. For example, placing the following code in `widgets.py` will make the chooser widget available to be imported with `from myapp.widgets import PersonChooserWidget`:

```
from .views import person_chooser_viewset

PersonChooserWidget = person_chooser_viewset.widget_class
```

The viewset also makes a StreamField chooser block class available, through the method `get_block_class`. Placing the following code in `blocks.py` will make a chooser block available for use in StreamField definitions by importing `from myapp.blocks import PersonChooserBlock`:

```
from .views import person_chooser_viewset

PersonChooserBlock = person_chooser_viewset.get_block_class()
```

(continues on next page)

(continued from previous page)

```
    name="PersonChooserBlock", module_path="myapp.blocks"
)
```

Chooser viewsets for non-model datasources

While the generic chooser views are primarily designed to use Django models as the data source, choosers based on other sources such as REST API endpoints can be implemented by overriding the individual methods that deal with data retrieval.

Within `wagtail.admin.views.generic.chooser`:

- `BaseChooseView.get_object_list()` - returns a list of records to be displayed in the chooser. (In the default implementation, this is a Django QuerySet, and the records are model instances.)
- `BaseChooseView.columns` - a list of `wagtail.admin.ui.tables.Column` objects specifying the fields of the record to display in the final table
- `BaseChooseView.apply_object_list_ordering(objects)` - given a list of records as returned from `get_object_list`, returns the list with the desired ordering applied
- `ChosenViewMixin.get_object(pk)` - returns the record identified by the given primary key
- `ChosenResponseMixin.get_chosen_response_data(item)` - given a record, returns the dictionary of data that will be passed back to the chooser widget to populate it (consisting of items `id` and `title`, unless the chooser widget's JavaScript has been customised)

Within `wagtail.admin.widgets`:

- `BaseChooser.get_instance(value)` - given a value that may be a record, a primary key or None, returns the corresponding record or None
- `BaseChooser.get_value_data_from_instance(item)` - given a record, returns the dictionary of data that will populate the chooser widget (consisting of items `id` and `title`, unless the widget's JavaScript has been customised)

For example, the following code will implement a chooser that runs against a JSON endpoint for the User model at `http://localhost:8000/api/users/`, set up with Django REST Framework using the default configuration and no pagination:

```
from django.views.generic.base import View
import requests

from wagtail.admin.ui.tables import Column, TitleColumn
from wagtail.admin.views.generic.chooser import (
    BaseChooseView, ChooseViewMixin, ChooseResultsViewMixin, ChosenResponseMixin,
    ChosenViewMixin, CreationFormMixin
)
from wagtail.admin.viewsets.chooser import ChooserViewSet
from wagtail.admin.widgets import BaseChooser


class BaseUserChooseView(BaseChooseView):
    @property
    def columns(self):
        return [
            TitleColumn(
```

(continues on next page)

(continued from previous page)

```

        "title",
        label="Title",
        accessor='username',
        id_accessor='id',
        url_name=self.chosen_url_name,
        link_attrs={"data-chooser-modal-choice": True},
    ),
    Column(
        "email", label="Email", accessor="email"
    )
]
]

def get_object_list(self):
    r = requests.get("http://localhost:8000/api/users/")
    r.raise_for_status()
    results = r.json()
    return results

def apply_object_list_ordering(self, objects):
    return objects

class UserChooseView(ChooseViewMixin, CreationFormMixin, BaseUserChooseView):
    pass

class UserChooseResultsView(ChooseResultsViewMixin, CreationFormMixin, BaseUserChooseView):
    pass

class UserChosenViewMixin(ChosenViewMixin):
    def get_object(self, pk):
        r = requests.get("http://localhost:8000/api/users/%d/" % int(pk))
        r.raise_for_status()
        return r.json()

class UserChosenResponseMixin(ChosenResponseMixin):
    def get_chosen_response_data(self, item):
        return {
            "id": item["id"],
            "title": item["username"],
        }

class UserChosenView(UserChosenViewMixin, UserChosenResponseMixin, View):
    pass

class BaseUserChooserWidget(BaseChooser):
    def get_instance(self, value):

```

(continues on next page)

(continued from previous page)

```

if value is None:
    return None
elif isinstance(value, dict):
    return value
else:
    r = requests.get("http://localhost:8000/api/users/%d/" % int(value))
    r.raise_for_status()
    return r.json()

def get_value_data_from_instance(self, instance):
    return {
        "id": instance["id"],
        "title": instance["username"],
    }

class UserChooserViewSet(ChooserViewSet):
    icon = "user"
    choose_one_text = "Choose a user"
    choose_another_text = "Choose another user"
    edit_item_text = "Edit this user"

    choose_view_class = UserChooseView
    choose_results_view_class = UserChooseResultsView
    chosen_view_class = UserChosenView
    base_widget_class = BaseUserChooserWidget

user_chooser_viewset = UserChooserViewSet("user_chooser", url_prefix="user-chooser")

```

If the data source implements its own pagination - meaning that the pagination mechanism built into the chooser should be bypassed - the `BaseChooseView.get_results_page(request)` method can be overridden instead of `get_object_list`. This should return an instance of `django.core.paginator.Page`. For example, if the API in the above example followed the conventions of the Wagtail API, implementing pagination with `offset` and `limit` URL parameters and returning a dict consisting of `meta` and `results`, the `BaseUserChooseView` implementation could be modified as follows:

```

from django.core.paginator import Page, Paginator

class APIPaginator(Paginator):
    """
    Customisation of Django's Paginator class for use when we don't want it to handle
    slicing on the result set, but still want it to generate the page numbering based
    on a known result count.
    """
    def __init__(self, count, per_page, **kwargs):
        self._count = int(count)
        super().__init__([], per_page, **kwargs)

    @property
    def count(self):
        return self._count

```

(continues on next page)

(continued from previous page)

```

class BaseUserChooseView(BaseChooseView):
    @property
    def columns(self):
        return [
            TitleColumn(
                "title",
                label="Title",
                accessor='username',
                id_accessor='id',
                url_name=self.chosen_url_name,
                link_attrs={"data-chooser-modal-choice": True},
            ),
            Column(
                "email", label="Email", accessor="email"
            )
        ]
    def get_results_page(self, request):
        try:
            page_number = int(request.GET.get('p', 1))
        except ValueError:
            page_number = 1

        r = requests.get("http://localhost:8000/api/users/", params={
            'offset': (page_number - 1) * self.per_page,
            'limit': self.per_page,
        })
        r.raise_for_status()
        result = r.json()
        paginator = APIPaginator(result['meta']['total_count'], self.per_page)
        page = Page(result['items'], page_number, paginator)
        return page

```

1.4.3 Template components

Working with objects that know how to render themselves as elements on an HTML template is a common pattern seen throughout the Wagtail admin. For example, the admin homepage is a view provided by the central `wagtail.admin` app, but brings together information panels sourced from various other modules of Wagtail, such as images and documents (potentially along with others provided by third-party packages). These panels are passed to the homepage via the `construct_homepage_panels` hook, and each one is responsible for providing its own HTML rendering. In this way, the module providing the panel has full control over how it appears on the homepage.

Wagtail implements this pattern using a standard object type known as a **component**. A component is a Python object that provides the following methods and properties:

`render_html(self, parent_context=None)`

Given a context dictionary from the calling template (which may be a `Context` object or a plain `dict` of context variables), returns the string representation to be inserted into the template. This will be subject to Django's HTML escaping rules, so a return value consisting of HTML should typically be returned as a `SafeString` instance.

`media`

A (possibly empty) `form.media` object defining JavaScript and CSS resources used by the component.

Note: Any object implementing this API can be considered a valid component; it does not necessarily have to inherit from the `Component` class described below, and user code that works with components should not assume this (for example, it must not use `isinstance` to check whether a given value is a component).

Creating components

The preferred way to create a component is to define a subclass of `wagtail.admin.ui.components.Component` and specify a `template_name` attribute on it. The rendered template will then be used as the component's HTML representation:

```
from wagtail.admin.ui.components import Component

class WelcomePanel(Component):
    template_name = 'my_app/panels/welcome.html'

my_welcome_panel = WelcomePanel()
```

`my_app/templates/my_app/panels/welcome.html`:

```
<h1>Welcome to my app!</h1>
```

For simple cases that don't require a template, the `render_html` method can be overridden instead:

```
from django.utils.html import format_html
from wagtail.admin.components import Component

class WelcomePanel(Component):
    def render_html(self, parent_context):
        return format_html("<h1>{}</h1>", "Welcome to my app!")
```

Passing context to the template

The `get_context_data` method can be overridden to pass context variables to the template. As with `render_html`, this receives the context dictionary from the calling template:

```
from wagtail.admin.ui.components import Component

class WelcomePanel(Component):
    template_name = 'my_app/panels/welcome.html'

    def get_context_data(self, parent_context):
        context = super().get_context_data(parent_context)
        context['username'] = parent_context['request'].user.username
        return context
```

`my_app/templates/my_app/panels/welcome.html`:

```
<h1>Welcome to my app, {{ username }}!</h1>
```

Adding media definitions

Like Django form widgets, components can specify associated JavaScript and CSS resources using either an inner `Media` class or a dynamic `media` property:

```
class WelcomePanel(Component):
    template_name = 'my_app/panels/welcome.html'

    class Media:
        css = {
            'all': ('my_app/css/welcome-panel.css',)
        }
```

Using components on your own templates

The `wagtailadmin_tags` tag library provides a `{% component %}` tag for including components on a template. This takes care of passing context variables from the calling template to the component (which would not be the case for a basic `{{ ... }}` variable tag). For example, given the view:

```
from django.shortcuts import render

def welcome_page(request):
    panels = [
        WelcomePanel(),
    ]

    render(request, 'my_app/welcome.html', {
        'panels': panels,
    })
```

the `my_app/welcome.html` template could render the panels as follows:

```
{% load wagtailadmin_tags %}
{% for panel in panels %}
    {% component panel %}
{% endfor %}
```

Note that it is your template's responsibility to output any media declarations defined on the components. For a Wagtail admin view, this is best done by constructing a `Media` object for the whole page within the view, passing this to the template, and outputting it via the base template's `extra_js` and `extra_css` blocks:

```
from django.forms import Media
from django.shortcuts import render

def welcome_page(request):
    panels = [
        WelcomePanel(),
    ]
```

(continues on next page)

(continued from previous page)

```
media = Media()
for panel in panels:
    media += panel.media

render(request, 'my_app/welcome.html', {
    'panels': panels,
    'media': media,
})
```

my_app/welcome.html:

```
{% extends "wagtailadmin/base.html" %}
{% load wagtailadmin_tags %}

{% block extra_js %}
    {{ block.super }}
    {{ media.js }}
{% endblock %}

{% block extra_css %}
    {{ block.super }}
    {{ media.css }}
{% endblock %}

{% block content %}
    {% for panel in panels %}
        {% component panel %}
    {% endfor %}
{% endblock %}
```

1.4.4 Using forms in admin views

Django's forms framework can be used within Wagtail admin views just like in any other Django app. However, Wagtail also provides various admin-specific form widgets, such as date/time pickers and choosers for pages, documents, images and snippets. By constructing forms using `wagtail.admin.forms.models.WagtailAdminModelForm` as the base class instead of `django.forms.models.ModelForm`, the most appropriate widget will be selected for each model field. For example, given the model and form definition:

```
from django.db import models

from wagtail.admin.forms.models import WagtailAdminModelForm
from wagtail.images.models import Image


class FeaturedImage(models.Model):
    date = models.DateField()
    image = models.ForeignKey(Image, on_delete=models.CASCADE)

    class Meta:
        pass
```

(continues on next page)

(continued from previous page)

```
model = FeaturedImage
```

the date and image fields on the form will use a date picker and image chooser widget respectively.

Defining admin form widgets

If you have implemented a form widget of your own, you can configure `WagtailAdminModelForm` to select it for a given model field type. This is done by calling the `wagtail.admin.forms.models.register_form_field_override` function, typically in an `AppConfig.ready` method.

```
register_form_field_override(model_field_class, to=None, override=None, exact_class=False)
```

Specify a set of options that will override the form field's defaults when `WagtailAdminModelForm` encounters a given model field type.

Parameters

- **`model_field_class`** – Specifies a model field class, such as `models.CharField`; the override will take effect on fields that are instances of this class.
- **`to`** – For `ForeignKey` fields, indicates the model that the field must point to for the override to take effect.
- **`override`** – A dict of keyword arguments to be passed to the form field's `__init__` method, such as `widget`.
- **`exact_class`** – If true, the override will only take effect for model fields that are of the exact type given by `model_field_class`, and not a subclass of it.

For example, if the app `wagtail.videos` implements a `Video` model and a `VideoChooser` form widget, the following `AppConfig` definition will ensure that `WagtailAdminModelForm` selects `VideoChooser` as the form widget for any foreign keys pointing to `Video`:

```
from django.apps import AppConfig
from django.db.models import ForeignKey

class WagtailVideosAppConfig(AppConfig):
    name = 'wagtail.videos'
    label = 'wagtailvideos'

    def ready(self):
        from wagtail.admin.forms.models import register_form_field_override
        from .models import Video
        from .widgets import VideoChooser
        register_form_field_override(ForeignKey, to=Video, override={'widget': VideoChooser})
```

Wagtail's edit views for pages, snippets and `ModelAdmin` use `WagtailAdminModelForm` as standard, so this change will take effect across the Wagtail admin; a foreign key to `Video` on a page model will automatically use the `VideoChooser` widget, with no need to specify this explicitly.

Panels

Panels (also known as edit handlers until Wagtail 3.0) are Wagtail's mechanism for specifying the content and layout of a model form without having to write a template. They are used for the editing interface for pages and snippets, as well as the [ModelAdmin](#) and [site settings](#) contrib modules.

See [Panel types](#) for the set of panel types provided by Wagtail. All panels inherit from the base class `wagtail.admin.panels.Panel`. A single panel object (usually `ObjectList` or `TabbedInterface`) exists at the top level and is the only one directly accessed by the view code; panels containing child panels inherit from the base class `wagtail.admin.panels.PanelGroup` and take care of recursively calling methods on their child panels where appropriate.

A view performs the following steps to render a model form through the panels mechanism:

- The top-level panel object for the model is retrieved. Usually this is done by looking up the model's `edit_handler` property and falling back on an `ObjectList` consisting of children given by the model's `panels` property. However, it may come from elsewhere - for example, the `ModelAdmin` module allows defining it on the `ModelAdmin` configuration object.
- The view calls `bind_to_model` on the top-level panel, passing the model class, and this returns a clone of the panel with a `model` property. As part of this process the `on_model_bound` method is invoked on each child panel, to allow it to perform additional initialisation that requires access to the model (for example, this is where `FieldPanel` retrieves the model field definition).
- The view then calls `get_form_class` on the top-level panel to retrieve a `ModelForm` subclass that can be used to edit the model. This proceeds as follows:
 - Retrieve a base form class from the model's `base_form_class` property, falling back on `wagtail.admin.forms.WagtailAdminModelForm`
 - Call `get_form_options` on each child panel - which returns a dictionary of properties including `fields` and `widgets` - and merge the results into a single dictionary
 - Construct a subclass of the base form class, with the options dict forming the attributes of the inner `Meta` class.
- An instance of the form class is created as per a normal Django form view.
- The view then calls `get_bound_panel` on the top-level panel, passing `instance`, `form` and `request` as keyword arguments. This returns a `BoundPanel` object, which follows [the template component API](#). Finally, the `BoundPanel` object (and its media definition) is rendered onto the template.

New panel types can be defined by subclassing `wagtail.admin.panels.Panel` - see [Panel API](#).

1.4.5 Adding reports

Reports are views with listings of pages matching a specific query. They can also export these listings in spreadsheet format. They are found in the `Reports` submenu: by default, the `Locked pages` report is provided, allowing an overview of locked pages on the site.

It is possible to create your own custom reports in the Wagtail admin. Two base classes are provided: `wagtail.admin.views.reports.ReportView`, which provides basic listing and spreadsheet export functionality, and `wagtail.admin.views.reports.PageReportView`, which additionally provides a default set of fields suitable for page listings. For this example, we'll add a report which shows any pages with unpublished changes.

```
# <project>/views.py
from wagtail.admin.views.reports import PageReportView
```

(continues on next page)

(continued from previous page)

```
class UnpublishedChangesReportView(PageReportView):
    pass
```

Defining your report

The most important attributes and methods to customise to define your report are:

get_queryset(self)

This retrieves the queryset of pages for your report. For our example:

```
# <project>/views.py

from wagtail.admin.views.reports import PageReportView
from wagtail.models import Page

class UnpublishedChangesReportView(PageReportView):

    def get_queryset(self):
        return Page.objects.filter(has_unpublished_changes=True)
```

template_name

(string)

The template used to render your report. For ReportView, this defaults to "wagtailadmin/reports/base_report.html", which provides an empty report page layout; for PageReportView, this defaults to "wagtailadmin/reports/base_page_report.html" which provides a listing based on the explorer views, displaying action buttons, as well as the title, time of the last update, status, and specific type of any pages. In this example, we'll change this to a new template in a later section.

title

(string)

The name of your report, which will be displayed in the header. For our example, we'll set it to "Pages with unpublished changes".

header_icon

(string)

The name of the icon, using the standard Wagtail icon names. For example, the locked pages view uses "locked", and for our example report, we'll set it to 'doc-empty-inverse'.

Spreadsheet exports

list_export

(list)

A list of the fields/attributes for each model which are exported as columns in the spreadsheet view. For `ReportView`, this is empty by default, and for `PageReportView`, it corresponds to the listing fields: the title, time of the last update, status, and specific type of any pages. For our example, we might want to know when the page was last published, so we'll set `list_export` as follows:

```
list_export = PageReportView.list_export + ['last_published_at']
```

export_headings

(dictionary)

A dictionary of any fields/attributes in `list_export` for which you wish to manually specify a heading for the spreadsheet column, and their headings. If unspecified, the heading will be taken from the field `verbose_name` if applicable, and the attribute string otherwise. For our example, `last_published_at` will automatically get a heading of "Last Published At", but a simple "Last Published" looks neater. We'll add that by setting `export_headings`:

```
export_headings = dict(last_published_at='Last Published', **PageReportView.export_headings)
```

custom_value_preprocess

(dictionary)

A dictionary of `(value_class_1, value_class_2, ...)` tuples mapping to `{export_format: preprocessing_function}` dictionaries, allowing custom preprocessing functions to be applied when exporting field values of specific classes (or their subclasses). If unspecified (and `ReportView.custom_field_preprocess` also does not specify a function), `force_str` will be used. To prevent preprocessing, set the `preprocessing_function` to `None`.

custom_field_preprocess

(dictionary)

A dictionary of `field_name` strings mapping to `{export_format: preprocessing_function}` dictionaries, allowing custom preprocessing functions to be applied when exporting field values of specific classes (or their subclasses). This will take priority over functions specified in `ReportView.custom_value_preprocess`. If unspecified (and `ReportView.custom_value_preprocess` also does not specify a function), `force_str` will be used. To prevent preprocessing, set the `preprocessing_function` to `None`.

Customising templates

For this example “pages with unpublished changes” report, we’ll add an extra column to the listing template, showing the last publication date for each page. To do this, we’ll extend two templates: `wagtailadmin/reports/base_page_report.html`, and `wagtailadmin/reports/listing/_list_page_report.html`.

```
{# <project>/templates/reports/unpublished_changes_report.html #}

{% extends 'wagtailadmin/reports/base_page_report.html' %}

{% block listing %}
    {% include 'reports/include/_list_unpublished_changes.html' %}
{% endblock %}
```

(continues on next page)

(continued from previous page)

```
{% block no_results %}
    <p>No pages with unpublished changes.</p>
{% endblock %}

{# <project>/templates/reports/include/_list_unpublished_changes.html #}

{% extends 'wagtailadmin/reports/listing/_list_page_report.html' %}

{% block extra_columns %}
    <th>Last Published</th>
{% endblock %}

{% block extra_page_data %}
    <td valign="top">
        {{ page.last_published_at }}
    </td>
{% endblock %}
```

Finally, we'll set `UnpublishedChangesReportView.template_name` to this new template: '`reports/unpublished_changes_report.html`'.

Adding a menu item and admin URL

To add a menu item for your new report to the `Reports` submenu, you will need to use the `register_reports_menu_item` hook (see: [Register Reports Menu Item](#)). To add an admin url for the report, you will need to use the `register_admin_urls` hook (see: [Register Admin URLs](#)). This can be done as follows:

```
# <project>/wagtail_hooks.py

from django.urls import path, reverse

from wagtail.admin.menu import AdminOnlyMenuItem
from wagtail import hooks

from .views import UnpublishedChangesReportView

@hooks.register('register_reports_menu_item')
def register_unpublished_changes_report_menu_item():
    return AdminOnlyMenuItem("Pages with unpublished changes", reverse('unpublished_changes_report'), classnames='icon icon-' + UnpublishedChangesReportView.header_icon, order=700)

@hooks.register('register_admin_urls')
def register_unpublished_changes_report_url():
    return [
        path('reports/unpublished-changes/', UnpublishedChangesReportView.as_view(), name='unpublished_changes_report'),
    ]
```

Here, we use the `AdminOnlyMenuItem` class to ensure our report icon is only shown to superusers. To make the report visible to all users, you could replace this with `MenuItem`.

The full code

```
# <project>/views.py

from wagtail.admin.views.reports import PageReportView
from wagtail.models import Page

class UnpublishedChangesReportView(PageReportView):

    header_icon = 'doc-empty-inverse'
    template_name = 'reports/unpublished_changes_report.html'
    title = "Pages with unpublished changes"

    list_export = PageReportView.list_export + ['last_published_at']
    export_headings = dict(last_published_at='Last Published', **PageReportView.export_headings)

    def get_queryset(self):
        return Page.objects.filter(has_unpublished_changes=True)
```

```
# <project>/wagtail_hooks.py

from django.urls import path, reverse

from wagtail.admin.menu import AdminOnlyMenuItem
from wagtail import hooks

from .views import UnpublishedChangesReportView

@hooks.register('register_reports_menu_item')
def register_unpublished_changes_report_menu_item():
    return AdminOnlyMenuItem("Pages with unpublished changes", reverse('unpublished_changes_report'), classnames='icon icon-' + UnpublishedChangesReportView.header_icon, order=700)

@hooks.register('register_admin_urls')
def register_unpublished_changes_report_url():
    return [
        path('reports/unpublished-changes/', UnpublishedChangesReportView.as_view(), name='unpublished_changes_report'),
    ]
```

```
{# <project>/templates/reports/unpublished_changes_report.html #}

{% extends 'wagtailadmin/reports/base_page_report.html' %}

{% block listing %}
    {% include 'reports/include/_list_unpublished_changes.html' %}
{% endblock %}

{% block no_results %}
```

(continues on next page)

(continued from previous page)

```
<p>No pages with unpublished changes.</p>
{% endblock %}
```

```
{# <project>/templates/reports/include/_list_unpublished_changes.html #-}

{% extends 'wagtailadmin/reports/listing/_list_page_report.html' %}

{% block extra_columns %}
    <th>Last Published</th>
{% endblock %}

{% block extra_page_data %}
    <td valign="top">
        {{ page.last_published_at }}
    </td>
{% endblock %}
```

1.4.6 Adding new Task types

The Workflow system allows users to create tasks, which represent stages of moderation.

Wagtail provides one built in task type: `GroupApprovalTask`, which allows any user in specific groups to approve or reject moderation.

However, it is possible to add your own task types in code. Instances of your custom task can then be created in the Tasks section of the Wagtail Admin.

Task models

All custom tasks must be models inheriting from `wagtailcore.Task`. In this set of examples, we'll set up a task which can be approved by only one specific user.

```
# <project>/models.py

from wagtail.models import Task

class UserApprovalTask(Task):
    pass
```

Subclassed Tasks follow the same approach as Pages: they are concrete models, with the specific subclass instance accessible by calling `Task.specific()`.

You can now add any custom fields. To make these editable in the admin, add the names of the fields into the `admin_form_fields` attribute:

For example:

```
# <project>/models.py

from django.conf import settings
from django.db import models
```

(continues on next page)

(continued from previous page)

```
from wagtail.models import Task

class UserApprovalTask(Task):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.SET_NULL, u
    ↪null=True, blank=False)

    admin_form_fields = Task.admin_form_fields + ['user']
```

Any fields that shouldn't be edited after task creation - for example, anything that would fundamentally change the meaning of the task in any history logs - can be added to `admin_form_READONLY_on_edit_fields`. For example:

```
# <project>/models.py

from django.conf import settings
from django.db import models
from wagtail.models import Task

class UserApprovalTask(Task):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.SET_NULL, u
    ↪null=True, blank=False)

    admin_form_fields = Task.admin_form_fields + ['user']

    # prevent editing of `user` after the task is created
    # by default, this attribute contains the 'name' field to prevent tasks from being u
    ↪renamed
    admin_form_READONLY_on_edit_fields = Task.admin_form_READONLY_on_edit_fields + ['user
    ↪']
```

Wagtail will choose a default form widget to use based on the field type. But you can override the form widget using the `admin_form_widgets` attribute:

```
# <project>/models.py

from django.conf import settings
from django.db import models
from wagtail.models import Task

from .widgets import CustomUserChooserWidget

class UserApprovalTask(Task):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.SET_NULL, u
    ↪null=True, blank=False)

    admin_form_fields = Task.admin_form_fields + ['user']

    admin_form_widgets = {
        'user': CustomUserChooserWidget,
    }
```

Custom TaskState models

You might also need to store custom state information for the task: for example, a rating left by an approving user. Normally, this is done on an instance of `TaskState`, which is created when a page starts the task. However, this can also be subclassed equivalently to `Task`:

```
# <project>/models.py

from wagtail.models import TaskState

class UserApprovalTaskState(TaskState):
    pass
```

Your custom task must then be instructed to generate an instance of your custom task state on start instead of a plain `TaskState` instance:

```
# <project>/models.py

from django.conf import settings
from django.db import models
from wagtail.models import Task, TaskState

class UserApprovalTaskState(TaskState):
    pass

class UserApprovalTask(Task):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.SET_NULL, null=True, blank=False)

    admin_form_fields = Task.admin_form_fields + ['user']

    task_state_class = UserApprovalTaskState
```

Customising behaviour

Both `Task` and `TaskState` have a number of methods which can be overridden to implement custom behaviour. Here are some of the most useful:

<code>Task.user_can_access_editor(page, user)</code>	<code>Task.user_can_lock(page, user)</code>	<code>Task.user_can_unlock(page, user)</code>
--	---	---

These methods determine if users usually without permissions can access the editor, lock, or unlock the page, by returning `True` or `False`. Note that returning `False` will not prevent users who would normally be able to perform those actions. For example, for our `UserApprovalTask`:

```
def user_can_access_editor(self, page, user):
    return user == self.user
```

`Task.page_locked_for_user(page, user)`:

This returns `True` if the page should be locked and uneditable by the user. It is used by `GroupApprovalTask` to lock the page to any users not in the approval group.

```
def page_locked_for_user(self, page, user):
    return user != self.user
```

Task.get_actions(page, user):

This returns a list of (action_name, action_verbose_name, action_requires_additional_data_from_modal) tuples, corresponding to the actions available for the task in the edit view menu. action_requires_additional_data_from_modal should be a boolean, returning True if choosing the action should open a modal for additional data input - for example, entering a comment.

For example:

```
def get_actions(self, page, user):
    if user == self.user:
        return [
            ('approve', "Approve", False),
            ('reject', "Reject", False),
            ('cancel', "Cancel", False),
        ]
    else:
        return []
```

Task.get_form_for_action(action):

Returns a form to be used for additional data input for the given action modal. By default, returns TaskStateCommentForm, with a single comment field. The form data returned in form.cleaned_data must be fully serializable as JSON.

Task.get_template_for_action(action):

Returns the name of a custom template to be used in rendering the data entry modal for that action.

Task.on_action(task_state, user, action_name, **kwargs):

This performs the actions specified in Task.get_actions(page, user): it is passed an action name, for example approve, and the relevant task state. By default, it calls approve and reject methods on the task state when the corresponding action names are passed through. Any additional data entered in a modal (see get_form_for_action and get_actions) is supplied as kwargs.

For example, let's say we wanted to add an additional option: cancelling the entire workflow:

```
def on_action(self, task_state, user, action_name):
    if action_name == 'cancel':
        return task_state.workflow_state.cancel(user=user)
    else:
        return super().on_action(task_state, user, workflow_state)
```

Task.get_task_states_user_can_moderate(user, **kwargs):

This returns a QuerySet of TaskStates (or subclasses) the given user can moderate - this is currently used to select pages to display on the user's dashboard.

For example:

```
def get_task_states_user_can_moderate(self, user, **kwargs):
    if user == self.user:
        # get all task states linked to the (base class of) current task
        return TaskState.objects.filter(status=TaskState.STATUS_IN_PROGRESS, task=self.
        ↵task_ptr)
```

(continues on next page)

(continued from previous page)

```
else:
    return TaskState.objects.none()
```

`Task.get_description()`

A class method that returns the human-readable description for the task.

For example:

```
@classmethod
def get_description(cls):
    return _("Members of the chosen Wagtail Groups can approve this task")
```

Adding notifications

Wagtail's notifications are sent by `wagtail.admin.mail.Notifier` subclasses: callables intended to be connected to a signal.

By default, email notifications are sent upon workflow submission, approval and rejection, and upon submission to a group approval task.

As an example, we'll add email notifications for when our new task is started.

```
# <project>/mail.py

from wagtail.admin.mail import EmailNotificationMixin, Notifier
from wagtail.models import TaskState

from .models import UserApprovalTaskState

class BaseUserApprovalTaskStateEmailNotifier(EmailNotificationMixin, Notifier):
    """A base notifier to send updates for UserApprovalTask events"""

    def __init__(self):
        # Allow UserApprovalTaskState and TaskState to send notifications
        super().__init__((UserApprovalTaskState, TaskState))

    def can_handle(self, instance, **kwargs):
        if super().can_handle(instance, **kwargs) and isinstance(instance.task_specific, UserApprovalTask):
            # Don't send notifications if a Task has been cancelled and then resumed - when page was updated to a new revision
            return not TaskState.objects.filter(workflow_state=instance.workflow_state, task=instance.task, status=TaskState.STATUS_CANCELLED).exists()
        return False

    def get_context(self, task_state, **kwargs):
        context = super().get_context(task_state, **kwargs)
        context['page'] = task_state.workflow_state.page
        context['task'] = task_state.task_specific
        return context
```

(continues on next page)

(continued from previous page)

```
def get_recipient_users(self, task_state, **kwargs):
    # Send emails to the user assigned to the task
    approving_user = task_state.task.specific.user

    recipients = {approving_user}

    return recipients

class UserApprovalTaskStateSubmissionEmailNotifier(BaseUserApprovalTaskStateEmailNotifier):
    """A notifier to send updates for UserApprovalTask submission events"""

    notification = 'submitted'
```

Similarly, you could define notifier subclasses for approval and rejection notifications.

Next, you need to instantiate the notifier, and connect it to the `task_submitted` signal.

```
# <project>/signal_handlers.py

from wagtail.signals import task_submitted
from .mail import UserApprovalTaskStateSubmissionEmailNotifier

task_submission_email_notifier = UserApprovalTaskStateSubmissionEmailNotifier()

def register_signal_handlers():
    task_submitted.connect(user_approval_task_submission_email_notifier, dispatch_uid=
    'user_approval_task_submitted_email_notification')
```

`register_signal_handlers()` should then be run on loading the app: for example, by adding it to the `ready()` method in your `AppConfig`.

```
# <project>/apps.py
from django.apps import AppConfig

class My AppConfig(AppConfig):
    name = 'myappname'
    label = 'myapplabel'
    verbose_name = 'My verbose app name'

    def ready(self):
        from .signal_handlers import register_signal_handlers
        register_signal_handlers()
```

Note: In Django versions before 3.2 your `AppConfig` subclass needs to be set as `default_app_config` in `<project>/__init__.py`. See the relevant section in the [Django docs](#) for the version you are using.

1.4.7 Audit log

Wagtail provides a mechanism to log actions performed on its objects. Common activities such as page creation, update, deletion, locking and unlocking, revision scheduling and privacy changes are automatically logged at the model level.

The Wagtail admin uses the action log entries to provide a site-wide and page specific history of changes. It uses a registry of ‘actions’ that provide additional context for the logged action.

The audit log-driven Page history replaces the revisions list page, but provide a filter for revision-specific entries.

Note: The audit log does not replace revisions.

The `wagtail.log_actions.log` function can be used to add logging to your own code.

`log(instance, action, user=None, uuid=None, title=None, data=None)`

Adds an entry to the audit log.

Parameters

- **instance** – The model instance that the action is performed on
- **action** – The code name for the action being performed. This can be one of the names listed below, or a custom action defined through the `register_log_actions` hook.
- **user** – Optional - the user initiating the action. For actions logged within an admin view, this defaults to the logged-in user.
- **uuid** – Optional - log entries given the same UUID indicates that they occurred as part of the same user action (for example a page being immediately published on creation).
- **title** – The string representation of the instance being logged. By default, Wagtail will attempt to use the instance’s `str` representation, or `get_admin_display_title` for page objects.
- **data** – Optional - a dictionary of additional JSON-serialisable data to store against the log entry

Note: When adding logging, you need to log the action or actions that happen to the object. For example, if the user creates and publishes a page, there should be a “create” entry and a “publish” entry. Or, if the user copies a published page and chooses to keep it published, there should be a “copy” and a “publish” entry for new page.

```
# mypackage/views.py
from wagtail.log_actions import log

def copy_for_translation(page):
    #
    page.copy(log_action='mypackage.copy_for_translation')

def my_method(request, page):
    #
    # Manually log an action
    data = {
        'make': {'it': 'so'}
    }
    log(
```

(continues on next page)

(continued from previous page)

```
instance=page, action='mypackage.custom_action', data=data
    )
```

Changed in version 2.15: The `log` function was added. Previously, logging was only implemented for pages, and invoked through the `PageLogEntry.objects.log_action` method.

Log actions provided by Wagtail

Action	Notes
<code>wagtail.create</code>	The object was created
<code>wagtail.edit</code>	The object was edited (for pages, saved as draft)
<code>wagtail.delete</code>	The object was deleted. Will only surface in the Site History for administrators
<code>wagtail.publish</code>	The page was published
<code>wagtail.publish.schedule</code>	Draft is scheduled for publishing
<code>wagtail.publish.scheduled</code>	Draft published via <code>publish_scheduled_pages</code> management command
<code>wagtail.schedule.cancel</code>	Draft scheduled for publishing cancelled via “Cancel scheduled publish”
<code>wagtail.unpublish</code>	The page was unpublished
<code>wagtail.unpublish.scheduled</code>	Page unpublished via <code>publish_scheduled_pages</code> management command
<code>wagtail.lock</code>	Page was locked
<code>wagtail.unlock</code>	Page was unlocked
<code>wagtail.moderation.approve</code>	The revision was approved for moderation
<code>wagtail.moderation.reject</code>	The revision was rejected
<code>wagtail.rename</code>	A page was renamed
<code>wagtail.revert</code>	The page was reverted to a previous draft
<code>wagtail.copy</code>	The page was copied to a new location
<code>wagtail.copy_for_translation</code>	The page was copied into a new locale for translation
<code>wagtail.move</code>	The page was moved to a new location
<code>wagtail.reorder</code>	The order of the page under its parent was changed
<code>wagtail.view_restriction.create</code>	The page was restricted
<code>wagtail.view_restriction.edit</code>	The page restrictions were updated
<code>wagtail.view_restriction.delete</code>	The page restrictions were removed
<code>wagtail.workflow.start</code>	The page was submitted for moderation in a Workflow
<code>wagtail.workflow.approve</code>	The draft was approved at a Workflow Task
<code>wagtail.workflow.reject</code>	The draft was rejected, and changes requested at a Workflow Task
<code>wagtail.workflow.resume</code>	The draft was resubmitted to the workflow
<code>wagtail.workflow.cancel</code>	The workflow was cancelled

Log context

The `wagtail.log_actions` module provides a context manager to simplify code that logs a large number of actions, such as import scripts:

```
from wagtail.log_actions import LogContext

with LogContext(user=User.objects.get(username='admin')):
    # ...
    log(page, 'wagtail.edit')
    # ...
    log(page, 'wagtail.publish')
```

All `log` calls within the block will then be attributed to the specified user, and assigned a common UUID. A log context is created automatically for views within the Wagtail admin.

Log models

Logs are stored in the database via the models `wagtail.models.PageLogEntry` (for actions on Page instances) and `wagtail.models.ModelLogEntry` (for actions on all other models). Page logs are stored in their own model to ensure that reports can be filtered according to the current user's permissions, which could not be done efficiently with a generic foreign key.

If your own models have complex reporting requirements that would make `ModelLogEntry` unsuitable, you can configure them to be logged to their own log model; this is done by subclassing the abstract `wagtail.models.BaseLogEntry` model, and registering that model with the log registry's `register_model` method:

```
from myapp.models import Sprocket, SprocketLogEntry
# here SprocketLogEntry is a subclass of BaseLogEntry

@hooks.register('register_log_actions')
def sprocket_log_model(actions):
    actions.register_model(Sprocket, SprocketLogEntry)
```

1.4.8 Customising the user account settings form

This document describes how to customise the user account settings form that can be found by clicking “Account settings” at the bottom of the main menu.

Adding new panels

Each panel on this form is a separate model form which can operate on an instance of either the user model, or the `wagtail.users.models.UserProfile`.

Basic example

Here is an example of how to add a new form that operates on the user model:

```
# forms.py

from django import forms
from django.contrib.auth import get_user_model

class CustomSettingsForm(forms.ModelForm):

    class Meta:
        model = get_user_model()
        fields = [...]
```

```
# hooks.py

from wagtail.admin.views.account import BaseSettingsPanel
from wagtail import hooks
from .forms import CustomSettingsForm

@hooks.register('register_account_settings_panel')
class CustomSettingsPanel(BaseSettingsPanel):
    name = 'custom'
    title = "My custom settings"
    order = 500
    form_class = CustomSettingsForm
    form_object = 'user'
```

The attributes are as follows:

- `name` - A unique name for the panel. All form fields are prefixed with this name, so it must be lowercase and cannot contain symbols -
- `title` - The heading that is displayed to the user
- `order` - Used to order panels on a tab. The builtin Wagtail panels start at 100 and increase by 100 for each panel.
- `form_class` - A `ModelForm` subclass that operates on a user or a profile
- `form_object` - Set to `user` to operate on the user, and `profile` to operate on the profile
- `tab` (optional) - Set which tab the panel appears on.
- `template_name` (optional) - Override the default template used for rendering the panel

Operating on the `UserProfile` model

To add a panel that alters data on the user's `wagtail.users.models.UserProfile` instance, set `form_object` to '`profile`':

```
# forms.py

from django import forms
from wagtail.users.models import UserProfile
```

(continues on next page)

(continued from previous page)

```
class CustomProfileSettingsForm(forms.ModelForm):

    class Meta:
        model = UserProfile
        fields = [...]
```

```
# hooks.py

from wagtail.admin.views.account import BaseSettingsPanel
from wagtail import hooks
from .forms import CustomProfileSettingsForm

@hooks.register('register_account_settings_panel')
class CustomSettingsPanel(BaseSettingsPanel):
    name = 'custom'
    title = "My custom settings"
    order = 500
    form_class = CustomProfileSettingsForm
    form_object = 'profile'
```

Creating new tabs

You can define a new tab using the `SettingsTab` class:

```
# hooks.py

from wagtail.admin.views.account import BaseSettingsPanel, SettingsTab
from wagtail import hooks
from .forms import CustomSettingsForm

custom_tab = SettingsTab('custom', "Custom settings", order=300)

@hooks.register('register_account_settings_panel')
class CustomSettingsPanel(BaseSettingsPanel):
    name = 'custom'
    title = "My custom settings"
    tab = custom_tab
    order = 100
    form_class = CustomSettingsForm
```

`SettingsTab` takes three arguments:

- `name` - A slug to use for the tab (this is placed after the `#` when linking to a tab)
- `title` - The display name of the title
- `order` - The order of the tab. The builtin Wagtail tabs start at `100` and increase by `100` for each tab

Customising the template

You can provide a custom template for the panel by specifying a template name:

```
# hooks.py

from wagtail.admin.views.account import BaseSettingsPanel
from wagtail import hooks
from .forms import CustomSettingsForm

@hooks.register('register_account_settings_panel')
class CustomSettingsPanel(BaseSettingsPanel):
    name = 'custom'
    title = "My custom settings"
    order = 500
    form_class = CustomSettingsForm
    template_name = 'myapp/admin/custom_settings.html'
```

```
{# templates/myapp/admin/custom_settings.html #}

{# This is the default template Wagtail uses, which just renders the form #-}

{% block content %}
    {% for field in form %}
        {% include "wagtailadmin/shared/field.html" with field=field %}
    {% endfor %}
{% endblock %}
```

1.4.9 Customising group edit/create views

The views for managing groups within the app are collected into a ‘viewset’ class, which acts as a single point of reference for all shared components of those views, such as forms. By subclassing the viewset, it is possible to override those components and customise the behaviour of the group management interface.

Custom edit/create forms

This example shows how to customise forms on the ‘edit group’ and ‘create group’ views in the Wagtail admin.

Let’s say you need to connect Active Directory groups with Django groups. We create a model for Active Directory groups as follows:

```
from django.contrib.auth.models import Group
from django.db import models

class ADGroup(models.Model):
    guid = models.CharField(verbose_name="GUID", max_length=64, db_index=True, unique=True)
    name = models.CharField(verbose_name="Group", max_length=255)
    domain = models.CharField(verbose_name="Domain", max_length=255, db_index=True)
    description = models.TextField(verbose_name="Description", blank=True, null=True)
    roles = models.ManyToManyField(Group, verbose_name="Role", related_name="adgroups", blank=True)
```

(continues on next page)

(continued from previous page)

```
class Meta:
    verbose_name = "AD group"
    verbose_name_plural = "AD groups"
```

However, there is no role field on the Wagtail group ‘edit’ or ‘create’ view. To add it, inherit from `wagtail.users.forms.GroupForm` and add a new field:

```
from django import forms

from wagtail.users.forms import GroupForm as WagtailGroupForm

from .models import ADGroup


class GroupForm(WagtailGroupForm):
    adgroups = forms.ModelMultipleChoiceField(
        label="AD groups",
        required=False,
        queryset=ADGroup.objects.order_by("name"),
    )

    class Meta(WagtailGroupForm.Meta):
        fields = WagtailGroupForm.Meta.fields + ("adgroups",)

    def __init__(self, initial=None, instance=None, **kwargs):
        if instance is not None:
            if initial is None:
                initial = []
            initial["adgroups"] = instance.adgroups.all()
        super().__init__(initial=initial, instance=instance, **kwargs)

    def save(self, commit=True):
        instance = super().save()
        instance.adgroups.set(self.cleaned_data["adgroups"])
        return instance
```

Now add your custom form into the group viewset by inheriting the default Wagtail `GroupViewSet` class and overriding the `get_form_class` method.

```
from wagtail.users.views.groups import GroupViewSet as WagtailGroupViewSet

from .forms import GroupForm


class GroupViewSet(WagtailGroupViewSet):
    def get_form_class(self, for_update=False):
        return GroupForm
```

Add the field to the group ‘edit’/‘create’ templates:

```
{% extends "wagtailusers/groups/edit.html" %}
{% load wagtailusers_tags wagtailadmin_tags i18n %}
```

(continues on next page)

(continued from previous page)

```
{% block extra_fields %}
    <li>{% include "wagtailadmin/shared/field.html" with field=form.adgroups %}</li>
{% endblock extra_fields %}
```

Finally we configure the `wagtail.users` application to use the custom viewset, by setting up a custom `AppConfig` class. Within your project folder (which will be the package containing the top-level settings and urls modules), create `apps.py` (if it does not exist already) and add:

```
from wagtail.users.apps import WagtailUsers AppConfig

class CustomUsersAppConfig(WagtailUsers AppConfig):
    group_viewset = "myapplication.someapp.viewsets.GroupViewSet"
```

Replace `wagtail.users` in `settings.INSTALLED_APPS` with the path to `CustomUsersAppConfig`.

```
INSTALLED_APPS = [
    ...,
    "myapplication.apps.CustomUsersAppConfig",
    # "wagtail.users",
    ...,
]
```

1.4.10 Rich text internals

At first glance, Wagtail's rich text capabilities appear to give editors direct control over a block of HTML content. In reality, it's necessary to give editors a representation of rich text content that is several steps removed from the final HTML output, for several reasons:

- The editor interface needs to filter out certain kinds of unwanted markup; this includes malicious scripting, font styles pasted from an external word processor, and elements which would break the validity or consistency of the site design (for example, pages will generally reserve the `<h1>` element for the page title, and so it would be inappropriate to allow users to insert their own additional `<h1>` elements through rich text).
- Rich text fields can specify a `features` argument to further restrict the elements permitted in the field - see [Rich Text Features](#).
- Enforcing a subset of HTML helps to keep presentational markup out of the database, making the site more maintainable, and making it easier to repurpose site content (including, potentially, producing non-HTML output such as [LaTeX](#)).
- Elements such as page links and images need to preserve metadata such as the page or image ID, which is not present in the final HTML representation.

This requires the rich text content to go through a number of validation and conversion steps; both between the editor interface and the version stored in the database, and from the database representation to the final rendered HTML.

For this reason, extending Wagtail's rich text handling to support a new element is more involved than simply saying (for example) “enable the `<blockquote>` element”, since various components of Wagtail - both client and server-side - need to agree on how to handle that feature, including how it should be exposed in the editor interface, how it should be represented within the database, and (if appropriate) how it should be translated when rendered on the front-end.

The components involved in Wagtail's rich text handling are described below.

Data format

Rich text data (as handled by `RichTextField`, and `RichTextBlock` within `StreamField`) is stored in the database in a format that is similar, but not identical, to HTML. For example, a link to a page might be stored as:

```
<p><a linktype="page" id="3">Contact us</a> for more information.</p>
```

Here, the `linktype` attribute identifies a rule that shall be used to rewrite the tag. When rendered on a template through the `|richtext` filter (see [rich text filter](#)), this is converted into valid HTML:

```
<p><a href="/contact-us/">Contact us</a> for more information.</p>
```

In the case of `RichTextBlock`, the block's value is a `RichText` object which performs this conversion automatically when rendered as a string, so the `|richtext` filter is not necessary.

Likewise, an image inside rich text content might be stored as:

```
<embed embedtype="image" id="10" alt="A pied wagtail" format="left" />
```

which is converted into an `img` element when rendered:

```

```

Again, the `embedtype` attribute identifies a rule that shall be used to rewrite the tag. All tags other than `` and `<embed embedtype="..." />` are left unchanged in the converted HTML.

A number of additional constraints apply to `` and `<embed embedtype="..." />` tags, to allow the conversion to be performed efficiently via string replacement:

- The tag name and attributes must be lower-case
- Attribute values must be quoted with double-quotes
- `embed` elements must use XML self-closing tag syntax (those that end in `/>` instead of a closing `</embed>` tag)
- The only HTML entities permitted in attribute values are `<`, `>`, `&` and `"`

The feature registry

Any app within your project can define extensions to Wagtail's rich text handling, such as new `linktype` and `embedtype` rules. An object known as the *feature registry* serves as a central source of truth about how rich text should behave. This object can be accessed through the `Register Rich Text Features` hook, which is called on startup to gather all definitions relating to rich text:

```
# my_app/wagtail_hooks.py

from wagtail import hooks

@hooks.register('register_rich_text_features')
```

(continues on next page)

(continued from previous page)

```
def register_my_feature(features):
    # add new definitions to 'features' here
```

Rewrite handlers

Rewrite handlers are classes that know how to translate the content of rich text tags like `` and `<embed embedtype="..." />` into front-end HTML. For example, the `PageLinkHandler` class knows how to convert the rich text tag `` into the HTML tag ``.

Rewrite handlers can also provide other useful information about rich text tags. For example, given an appropriate tag, `PageLinkHandler` can be used to extract which page is being referred to. This can be useful for downstream code that may want information about objects being referenced in rich text.

You can create custom rewrite handlers to support your own new `linktype` and `embedtype` tags. New handlers must be Python classes that inherit from either `wagtail.richtext.LinkHandler` or `wagtail.richtext.EmbedHandler`. Your new classes should override at least some of the following methods (listed here for `LinkHandler`, although `EmbedHandler` has an identical signature):

`class LinkHandler`

`identifier`

Required. The `identifier` attribute is a string that indicates which rich text tags should be handled by this handler.

For example, `PageLinkHandler.identifier` is set to the string "page", indicating that any rich text tags with `` should be handled by it.

`expand_db_attributes(attrs)`

Required. The `expand_db_attributes` method is expected to take a dictionary of attributes from a database rich text `<a>` tag (`<embed>` for `EmbedHandler`) and use it to generate valid frontend HTML.

For example, `PageLinkHandler.expand_db_attributes` might receive `{'id': 123}`, use it to retrieve the Wagtail page with ID 123, and render a link to its URL like ``.

`get_model()`

Optional. The static `get_model` method only applies to those handlers that are used to render content related to Django models. This method allows handlers to expose the type of content that they know how to handle.

For example, `PageLinkHandler.get_model` returns the Wagtail class `Page`.

Handlers that aren't related to Django models can leave this method undefined, and calling it will raise `NotImplementedError`.

`get_instance(attrs)`

Optional. The static or classmethod `get_instance` method also only applies to those handlers that are used to render content related to Django models. This method is expected to take a dictionary of attributes from a database rich text `<a>` tag (`<embed>` for `EmbedHandler`) and use it to return the specific Django model instance being referred to.

For example, `PageLinkHandler.get_instance` might receive `{'id': 123}` and return the instance of the Wagtail `Page` class with ID 123.

If left undefined, a default implementation of this method will query the `id` model field on the class returned by `get_model` using the provided `id` attribute; this can be overridden in your own handlers should you want to use some other model field.

Below is an example custom rewrite handler that implements these methods to add support for rich text linking to user email addresses. It supports the conversion of rich text tags like `` to valid HTML like ``. This example assumes that equivalent front-end functionality has been added to allow users to insert these kinds of links into their rich text editor.

```
from django.contrib.auth import get_user_model
from wagtail.rich_text import LinkHandler

class UserLinkHandler(LinkHandler):
    identifier = 'user'

    @staticmethod
    def get_model():
        return get_user_model()

    @classmethod
    def get_instance(cls, attrs):
        model = cls.get_model()
        return model.objects.get(username=attrs['username'])

    @classmethod
    def expand_db_attributes(cls, attrs):
        user = cls.get_instance(attrs)
        return '<a href="mailto:%s">' % user.email
```

Registering rewrite handlers

Rewrite handlers must also be registered with the feature registry via the `register rich text features` hook. Independent methods for registering both link handlers and embed handlers are provided.

```
FeatureRegistry.register_link_type(handler)
```

This method allows you to register a custom handler deriving from `wagtail.rich_text.LinkHandler`, and adds it to the list of link handlers available during rich text conversion.

```
# my_app/wagtail_hooks.py

from wagtail import hooks
from my_app.handlers import MyCustomLinkHandler

@hooks.register('register_rich_text_features')
def register_link_handler(features):
    features.register_link_type(MyCustomLinkHandler)
```

It is also possible to define link rewrite handlers for Wagtail's built-in `external` and `email` links, even though they do not have a predefined `linktype`. For example, if you want external links to have a `rel="nofollow"` attribute for SEO purposes:

```
from django.utils.html import escape
from wagtail import hooks
from wagtail.rich_text import LinkHandler

class NoFollowExternalLinkHandler(LinkHandler):
```

(continues on next page)

(continued from previous page)

```
identifier = 'external'

@classmethod
def expand_db_attributes(cls, attrs):
    href = attrs["href"]
    return '<a href="%s" rel="nofollow">' % escape(href)

@hooks.register('register_rich_text_features')
def register_external_link(features):
    features.register_link_type(NoFollowExternalLinkHandler)
```

Similarly you can use `email` linktype to add a custom rewrite handler for email links (for example to obfuscate emails in rich text).

`FeatureRegistry.register_embed_type(handler)`

This method allows you to register a custom handler deriving from `wagtail.rich_text.EmbedHandler`, and adds it to the list of embed handlers available during rich text conversion.

```
# my_app/wagtail_hooks.py

from wagtail import hooks
from my_app.handlers import MyCustomEmbedHandler

@hooks.register('register_rich_text_features')
def register_embed_handler(features):
    features.register_embed_type(MyCustomEmbedHandler)
```

Editor widgets

The editor interface used on rich text fields can be configured with the `WAGTAILADMIN_RICH_TEXT_EDITORS` setting. Wagtail provides an implementation: `wagtail.admin.rich_text.DraftailRichTextArea` (the Draftail editor based on `Draft.js`).

It is possible to create your own rich text editor implementation. At minimum, a rich text editor is a Django `class django.forms.Widget` subclass whose constructor accepts an `options` keyword argument (a dictionary of editor-specific configuration options sourced from the `OPTIONS` field in `WAGTAILADMIN_RICH_TEXT_EDITORS`), and which consumes and produces string data in the HTML-like format described above.

Typically, a rich text widget also receives a `features` list, passed from either `RichTextField` / `RichTextBlock` or the `features` option in `WAGTAILADMIN_RICH_TEXT_EDITORS`, which defines the features available in that instance of the editor (see [rich text features](#)). To opt in to supporting features, set the attribute `accepts_features = True` on your widget class; the widget constructor will then receive the feature list as a keyword argument `features`.

There is a standard set of recognised feature identifiers as listed under [rich text features](#), but this is not a definitive list; feature identifiers are only defined by convention, and it is up to each editor widget to determine which features it will recognise, and adapt its behaviour accordingly. Individual editor widgets might implement fewer or more features than the default set, either as built-in functionality or through a plugin mechanism if the editor widget has one.

For example, a third-party Wagtail extension might introduce `table` as a new rich text feature, and provide implementations for the Draftail editor (which provides a plugin mechanism). In this case, the third-party extension will not be aware of your custom editor widget, and so the widget will not know how to handle the `table` feature identifier. Editor widgets should silently ignore any feature identifiers that they do not recognise.

The `default_features` attribute of the feature registry is a list of feature identifiers to be used whenever an explicit feature list has not been given in `RichTextField` / `RichTextBlock` or `WAGTAILADMIN_RICH_TEXT_EDITORS`. This list can be modified within the `register_rich_text_features` hook to make new features enabled by default, and retrieved by calling `get_default_features()`.

```
@hooks.register('register_rich_text_features')
def make_h1_default(features):
    features.default_features.append('h1')
```

Outside of the `register_rich_text_features` hook - for example, inside a widget class - the feature registry can be imported as the object `wagtail.rich_text.features`. A possible starting point for a rich text editor with feature support would be:

```
from django.forms import widgets
from wagtail.rich_text import features

class CustomRichTextArea(widgets.TextArea):
    accepts_features = True

    def __init__(self, *args, **kwargs):
        self.options = kwargs.pop('options', None)

        self.features = kwargs.pop('features', None)
        if self.features is None:
            self.features = features.get_default_features()

    super().__init__(*args, **kwargs)
```

Editor plugins

`FeatureRegistry.register_editor_plugin(editor_name, feature_name, plugin_definition)`

Rich text editors often provide a plugin mechanism to allow extending the editor with new functionality. The `register_editor_plugin` method provides a standardised way for `register_rich_text_features` hooks to define plugins to be pulled in to the editor when a given rich text feature is enabled.

`register_editor_plugin` is passed an editor name (a string uniquely identifying the editor widget - Wagtail uses the identifier `draftail` for the built-in editor), a feature identifier, and a plugin definition object. This object is specific to the editor widget and can be any arbitrary value, but will typically include a `Django form media` definition referencing the plugin's JavaScript code - which will then be merged into the editor widget's own media definition - along with any relevant configuration options to be passed when instantiating the editor.

`FeatureRegistry.get_editor_plugin(editor_name, feature_name)`

Within the editor widget, the plugin definition for a given feature can be retrieved via the `get_editor_plugin` method, passing the editor's own identifier string and the feature identifier. This will return `None` if no matching plugin has been registered.

For details of the plugin formats for Wagtail's built-in editors, see [Extending the Draftail Editor](#).

Format converters

Editor widgets will often be unable to work directly with Wagtail’s rich text format, and require conversion to their own native format. For Draftail, this is a JSON-based format known as ContentState (see [How Draft.js Represents Rich Text Data](#)). Editors based on HTML’s `contentEditable` mechanism require valid HTML, and so Wagtail uses a convention referred to as “editor HTML”, where the additional data required on link and embed elements is stored in `data-` attributes, for example: `Contact us`.

Wagtail provides two utility classes, `wagtail.admin.rich_text.converters.contentstate.ContentstateConverter` and `wagtail.admin.rich_text.converters.editor_html.EditorHTMLConverter`, to perform conversions between rich text format and the native editor formats. These classes are independent of any editor widget, and distinct from the rewriting process that happens when rendering rich text onto a template.

Both classes accept a `features` list as an argument to their constructor, and implement two methods, `from_database_format(data)` which converts Wagtail rich text data to the editor’s format, and `to_database_format(data)` which converts editor data to Wagtail rich text format.

As with editor plugins, the behaviour of a converter class can vary according to the feature list passed to it. In particular, it can apply whitelisting rules to ensure that the output only contains HTML elements corresponding to the currently active feature set. The feature registry provides a `register_converter_rule` method to allow `register_rich_text_features` hooks to define conversion rules that will be activated when a given feature is enabled.

`FeatureRegistry.register_converter_rule(converter_name, feature_name, rule_definition)`

`register_editor_plugin` is passed a converter name (a string uniquely identifying the converter class - Wagtail uses the identifiers `contentstate` and `editorhtml`), a feature identifier, and a rule definition object. This object is specific to the converter and can be any arbitrary value.

For details of the rule definition format for the `contentstate` converter, see [Extending the Draftail Editor](#).

`FeatureRegistry.get_converter_rule(converter_name, feature_name)`

Within a converter class, the rule definition for a given feature can be retrieved via the `get_converter_rule` method, passing the converter’s own identifier string and the feature identifier. This will return `None` if no matching rule has been registered.

1.4.11 Extending the Draftail Editor

Wagtail’s rich text editor is built with `Draftail`, and its functionality can be extended through plugins.

Plugins come in three types:

- Inline styles – To format a portion of a line, for example `bold`, `italic`, `monospace`.
- Blocks – To indicate the structure of the content, for example `blockquote`, `ol`.
- Entities – To enter additional data/metadata, for example `link` (with a URL), `image` (with a file).

All of these plugins are created with a similar baseline, which we can demonstrate with one of the simplest examples – a custom feature for an inline style of `mark`. Place the following in a `wagtail_hooks.py` file in any installed app:

```
import wagtail.admin.rich_text.editors.draftail.features as draftail_features
from wagtail.admin.rich_text.converters.html_to_contentstate import_
    InlineStyleElementHandler
from wagtail import hooks
```

(continues on next page)

(continued from previous page)

```

# 1. Use the register_rich_text_features hook.
@hooks.register('register_rich_text_features')
def register_mark_feature(features):
    """
    Registering the `mark` feature, which uses the `MARK` Draft.js inline style type,
    and is stored as HTML with a `` tag.
    """

    feature_name = 'mark'
    type_ = 'MARK'
    tag = 'mark'

    # 2. Configure how Draftail handles the feature in its toolbar.
    control = {
        'type': type_,
        'label': '',
        'description': 'Mark',
        # This isn't even required - Draftail has predefined styles for MARK.
        # 'style': {'textDecoration': 'line-through'},
    }

    # 3. Call register_editor_plugin to register the configuration for Draftail.
    features.register_editor_plugin(
        'draftail', feature_name, draftail_features.InlineStyleFeature(control)
    )

    # 4. configure the content transform from the DB to the editor and back.
    db_conversion = {
        'from_database_format': {tag: InlineStyleElementHandler(type_)},
        'to_database_format': {'style_map': {type_: tag}},
    }

    # 5. Call register_converter_rule to register the content transformation conversion.
    features.register_converter_rule('contentstate', feature_name, db_conversion)

    # 6. (optional) Add the feature to the default features list to make it available
    # on rich text fields that do not specify an explicit 'features' list
    features.default_features.append('mark')

```

These steps will always be the same for all Draftail plugins. The important parts are to:

- Consistently use the feature's Draft.js type or Wagtail feature names where appropriate.
- Give enough information to Draftail so it knows how to make a button for the feature, and how to render it (more on this later).
- Configure the conversion to use the right HTML element (as they are stored in the DB).

For detailed configuration options, head over to the [Draftail documentation](#) to see all of the details. Here are some parts worth highlighting about controls:

- The `type` is the only mandatory piece of information.
- To display the control in the toolbar, combine `icon`, `label` and `description`.
- The controls' `icon` can be a string to use an icon font with CSS classes, say `'icon': 'fas fa-user'`,. It

can also be an array of strings, to use SVG paths, or SVG symbol references for example 'icon': ['M100 100 H 900 V 900 H 100 Z'],. The paths need to be set for a 1024x1024 viewbox.

Creating new inline styles

In addition to the initial example, inline styles take a `style` property to define what CSS rules will be applied to text in the editor. Be sure to read the [Draftail documentation](#) on inline styles.

Finally, the DB to/from conversion uses an `InlineStyleElementHandler` to map from a given tag (`<mark>` in the example above) to a Draftail type, and the inverse mapping is done with `Draft.js` exporter configuration of the `style_map`.

Creating new blocks

Blocks are nearly as simple as inline styles:

```
import wagtail.admin.rich_text.editors.draftail.features as draftail_features
from wagtail.admin.rich_text.converters.html_to_contentstate import BlockElementHandler

@hooks.register('register_rich_text_features')
def register_help_text_feature(features):
    """
    Registering the `help-text` feature, which uses the `help-text` Draft.js block type,
    and is stored as HTML with a `

` tag.
    """

    feature_name = 'help-text'
    type_ = 'help-text'

    control = {
        'type': type_,
        'label': '?',
        'description': 'Help text',
        # Optionally, we can tell Draftail what element to use when displaying those
        # blocks in the editor.
        'element': 'div',
    }

    features.register_editor_plugin(
        'draftail', feature_name, draftail_features.BlockFeature(control, css={'all': [
            'help-text.css'
        ]})
    )

    features.register_converter_rule('contentstate', feature_name, {
        'from_database_format': {'div[class=help-text]': BlockElementHandler(type_)},
        'to_database_format': {'block_map': {type_: {'element': 'div', 'props': {'class': ': help-text'}}}},
    })


```

Here are the main differences:

- We can configure an `element` to tell Draftail how to render those blocks in the editor.
- We register the plugin with `BlockFeature`.
- We set up the conversion with `BlockElementHandler` and `block_map`.

Optionally, we can also define styles for the blocks with the `Draftail-block--help-text` (`Draftail-block--<block type>`) CSS class.

That's it! The extra complexity is that you may need to write CSS to style the blocks in the editor.

Creating new entities

Warning: This is an advanced feature. Please carefully consider whether you really need this.

Entities aren't simply formatting buttons in the toolbar. They usually need to be much more versatile, communicating to APIs or requesting further user input. As such,

- You will most likely need to write a **hefty dose of JavaScript**, some of it with React.
- The API is very **low-level**. You will most likely need some **Draft.js knowledge**.
- Custom UIs in rich text can be brittle. Be ready to spend time **testing in multiple browsers**.

The good news is that having such a low-level API will enable third-party Wagtail plugins to innovate on rich text features, proposing new kinds of experiences. But in the meantime, consider implementing your UI through `StreamField` instead, which has a battle-tested API meant for Django developers.

Here are the main requirements to create a new entity feature:

- Like for inline styles and blocks, register an editor plugin.
- The editor plugin must define a **source**: a React component responsible for creating new entity instances in the editor, using the Draft.js API.
- The editor plugin also needs a **decorator** (for inline entities) or **block** (for block entities): a React component responsible for displaying entity instances within the editor.
- Like for inline styles and blocks, set up the to/from DB conversion.
- The conversion usually is more involved, since entities contain data that needs to be serialised to HTML.

To write the React components, Wagtail exposes its own React, Draft.js and Draftail dependencies as global variables. Read more about this in [extending clientside components](#). To go further, please look at the [Draftail documentation](#) as well as the [Draft.js exporter documentation](#).

Here is a detailed example to showcase how those tools are used in the context of Wagtail. For the sake of our example, we can imagine a news team working at a financial newspaper. They want to write articles about the stock market, refer to specific stocks anywhere inside of their content (for example “\$TSLA” tokens in a sentence), and then have their article automatically enriched with the stock’s information (a link, a number, a sparkline).

The editor toolbar could contain a “stock chooser” that displays a list of available stocks, then inserts the user’s selection as a textual token. For our example, we will just pick a stock at random:

Those tokens are then saved in the rich text on publish. When the news article is displayed on the site, we then insert live market data coming from an API next to each token:

Anyone following Elon Musk’s `$TSLA` — should also look into `$BTC` —.

In order to achieve this, we start with registering the rich text feature like for inline styles and blocks:

```

@hooks.register('register_rich_text_features')
def register_stock_feature(features):
    features.default_features.append('stock')
    """
    Registering the `stock` feature, which uses the `STOCK` Draft.js entity type,
    and is stored as HTML with a `` tag.
    """
    feature_name = 'stock'
    type_ = 'STOCK'

    control = {
        'type': type_,
        'label': '$',
        'description': 'Stock',
    }

    features.register_editor_plugin(
        'draftail', feature_name, draftail_features.EntityFeature(
            control,
            js=['stock.js'],
            css={'all': ['stock.css']}
        )
    )

    features.register_converter_rule('contentstate', feature_name, {
        # Note here that the conversion is more complicated than for blocks and inline_
        # styles.
        'from_database_format': {'span[data-stock]': StockEntityElementHandler(type_)},
        'to_database_format': {'entity_decorators': {type_: stock_entity_decorator}},
    })

```

The `js` and `css` keyword arguments on `EntityFeature` can be used to specify additional JS and CSS files to load when this feature is active. Both are optional. Their values are added to a `Media` object, more documentation on these objects is available in the [Django Form Assets documentation](#).

Since entities hold data, the conversion to/from database format is more complicated. We have to create the two handlers:

```

from draftjs_exporter.dom import DOM
from wagtail.admin.rich_text.converters.html_to_contentstate import_
    InlineEntityElementHandler

def stock_entity_decorator(props):
    """
    Draft.js ContentState to database HTML.
    Converts the STOCK entities into a span tag.
    """
    return DOM.create_element('span', {
        'data-stock': props['stock'],
    }, props['children'])

class StockEntityElementHandler(InlineEntityElementHandler):

```

(continues on next page)

(continued from previous page)

```

"""
Database HTML to Draft.js ContentState.
Converts the span tag into a STOCK entity, with the right data.
"""

mutability = 'IMMUTABLE'

def get_attribute_data(self, attrs):
    """
    Take the `stock` value from the `data-stock` HTML attribute.
    """
    return { 'stock': attrs['data-stock'] }

```

Note how they both do similar conversions, but use different APIs. `to_database_format` is built with the Draft.js exporter components API, whereas `from_database_format` uses a Wagtail API.

The next step is to add JavaScript to define how the entities are created (the source), and how they are displayed (the decorator). Within `stock.js`, we define the source component:

```

const React = window.React;
const Modifier = window.DraftJS.Modifier;
const EditorState = window.DraftJS.EditorState;

const DEMO_STOCKS = ['AMD', 'AAPL', 'TWTR', 'TSLA', 'BTC'];

// Not a real React component - just creates the entities as soon as it is rendered.
class StockSource extends React.Component {
    componentDidMount() {
        const { editorState, entityType, onComplete } = this.props;

        const content = editorState.getCurrentContent();
        const selection = editorState.getSelection();

        const randomStock =
            DEMO_STOCKS[Math.floor(Math.random() * DEMO_STOCKS.length)];

        // Uses the Draft.js API to create a new entity with the right data.
        const contentWithEntity = content.createEntity(
            entityType.type,
            'IMMUTABLE',
            {
                stock: randomStock,
            },
        );
        const entityKey = contentWithEntity.getLastCreatedEntityKey();

        // We also add some text for the entity to be activated on.
        const text = ` ${randomStock} `;

        const newContent = Modifier.replaceText(
            content,
            selection,
            text,
            null,
        );
    }
}

```

(continues on next page)

(continued from previous page)

```

        entityKey,
    );
    const nextState = EditorState.push(
        editorState,
        newContent,
        'insert-characters',
    );
    onComplete(nextState);
}

render() {
    return null;
}
}

```

This source component uses data and callbacks provided by [Draftail](#). It also uses dependencies from global variables – see [Extending clientside components](#).

We then create the decorator component:

```

const Stock = (props) => {
    const { entityKey, contentState } = props;
    const data = contentState.getEntity(entityKey).getData();

    return React.createElement(
        'a',
        {
            role: 'button',
            onMouseUp: () => {
                window.open(`https://finance.yahoo.com/quote/${data.stock}`);
            },
            ...,
            props.children,
        );
};

```

This is a straightforward React component. It does not use JSX since we do not want to have to use a build step for our JavaScript.

Finally, we register the JS components of our plugin:

```

window.draftail.registerPlugin({
    type: 'STOCK',
    source: StockSource,
    decorator: Stock,
});

```

And that's it! All of this setup will finally produce the following HTML on the site's front-end:

```

<p>
    Anyone following Elon Musk's <span data-stock="TSLA">$TSLA</span> should
    also look into <span data-stock="BTC">$BTC</span>.
</p>

```

To fully complete the demo, we can add a bit of JavaScript to the front-end in order to decorate those tokens with links and a little sparkline.

```
document.querySelectorAll('[data-stock]').forEach((elt) => {
    const link = document.createElement('a');
    link.href = `https://finance.yahoo.com/quote/${elt.dataset.stock}`;
    link.innerHTML = `${elt.innerHTML}<svg width="50" height="20" stroke-width="2" style="stroke:#007bff; fill:#007bff"><path d="M4 14.19 L 4 14.19 L 13.2 14.21 L 22.4 13.77 L 31.59 13.99 L 40.8 13.46 L 50 11.68 L 59.19 11.35 L 68.39 10.68 L 77.6 7.11 L 86.8 7.85 L 96 4" fill="none"></path><path d="M4 14.19 L 4 14.19 L 13.2 14.21 L 22.4 13.77 L 31.59 13.99 L 40.8 13.46 L 50 11.68 L 59.19 11.35 L 68.39 10.68 L 77.6 7.11 L 86.8 7.85 L 96 4 V 20 L 4 20 Z" stroke="none"></path></svg>`;

    elt.innerHTML = '';
    elt.appendChild(link);
});
```

Custom block entities can also be created (have a look at the separate [Draftail documentation](#)), but these are not detailed here since `StreamField` is the go-to way to create block-level rich text in Wagtail.

Integration of the Draftail widgets

To further customise how the Draftail widgets are integrated into the UI, there are additional extension points for CSS and JS:

- In JavaScript, use the `[data-draftail-input]` attribute selector to target the input which contains the data, and `[data-draftail-editor-wrapper]` for the element which wraps the editor.
- The editor instance is bound on the input field for imperative access. Use `document.querySelector('[data-draftail-input]').draftailEditor`.
- In CSS, use the classes prefixed with `Draftail-`.

1.4.12 Adding custom bulk actions

This document describes how to add custom bulk actions to different listings.

Registering a custom bulk action

```
from wagtail.admin.views.bulk_action import BulkAction
from wagtail import hooks

@hooks.register('register_bulk_action')
class CustomDeleteBulkAction(BulkAction):
    display_name = _("Delete")
    aria_label = _("Delete selected objects")
    action_type = "delete"
    template_name = "/path/to/confirm_bulk_delete.html"
    models = [...]

    @classmethod
```

(continues on next page)

(continued from previous page)

```
def execute_action(cls, objects, **kwargs):
    for obj in objects:
        do_something(obj)
    return num_parent_objects, num_child_objects # return the count of updated
→objects
```

The attributes are as follows:

- `display_name` - The label that will be displayed on the button in the user interface
- `aria_label` - The `aria-label` attribute that will be applied to the button in the user interface
- `action_type` - A unique identifier for the action. Will be required in the url for bulk actions
- `template_name` - The path to the confirmation template
- `models` - A list of models on which the bulk action can act
- `action_priority` (optional) - A number that is used to determine the placement of the button in the list of buttons
- `classes` (optional) - A set of CSS classnames that will be used on the button in the user interface

An example for a confirmation template is as follows:

```
<!-- /path/to/confirm_bulk_delete.html -->

{% extends 'wagtailadmin/bulk_actions/confirmation/base.html' %}
{% load i18n wagtailadmin_tags %}

{% block titletag %}{% blocktrans trimmed count counter=items|length %}Delete 1 item{%_
→plural %}Delete {{ counter }} items{% endblocktrans %}{% endblock %}

{% block header %}
    {% trans "Delete" as del_str %}
    {% include "wagtailadmin/shared/header.html" with title=del_str icon="doc-empty-
→inverse" %}
{% endblock header %}

{% block items_with_access %}
    {% if items %}
        <p>{% trans "Are you sure you want to delete these items?" %}</p>
        <ul>
            {% for item in items %}
                <li>
                    <a href="" target="_blank" rel="nofollow">{{ item.item.title }}</a>
                </li>
            {% endfor %}
        </ul>
    {% endif %}
{% endblock items_with_access %}

{% block items_with_no_access %}
    {% blocktrans trimmed as var no_access_msg count counter=items_with_no_access|length %}
        You don't have permission to delete this item
    {% plural %}
        You don't have permission to
        delete these items
    {% endblocktrans %}

```

(continues on next page)

(continued from previous page)

```
{% include './list_items_with_no_access.html' with items=items_with_no_access no_access_
↪msg=no_access_msg %}

{% endblock items_with_no_access %}

{% block form_section %}
{% if items %}
    {% trans 'Yes, delete' as action_button_text %}
    {% trans "No, don't delete" as no_action_button_text %}
    {% include 'wagtailadmin/bulk_actions/confirmation/form.html' with action_button_
↪class="serious" %}
{% else %}
    {% include 'wagtailadmin/bulk_actions/confirmation/go_back.html' %}
{% endif %}
{% endblock form_section %}
```

```
<!-- ./list_items_with_no_access.html -->
{% extends 'wagtailadmin/bulk_actions/confirmation/list_items_with_no_access.html' %}
{% load i18n %}

{% block per_item %}
    {% if item.can_edit %}
        <a href="{% url 'wagtailadmin_pages:edit' item.item.id %}" target="_blank" rel=
↪" noreferrer">{{ item.item.title }}</a>
    {% else %}
        {{ item.item.title }}
    {% endif %}
{% endblock per_item %}
```

The `execute_action` classmethod is the only method that must be overridden for the bulk action to work properly. It takes a list of objects as the only required argument, and a bunch of keyword arguments that can be supplied by overriding the `get_execution_context` method. For example.

```
@classmethod
def execute_action(cls, objects, **kwargs):
    # the kwargs here is the output of the get_execution_context method
    user = kwargs.get('user', None)
    num_parent_objects, num_child_objects = 0, 0
    # you could run the action per object or run them in bulk using django's bulk update
    # and delete methods
    for obj in objects:
        num_child_objects += obj.get_children().count()
        num_parent_objects += 1
        obj.delete(user=user)
        num_parent_objects += 1
    return num_parent_objects, num_child_objects
```

The `get_execution_context` method can be overridden to provide context to the `execute_action`

```
def get_execution_context(self):
    return { 'user': self.request.user }
```

The `get_context_data` method can be overridden to pass additional context to the confirmation template.

```
def get_context_data(self, **kwargs):
    context = super().get_context_data(**kwargs)
    context['new_key'] = some_value
    return context
```

The `check_perm` method can be overridden to check if an object has some permission or not. Objects for which the `check_perm` returns `False` will be available in the context under the key '`items_with_no_access`'.

```
def check_perm(self, obj):
    return obj.has_perm('some_perm') # returns True or False
```

The success message shown on the admin can be customised by overriding the `get_success_message` method.

```
def get_success_message(self, num_parent_objects, num_child_objects):
    return _("{} objects, including {} child objects have been updated".format(num_
    ↴parent_objects, num_child_objects))
```

Adding bulk actions to the page explorer

When creating a custom bulk action class for pages, subclass from `wagtail.admin.views.pages.bulk_actions.page_bulk_action.PageBulkAction` instead of `wagtail.admin.views.bulk_action.BulkAction`

Basic example

```
from wagtail.admin.views.pages.bulk_actions.page_bulk_action import PageBulkAction
from wagtail import hooks

@hooks.register('register_bulk_action')
class CustomPageBulkAction(PageBulkAction):
    ...
```

Adding bulk actions to the Images listing

When creating a custom bulk action class for images, subclass from `wagtail.images.views.bulk_actions.image_bulk_action.ImageBulkAction` instead of `wagtail.admin.views.bulk_action.BulkAction`

Basic example

```
from wagtail.images.views.bulk_actions.image_bulk_action import ImageBulkAction
from wagtail import hooks

@hooks.register('register_bulk_action')
class CustomImageBulkAction(ImageBulkAction):
    ...
```

Adding bulk actions to the documents listing

When creating a custom bulk action class for documents, subclass from `wagtail.documents.views.bulk_actions.document_bulk_action.DocumentBulkAction` instead of `wagtail.admin.views.bulk_action.BulkAction`

Basic example

```
from wagtail.documents.views.bulk_actions.document_bulk_action import DocumentBulkAction
from wagtail import hooks

@hooks.register('register_bulk_action')
class CustomDocumentBulkAction(DocumentBulkAction):
    ...
```

Adding bulk actions to the user listing

When creating a custom bulk action class for users, subclass from `wagtail.users.views.bulk_actions.user_bulk_action.UserBulkAction` instead of `wagtail.admin.views.bulk_action.BulkAction`

Basic example

```
from wagtail.users.views.bulk_actions.user_bulk_action import UserBulkAction
from wagtail import hooks

@hooks.register('register_bulk_action')
class CustomUserBulkAction(UserBulkAction):
    ...
```

Adding bulk actions to the snippets listing

When creating a custom bulk action class for snippets, subclass from `wagtail.snippets.views.bulk_actions.snippet_bulk_action.SnippetBulkAction` instead of `wagtail.admin.views.bulk_action.BulkAction`

Basic example

```
from wagtail.snippets.views.bulk_actions.snippet_bulk_action import SnippetBulkAction
from wagtail import hooks

@hooks.register('register_bulk_action')
class CustomSnippetBulkAction(SnippetBulkAction):
    # ...
```

If you want to apply an action only to certain snippets, override the `models` list in the action class

```
from wagtail.snippets.bulk_actions.snippet_bulk_action import SnippetBulkAction
from wagtail import hooks

@hooks.register('register_bulk_action')
class CustomSnippetBulkAction(SnippetBulkAction):
    models = [SnippetA, SnippetB]
    # ...
```

1.5 Reference

1.5.1 Pages

Wagtail requires a little careful setup to define the types of content that you want to present through your website. The basic unit of content in Wagtail is the :class:`~wagtail.models.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 organises 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 organise and interrelate your content, but here we've sketched out some strategies for organising 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 organise content for editing, letting you navigate up and down levels in the tree through its Explorer menu. This method of organisation 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 organise 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/kwags` 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 behaviour to this process by overriding `Page` class methods such as `route()` and `serve()` in your own models. For examples, see [Recipes](#).

Scheduled Publishing

Page publishing can be scheduled through the *Go live date/time* feature in the *Settings* tab of the *Edit* page. This allows you to set up initial page publishing or a page update in advance. In order for pages to be published at the scheduled time you should set up the `publish_scheduled_pages` management command.

The basic workflow is as follows:

- Scheduling a revision for a page that is not currently live means that page will go live when the scheduled time comes.
- Scheduling a revision for a page that is already live means that revision will be published when the time comes.
- If page has a scheduled revision and you set another revision to publish immediately, the scheduled revision will be unscheduled.

The *Revisions* view for a given page will show which revision is scheduled and when it is scheduled for. A scheduled revision in the list will also provide an *Unschedule* button to cancel it.

Recipes

Overriding the `serve()` Method

Wagtail defaults to serving `Page`-derived models by passing a reference to the page object to a Django HTML template matching the model's name, but suppose you wanted to serve something other than HTML? You can override the `serve()` method provided by the `Page` class and handle the Django request and response more directly.

Consider this example of an `EventPage` object which is served 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().serve(request)
```

`serve()` takes a Django request object and returns a Django response object. Wagtail returns a `TemplateResponse` object with the template and context which it generates, which allows middleware to function as intended, so keep in mind that a simpler response object like a `HttpResponse` will not receive these benefits.

With this strategy, you could use Django or Python utilities to render your model in JSON or XML or any other format you'd like.

Adding Endpoints with Custom `route()` Methods

Note: A much simpler way of adding more endpoints to pages is provided by the `RoutablePageMixin` mixin.

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(...):
    ...
```

(continues on next page)

(continued from previous 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.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, let's make a simple model which prints out all of its child path components.

First, `models.py`:

```

from django.shortcuts import render
from wagtail.url_routing import RouteResult
from django.http.response import Http404
from wagtail.models import Page

# ...

class Echoer(Page):

    def route(self, request, path_components):
        if path_components:
            # tell Wagtail to call self.serve() with an additional 'path_components' kwarg
            return RouteResult(self, kwargs={'path_components': path_components})
        else:

```

(continues on next page)

(continued from previous page)

```

if self.live:
    # tell Wagtail to call self.serve() with no further args
    return RouteResult(self)
else:
    raise Http404

def serve(self, path_components=[]):
    return render(request, self.template, {
        'page': self,
        'echo': ' '.join(path_components),
    })

```

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, variant='radian')

```

Tagging

Wagtail provides tagging capabilities through the combination of two Django modules, `django-taggit` (which provides a general-purpose tagging implementation) and `django-modelcluster` (which extends `django-taggit`'s `TaggableManager` to allow tag relations to be managed in memory without writing to the database - necessary for handling previews and revisions). To add tagging to a page model, you'll need to define a 'through' model inheriting from `TaggedItemBase` to set up the many-to-many relationship between `django-taggit`'s `Tag` model and your page model, and add a `ClusterTaggableManager` accessor to your page model to present this relation as a single tag field.

In this example, we set up tagging on `BlogPage` through a `BlogPageTag` model:

```

# models.py

from modelcluster.fields import ParentalKey
from modelcluster.contrib.taggit import ClusterTaggableManager
from taggit.models import TaggedItemBase

class BlogPageTag(TaggedItemBase):
    content_object = ParentalKey('demo.BlogPage', on_delete=models.CASCADE, related_name='tagged_items')

```

(continues on next page)

(continued from previous page)

```
class BlogPage(Page):
    ...
    tags = ClusterTaggableManager(through=BlogPageTag, blank=True)

    promote_panels = Page.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.

We can now make use of the many-to-many tag relationship in our views and templates. For example, we can set up the blog's index page to accept a `?tag=...` query parameter to filter the `BlogPage` listing by tag:

```
from django.shortcuts import render

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

        # Get blog entries
        blog_entries = BlogPage.objects.child_of(self).live()

        # Filter by tag
        tag = request.GET.get('tag')
        if tag:
            blog_entries = blog_entries.filter(tags__name=tag)

        context['blog_entries'] = blog_entries
        return context
```

Here, `blog_entries.filter(tags__name=tag)` follows the `tags` relation on `BlogPage`, to filter the listing to only those pages with a matching tag name before passing this to the template for rendering. We can now update the `blog_page.html` template to show a list of tags associated with the page, with links back to the filtered index page:

```
{% for tag in page.tags.all %}
    <a href="{% pageurl page.blog_index %}?tag={{ tag }}">{{ tag }}</a>
{% endfor %}
```

Iterating through `page.tags.all` will display each tag associated with `page`, while the links 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.

The same approach can be used to add tagging to non-page models managed through `Snippets` and `ModelAdmin`. In this case, the model must inherit from `modelcluster.models.ClusterableModel` to be compatible with `ClusterTaggableManager`.

Custom tag models

In the above example, any newly-created tags will be added to django-taggit’s default Tag model, which will be shared by all other models using the same recipe as well as Wagtail’s image and document models. In particular, this means that the autocomplete suggestions on tag fields will include tags previously added to other models. To avoid this, you can set up a custom tag model inheriting from TagBase, along with a ‘through’ model inheriting from ItemBase, which will provide an independent pool of tags for that page model.

```
from django.db import models
from modelcluster.contrib.taggit import ClusterTaggableManager
from modelcluster.fields import ParentalKey
from taggit.models import TagBase, ItemBase

class BlogTag(TagBase):
    class Meta:
        verbose_name = "blog tag"
        verbose_name_plural = "blog tags"

class TaggedBlog(ItemBase):
    tag = models.ForeignKey(
        BlogTag, related_name="tagged_blogs", on_delete=models.CASCADE
    )
    content_object = ParentalKey(
        to='demo.BlogPage',
        on_delete=models.CASCADE,
        related_name='tagged_items'
    )

class BlogPage(Page):
    ...
    tags = ClusterTaggableManager(through='demo.TaggedBlog', blank=True)
```

Within the admin, the tag field will automatically recognise the custom tag model being used, and will offer autocomplete suggestions taken from that tag model.

Disabling free tagging

By default, tag fields work on a “free tagging” basis: editors can enter anything into the field, and upon saving, any tag text not recognised as an existing tag will be created automatically. To disable this behaviour, and only allow editors to enter tags that already exist in the database, custom tag models accept a `free_tagging = False` option:

```
from taggit.models import TagBase
from wagtail.snippets.models import register_snippet

@register_snippet
class BlogTag(TagBase):
    free_tagging = False

    class Meta:
        verbose_name = "blog tag"
        verbose_name_plural = "blog tags"
```

Here we have registered `BlogTag` as a snippet, to provide an interface for administrators (and other users with the appropriate permissions) to manage the allowed set of tags. With the `free_tagging = False` option set, editors can no longer enter arbitrary text into the tag field, and must instead select existing tags from the autocomplete dropdown.

Managing tags with Wagtail's ModelAdmin

In order to manage all the tags used in a project, you can use the `ModelAdmin` to add the `Tag` model to the Wagtail admin. This will allow you to have a tag admin interface within the main menu in which you can add, edit or delete your tags.

Tags that are removed from a content don't get deleted from the `Tag` model and will still be shown in typeahead tag completion. So having a tag interface is a great way to completely get rid of tags you don't need.

To add the tag interface, add the following block of code to a `wagtail_hooks.py` file within any your project's apps:

```
from wagtail.contrib.modeladmin.options import ModelAdmin, modeladmin_register
from wagtail.admin.edit_handlers import FieldPanel
from taggit.models import Tag

class TagsModelAdmin(ModelAdmin):
    Tag.panels = [FieldPanel("name")]  # only show the name field
    model = Tag
    menu_label = "Tags"
    menu_icon = "tag"  # change as required
    menu_order = 200  # will put in 3rd place (000 being 1st, 100 2nd)
    list_display = ["name", "slug"]
    search_fields = ("name",)

modeladmin_register(TagsModelAdmin)
```

A `Tag` model has a `name` and `slug` required fields. If you decide to add a tag, it is recommended to only display the `name` field panel as the `slug` field is autofilled when the `name` field is filled and you don't need to enter the same name in both the fields.

Panel types

Built-in Fields and Choosers

Django's field types are automatically recognised and provided with an appropriate widget for input. Just define that field the normal Django way and pass the field name into `FieldPanel` when defining your panels. Wagtail will take care of the rest.

Here are some Wagtail-specific types that you might include as fields in your models.

FieldPanel

```
class wagtail.admin.panels.FieldPanel(field_name, classname=None, widget=None, heading='',
                                      disable_comments=False, permission=None)
```

This is the panel used for basic Django field types.

field_name

This is the name of the class property used in your model definition.

classname

This is a string of optional CSS classes given to the panel which are used in formatting and scripted interactivity.

The CSS class `title` can be used to give the field a larger text size, suitable for representing page titles and section headings.

widget (*optional*)

This parameter allows you to specify a [Django form widget](#) to use instead of the default widget for this field type.

heading (*optional*)

This allows you to override the heading for the panel, which will otherwise be set automatically using the form field's label (taken in turn from a model field's `verbose_name`).

disable_comments (*optional*)

This allows you to prevent a field level comment button showing for this panel if set to `True` (see [Commenting](#)).

permission (*optional*)

Allows a field to be selectively shown to users with sufficient permission. Accepts a permission codename such as `'myapp.change_blog_category'` - if the logged-in user does not have that permission, the field will be omitted from the form. See Django's documentation on [custom permissions](#) for details on how to set permissions up; alternatively, if you want to set a field as only available to superusers, you can use any arbitrary string (such as `'superuser'`) as the codename, since superusers automatically pass all permission tests.

StreamFieldPanel

```
class wagtail.admin.panels.StreamFieldPanel(field_name, classname=None, widget=None)
```

Deprecated; use `FieldPanel` instead.

Changed in version 3.0: `StreamFieldPanel` is no longer required for `StreamField`.

MultiFieldPanel

```
class wagtail.admin.panels.MultiFieldPanel(children, heading='', classname=None)
```

This panel condenses several `FieldPanel`s or choosers, from a list or tuple, under a single heading string.

children

A list or tuple of child panels

heading

A heading for the fields

InlinePanel

```
class wagtail.admin.panels.InlinePanel(relation_name, panels=None, classname='', heading='', label='',
                                         help_text='', min_num=None, max_num=None)
```

This panel allows for the creation of a “cluster” of related objects over a join to a separate model, such as a list of related links or slides to an image carousel.

This is a powerful but complex feature which will take some space to cover, so we’ll skip over it for now. For a full explanation on the usage of `InlinePanel`, see [Inline Panels and Model Clusters](#).

Collapsing InlinePanels to save space

Note that you can use `classname="collapsed"` to load the panel collapsed under its heading in order to save space in the Wagtail admin.

FieldRowPanel

```
class wagtail.admin.panels.FieldRowPanel(children, classname=None)
```

This panel creates a columnar layout in the editing interface, where each of the child Panels appears alongside each other rather than below.

Use of `FieldRowPanel` particularly helps reduce the “snow-blindness” effect of seeing so many fields on the page, for complex models. It also improves the perceived association between fields of a similar nature. For example if you created a model representing an “Event” which had a starting date and ending date, it may be intuitive to find the start and end date on the same “row”.

`children`

A list or tuple of child panels to display on the row

`classname`

A class to apply to the `FieldRowPanel` as a whole

HelpPanel

```
class wagtail.admin.panels.HelpPanel(content='', template='wagtailadmin/panels/help_panel.html',
                                         heading='', classname '')
```

`content`

HTML string that gets displayed in the panel.

`template`

Path to a template rendering the full panel HTML.

`heading`

A heading for the help content.

`classname`

String of CSS classes given to the panel which are used in formatting and scripted interactivity.

PageChooserPanel

```
class wagtail.admin.panels.PageChooserPanel(field_name, page_type=None, can_choose_root=False)
```

You can explicitly link [Page](#)-derived models together using the [Page](#) model and PageChooserPanel.

```
from wagtail.models import Page
from wagtail.admin.panels import PageChooserPanel

class BookPage(Page):
    related_page = models.ForeignKey(
        'wagtailcore.Page',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+',
    )

    content_panels = Page.content_panels + [
        PageChooserPanel('related_page', 'demo.PublisherPage'),
    ]
```

PageChooserPanel takes one required argument, the field name. Optionally, specifying a page type (in the form of an "appname.modelname" string) will filter the chooser to display only pages of that type. A list or tuple of page types can also be passed in, to allow choosing a page that matches any of those page types:

```
PageChooserPanel('related_page', ['demo.PublisherPage', 'demo.AuthorPage'])
```

Passing `can_choose_root=True` will allow the editor to choose the tree root as a page. Normally this would be undesirable, since the tree root is never a usable page, but in some specialised cases it may be appropriate; for example, a page with an automatic “related articles” feed could use a PageChooserPanel to select which subsection articles will be taken from, with the root corresponding to ‘everywhere’.

Changed in version 3.0: FieldPanel now also provides a page chooser interface for foreign keys to page models. PageChooserPanel is only required when specifying the `page_type` or `can_choose_root` parameters.

ImageChooserPanel

```
class wagtail.images.edit_handlers.ImageChooserPanel(field_name)
```

Deprecated; use FieldPanel instead.

Changed in version 3.0: ImageChooserPanel is no longer required to obtain an image chooser interface.

FormSubmissionsPanel

```
class wagtail.contrib.forms.panels.FormSubmissionsPanel
```

This panel adds a single, read-only section in the edit interface for pages implementing the `AbstractForm` model. It includes the number of total submissions for the given form and also a link to the listing of submissions.

```
from wagtail.contrib.forms.models import AbstractForm
from wagtail.contrib.forms.panels import FormSubmissionsPanel
```

(continues on next page)

(continued from previous page)

```
class ContactFormPage(AbstractForm):
    content_panels = [
        FormSubmissionsPanel(),
    ]
```

DocumentChooserPanel

```
class wagtail.documents.edit_handlers.DocumentChooserPanel(field_name)
```

Deprecated; use `FieldPanel` instead.

Changed in version 3.0: `DocumentChooserPanel` is no longer required to obtain a document chooser interface.

SnippetChooserPanel

```
class wagtail.snippets.edit_handlers.SnippetChooserPanel(field_name, snippet_type=None)
```

Deprecated; use `FieldPanel` instead.

Changed in version 3.0: `SnippetChooserPanel` is no longer required to obtain a document chooser interface.

Field Customisation

By adding CSS classes to your panel definitions or adding extra parameters to your field definitions, you can control much of how your fields will display in the Wagtail page editing interface. Wagtail's page editing interface takes much of its behaviour from Django's admin, so you may find many options for customisation covered there. (See [Django model field reference](#)).

Titles

Use `classname="title"` to make Page's built-in title field stand out with more vertical padding.

Collapsible

Changed in version 4.0: All panels are now collapsible by default.

Using `classname="collapsed"` will load the editor page with the panel collapsed under its heading.

```
content_panels = [
    MultiFieldPanel(
        [
            FieldPanel('cover'),
            FieldPanel('book_file'),
            FieldPanel('publisher'),
        ],
        heading="Collection of Book Fields",
        classname="collapsed"
    ),
]
```

Placeholder Text

By default, Wagtail uses the field's label as placeholder text. To change it, pass to the FieldPanel a widget with a placeholder attribute set to your desired text. You can select widgets from [Django's form widgets](#), or any of the Wagtail's widgets found in `wagtail.admin.widgets`.

For example, to customise placeholders for a Book model exposed via ModelAdmin:

```
# models.py
from django import forms           # the default Django widgets live here
from wagtail.admin import widgets   # to use Wagtail's special datetime widget

class Book(models.Model):
    title = models.CharField(max_length=256)
    release_date = models.DateField()
    price = models.DecimalField(max_digits=5, decimal_places=2)

    # you can create them separately
    title_widget = forms.TextInput(
        attrs = {
            'placeholder': 'Enter Full Title'
        }
    )
    # using the correct widget for your field type and desired effect
    date_widget = widgets.AdminDateInput(
        attrs = {
            'placeholder': 'dd-mm-yyyy'
        }
    )

    panels = [
        FieldPanel('title', widget=title_widget), # then add them as a variable
        FieldPanel('release_date', widget=date_widget),
        FieldPanel('price', widget=forms.NumberInput(attrs={'placeholder': 'Retail price',
→on release'})) # or directly inline
    ]
```

Required Fields

To make input or chooser selection mandatory for a field, add `blank=False` to its model definition.

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`.

Inline Panels and Model Clusters

The `django-modelcluster` module allows for streamlined relation of extra models to a Wagtail page via a `ForeignKey`-like relationship called `ParentalKey`. Normally, your related objects “cluster” would need to be created beforehand (or asynchronously) before being linked to a Page; however, objects related to a Wagtail page via `ParentalKey` can be created on-the-fly and saved to a draft revision of a Page object.

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.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', on_delete=models.CASCADE, related_name='related_links')

class BookPage(Page):
    # ...

    content_panels = Page.content_panels + [
        InlinePanel('related_links', heading="Related Links", label="Related link"),
    ]
```

The `RelatedLink` class is a vanilla Django abstract model. The `BookPageRelatedLinks` model extends it with capability for being ordered in the Wagtail interface via the `Orderable` class as well as adding a `page` property which links the model to the `BookPage` model we’re adding the related links objects to. Finally, in the panel definitions for `BookPage`, we’ll add an `InlinePanel` to provide an interface for it all. Let’s look again at the parameters that `InlinePanel` accepts:

```
InlinePanel(relation_name, panels=None, heading='', label='', help_text='', min_num=None,
→ max_num=None)
```

The `relation_name` is the `related_name` label given to the cluster’s `ParentalKey` relation. You can add the `panels` manually or make them part of the cluster model. `heading` and `help_text` provide a heading and caption, respectively, for the Wagtail editor. `label` sets the text on the add button and child panels, and is used as the heading when `heading` is not present. Finally, `min_num` and `max_num` allow you to set the minimum/maximum number of forms that the user must submit.

For another example of using model clusters, see [Tagging](#).

For more on `django-modelcluster`, visit the [django-modelcluster](#) github project page

Model Reference

`wagtail.models` is split into submodules for maintainability. All definitions intended as public should be imported here (with ‘noqa’ comments as required) and outside code should continue to import them from `wagtail.models` (e.g. `from wagtail.models import Site`, not `from wagtail.models.sites import Site`.)

Submodules should take care to keep the direction of dependencies consistent; where possible they should implement low-level generic functionality which is then imported by higher-level models such as `Page`.

This document contains reference information for the model classes inside the `wagtailcore` module.

Page

Database fields

`class wagtail.models.Page`

`title`

(text)

Human-readable title of the page.

`draft_title`

(text)

Human-readable title of the page, incorporating any changes that have been made in a draft edit (in contrast to the `title` field, which for published pages will be the title as it exists in the current published version).

`slug`

(text)

This is used for constructing the page’s URL.

For example: `http://domain.com/blog/[my-slug]/`

`content_type`

(foreign key to `django.contrib.contenttypes.models.ContentType`)

A foreign key to the `ContentType` object that represents the specific model of this page.

`live`

(boolean)

A boolean that is set to `True` if the page is published.

Note: this field defaults to `True` meaning that any pages that are created programmatically will be published by default.

`has_unpublished_changes`

(boolean)

A boolean that is set to `True` when the page is either in draft or published with draft changes.

owner

(foreign key to user model)

A foreign key to the user that created the page.

first_published_at

(date/time)

The date/time when the page was first published.

last_published_at

(date/time)

The date/time when the page was last published.

seo_title

(text)

Alternate SEO-crafted title, for use in the page's <title> HTML tag.

search_description

(text)

SEO-crafted description of the content, used for search indexing. This is also suitable for the page's <meta name="description"> HTML tag.

show_in_menus

(boolean)

Toggles whether the page should be included in site-wide menus, and is shown in the `promote_panels` within the Page editor.

Wagtail does not include any menu implementation by default, which means that this field will not do anything in the front facing content unless built that way in a specific Wagtail installation.

However, this is used by the `in_menu()` QuerySet filter to make it easier to query for pages that use this field.

Defaults to `False` and can be overridden on the model with `show_in_menus_default = True`.

Note: To set the global default for all pages, set `Page.show_in_menus_default = True` once where you first import the Page model.

locked

(boolean)

When set to `True`, the Wagtail editor will not allow any users to edit the content of the page.

If `locked_by` is also set, only that user can edit the page.

locked_by

(foreign key to user model)

The user who has currently locked the page. Only this user can edit the page.

If this is `None` when `locked` is `True`, nobody can edit the page.

locked_at

(date/time)

The date/time when the page was locked.

alias_of

(foreign key to another page)

If set, this page is an alias of the page referenced in this field.

locale

(foreign key to Locale)

This foreign key links to the `Locale` object that represents the page language.

translation_key

(uuid)

A UUID that is shared between translations of a page. These are randomly generated when a new page is created and copied when a translation of a page is made.

A `translation_key` value can only be used on one page in each locale.

Methods and properties

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.

Note: See also `django-treebeard`'s `node API <https://django-treebeard.readthedocs.io/en/latest/api.html>. `Page` is a subclass of `materialized path tree` nodes.

class wagtail.models.Page**get_specific(deferred=False, copy_attrs=None, copy_attrs_exclude=None)**

Return this page in its most specific subclassed form.

By default, a database query is made to fetch all field values for the specific object. If you only require access to custom methods or other non-field attributes on the specific object, you can use `deferred=True` to avoid this query. However, any attempts to access specific field values from the returned object will trigger additional database queries.

By default, references to all non-field attribute values are copied from current object to the returned one. This includes:

- Values set by a queryset, for example: annotations, or values set as a result of using `select_related()` or `prefetch_related()`.
- Any `cached_property` values that have been evaluated.
- Attributes set elsewhere in Python code.

For fine-grained control over which non-field values are copied to the returned object, you can use `copy_attrs` to specify a complete list of attribute names to include. Alternatively, you can use `copy_attrs_exclude` to specify a list of attribute names to exclude.

If called on a page object that is already an instance of the most specific class (e.g. an `EventPage`), the object will be returned as is, and no database queries or other operations will be triggered.

If the page was originally created using a page type that has since been removed from the codebase, a generic `Page` object will be returned (without any custom field values or other functionality present on the original class). Usually, deleting these pages is the best course of action, but there is currently no safe way for Wagtail to do that at migration time.

specific

Returns this page in its most specific subclassed form with all field values fetched from the database. The result is cached in memory.

specific_deferred

Returns this page in its most specific subclassed form without any additional field values being fetched from the database. The result is cached in memory.

specific_class

Return the class that this page would be if instantiated in its most specific form.

If the model class can no longer be found in the codebase, and the relevant `ContentType` has been removed by a database migration, the return value will be `None`.

If the model class can no longer be found in the codebase, but the relevant `ContentType` is still present in the database (usually a result of switching between git branches without running or reverting database migrations beforehand), the return value will be `None`.

cached_content_type

Return this page's `content_type` value from the `ContentType` model's cached manager, which will avoid a database query if the object is already in memory.

page_type_display_name

A human-readable version of this page's type

get_url(*request=None*, *current_site=None*)

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.

Accepts an optional but recommended `request` keyword argument that, if provided, will be used to cache site-level URL information (thereby avoiding repeated database / cache lookups) and, via the `Site.find_for_request()` function, determine whether a relative or full URL is most appropriate.

get_full_url(*request=None*)

Return the full URL (including protocol / domain) to this page, or `None` if it is not routable

full_url

Return the full URL (including protocol / domain) to this page, or `None` if it is not routable

relative_url(*current_site*, *request=None*)

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.

Accepts an optional but recommended `request` keyword argument that, if provided, will be used to cache site-level URL information (thereby avoiding repeated database / cache lookups).

get_site()

Return the `Site` object that this page belongs to.

get_url_parts(*request=None*)

Determine the URL for this page and return it as a tuple of `(site_id, site_root_url, page_url_relative_to_site_root)`. Return `None` if the page is not routable.

This is used internally by the `full_url`, `url`, `relative_url` and `get_site` properties and methods; pages with custom URL routing should override this method in order to have those operations return the custom URLs.

Accepts an optional keyword argument `request`, which may be used to avoid repeated database / cache lookups. Typically, a page model that overrides `get_url_parts` should not need to deal with `request` directly, and should just pass it to the original method when calling `super`.

route(`request`, `path_components`)

serve(`request`, *`args`, **`kwargs`)

context_object_name = None

Custom name for page instance in page's Context.

get_context(`request`, *`args`, **`kwargs`)

get_template(`request`, *`args`, **`kwargs`)

get_admin_display_title()

Return the title for this page as it should appear in the admin backend; override this if you wish to display extra contextual information about the page, such as language. By default, returns `draft_title`.

preview_modes

A list of (`internal_name`, `display_name`) tuples for the modes in which this object can be displayed for preview/moderation purposes. Ordinarily an object will only have one display mode, but subclasses can override this - for example, a page containing a form might have a default view of the form, and a post-submission 'thank you' page. Set to [] to completely disable previewing for this model.

serve_preview(`request`, `mode_name`)

Returns an HTTP response for use in object previews.

This method can be overridden to implement custom rendering and/or routing logic.

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/hidden. This request will always be a GET.

get_parent(`update=False`)

Returns the parent node of the current node object. Caches the result in the object itself to help in loops.

get_ancestors(`inclusive=False`)

Returns a queryset of the current page's ancestors, starting at the root page and descending to the parent, or to the current page itself if `inclusive` is true.

get_descendants(`inclusive=False`)

Returns a queryset of all pages underneath the current page, any number of levels deep. If `inclusive` is true, the current page itself is included in the queryset.

get_siblings(`inclusive=True`)

Returns a queryset of all other pages with the same parent as the current page. If `inclusive` is true, the current page itself is included in the queryset.

get_translations(`inclusive=False`)

Returns a queryset containing the translations of this instance.

get_translation(`locale`)

Finds the translation in the specified locale.

If there is no translation in that locale, this raises a `model.DoesNotExist` exception.

get_translation_or_none(locale)

Finds the translation in the specified locale.

If there is no translation in that locale, this returns None.

has_translation(locale)

Returns True if a translation exists in the specified locale.

copy_for_translation(locale, copy_parents=False, alias=False, exclude_fields=None)

Creates a copy of this page in the specified locale.

localized

Finds the translation in the current active language.

If there is no translation in the active language, self is returned.

Note: This will not return the translation if it is in draft. If you want to include drafts, use the .localized_draft attribute instead.

localized_draft

Finds the translation in the current active language.

If there is no translation in the active language, self is returned.

Note: This will return translations that are in draft. If you want to exclude these, use the .localized attribute.

search_fields

A list of fields to be indexed by the search engine. See Search docs [Indexing extra fields](#)

subpage_types

A list of page models which can be created as children of this page type. For example, a BlogIndex page might allow a BlogPage as a child, but not a JobPage:

```
class BlogIndex(Page):
    subpage_types = ['mysite.BlogPage', 'mysite.BlogArchivePage']
```

The creation of child pages can be blocked altogether for a given page by setting its subpage_types attribute to an empty array:

```
class BlogPage(Page):
    subpage_types = []
```

parent_page_types

A list of page models which are allowed as parent page types. For example, a BlogPage may only allow itself to be created below the BlogIndex page:

```
class BlogPage(Page):
    parent_page_types = ['mysite.BlogIndexPage']
```

Pages can block themselves from being created at all by setting parent_page_types to an empty array (this is useful for creating unique pages that should only be created once):

```
class HiddenPage(Page):
    parent_page_types = []
```

To allow for a page to be only created under the root page (for example for HomePage models) set the parent_page_type to ['wagtailcore.Page'].

```
class HomePage(Page):
    parent_page_types = ['wagtailcore.Page']
```

classmethod can_exist_under(parent)

Checks if this page type can exist as a subpage under a parent page instance.

See also: [Page.can_create_at\(\)](#) and [Page.can_move_to\(\)](#)

classmethod can_create_at(parent)

Checks if this page type can be created as a subpage under a parent page instance.

can_move_to(parent)

Checks if this page instance can be moved to be a subpage of a parent page instance.

get_route_paths()

New in version 2.16.

Returns a list of paths that this page can be viewed at.

These values are combined with the dynamic portion of the page URL to automatically create redirects when the page’s URL changes.

Note: If using `RoutablePageMixin`, you may want to override this method to include the paths of popular routes.

Note: Redirect paths are ‘normalized’ to apply consistent ordering to GET parameters, so you don’t need to include every variation. Fragment identifiers are discarded too, so should be avoided.

password_required_template

Defines which template file should be used to render the login form for Protected pages using this model. This overrides the default, defined using `PASSWORD_REQUIRED_TEMPLATE` in your settings. See [Private pages](#)

is_creatable

Controls if this page can be created through the Wagtail administration. Defaults to `True`, and is not inherited by subclasses. This is useful when using [multi-table inheritance](#), to stop the base model from being created as an actual page.

max_count

Controls the maximum number of pages of this type that can be created through the Wagtail administration interface. This is useful when needing “allow at most 3 of these pages to exist”, or for singleton pages.

max_count_per_parent

Controls the maximum number of pages of this type that can be created under any one parent page.

exclude_fields_in_copy

An array of field names that will not be included when a Page is copied. Useful when you have relations that do not use `ClusterableModel` or should not be copied.

```
class BlogPage(Page):
    exclude_fields_in_copy = ['special_relation', 'custom_uuid']
```

The following fields will always be excluded in a copy - `['id', 'path', 'depth', 'numchild', 'url_path', 'path']`.

`base_form_class`

The form class used as a base for editing Pages of this type in the Wagtail page editor. This attribute can be set on a model to customise the Page editor form. Forms must be a subclass of [WagtailAdminPageForm](#). See [Customising generated forms](#) for more information.

`with_content_json(content)`

Returns a new version of the page with field values updated to reflect changes in the provided `content` (which usually comes from a previously-saved page revision).

Certain field values are preserved in order to prevent errors if the returned page is saved, such as `id`, `content_type` and some tree-related values. The following field values are also preserved, as they are considered to be meaningful to the page as a whole, rather than to a specific revision:

- `draft_title`
- `live`
- `has_unpublished_changes`
- `owner`
- `locked`
- `locked_by`
- `locked_at`
- `latest_revision`
- `latest_revision_created_at`
- `first_published_at`
- `alias_of`
- `wagtail_admin_comments (COMMENTS_RELATION_NAME)`

`save(clean=True, user=None, log_action=False, **kwargs)`

Overrides default method behaviour to make additional updates unique to pages, such as updating the `url_path` value of descendant page to reflect changes to this page's slug.

New pages should generally be saved via the `add_child()` or `add_sibling()` method of an existing page, which will correctly set the `path` and `depth` fields on the new page before saving it.

By default, pages are validated using `full_clean()` before attempting to save changes to the database, which helps to preserve validity when restoring pages from historic revisions (which might not necessarily reflect the current model state). This validation step can be bypassed by calling the method with `clean=False`.

`create_alias(*, recursive=False, parent=None, update_slug=None, update_locale=None, user=None, log_action='wagtail.create_alias', reset_translation_key=True, _mpnode_attrs=None)`

`update_aliases(*, revision=None, user=None, _content=None, _updated_ids=None)`

Publishes all aliases that follow this page with the latest content from this page.

This is called by Wagtail whenever a page with aliases is published.

Parameters

- **revision (Revision, optional)** – The revision of the original page that we are updating to (used for logging purposes).
- **user (User, optional)** – The user who is publishing (used for logging purposes).

has_workflow

Returns True if the page or an ancestor has an active workflow assigned, otherwise False

get_workflow()

Returns the active workflow assigned to the page or its nearest ancestor

workflow_in_progress

Returns True if a workflow is in progress on the current page, otherwise False

current_workflow_state

Returns the in progress or needs changes workflow state on this page, if it exists

current_workflow_task_state

Returns (specific class of) the current task state of the workflow on this page, if it exists

current_workflow_task

Returns (specific class of) the current task in progress on this page, if it exists

Site

The **Site** model is useful for multi-site installations as it allows an administrator to configure which part of the tree to use for each hostname that the server responds on.

The [`find_for_request\(\)`](#) function returns the Site object that will handle the given HTTP request.

Database fields

class wagtail.models.Site**hostname**

(text)

This is the hostname of the site, excluding the scheme, port and path.

For example: `www.mysite.com`

Note: If you're looking for how to get the root url of a site, use the [`root_url`](#) attribute.

port

(number)

This is the port number that the site responds on.

site_name

(text - optional)

A human-readable name for the site. This is not used by Wagtail itself, but is suitable for use on the site front-end, such as in <title> elements.

For example: Rod's World of Birds

root_page

(foreign key to [`Page`](#))

This is a link to the root page of the site. This page will be what appears at the / URL on the site and would usually be a homepage.

is_default_site

(boolean)

This is set to True if the site is the default. Only one site can be the default.

The default site is used as a fallback in situations where a site with the required hostname/port couldn't be found.

Methods and properties

class wagtail.models.Site

static find_for_request(request)

Find the site object responsible for responding to this HTTP request object. Try:

- unique hostname first
- then hostname and port
- if there is no matching hostname at all, or no matching hostname:port combination, fall back to the unique default site, or raise an exception

NB this means that high-numbered ports on an extant hostname may still be routed to a different hostname which is set as the default

The site will be cached via `request._wagtail_site`

root_url

This returns the URL of the site. It is calculated from the `hostname` and the `port` fields.

The scheme part of the URL is calculated based on value of the `port` field:

- 80 = `http://`
- 443 = `https://`
- Everything else will use the `http://` scheme and the port will be appended to the end of the hostname (for example `http://mysite.com:8000/`)

static get_site_root_paths()

Return a list of `SiteRootPath` instances, most specific path first - used to translate `url_paths` into actual URLs with hostnames

Each root path is an instance of the `SiteRootPath` named tuple, and have the following attributes:

- `site_id` - The ID of the Site record
- `root_path` - The internal URL path of the site's home page (for example '/home/')
- `root_url` - The scheme/domain name of the site (for example '`https://www.example.com/`')
- `language_code` - The language code of the site (for example 'en')

Locale

The `Locale` model defines the set of languages and/or locales that can be used on a site. Each `Locale` record corresponds to a “language code” defined in the `:ref:wagtail_content_languages_setting` setting.

Wagtail will initially set up one `Locale` to act as the default language for all existing content. This first locale will automatically pick the value from `WAGTAIL_CONTENT_LANGUAGES` that most closely matches the site primary language code defined in `LANGUAGE_CODE`. If the primary language code is changed later, Wagtail will **not** automatically create a new `Locale` record or update an existing one.

Before internationalisation is enabled, all pages use this primary `Locale` record. This is to satisfy the database constraints, and makes it easier to switch internationalisation on at a later date.

Changing `WAGTAIL_CONTENT_LANGUAGES`

Languages can be added or removed from `WAGTAIL_CONTENT_LANGUAGES` over time.

Before removing an option from `WAGTAIL_CONTENT_LANGUAGES`, it’s important that the `Locale` record is updated to a use a different content language or is deleted. Any `Locale` instances that have invalid content languages are automatically filtered out from all database queries making them unable to be edited or viewed.

Methods and properties

```
class wagtail.models.Locale
```

`language_code`

The language code that represents this locale

The language code can either be a language code on its own (such as `en`, `fr`), or it can include a region code (such as `en-gb`, `fr-fr`).

`classmethod get_default()`

Returns the default `Locale` based on the site’s `LANGUAGE_CODE` setting

`classmethod get_active()`

Returns the `Locale` that corresponds to the currently activated language in Django.

`get_display_name()`

`TranslatableMixin`

`TranslatableMixin` is an abstract model that can be added to any non-page Django model to make it translatable. Pages already include this mixin, so there is no need to add it.

Database fields

The `locale` and `translation_key` fields have a unique key constraint to prevent the object being translated into a language more than once.

```
class wagtail.models.TranslatableMixin
```

`locale`

(Foreign Key to `Locale`)

For pages, this defaults to the locale of the parent page.

`translation_key`

(uuid)

A UUID that is randomly generated whenever a new model instance is created. This is shared with all translations of that instance so can be used for querying translations.

Methods and properties

```
class wagtail.models.TranslatableMixin
```

`get_translations(inclusive=False)`

Returns a queryset containing the translations of this instance.

`get_translation(locale)`

Finds the translation in the specified locale.

If there is no translation in that locale, this raises a `model.DoesNotExist` exception.

`get_translation_or_none(locale)`

Finds the translation in the specified locale.

If there is no translation in that locale, this returns `None`.

`has_translation(locale)`

Returns `True` if a translation exists in the specified locale.

`copy_for_translation(locale)`

Creates a copy of this instance with the specified locale.

Note that the copy is initially unsaved.

`classmethod get_translation_model()`

Returns this model's "Translation model".

The "Translation model" is the model that has the `locale` and `translation_key` fields. Typically this would be the current model, but it may be a super-class if multi-table inheritance is in use (as is the case for `wagtailcore.Page`).

`localized`

Finds the translation in the current active language.

If there is no translation in the active language, `self` is returned.

Note: This will not return the translation if it is in draft. If you want to include drafts, use the `.localized_draft` attribute instead.

PreviewableMixin

`PreviewableMixin` is a mixin class that can be added to any non-page Django model to allow previewing its instances. Pages already include this mixin, so there is no need to add it.

New in version 4.0: The class is added to allow snippets to have live preview in the editor. See [Making snippets previewable](#) for more details.

Methods and properties

`class wagtail.models.PreviewableMixin`

`preview_modes`

A list of (`internal_name`, `display_name`) tuples for the modes in which this object can be displayed for preview/moderation purposes. Ordinarily an object will only have one display mode, but subclasses can override this - for example, a page containing a form might have a default view of the form, and a post-submission ‘thank you’ page. Set to `[]` to completely disable previewing for this model.

`default_preview_mode`

The default preview mode to use in live preview. This default is also used in areas that do not give the user the option of selecting a mode explicitly, e.g. in the moderator approval workflow. If `preview_modes` is empty, an `IndexError` will be raised.

`is_previewable()`

Returns `True` if at least one preview mode is specified in `preview_modes`.

`get_preview_context(request, mode_name)`

Returns a context dictionary for use in templates for previewing this object.

`get_preview_template(request, mode_name)`

Returns a template to be used when previewing this object.

Subclasses of `PreviewableMixin` must override this method to return the template name to be used in the preview. Alternatively, subclasses can also override the `serve_preview` method to completely customise the preview rendering logic.

`serve_preview(request, mode_name)`

Returns an HTTP response for use in object previews.

This method can be overridden to implement custom rendering and/or routing logic.

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/hidden. This request will always be a GET.

RevisionMixin

`RevisionMixin` is an abstract model that can be added to any non-page Django model to allow saving revisions of its instances. Pages already include this mixin, so there is no need to add it.

New in version 4.0: The model is added to allow snippets to save revisions, revert to a previous revision, and compare changes between revisions. See [Saving revisions of snippets](#) for more details.

Database fields

```
class wagtail.models.RevisionMixin
```

latest_revision

(foreign key to [Revision](#))

This points to the latest revision created for the object. This reference is stored in the database for performance optimisation.

Methods and properties

```
class wagtail.models.RevisionMixin
```

revisions

Returns revisions that belong to the object.

Subclasses should define a [GenericRelation](#) to [Revision](#) and override this property to return that [GenericRelation](#). This allows subclasses to customise the `related_query_name` of the [GenericRelation](#) and add custom logic (e.g. to always use the specific instance in [Page](#)).

```
save_revision(user=None, submitted_for_moderation=False, approved_go_live_at=None, changed=True,  
log_action=False, previous_revision=None, clean=True)
```

Creates and saves a revision.

Parameters

- **user** – The user performing the action.
- **submitted_for_moderation** – Indicates whether the object was submitted for moderation.
- **approved_go_live_at** – The date and time the revision is approved to go live.
- **changed** – Indicates whether there were any content changes.
- **log_action** – Flag for logging the action. Pass `False` to skip logging. Can be passed an action string. Defaults to "wagtail.edit" when no `previous_revision` param is passed, otherwise "wagtail.revert".
- **previous_revision** ([Revision](#)) – Indicates a revision reversal. Should be set to the previous revision instance.
- **clean** – Set this to `False` to skip cleaning object content before saving this revision.

Returns The newly created revision.

```
get_latest_revision_as_object()
```

Returns the latest revision of the object as an instance of the model. If no latest revision exists, returns the object itself.

```
with_content_json(content)
```

Returns a new version of the object with field values updated to reflect changes in the provided `content` (which usually comes from a previously-saved revision).

Certain field values are preserved in order to prevent errors if the returned object is saved, such as `id`. The following field values are also preserved, as they are considered to be meaningful to the object as a whole, rather than to a specific revision:

- `latest_revision`

If `TranslatableMixin` is applied, the following field values are also preserved:

- `translation_key`
- `locale`

DraftStateMixin

`DraftStateMixin` is an abstract model that can be added to any non-page Django model to allow its instances to have unpublished changes. This mixin requires `RevisionMixin` to be applied. Pages already include this mixin, so there is no need to add it.

New in version 4.0: The model is added to allow snippets to have changes that are not immediately reflected to the instance. See [Saving draft changes of snippets](#) for more details.

Database fields

```
class wagtail.models.DraftStateMixin
```

live

(boolean)

A boolean that is set to True if the object is published.

Note: this field defaults to True meaning that any objects that are created programmatically will be published by default.

live_revision

(foreign key to `Revision`)

This points to the revision that is currently live.

has_unpublished_changes

(boolean)

A boolean that is set to True when the object is either in draft or published with draft changes.

first_published_at

(date/time)

The date/time when the object was first published.

last_published_at

(date/time)

The date/time when the object was last published.

Methods and properties

```
class wagtail.models.DraftStateMixin
```

publish(revision, user=None, changed=True, log_action=True, previous_revision=None)

Publish a revision of the object by applying the changes in the revision to the live object.

Parameters

- `revision` (`Revision`) – Revision to publish.

- **user** – The publishing user.
- **changed** – Indicated whether content has changed.
- **log_action** – Flag for the logging action, pass `False` to skip logging.
- **previous_revision** (`Revision`) – Indicates a revision reversal. Should be set to the previous revision instance.

`unpublish(set_expired=False, commit=True, user=None, log_action=True)`

Unpublish the live object.

Parameters

- **set_expired** – Mark the object as expired.
- **commit** – Commit the changes to the database.
- **user** – The unpublishing user.
- **log_action** – Flag for the logging action, pass `False` to skip logging.

`with_content_json(content)`

Similar to `RevisionMixin.with_content_json()`, but with the following fields also preserved:

- `live`
- `has_unpublished_changes`
- `first_published_at`

Revision

Every time a page is edited, a new `Revision` is created and saved to the database. It can be used to find the full history of all changes that have been made to a page and it also provides a place for new changes to be kept before going live.

- Revisions can be created from any instance of `RevisionMixin` by calling its `save_revision()` method.
- The content of the page is JSON-serialisable and stored in the `content` field.
- You can retrieve a `Revision` as an instance of the object's model by calling the `as_object()` method.

Changed in version 4.0: The model has been renamed from `PageRevision` to `Revision` and it now references the `Page` model using a `GenericForeignKey`.

Database fields

`class wagtail.models.Revision`

`content_object`

(generic foreign key)

The object this revision belongs to. For page revisions, the object is an instance of the specific class.

`content_type`

(foreign key to `ContentType`)

The content type of the object this revision belongs to. For page revisions, this means the content type of the specific page type.

base_content_type(foreign key to [ContentType](#))

The base content type of the object this revision belongs to. For page revisions, this means the content type of the [Page](#) model.

object_id

(string)

The primary key of the object this revision belongs to.

submitted_for_moderation

(boolean)

True if this revision is in moderation.

created_at

(date/time)

The time the revision was created.

user

(foreign key to user model)

The user that created the revision.

content

(dict)

The JSON content for the object at the time the revision was created.

Changed in version 3.0: The field has been renamed from `content_json` to `content` and it now uses `JSONField` instead of `TextField`.

Managers

class wagtail.models.Revision**objects**

This default manager is used to retrieve all of the Revision objects in the database. It also provides a `for_instance()` method that lets you query for revisions of a specific object.

Example:

```
Revision.objects.all()
Revision.objects.for_instance(my_object)
```

page_revisions

This manager extends the default manager and is used to retrieve all of the Revision objects that belong to pages.

Example:

```
Revision.page_revisions.all()
```

New in version 4.0: This manager is added as a shorthand to retrieve page revisions.

submitted_revisions

This manager extends the default manager and is used to retrieve all of the Revision objects that are awaiting moderator approval.

Example:

```
Revision.submitted_revisions.all()
```

Methods and properties

```
class wagtail.models.Revision
```

as_object()

This method retrieves this revision as an instance of its object's specific class. If the revision belongs to a page, it will be an instance of the [Page](#)'s specific subclass.

Changed in version 4.0: This method has been renamed from `as_page_object()` to `as_object()`.

approve_moderation(user=None)

Calling this on a revision that's in moderation will mark it as approved and publish it.

reject_moderation(user=None)

Calling this on a revision that's in moderation will mark it as rejected.

is_latest_revision()

Returns True if this revision is the object's latest revision.

publish(user=None, changed=True, log_action=True, previous_revision=None)

Calling this will copy the content of this revision into the live object. If the object is in draft, it will be published.

base_content_object

This property returns the object this revision belongs to as an instance of the base class.

GroupPagePermission

Database fields

```
class wagtail.models.GroupPagePermission
```

group

(foreign key to `django.contrib.auth.models.Group`)

page

(foreign key to [Page](#))

permission_type

(choice list)

PageViewRestriction

Database fields

```
class wagtail.models.PageViewRestriction

    page
        (foreign key to Page)
    password
        (text)
```

Orderable (abstract)

Database fields

```
class wagtail.models.Orderable

    sort_order
        (number)
```

Workflow

Workflows represent sequences of tasks which must be approved for an action to be performed on a page - typically publication.

Database fields

```
class wagtail.models.Workflow

    name
        (text)
        Human-readable name of the workflow.

    active
        (boolean)
        Whether or not the workflow is active: active workflows can be added to pages, and started. Inactive workflows cannot.
```

Methods and properties

```
class wagtail.models.Workflow

    start(page, user)
        Initiates a workflow by creating an instance of WorkflowState

    tasks
        Returns all Task instances linked to this workflow
```

deactivate(*user=None*)

Sets the workflow as inactive, and cancels all in progress instances of `WorkflowState` linked to this workflow

all_pages()

Returns a queryset of all the pages that this Workflow applies to.

WorkflowState

Workflow states represent the status of a started workflow on a page.

Database fields

class wagtail.models.WorkflowState

page

(foreign key to `Page`)

The page on which the workflow has been started

workflow

(foreign key to `Workflow`)

The workflow whose state the `WorkflowState` represents

status

(text)

The current status of the workflow (options are `WorkflowState.STATUS_CHOICES`)

created_at

(date/time)

When this instance of `WorkflowState` was created - when the workflow was started

requested_by

(foreign key to user model)

The user who started this workflow

current_task_state

(foreign key to `TaskState`)

The `TaskState` model for the task the workflow is currently at: either completing (if in progress) or the final task state (if finished)

Methods and properties

class wagtail.models.WorkflowState

STATUS_CHOICES

A tuple of the possible options for the `status` field, and their verbose names. Options are `STATUS_IN_PROGRESS`, `STATUS_APPROVED`, `STATUS_CANCELLED` and `STATUS_NEEDS_CHANGES`.

update(user=None, next_task=None)

Checks the status of the current task, and progresses (or ends) the workflow if appropriate. If the workflow progresses, next_task will be used to start a specific task next if provided.

get_next_task()

Returns the next active task, which has not been either approved or skipped

cancel(user=None)

Cancels the workflow state

finish(user=None)

Finishes a successful in progress workflow, marking it as approved and performing the on_finish action

resume(user=None)

Put a STATUS_NEEDS_CHANGES workflow state back into STATUS_IN_PROGRESS, and restart the current task

copy_approved_task_states_to_revision(revision)

This creates copies of previously approved task states with page_revision set to a different revision.

all_tasks_with_status()

Returns a list of Task objects that are linked with this workflow state's workflow. The status of that task in this workflow state is annotated in the .status field. And a displayable version of that status is annotated in the .status_display field.

This is different to querying TaskState as it also returns tasks that haven't been started yet (so won't have a TaskState).

revisions()

Returns all page revisions associated with task states linked to the current workflow state

Task

Tasks represent stages in a workflow which must be approved for the workflow to complete successfully.

Database fields

class wagtail.models.Task**name**

(text)

Human-readable name of the task.

active

(boolean)

Whether or not the task is active: active workflows can be added to workflows, and started. Inactive workflows cannot, and are skipped when in an existing workflow.

content_type

(foreign key to `django.contrib.contenttypes.models.ContentType`)

A foreign key to the `ContentType` object that represents the specific model of this task.

Methods and properties

`class wagtail.models.Task`

`workflows`

Returns all Workflow instances that use this task

`active_workflows`

Return a QuerySet` of active workflows that this task is part of

`task_state_class`

The specific task state class to generate to store state information for this task. If not specified, this will be TaskState.

`classmethod get_verbose_name()`

Returns the human-readable “verbose name” of this task model e.g “Group approval task”.

`specific`

Return this Task in its most specific subclassed form.

`start(workflow_state, user=None)`

Start this task on the provided workflow state by creating an instance of TaskState

`on_action(task_state, user, action_name, **kwargs)`

Performs an action on a task state determined by the `action_name` string passed

`user_can_access_editor(page, user)`

Returns True if a user who would not normally be able to access the editor for the page should be able to if the page is currently on this task. Note that returning False does not remove permissions from users who would otherwise have them.

`user_can_lock(page, user)`

Returns True if a user who would not normally be able to lock the page should be able to if the page is currently on this task. Note that returning False does not remove permissions from users who would otherwise have them.

`user_can_unlock(page, user)`

Returns True if a user who would not normally be able to unlock the page should be able to if the page is currently on this task. Note that returning False does not remove permissions from users who would otherwise have them.

`page_locked_for_user(page, user)`

Returns True if the page should be locked to a given user’s edits. This can be used to prevent editing by non-reviewers.

`get_actions(page, user)`

Get the list of action strings (name, verbose_name, whether the action requires additional data - see `get_form_for_action`) for actions the current user can perform for this task on the given page. These strings should be the same as those able to be passed to `on_action`

`get_task_states_user_can_moderate(user, **kwargs)`

Returns a QuerySet of the task states the current user can moderate

`deactivate(user=None)`

Set `active` to False and cancel all in progress task states linked to this task

`get_form_for_action(action)`

```
get_template_for_action(action)
classmethod get_description()
    Returns the task description.
```

TaskState

Task states store state information about the progress of a task on a particular page revision.

Database fields

```
class wagtail.models.TaskState
```

workflow_state

(foreign key to WorkflowState)

The workflow state which started this task state.

page_revision

(foreign key to Revision)

The page revision this task state was created on.

task

(foreign key to Task)

The task that this task state is storing state information for.

status

(text)

The completion status of the task on this revision. Options are available in TaskState.STATUS_CHOICES

content_type

(foreign key to django.contrib.contenttypes.models.ContentType)

A foreign key to the [ContentType](#) object that represents the specific model of this task.

started_at

(date/time)

When this task state was created.

finished_at

(date/time)

When this task state was cancelled, rejected, or approved.

finished_by

(foreign key to user model)

The user who completed (cancelled, rejected, approved) the task.

comment

(text)

A text comment, typically added by a user when the task is completed.

Methods and properties

`class wagtail.models.TaskState`

`STATUS_CHOICES`

A tuple of the possible options for the `status` field, and their verbose names. Options are `STATUS_IN_PROGRESS`, `STATUS_APPROVED`, `STATUS_CANCELLED`, `STATUS_REJECTED` and `STATUS_SKIPPED`.

`exclude_fields_in_copy`

A list of fields not to copy when the `TaskState.copy()` method is called.

`specific`

Return this `TaskState` in its most specific subclassed form.

`approve(user=None, update=True, comment='')`

Approve the task state and update the workflow state

`reject(user=None, update=True, comment='')`

Reject the task state and update the workflow state

`task_type_started_at`

Finds the first chronological `started_at` for successive `TaskStates` - ie `started_at` if the task had not been restarted

`cancel(user=None, resume=False, comment='')`

Cancel the task state and update the workflow state. If `resume` is set to True, then upon update the workflow state is passed the current task as `next_task`, causing it to start a new task state on the current task if possible

`copy(update_attrs=None, exclude_fields=None)`

Copy this task state, excluding the attributes in the `exclude_fields` list and updating any attributes to values specified in the `update_attrs` dictionary of attribute: new value pairs

`get_comment()`

Returns a string that is displayed in workflow history.

This could be a comment by the reviewer, or generated. Use `mark_safe` to return HTML.

WorkflowTask

Represents the ordering of a task in a specific workflow.

Database fields

`class wagtail.models.WorkflowTask`

`workflow`

(foreign key to `Workflow`)

`task`

(foreign key to `Task`)

sort_order

(number)

The ordering of the task in the workflow.

WorkflowPage

Represents the assignment of a workflow to a page and its descendants.

Database fields

```
class wagtail.models.WorkflowPage
    workflow
        (foreign key to Workflow)
    page
        (foreign key to Page)
```

BaseLogEntry

An abstract base class that represents a record of an action performed on an object.

Database fields

```
class wagtail.models.BaseLogEntry
    content_type
        (foreign key to django.contrib.contenttypes.models.ContentType)
        A foreign key to the ContentType object that represents the specific model of this model.
    label
        (text)
        The object title at the time of the entry creation
        Note: Wagtail will attempt to use get_admin_display_title or the string representation of the object passed to log_action()
    user
        (foreign key to user model)
        A foreign key to the user that triggered the action.
    revision
        (foreign key to Revision)
        A foreign key to the current revision.
```

data

(dict)

The JSON representation of any additional details for each action. For example source page id and title when copying from a page. Or workflow id/name and next step id/name on a workflow transition

Changed in version 3.0: The field has been renamed from `data_json` to `data` and it now uses `JSONField` instead of `TextField`.

timestamp

(date/time)

The date/time when the entry was created.

content_changed

(boolean)

A boolean that can set to True when the content has changed.

deleted

(boolean)

A boolean that can set to True when the object is deleted. Used to filter entries in the Site History report.

Methods and properties

`class wagtail.models.BaseLogEntry`

user_display_name

Returns the display name of the associated user; `get_full_name` if available and non-empty, otherwise `get_username`. Defaults to ‘system’ when none is provided

comment

object_verbose_name

object_id()

`PageLogEntry`

Represents a record of an action performed on an `Page`, subclasses `BaseLogEntry`.

Database fields

`class wagtail.models.PageLogEntry`

page

(foreign key to `Page`)

A foreign key to the page the action is performed on.

Comment

Represents a comment on a page.

Database fields

```
class wagtail.models.Comment
```

page

(parental key to *Page*)

A parental key to the page the comment has been added to.

user

(foreign key to user model)

A foreign key to the user who added this comment.

text

(text)

The text content of the comment.

contentpath

(text)

The path to the field or streamfield block the comment is attached to, in the form `field.streamfield_block_id`.

position

(text)

An identifier for the position of the comment within its field. The format used is determined by the field.

created_at

(date/time)

The date/time when the comment was created.

updated_at

(date/time)

The date/time when the comment was updated.

revision_created

(foreign key to *Revision*)

A foreign key to the revision on which the comment was created.

resolved_at

(date/time)

The date/time when the comment was resolved, if any.

resolved_by

(foreign key to user model)

A foreign key to the user who resolved this comment, if any.

CommentReply

Represents a reply to a comment thread.

Database fields

```
class wagtail.models.CommentReply
```

comment

(parental key to *Comment*)

A parental key to the comment that started the thread.

user

(foreign key to user model)

A foreign key to the user who added this comment.

text

(text)

The text content of the comment.

created_at

(date/time)

The date/time when the comment was created.

updated_at

(date/time)

The date/time when the comment was updated.

PageSubscription

Represents a user's subscription to email notifications about page events. Currently only used for comment notifications.

Database fields

```
class wagtail.models.PageSubscription
```

page

(parental key to *Page*)

user

(foreign key to user model)

comment_notifications

(boolean)

Whether the user should receive comment notifications for all comments, or just comments in threads they participate in.

Page QuerySet reference

All models that inherit from `Page` are given some extra QuerySet methods accessible from their `.objects` attribute.

Examples

Selecting only live pages

```
live_pages = Page.objects.live()
```

Selecting published EventPages that are descendants of events_index

```
events = EventPage.objects.live().descendant_of(events_index)
```

Getting a list of menu items

```
# This gets a QuerySet of live children of the homepage with ``show_in_menus`` set
menu_items = homepage.get_children().live().in_menu()
```

Reference

```
class wagtail.query.PageQuerySet(*args, **kwargs)
```

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
```

`not_in_menu()`

This filters the QuerySet to only contain pages that are not in the menus.

`in_site(site)`

This filters the QuerySet to only contain pages within the specified site.

Example:

```
# Get all the EventPages in the current site
site = Site.find_for_request(request)
site_events = EventPage.objects.in_site(site)
```

`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()
```

not_child_of(*other*)

This filters the QuerySet to not contain any pages that are direct children of the specified page.

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)
```

parent_of(*other*)

This filters the QuerySet to only contain the parent of the specified page.

not_parent_of(*other*)

This filters the QuerySet to exclude the parent of the specified page.

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()
```

not_sibling_of(*other*, *inclusive=True*)

This filters the QuerySet to not contain any pages that are siblings of the specified page.

By default, inclusive is set to True so it will exclude the specified page from the results.

If inclusive is set to False, the page will be included in the results.

public()

Filters the QuerySet to only contain pages that are not in a private section and their descendants.

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()
```

not_public()

Filters the QuerySet to only contain pages that are in a private section and their descendants.

private()

Filters the QuerySet to only contain pages that are in a private section and their descendants.

New in version 4.1: The `private` method was added as an alias of `not_public`.

search(*query*, *fields=None*, *operator=None*, *order_by_relevance=True*, *partial_match=True*, *backend='default'*)

This runs a search query on all the items in the QuerySet

See: [Searching QuerySets](#)

Example:

```
# Search future events
results = EventPage.objects.live().filter(date__gt=timezone.now()).search("Hello
←")
```

type(types*)**

This filters the QuerySet to only contain pages that are an instance of the specified model(s) (including subclasses).

Example:

```
# Find all pages that are of type AbstractEmailForm, or one of its subclasses
form_pages = Page.objects.type(AbstractEmailForm)

# Find all pages that are of type AbstractEmailForm or AbstractEventPage, or
# one of their subclasses
form_and_event_pages = Page.objects.type(AbstractEmailForm, AbstractEventPage)
```

not_type(types*)**

This filters the QuerySet to exclude any pages which are an instance of the specified model(s).

exact_type(*types)

This filters the QuerySet to only contain pages that are an instance of the specified model(s) (matching the model exactly, not subclasses).

Example:

```
# Find all pages that are of the exact type EventPage
event_pages = Page.objects.exact_type(EventPage)

# Find all page of the exact type EventPage or NewsPage
news_and_events_pages = Page.objects.exact_type(EventPage, NewsPage)
```

Note: If you are only interested in pages of a single type, it is clearer (and often more efficient) to use the specific model's manager to get a queryset. For example:

```
event_pages = EventPage.objects.all()
```

not_exact_type(*types)

This filters the QuerySet to exclude any pages which are an instance of the specified model(s) (matching the model exactly, not subclasses).

Example:

```
# First, find all news and event pages
news_and_events = Page.objects.type(NewsPage, EventPage)

# Now exclude pages with an exact type of EventPage or NewsPage,
# leaving only instance of more 'specialist' types
specialised_news_and_events = news_and_events.not_exact_type(NewsPage, EventPage)
```

unpublish()

This unpublishes all live pages in the QuerySet.

Example:

```
# Unpublish current_page and all of its children
Page.objects.descendant_of(current_page, inclusive=True).unpublish()
```

specific(defer=False)

This efficiently gets all the specific pages for the queryset, using the minimum number of queries.

When the “defer” keyword argument is set to True, only generic page field values will be loaded and all specific fields will be deferred.

Example:

```
# Get the specific instance of all children of the homepage,
# in a minimum number of database queries.
homepage.get_children().specific()
```

See also: [Page.specific](#)

defer_streamfields()

Apply to a queryset to prevent fetching/decoding of StreamField values on evaluation. Useful when working with potentially large numbers of results, where StreamField values are unlikely to be needed. For example, when generating a sitemap or a long list of page links.

Example:

```
# Apply to a queryset to avoid fetching StreamField values
# for a specific model
EventPage.objects.all().defer_streamfields()

# Or combine with specific() to avoid fetching StreamField
# values for all models
homepage.get_children().defer_streamfields().specific()
```

first_common_ancestor(include_self=False, strict=False)

Find the first ancestor that all pages in this queryset have in common. For example, consider a page hierarchy like:

```
- Home/
  - Foo Event Index/
    - Foo Event Page 1/
    - Foo Event Page 2/
  - Bar Event Index/
    - Bar Event Page 1/
    - Bar Event Page 2/
```

The common ancestors for some queries would be:

```
>>> Page.objects\
...     .type(EventPage) \
...     .first_common_ancestor()
<Page: Home>
>>> Page.objects\
...     .type(EventPage) \
...     .filter(title__contains='Foo') \
...     .first_common_ancestor()
<Page: Foo Event Index>
```

This method tries to be efficient, but if you have millions of pages scattered across your page tree, it will be slow.

If `include_self` is True, the ancestor can be one of the pages in the queryset:

```
>>> Page.objects\
...     .filter(title__contains='Foo') \
...     .first_common_ancestor()
<Page: Foo Event Index>
>>> Page.objects\
...     .filter(title__exact='Bar Event Index') \
...     .first_common_ancestor()
<Page: Bar Event Index>
```

A few invalid cases exist: when the queryset is empty, when the root Page is in the queryset and `include_self` is False, and when there are multiple page trees with no common root (a case Wagtail

does not support). If `strict` is False (the default), then the first root node is returned in these cases. If `strict` is True, then a `ObjectDoesNotExist` is raised.

1.5.2 StreamField reference

StreamField block reference

This document details the block types provided by Wagtail for use in `StreamField`, and how they can be combined into new block types.

```
class wagtail.fields.StreamField(blocks, use_json_field=None, blank=False, min_num=None,
                                max_num=None, block_counts=None, collapsed=False)
```

A model field for representing long-form content as a sequence of content blocks of various types. See [How to use StreamField for mixed content](#).

Parameters

- **blocks** – A list of block types, passed as either a list of `(name, block_definition)` tuples or a `StreamBlock` instance.
- **use_json_field** – When true, the field uses `JSONField` as its internal type, allowing the use of `JSONField` lookups and transforms. When false, it uses `TextField` instead. This argument **must** be set to True/False.
- **blank** – When false (the default), at least one block must be provided for the field to be considered valid.
- **min_num** – Minimum number of sub-blocks that the stream must have.
- **max_num** – Maximum number of sub-blocks that the stream may have.
- **block_counts** – Specifies the minimum and maximum number of each block type, as a dictionary mapping block names to dicts with (optional) `min_num` and `max_num` fields.
- **collapsed** – When true, all blocks are initially collapsed.

Changed in version 3.0: The required `use_json_field` argument is added.

```
body = StreamField([
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
], block_counts={
    'heading': {'min_num': 1},
    'image': {'max_num': 5},
}, use_json_field=True)
```

Block options

All block definitions accept the following optional keyword arguments:

- **default**
 - The default value that a new ‘empty’ block should receive.
- **label**
 - The label to display in the editor interface when referring to this block - defaults to a prettified version of the block name (or, in a context where no name is assigned - such as within a `ListBlock` - the empty string).
- **icon**
 - The name of the icon to display for this block type in the menu of available block types. For a list of icon names, see the Wagtail style guide, which can be enabled by adding `wagtail.contrib.styleguide` to your project’s `INSTALLED_APPS`.
- **template**
 - The path to a Django template that will be used to render this block on the front end. See [Template rendering](#)
- **group**
 - The group used to categorize this block. Any blocks with the same group name will be shown together in the editor interface with the group name as a heading.

Field block types

`class wagtail.blocks.CharBlock`

A single-line text input. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **max_length** – The maximum allowed length of the field.
- **min_length** – The minimum allowed length of the field.
- **help_text** – Help text to display alongside the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.TextBlock`

A multi-line text input. As with `CharBlock`, the following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **max_length** – The maximum allowed length of the field.
- **min_length** – The minimum allowed length of the field.
- **help_text** – Help text to display alongside the field.
- **rows** – Number of rows to show on the textarea (defaults to 1).
- **validators** – A list of validation functions for the field (see [Django Validators](#)).

- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.EmailBlock`

A single-line email input that validates that the value is a valid e-mail address. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **help_text** – Help text to display alongside the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.IntegerBlock`

A single-line integer input that validates that the value is a valid whole number. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **max_value** – The maximum allowed numeric value of the field.
- **min_value** – The minimum allowed numeric value of the field.
- **help_text** – Help text to display alongside the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.FloatBlock`

A single-line Float input that validates that the value is a valid floating point number. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **max_value** – The maximum allowed numeric value of the field.
- **min_value** – The minimum allowed numeric value of the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.DecimalBlock`

A single-line decimal input that validates that the value is a valid decimal number. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **help_text** – Help text to display alongside the field.
- **max_value** – The maximum allowed numeric value of the field.

- **min_value** – The minimum allowed numeric value of the field.
- **max_digits** – The maximum number of digits allowed in the number. This number must be greater than or equal to `decimal_places`.
- **decimal_places** – The number of decimal places to store with the number.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field's `class` attribute when rendered on the page editing form.

`class wagtail.blocks.RegexBlock`

A single-line text input that validates a string against a regular expression. The regular expression used for validation must be supplied as the first argument, or as the keyword argument `regex`.

```
blocks.RegexBlock(regex=r'^[0-9]{3}$', error_messages={  
    'invalid': "Not a valid library card number."  
})
```

The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **regex** – Regular expression to validate against.
- **error_messages** – Dictionary of error messages, containing either or both of the keys `required` (for the message shown on an empty value) or `invalid` (for the message shown on a non-matching value).
- **required** – If true (the default), the field cannot be left blank.
- **help_text** – Help text to display alongside the field.
- **max_length** – The maximum allowed length of the field.
- **min_length** – The minimum allowed length of the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field's `class` attribute when rendered on the page editing form.

`class wagtail.blocks.URLBlock`

A single-line text input that validates that the string is a valid URL. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **max_length** – The maximum allowed length of the field.
- **min_length** – The minimum allowed length of the field.
- **help_text** – Help text to display alongside the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field's `class` attribute when rendered on the page editing form.

`class wagtail.blocks.BooleanBlock`

A checkbox. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the checkbox must be ticked to proceed. As with Django’s BooleanField, a checkbox that can be left ticked or unticked must be explicitly denoted with required=False.
- **help_text** – Help text to display alongside the field.
- **form_classname** – A value to add to the form field’s class attribute when rendered on the page editing form.

`class wagtail.blocks.DateBlock`

A date picker. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **format** – Date format. This must be one of the recognised formats listed in the DATE_INPUT_FORMATS setting. If not specified Wagtail will use the WAGTAIL_DATE_FORMAT setting with fallback to ‘%Y-%m-%d’.
- **required** – If true (the default), the field cannot be left blank.
- **help_text** – Help text to display alongside the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s class attribute when rendered on the page editing form.

`class wagtail.blocks.TimeBlock`

A time picker. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **help_text** – Help text to display alongside the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s class attribute when rendered on the page editing form.

`class wagtail.blocks.DateTimeBlock`

A combined date / time picker. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **format** – Date/time format. This must be one of the recognised formats listed in the DATETIME_INPUT_FORMATS setting. If not specified Wagtail will use the WAGTAIL_DATETIME_FORMAT setting with fallback to ‘%Y-%m-%d %H:%M’.
- **required** – If true (the default), the field cannot be left blank.
- **help_text** – Help text to display alongside the field.

- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.RichTextBlock`

A WYSIWYG editor for creating formatted text including links, bold / italics etc. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **editor** – The rich text editor to be used (see [WAGTAILADMIN_RICH_TEXT_EDITORS](#)).
- **features** – Specifies the set of features allowed (see [Limiting features in a rich text field](#)).
- **required** – If true (the default), the field cannot be left blank.
- **max_length** – The maximum allowed length of the field. Only text is counted; rich text formatting, embedded content and paragraph / line breaks do not count towards the limit.
- **help_text** – Help text to display alongside the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.RawHTMLBlock`

A text area for entering raw HTML which will be rendered unescaped in the page output. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **max_length** – The maximum allowed length of the field.
- **min_length** – The minimum allowed length of the field.
- **help_text** – Help text to display alongside the field.
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

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

`class wagtail.blocks.BlockQuoteBlock`

A text field, the contents of which will be wrapped in an HTML `<blockquote>` tag pair in the page output. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **max_length** – The maximum allowed length of the field.
- **min_length** – The minimum allowed length of the field.
- **help_text** – Help text to display alongside the field.

- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.ChoiceBlock`

A dropdown select box for choosing one item from a list of choices. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **choices** – A list of choices, in any format accepted by Django’s `choices` parameter for model fields, or a callable returning such a list.
- **required** – If true (the default), the field cannot be left blank.
- **help_text** – Help text to display alongside the field.
- **widget** – The form widget to render the field with (see [Django Widgets](#)).
- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

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

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

could be rewritten as a subclass of `ChoiceBlock`:

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

`StreamField` definitions can then refer to `DrinksChoiceBlock()` in place of the full `ChoiceBlock` definition. Note that this only works when `choices` is a fixed list, not a callable.

`class wagtail.blocks.MultipleChoiceBlock`

A select box for choosing multiple items from a list of choices. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **choices** – A list of choices, in any format accepted by Django’s `choices` parameter for model fields, or a callable returning such a list.
- **required** – If true (the default), the field cannot be left blank.
- **help_text** – Help text to display alongside the field.
- **widget** – The form widget to render the field with (see [Django Widgets](#)).

- **validators** – A list of validation functions for the field (see [Django Validators](#)).
- **form_classname** – A value to add to the form field’s `class` attribute when rendered on the page editing form.

`class wagtail.blocks.PageChooserBlock`

A control for selecting a page object, using Wagtail’s page browser. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **page_type** – Restrict choices to one or more specific page types; by default, any page type may be selected. Can be specified as a page model class, model name (as a string), or a list or tuple of these.
- **can_choose_root** – Defaults to false. If true, the editor can choose the tree root as a page. Normally this would be undesirable, since the tree root is never a usable page, but in some specialised cases it may be appropriate. For example, a block providing a feed of related articles could use a `PageChooserBlock` to select which subsection of the site articles will be taken from, with the root corresponding to ‘everywhere’.

`class wagtail.documents.blocks.DocumentChooserBlock`

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

Parameters required – If true (the default), the field cannot be left blank.

`class wagtail.images.blocks.ImageChooserBlock`

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

Parameters required – If true (the default), the field cannot be left blank.

`class wagtail.snippets.blocks.SnippetChooserBlock`

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

Parameters required – If true (the default), the field cannot be left blank.

`class wagtail.embeds.blocks.EmbedBlock`

A field for the editor to enter a URL to a media item (such as a YouTube video) to appear as embedded media on the page. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **required** – If true (the default), the field cannot be left blank.
- **max_width** – The maximum width of the embed, in pixels; this will be passed to the provider when requesting the embed.
- **max_height** – The maximum height of the embed, in pixels; this will be passed to the provider when requesting the embed.
- **max_length** – The maximum allowed length of the field.
- **min_length** – The minimum allowed length of the field.
- **help_text** – Help text to display alongside the field.

Structural block types

```
class wagtail.blocks.StaticBlock
```

A block which doesn't have any fields, thus passes no particular values to its template during rendering. This can be useful if you need the editor to be able to insert some content which is always the same or doesn't need to be configured within the page editor, such as an address, embed code from third-party services, or more complex pieces of code if the template uses template tags. The following additional keyword argument is accepted:

Parameters `admin_text` – A text string to display in the admin when this block is used. By default, some default text (which contains the `label` keyword argument if you pass it) will be displayed in the editor interface, so that the block doesn't look empty, but this can be customised by passing `admin_text`:

```
blocks.StaticBlock(
    admin_text='Latest posts: no configuration needed.',
    # or admin_text=mark_safe('<b>Latest posts</b>: no configuration needed.'),
    template='latest_posts.html')
```

`StaticBlock` can also be subclassed to produce a reusable block with the same configuration everywhere it is used:

```
class LatestPostsStaticBlock(blocks.StaticBlock):
    class Meta:
        icon = 'user'
        label = 'Latest posts'
        admin_text = '{label}: configured elsewhere'.format(label=label)
        template = 'latest_posts.html'
```

```
class wagtail.blocks.StructBlock
```

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

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

Alternatively, `StructBlock` can be subclassed to specify a reusable set of sub-blocks:

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

    class Meta:
        icon = 'user'
```

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

This defines `PersonBlock()` as a block type for use in `StreamField` definitions:

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

The following additional options are available as either keyword arguments or Meta class attributes:

Parameters

- **form_classname** – An HTML `class` attribute to set on the root element of this block as displayed in the editing interface. Defaults to `struct-block`; note that the admin interface has CSS styles defined on this class, so it is advised to include `struct-block` in this value when overriding. See [Custom editing interfaces for StructBlock](#).
- **form_template** – Path to a Django template to use to render this block's form. See [Custom editing interfaces for StructBlock](#).
- **value_class** – A subclass of `wagtail.blocks.StructValue` to use as the type of returned values for this block. See [Additional methods and properties on StructBlock values](#).
- **label_format** – Determines the label shown when the block is collapsed in the editing interface. By default, the value of the first sub-block in the `StructBlock` is shown, but this can be customised by setting a string here with block names contained in braces - for example `label_format = "Profile for {first_name} {surname}"`

`class wagtail.blocks.ListBlock`

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

```
body = StreamField([
    # ...
    ('ingredients_list', blocks.ListBlock(blocks.CharBlock(label="Ingredient"))),
], use_json_field=True)
```

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

```
body = StreamField([
    # ...
    ('ingredients_list', blocks.ListBlock(blocks.StructBlock([
        ('ingredient', blocks.CharBlock()),
        ('amount', blocks.CharBlock(required=False)),
    ]))),
], use_json_field=True)
```

The following additional options are available as either keyword arguments or Meta class attributes:

Parameters

- **form_classname** – An HTML `class` attribute to set on the root element of this block as displayed in the editing interface.
- **min_num** – Minimum number of sub-blocks that the list must have.
- **max_num** – Maximum number of sub-blocks that the list may have.
- **collapsed** – When true, all sub-blocks are initially collapsed.

class wagtail.blocks.StreamBlock

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

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

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

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

    class Meta:
        icon='cogs'
```

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

```
class HomePage(Page):
    carousel = StreamField(
        CarouselBlock(max_num=10, block_counts={'video': {'max_num': 2}}),
        use_json_field=True
    )
```

StreamBlock accepts the following additional options as either keyword arguments or `Meta` properties:

param required If true (the default), at least one sub-block must be supplied. This is ignored when using the StreamBlock as the top-level block of a StreamField; in this case the StreamField's blank property is respected instead.

param min_num Minimum number of sub-blocks that the stream must have.

param max_num Maximum number of sub-blocks that the stream may have.

param block_counts Specifies the minimum and maximum number of each block type, as a dictionary mapping block names to dicts with (optional) `min_num` and `max_num` fields.

param collapsed When true, all sub-blocks are initially collapsed.

param form_classname An HTML class attribute to set on the root element of this block as displayed in the editing interface.

```
body = StreamField([
    # ...
    ('event_promotions', blocks.StreamBlock([
        ('hashtag', blocks.CharBlock()),
        ('post_date', blocks.DateBlock()),
    ], form_classname='event-promotions')),
], use_json_field=True)
```

```
class EventPromotionsBlock(blocks.StreamBlock):
    hashtag = blocks.CharBlock()
    post_date = blocks.DateBlock()

    class Meta:
        form_classname = 'event-promotions'
```

Form widget client-side API

In order for the StreamField editing interface to dynamically create form fields, any Django form widgets used within StreamField blocks must have an accompanying JavaScript implementation, defining how the widget is rendered client-side and populated with data, and how to extract data from that field. Wagtail provides this implementation for widgets inheriting from `django.forms.widgets.Input`, `django.forms.Textarea`, `django.forms.Select` and `django.forms.RadioSelect`. For any other widget types, or ones which require custom client-side behaviour, you will need to provide your own implementation.

The `telepath` library is used to set up mappings between Python widget classes and their corresponding JavaScript implementations. To create a mapping, define a subclass of `wagtail.widget_adapters.WidgetAdapter` and register it with `wagtail.telepath.register`.

```
from wagtail.telepath import register
from wagtail.widget_adapters import WidgetAdapter

class FancyInputAdapter(WidgetAdapter):
    # Identifier matching the one registered on the client side
    js_constructor = 'myapp.widgets.FancyInput'

    # Arguments passed to the client-side object
    def js_args(self, widget):
        return [
            # Arguments typically include the widget's HTML representation
            # and label ID rendered with __NAME__ and __ID__ placeholders,
            # for use in the client-side render() method
            widget.render('__NAME__', None, attrs={'id': '__ID__'}),
            widget.id_for_label('__ID__'),
            widget.extra_options,
        ]

    class Media:
        # JS / CSS includes required in addition to the widget's own media;
        # generally this will include the client-side adapter definition
        js = ['myapp/js/fancy-input-adapter.js']
```

(continues on next page)

(continued from previous page)

```
register(FancyInputAdapter(), FancyInput)
```

The JavaScript object associated with a widget instance should provide a single method:

render(*placeholder, name, id, initialState*)

Render a copy of this widget into the current page, and perform any initialisation required.

Arguments

- **placeholder** – An HTML DOM element to be replaced by the widget’s HTML.
- **name** – A string to be used as the `name` attribute on the input element. For widgets that use multiple input elements (and have server-side logic for collating them back into a final value), this can be treated as a prefix, with further elements delimited by dashes. (For example, if `name` is ‘`person-0`’, the widget may create elements with names `person-0-first_name` and `person-0-surname` without risking collisions with other field names on the form.)
- **id** – A string to be used as the `id` attribute on the input element. As with `name`, this can be treated as a prefix for any further identifiers.
- **initialState** – The initial data to populate the widget with.

A widget’s state will often be the same as the form field’s value, but may contain additional data beyond what is processed in the form submission. For example, a page chooser widget consists of a hidden form field containing the page ID, and a read-only label showing the page title: in this case, the page ID by itself does not provide enough information to render the widget, and so the state is defined as a dictionary with `id` and `title` items.

The value returned by `render` is a ‘bound widget’ object allowing this widget instance’s data to be accessed. This object should implement the following attributes and methods:

idForLabel

The HTML ID to use as the `for` attribute of a label referencing this widget, or null if no suitable HTML element exists.

getValue()

Returns the submittable value of this widget (typically the same as the input element’s value).

getState()

Returns the internal state of this widget, as a value suitable for passing as the `render` method’s `initialState` argument.

setState(*newState*)

Optional: updates this widget’s internal state to the passed value.

focus(*soft*)

Sets the browser’s focus to this widget, so that it receives input events. Widgets that do not have a concept of focus should do nothing. If `soft` is true, this indicates that the focus event was not explicitly triggered by a user action (for example, when a new block is inserted, and the first field is focused as a convenience to the user) - in this case, the widget should avoid performing obtrusive UI actions such as opening modals.

StreamField block reference

Details the block types provided by Wagtail for use in StreamField and how they can be combined into new block types.

Form widget client-side API

Defines the JavaScript API that must be implemented for any form widget used within a StreamField block.

1.5.3 Contrib modules

Wagtail ships with a variety of extra optional modules.

Settings

The `wagtail.contrib.settings` module allows you to define models that hold settings which are either common across all site records, or specific to each site.

Settings are editable by administrators within the Wagtail admin, and can be accessed in code as well as in templates.

Installation

Add `wagtail.contrib.settings` to your `INSTALLED_APPS`:

```
INSTALLED_APPS += [
    'wagtail.contrib.settings',
]
```

Note: If you are using `settings` within templates, you will also need to update your `TEMPLATES` settings (discussed later in this page).

Defining settings

Create a model that inherits from either:

- `BaseGenericSetting` for generic settings across all sites
- `BaseSiteSetting` for site-specific settings

and register it using the `register_setting` decorator:

```
from django.db import models
from wagtail.contrib.settings.models import (
    BaseGenericSetting,
    BaseSiteSetting,
    register_setting,
)

@register_setting
class GenericSocialMediaSettings(BaseGenericSetting):
    facebook = models.URLField()
```

(continues on next page)

(continued from previous page)

```
@register_setting
class SiteSpecificSocialMediaSettings(BaseSiteSetting):
    facebook = models.URLField()
```

Links to your settings will appear in the Wagtail admin ‘Settings’ menu.

Edit handlers

Settings use edit handlers much like the rest of Wagtail. Add a `panels` setting to your model defining all the edit handlers required:

```
@register_setting
class GenericImportantPages(BaseGenericSetting):
    donate_page = models.ForeignKey(
        'wagtailcore.Page', null=True, on_delete=models.SET_NULL, related_name='+'
    )
    sign_up_page = models.ForeignKey(
        'wagtailcore.Page', null=True, on_delete=models.SET_NULL, related_name='+'
    )

    panels = [
        FieldPanel('donate_page'),
        FieldPanel('sign_up_page'),
    ]

@register_setting
class SiteSpecificImportantPages(BaseSiteSetting):
    donate_page = models.ForeignKey(
        'wagtailcore.Page', null=True, on_delete=models.SET_NULL, related_name='+'
    )
    sign_up_page = models.ForeignKey(
        'wagtailcore.Page', null=True, on_delete=models.SET_NULL, related_name='+'
    )

    panels = [
        FieldPanel('donate_page'),
        FieldPanel('sign_up_page'),
    ]
```

You can also customise the edit handlers *like you would do for Page model* with a custom `edit_handler` attribute:

```
from wagtail.admin.panels import TabbedInterface, ObjectList

@register_setting
class MySettings(BaseGenericSetting):
    # ...
    first_tab_panels = [
        FieldPanel('field_1'),
    ]
    second_tab_panels = [
        FieldPanel('field_2'),
```

(continues on next page)

(continued from previous page)

```
]

edit_handler = TabbedInterface([
    ObjectList(first_tab_panels, heading='First tab'),
    ObjectList(second_tab_panels, heading='Second tab'),
])
```

Apearance

You can change the label used in the menu by changing the `verbose_name` of your model.

You can add an icon to the menu by passing an `icon` argument to the `register_setting` decorator:

```
@register_setting(icon='placeholder')
class GenericSocialMediaSettings(BaseGenericSetting):
    ...
    class Meta:
        verbose_name = "Social media settings for all sites"

@register_setting(icon='placeholder')
class SiteSpecificSocialMediaSettings(BaseSiteSetting):
    ...
    class Meta:
        verbose_name = "Site-specific social media settings"
```

For a list of all available icons, please see the [styleguide](#).

Using the settings

Settings can be used in both Python code and in templates.

Using in Python

Generic settings

If you require access to a generic setting in a view, the `BaseGenericSetting.load()` method allows you to retrieve the generic settings:

```
def view(request):
    social_media_settings = GenericSocialMediaSettings.load(request_or_site=request)
    ...
```

Site-specific settings

If you require access to a site-specific setting in a view, the `BaseSiteSetting.for_request()` method allows you to retrieve the site-specific settings for the current request:

```
def view(request):
    social_media_settings = SiteSpecificSocialMediaSettings.for_request(request=request)
    ...
```

In places where the request is unavailable, but you know the `Site` you wish to retrieve settings for, you can use `BaseSiteSetting.for_site` instead:

```
def view(request):
    social_media_settings = SiteSpecificSocialMediaSettings.for_site(site=user.origin_
    ↵site)
    ...
```

Using in Django templates

Add the `wagtail.contrib.settings.context_processors.settings` context processor to your settings:

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'wagtail.contrib.settings.context_processors.settings',
            ]
        }
    }
]
```

Then access the generic settings through `{{ settings }}`:

```
{{ settings.app_label.GenericSocialMediaSettings.facebook }}
{{ settings.app_label.SiteSpecificSocialMediaSettings.facebook }}
```

Note: Replace `app_label` with the label of the app containing your settings model.

If you are not in a `RequestContext`, then context processors will not have run, and the `settings` variable will not be available. To get the `settings`, use the provided `{% get_settings %}` template tag.

```
{% load wagtailsettings_tags %}
{% get_settings %}
{{ settings.app_label.GenericSocialMediaSettings.facebook }}
{{ settings.app_label.SiteSpecificSocialMediaSettings.facebook }}
```

By default, the tag will create or update a `settings` variable in the context. If you want to assign to a different context variable instead, use `{% get_settings as other_variable_name %}`:

```
{% load wagtailsettings_tags %}  
{% get_settings as wagtail_settings %}  
{{ wagtail_settings.app_label.GenericSocialMediaSettings.facebook }}  
{{ wagtail_settings.app_label.SiteSpecificSocialMediaSettings.facebook }}
```

Using in Jinja2 templates

Add `wagtail.contrib.settings.jinja2tags.settings` extension to your Jinja2 settings:

```
TEMPLATES = [  
    ...  
    {  
        'BACKEND': 'django.template.backends.jinja2.Jinja2',  
        'APP_DIRS': True,  
        'OPTIONS': {  
            'extensions': [  
                ...  
  
                'wagtail.contrib.settings.jinja2tags.settings',  
            ],  
        },  
    },  
]
```

Then access the settings through the `settings()` template function:

```
{{ settings("app_label.GenericSocialMediaSettings").facebook }}  
{{ settings("app_label.SiteSpecificSocialMediaSettings").facebook }}
```

Note: Replace `app_label` with the label of the app containing your settings model.

If there is no `request` available in the template at all, you can use the settings for the default site instead:

```
{{ settings("app_label.GenericSocialMediaSettings", use_default_site=True).facebook }}  
{{ settings("app_label.SiteSpecificSocialMediaSettings", use_default_site=True).facebook }}  
→{}
```

Note: You can not reliably get the correct settings instance for the current site from this template tag if the request object is not available. This is only relevant for multi-site instances of Wagtail.

You can store the settings instance in a variable to save some typing, if you have to use multiple values from one model:

```
{% with generic_social_settings=settings("app_label.GenericSocialMediaSettings") %}  
    Follow us on Twitter at @{{ generic_social_settings.facebook }},  
    or Instagram at @{{ generic_social_settings.instagram }}.  
{% endwith %}  
  
{% with site_social_settings=settings("app_label.SiteSpecificSocialMediaSettings") %}  
    Follow us on Twitter at @{{ site_social_settings.facebook }},  
    or Instagram at @{{ site_social_settings.instagram }}.  
{% endwith %}
```

Or, alternately, using the `set` tag:

```
{% set generic_social_settings=settings("app_label.GenericSocialMediaSettings") %}
{% set site_social_settings=settings("app_label.SiteSpecificSocialMediaSettings") %}
```

Utilising select_related to improve efficiency

For models with foreign key relationships to other objects (for example pages), which are very often needed to output values in templates, you can set the `select_related` attribute on your model to have Wagtail utilise Django's `QuerySet.select_related()` method to fetch the settings object and related objects in a single query. With this, the initial query is more complex, but you will be able to freely access the foreign key values without any additional queries, making things more efficient overall.

Building on the `GenericImportantPages` example from the previous section, the following shows how `select_related` can be set to improve efficiency:

```
@register_setting
class GenericImportantPages(BaseGenericSetting):

    # Fetch these pages when looking up GenericImportantPages for or a site
    select_related = ["donate_page", "sign_up_page"]

    donate_page = models.ForeignKey(
        'wagtailcore.Page', null=True, on_delete=models.SET_NULL, related_name='+'
    )
    sign_up_page = models.ForeignKey(
        'wagtailcore.Page', null=True, on_delete=models.SET_NULL, related_name='+'
    )

    panels = [
        FieldPanel('donate_page'),
        FieldPanel('sign_up_page'),
    ]
```

With these additions, the following template code will now trigger a single database query instead of three (one to fetch the settings, and two more to fetch each page):

```
{% load wagtailcore_tags %}
{% pageurl settings.app_label.GenericImportantPages.donate_page %}
{% pageurl settings.app_label.GenericImportantPages.sign_up_page %}
```

Utilising the page_url setting shortcut

If, like in the previous section, your settings model references pages, and you often need to output the URLs of those pages in your project, you can likely use the setting model's `page_url` shortcut to do that more cleanly. For example, instead of doing the following:

```
{% load wagtailcore_tags %}
{% pageurl settings.app_label.GenericImportantPages.donate_page %}
{% pageurl settings.app_label.GenericImportantPages.sign_up_page %}
```

You could write:

```
{{ settings.app_label.GenericImportantPages.page_url.donate_page }}  
{{ settings.app_label.GenericImportantPages.page_url.sign_up_page }}
```

Using the `page_url` shortcut has a few of advantages over using the tag:

1. The ‘specific’ page is automatically fetched to generate the URL, so you don’t have to worry about doing this (or forgetting to do this) yourself.
2. The results are cached, so if you need to access the same page URL in more than one place (for example in a form and in footer navigation), using the `page_url` shortcut will be more efficient.
3. It’s more concise, and the syntax is the same whether using it in templates or views (or other Python code), allowing you to write more consistent code.

When using the `page_url` shortcut, there are a couple of points worth noting:

1. The same limitations that apply to the `{% pageurl %}` tag apply to the shortcut: If the settings are accessed from a template context where the current request is not available, all URLs returned will include the site’s scheme/domain, and URL generation will not be quite as efficient.
2. If using the shortcut in views or other Python code, the method will raise an `AttributeError` if the attribute you request from `page_url` is not an attribute on the settings object.
3. If the settings object DOES have the attribute, but the attribute returns a value of `None` (or something that is not a Page), the shortcut will return an empty string.

Form builder

The `wagtailforms` module allows you to set up single-page forms, such as a ‘Contact us’ form, as pages of a Wagtail site. It provides a set of base models that site implementers can extend to create their own `FormPage` type with their own site-specific templates. Once a page type has been set up in this way, editors can build forms within the usual page editor, consisting of any number of fields. Form submissions are stored for later retrieval through a new ‘Forms’ section within the Wagtail admin interface; in addition, they can be optionally e-mailed to an address specified by the editor.

Note: wagtailforms is not a replacement for Django’s form support. It is designed as a way for page authors to build general-purpose data collection forms without having to write code. If you intend to build a form that assigns specific behaviour to individual fields (such as creating user accounts), or needs a custom HTML layout, you will almost certainly be better served by a standard Django form, where the fields are fixed in code rather than defined on-the-fly by a page author. See the [wagtail-form-example](#) project for an example of integrating a Django form into a Wagtail page.

Usage

Add `wagtail.contrib.forms` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [  
    ...  
    'wagtail.contrib.forms',  
]
```

Within the `models.py` of one of your apps, create a model that extends `wagtail.contrib.forms.models.AbstractEmailForm`:

```

from django.db import models
from modelcluster.fields import ParentalKey
from wagtail.admin.panels import (
    FieldPanel, FieldRowPanel,
    InlinePanel, MultiFieldPanel
)
from wagtail.fields import RichTextField
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields')

class FormPage(AbstractEmailForm):
    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

    content_panels = AbstractEmailForm.content_panels + [
        FieldPanel('intro'),
        InlinePanel('form_fields', label="Form fields"),
        FieldPanel('thank_you_text'),
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('from_address', classname="col6"),
                FieldPanel('to_address', classname="col6"),
            ]),
            FieldPanel('subject'),
        ], "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.

Date and datetime values in a form response will be formatted with the `SHORT_DATE_FORMAT` and `SHORT_DATETIME_FORMAT` respectively. (see [Custom render_email method](#) for how to customise the email content).

If you do not want your form page type to offer form-to-email functionality, you can inherit from `AbstractForm` instead of `AbstractEmailForm`, and omit the `to_address`, `from_address` and `subject` fields from the `content_panels` definition.

You now need to create two templates named `form_page.html` and `form_page_landing.html` (where `form_page` is the underscore-formatted version of the class name). `form_page.html` differs from a standard Wagtail template in that it is passed a variable `form`, containing a Django Form object, in addition to the usual `page` variable. A very basic template for the form would thus be:

```

{% load wagtailcore_tags %}
<html>
  <head>
    <title>{{ page.title }}</title>
  </head>

```

(continues on next page)

(continued from previous page)

```
<body>
    <h1>{{ page.title }}</h1>
    {{ page.intro|richtext }}
    <form action="{% pageurl page %}" method="POST">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit">
    </form>
</body>
</html>
```

`form_page_landing.html` is a standard Wagtail template, displayed after the user makes a successful form submission, `form_submission` will be available in this template. If you want to dynamically override the landing page template, you can do so with the `get_landing_page_template` method (in the same way that you would with `get_template`).

Displaying form submission information

`FormSubmissionsPanel` can be added to your page's panel definitions to display the number of form submissions and the time of the most recent submission, along with a quick link to access the full submission data:

```
from wagtail.contrib.forms.panels import FormSubmissionsPanel

class FormPage(AbstractEmailForm):
    # ...

    content_panels = AbstractEmailForm.content_panels + [
        FormSubmissionsPanel(),
        FieldPanel('intro'),
        # ...
    ]
```

Index

Form builder customisation

For a basic usage example see [form builder usage](#).

Custom `related_name` for form fields

If you want to change `related_name` for form fields (by default `AbstractForm` and `AbstractEmailForm` expect `form_fields` to be defined), you will need to override the `get_form_fields` method. You can do this as shown below.

```
from modelcluster.fields import ParentalKey
from wagtail.admin.panels import (
    FieldPanel, FieldRowPanel,
    InlinePanel, MultiFieldPanel
```

(continues on next page)

(continued from previous page)

```

)
from wagtail.fields import RichTextField
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='custom_form_
    ↵fields')

class FormPage(AbstractEmailForm):
    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

    content_panels = AbstractEmailForm.content_panels + [
        FieldPanel('intro'),
        InlinePanel('custom_form_fields', label="Form fields"),
        FieldPanel('thank_you_text'),
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('from_address', classname="col6"),
                FieldPanel('to_address', classname="col6"),
            ]),
            FieldPanel('subject'),
        ], "Email"),
    ]

    def get_form_fields(self):
        return self.custom_form_fields.all()

```

Custom form submission model

If you need to save additional data, you can use a custom form submission model. To do this, you need to:

- Define a model that extends `wagtail.contrib.forms.models.AbstractFormSubmission`.
- Override the `get_submission_class` and `process_form_submission` methods in your page model.

Example:

```

import json

from django.conf import settings
from django.db import models
from modelcluster.fields import ParentalKey
from wagtail.admin.panels import (
    FieldPanel, FieldRowPanel,
    InlinePanel, MultiFieldPanel
)
from wagtail.fields import RichTextField
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField, ↵
    AbstractFormSubmission

```

(continues on next page)

(continued from previous page)

```

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields')

class FormPage(AbstractEmailForm):
    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

    content_panels = AbstractEmailForm.content_panels + [
        FieldPanel('intro'),
        InlinePanel('form_fields', label="Form fields"),
        FieldPanel('thank_you_text'),
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('from_address', classname="col6"),
                FieldPanel('to_address', classname="col6"),
            ]),
            FieldPanel('subject'),
        ], "Email"),
    ]

    def get_submission_class(self):
        return CustomFormSubmission

    def process_form_submission(self, form):
        self.get_submission_class().objects.create(
            form_data=form.cleaned_data,
            page=self, user=form.user
        )

```

```

class CustomFormSubmission(AbstractFormSubmission):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)

```

Add custom data to CSV export

If you want to add custom data to the CSV export, you will need to:

- Override the `get_data_fields` method in page model.
- Override `get_data` in the submission model.

The example below shows how to add a username to the CSV export. Note that this code also changes the submissions list view.

```

import json

from django.conf import settings
from django.db import models
from modelcluster.fields import ParentalKey

```

(continues on next page)

(continued from previous page)

```

from wagtail.admin.panels import (
    FieldPanel, FieldRowPanel,
    InlinePanel, MultiFieldPanel
)
from wagtail.fields import RichTextField
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField, ↴
    AbstractFormSubmission

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields')

class FormPage(AbstractEmailForm):
    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

    content_panels = AbstractEmailForm.content_panels + [
        FieldPanel('intro'),
        InlinePanel('form_fields', label="Form fields"),
        FieldPanel('thank_you_text'),
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('from_address', classname="col6"),
                FieldPanel('to_address', classname="col6"),
            ]),
            FieldPanel('subject'),
        ], "Email"),
    ]

    def get_data_fields(self):
        data_fields = [
            ('username', 'Username'),
        ]
        data_fields += super().get_data_fields()

        return data_fields

    def get_submission_class(self):
        return CustomFormSubmission

    def process_form_submission(self, form):
        self.get_submission_class().objects.create(
            form_data=form.cleaned_data,
            page=self, user=form.user
        )

class CustomFormSubmission(AbstractFormSubmission):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)

    def get_data(self):

```

(continues on next page)

(continued from previous page)

```

form_data = super().get_data()
form_data.update({
    'username': self.user.username,
})

return form_data

```

Check that a submission already exists for a user

If you want to prevent users from filling in a form more than once, you need to override the `serve` method in your page model.

Example:

```

import json

from django.conf import settings
from django.db import models
from django.shortcuts import render
from modelcluster.fields import ParentalKey
from wagtail.admin.panels import (
    FieldPanel, FieldRowPanel,
    InlinePanel, MultiFieldPanel
)
from wagtail.fields import RichTextField
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField, ↴
    AbstractFormSubmission


class FormField(AbstractFormField):
    page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields')


class FormPage(AbstractEmailForm):
    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

    content_panels = AbstractEmailForm.content_panels + [
        FieldPanel('intro'),
        InlinePanel('form_fields', label="Form fields"),
        FieldPanel('thank_you_text'),
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('from_address', classname="col6"),
                FieldPanel('to_address', classname="col6"),
            ]),
            FieldPanel('subject'),
        ], "Email"),
    ]

    def serve(self, request, *args, **kwargs):

```

(continues on next page)

(continued from previous page)

```

    if self.get_submission_class().objects.filter(page=self, user_pk=request.user.
pk).exists():
    return render(
        request,
        self.template,
        self.get_context(request)
    )

return super().serve(request, *args, **kwargs)

def get_submission_class(self):
    return CustomFormSubmission

def process_form_submission(self, form):
    self.get_submission_class().objects.create(
        form_data=form.cleaned_data,
        page=self, user=form.user
    )

class CustomFormSubmission(AbstractFormSubmission):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)

    class Meta:
        unique_together = ('page', 'user')

```

Your template should look like this:

```

{% load wagtailcore_tags %}

<html>
    <head>
        <title>{{ page.title }}</title>
    </head>
    <body>
        <h1>{{ page.title }}</h1>

        {% if user.is_authenticated and user.is_active or request.is_preview %}
            {% if form %}
                <div>{{ page.intro|richtext }}</div>
                <form action="{% pageurl page %}" method="POST">
                    {% csrf_token %}
                    {{ form.as_p }}
                    <input type="submit">
                </form>
            {% else %}
                <div>You can fill in the form only one time.</div>
            {% endif %}
        {% else %}
            <div>To fill in the form, you must log in.</div>
        {% endif %}
    </body>
</html>

```

Multi-step form

The following example shows how to create a multi-step form.

```
from django.core.paginator import Paginator, PageNotAnInteger, EmptyPage
from django.shortcuts import render
from modelcluster.fields import ParentalKey
from wagtail.admin.panels import (
    FieldPanel, FieldRowPanel,
    InlinePanel, MultiFieldPanel
)
from wagtail.fields import RichTextField
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields')

class FormPage(AbstractEmailForm):
    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

    content_panels = AbstractEmailForm.content_panels + [
        FieldPanel('intro'),
        InlinePanel('form_fields', label="Form fields"),
        FieldPanel('thank_you_text'),
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('from_address', classname="col6"),
                FieldPanel('to_address', classname="col6"),
            ]),
            FieldPanel('subject'),
        ], "Email"),
    ]

    def get_form_class_for_step(self, step):
        return self.form_builder(step.object_list).get_form_class()

    def serve(self, request, *args, **kwargs):
        """
        Implements a simple multi-step form.

        Stores each step into a session.
        When the last step was submitted correctly, saves whole form into a DB.
        """

        session_key_data = 'form_data-%s' % self.pk
        is_last_step = False
        step_number = request.GET.get('p', 1)

        paginator = Paginator(self.get_form_fields(), per_page=1)
        try:
```

(continues on next page)

(continued from previous page)

```

        step = paginator.page(step_number)
    except PageNotAnInteger:
        step = paginator.page(1)
    except EmptyPage:
        step = paginator.page(paginator.num_pages)
        is_last_step = True

    if request.method == 'POST':
        # The first step will be submitted with step_number == 2,
        # so we need to get a form from previous step
        # Edge case - submission of the last step
        prev_step = step if is_last_step else paginator.page(step.previous_page_
number())

        # Create a form only for submitted step
        prev_form_class = self.get_form_class_for_step(prev_step)
        prev_form = prev_form_class(request.POST, page=self, user=request.user)
        if prev_form.is_valid():
            # If data for step is valid, update the session
            form_data = request.session.get(session_key_data, {})
            form_data.update(prev_form.cleaned_data)
            request.session[session_key_data] = form_data

        if prev_step.has_next():
            # Create a new form for a following step, if the following step is_
present
            form_class = self.get_form_class_for_step(step)
            form = form_class(page=self, user=request.user)
        else:
            # If there is no next step, create form for all fields
            form = self.get_form(
                request.session[session_key_data],
                page=self, user=request.user
            )

        if form.is_valid():
            # Perform validation again for whole form.
            # After successful validation, save data into DB,
            # and remove from the session.
            form_submission = self.process_form_submission(form)
            del request.session[session_key_data]
            # render the landing page
            return self.render_landing_page(request, form_submission, *args,_
**kwargs)
        else:
            # If data for step is invalid
            # we will need to display form again with errors,
            # so restore previous state.
            form = prev_form
            step = prev_step
    else:
        # Create empty form for non-POST requests

```

(continues on next page)

(continued from previous page)

```

        form_class = self.get_form_class_for_step(step)
        form = form_class(page=self, user=request.user)

        context = self.get_context(request)
        context['form'] = form
        context['fields_step'] = step
        return render(
            request,
            self.template,
            context
        )
    )

```

Your template for this form page should look like this:

```

{% load wagtailcore_tags %}

<html>
  <head>
    <title>{{ page.title }}</title>
  </head>
  <body>
    <h1>{{ page.title }}</h1>

    <div>{{ page.intro|richtext }}</div>
    <form action="{% pageurl page %}?p={{ fields_step.number|add:"1" }}" method="POST"
      >
      {% csrf_token %}
      {{ form.as_p }}
      <input type="submit">
    </form>
  </body>
</html>

```

Note that the example shown before allows the user to return to a previous step, or to open a second step without submitting the first step. Depending on your requirements, you may need to add extra checks.

Show results

If you are implementing polls or surveys, you may want to show results after submission. The following example demonstrates how to do this.

First, you need to collect results as shown below:

```

from modelcluster.fields import ParentalKey
from wagtail.admin.panels import (
    FieldPanel, FieldRowPanel,
    InlinePanel, MultiFieldPanel
)
from wagtail.fields import RichTextField
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField

class FormField(AbstractFormField):

```

(continues on next page)

(continued from previous page)

```

page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields')

class FormPage(AbstractEmailForm):
    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

    content_panels = AbstractEmailForm.content_panels + [
        FieldPanel('intro'),
        InlinePanel('form_fields', label="Form fields"),
        FieldPanel('thank_you_text'),
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('from_address', classname="col6"),
                FieldPanel('to_address', classname="col6"),
            ]),
            FieldPanel('subject'),
        ], "Email"),
    ]

    def get_context(self, request, *args, **kwargs):
        context = super().get_context(request, *args, **kwargs)

        # If you need to show results only on landing page,
        # you may need check request.method

        results = dict()
        # Get information about form fields
        data_fields = [
            (field.clean_name, field.label)
            for field in self.get_form_fields()
        ]

        # Get all submissions for current page
        submissions = self.get_submission_class().objects.filter(page=self)
        for submission in submissions:
            data = submission.get_data()

            # Count results for each question
            for name, label in data_fields:
                answer = data.get(name)
                if answer is None:
                    # Something wrong with data.
                    # Probably you have changed questions
                    # and now we are receiving answers for old questions.
                    # Just skip them.
                    continue

                if type(answer) is list:
                    # Answer is a list if the field type is 'Checkboxes'
                    answer = u', '.join(answer)

```

(continues on next page)

(continued from previous page)

```

        question_stats = results.get(label, {})
        question_stats[answer] = question_stats.get(answer, 0) + 1
        results[label] = question_stats

    context.update({
        'results': results,
    })
    return context

```

Next, you need to transform your template to display the results:

```

{% load wagtailcore_tags %}

<html>
  <head>
    <title>{{ page.title }}</title>
  </head>
  <body>
    <h1>{{ page.title }}</h1>

    <h2>Results</h2>
    {% for question, answers in results.items %}
      <h3>{{ question }}</h3>
      {% for answer, count in answers.items %}
        <div>{{ answer }}: {{ count }}</div>
      {% endfor %}
    {% endfor %}

    <div>{{ page.intro|richtext }}</div>
    <form action="{% pageurl page %}" method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <input type="submit">
    </form>
  </body>
</html>

```

You can also show the results on the landing page.

Custom landing page redirect

You can override the `render_landing_page` method on your `FormPage` to change what is rendered when a form submits.

In this example below we have added a `thank_you_page` field that enables custom redirects after a form submits to the selected page.

When overriding the `render_landing_page` method, we check if there is a linked `thank_you_page` and then redirect to it if it exists.

Finally, we add a URL param of `id` based on the `form_submission` if it exists.

```

from django.shortcuts import redirect
from wagtail.admin.panels import FieldPanel, FieldRowPanel, InlinePanel, MultiFieldPanel

```

(continues on next page)

(continued from previous page)

```

from wagtail.contrib.forms.models import AbstractEmailForm

class FormPage(AbstractEmailForm):

    # intro, thank_you_text, ...

    thank_you_page = models.ForeignKey(
        'wagtailcore.Page',
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+',
    )

    def render_landing_page(self, request, form_submission=None, *args, **kwargs):
        if self.thank_you_page:
            url = self.thank_you_page.url
            # if a form_submission instance is available, append the id to URL
            # when previewing landing page, there will not be a form_submission instance
            if form_submission:
                url += '?id=%s' % form_submission.id
            return redirect(url, permanent=False)
        # if no thank_you_page is set, render default landing page
        return super().render_landing_page(request, form_submission, *args, **kwargs)

    content_panels = AbstractEmailForm.content_panels + [
        FieldPanel('intro'),
        InlinePanel('form_fields'),
        FieldPanel('thank_you_text'),
        FieldPanel('thank_you_page'),
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('from_address', classname='col6'),
                FieldPanel('to_address', classname='col6'),
            ]),
            FieldPanel('subject'),
        ], 'Email'),
    ]

```

Customise form submissions listing in Wagtail Admin

The Admin listing of form submissions can be customised by setting the attribute `submissions_list_view_class` on your `FormPage` model.

The list view class must be a subclass of `SubmissionsListView` from `wagtail.contrib.forms.views`, which is a child class of Django's class based `ListView`.

Example:

```

from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField
from wagtail.contrib.forms.views import SubmissionsListView

```

(continues on next page)

(continued from previous page)

```

class CustomSubmissionsListView(SubmissionsListView):
    paginate_by = 50 # show more submissions per page, default is 20
    ordering = ('submit_time',) # order submissions by oldest first, normally newest
    ↪first
    ordering_csv = ('-submit_time',) # order csv export by newest first, normally
    ↪oldest first

    # override the method to generate csv filename
    def get_csv_filename(self):
        """ Returns the filename for CSV file with page slug at start"""
        filename = super().get_csv_filename()
        return self.form_page.slug + '-' + filename

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', related_name='form_fields')

class FormPage(AbstractEmailForm):
    """Form Page with customised submissions listing view"""

    # set custom view class as class attribute
    submissions_list_view_class = CustomSubmissionsListView

    intro = RichTextField(blank=True)
    thank_you_text = RichTextField(blank=True)

    # content_panels = ...

```

Adding a custom field type

First, make the new field type available in the page editor by changing your `FormField` model.

- Create a new set of choices which includes the original `FORM_FIELD_CHOICES` along with new field types you want to make available.
- Each choice must contain a unique key and a human readable name of the field, for example `('slug', 'URL Slug')`
- Override the `field_type` field in your `FormField` model with `choices` attribute using these choices.
- You will need to run `./manage.py makemigrations` and `./manage.py migrate` after this step.

Then, create and use a new form builder class.

- Define a new form builder class that extends the `FormBuilder` class.
- Add a method that will return a created Django form field for the new field type.
- Its name must be in the format: `create_<field_type_key>_field`, for example `create_slug_field`
- Override the `form_builder` attribute in your form page model to use your new form builder class.

Example:

```

from django import forms
from django.db import models
from modelcluster.fields import ParentalKey
from wagtail.contrib.forms import FormBuilder
from wagtail.contrib.forms.models import (
    AbstractEmailForm, AbstractFormField, FORM_FIELD_CHOICES)

class FormField(AbstractFormField):
    # extend the built in field type choices
    # our field type key will be 'ipaddress'
    CHOICES = FORM_FIELD_CHOICES + (('ipaddress', 'IP Address'),)

    page = ParentalKey('FormPage', related_name='form_fields')
    # override the field_type field with extended choices
    field_type = models.CharField(
        verbose_name='field type',
        max_length=16,
        # use the choices tuple defined above
        choices=CHOICES
    )

class CustomFormBuilder(FormBuilder):
    # create a function that returns an instanced Django form field
    # function name must match create_<field_type_key>_field
    def create_ipaddress_field(self, field, options):
        # return `forms.GenericIPAddressField(**options)` not `forms.SlugField`
        # returns created a form field with the options passed in
        return forms.GenericIPAddressField(**options)

class FormPage(AbstractEmailForm):
    # intro, thank_you_text, edit_handlers, etc...

    # use custom form builder defined above
    form_builder = CustomFormBuilder

```

Custom render_email method

If you want to change the content of the email that is sent when a form submits you can override the `render_email` method.

To do this, you need to:

- Ensure you have your form model defined that extends `wagtail.contrib.forms.models.AbstractEmailForm`.
- Override the `render_email` method in your page model.

Example:

```

from datetime import date
# ... additional wagtail imports
from wagtail.contrib.forms.models import AbstractEmailForm

class FormPage(AbstractEmailForm):
    # ... fields, content_panels, etc

    def render_email(self, form):
        # Get the original content (string)
        email_content = super().render_email(form)

        # Add a title (not part of original method)
        title = '{}: {}'.format('Form', self.title)

        content = [title, '', email_content, '']

        # Add a link to the form page
        content.append('{}: {}'.format('Submitted Via', self.full_url))

        # Add the date the form was submitted
        submitted_date_str = date.today().strftime('%x')
        content.append('{}: {}'.format('Submitted on', submitted_date_str))

        # Content is joined with a new line to separate each text line
        content = '\n'.join(content)

    return content

```

Custom send_mail method

If you want to change the subject or some other part of how an email is sent when a form submits you can override the `send_mail` method.

To do this, you need to:

- Ensure you have your form model defined that extends `wagtail.contrib.forms.models.AbstractEmailForm`.
- In your `models.py` file, import the `wagtail.admin.mail.send_mail` function.
- Override the `send_mail` method in your page model.

Example:

```

from datetime import date
# ... additional wagtail imports
from wagtail.admin.mail import send_mail
from wagtail.contrib.forms.models import AbstractEmailForm

class FormPage(AbstractEmailForm):
    # ... fields, content_panels, etc

```

(continues on next page)

(continued from previous page)

```
def send_mail(self, form):
    # `self` is the FormPage, `form` is the form's POST data on submit

    # Email addresses are parsed from the FormPage's addresses field
    addresses = [x.strip() for x in self.to_address.split(',')]

    # Subject can be adjusted (adding submitted date), be sure to include the form's
    # defined subject field
    submitted_date_str = date.today().strftime('%x')
    subject = f"{self.subject} - {submitted_date_str}"

    send_mail(subject, self.render_email(form), addresses, self.from_address,)
```

Custom `clean_name` generation

- Each time a new FormField is added a `clean_name` also gets generated based on the user entered label.
- AbstractFormField has a method `get_field_clean_name` to convert the label into a HTML valid lower_snake_case ASCII string using the AnyAscii library which can be overridden to generate a custom conversion.
- The resolved `clean_name` is also used as the form field name in rendered HTML forms.
- Ensure that any conversion will be unique enough to not create conflicts within your FormPage instance.
- This method gets called on creation of new fields only and as such will not have access to its own Page or pk. This does not get called when labels are edited as modifying the `clean_name` after any form responses are submitted will mean those field responses will not be retrieved.
- This method gets called for form previews and also validation of duplicate labels.

```
import uuid

from django.db import models
from modelcluster.fields import ParentalKey

# ... other field and edit_handler imports
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields')

    def get_field_clean_name(self):
        clean_name = super().get_field_clean_name()
        id = str(uuid.uuid4())[:8] # short uuid
        return f'{id}_{clean_name}'


class FormPage(AbstractEmailForm):
    # ... page definitions
```

Using FormMixin or EmailFormMixin to use with other Page subclasses

If you need to add form behaviour while extending an additional class, you can use the base mixins instead of the abstract modals.

```
from wagtail.models import Page
from wagtail.contrib.forms.models import EmailFormMixin, FormMixin

class BasePage(Page):
    """
    A shared base page used throughout the project.
    """

    # ...

class FormPage(FormMixin, BasePage):
    intro = RichTextField(blank=True)
    # ...

class EmailFormPage(EmailFormMixin, FormMixin, BasePage):
    intro = RichTextField(blank=True)
    # ...
```

Sitemap generator

This document describes how to create XML sitemaps for your Wagtail website using the `wagtail.contrib.sitemaps` module.

Note: As of Wagtail 1.10 the Django contrib sitemap app is used to generate sitemaps. However since Wagtail requires the Site instance to be available during the sitemap generation you will have to use the views from the `wagtail.contrib.sitemaps.views` module instead of the views provided by Django (`django.contrib.sitemaps.views`).

The usage of these views is otherwise identical, which means that customisation and caching of the sitemaps are done using the default Django patterns. See the Django documentation for in-depth information.

Basic configuration

You firstly need to add "`django.contrib.sitemaps`" to `INSTALLED_APPS` in your Django settings file:

```
INSTALLED_APPS = [
    ...
    "django.contrib.sitemaps",
]
```

Then, in `urls.py`, you need to add a link to the `wagtail.contrib.sitemaps.views.sitemap` view which generates the sitemap:

```
from wagtail.contrib.sitemaps.views import sitemap

urlpatterns = [
    ...
    path('sitemap.xml', sitemap),
    ...

    # Ensure that the 'sitemap' line appears above the default Wagtail page serving route
    re_path(r'', include(wagtail_urls)),
]
```

You should now be able to browse to `/sitemap.xml` and see the sitemap working. By default, all published pages in your website will be added to the site map.

Setting the hostname

By default, the sitemap uses the hostname defined in the Wagtail Admin's `Sites` area. If your default site is called `localhost`, then URLs in the sitemap will look like:

```
<url>
  <loc>http://localhost/about/</loc>
  <lastmod>2015-09-26</lastmod>
</url>
```

For tools like Google Search Tools to properly index your site, you need to set a valid, crawlable hostname. If you change the site's hostname from `localhost` to `mysite.com`, `sitemap.xml` will contain the correct URLs:

```
<url>
  <loc>http://mysite.com/about/</loc>
  <lastmod>2015-09-26</lastmod>
</url>
```

If you change the site's port to 443, the `https` scheme will be used. Find out more about [working with Sites](#).

Customising

URLs

The `Page` class defines a `get_sitemap_urls` method which you can override to customise sitemaps per `Page` instance. This method must accept a request object and 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 `sitemap.xml` template in order for them to be displayed in the sitemap.

Serving multiple sitemaps

If you want to support the sitemap indexes from Django then you will need to use the index view from `wagtail.contrib.sitemaps.views` instead of the index view from `django.contrib.sitemaps.views`. Please see the Django documentation for further details.

Frontend cache invalidator

Many websites use a frontend cache such as Varnish, Squid, Cloudflare or CloudFront 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.frontend_cache`" to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    "wagtail.contrib.frontend_cache"
]
```

The `wagtailfrontendcache` module provides a set of signal handlers which will automatically purge the cache whenever a page is published or deleted. These signal handlers are automatically registered when the `wagtail.contrib.frontend_cache` app is loaded.

Varnish/Squid

Add a new item into the `WAGTAILFRONTENDCACHE` setting and set the `BACKEND` parameter to `wagtail.contrib.frontend_cache.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.frontend_cache.backends.HTTPBackend',
        'LOCATION': 'http://localhost:8000',
    },
}
WAGTAILFRONTENDCACHE_LANGUAGES = []
```

Set `WAGTAILFRONTENDCACHE_LANGUAGES` to a list of languages (typically equal to `[l[0] for l in settings.LANGUAGES]`) to also purge the urls for each language of a purging url. This setting needs `settings.USE_I18N` to be True to work. Its default is an empty list.

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.frontend_cache.backends.CloudflareBackend`.

This backend can be configured to use an account-wide API key, or an API token with restricted access.

To use an account-wide API key, find the key as described in the [Cloudflare documentation](#) and specify `EMAIL` and `API_KEY` parameters.

To use a limited API token, [create a token](#) configured with the ‘Zone, Cache Purge’ permission and specify the `BEARER_TOKEN` parameter.

A `ZONEID` parameter will need to be set for either option. To find the `ZONEID` for your domain, read the [Cloudflare API Documentation](#).

With an API key:

```
# settings.py

WAGTAILFRONTENDCACHE = {
    'cloudflare': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudflareBackend',
        'EMAIL': 'your-cloudflare-email-address@example.com',
        'API_KEY': 'your cloudflare api key',
        'ZONEID': 'your cloudflare domain zone id',
    },
}
```

With an API token:

```
# settings.py

WAGTAILFRONTENDCACHE = {
    'cloudflare': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudflareBackend',
        'BEARER_TOKEN': 'your cloudflare bearer token',
        'ZONEID': 'your cloudflare domain zone id',
    },
}
```

Amazon CloudFront

Within Amazon Web Services you will need at least one CloudFront web distribution. If you don't have one, you can get one here: [CloudFront getting started](#)

Add an item into the `WAGTAILFRONTENDCACHE` and set the `BACKEND` parameter to `wagtail.contrib.frontend_cache.backends.CloudfrontBackend`. This backend requires one extra parameter, `DISTRIBUTION_ID` (your CloudFront generated distribution id).

```
WAGTAILFRONTENDCACHE = {
    'cloudfront': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudfrontBackend',
        'DISTRIBUTION_ID': 'your-distribution-id',
    },
}
```

Configuration of credentials can done in multiple ways. You won't need to store them in your Django settings file. You can read more about this here: [Boto 3 Docs](#).

In case you run multiple sites with Wagtail and each site has its CloudFront distribution, provide a mapping instead of a single distribution. Make sure the mapping matches with the hostnames provided in your site settings.

```
WAGTAILFRONTENDCACHE = {
    'cloudfront': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudfrontBackend',
        'DISTRIBUTION_ID': {
            'www.wagtail.org': 'your-distribution-id',
            'www.madewithwagtail.org': 'your-distribution-id',
        },
    },
}
```

Note: In most cases, absolute URLs with `www` prefixed domain names should be used in your mapping. Only drop the `www` prefix if you're absolutely sure you're not using it (for example a subdomain).

Azure CDN

With [Azure CDN](#) you will need a CDN profile with an endpoint configured.

The third-party dependencies of this backend are:

PyPI Package	Essential	Reason
<code>azure-mgmt-cdn</code>	Yes	Interacting with the CDN service.
<code>azure-identity</code>	No	Obtaining credentials. It's optional if you want to specify your own credential using a <code>CREENTIALS</code> setting (more details below).
<code>azure-mgmt-resources</code>	No	For obtaining the subscription ID. Redundant if you want to explicitly specify a <code>SUBSCRIPTION_ID</code> setting (more details below).

Add an item into the `WAGTAILFRONTENDCACHE` and set the `BACKEND` parameter to `wagtail.contrib.frontend_cache.backends.AzureCdnBackend`. This backend requires the following settings to be set:

- RESOURCE_GROUP_NAME - the resource group that your CDN profile is in.
- CDN_PROFILE_NAME - the profile name of the CDN service that you want to use.
- CDN_ENDPOINT_NAME - the name of the endpoint you want to be purged.

```
WAGTAILFRONTENDCACHE = {
    'azure_cdn': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.AzureCdnBackend',
        'RESOURCE_GROUP_NAME': 'MY-WAGTAIL-RESOURCE-GROUP',
        'CDN_PROFILE_NAME': 'wagtaiilio',
        'CDN_ENDPOINT_NAME': 'wagtaiilio-cdn-endpoint-123',
    },
}
```

By default the credentials will use `azure.identity.DefaultAzureCredential`. To modify the credential object used, please use `CREDENTIALS` setting. Read about your options on the [Azure documentation](#).

```
from azure.common.credentials import ServicePrincipalCredentials

WAGTAILFRONTENDCACHE = {
    'azure_cdn': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.AzureCdnBackend',
        'RESOURCE_GROUP_NAME': 'MY-WAGTAIL-RESOURCE-GROUP',
        'CDN_PROFILE_NAME': 'wagtaiilio',
        'CDN_ENDPOINT_NAME': 'wagtaiilio-cdn-endpoint-123',
        'CREDENTIALS': ServicePrincipalCredentials(
            client_id='your client id',
            secret='your client secret',
        )
    },
}
```

Another option that can be set is `SUBSCRIPTION_ID`. By default the first encountered subscription will be used, but if your credential has access to more subscriptions, you should set this to an explicit value.

Azure Front Door

With [Azure Front Door](#) you will need a Front Door instance with caching enabled.

The third-party dependencies of this backend are:

PyPI Package	Es-sen-tial	Reason
<code>azure-mgmt-frontdoor</code>	Yes	Interacting with the Front Door service.
<code>azure-identity</code>	No	Obtaining credentials. It's optional if you want to specify your own credential using a <code>CREDENTIALS</code> setting (more details below).
<code>azure-mgmt-resource</code>	No	For obtaining the subscription ID. Redundant if you want to explicitly specify a <code>SUBSCRIPTION_ID</code> setting (more details below).

Add an item into the `WAGTAILFRONTENDCACHE` and set the `BACKEND` parameter to `wagtail.contrib.frontend_cache.backends.AzureFrontDoorBackend`. This backend requires the following settings to be set:

- RESOURCE_GROUP_NAME - the resource group that your Front Door instance is part of.

- FRONT_DOOR_NAME - your configured Front Door instance name.

```
WAGTAILFRONTENDCACHE = {
    'azure_front_door': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.AzureFrontDoorBackend',
        'RESOURCE_GROUP_NAME': 'MY-WAGTAIL-RESOURCE-GROUP',
        'FRONT_DOOR_NAME': 'wagtail-io-front-door',
    },
}
```

By default the credentials will use `azure.identity.DefaultAzureCredential`. To modify the credential object used, please use `CREDENTIALS` setting. Read about your options on the [Azure documentation](#).

```
from azure.common.credentials import ServicePrincipalCredentials

WAGTAILFRONTENDCACHE = {
    'azure_front_door': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.AzureFrontDoorBackend',
        'RESOURCE_GROUP_NAME': 'MY-WAGTAIL-RESOURCE-GROUP',
        'FRONT_DOOR_NAME': 'wagtail-io-front-door',
        'CREDENTIALS': ServicePrincipalCredentials(
            client_id='your client id',
            secret='your client secret',
        )
    },
}
```

Another option that can be set is `SUBSCRIPTION_ID`. By default the first encountered subscription will be used, but if your credential has access to more subscriptions, you should set this to an explicit value.

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 + 1):
            yield '/?page=' + str(page_number)
```

Invalidating index pages

Pages that list other pages (such as a blog index) may need to be purged as well so any changes to a blog page are also reflected on the index (for example, a blog post was added, deleted or its title/thumbnaill was changed).

To purge these pages, we need to write a signal handler that listens for Wagtail's `page_published` and `page_unpublished` signals for blog pages (note, `page_published` is called both when a page is created and updated). This signal handler would trigger the invalidation of the index page using the `PurgeBatch` class which is used to construct and dispatch invalidation requests.

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

from wagtail.signals import page_published
from wagtail.contrib.frontend_cache.utils import PurgeBatch

...

def blog_page_changed(blog_page):
    # Find all the live BlogIndexPages that contain this blog_page
    batch = PurgeBatch()
    for blog_index in BlogIndexPage.objects.live():
        if blog_page in blog_index.get_blog_items().object_list:
            batch.add_page(blog_index)

    # Purge all the blog indexes we found in a single request
    batch.purge()

@receiver(page_published, sender=BlogPage)
def blog_published_handler(instance, **kwargs):
    blog_page_changed(instance)

@receiver(pre_delete, sender=BlogPage)
def blog_deleted_handler(instance, **kwargs):
    blog_page_changed(instance)
```

Invalidating URLs

The `PurgeBatch` class provides a `.add_url(url)` and a `.add_urls(urls)` for adding individual URLs to the purge batch.

For example, this could be useful for purging a single page on a blog index:

```
from wagtail.contrib.frontend_cache.utils import PurgeBatch

# Purge the first page of the blog index
batch = PurgeBatch()
batch.add_url(blog_index.url + '?page=1')
batch.purge()
```

The PurgeBatch class

All of the methods available on `PurgeBatch` are listed below:

class wagtail.contrib.frontend_cache.utils.PurgeBatch(urls=None)

Represents a list of URLs to be purged in a single request

add_url(url)

Adds a single URL

add_urls(urls)

Adds multiple URLs from an iterable

This is equivalent to running `.add_url(url)` on each URL individually

add_page(page)

Adds all URLs for the specified page

This combines the page's full URL with each path that is returned by the page's `.get_cached_paths` method

add_pages(pages)

Adds multiple pages from a QuerySet or an iterable

This is equivalent to running `.add_page(page)` on each page individually

purge(backend_settings=None, backends=None)

Performs the purge of all the URLs in this batch

This method takes two optional keyword arguments: `backend_settings` and `backends`

- `backend_settings` can be used to override the `WAGTAILFRONTENDCACHE` setting for just this call
- `backends` can be set to a list of backend names. When set, the invalidation request will only be sent to these backends

RoutablePageMixin

The `RoutablePageMixin` mixin provides a convenient way for a page to respond on multiple sub-URLs with different views. For example, a blog section on a site might provide several different types of index page at URLs like `/blog/2013/06/`, `/blog/authors/bob/`, `/blog/tagged/python/`, all served by the same page instance.

A Page using `RoutablePageMixin` exists within the page tree like any other page, but URL paths underneath it are checked against a list of patterns. If none of the patterns match, control is passed to subpages as usual (or failing that, a 404 error is thrown).

By default a route for `r'^$'` exists, which serves the content exactly like a normal Page would. It can be overridden by using `@re_path(r'^$')` or `@path('')` on any other method of the inheriting class.

Installation

Add "`wagtail.contrib.routable_page`" to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    "wagtail.contrib.routable_page",
]
```

The basics

To use `RoutablePageMixin`, you need to make your class inherit from both `:class:wagtail.contrib.routable_page.models.RoutablePageMixin` and `wagtail.models.Page`, then define some view methods and decorate them with `path` or `re_path`.

These view methods behave like ordinary Django view functions, and must return an `HttpResponse` object; typically this is done through a call to `django.shortcuts.render`.

The `path` and `re_path` decorators from `wagtail.contrib.routable_page.models.path` are similar to [the Django django.urls path and re_path functions](#). The former allows the use of plain paths and converters while the latter lets you specify your URL patterns as regular expressions.

Here's an example of an `EventIndexPage` with three views, assuming that an `EventPage` model with an `event_date` field has been defined elsewhere:

```
import datetime
from django.http import JsonResponse
from wagtail.fields import RichTextField
from wagtail.models import Page
from wagtail.contrib.routable_page.models import RoutablePageMixin, path


class EventIndexPage(RoutablePageMixin, Page):

    # Routable pages can have fields like any other - here we would
    # render the intro text on a template with {{ page.intro/richtext }}
    intro = RichTextField()

    @path('')
    def current_events(self, request):
        """
        View function for the current events page
        """
        events = EventPage.objects.live().filter(event_date__gte=datetime.date.today())

        # NOTE: We can use the RoutablePageMixin.render() method to render
        # the page as normal, but with some of the context values overridden
        return self.render(request, context_overrides={
            'title': "Current events",
            'events': events,
        })

    @path('past/')
    def past_events(self, request):
        """
        View function for the past events page
        """
        events = EventPage.objects.live().filter(event_date__lt=datetime.date.today())

        # NOTE: We are overriding the template here, as well as few context values
        return self.render(
            request,
            context_overrides={
```

(continues on next page)

(continued from previous page)

```

        'title': "Past events",
        'events': events,
    },
    template="events/event_index_historical.html",
)

# Multiple routes!
@path('year/<int:year>/')
@path('year/current/')
def events_for_year(self, request, year=None):
    """
    View function for the events for year page
    """

    if year is None:
        year = datetime.date.today().year

    events = EventPage.objects.live().filter(event_date__year=year)

    return self.render(request, context_overrides={
        'title': "Events for %d" % year,
        'events': events,
    })

@re_path(r'^year/(\d+)/count/$')
def count_for_year(self, request, year=None):
    """
    View function that returns a simple JSON response that
    includes the number of events scheduled for a specific year
    """

    events = EventPage.objects.live().filter(event_date__year=year)

    # NOTE: The usual template/context rendering process is irrelevant
    # here, so we'll just return a HttpResponse directly
    return JsonResponse({'count': events.count()})

```

Rendering other pages

Another way of returning an `HttpResponse` is to call the `serve` method of another page. (Calling a page's own `serve` method within a view method is not valid, as the view method is already being called within `serve`, and this would create a circular definition).

For example, `EventIndexPage` could be extended with a `next/` route that displays the page for the next event:

```

@path('next/')
def next_event(self, request):
    """
    Display the page for the next event
    """

    future_events = EventPage.objects.live().filter(event_date__gt=datetime.date.today())
    next_event = future_events.order_by('event_date').first()

```

(continues on next page)

(continued from previous page)

```
return next_event.serve(request)
```

Reversing URLs

RoutablePageMixin adds a `reverse_subpage()` method to your page model which you can use for reversing URLs. For example:

```
# The URL name defaults to the view method name.
>>> event_page.reverse_subpage('events_for_year', args=(2015, ))
'year/2015/'
```

This method only returns the part of the URL within the page. To get the full URL, you must append it to the values of either the `url` or the `full_url` attribute on your page:

```
>>> event_page.url + event_page.reverse_subpage('events_for_year', args=(2015, ))
'/events/year/2015/'

>>> event_page.full_url + event_page.reverse_subpage('events_for_year', args=(2015, ))
'http://example.com/events/year/2015/'
```

Changing route names

The route name defaults to the name of the view. You can override this name with the `name` keyword argument on `@path` or `re_path`:

```
from wagtail.models import Page
from wagtail.contrib.routable_page.models import RoutablePageMixin, route

class EventPage(RoutablePageMixin, Page):
    ...

    @re_path(r'^year/(\d+)/$', name='year')
    def events_for_year(self, request, year):
        """
        View function for the events for year page
        """
        ...


```

```
>>> event_page.url + event_page.reverse_subpage('year', args=(2015, ))
'/events/year/2015/'
```

The RoutablePageMixin class

```
class wagtail.contrib.routable_page.models.RoutablePageMixin
```

This class can be mixed in to a Page model, allowing extra routes to be added to it.

```
render(request, *args, template=None, context_overrides=None, **kwargs)
```

This method replicates what `Page.serve()` usually does when `RoutablePageMixin` is not used. By default, `Page.get_template()` is called to derive the template to use for rendering, and `Page.get_context()` is always called to gather the data to be included in the context.

You can use the `context_overrides` keyword argument as a shortcut to override or add new values to the context. For example:

```
@path('')
def upcoming_events(self, request):
    return self.render(request, context_overrides={
        'title': "Current events",
        'events': EventPage.objects.live().future(),
    })
```

You can also use the `template` argument to specify an alternative template to use for rendering. For example:

```
@path('past/')
def past_events(self, request):
    return self.render(
        request,
        context_overrides={
            'title': "Past events",
            'events': EventPage.objects.live().past(),
        },
        template="events/event_index_historical.html",
    )
```

```
classmethod get_subpage_urls()
```

```
resolve_subpage(path)
```

This method takes a URL path and finds the view to call.

Example:

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

```
reverse_subpage(name, args=None, kwargs=None)
```

This method takes a route name/arguments and returns a URL path.

Example:

```
url = page.url + page.reverse_subpage('events_for_year', kwargs={'year': '2014'})
```

The routablepageurl template tag

```
wagtail.contrib.routable_page.templatetags.wagtailroutablepage_tags.routablepageurl(context,
    page,
    url_name,
    *args,
    **kwargs)
```

`routablepageurl` is similar to `pageurl`, but works with pages using `RoutablePageMixin`. It behaves like a hybrid between the built-in `reverse`, and `pageurl` from Wagtail.

`page` is the `RoutablePage` that URLs will be generated from.

`url_name` is a URL name defined in `page.subpage_urls`.

Positional arguments and keyword arguments should be passed as normal positional arguments and keyword arguments.

Example:

```
{% load wagtailroutablepage_tags %}

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

ModelAdmin

The `modeladmin` module allows you to add any model in your project to the Wagtail admin. You can create customisable listing pages for a model, including plain Django models, and add navigation elements so that a model can be accessed directly from the Wagtail admin. Simply extend the `ModelAdmin` class, override a few attributes to suit your needs, register it with Wagtail using an easy one-line `modeladmin_register` method (you can copy and paste from the examples below), and you're good to go. Your model doesn't need to extend `Page` or be registered as a `Snippet`, and it won't interfere with any of the existing admin functionality that Wagtail provides.

Summary of features

- A customisable list view, allowing you to control what values are displayed for each row, available options for result filtering, default ordering, spreadsheet downloads and more.
- Access your list views from the Wagtail admin menu easily with automatically generated menu items, with automatic ‘active item’ highlighting. Control the label text and icons used with easy-to-change attributes on your class.
- An additional `ModelAdminGroup` class, that allows you to group your related models, and list them together in their own submenu, for a more logical user experience.
- Simple, robust `add` and `edit` views for your non-`Page` models that use the panel configurations defined on your model using Wagtail’s edit panels.
- For `Page` models, the system directs to Wagtail’s existing add and edit views, and returns you back to the correct list page, for a seamless experience.
- Full respect for permissions assigned to your Wagtail users and groups. Users will only be able to do what you want them to!

- All you need to easily hook your `ModelAdmin` classes into Wagtail, taking care of URL registration, menu changes, and registering any missing model permissions, so that you can assign them to Groups.
- **Built to be customisable** - While `modeladmin` provides a solid experience out of the box, you can easily use your own templates, and the `ModelAdmin` class has a large number of methods that you can override or extend, allowing you to customise the behaviour to a greater degree.

Want to know more about customising `ModelAdmin`?

`modeladmin` customisation primer

The `modeladmin` app is designed to offer you as much flexibility as possible in how your model and its objects are represented in Wagtail's CMS. This page aims to provide you with some background information to help you gain a better understanding of what the app can do, and to point you in the right direction, depending on the kind of customisations you're looking to make.

- *Wagtail's ModelAdmin class isn't the same as Django's*
- *Changing what appears in the listing*
- *Adding additional stylesheets and/or JavaScript*
- *Overriding templates*
- *Overriding views*
- *Overriding helper classes*

Wagtail's ModelAdmin class isn't the same as Django's

Wagtail's `ModelAdmin` class is designed to be used in a similar way to Django's class of the same name, and it often uses the same attribute and method names to achieve similar things. However, there are a few key differences:

Add & edit forms are still defined by panels and `edit_handlers`

In Wagtail, controlling which fields appear in add/edit forms for your `Model`, and defining how they are grouped and ordered, is achieved by adding a `panels` attribute, or `edit_handler` to your `Model` class. This remains the same whether your model is a `Page` type, a snippet, or just a standard Django `Model`. Because of this, Wagtail's `ModelAdmin` class is mostly concerned with 'listing' configuration. For example, `list_display`, `list_filter` and `search_fields` attributes are present and support largely the same values as Django's `ModelAdmin` class, while `fields`, `fieldsets`, `exclude` and other attributes you may be used to using to configure Django's add/edit views, simply aren't supported by Wagtail's version.

'Page type' models need to be treated differently from other models

While `modeladmin`'s listing view and its supported customisation options work in exactly the same way for all types of `Model`, when it comes to the other management views, the treatment differs depending on whether your `ModelAdmin` class is representing a page type model (that extends `wagtailcore.models.Page`) or not.

Pages in Wagtail have some unique properties, and require additional views, interface elements and general treatment in order to be managed effectively. For example, they have a tree structure that must be preserved properly as pages are added, deleted and moved around. They also have a revisions system, their own permission considerations, and the facility to preview changes before saving changes. Because of this added complexity, Wagtail provides its own specific views for managing any custom page types you might add to your project (whether you create a `ModelAdmin` class for them or not).

In order to deliver a consistent user experience, `modeladmin` simply redirects users to Wagtail's existing page management views wherever possible. You should bear this in mind if you ever find yourself wanting to change what happens when pages of a certain type are added, deleted, published, or have some other action applied to them. Customising the `CreateView` or `EditView` for your page type `Model` (even if just to add an additional stylesheet or JavaScript), simply won't have any effect, as those views are not used.

If you do find yourself needing to customise the add, edit or other behaviour for a page type model, you should take a look at the following part of the documentation: [Hooks](#).

Wagtail's `ModelAdmin` class is 'modular'

Unlike Django's class of the same name, `wagtailadmin`'s `ModelAdmin` acts primarily as a 'controller' class. While it does have a set of attributes and methods to enable you to configure how various components should treat your model, it has been deliberately designed to do as little work as possible by itself; it designates all of the real work to a set of separate, swappable components.

The theory is: If you want to do something differently, or add some functionality that `modeladmin` doesn't already have, you can create new classes (or extend the ones provided by `modeladmin`) and easily configure your `ModelAdmin` class to use them instead of the defaults.

- Learn more about [Overriding views](#)
- Learn more about [Overriding helper classes](#)

Changing what appears in the listing

You should familiarise yourself with the attributes and methods supported by the `ModelAdmin` class, that allow you to change what is displayed in the `IndexView`. The following page should give you everything you need to get going: [Customising IndexView - the listing view](#)

Adding additional stylesheets and/or JavaScript

The `ModelAdmin` class provides several attributes to enable you to easily add additional stylesheets and JavaScript to the admin interface for your model. Each attribute simply needs to be a list of paths to the files you want to include. If the path is for a file in your project's static directory, then Wagtail will automatically prepend the path with `STATIC_URL` so that you don't need to repeat it each time in your list of paths.

If you'd like to add styles or scripts to the `IndexView`, you should set the following attributes:

- `index_view_extra_css` - Where each item is the path name of a pre-compiled stylesheet that you'd like to include.

- `index_view_extra_js` - Where each item is the path name of a JavaScript file that you'd like to include.

If you'd like to do the same for `CreateView` and `EditView`, you should set the following attributes:

- `form_view_extra_css` - Where each item is the path name of a pre-compiled stylesheet that you'd like to include.
- `form_view_extra_js` - Where each item is the path name of a JavaScript file that you'd like to include.

And if you're using the `InspectView` for your model, and want to do the same for that view, you should set the following attributes:

- `inspect_view_extra_css` - Where each item is the path name of a pre-compiled stylesheet that you'd like to include.
- `inspect_view_extra_js` - Where each item is the path name of a JavaScript file that you'd like to include.

Overriding templates

For all `modeladmin` views, Wagtail looks for templates in the following folders within your project or app, before resorting to the defaults:

1. `templates/modeladmin/app-name/model-name/`
2. `templates/modeladmin/app-name/`
3. `templates/modeladmin/`

So, to override the template used by `IndexView` for example, you'd create a new `index.html` template and put it in one of those locations. For example, if you wanted to do this for an `ArticlePage` model in a `news` app, you'd add your custom template as `news/templates/modeladmin/news/articlepage/index.html`.

For reference, `modeladmin` looks for templates with the following names for each view:

- '`index.html`' for `IndexView`
- '`inspect.html`' for `InspectView`
- '`create.html`' for `CreateView`
- '`edit.html`' for `EditView`
- '`delete.html`' for `DeleteView`
- '`choose_parent.html`' for `ChooseParentView`

To add extra information to a block within one of the above Wagtail templates, use Django's `{{ block.super }}` within the `{% block ... %}` that you wish to extend. For example, if you wish to display an image in an edit form below the fields of the model that is being edited, you could do the following:

```
{% extends "modeladmin/edit.html" %}  
{% load static %}  
  
{% block content %}  
    {{ block.super }}  
    <div>  
          
    </div>  
{% endblock %}
```

If for any reason you'd rather bypass the above behaviour and explicitly specify a template for a specific view, you can set either of the following attributes on your `ModelAdmin` class:

- `index_template_name` to specify a template for `IndexView`
- `inspect_template_name` to specify a template for `InspectView`
- `create_template_name` to specify a template for `CreateView`
- `edit_template_name` to specify a template for `EditView`
- `delete_template_name` to specify a template for `DeleteView`
- `choose_parent_template_name` to specify a template for `ChooseParentView`

Overriding views

For all of the views offered by `ModelAdmin`, the class provides an attribute that you can override in order to tell it which class you'd like to use:

- `index_view_class`
- `inspect_view_class`
- `create_view_class` (not used for ‘page type’ models)
- `edit_view_class` (not used for ‘page type’ models)
- `delete_view_class` (not used for ‘page type’ models)
- `choose_parent_view_class` (only used for ‘page type’ models)

For example, if you'd like to create your own view class and use it for the `IndexView`, you would do the following:

```
from wagtail.contrib.modeladmin.views import IndexView
from wagtail.contrib.modeladmin.options import ModelAdmin
from .models import MyModel

class MyCustomIndexView(IndexView):
    # New functionality and existing method overrides added here
    ...

class MyModelAdmin(ModelAdmin):
    model = MyModel
    index_view_class = MyCustomIndexView
```

Or, if you have no need for any of `IndexView`'s existing functionality in your view and would rather create your own view from scratch, `modeladmin` will support that too. However, it's highly recommended that you use `modeladmin.views.WMABaseView` as a base for your view. It'll make integrating with your `ModelAdmin` class much easier and will provide a bunch of useful attributes and methods to get you started.

You can also use the `url_helper` to easily reverse URLs for any `ModelAdmin` see [Reversing ModelAdmin URLs](#).

Overriding helper classes

While ‘view classes’ are responsible for a lot of the work, there are also a number of other tasks that `modeladmin` must do regularly, that need to be handled in a consistent way, and in a number of different places. These tasks are designated to a set of simple classes (in `modeladmin`, these are termed ‘helper’ classes) and can be found in `wagtail.contrib.modeladmin.helpers`.

If you ever intend to write and use your own custom views with `modeladmin`, you should familiarise yourself with these helpers, as they are made available to views via the `modeladmin.views.WMABaseView` view.

There are three types of ‘helper class’:

- **URL helpers** - That help with the consistent generation, naming and referencing of urls.
- **Permission helpers** - That help with ensuring only users with sufficient permissions can perform certain actions, or see options to perform those actions.
- **Button helpers** - That, with the help of the other two, helps with the generation of buttons for use in a number of places.

The `ModelAdmin` class allows you to define and use your own helper classes by setting values on the following attributes:

`ModelAdmin.url_helper_class`

By default, the `modeladmin.helpers.url.PageAdminURLHelper` class is used when your model extends `wagtailcore.models.Page`, otherwise `modeladmin.helpers.url.AdminURLHelper` is used.

If you find that the above helper classes don’t work for your needs, you can easily create your own helper class by sub-classing `AdminURLHelper` or `PageAdminURLHelper` (if your model extends Wagtail’s Page model), and making any necessary additions/overrides.

Once your class is defined, set the `url_helper_class` attribute on your `ModelAdmin` class to use your custom URL-Helper, like so:

```
from wagtail.contrib.modeladmin.helpers import AdminURLHelper
from wagtail.contrib.modeladmin.options import ModelAdmin, modeladmin_register
from .models import MyModel

class MyURLHelper(AdminURLHelper):
    ...

class MyModelAdmin(ModelAdmin):
    model = MyModel
    url_helper_class = MyURLHelper

modeladmin_register(MyModelAdmin)
```

Or, if you have a more complicated use case, where simply setting that attribute isn’t possible (due to circular imports, for example) or doesn’t meet your needs, you can override the `get_url_helper_class` method, like so:

```
class MyModelAdmin(ModelAdmin):
    model = MyModel

    def get_url_helper_class(self):
```

(continues on next page)

(continued from previous page)

```
if self.some_attribute is True:
    return MyURLHelper
return AdminURLHelper
```

ModelAdmin.permission_helper_class

By default, the `modeladmin.helpers.permission.PagePermissionHelper` class is used when your model extends `wagtailcore.models.Page`, otherwise `modeladmin.helpers.permission.PermissionHelper` is used.

If you find that the above helper classes don't work for your needs, you can easily create your own helper class, by sub-classing `PermissionHelper` (or `PagePermissionHelper` if your model extends Wagtail's `Page` model), and making any necessary additions/overrides. Once defined, you set the `permission_helper_class` attribute on your `ModelAdmin` class to use your custom class instead of the default, like so:

```
from wagtail.contrib.modeladmin.helpers import PermissionHelper
from wagtail.contrib.modeladmin.options import ModelAdmin, modeladmin_register
from .models import MyModel

class MyPermissionHelper(PermissionHelper):
    ...

class MyModelAdmin(ModelAdmin):
    model = MyModel
    permission_helper_class = MyPermissionHelper

modeladmin_register(MyModelAdmin)
```

Or, if you have a more complicated use case, where simply setting an attribute isn't possible or doesn't meet your needs, you can override the `get_permission_helper_class` method, like so:

```
class MyModelAdmin(ModelAdmin):
    model = MyModel

    def get_permission_helper_class(self):
        if self.some_attribute is True:
            return MyPermissionHelper
        return PermissionHelper
```

ModelAdmin.button_helper_class

By default, the `modeladmin.helpers.button.PageButtonHelper` class is used when your model extends `wagtailcore.models.Page`, otherwise `modeladmin.helpers.button.ButtonHelper` is used.

If you wish to add or change buttons for your model's `IndexView`, you'll need to create your own button helper class by sub-classing `ButtonHelper` or `PageButtonHelper` (if your model extend's Wagtail's `Page` model), and make any necessary additions/overrides. Once defined, you set the `button_helper_class` attribute on your `ModelAdmin` class to use your custom class instead of the default, like so:

```

from wagtail.contrib.modeladmin.helpers import ButtonHelper
from wagtail.contrib.modeladmin.options import ModelAdmin, modeladmin_register
from .models import MyModel


class MyButtonHelper(ButtonHelper):
    def add_button(self, classnames_add=None, classnames_exclude=None):
        if classnames_add is None:
            classnames_add = []
        if classnames_exclude is None:
            classnames_exclude = []
        classnames = self.add_button_classnames + classnames_add
        cn = self.finalise_classname(classnames, classnames_exclude)
        return {
            'url': self.url_helper.create_url,
            'label': _('Add %s') % self.verbose_name,
            'classname': cn,
            'title': _('Add a new %s') % self.verbose_name,
        }

    def inspect_button(self, pk, classnames_add=None, classnames_exclude=None):
        ...

    def edit_button(self, pk, classnames_add=None, classnames_exclude=None):
        ...

    def delete_button(self, pk, classnames_add=None, classnames_exclude=None):
        ...


class MyModelAdmin(ModelAdmin):
    model = MyModel
    button_helper_class = MyButtonHelper

modeladmin_register(MyModelAdmin)

```

To customise the buttons found in the ModelAdmin List View you can change the returned dictionary in the `add_button`, `delete_button`, `edit_button` or `inspect_button` methods. For example if you wanted to change the Delete button you could modify the `delete_button` method in your `ButtonHelper` like so:

```

class MyButtonHelper(ButtonHelper):
    ...
    def delete_button(self, pk, classnames_add=None, classnames_exclude=None):
        ...
        return {
            'url': reverse("your_custom_url"),
            'label': _('Delete'),
            'classname': "custom-css-class",
            'title': _('Delete this item')
        }

```

Or, if you have a more complicated use case, where simply setting an attribute isn't possible or doesn't meet your needs, you can override the `get_button_helper_class` method, like so:

```
class MyModelAdmin(ModelAdmin):
    model = MyModel

    def get_button_helper_class(self):
        if self.some_attribute is True:
            return MyButtonHelper
        return ButtonHelper
```

Using helpers in your custom views

As long as you sub-class `modeladmin.views.WMABaseView` (or one of the more ‘specific’ view classes) to create your custom view, instances of each helper should be available on instances of your class as:

- `self.url_helper`
- `self.permission_helper`
- `self.button_helper`

Unlike the other two, `self.button_helper` isn’t populated right away when the view is instantiated. In order to show the right buttons for the right users, `ButtonHelper` instances need to be ‘request aware’, so `self.button_helper` is only set once the view’s `dispatch()` method has run, which takes a `HttpRequest` object as an argument, from which the current user can be identified.

Customising the base URL path

You can use the following attributes and methods on the `ModelAdmin` class to alter the base URL path used to represent your model in Wagtail’s admin area.

- `ModelAdmin.base_url_path`

`ModelAdmin.base_url_path`

Expected value: A string.

Set this attribute to a string value to override the default base URL path used for the model to `admin/{base_url_path}`. If not set, the base URL path will be `admin/{app_label}/{model_name}`.

Customising the menu item

You can use the following attributes and methods on the `ModelAdmin` class to alter the menu item used to represent your model in Wagtail’s admin area.

- `ModelAdmin.menu_label`
- `ModelAdmin.menu_icon`
- `ModelAdmin.menu_order`
- `ModelAdmin.add_to_settings_menu`

- `ModelAdmin.add_to_admin_menu`
- `ModelAdmin.menu_item_name`

`ModelAdmin.menu_label`

Expected value: A string.

Set this attribute to a string value to override the label used for the menu item that appears in Wagtail's sidebar. If not set, the menu item will use `verbose_name_plural` from your model's Meta data.

`ModelAdmin.menu_icon`

Expected value: A string matching one of Wagtail's icon class names.

If you want to change the icon used to represent your model, you can set the `menu_icon` attribute on your class to use one of the other icons available in Wagtail's CMS. The same icon will be used for the menu item in Wagtail's sidebar, and will also appear in the header on the list page and other views for your model. If not set, '`doc-full-inverse`' will be used for page-type models, and '`snippet`' for others.

If you're using a `ModelAdminGroup` class to group together several `ModelAdmin` classes in their own sub-menu, and want to change the menu item used to represent the group, you should override the `menu_icon` attribute on your `ModelAdminGroup` class ('`folder-open-inverse`' is the default).

`ModelAdmin.menu_order`

Expected value: An integer between 1 and 999.

If you want to change the position of the menu item for your model (or group of models) in Wagtail's sidebar, you do that by setting `menu_order`. The value should be an integer between 1 and 999. The lower the value, the higher up the menu item will appear.

Wagtail's 'Explorer' menu item has an order value of `100`, so supply a value greater than that if you wish to keep the explorer menu item at the top.

`ModelAdmin.add_to_settings_menu`

Expected value: True or False

If you'd like the menu item for your model to appear in Wagtail's 'Settings' sub-menu instead of at the top level, add `add_to_settings_menu = True` to your `ModelAdmin` class.

This will only work for individual `ModelAdmin` classes registered with their own `modeladmin_register` call. It won't work for members of a `ModelAdminGroup`.

`ModelAdmin.add_to_admin_menu`

Expected value: True or False

If you'd like this model admin to be excluded from the menu, set to False.

`ModelAdmin.menu_item_name`

Expected value: A string or None

Passed on as the name parameter when initialising the `MenuItem` for this class, becoming the name attribute value for that instance.

Customising IndexView - the listing view

For the sake of consistency, this section of the docs will refer to the listing view as `IndexView`, because that is the view class that does all the heavy lifting.

You can use the following attributes and methods on the `ModelAdmin` class to alter how your model data is treated and represented by the `IndexView`.

- `ModelAdmin.list_display`
- `ModelAdmin.list_export`
- `ModelAdmin.list_filter`
- `ModelAdmin.export_filename`
- `ModelAdmin.search_fields`
- `ModelAdmin.search_handler_class`
- `ModelAdmin.extra_search_kwargs`
- `ModelAdmin.ordering`
- `ModelAdmin.list_per_page`
- `ModelAdmin.get_queryset()`
- `ModelAdmin.get_extra_attrs_for_row()`
- `ModelAdmin.get_extra_class_names_for_field_col()`
- `ModelAdmin.get_extra_attrs_for_field_col()`
- `wagtail.contrib.modeladmin.mixins.ThumbnailMixin`
- `ModelAdmin.list_display_add_buttons`
- `ModelAdmin.index_view_extra_css`
- `ModelAdmin.index_view_extra_js`
- `ModelAdmin.index_template_name`
- `ModelAdmin.index_view_class`

ModelAdmin.list_display

Expected value: A list or tuple, where each item is the name of a field or single-argument callable on your model, or a similarly simple method defined on the `ModelAdmin` class itself.

Default value: `('__str__',)`

Set `list_display` to control which fields are displayed in the `IndexView` for your model.

You have three possible values that can be used in `list_display`:

- A field of the model. For example:

```
from wagtail.contrib.modeladmin.options import ModelAdmin
from .models import Person

class PersonAdmin(ModelAdmin):
    model = Person
    list_display = ('first_name', 'last_name')
```

- The name of a custom method on your `ModelAdmin` class, that accepts a single parameter for the model instance. For example:

```
from wagtail.contrib.modeladmin.options import ModelAdmin
from .models import Person

class PersonAdmin(ModelAdmin):
    model = Person
    list_display = ('upper_case_name',)

    def upper_case_name(self, obj):
        return ("%s %s" % (obj.first_name, obj.last_name)).upper()
    upper_case_name.short_description = 'Name'
```

- The name of a method on your `Model` class that accepts only `self` as an argument. For example:

```
from django.db import models
from wagtail.contrib.modeladmin.options import ModelAdmin

class Person(models.Model):
    name = models.CharField(max_length=50)
    birthday = models.DateField()

    def decade_born_in(self):
        return self.birthday.strftime('%Y')[:3] + "'s"
    decade_born_in.short_description = 'Birth decade'

class PersonAdmin(ModelAdmin):
    model = Person
    list_display = ('name', 'decade_born_in')
```

A few special cases to note about `list_display`:

- If the field is a `ForeignKey`, Django will display the output of `__str__()` of the related object.

- If the string provided is a method of the model or ModelAdmin class, Django will HTML-escape the output by default. To escape user input and allow your own unescaped tags, use `format_html()`. For example:

```
from django.db import models
from django.utils.html import format_html
from wagtail.contrib.modeladmin.options import ModelAdmin

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def styled_name(self):
        return format_html(
            '<span style="color: #{};">{} {}</span>',
            self.color_code,
            self.first_name,
            self.last_name,
        )

class PersonAdmin(ModelAdmin):
    model = Person
    list_display = ('first_name', 'last_name', 'styled_name')
```

- If the value of a field is `None`, an empty string, or an iterable without elements, Wagtail will display a dash (-) for that column. You can override this by setting `empty_value_display` on your ModelAdmin class. For example:

```
from wagtail.contrib.modeladmin.options import ModelAdmin

class PersonAdmin(ModelAdmin):
    empty_value_display = 'N/A'
    ...
```

Or, if you'd like to change the value used depending on the field, you can override ModelAdmin's `get_empty_value_display()` method, like so:

```
from django.db import models
from wagtail.contrib.modeladmin.options import ModelAdmin

class Person(models.Model):
    name = models.CharField(max_length=100)
    nickname = models.CharField(blank=True, max_length=100)
    likes_cat_gifs = models.NullBooleanField()

class PersonAdmin(ModelAdmin):
    model = Person
    list_display = ('name', 'nickname', 'likes_cat_gifs')

    def get_empty_value_display(self, field_name=None):
        if field_name == 'nickname':
            return 'None given'
```

(continues on next page)

(continued from previous page)

```
if field_name == 'likes_cat_gifs':
    return 'Unanswered'
return super().get_empty_value_display(field_name)
```

The `__str__()` method is just as valid in `list_display` as any other model method, so it's perfectly OK to do this:

```
list_display = ('__str__', 'some_other_field')
```

By default, the ability to sort results by an item in `list_display` is only offered when it's a field that has an actual database value (because sorting is done at the database level). However, if the output of the method is representative of a database field, you can indicate this fact by setting the `admin_order_field` attribute on that method, like so:

```
from django.db import models
from django.utils.html import format_html
from wagtail.contrib.modeladmin.options import ModelAdmin

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def styled_first_name(self):
        return format_html(
            '<span style="color: #{}; ">{}</span>',
            self.color_code,
            self.first_name,
        )
    styled_first_name.admin_order_field = 'first_name'

class PersonAdmin(ModelAdmin):
    model = Person
    list_display = ('styled_first_name', 'last_name')
```

The above will tell Wagtail to order by the `first_name` field when trying to sort by `styled_first_name` in the index view.

The above will tell Wagtail to order by the `first_name` field when trying to sort by `styled_first_name` in the index view.

To indicate descending order with `admin_order_field` you can use a hyphen prefix on the field name. Using the above example, this would look like:

.. code-block:: python

```
styled_first_name.admin_order_field = '-first_name'
```

`admin_order_field` supports query lookups to sort by values on related models, too. This example includes an “author first name” column in the list display and allows sorting it by first name:

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

(continues on next page)

(continued from previous page)

```
class Blog(models.Model):
    title = models.CharField(max_length=255)
    author = models.ForeignKey(Person, on_delete=models.CASCADE)

    def author_first_name(self, obj):
        return obj.author.first_name

    author_first_name.admin_order_field = 'author__first_name'
```

- Elements of `list_display` can also be properties. Please note however, that due to the way properties work in Python, setting `short_description` on a property is only possible when using the `property()` function and `not` with the `@property` decorator.

For example:

```
from django.db import models
from wagtail.contrib.modeladmin.options import ModelAdmin

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def full_name_property(self):
        return self.first_name + ' ' + self.last_name
    full_name_property.short_description = "Full name of the person"

    full_name = property(full_name_property)

class PersonAdmin(ModelAdmin):
    list_display = ('full_name',)
```

ModelAdmin.list_export

Expected value: A list or tuple, where each item is the name of a field or single-argument callable on your model, or a similarly simple method defined on the `ModelAdmin` class itself.

Set `list_export` to set the fields you wish to be exported as columns when downloading a spreadsheet version of your `index_view`

```
class PersonAdmin(ModelAdmin):
    list_export = ('is_staff', 'company')
```

`ModelAdmin.list_filter`

Expected value: A list or tuple, where each item is the name of model field of type BooleanField, CharField, DateField, DateTimeField, IntegerField or ForeignKey.

Set `list_filter` to activate filters in the right sidebar of the list page for your model. For example:

```
class PersonAdmin(ModelAdmin):
    list_filter = ('is_staff', 'company')
```

`ModelAdmin.export_filename`

Expected value: A string specifying the filename of an exported spreadsheet, without file extensions.

```
class PersonAdmin(ModelAdmin):
    export_filename = 'people_spreadsheet'
```

`ModelAdmin.search_fields`

Expected value: A list or tuple, where each item is the name of a model field of type CharField, TextField, RichTextField or StreamField.

Set `search_fields` to enable a search box at the top of the index page for your model. You should add names of any fields on the model that should be searched whenever somebody submits a search query using the search box.

Searching is handled via Django's QuerySet API by default, see [`ModelAdmin.search_handler_class`](#) about changing this behaviour. This means by default it will work for all models, whatever search backend your project is using, and without any additional setup or configuration.

`ModelAdmin.search_handler_class`

Expected value: A subclass of `wagtail.contrib.modeladmin.helpers.search.BaseSearchHandler`

The default value is `DjangoORMSearchHandler`, which uses the Django ORM to perform lookups on the fields specified by `search_fields`.

If you would prefer to use the built-in Wagtail search backend to search your models, you can use the `WagtailBackendSearchHandler` class instead. For example:

```
from wagtail.contrib.modeladmin.helpers import WagtailBackendSearchHandler

from .models import Person

class PersonAdmin(ModelAdmin):
    model = Person
    search_handler_class = WagtailBackendSearchHandler
```

Extra considerations when using WagtailBackendSearchHandler

ModelAdmin.search_fields is used differently

The value of `search_fields` is passed to the underlying search backend to limit the fields used when matching. Each item in the list must be indexed on your model using `index.SearchField`.

To allow matching on **any** indexed field, set the `search_fields` attribute on your `ModelAdmin` class to `None`, or remove it completely.

Indexing extra fields using index.FilterField

The underlying search backend must be able to interpret all of the fields and relationships used in the queryset created by `IndexView`, including those used in `prefetch()` or `select_related()` queryset methods, or used in `list_display`, `list_filter` or `ordering`.

Be sure to test things thoroughly in a development environment (ideally using the same search backend as you use in production). Wagtail will raise an `IndexError` if the backend encounters something it does not understand, and will tell you what you need to change.

ModelAdmin.extra_search_kwargs

Expected value: A dictionary of keyword arguments that will be passed on to the `search()` method of `search_handler_class`.

For example, to override the `WagtailBackendSearchHandler` default operator you could do the following:

```
from wagtail.contrib.modeladmin.helpers import WagtailBackendSearchHandler
from wagtail.search.utils import OR

from .models import IndexedModel

class DemoAdmin(ModelAdmin):
    model = IndexedModel
    search_handler_class = WagtailBackendSearchHandler
    extra_search_kwargs = {'operator': OR}
```

ModelAdmin.ordering

Expected value: A list or tuple in the same format as a model's `ordering` parameter.

Set `ordering` to specify the default ordering of objects when listed by `IndexView`. If not provided, the model's default ordering will be respected.

If you need to specify a dynamic order (for example, depending on user or language) you can override the `get_ordering()` method instead.

ModelAdmin.list_per_page

Expected value: A positive integer

Set `list_per_page` to control how many items appear on each paginated page of the index view. By default, this is set to `100`.

ModelAdmin.get_queryset()

Must return: A QuerySet

The `get_queryset` method returns the ‘base’ QuerySet for your model, to which any filters and search queries are applied. By default, the `all()` method of your model’s default manager is used. But, if for any reason you only want a certain sub-set of objects to appear in the IndexView listing, overriding the `get_queryset` method on your `ModelAdmin` class can help you with that. The method takes an `HttpRequest` object as a parameter, so limiting objects by the current logged-in user is possible.

For example:

```
from django.db import models
from wagtail.contrib.modeladmin.options import ModelAdmin

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    managed_by = models.ForeignKey('auth.User', on_delete=models.CASCADE)

class PersonAdmin(ModelAdmin):
    model = Person
    list_display = ('first_name', 'last_name')

    def get_queryset(self, request):
        qs = super().get_queryset(request)
        # Only show people managed by the current user
        return qs.filter(managed_by=request.user)
```

ModelAdmin.get_extra_attrs_for_row()

Must return: A dictionary

The `get_extra_attrs_for_row` method allows you to add html attributes to the opening `<tr>` tag for each result, in addition to the `data-object_pk` and `class` attributes already added by the `result_row_display` template tag.

If you want to add additional CSS classes, simply provide those class names as a string value using the ‘`class`’ key, and the `odd/even` will be appended to your custom class names when rendering.

For example, if you wanted to add some additional class names based on field values, you could do something like:

```
from decimal import Decimal
from django.db import models
from wagtail.contrib.modeladmin.options import ModelAdmin

class BankAccount(models.Model):
```

(continues on next page)

(continued from previous page)

```

name = models.CharField(max_length=50)
account_number = models.CharField(max_length=50)
balance = models.DecimalField(max_digits=5, num_places=2)

class BankAccountAdmin(ModelAdmin):
    list_display = ('name', 'account_number', 'balance')

    def get_extra_attrs_for_row(self, obj, context):
        if obj.balance < Decimal('0.00'):
            classname = 'balance-negative'
        else:
            classname = 'balance-positive'
        return {
            'class': classname,
        }
    
```

`ModelAdmin.get_extra_class_names_for_field_col()`

Must return: A list

The `get_extra_class_names_for_field_col` method allows you to add additional CSS class names to any of the columns defined by `list_display` for your model. The method takes two parameters:

- `obj`: the object being represented by the current row
- `field_name`: the item from `list_display` being represented by the current column

For example, if you'd like to apply some conditional formatting to a cell depending on the row's value, you could do something like:

```

from decimal import Decimal
from django.db import models
from wagtail.contrib.modeladmin.options import ModelAdmin

class BankAccount(models.Model):
    name = models.CharField(max_length=50)
    account_number = models.CharField(max_length=50)
    balance = models.DecimalField(max_digits=5, num_places=2)

class BankAccountAdmin(ModelAdmin):
    list_display = ('name', 'account_number', 'balance')

    def get_extra_class_names_for_field_col(self, obj, field_name):
        if field_name == 'balance':
            if obj.balance <= Decimal('-100.00'):
                return ['brand-danger']
            elif obj.balance <= Decimal('0.00'):
                return ['brand-warning']
            elif obj.balance <= Decimal('50.00'):
                return ['brand-info']
            else:
                return []
    
```

(continues on next page)

(continued from previous page)

```

    return ['brand-success']
return []

```

ModelAdmin.get_extra_attrs_for_field_col()**Must return:** A dictionary

The `get_extra_attrs_for_field_col` method allows you to add additional HTML attributes to any of the columns defined in `list_display`. Like the `get_extra_class_names_for_field_col` method above, this method takes two parameters:

- `obj`: the object being represented by the current row
- `field_name`: the item from `list_display` being represented by the current column

For example, you might like to add some tooltip text to a certain column, to help give the value more context:

```

from django.db import models
from wagtail.contrib.modeladmin.options import ModelAdmin

class Person(models.Model):
    name = models.CharField(max_length=100)
    likes_cat_gifs = models.NullBooleanField()

class PersonAdmin(ModelAdmin):
    model = Person
    list_display = ('name', 'likes_cat_gifs')

    def get_extra_attrs_for_field_col(self, obj, field_name=None):
        attrs = super().get_extra_attrs_for_field_col(obj, field_name)
        if field_name == 'likes_cat_gifs' and obj.likes_cat_gifs is None:
            attrs.update({
                'title': (
                    'The person was shown several cat gifs, but failed to '
                    'indicate a preference.'
                ),
            })
        return attrs

```

Or you might like to add one or more data attributes to help implement some kind of interactivity using JavaScript:

```

from django.db import models
from wagtail.contrib.modeladmin.options import ModelAdmin

class Event(models.Model):
    title = models.CharField(max_length=255)
    start_date = models.DateField()
    end_date = models.DateField()
    start_time = models.TimeField()
    end_time = models.TimeField()

```

(continues on next page)

(continued from previous page)

```

class EventAdmin(ModelAdmin):
    model = Event
    list_display = ('title', 'start_date', 'end_date')

    def get_extra_attrs_for_field_col(self, obj, field_name=None):
        attrs = super().get_extra_attrs_for_field_col(obj, field_name)
        if field_name == 'start_date':
            # Add the start time as data to the 'start_date' cell
            attrs.update({ 'data-time': obj.start_time.strftime('%H:%M') })
        elif field_name == 'end_date':
            # Add the end time as data to the 'end_date' cell
            attrs.update({ 'data-time': obj.end_time.strftime('%H:%M') })
        return attrs

```

wagtail.contrib.modeladmin.mixins.ThumbnailMixin

If you're using `wagtailimages.Image` to define an image for each item in your model, `ThumbnailMixin` can help you add thumbnail versions of that image to each row in `IndexView`. To use it, simply extend `ThumbnailMixin` as well as `ModelAdmin` when defining your `ModelAdmin` class, and change a few attributes to change the thumbnail to your liking, like so:

```

from django.db import models
from wagtail.contrib.modeladmin.mixins import ThumbnailMixin
from wagtail.contrib.modeladmin.options import ModelAdmin

class Person(models.Model):
    name = models.CharField(max_length=255)
    avatar = models.ForeignKey('wagtailimages.Image', on_delete=models.SET_NULL, null=True)
    likes_cat_gifs = models.NullBooleanField()

    class PersonAdmin(ThumbnailMixin, ModelAdmin):

        # Add 'admin_thumb' to list_display, where you want the thumbnail to appear
        list_display = ('admin_thumb', 'name', 'likes_cat_gifs')

        # Optionally tell IndexView to add buttons to a different column (if the
        # first column contains the thumbnail, the buttons are likely better off
        # displayed elsewhere)
        list_display_add_buttons = 'name'

        """
        Set 'thumb_image_field_name' to the name of the ForeignKey field that
        links to 'wagtailimages.Image'
        """
        thumb_image_field_name = 'avatar'

        # Optionally override the filter spec used to create each thumb
        thumb_image_filter_spec = 'fill-100x100' # this is the default

```

(continues on next page)

(continued from previous page)

```
# Optionally override the 'width' attribute value added to each `<img>` tag
thumb_image_width = 50 # this is the default

# Optionally override the class name added to each `<img>` tag
thumb_classname = 'admin-thumb' # this is the default

# Optionally override the text that appears in the column header
thumb_col_header_text = 'image' # this is the default

# Optionally specify a fallback image to be used when the object doesn't
# have an image set, or the image has been deleted. It can be an image from
# your static files folder, or an external URL.
thumb_default = 'https://lorempixel.com/100/100'
```

`ModelAdmin.list_display_add_buttons`

Expected value: A string matching one of the items in `list_display`.

If for any reason you'd like to change which column the action buttons appear in for each row, you can specify a different column using `list_display_add_buttons` on your `ModelAdmin` class. The value must match one of the items your class's `list_display` attribute. By default, buttons are added to the first column of each row.

See the `ThumbnailMixin` example above to see how `list_display_add_buttons` can be used.

`ModelAdmin.index_view_extra_css`

Expected value: A list of path names of additional stylesheets to be added to the `IndexView`

See the following part of the docs to find out more: [Adding additional stylesheets and/or JavaScript](#)

`ModelAdmin.index_view_extra_js`

Expected value: A list of path names of additional js files to be added to the `IndexView`

See the following part of the docs to find out more: [Adding additional stylesheets and/or JavaScript](#)

`ModelAdmin.index_template_name`

Expected value: The path to a custom template to use for `IndexView`

See the following part of the docs to find out more: [Overriding templates](#)

ModelAdmin.index_view_class

Expected value: A custom view class to replace `modeladmin.views.IndexView`

See the following part of the docs to find out more: [Overriding views](#)

Customising CreateView, EditView and DeleteView

Note: **NOTE:** `modeladmin` only provides ‘create’, ‘edit’ and ‘delete’ functionality for non page type models (models that do not extend `wagtailcore.models.Page`). If your model is a ‘page type’ model, customising any of the following will not have any effect:

Changing which fields appear in CreateView & EditView

`edit_handler` can be used on any Django `models.Model` class, just like it can be used for Page models or other models registered as Snippets in Wagtail.

To change the way your `MyPageModel` is displayed in the `CreateView` and the `EditView`, simply define an `edit_handler` or `panels` attribute on your model class.

```
class MyPageModel(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    address = models.TextField()

    panels = [
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('first_name', classname='fn'),
                FieldPanel('last_name', classname='ln'),
            ]),
            FieldPanel('address', classname='custom1'),
        ])
    ]
```

Or alternatively:

```
class MyPageModel(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    address = models.TextField()

    custom_panels = [
        MultiFieldPanel([
            FieldRowPanel([
                FieldPanel('first_name', classname='fn'),
                FieldPanel('last_name', classname='ln'),
            ]),
            FieldPanel('address', classname='custom1'),
        ])
    ]
```

(continues on next page)

(continued from previous page)

```
]
edit_handler = ObjectList(custom_panels)
# or
edit_handler = TabbedInterface([
    ObjectList(custom_panels, heading='First Tab'),
    ObjectList(...)
])
```

`edit_handler` and `panels` can alternatively be defined on a `ModelAdmin` definition. This feature is especially useful for use cases where you have to work with models that are ‘out of reach’ (due to being part of a third-party package, for example).

```
class BookAdmin(ModelAdmin):
    model = Book

    panels = [
        FieldPanel('title'),
        FieldPanel('author'),
    ]
```

Or alternatively:

```
class BookAdmin(ModelAdmin):
    model = Book

    custom_panels = [
        FieldPanel('title'),
        FieldPanel('author'),
    ]
    edit_handler = ObjectList(custom_panels)
```

`ModelAdmin.form_view_extra_css`

Expected value: A list of path names of additional stylesheets to be added to `CreateView` and `EditView`

See the following part of the docs to find out more: [Adding additional stylesheets and/or JavaScript](#)

ModelAdmin.form_view_extra_js

Expected value: A list of path names of additional js files to be added to CreateView and EditView

See the following part of the docs to find out more: [Adding additional stylesheets and/or JavaScript](#)

ModelAdmin.create_template_name

Expected value: The path to a custom template to use for CreateView

See the following part of the docs to find out more: [Overriding templates](#)

ModelAdmin.create_view_class

Expected value: A custom view class to replace modeladmin.views.CreateView

See the following part of the docs to find out more: [Overriding views](#)

ModelAdmin.edit_template_name

Expected value: The path to a custom template to use for EditView

See the following part of the docs to find out more: [Overriding templates](#)

ModelAdmin.edit_view_class

Expected value: A custom view class to replace modeladmin.views.EditView

See the following part of the docs to find out more: [Overriding views](#)

ModelAdmin.delete_template_name

Expected value: The path to a custom template to use for DeleteView

See the following part of the docs to find out more: [Overriding templates](#)

ModelAdmin.delete_view_class

Expected value: A custom view class to replace modeladmin.views.DeleteView

See the following part of the docs to find out more: [Overriding views](#)

`ModelAdmin.form_fields_exclude`

Expected value: A list or tuple of fields names

When using `CreateView` or `EditView` to create or update model instances, this value will be passed to the edit form, so that any named fields will be excluded from the form. This is particularly useful when registering `ModelAdmin` classes for models from third-party apps, where defining panel configurations on the Model itself is more complicated.

`ModelAdmin.prepopulated_fields`

Expected value: A dict mapping prepopulated fields to a tuple of fields to prepopulate from

When using `CreateView` or `EditView` to create or update model instances, the fields corresponding to the keys in the dict are prepopulated using the fields in the corresponding tuple. The main use for this functionality is to automatically generate the value for `SlugField` fields from one or more other fields. The generated value is produced by concatenating the values of the source fields, and then by transforming that result into a valid slug (for example substituting dashes for spaces; lowercasing ASCII letters; and removing various English stop words such as ‘a’, ‘an’, ‘as’, and similar).

Prepopulated fields aren’t modified by JavaScript after a value has been saved. It’s usually undesired that slugs change (which would cause an object’s URL to change if the slug is used in it).

`prepopulated_fields` doesn’t accept `DateTimeField`, `ForeignKey`, `OneToOneField`, and `ManyToManyField` fields.

`ModelAdmin.get_edit_handler()`

Must return: An instance of `wagtail.admin.panels.ObjectList`

Returns the appropriate `edit_handler` for the `modeladmin` class. `edit_handlers` can be defined either on the model itself or on the `modeladmin` (as property `edit_handler` or `panels`). Falls back to extracting panel / edit handler definitions from the model class.

Enabling & customising `InspectView`

The `InspectView` is disabled by default, as it’s not often useful for most models. However, if you need a view that enables users to view more detailed information about an instance without the option to edit it, you can easily enable the inspect view by setting `inspect_view_enabled=True` on your `ModelAdmin` class.

When `InspectView` is enabled, an ‘Inspect’ button will automatically appear for each row in your index / listing view, linking to a new page that shows a list of field values for that particular object.

By default, all ‘concrete’ fields (where the field value is stored as a column in the database table for your model) will be shown. You can customise what values are displayed by adding the following attributes to your `ModelAdmin` class:

- `ModelAdmin.inspect_view_fields`
- `ModelAdmin.inspect_view_fields_exclude`
- `ModelAdmin.inspect_view_extra_css`
- `ModelAdmin.inspect_view_extra_js`
- `ModelAdmin.inspect_template_name`
- `ModelAdmin.inspect_view_class`

`ModelAdmin.inspect_view_fields`

Expected value: A list or tuple, where each item is the name of a field or attribute on the instance that you'd like `InspectView` to render.

A sensible value will be rendered for most field types.

If you have `wagtail.images` installed, and the value happens to be an instance of `wagtailimages.models.Image` (or a custom model that subclasses `wagtailimages.models.AbstractImage`), a thumbnail of that image will be rendered.

If you have `wagtail.documents` installed, and the value happens to be an instance of `wagtaildocs.models.Document` (or a custom model that subclasses `wagtaildocs.models.AbstractDocument`), a link to that document will be rendered, along with the document title, file extension and size.

`ModelAdmin.inspect_view_fields_exclude`

Expected value: A list or tuple, where each item is the name of a field that you'd like to exclude from `InspectView`

Note: If both `inspect_view_fields` and `inspect_view_fields_exclude` are set, `inspect_view_fields_exclude` will be ignored.

`ModelAdmin.inspect_view_extra_css`

Expected value: A list of path names of additional stylesheets to be added to the `InspectView`

See the following part of the docs to find out more: [Adding additional stylesheets and/or JavaScript](#)

`ModelAdmin.inspect_view_extra_js`

Expected value: A list of path names of additional js files to be added to the `InspectView`

See the following part of the docs to find out more: [Adding additional stylesheets and/or JavaScript](#)

`ModelAdmin.inspect_template_name`

Expected value: The path to a custom template to use for `InspectView`

See the following part of the docs to find out more: [Overriding templates](#)

`ModelAdmin.inspect_view_class`

Expected value: A custom view class to replace `modeladmin.views.InspectView`

See the following part of the docs to find out more: [Overriding views](#)

Customising ChooseParentView

When adding a new page via Wagtail's explorer view, you essentially choose where you want to add a new page by navigating the relevant part of the page tree and choosing to 'add a child page' to your chosen parent page. Wagtail then asks you to select what type of page you'd like to add.

When adding a page from a `ModelAdmin` list page, we know what type of page needs to be added, but we might not automatically know where in the page tree it should be added. If there's only one possible choice of parent for a new page (as defined by setting `parent_page_types` and `subpage_types` attributes on your models), then we skip a step and use that as the parent. Otherwise, the user must specify a parent page using `modeladmin`'s `ChooseParentView`.

It should be very rare that you need to customise this view, but in case you do, `modeladmin` offers the following attributes that you can override:

- `ModelAdmin.choose_parent_template_name`
- `ModelAdmin.choose_parent_view_class`

`ModelAdmin.choose_parent_template_name`

Expected value: The path to a custom template to use for `ChooseParentView`

See the following part of the docs to find out more: [Overriding templates](#)

`ModelAdmin.choose_parent_view_class`

Expected value: A custom view class to replace `modeladmin.views.ChooseParentView`

See the following part of the docs to find out more: [Overriding views](#)

Additional tips and tricks

This section explores some of `modeladmin`'s lesser-known features, and provides examples to help with `modeladmin` customisation. More pages will be added in future.

Adding a custom clean method to your `ModelAdmin` models

The simplest way is to extend your `ModelAdmin` model and add a `clean()` method to it. For example:

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

class ModelAdminModel(models.Model):
    def clean(self):
        if self.image.width < 1920 or self.image.height < 1080:
            raise forms.ValidationError("The image must be at least 1920x1080 pixels in size.")
```

This will run the `clean` and raise the `ValidationError` whenever you save the model and the check fails. The error will be displayed at the top of the wagtail admin.

If you want more fine grained-control you can add a custom `clean()` method to the `WagtailAdminPageForm` of your model. You can override the form of your `ModelAdmin` in a similar matter as wagtail Pages.

So, create a custom `WagtailAdminPageForm`:

```
from wagtail.admin.forms import WagtailAdminPageForm

class ModelAdminModelForm(WagtailAdminPageForm):
    def clean(self):
        cleaned_data = super().clean()
        image = cleaned_data.get("image")
        if image and image.width < 1920 or image.height < 1080:
            self.add_error("image", "The image must be at least 1920x1080px")

    return cleaned_data
```

And then set the `base_form_class` of your model:

```
from django.db import models

class ModelAdminModel(models.Model):
    base_form_class = ModelAdminModelForm
```

Using `self.add_error` will display the error to the particular field that has the error.

Reversing ModelAdmin URLs

It's sometimes useful to be able to derive the `index` (listing) or `create` URLs for a model along with the `edit`, `delete` or `inspect` URL for a specific object in a model you have registered via the `modeladmin` app.

Wagtail itself does this by instantiating each `ModelAdmin` class you have registered, and using the `url_helper` attribute of each instance to determine what these URLs are.

You can take a similar approach in your own code too, by creating a `ModelAdmin` instance yourself, and using its `url_helper` to determine URLs.

See below for some examples:

- *Getting the edit or delete or inspect URL for an object*
- *Getting the index or create URL for a model*

Getting the edit or delete or inspect URL for an object

In this example, we will provide a quick way to edit the Author that is linked to a blog post from the admin page listing menu. We have defined an `AuthorModelAdmin` class and registered it with Wagtail to allow `Author` objects to be administered via the admin area. The `BlogPage` model has an `author` field (a `ForeignKey` to the `Author` model) to allow a single author to be specified for each post.

```
# file: wagtail_hooks.py

from wagtail.admin.widgets import PageListingButton
```

(continues on next page)

(continued from previous page)

```

from wagtail.contrib.modeladmin.options import ModelAdmin, modeladmin_register
from wagtail import hooks

# Author & BlogPage model not shown in this example
from models import Author

# ensure our modeladmin is created
class AuthorModelAdmin(ModelAdmin):
    model = Author
    menu_order = 200

    # Creating an instance of `AuthorModelAdmin`
author_modeladmin = AuthorModelAdmin()

@hooks.register('register_page_listing_buttons')
def add_author_edit_buttons(page, page_perms, next_url=None):
    """
    For pages that have an author, add an additional button to the page listing,
    linking to the 'edit' page for that author.
    """
    author_id = getattr(page, 'author_id', None)
    if author_id:
        # the url helper will return something like: /admin/my-app/author/edit/2/
        author_edit_url = author_modeladmin.url_helper.get_action_url('edit', author_
-id)
        yield PageListingButton('Edit Author', author_edit_url, priority=10)

modeladmin_register(AuthorModelAdmin)

```

As you can see from the example above, when using `get_action_url()` to generate object-specific URLs, the target object's primary key value must be supplied so that it can be included in the resulting URL (for example `"/admin/my-app/author/edit/2/"`). The following object-specific action names are supported by `get_action_url()`:

- 'edit' Returns a URL for updating a specific object.
 - 'delete' Returns a URL for deleting a specific object.
 - 'inspect' Returns a URL for viewing details of a specific object.
- **NOTE:** This will only work if `inspect_view_enabled` is set to `True` on your `ModelAdmin` class.

Note: If you are using string values as primary keys for your model, you may need to handle cases where the key contains characters that are not URL safe. Only alphanumerics ([0-9a-zA-Z]), or the following special characters are safe: \$, -, _, ., +, !, *, ', (,).

`django.contrib.admin.utils.quote()` can be used to safely encode these primary key values before passing them to `get_action_url()`. Failure to do this may result in Wagtail not being able to recognise the primary key when the URL is visited, resulting in 404 errors.

Getting the index or create URL for a model

There are URLs available for the model listing view (action is 'index') and the create model view (action is 'create'). Each of these has an equivalent shortcut available; `url_helper.index_url` and `url_helper.create_url`.

For example:

```
from .wagtail_hooks import AuthorModelAdmin

url_helper = AuthorModelAdmin().url_helper

index_url = url_helper.get_action_url('index')
# OR we can use the 'index_url' shortcut
also_index_url = url_helper.index_url # note: do not call this property as a function
# both will output /admin/my-app/author

create_url = url_helper.get_action_url('create')
# OR we can use the 'create_url' shortcut
also_create_url = url_helper.create_url # note: do not call this property as a function
# both will output /admin/my-app/author/create
```

Note: If you have registered a page type with `modeladmin` (for example `BlogPage`), and pages of that type can be added to more than one place in the page tree, when a user visits the `create` URL, they'll be automatically redirected to another view to choose a parent for the new page. So, this isn't something you need to check or cater for in your own code.

To customise `url_helper` behaviour, see [ModelAdmin.url_helper_class](#).

Installation

Add `wagtail.contrib.modeladmin` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'wagtail.contrib.modeladmin',
]
```

How to use

A simple example

Let's say your website is for a local library. They have a model called `Book` that appears across the site in many places. You can define a normal Django model for it, then use `ModelAdmin` to create a menu in Wagtail's admin to create, view, and edit `Book` entries.

`models.py` looks like this:

```
from django.db import models
from wagtail.admin.panels import FieldPanel

class Book(models.Model):
    title = models.CharField(max_length=255)
    author = models.CharField(max_length=255)
    cover_photo = models.ForeignKey(
        'wagtailimages.Image',
        null=True, blank=True,
        on_delete=models.SET_NULL,
        related_name='+'
    )

    panels = [
        FieldPanel('title'),
        FieldPanel('author'),
        FieldPanel('cover_photo')
    ]
```

Note: You can specify panels like `MultiFieldPanel` within the `panels` attribute of the model. This lets you use Wagtail-specific layouts in an otherwise traditional Django model.

`wagtail_hooks.py` in your app directory would look something like this:

```
from wagtail.contrib.modeladmin.options import (
    ModelAdmin, modeladmin_register)
from .models import Book

class BookAdmin(ModelAdmin):
    model = Book
    base_url_path = 'bookadmin' # customise the URL from default to admin/bookadmin
    menu_label = 'Book' # ditch this to use verbose_name_plural from model
    menu_icon = 'pilcrow' # change as required
    menu_order = 200 # will put in 3rd place (000 being 1st, 100 2nd)
    add_to_settings_menu = False # or True to add your model to the Settings sub-
    ↵menu
    exclude_from_explorer = False # or True to exclude pages of this type from
    ↵Wagtail's explorer view
    add_to_admin_menu = True # or False to exclude your model from the menu
    list_display = ('title', 'author')
    list_filter = ('author',)
    search_fields = ('title', 'author')

# Now you just need to register your customised ModelAdmin class with Wagtail
modeladmin_register(BookAdmin)
```

A more complicated example

In addition to `Book`, perhaps we also want to add `Author` and `Genre` models to our app and display a menu item for each of them, too. Creating lots of menus can add up quickly, so it might be a good idea to group related menus together. This section show you how to create one menu called `Library` which expands to show submenus for `Book`, `Author`, and `Genre`.

Assume we've defined `Book`, `Author`, and `Genre` models in `models.py`.

`wagtail_hooks.py` in your app directory would look something like this:

```
from wagtail.contrib.modeladmin.options import (
    ModelAdmin, ModelAdminGroup, modeladmin_register)
from .models import (
    Book, Author, Genre)

class BookAdmin(ModelAdmin):
    model = Book
    menu_label = 'Book' # ditch this to use verbose_name_plural from model
    menu_icon = 'pilcrow' # change as required
    list_display = ('title', 'author')
    list_filter = ('genre', 'author')
    search_fields = ('title', 'author')

class AuthorAdmin(ModelAdmin):
    model = Author
    menu_label = 'Author' # ditch this to use verbose_name_plural from model
    menu_icon = 'user' # change as required
    list_display = ('first_name', 'last_name')
    list_filter = ('first_name', 'last_name')
    search_fields = ('first_name', 'last_name')

class GenreAdmin(ModelAdmin):
    model = Genre
    menu_label = 'Genre' # ditch this to use verbose_name_plural from model
    menu_icon = 'group' # change as required
    list_display = ('name',)
    list_filter = ('name',)
    search_fields = ('name',)

class LibraryGroup(ModelAdminGroup):
    menu_label = 'Library'
    menu_icon = 'folder-open-inverse' # change as required
    menu_order = 200 # will put in 3rd place (000 being 1st, 100 2nd)
    items = (BookAdmin, AuthorAdmin, GenreAdmin)

# When using a ModelAdminGroup class to group several ModelAdmin classes together,
# you only need to register the ModelAdminGroup class with Wagtail:
modeladmin_register(LibraryGroup)
```

Registering multiple classes in one wagtail_hooks.py file

Each time you call `modeladmin_register(MyAdmin)` it creates a new top-level menu item in Wagtail's left sidebar. You can call this multiple times within the same `wagtail_hooks.py` file if you want. The example below will create 3 top-level menus.

```
class BookAdmin(ModelAdmin):
    model = Book
    ...

class MovieAdmin(ModelAdmin):
    model = MovieModel
    ...

class MusicAdminGroup(ModelAdminGroup):
    menu_label = _("Music")
    items = (AlbumAdmin, ArtistAdmin)
    ...

modeladmin_register(BookAdmin)
modeladmin_register(MovieAdmin)
modeladmin_register(MusicAdminGroup)
```

Promoted search results

The `searchpromotions` module provides the models and user interface for managing “Promoted search results” and displaying them in a search results page.

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

Installation

The `searchpromotions` module is not enabled by default. To install it, add `wagtail.contrib.search_promotions` to `INSTALLED_APPS` in your project's Django settings file.

```
INSTALLED_APPS = [
    ...
    'wagtail.contrib.search_promotions',
]
```

This app contains migrations so make sure you run the `migrate` django-admin command after installing.

Usage

Once installed, a new menu item called “Promoted search results” should appear in the “Settings” menu. This is where you can assign pages to popular search terms.

Displaying on a search results page

To retrieve a list of promoted search results for a particular search query, you can use the `{% get_search_promotions %}` template tag from the `wagtailsearchpromotions_tags` templatetag library:

```
{% load wagtailcore_tags wagtailsearchpromotions_tags %}

...
{% get_search_promotions search_query as search_promotions %}

<ul>
    {% for search_promotion in search_promotions %}
        <li>
            <a href="{% pageurl search_promotion.page %}">
                <h2>{{ search_promotion.page.title }}</h2>
                <p>{{ search_promotion.description }}</p>
            </a>
        </li>
    {% endfor %}
</ul>
```

Simple translation

The `simple_translation` module provides a user interface that allows users to copy pages and translatable snippets into another language.

- Copies are created in the source language (not translated)
- Copies of pages are in draft status

Content editors need to translate the content and publish the pages.

Note: Simple Translation is optional. It can be switched out by third-party packages. Like the more advanced `wagtail-localize`.

Basic configuration

Add `"wagtail.contrib.simple_translation"` to `INSTALLED_APPS` in your settings file:

```
INSTALLED_APPS = [
    ...
    "wagtail.contrib.simple_translation",
]
```

Run `python manage.py migrate` to create the necessary permissions.

In the Wagtail admin, go to settings and give some users or groups the “Can submit translations” permission.

Page tree synchronisation

Depending on your use case, it may be useful to keep the page trees in sync between different locales.

You can enable this feature by setting `WAGTAILSIMPLETRANSLATION_SYNC_PAGE_TREE` to True.

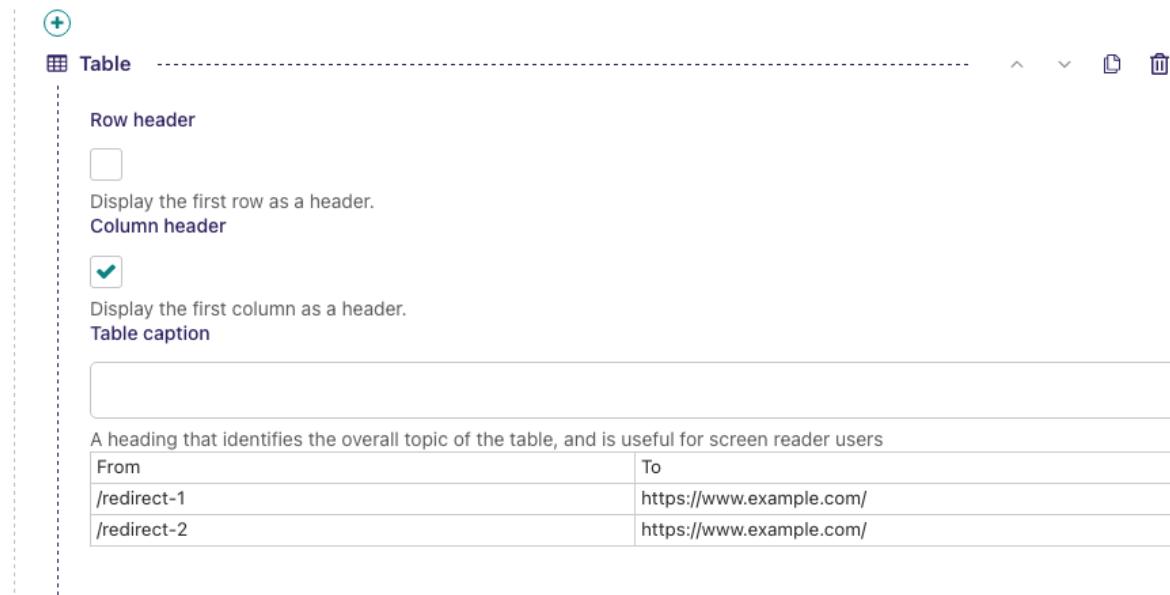
```
WAGTAILSIMPLETRANSLATION_SYNC_PAGE_TREE = True
```

When this feature is turned on, every time an editor creates a page, Wagtail creates an alias for that page under the page trees of all the other locales.

For example, when an editor creates the page `"/en/blog/my-blog-post/"`, Wagtail creates an alias of that page at `"/fr/blog/my-blog-post/"` and `"/de/blog/my-blog-post/"`.

TableBlock

The TableBlock module provides an HTML table block type for StreamField. This module uses [handsontable 6.2.2](#) to provide users with the ability to create and edit HTML tables in Wagtail. Table blocks provides a caption field for accessibility.



Installation

Add `"wagtail.contrib.table_block"` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    "wagtail.contrib.table_block",
]
```

Basic Usage

After installation, the `TableBlock` module can be used in a similar fashion to other StreamField blocks in the Wagtail core.

Import the `TableBlock` from `wagtail.contrib.table_block.blocks` import `TableBlock` and add it to your StreamField declaration.

```
class DemoStreamBlock(StreamBlock):
    ...
    table = TableBlock()
```

Then, on your page template, the `{% include_block %}` tag (called on either the individual block, or the StreamField value as a whole) will render any table blocks it encounters as an HTML `<table>` element:

```
{% load wagtailcore_tags %}

{% include_block page.body %}
```

Or:

```
{% load wagtailcore_tags %}

{% for block in page.body %}
    {% if block.block_type == 'table' %}
        {% include_block block %}
    {% else %}
        {%# rendering for other block types %}
        {% endif %}
    {% endfor %}
```

Advanced Usage

Default Configuration

When defining a `TableBlock`, Wagtail provides the ability to pass an optional `table_options` dictionary. The default `TableBlock` dictionary looks like this:

```
default_table_options = {
    'minSpareRows': 0,
    'startRows': 3,
    'startCols': 3,
    'colHeaders': False,
    'rowHeaders': False,
    'contextMenu': [
        'row_above',
        'row_below',
        '-----',
        'col_left',
        'col_right',
        '-----',
        'remove_row',
```

(continues on next page)

(continued from previous page)

```
'remove_col',
'-----',
'undo',
'redo'
],
'editor': 'text',
'stretchH': 'all',
'height': 108,
'language': language,
'renderer': 'text',
'autoColumnSize': False,
}
```

Configuration Options

Every key in the `table_options` dictionary maps to a [handsontable](#) option. These settings can be changed to alter the behaviour of tables in Wagtail. The following options are available:

- `minSpareRows` - The number of rows to append to the end of an empty grid. The default setting is 0.
- `startRows` - The default number of rows for a new table.
- `startCols` - The default number of columns for new tables.
- `colHeaders` - Can be set to `True` or `False`. This setting designates if new tables should be created with column headers. **Note:** this only sets the behaviour for newly created tables. Page editors can override this by checking the the “Column header” checkbox in the table editor in the Wagtail admin.
- `rowHeaders` - Operates the same as `colHeaders` to designate if new tables should be created with the first column as a row header. Just like `colHeaders` this option can be overridden by the page editor in the Wagtail admin.
- `contextMenu` - Enables or disables the Handsontable right-click menu. By default this is set to `True`. Alternatively you can provide a list or a dictionary with [specific options](#).
- `editor` - Defines the editor used for table cells. The default setting is `text`.
- `stretchH` - Sets the default horizontal resizing of tables. Options include, ‘none’, ‘last’, and ‘all’. By default `TableBlock` uses ‘all’ for the even resizing of columns.
- `height` - The default height of the grid. By default `TableBlock` sets the height to 108 for the optimal appearance of new tables in the editor. This is optimized for tables with `startRows` set to 3. If you change the number of `startRows` in the configuration, you might need to change the `height` setting to improve the default appearance in the editor.
- `language` - The default language setting. By default `TableBlock` tries to get the language from `django.utils.translation.get_language`. If needed, this setting can be overridden here.
- `renderer` - The default setting Handsontable uses to render the content of table cells.
- `autoColumnSize` - Enables or disables the `autoColumnSize` plugin. The `TableBlock` default setting is `False`.

A complete list of [handsontable options](#) can be found on the Handsontable website.

Changing the default table_options

To change the default table options just pass a new table_options dictionary when a new TableBlock is declared.

```
new_table_options = {
    'minSpareRows': 0,
    'startRows': 6,
    'startCols': 4,
    'colHeaders': False,
    'rowHeaders': False,
    'contextMenu': True,
    'editor': 'text',
    'stretchH': 'all',
    'height': 216,
    'language': 'en',
    'renderer': 'text',
    'autoColumnSize': False,
}

class DemoStreamBlock(StreamBlock):
    ...
    table = TableBlock(table_options=new_table_options)
```

Supporting cell alignment

You can activate the alignment option by setting a custom contextMenu which allows you to set the alignment on a cell selection. HTML classes set by handsontable will be kept on the rendered block. You'll then be able to apply your own custom CSS rules to preserve the style. Those class names are:

- Horizontal: htLeft, htCenter, htRight, htJustify
- Vertical: htTop, htMiddle, htBottom

```
new_table_options = {
    'contextMenu': [
        'row_above',
        'row_below',
        '-----',
        'col_left',
        'col_right',
        '-----',
        'remove_row',
        'remove_col',
        '-----',
        'undo',
        'redo',
        '-----',
        'copy',
        'cut',
        '-----',
        'alignment',
    ],
}
```

(continues on next page)

(continued from previous page)

```
class DemoStreamBlock(StreamBlock):
    ...
    table = TableBlock(table_options=new_table_options)
```

Typed table block

The `typed_table_block` module provides a StreamField block type for building tables consisting of mixed data types. Developers can specify a set of block types (such as `RichTextBlock` or `FloatBlock`) to be available as column types; page authors can then build up tables of any size by choosing column types from that list, in much the same way that they would insert blocks into a StreamField. Within each column, authors enter data using the standard editing control for that field (such as the Draftail editor for rich text cells).

Installation

Add "wagtail.contrib.typed_table_block" to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    "wagtail.contrib.typed_table_block",
]
```

Usage

`TypedTableBlock` can be imported from the module `wagtail.contrib.typed_table_block.blocks` and used within a StreamField definition. Just like `StructBlock` and `StreamBlock`, it accepts a list of `(name, block_type)` tuples to use as child blocks:

```
from wagtail.contrib.typed_table_block.blocks import TypedTableBlock
from wagtail import blocks
from wagtail.images.blocks import ImageChooserBlock

class DemoStreamBlock(blocks.StreamBlock):
    title = blocks.CharBlock()
    paragraph = blocks.RichTextBlock()
    table = TypedTableBlock([
        ('text', blocks.CharBlock()),
        ('numeric', blocks.FloatBlock()),
        ('rich_text', blocks.RichTextBlock()),
        ('image', ImageChooserBlock())
    ])
```

To keep the UI as simple as possible for authors, it's generally recommended to use Wagtail's basic built-in block types as column types, as above. However, all custom block types and parameters are supported. For example, to define a 'country' column type consisting of a dropdown of country choices:

```
table = TypedTableBlock([
    ('text', blocks.CharBlock()),
    ('numeric', blocks.FloatBlock()),
```

(continues on next page)

(continued from previous page)

```
('rich_text', blocks.RichTextBlock()),
('image', ImageChooserBlock()),
('country', ChoiceBlock(choices=[
    ('be', 'Belgium'),
    ('fr', 'France'),
    ('de', 'Germany'),
    ('nl', 'Netherlands'),
    ('pl', 'Poland'),
    ('uk', 'United Kingdom'),
]),
])
```

On your page template, the `{% include_block %}` tag (called on either the individual block, or the StreamField value as a whole) will render any typed table blocks as an HTML `<table>` element.

```
{% load wagtailcore_tags %}

{% include_block page.body %}
```

Or:

```
{% load wagtailcore_tags %}

{% for block in page.body %}
    {% if block.block_type == 'table' %}
        {% include_block block %}
    {% else %}
        {%# rendering for other block types %}
        {% endif %}
    {% endfor %}
```

Redirects

The `redirects` module provides the models and user interface for managing arbitrary redirection between urls and Pages or other urls.

Installation

The `redirects` module is not enabled by default. To install it, add `wagtail.contrib.redirects` to `INSTALLED_APPS` and `wagtail.contrib.redirects.middleware.RedirectMiddleware` to `MIDDLEWARE` in your project's Django settings file.

```
INSTALLED_APPS = [
    # ...

    'wagtail.contrib.redirects',
]

MIDDLEWARE = [
    # ...
]
```

(continues on next page)

(continued from previous page)

```
# all other django middleware first  
  
'wagtail.contrib.redirects.middleware.RedirectMiddleware',  
]
```

This app contains migrations so make sure you run the `migrate` django-admin command after installing.

Usage

Once installed, a new menu item called “Redirects” should appear in the “Settings” menu. This is where you can add arbitrary redirects to your site.

For an editor’s guide to the interface, see [Managing Redirects](#).

Automatic redirect creation

New in version 2.16.

Wagtail automatically creates permanent redirects for pages (and their descendants) when they are moved or their slug is changed. This helps to preserve SEO rankings of pages over time, and helps site visitors get to the right place when using bookmarks or using outdated links.

Creating redirects for alternative page routes

If your project uses `RoutablePageMixin` to create pages with alternative routes, you might want to consider overriding the `get_route_paths()` method for those page types. Adding popular route paths to this list will result in the creation of additional redirects; helping visitors to alternative routes to get to the right place also.

For more information, please see :meth:`~wagtail.models.Page.get_route_paths`.

Disabling automatic redirect creation

New in version 4.0: When generating redirects, custom field values are now fetched as part of the initial database query, so using custom field values in overridden url methods will no longer trigger additional per-object queries.

Wagtail’s default implementation works best for small-to-medium sized projects (5000 pages or fewer) that mostly use Wagtail’s built-in methods for URL generation.

Overrides to the following `Page` methods are respected when generating redirects, but use of specific page fields in those overrides will trigger additional database queries.

- `get_url_parts()`
- `get_route_paths()`

If you find the feature is not a good fit for your project, you can disable it by adding the following to your project settings:

```
WAGTAILREDIRECTS_AUTO_CREATE = False
```

Management commands

`import_redirects`

```
$ ./manage.py import_redirects
```

This command imports and creates redirects from a file supplied by the user.

Options:

Option	Description
src	This is the path to the file you wish to import redirects from.
site	This is the site for the site you wish to save redirects to.
permanent	If the redirects imported should be permanent (True) or not (False). It's True by default.
from	The column index you want to use as redirect from value.
to	The column index you want to use as redirect to value.
dry_run	Lets you run a import without doing any changes.
ask	Lets you inspect and approve each redirect before it is created.

The Redirect class

```
class wagtail.contrib.redirects.models.Redirect(id, old_path, site, is_permanent, redirect_page,
                                                redirect_page_route_path, redirect_link,
                                                automatically_created, created_at)

static add_redirect(old_path, redirect_to=None, is_permanent=True, page_route_path=None,
                    site=None, automatically_created=False)
```

Create and save a Redirect instance with a single method.

Parameters

- **old_path** – the path you wish to redirect
- **site** – the Site (instance) the redirect is applicable to (if not all sites)
- **redirect_to** – a Page (instance) or path (string) where the redirect should point
- **is_permanent** – whether the redirect should be indicated as permanent (i.e. 301 redirect)

Returns Redirect instance

Legacy richtext

Provides the legacy richtext wrapper.

Place `wagtail.contrib.legacy.richtext` before `wagtail` in `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    ...
    "wagtail.contrib.legacy.richtext",
    "wagtail",
    ...
]
```

The `{{ page.body|richtext }}` template filter will now render:

```
<div class="rich-text">...</div>
```

Settings

Settings that are editable by administrators within the Wagtail admin - either site-specific or generic across all sites.

Form builder

Allows forms to be created by admins and provides an interface for browsing form submissions.

Sitemap generator

Provides a view that generates a Google XML sitemap of your public Wagtail content.

Frontend cache invalidator

A module for automatically purging pages from a cache (Varnish, Squid, Cloudflare or Cloudfront) when their content is changed.

RoutablePageMixin

Provides a way of embedding Django URLconfs into pages.

ModelAdmin

A module allowing for more customisable representation and management of custom models in Wagtail's admin area.

Promoted search results

A module for managing "Promoted Search Results"

Simple translation

A module for copying translatable (pages and snippets) to another language.

TableBlock

Provides a TableBlock for adding HTML tables to pages.

Typed table block

Provides a StreamField block for authoring tables, where cells can be any block type including rich text.

Redirects

Provides a way to manage redirects.

Legacy richtext

Provides the legacy richtext wrapper (<div class="rich-text"></div>).

1.5.4 Management commands

publish_scheduled_pages

```
$ ./manage.py publish_scheduled_pages
```

This command publishes, updates or unpublishes pages that have had these actions scheduled by an editor. We recommend running 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.

purge_revisions

```
$ manage.py purge_revisions [--days=<number of days>]
```

This command deletes old page revisions which are not in moderation, live, approved to go live, or the latest revision for a page. If the days argument is supplied, only revisions older than the specified number of days will be deleted.

update_index

```
$ ./manage.py update_index [--backend <backend name>]
```

This command rebuilds the search index from scratch.

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

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
```

The `--chunk_size` option can be used to set the size of chunks that are indexed at a time. This defaults to 1000 but may need to be reduced for larger document sizes.

Indexing the schema only

You can prevent the `update_index` command from indexing any data by using the `--schema-only` option:

```
$ python manage.py update_index --schema-only
```

wagtail_update_index

An alias for the `update_index` command that can be used when another installed package (such as Haystack) provides a command named `update_index`. In this case, the other package's entry in `INSTALLED_APPS` should appear above `wagtail.search` so that its `update_index` command takes precedence over Wagtail's.

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 (or a number of days configurable through the `WAGTAILSEARCH_HITS_MAX_AGE` setting).

wagtail_update_image_renditions

```
$ ./manage.py wagtail_update_image_renditions
```

This command provides the ability to regenerate image renditions. This is useful if you have deployed to a server where the image renditions have not yet been generated or you have changed the underlying image rendition behaviour and need to ensure all renditions are created again.

This does not remove rendition images that are unused, this can be done by clearing the folder using `rm -rf` or similar, once this is done you can then use the management command to generate the renditions.

Options:

- **`--purge-only`** : This argument will purge all image renditions without regenerating them. They will be regenerated when next requested.

1.5.5 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 is saved or when the main menu is constructed.

Note: Hooks are typically used to customise the view-level behaviour of the Wagtail admin and front-end. For customisations that only deal with model-level behaviour - such as calling an external service when a page or document is added - it is often better to use [Django's signal mechanism](#) (see also: [Wagtail signals](#)), as these are not dependent on a user taking a particular path through the admin interface.

Registering functions with a Wagtail hook is done through the `@hooks.register` decorator:

```
from wagtail 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)
```

If you need your hooks to run in a particular order, you can pass the `order` parameter. If `order` is not specified then the hooks proceed in the order given by `INSTALLED_APPS`. Wagtail uses hooks internally, too, so you need to be aware of order when overriding built-in Wagtail functionality (such as removing default summary items):

```
@hooks.register('name_of_hook', order=1) # This will run after every hook in the
                                         ↵wagtail core
def my_hook_function(arg1, arg2...)
    # your code here

@hooks.register('name_of_hook', order=-1) # This will run before every hook in the
                                         ↵wagtail core
def my_other_hook_function(arg1, arg2...)
    # your code here
```

(continues on next page)

(continued from previous page)

```
@hooks.register('name_of_hook', order=2) # This will run after `my_hook_function`  
def yet_another_hook_function(arg1, arg2...)  
    # your code here
```

Unit testing hooks

Hooks are usually registered on startup and can't be changed at runtime. But when writing unit tests, you might want to register a hook function just for a single test or block of code and unregister it so that it doesn't run when other tests are run.

You can register hooks temporarily using the `hooks.register_temporarily` function, this can be used as both a decorator and a context manager. Here's an example of how to register a hook function for just a single test:

```
def my_hook_function():  
    pass  
  
class MyHookTest(TestCase):  
  
    @hooks.register_temporarily('name_of_hook', my_hook_function)  
    def test_my_hook_function(self):  
        # Test with the hook registered here  
        pass
```

And here's an example of registering a hook function for a single block of code:

```
def my_hook_function():  
    pass  
  
with hooks.register_temporarily('name_of_hook', my_hook_function):  
    # Hook is registered here  
    ...  
  
# Hook is unregistered here
```

If you need to register multiple hooks in a `with` block, you can pass the hooks in as a list of tuples:

```
def my_hook(...):  
    pass  
  
def my_other_hook(...):  
    pass  
  
with hooks.register_temporarily([  
    ('hook_name', my_hook),  
    ('hook_name', my_other_hook),  
]):  
    # All hooks are registered here  
    ...  
  
# All hooks are unregistered here
```

The available hooks are listed below.

Admin modules

Hooks for building new areas of the admin interface (alongside pages, images, documents and so on).

`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 panel objects, and should modify this list in-place as required. Panel objects are [Template components](#) with an additional `order` property, an integer that determines the panel's position in the final ordered list. The default panels use integers between 100 and 300.

```
from django.utils.safestring import mark_safe

from wagtail.admin.ui.components import Component
from wagtail import hooks

class WelcomePanel(Component):
    order = 50

    def render_html(self, parent_context):
        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):
    panels.append(WelcomePanel())
```

`construct_homepage_summary_items`

Add or remove items from the ‘site summary’ bar on the admin homepage (which shows the number of pages and other object that exist on the site). The callable passed into this hook should take a `request` object and a list of summary item objects, and should modify this list in-place as required. Summary item objects are instances of `wagtail.admin.site_summary.SummaryItem`, which extends the [Component class](#) with the following additional methods and properties:

`SummaryItem(request)`

Constructor; receives the request object its argument

`order`

An integer that specifies the item’s position in the sequence.

`is_shown()`

Returns a boolean indicating whether the summary item should be shown on this request.

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 not have their `is_shown` check applied.

```
from wagtail 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']
```

describe_collection_contents

Called when Wagtail needs to find out what objects exist in a collection, if any. Currently this happens on the confirmation before deleting a collection, to ensure that non-empty collections cannot be deleted. The callable passed to this hook will receive a `collection` object, and should return either `None` (to indicate no objects in this collection), or a dict containing the following keys:

- `count` - A numeric count of items in this collection
- `count_text` - A human-readable string describing the number of items in this collection, such as “3 documents”. (Sites with multi-language support should return a translatable string here, most likely using the `django.utils.translation.ngettext` function.)
- `url` (optional) - A URL to an index page that lists the objects being described.

register_account_settings_panel

Registers a new settings panel class to add to the “Account” view in the admin.

This hook can be added to a sub-class of `BaseSettingsPanel`. For example:

```
from wagtail.admin.views.account import BaseSettingsPanel
from wagtail import hooks

@hooks.register('register_account_settings_panel')
class CustomSettingsPanel(BaseSettingsPanel):
    name = 'custom'
    title = "My custom settings"
    order = 500
    form_class = CustomSettingsForm
```

Alternatively, it can also be added to a function. For example, this function is equivalent to the above:

```
from wagtail.admin.views.account import BaseSettingsPanel
from wagtail import hooks

def register_custom_settings_panel():
    class CustomSettingsPanel(BaseSettingsPanel):
        name = 'custom'
        title = "My custom settings"
```

(continues on next page)

(continued from previous page)

```
order = 500
form_class = CustomSettingsForm

@hooks.register('register_account_settings_panel')
def register_custom_settings_panel(request, user, profile):
    return CustomSettingsPanel(request, user, profile)
```

More details about the options that are available can be found at [Customising the user account settings form](#).

register_account_menu_item

Add an item to the “More actions” tab on the “Account” page within the Wagtail admin. The callable for this hook should return a dict with the keys `url`, `label` and `help_text`. For example:

```
from django.urls import reverse
from wagtail import hooks

@hooks.register('register_account_menu_item')
def register_account_delete_account(request):
    return {
        'url': reverse('delete-account'),
        'label': 'Delete account',
        'help_text': 'This permanently deletes your account.'
    }
```

register_admin_menu_item

Add an item to the Wagtail admin menu. The callable passed to this hook must return an instance of `wagtail.admin.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.
- `icon_name` - icon to display against the menu item; no defaults, optional, but should be set for top-level menu items so they can be identified when collapsed.
- `classnames` - additional classnames applied to the link
- `order` - an integer which determines the item’s position in the menu

For menu items that are only available to superusers, the subclass `wagtail.admin.menu.AdminOnlyMenuItem` can be used in place of `MenuItem`.

`MenuItem` can be further subclassed to customise its initialisation or conditionally show or hide the item for specific requests (for example, to apply permission checks); see the source code (`wagtail/admin/menu.py`) for details.

```
from django.urls import reverse

from wagtail import hooks
from wagtail.admin.menu import MenuItem

@hooks.register('register_admin_menu_item')
```

(continues on next page)

(continued from previous page)

```
def register_frank_menu_item():
    return MenuItem('Frank', reverse('frank'), icon_name='folder-inverse', order=10000)
```

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 HttpResponseRedirect
from django.urls import path

from wagtail import hooks

def admin_view(request):
    return HttpResponseRedirect(
        "I have approximate knowledge of many things!",
        content_type="text/plain")

@hooks.register('register_admin_urls')
def urlconf_time():
    return [
        path('how_did_you_almost_know_my_name/', admin_view, name='frank'),
    ]
```

register_group_permission_panel

Add a new panel to the Groups form in the ‘settings’ area. The callable passed to this hook must return a ModelForm / ModelFormSet-like class, with a constructor that accepts a group object as its `instance` keyword argument, and which implements the methods `save`, `is_valid`, and `as_admin_panel` (which returns the HTML to be included on the group edit page).

register_settings_menu_item

As `register_admin_menu_item`, but registers menu items into the ‘Settings’ sub-menu rather than the top-level menu.

construct_settings_menu

As `construct_main_menu`, but modifies the ‘Settings’ sub-menu rather than the top-level menu.

`register_reports_menu_item`

As `register_admin_menu_item`, but registers menu items into the ‘Reports’ sub-menu rather than the top-level menu.

`construct_reports_menu`

As `construct_main_menu`, but modifies the ‘Reports’ sub-menu rather than the top-level menu.

`register_admin_search_area`

Add an item to the Wagtail admin search “Other Searches”. Behaviour of this hook is similar to `register_admin_menu_item`. The callable passed to this hook must return an instance of `wagtail.admin.search.SearchArea`. New items can be constructed from the `SearchArea` class by passing the following parameters:

- `label` - text displayed in the “Other Searches” option box.
- `name` - an internal name used to identify the search option; defaults to the slugified form of the label.
- `url` - the URL of the target search page.
- `classnames` - arbitrary CSS classnames applied to the link
- `icon_name` - icon to display next to the label.
- `attrs` - additional HTML attributes to apply to the link.
- `order` - an integer which determines the item’s position in the list of options.

Setting the URL can be achieved using `reverse()` on the target search page. The GET parameter ‘`q`’ will be appended to the given URL.

A template tag, `search_other` is provided by the `wagtailadmin_tags` template module. This tag takes a single, optional parameter, `current`, which allows you to specify the name of the search option currently active. If the parameter is not given, the hook defaults to a reverse lookup of the page’s URL for comparison against the `url` parameter.

`SearchArea` can be subclassed to customise the HTML output, specify JavaScript files required by the option, or conditionally show or hide the item for specific requests (for example, to apply permission checks); see the source code (`wagtail/admin/search.py`) for details.

```
from django.urls import reverse
from wagtail import hooks
from wagtail.admin.search import SearchArea

@hooks.register('register_admin_search_area')
def register_frank_search_area():
    return SearchArea('Frank', reverse('frank'), icon_name='folder-inverse', order=10000)
```

register_permissions

Return a QuerySet of Permission objects to be shown in the Groups administration area.

```
from django.contrib.auth.models import Permission
from wagtail import hooks

@hooks.register('register_permissions')
def register_permissions():
    app = 'blog'
    model = 'extramodelset'

    return Permission.objects.filter(content_type__app_label=app, codename__in=[
        f"view_{model}", f"add_{model}", f"change_{model}", f"delete_{model}"
    ])
```

filter_form_submissions_for_user

Allows access to form submissions to be customised on a per-user, per-form basis.

This hook takes two parameters:

- The user attempting to access form submissions
- A QuerySet of form pages

The hook must return a QuerySet containing a subset of these form pages which the user is allowed to access the submissions for.

For example, to prevent non-superusers from accessing form submissions:

```
from wagtail import hooks

@hooks.register('filter_form_submissions_for_user')
def construct_forms_for_user(user, queryset):
    if not user.is_superuser:
        queryset = queryset.none()

    return queryset
```

Editor interface

Hooks for customising the editing interface for pages and snippets.

register_rich_text_features

Rich text fields in Wagtail work with a list of ‘feature’ identifiers that determine which editing controls are available in the editor, and which elements are allowed in the output; for example, a rich text field defined as `RichTextField(features=['h2', 'h3', 'bold', 'italic', 'link'])` would allow headings, bold / italic formatting and links, but not (for example) bullet lists or images. The `register_rich_text_features` hook allows new feature identifiers to be defined - see [Limiting features in a rich text field](#) for details.

insert_editor_css

Add additional CSS files or snippets to the page editor.

```
from django.templatetags.static import static
from django.utils.html import format_html

from wagtail import hooks

@hooks.register('insert_editor_css')
def editor_css():
    return format_html(
        '<link rel="stylesheet" href="{}">',
        static('demo/css/vendor/font-awesome/css/font-awesome.min.css')
    )
```

insert_global_admin_css

Add additional CSS files or snippets to all admin pages.

```
from django.utils.html import format_html
from django.templatetags.static import static

from wagtail import hooks

@hooks.register('insert_global_admin_css')
def global_admin_css():
    return format_html('<link rel="stylesheet" href="{}">', static('my/wagtail/theme.css'))
```

insert_editor_js

Add additional JavaScript files or code snippets to the page editor.

```
from django.utils.html import format_html_join
from django.utils.safestring import mark_safe
from django.templatetags.static import static

from wagtail import hooks

@hooks.register('insert_editor_js')
```

(continues on next page)

(continued from previous page)

```
def editor_js():
    js_files = [
        'js/fireworks.js', # https://fireworks.js.org
    ]
    js_includes = format_html_join('\n', '<script src="{0}"></script>',
        ((static(filename),) for filename in js_files))
    )
    return js_includes + mark_safe(
        """
        <script>
            window.addEventListener('DOMContentLoaded', (event) => {
                var container = document.createElement('div');
                container.style.cssText = 'position: fixed; width: 100%; height: 100%; z-
        ↪index: 100; top: 0; left: 0; pointer-events: none;';
                container.id = 'fireworks';
                document.getElementById('main').prepend(container);
                var options = { "acceleration": 1.2, "autoresize": true, "mouse": {
        ↪"click": true, "max": 3 } };
                var fireworks = new Fireworks(document.getElementById('fireworks'), ↪
        ↪options);
                fireworks.start();
            });
        </script>
        """
    )
)
```

insert_global_admin_js

Add additional JavaScript files or code snippets to all admin pages.

```
from django.utils.safestring import mark_safe

from wagtail import hooks

@hooks.register('insert_global_admin_js')
def global_admin_js():
    return mark_safe(
        '<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r74/three.js"><
        ↪script>',
    )
```

register_page_header_buttons

Add buttons to the secondary dropdown menu in the page header. This works similarly to the `register_page_listing_buttons` hook.

This example will add a simple button to the secondary dropdown menu:

```
from wagtail.admin import widgets as wagtailadmin_widgets

@hooks.register('register_page_header_buttons')
def page_header_buttons(page, page_perms, next_url=None):
    yield wagtailadmin_widgets.Button(
        'A dropdown button',
        '/goes/to/a/url/',
        priority=60
    )
```

The arguments passed to the hook are as follows:

- `page` - the page object to generate the button for
- `page_perms` - a `PagePermissionTester` object that can be queried to determine the current user's permissions on the given page
- `next_url` - the URL that the linked action should redirect back to on completion of the action, if the view supports it

The `priority` argument controls the order the buttons are displayed in the dropdown. Buttons are ordered from low to high priority, so a button with `priority=10` will be displayed before a button with `priority=60`.

Editor workflow

Hooks for customising the way users are directed through the process of creating page content.

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 HttpResponseRedirect

from wagtail import hooks

@hooks.register('after_create_page')
def do_after_page_create(request, page):
    return HttpResponseRedirect("Congrats on making content!", content_type="text/plain")
```

If you set attributes on a `Page` object, you should also call `save_revision()`, since the edit and index view pick up their data from the revisions table rather than the actual saved page record.

```
@hooks.register('after_create_page')
def set_attribute_after_page_create(request, page):
```

(continues on next page)

(continued from previous page)

```
page.title = 'Persistent Title'
new_revision = page.save_revision()
if page.live:
    # page has been created and published at the same time,
    # so ensure that the updated title is on the published version too
    new_revision.publish()
```

before_create_page

Called at the beginning of the “create page” view passing in the request, the parent page and page model class.

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 and skip the rest of the view.

Unlike, `after_create_page`, this is run both for both GET and POST requests.

This can be used to completely override the editor on a per-view basis:

```
from wagtail import hooks

from .models import AwesomePage
from .admin_views import edit_awesome_page

@hooks.register('before_create_page')
def before_create_page(request, parent_page, page_class):
    # Use a custom create view for the AwesomePage model
    if page_class == AwesomePage:
        return create_awesome_page(request, parent_page)
```

after_delete_page

Do something after a Page object is deleted. Uses the same behaviour as `after_create_page`.

before_delete_page

Called at the beginning of the “delete page” view passing in the request and the page object.

Uses the same behaviour as `before_create_page`, is is run both for both GET and POST requests.

```
from django.shortcuts import redirect
from django.utils.html import format_html

from wagtail.admin import messages
from wagtail import hooks

from .models import AwesomePage

@hooks.register('before_delete_page')
def before_delete_page(request, page):
```

(continues on next page)

(continued from previous page)

```
"""Block awesome page deletion and show a message."""

if request.method == 'POST' and page.specific_class in [AwesomePage]:
    messages.warning(request, "Awesome pages cannot be deleted, only unpublished")
    return redirect('wagtailadmin_pages:delete', page.pk)
```

after_edit_page

Do something with a Page object after it has been updated. Uses the same behaviour as `after_create_page`.

before_edit_page

Called at the beginning of the “edit page” view passing in the request and the page object.

Uses the same behaviour as `before_create_page`.

after_publish_page

Do something with a Page object after it has been published via page create view or page edit view.

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 and skip the rest of the view.

before_publish_page

Do something with a Page object before it has been published via page create view or page edit view.

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 and skip the rest of the view.

after_unpublish_page

Called after unpublish action in “unpublish” view passing in the request and the 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 and skip the rest of the view.

before_unpublish_page

Called before unpublish action in “unpublish” view passing in the request and the 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 and skip the rest of the view.

after_copy_page

Do something with a Page object after it has been copied passing in the request, page object and the new copied page. Uses the same behaviour as `after_create_page`.

before_copy_page

Called at the beginning of the “copy page” view passing in the request and the page object.

Uses the same behaviour as `before_create_page`.

after_move_page

Do something with a Page object after it has been moved passing in the request and page object. Uses the same behaviour as `after_create_page`.

before_move_page

Called at the beginning of the “move page” view passing in the request, the page object and the destination page object.

Uses the same behaviour as `before_create_page`.

before_convert_alias_page

Called at the beginning of the `convert_alias` view, which is responsible for converting alias pages into normal Wagtail pages.

The request and the page being converted are passed in as arguments to the hook.

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 and skip the rest of the view.

after_convert_alias_page

Do something with a Page object after it has been converted from an alias.

The request and the page that was just converted are passed in as arguments to the hook.

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 and skip the rest of the view.

construct_translated_pages_to_cascade_actions

Return additional pages to process in a synced tree setup.

This hook is only triggered on unpublishing a page when `WAGTAIL_I18N_ENABLED = True`.

The list of pages and the action are passed in as arguments to the hook.

The function should return a dictionary with the page from the pages list as key, and a list of additional pages to perform the action on. We recommend they are non-aliased, direct translations of the pages from the function argument.

register_page_action_menu_item

Add an item to the popup menu of actions on the page creation and edit views. The callable passed to this hook must return an instance of `wagtail.admin.action_menu.ActionMenuItem`. `ActionMenuItem` is a subclass of `Component` and so the rendering of the menu item can be customised through `template_name`, `get_context_data`, `render_html` and `Media`. In addition, the following attributes and methods are available to be overridden:

- `order` - an integer (default 100) which determines the item's position in the menu. Can also be passed as a keyword argument to the object constructor. The lowest-numbered item in this sequence will be selected as the default menu item; as standard, this is "Save draft" (which has an `order` of 0).
- `label` - the displayed text of the menu item
- `get_url` - a method which returns a URL for the menu item to link to; by default, returns `None` which causes the menu item to behave as a form submit button instead
- `name` - value of the `name` attribute of the submit button, if no URL is specified
- `icon_name` - icon to display against the menu item
- `classname` - a `class` attribute value to add to the button element
- `is_shown` - a method which returns a boolean indicating whether the menu item should be shown; by default, `true` except when editing a locked page

The `get_url`, `is_shown`, `get_context_data` and `render_html` methods all accept a context dictionary containing the following fields:

- `view` - name of the current view: '`create`', '`edit`' or '`revisions_revert`'
- `page` - for `view = 'edit'` or '`revisions_revert`', the page being edited
- `parent_page` - for `view = 'create'`, the parent page of the page being created
- `request` - the current request object
- `user_page_permissions` - a `UserPagePermissionsProxy` object for the current user, to test permissions against

```
from wagtail import hooks
from wagtail.admin.action_menu import ActionMenuItem

class GuacamoleMenuItem(ActionMenuItem):
    name = 'action-guacamole'
    label = "Guacamole"

    def get_url(self, context):
        return "https://www.youtube.com/watch?v=dNJdJIwCF_Y"

@hooks.register('register_page_action_menu_item')
def register_guacamole_menu_item():
    return GuacamoleMenuItem(order=10)
```

construct_page_action_menu

Modify the final list of action menu items on the page creation and edit views. The callable passed to this hook receives a list of `ActionMenuItem` objects, a request object and a context dictionary as per `register_page_action_menu_item`, and should modify the list of menu items in-place.

```
@hooks.register('construct_page_action_menu')
def remove_submit_to_moderator_option(menu_items, request, context):
    menu_items[:] = [item for item in menu_items if item.name != 'action-submit']
```

The `construct_page_action_menu` hook is called after the menu items have been sorted by their order attributes, and so setting a menu item's order will have no effect at this point. Instead, items can be reordered by changing their position in the list, with the first item being selected as the default action. For example, to change the default action to Publish:

```
@hooks.register('construct_page_action_menu')
def make_publish_default_action(menu_items, request, context):
    for index, item in enumerate(menu_items):
        if item.name == 'action-publish':
            # move to top of list
            menu_items.pop(index)
            menu_items.insert(0, item)
            break
```

construct_page_listing_buttons

Modify the final list of page listing buttons in the page explorer. The callable passed to this hook receives a list of `PageListingButton` objects, a page, a page perms object, and a context dictionary as per `register_page_listing_buttons`, and should modify the list of listing items in-place.

```
@hooks.register('construct_page_listing_buttons')
def remove_page_listing_button_item(buttons, page, page_perms, context=None):
    if page.is_root:
        buttons.pop() # removes the last 'more' dropdown button on the root page listing
    ↵buttons
```

construct_wagtail_userbar

Add or remove items from the wagtail userbar. Add, edit, and moderation tools are provided by default. The callable passed into the hook must take the `request` object and a list of menu objects, `items`. The menu item objects must have a `render` method which can take a `request` object and return the HTML string representing the menu item. See the userbar templates and menu item classes for more information.

```
from wagtail import hooks

class UserbarPuppyLinkItem:
    def render(self, request):
        return '<li><a href="http://cuteoverload.com/tag/puppehs/" ' \
               + 'target="_parent" role="menuitem" class="action icon icon-wagtail">Puppies!
    ↵</a></li>'
```

(continues on next page)

(continued from previous page)

```
@hooks.register('construct_wagtail_userbar')
def add_puppy_link_item(request, items):
    return items.append( UserbarPuppyLinkItem() )
```

Admin workflow

Hooks for customising the way admins are directed through the process of editing users.

`after_create_user`

Do something with a `User` object after it has been saved to the database. The callable passed to this hook should take a `request` object and a `user` 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 User index page.

```
from django.http import HttpResponseRedirect

from wagtail import hooks

@hooks.register('after_create_user')
def do_after_page_create(request, user):
    return HttpResponseRedirect("Congrats on creating a new user!", content_type="text/plain")
```

`before_create_user`

Called at the beginning of the “create user” view passing in the request.

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 and skip the rest of the view.

Unlike, `after_create_user`, this is run both for both GET and POST requests.

This can be used to completely override the user editor on a per-view basis:

```
from django.http import HttpResponseRedirect

from wagtail import hooks

from .models import AwesomePage
from .admin_views import edit_awesome_page

@hooks.register('before_create_user')
def before_create_page(request):
    return HttpResponseRedirect("A user creation form", content_type="text/plain")
```

after_delete_user

Do something after a User object is deleted. Uses the same behaviour as `after_create_user`.

before_delete_user

Called at the beginning of the “delete user” view passing in the request and the user object.

Uses the same behaviour as `before_create_user`.

after_edit_user

Do something with a User object after it has been updated. Uses the same behaviour as `after_create_user`.

before_edit_user

Called at the beginning of the “edit user” view passing in the request and the user object.

Uses the same behaviour as `before_create_user`.

Choosers

construct_page_chooser_queryset

Called when rendering the page chooser view, to allow the page listing QuerySet to be customised. The callable passed into the hook will receive the current page QuerySet and the request object, and must return a Page QuerySet (either the original one, or a new one).

```
from wagtail import hooks

@hooks.register('construct_page_chooser_queryset')
def show_my_pages_only(pages, request):
    # Only show own pages
    pages = pages.filter(owner=request.user)

    return pages
```

construct_document_chooser_queryset

Called when rendering the document chooser view, to allow the document listing QuerySet to be customised. The callable passed into the hook will receive the current document QuerySet and the request object, and must return a Document QuerySet (either the original one, or a new one).

```
from wagtail import hooks

@hooks.register('construct_document_chooser_queryset')
def show_my_uploaded_documents_only(documents, request):
    # Only show uploaded documents
    documents = documents.filter(uploaded_by_user=request.user)
```

(continues on next page)

(continued from previous page)

```
return documents
```

construct_image_chooser_queryset

Called when rendering the image chooser view, to allow the image listing QuerySet to be customised. The callable passed into the hook will receive the current image QuerySet and the request object, and must return an Image QuerySet (either the original one, or a new one).

```
from wagtail import hooks

@hooks.register('construct_image_chooser_queryset')
def show_my_uploaded_images_only(images, request):
    # Only show uploaded images
    images = images.filter(uploaded_by_user=request.user)

return images
```

Page explorer

construct_explorer_page_queryset

Called when rendering the page explorer view, to allow the page listing QuerySet to be customised. The callable passed into the hook will receive the parent page object, the current page QuerySet, and the request object, and must return a Page QuerySet (either the original one, or a new one).

```
from wagtail import hooks

@hooks.register('construct_explorer_page_queryset')
def show_my_profile_only(parent_page, pages, request):
    # If we're in the 'user-profiles' section, only show the user's own profile
    if parent_page.slug == 'user-profiles':
        pages = pages.filter(owner=request.user)

return pages
```

register_page_listing_buttons

Add buttons to the actions list for a page in the page explorer. This is useful when adding custom actions to the listing, such as translations or a complex workflow.

This example will add a simple button to the listing:

```
from wagtail.admin import widgets as wagtailadmin_widgets

@hooks.register('register_page_listing_buttons')
def page_listing_buttons(page, page_perms, next_url=None):
    yield wagtailadmin_widgets.PageListingButton(
```

(continues on next page)

(continued from previous page)

```
'A page listing button',
'/goes/to/a/url/',
priority=10
)
```

The arguments passed to the hook are as follows:

- `page` - the page object to generate the button for
- `page_perms` - a `PagePermissionTester` object that can be queried to determine the current user's permissions on the given page
- `next_url` - the URL that the linked action should redirect back to on completion of the action, if the view supports it

The `priority` argument controls the order the buttons are displayed in. Buttons are ordered from low to high priority, so a button with `priority=10` will be displayed before a button with `priority=20`.

`register_page_listing_more_buttons`

Add buttons to the “More” dropdown menu for a page in the page explorer. This works similarly to the `register_page_listing_buttons` hook but is useful for lesser-used custom actions that are better suited for the dropdown.

This example will add a simple button to the dropdown menu:

```
from wagtail.admin import widgets as wagtailadmin_widgets

@hooks.register('register_page_listing_more_buttons')
def page_listing_more_buttons(page, page_perms, next_url=None):
    yield wagtailadmin_widgets.Button(
        'A dropdown button',
        '/goes/to/a/url/',
        priority=60
    )
```

The arguments passed to the hook are as follows:

- `page` - the page object to generate the button for
- `page_perms` - a `PagePermissionTester` object that can be queried to determine the current user's permissions on the given page
- `next_url` - the URL that the linked action should redirect back to on completion of the action, if the view supports it

The `priority` argument controls the order the buttons are displayed in the dropdown. Buttons are ordered from low to high priority, so a button with `priority=10` will be displayed before a button with `priority=60`.

Buttons with dropdown lists

The admin widgets also provide `ButtonWithDropdownFromHook`, which allows you to define a custom hook for generating a dropdown menu that gets attached to your button.

Creating a button with a dropdown menu involves two steps. Firstly, you add your button to the `register_page_listing_buttons` hook, just like the example above. Secondly, you register a new hook that yields the contents of the dropdown menu.

This example shows how Wagtail's default admin dropdown is implemented. You can also see how to register buttons conditionally, in this case by evaluating the `page_perms`:

```
from wagtail.admin import widgets as wagtailadmin_widgets

@hooks.register('register_page_listing_buttons')
def page_custom_listing_buttons(page, page_perms, next_url=None):
    yield wagtailadmin_widgets.ButtonWithDropdownFromHook(
        'More actions',
        hook_name='my_button_dropdown_hook',
        page=page,
        page_perms=page_perms,
        next_url=next_url,
        priority=50
    )

@hooks.register('my_button_dropdown_hook')
def page_custom_listing_more_buttons(page, page_perms, next_url=None):
    if page_perms.can_move():
        yield wagtailadmin_widgets.Button('Move', reverse('wagtailadmin_pages:move', args=[page.id]), priority=10)
    if page_perms.can_delete():
        yield wagtailadmin_widgets.Button('Delete', reverse('wagtailadmin_pages:delete', args=[page.id]), priority=30)
    if page_perms.can_unpublish():
        yield wagtailadmin_widgets.Button('Unpublish', reverse('wagtailadmin_pages:unpublish', args=[page.id]), priority=40)
```

The template for the dropdown button can be customised by overriding `wagtailadmin/pages/listing/_button_with_dropdown.html`. The JavaScript that runs the dropdowns makes use of custom data attributes, so you should leave `data-dropdown` and `data-dropdown-toggle` in the markup if you customise it.

Page serving

`before_serve_page`

Called when Wagtail is about to serve a page. The callable passed into the hook will receive the `page` object, the `request` object, and the `args` and `kwargs` that will be passed to the page's `serve()` method. If the callable returns an `HttpResponse`, that response will be returned immediately to the user, and Wagtail will not proceed to call `serve()` on the page.

```
from django.http import HttpResponse

from wagtail import hooks
```

(continues on next page)

(continued from previous page)

```
@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>")
```

Document serving

before_serve_document

Called when Wagtail is about to serve a document. The callable passed into the hook will receive the document object and the request object. If the callable returns an `HttpResponse`, that response will be returned immediately to the user, instead of serving the document. Note that this hook will be skipped if the `WAGTAILDOCS_SERVE_METHOD` setting is set to `direct`.

Snippets

Hooks for working with registered Snippets.

after_edit_snippet

Called when a Snippet is edited. The callable passed into the hook will receive the model instance, the request object. If the callable returns an `HttpResponse`, that response will be returned immediately to the user, and Wagtail will not proceed to call `redirect()` to the listing view.

```
from django.http import HttpResponse

from wagtail import hooks

@hooks.register('after_edit_snippet')
def after_snippet_update(request, instance):
    return HttpResponse(f"Congrats on editing a snippet with id {instance.pk}", content_type="text/plain")
```

before_edit_snippet

Called at the beginning of the edit snippet view. The callable passed into the hook will receive the model instance, the request object. If the callable returns an `HttpResponse`, that response will be returned immediately to the user, and Wagtail will not proceed to call `redirect()` to the listing view.

```
from django.http import HttpResponse

from wagtail import hooks

@hooks.register('before_edit_snippet')
def block_snippet_edit(request, instance):
    if isinstance(instance, RestrictedSnippet) and instance.prevent_edit:
        return HttpResponse("Sorry, you can't edit this snippet", content_type="text/plain")
```

(continues on next page)

(continued from previous page)

after_create_snippet

Called when a Snippet is created. `after_create_snippet` and `after_edit_snippet` work in identical ways. The only difference is where the hook is called.

before_create_snippet

Called at the beginning of the create snippet view. Works in a similar way to `before_edit_snippet` except the model is passed as an argument instead of an instance.

after_delete_snippet

Called when a Snippet is deleted. The callable passed into the hook will receive the model instance(s) as a queryset along with the request object. If the callable returns an `HttpResponse`, that response will be returned immediately to the user, and Wagtail will not proceed to call `redirect()` to the listing view.

```
from django.http import HttpResponse

from wagtail import hooks

@hooks.register('after_delete_snippet')
def after_snippet_delete(request, instances):
    # "instances" is a QuerySet
    total = len(instances)
    return HttpResponse(f"{total} snippets have been deleted", content_type="text/plain")
```

before_delete_snippet

Called at the beginning of the delete snippet view. The callable passed into the hook will receive the model instance(s) as a queryset along with the request object. If the callable returns an `HttpResponse`, that response will be returned immediately to the user, and Wagtail will not proceed to call `redirect()` to the listing view.

```
from django.http import HttpResponse

from wagtail import hooks

@hooks.register('before_delete_snippet')
def before_snippet_delete(request, instances):
    # "instances" is a QuerySet
    total = len(instances)

    if request.method == 'POST':
        # Override the deletion behaviour
        instances.delete()

        return HttpResponse(f"{total} snippets have been deleted", content_type="text/plain")
```

(continues on next page)

(continued from previous page)

register_snippet_action_menu_item

Add an item to the popup menu of actions on the snippet creation and edit views. The callable passed to this hook must return an instance of `wagtail.snippets.action_menu.ActionMenuItem`. `ActionMenuItem` is a subclass of `Component` and so the rendering of the menu item can be customised through `template_name`, `get_context_data`, `render_html` and `Media`. In addition, the following attributes and methods are available to be overridden:

- `order`- an integer (default 100) which determines the item's position in the menu. Can also be passed as a keyword argument to the object constructor. The lowest-numbered item in this sequence will be selected as the default menu item; as standard, this is "Save draft" (which has an order of 0).
- `label` - the displayed text of the menu item
- `get_url` - a method which returns a URL for the menu item to link to; by default, returns `None` which causes the menu item to behave as a form submit button instead
- `name` - value of the `name` attribute of the submit button if no URL is specified
- `icon_name` - icon to display against the menu item
- `classname` - a `class` attribute value to add to the button element
- `is_shown` - a method which returns a boolean indicating whether the menu item should be shown; by default, `true` except when editing a locked page

The `get_url`, `is_shown`, `get_context_data` and `render_html` methods all accept a context dictionary containing the following fields:

- `view` - name of the current view: '`create`' or '`edit`'
- `model` - the snippet's model class
- `instance` - for `view = 'edit'`, the instance being edited
- `request` - the current request object

```
from wagtail import hooks
from wagtail.snippets.action_menu import ActionMenuItem

class GuacamoleMenuItem(ActionMenuItem):
    name = 'action-guacamole'
    label = "Guacamole"

    def get_url(self, context):
        return "https://www.youtube.com/watch?v=dNJdJIwCF_Y"

@hooks.register('register_snippet_action_menu_item')
def register_guacamole_menu_item():
    return GuacamoleMenuItem(order=10)
```

construct_snippet_action_menu

Modify the final list of action menu items on the snippet creation and edit views. The callable passed to this hook receives a list of `ActionMenuItem` objects, a request object and a context dictionary as per `register_snippet_action_menu_item`, and should modify the list of menu items in-place.

```
@hooks.register('construct_snippet_action_menu')
def remove_delete_option(menu_items, request, context):
    menu_items[:] = [item for item in menu_items if item.name != 'delete']
```

The `construct_snippet_action_menu` hook is called after the menu items have been sorted by their order attributes, and so setting a menu item's order will have no effect at this point. Instead, items can be reordered by changing their position in the list, with the first item being selected as the default action. For example, to change the default action to Delete:

```
@hooks.register('construct_snippet_action_menu')
def make_delete_default_action(menu_items, request, context):
    for index, item in enumerate(menu_items):
        if item.name == 'delete':
            # move to top of list
            menu_items.pop(index)
            menu_items.insert(0, item)
            break
```

register_snippet_listing_buttons

Add buttons to the actions list for a snippet in the snippets listing. This is useful when adding custom actions to the listing, such as translations or a complex workflow.

This example will add a simple button to the listing:

```
from wagtail.snippets import widgets as wagtailsnippets_widgets

@hooks.register('register_snippet_listing_buttons')
def snippet_listing_buttons(snippet, user, next_url=None):
    yield wagtailsnippets_widgets.SnippetListingButton(
        'A page listing button',
        '/goes/to/a/url/',
        priority=10
    )
```

The arguments passed to the hook are as follows:

- `snippet` - the snippet object to generate the button for
- `user` - the user who is viewing the snippets listing
- `next_url` - the URL that the linked action should redirect back to on completion of the action, if the view supports it

The `priority` argument controls the order the buttons are displayed in. Buttons are ordered from low to high priority, so a button with `priority=10` will be displayed before a button with `priority=20`.

construct_snippet_listing_buttons

Modify the final list of snippet listing buttons. The callable passed to this hook receives a list of `SnippetListingButton` objects, a user, and a context dictionary as per `register_snippet_listing_buttons`, and should modify the list of menu items in-place.

```
@hooks.register('construct_snippet_listing_buttons')
def remove_snippet_listing_button_item(buttons, snippet, user, context=None):
    buttons.pop() # Removes the 'delete' button
```

Bulk actions

Hooks for registering and customising bulk actions. See [Adding custom bulk actions](#) on how to write custom bulk actions.

register_bulk_action

Registers a new bulk action to add to the list of bulk actions in the explorer

This hook must be registered with a sub-class of `BulkAction`. For example:

```
from wagtail.admin.views.bulk_action import BulkAction
from wagtail import hooks

@hooks.register("register_bulk_action")
class CustomBulkAction(BulkAction):
    display_name = _("Custom Action")
    action_type = "action"
    aria_label = _("Do custom action")
    template_name = "/path/to/template"
    models = [...] # list of models the action should execute upon

    @classmethod
    def execute_action(cls, objects, **kwargs):
        for object in objects:
            do_something(object)
        return num_parent_objects, num_child_objects # return the count of updated
    ↵objects
```

before_bulk_action

Do something right before a bulk action is executed (before the `execute_action` method is called)

This hook can be used to return an HTTP response. For example:

```
from wagtail import hooks

@hooks.register("before_bulk_action")
def hook_func(request, action_type, objects, action_class_instance):
```

(continues on next page)

(continued from previous page)

```
if action_type == 'delete':
    return HttpResponse(f"{len(objects)} objects would be deleted", content_type="text/plain")
```

after_bulk_action

Do something right after a bulk action is executed (after the `execute_action` method is called)

This hook can be used to return an HTTP response. For example:

```
from wagtail import hooks

@hooks.register("after_bulk_action")
def hook_func(request, action_type, objects, action_class_instance):
    if action_type == 'delete':
        return HttpResponse(f"{len(objects)} objects have been deleted", content_type="text/plain")
```

Audit log

register_log_actions

See [Audit log](#)

To add new actions to the registry, call the `register_action` method with the action type, its label and the message to be displayed in administrative listings.

```
from django.utils.translation import gettext_lazy as _

from wagtail import hooks

@hooks.register('register_log_actions')
def additional_log_actions(actions):
    actions.register_action('wagtail_package.echo', _('Echo'), _('Sent an echo'))
```

Alternatively, for a log message that varies according to the log entry's data, create a subclass of `wagtail.log_actions.LogFormatter` that overrides the `format_message` method, and use `register_action` as a decorator on that class:

```
from django.utils.translation import gettext_lazy as _

from wagtail import hooks
from wagtail.log_actions import LogFormatter

@hooks.register('register_log_actions')
def additional_log_actions(actions):
    @actions.register_action('wagtail_package.greet_audience')
    class GreetingActionFormatter(LogFormatter):
        label = _('Greet audience')
```

(continues on next page)

(continued from previous page)

```
def format_message(self, log_entry):
    return _('Hello %(audience)s') % {
        'audience': log_entry.data['audience'],
    }
```

Changed in version 2.15: The `LogFormatter` class was introduced. Previously, dynamic messages were achieved by passing a callable as the `message` argument to `register_action`.

1.5.6 Signals

Wagtail's `Revision` and `Page` implement `Signals` from `django.dispatch`. Signals are useful for creating side-effects from page publish/unpublish events.

For example, you could use signals to send publish notifications to a messaging service, or POST messages to another app that's consuming the API, such as a static site generator.

page_published

This signal is emitted from a `Revision` when a page revision is set to published.

- `sender` - The page class.
- `instance` - The specific Page instance.
- `revision` - The Revision that was published.
- `kwargs` - Any other arguments passed to `page_published.send()`.

To listen to a signal, implement `page_published.connect(receiver, sender, **kwargs)`. Here's a simple example showing how you might notify your team when something is published:

```
from wagtail.signals import page_published
import requests

# Let everyone know when a new page is published
def send_to_slack(sender, **kwargs):
    instance = kwargs['instance']
    url = 'https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXX'
    values = {
        "text": "%s was published by %s" % (instance.title, instance.owner.username),
        "channel": "#publish-notifications",
        "username": "the squid of content",
        "icon_emoji": ":octopus:"
    }
    response = requests.post(url, values)

# Register a receiver
page_published.connect(send_to_slack)
```

Receiving specific model events

Sometimes you're not interested in receiving signals for every model, or you want to handle signals for specific models in different ways. For instance, you may wish to do something when a new blog post is published:

```
from wagtail.signals import page_published
from mysite.models import BlogPostPage

# Do something clever for each model type
def receiver(sender, **kwargs):
    # Do something with blog posts
    pass

# Register listeners for each page model class
page_published.connect(receiver, sender=BlogPostPage)
```

Wagtail provides access to a list of registered page types through the `get_page_models()` function in `wagtail.models`.

Read the [Django documentation](#) for more information about specifying senders.

`page_unpublished`

This signal is emitted from a Page when the page is unpublished.

- `sender` - The page class.
- `instance` - The specific Page instance.
- `kwargs` - Any other arguments passed to `page_unpublished.send()`

`pre_page_move` and `post_page_move`

These signals are emitted from a Page immediately before and after it is moved.

Subscribe to `pre_page_move` if you need to know values BEFORE any database changes are applied. For example: Getting the page's previous URL, or that of its descendants.

Subscribe to `post_page_move` if you need to know values AFTER database changes have been applied. For example: Getting the page's new URL, or that of its descendants.

The following arguments are emitted for both signals:

- `sender` - The page class.
- `instance` - The specific Page instance.
- `parent_page_before` - The parent page of `instance` **before** moving.
- `parent_page_after` - The parent page of `instance` **after** moving.
- `url_path_before` - The value of `instance.url_path` **before** moving.
- `url_path_after` - The value of `instance.url_path` **after** moving.
- `kwargs` - Any other arguments passed to `pre_page_move.send()` or `post_page_move.send()`.

Distinguishing between a ‘move’ and a ‘reorder’

The signal can be emitted as a result of a page being moved to a different section (a ‘move’), or as a result of a page being moved to a different position within the same section (a ‘reorder’). Knowing the difference between the two can be particularly useful, because only a ‘move’ affects a page’s URL (and that of its descendants), whereas a ‘reorder’ only affects the natural page order; which is probably less impactful.

The best way to distinguish between a ‘move’ and ‘reorder’ is to compare the `url_path_before` and `url_path_after` values. For example:

```
from wagtail.signals import pre_page_move
from wagtail.contrib.frontend_cache.utils import purge_page_from_cache

# Clear a page's old URLs from the cache when it moves to a different section
def clear_page_url_from_cache_on_move(sender, **kwargs):

    if kwargs['url_path_before'] == kwargs['url_path_after']:
        # No URLs are changing :) nothing to do here!
        return

    # The page is moving to a new section (possibly even a new site)
# so clear old URL(s) from the cache
purge_page_from_cache(kwargs['instance'])

# Register a receiver
pre_page_move.connect(clear_old_page_urls_from_cache)
```

page_slug_changed

This signal is emitted from a Page when a change to its slug is published.

The following arguments are emitted by this signal:

- `sender` - The page class.
- `instance` - The updated (and saved), specific Page instance.
- `instance_before` - A copy of the specific Page instance from **before** the changes were saved.

workflow_submitted

This signal is emitted from a `WorkflowState` when a page is submitted to a workflow.

- `sender` - `WorkflowState`
- `instance` - The specific `WorkflowState` instance.
- `user` - The user who submitted the workflow
- `kwargs` - Any other arguments passed to `workflow_submitted.send()`

workflow_rejected

This signal is emitted from a `WorkflowState` when a page is rejected from a workflow.

- `sender` - `WorkflowState`
- `instance` - The specific `WorkflowState` instance.
- `user` - The user who rejected the workflow
- `kwargs` - Any other arguments passed to `workflow_rejected.send()`

workflow_approved

This signal is emitted from a `WorkflowState` when a page's workflow completes successfully

- `sender` - `WorkflowState`
- `instance` - The specific `WorkflowState` instance.
- `user` - The user who last approved the workflow
- `kwargs` - Any other arguments passed to `workflow_approved.send()`

workflow_cancelled

This signal is emitted from a `WorkflowState` when a page's workflow is cancelled

- `sender` - `WorkflowState`
- `instance` - The specific `WorkflowState` instance.
- `user` - The user who cancelled the workflow
- `kwargs` - Any other arguments passed to `workflow_cancelled.send()`

task_submitted

This signal is emitted from a `TaskState` when a page is submitted to a task.

- `sender` - `TaskState`
- `instance` - The specific `TaskState` instance.
- `user` - The user who submitted the page to the task
- `kwargs` - Any other arguments passed to `task_submitted.send()`

task_rejected

This signal is emitted from a `TaskState` when a page is rejected from a task.

- `sender` - `TaskState`
- `instance` - The specific `TaskState` instance.
- `user` - The user who rejected the task
- `kwargs` - Any other arguments passed to `task_rejected.send()`

task_approved

This signal is emitted from a TaskState when a page's task is approved

- `sender` - TaskState
- `instance` - The specific TaskState instance.
- `user` - The user who approved the task
- `kwargs` - Any other arguments passed to `task_approved.send()`

task_cancelled

This signal is emitted from a TaskState when a page's task is cancelled.

- `sender` - TaskState
- `instance` - The specific TaskState instance.
- `user` - The user who cancelled the task
- `kwargs` - Any other arguments passed to `task_cancelled.send()`

1.5.7 Settings

Wagtail makes use of the following settings, in addition to Django's core settings`:

Sites

WAGTAIL_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.

WAGTAILADMIN_BASE_URL

```
WAGTAILADMIN_BASE_URL = 'http://example.com'
```

This is the base URL used by the Wagtail admin site. It is typically used for generating URLs to include in notification emails.

If this setting is not present, Wagtail will try to fall back to `request.site.root_url` or to the request's host name.

Changed in version 3.0: This setting was previously named `BASE_URL` and was undocumented, using `BASE_URL` will be removed in a future release.

Append Slash

WAGTAIL_APPEND_SLASH

```
# Don't add a trailing slash to Wagtail-served URLs
WAGTAIL_APPEND_SLASH = False
```

Similar to Django's APPEND_SLASH, this setting controls how Wagtail will handle requests that don't end in a trailing slash.

When WAGTAIL_APPEND_SLASH is True (default), requests to Wagtail pages which omit a trailing slash will be redirected by Django's [CommonMiddleware](#) to a URL with a trailing slash.

When WAGTAIL_APPEND_SLASH is False, requests to Wagtail pages will be served both with and without trailing slashes. Page links generated by Wagtail, however, will not include trailing slashes.

Note: If you use the False setting, keep in mind that serving your pages both with and without slashes may affect search engines' ability to index your site. See this [Google Search Central Blog post](#) for more details.

Search

WAGTAILSEARCH_BACKENDS

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.search.backends.elasticsearch5',
        'INDEX': 'myapp'
    }
}
```

Define a search backend. For a full explanation, see [Backends](#).

WAGTAILSEARCH_HITS_MAX_AGE

```
WAGTAILSEARCH_HITS_MAX_AGE = 14
```

Set the number of days (default 7) that search query logs are kept for; these are used to identify popular search terms for [promoted search results](#). Queries older than this will be removed by the `search_garbage_collect` command.

Internationalisation

Wagtail supports internationalisation of content by maintaining separate trees of pages for each language.

For a guide on how to enable internationalisation on your site, see the [configuration guide](#).

WAGTAIL_I18N_ENABLED

(boolean, default `False`)

When set to `True`, Wagtail's internationalisation features will be enabled:

```
WAGTAIL_I18N_ENABLED = True
```

WAGTAIL_CONTENT_LANGUAGES

(list, default `[]`)

A list of languages and/or locales that Wagtail content can be authored in.

For example:

```
WAGTAIL_CONTENT_LANGUAGES = [
    ('en', _("English")),
    ('fr', _("French")),
]
```

Each item in the list is a 2-tuple containing a language code and a display name. The language code can either be a language code on its own (such as `en`, `fr`), or it can include a region code (such as `en-gb`, `fr-fr`). You can mix the two formats if you only need to localize in some regions but not others.

This setting follows the same structure of Django's `LANGUAGES` setting, so they can both be set to the same value:

```
LANGUAGES = WAGTAIL_CONTENT_LANGUAGES = [
    ('en-gb', _("English (United Kingdom)")),
    ('en-us', _("English (United States)")),
    ('es-es', _("Spanish (Spain)")),
    ('es-mx', _("Spanish (Mexico)")),
]
```

However having them separate allows you to configure many different regions on your site yet have them share Wagtail content (but defer on things like date formatting, currency, etc):

```
LANGUAGES = [
    ('en', _("English (United Kingdom)")),
    ('en-us', _("English (United States)")),
    ('es', _("Spanish (Spain)")),
    ('es-mx', _("Spanish (Mexico)")),
]
```

```
WAGTAIL_CONTENT_LANGUAGES = [
    ('en', _("English")),
    ('es', _("Spanish")),
]
```

This would mean that your site will respond on the `https://www.mysite.com/es/` and `https://www.mysite.com/es-MX/` URLs, but both of them will serve content from the same "Spanish" tree in Wagtail.

Note: `WAGTAIL_CONTENT_LANGUAGES` must be a subset of `LANGUAGES`

Note that all languages that exist in `WAGTAIL_CONTENT_LANGUAGES` must also exist in your `LANGUAGES` setting. This is so that Wagtail can generate a live URL to these pages from an untranslated context (such as the admin interface).

Embeds

Wagtail supports generating embed code from URLs to content on an external providers such as Youtube or Twitter. By default, Wagtail will fetch the embed code directly from the relevant provider's site using the oEmbed protocol. Wagtail has a builtin list of the most common providers.

The embeds fetching can be fully configured using the `WAGTAILEMBEDS_FINDERS` setting. This is fully documented in [Configuring embed “finders”](#).

`WAGTAILEMBEDS_RESPONSIVE_HTML`

```
WAGTAILEMBEDS_RESPONSIVE_HTML = True
```

Adds `class="responsive-object"` and an inline `padding-bottom` style to embeds, to assist in making them responsive. See [Responsive Embeds](#) for details.

Dashboard

`WAGTAILADMIN_RECENT_EDITS_LIMIT`

```
WAGTAILADMIN_RECENT_EDITS_LIMIT = 5
```

This setting lets you change the number of items shown at ‘Your most recent edits’ on the dashboard.

General editing

`WAGTAILADMIN_RICH_TEXT_EDITORS`

```
WAGTAILADMIN_RICH_TEXT_EDITORS = {
    'default': {
        'WIDGET': 'wagtail.admin.rich_text.DraftailRichTextArea',
        'OPTIONS': {
            'features': ['h2', 'bold', 'italic', 'link', 'document-link']
        }
    },
    'secondary': {
        'WIDGET': 'some.external.RichTextEditor',
    }
}
```

Customise the behaviour of rich text fields. By default, `RichTextField` and `RichTextBlock` use the configuration given under the `'default'` key, but this can be overridden on a per-field basis through the `editor` keyword argument, for example `body = RichTextField(editor='secondary')`. Within each configuration block, the following fields are recognised:

- **WIDGET:** The rich text widget implementation to use. Wagtail provides `wagtail.admin.rich_text.DraftailRichTextArea` (a modern extensible editor which enforces well-structured markup). Other widgets may be provided by third-party packages.
- **OPTIONS:** Configuration options to pass to the widget. Recognised options are widget-specific, but `DraftailRichTextArea` accept a `features` list indicating the active rich text features (see [Limiting features in a rich text field](#)).

If a 'default' editor is not specified, rich text fields that do not specify an `editor` argument will use the Draftail editor with the default feature set enabled.

WAGTAILADMIN_EXTERNAL_LINK_CONVERSION

```
WAGTAILADMIN_EXTERNAL_LINK_CONVERSION = 'exact'
```

Customise Wagtail's behaviour when an internal page url is entered in the external link chooser. Possible values for this setting are 'all', 'exact', 'confirm', or ''. The default, 'all', means that Wagtail will automatically convert submitted urls that exactly match page urls to the corresponding internal links. If the url is an inexact match - for example, the submitted url has query parameters - then Wagtail will confirm the conversion with the user. 'exact' means that any inexact matches will be left as external urls, and the confirmation step will be skipped. 'confirm' means that every link conversion will be confirmed with the user, even if the match is exact. '' means that Wagtail will not attempt to convert any urls entered to internal page links.

WAGTAIL_DATE_FORMAT, WAGTAIL_DATETIME_FORMAT, WAGTAIL_TIME_FORMAT

```
WAGTAIL_DATE_FORMAT = '%d.%m.%Y.'
WAGTAIL_DATETIME_FORMAT = '%d.%m.%Y. %H:%M'
WAGTAIL_TIME_FORMAT = '%H:%M'
```

Specifies the date, time and datetime format to be used in input fields in the Wagtail admin. The format is specified in [Python datetime module syntax](#), and must be one of the recognised formats listed in the `DATE_INPUT_FORMATS`, `TIME_INPUT_FORMATS`, or `DATETIME_INPUT_FORMATS` setting respectively (see [DATE_INPUT_FORMATS](#)).

Page editing

WAGTAILADMIN_COMMENTS_ENABLED

```
# Disable commenting
WAGTAILADMIN_COMMENTS_ENABLED = False
```

Sets whether commenting is enabled for pages (True by default).

WAGTAIL_ALLOW_UNICODE_SLUGS

```
WAGTAIL_ALLOW_UNICODE_SLUGS = True
```

By default, page slugs can contain any alphanumeric characters, including non-Latin alphabets. Set this to `False` to limit slugs to ASCII characters.

WAGTAIL_AUTO_UPDATE_PREVIEW

```
WAGTAIL_AUTO_UPDATE_PREVIEW = True
```

When enabled, the preview panel in the page editor is automatically updated on each change. If set to `False`, a refresh button will be shown and the preview is only updated when the button is clicked. This behaviour is enabled by default.

To completely disable the preview panel, set `preview_modes` to be empty on your model `preview_modes = []`.

WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL

```
WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL = 500
```

The interval (in milliseconds) to check for changes made in the page editor before updating the preview. The default value is `500`.

WAGTAILADMIN_GLOBAL_PAGE_EDIT_LOCK

`WAGTAILADMIN_GLOBAL_PAGE_EDIT_LOCK` can be set to `True` to prevent users from editing pages that they have locked.

WAGTAILADMIN_UNSAFE_PAGE_DELETION_LIMIT

```
WAGTAILADMIN_UNSAFE_PAGE_DELETION_LIMIT = 20
```

This setting enables an additional confirmation step when deleting a page with a large number of child pages. If the number of pages is greater than or equal to this limit (10 by default), the user must enter the site name (as defined by `WAGTAIL_SITE_NAME`) to proceed.

Images

WAGTAILIMAGES_IMAGE_MODEL

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

This setting lets you provide your own image model for use in Wagtail, which should extend the built-in `AbstractImage` class.

WAGTAILIMAGES_IMAGE_FORM_BASE

```
WAGTAILIMAGES_IMAGE_FORM_BASE = 'myapp.forms.MyImageBaseForm'
```

This setting lets you provide your own image base form for use in Wagtail, which should extend the built-in `BaseImageForm` class. You can use it to specify or override the widgets to use in the admin form.

WAGTAILIMAGES_MAX_UPLOAD_SIZE

```
WAGTAILIMAGES_MAX_UPLOAD_SIZE = 20 * 1024 * 1024 # 20MB
```

This setting lets you override the maximum upload size for images (in bytes). If omitted, Wagtail will fall back to using its 10MB default value.

WAGTAILIMAGES_MAX_IMAGE_PIXELS

```
WAGTAILIMAGES_MAX_IMAGE_PIXELS = 128000000 # 128 megapixels
```

This setting lets you override the maximum number of pixels an image can have. If omitted, Wagtail will fall back to using its 128 megapixels default value. The pixel count takes animation frames into account - for example, a 25-frame animation of size 100x100 is considered to have $100 \times 100 \times 25 = 250000$ pixels.

WAGTAILIMAGES_FEATURE_DETECTION_ENABLED

```
WAGTAILIMAGES_FEATURE_DETECTION_ENABLED = True
```

This setting enables feature detection once OpenCV is installed, see all details on the [Feature Detection](#) documentation.

WAGTAILIMAGES_INDEX_PAGE_SIZE

```
WAGTAILIMAGES_INDEX_PAGE_SIZE = 20
```

Specifies the number of images per page shown on the main Images listing in the Wagtail admin.

WAGTAILIMAGES_USAGE_PAGE_SIZE

```
WAGTAILIMAGES_USAGE_PAGE_SIZE = 20
```

Specifies the number of items per page shown when viewing an image's usage (see [WAGTAIL_USAGE_COUNT_ENABLED](#)).

WAGTAILIMAGES_CHOOSER_PAGE_SIZE

```
WAGTAILIMAGES_CHOOSER_PAGE_SIZE = 12
```

Specifies the number of images shown per page in the image chooser modal.

WAGTAILIMAGES_RENDERING_STORAGE

```
WAGTAILIMAGES_RENDERING_STORAGE = 'myapp.backends.MyCustomStorage'
```

This setting allows image renditions to be stored using an alternative storage backend. The default is `None`, which will use Django's default `FileSystemStorage`.

Custom storage classes should subclass `django.core.files.storage.Storage`. See the [Django file storage API](#).

Documents

WAGTAILDOCS_DOCUMENT_MODEL

```
WAGTAILDOCS_DOCUMENT_MODEL = 'myapp.MyDocument'
```

This setting lets you provide your own document model for use in Wagtail, which should extend the built-in `AbstractDocument` class.

WAGTAILDOCS_DOCUMENT_FORM_BASE

```
WAGTAILDOCS_DOCUMENT_FORM_BASE = 'myapp.forms.MyDocumentBaseForm'
```

This setting lets you provide your own Document base form for use in Wagtail, which should extend the built-in `BaseDocumentForm` class. You can use it to specify or override the widgets to use in the admin form.

WAGTAILDOCS_SERVE_METHOD

```
WAGTAILDOCS_SERVE_METHOD = 'redirect'
```

Determines how document downloads will be linked to and served. Normally, requests for documents are sent through a Django view, to perform privacy checks (see [Privacy settings](#)) and potentially other housekeeping tasks such as hit counting. To fully protect against users bypassing this check, it needs to happen in the same request where the document is served; however, this incurs a performance hit as the document then needs to be served by the Django server. In particular, this cancels out much of the benefit of hosting documents on external storage, such as S3 or a CDN.

For this reason, Wagtail provides a number of serving methods which trade some of the strictness of the permission check for performance:

- '`direct`' - links to documents point directly to the URL provided by the underlying storage, bypassing the Django view that provides the permission check. This is most useful when deploying sites as fully static HTML (for example using `wagtail-bakery` or `Gatsby`).

- 'redirect' - links to documents point to a Django view which will check the user's permission; if successful, it will redirect to the URL provided by the underlying storage to allow the document to be downloaded. This is most suitable for remote storage backends such as S3, as it allows the document to be served independently of the Django server. Note that if a user is able to guess the latter URL, they will be able to bypass the permission check; some storage backends may provide configuration options to generate a random or short-lived URL to mitigate this.
- 'serve_view' - links to documents point to a Django view which both checks the user's permission, and serves the document. Serving will be handled by `django-sendfile`, if this is installed and supported by your server configuration, or as a streaming response from Django if not. When using this method, it is recommended that you configure your webserver to *disallow* serving documents directly from their location under `MEDIA_ROOT`, as this would provide a way to bypass the permission check.

If `WAGTAILDOCS_SERVE_METHOD` is unspecified or set to `None`, the default method is 'redirect' when a remote storage backend is in use (one that exposes a URL but not a local filesystem path), and 'serve_view' otherwise. Finally, some storage backends may not expose a URL at all; in this case, serving will proceed as for 'serve_view'.

WAGTAILDOCS_CONTENT_TYPES

```
WAGTAILDOCS_CONTENT_TYPES = {  
    'pdf': 'application/pdf',  
    'txt': 'text/plain',  
}
```

Specifies the MIME content type that will be returned for the given file extension, when using the `serve_view` method. Content types not listed here will be guessed using the Python `mimetypes.guess_type` function, or `application/octet-stream` if unsuccessful.

WAGTAILDOCS_INLINE_CONTENT_TYPES

```
WAGTAILDOCS_INLINE_CONTENT_TYPES = ['application/pdf', 'text/plain']
```

A list of MIME content types that will be shown inline in the browser (by serving the HTTP header `Content-Disposition: inline`) rather than served as a download, when using the `serve_view` method. Defaults to `application/pdf`.

WAGTAILDOCS_EXTENSIONS

```
WAGTAILDOCS_EXTENSIONS = ['pdf', 'docx']
```

A list of allowed document extensions that will be validated during document uploading. If this isn't supplied all document extensions are allowed. Warning: this doesn't always ensure that the uploaded file is valid as files can be renamed to have an extension no matter what data they contain.

User Management

`WAGTAIL_PASSWORD_MANAGEMENT_ENABLED`

```
WAGTAIL_PASSWORD_MANAGEMENT_ENABLED = True
```

This specifies whether users are allowed to change their passwords (enabled by default).

`WAGTAIL_PASSWORD_RESET_ENABLED`

```
WAGTAIL_PASSWORD_RESET_ENABLED = True
```

This specifies whether users are allowed to reset their passwords. Defaults to the same as `WAGTAIL_PASSWORD_MANAGEMENT_ENABLED`. Password reset emails will be sent from the address specified in Django's `DEFAULT_FROM_EMAIL` setting.

`WAGTAILUSERS_PASSWORD_ENABLED`

```
WAGTAILUSERS_PASSWORD_ENABLED = True
```

This specifies whether password fields are shown when creating or editing users through Settings -> Users (enabled by default). Set this to False (along with `WAGTAIL_PASSWORD_MANAGEMENT_ENABLED` and `WAGTAIL_PASSWORD_RESET_ENABLED`) if your users are authenticated through an external system such as LDAP.

`WAGTAILUSERS_PASSWORD_REQUIRED`

```
WAGTAILUSERS_PASSWORD_REQUIRED = True
```

This specifies whether password is a required field when creating a new user. True by default; ignored if `WAGTAILUSERS_PASSWORD_ENABLED` is false. If this is set to False, and the password field is left blank when creating a user, then that user will have no usable password; in order to log in, they will have to reset their password (if `WAGTAIL_PASSWORD_RESET_ENABLED` is True) or use an alternative authentication system such as LDAP (if one is set up).

`WAGTAIL_EMAIL_MANAGEMENT_ENABLED`

```
WAGTAIL_EMAIL_MANAGEMENT_ENABLED = True
```

This specifies whether users are allowed to change their email (enabled by default).

WAGTAILADMIN_USER_PASSWORD_RESET_FORM

```
WAGTAILADMIN_USER_PASSWORD_RESET_FORM = 'users.forms.PasswordResetForm'
```

Allows the default PasswordResetForm to be extended with extra fields.

WAGTAIL_USER_EDIT_FORM

```
WAGTAIL_USER_EDIT_FORM = 'users.forms.CustomUserEditForm'
```

Allows the default UserEditForm class to be overridden with a custom form when a custom user model is being used and extra fields are required in the user edit form.

For further information See [Custom user models](#).

WAGTAIL_USER_CREATION_FORM

```
WAGTAIL_USER_CREATION_FORM = 'users.forms.CustomUserCreationForm'
```

Allows the default UserCreationForm class to be overridden with a custom form when a custom user model is being used and extra fields are required in the user creation form.

For further information See [Custom user models](#).

WAGTAIL_USER_CUSTOM_FIELDS

```
WAGTAIL_USER_CUSTOM_FIELDS = ['country']
```

A list of the extra custom fields to be appended to the default list.

For further information See [Custom user models](#).

WAGTAILADMIN_USER_LOGIN_FORM

```
WAGTAILADMIN_USER_LOGIN_FORM = 'users.forms.LoginForm'
```

Allows the default LoginForm to be extended with extra fields.

User preferences

WAGTAIL_GRAVATAR_PROVIDER_URL

```
WAGTAIL_GRAVATAR_PROVIDER_URL = '//www.gravatar.com/avatar'
```

If a user has not uploaded a profile picture, Wagtail will look for an avatar linked to their email address on gravatar.com. This setting allows you to specify an alternative provider such as like robohash.org, or can be set to None to disable the use of remote avatars completely.

WAGTAIL_USER_TIME_ZONES

Logged-in users can choose their current time zone for the admin interface in the account settings. If no time zone selected by the user, then TIME_ZONE will be used. (Note that time zones are only applied to datetime fields, not to plain time or date fields. This is a Django design decision.)

The list of time zones is by default the common_timezones list from pytz. It is possible to override this list via the WAGTAIL_USER_TIME_ZONES setting. If there is zero or one time zone permitted, the account settings form will be hidden.

```
WAGTAIL_USER_TIME_ZONES = ['America/Chicago', 'Australia/Sydney', 'Europe/Rome']
```

WAGTAILADMIN_PERMITTED_LANGUAGES

Users can choose between several languages for the admin interface in the account settings. The list of languages is by default all the available languages in Wagtail with at least 90% coverage. To change it, set WAGTAILADMIN_PERMITTED_LANGUAGES:

```
WAGTAILADMIN_PERMITTED_LANGUAGES = [('en', 'English'), ('pt', 'Portuguese')]
```

Since the syntax is the same as Django LANGUAGES, you can do this so users can only choose between front office languages:

```
LANGUAGES = WAGTAILADMIN_PERMITTED_LANGUAGES = [('en', 'English'), ('pt', 'Portuguese')]
```

Email notifications

WAGTAILADMIN_NOTIFICATION_FROM_EMAIL

```
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, Wagtail will fall back to using Django's DEFAULT_FROM_EMAIL setting.

WAGTAILADMIN_NOTIFICATION_USE_HTML

```
WAGTAILADMIN_NOTIFICATION_USE_HTML = True
```

Notification emails are sent in `text/plain` by default, change this to use HTML formatting.

WAGTAILADMIN_NOTIFICATION_INCLUDE_SUPERUSERS

```
WAGTAILADMIN_NOTIFICATION_INCLUDE_SUPERUSERS = False
```

Notification emails are sent to moderators and superusers by default. You can change this to exclude superusers and only notify moderators.

Wagtail update notifications

WAGTAIL_ENABLE_UPDATE_CHECK

```
WAGTAIL_ENABLE_UPDATE_CHECK = True
```

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

If admins should only be informed of new long term support (LTS) versions, then set this setting to "lts" (the setting is case-insensitive).

Frontend authentication

PASSWORD_REQUIRED_TEMPLATE

```
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.

DOCUMENT_PASSWORD_REQUIRED_TEMPLATE

```
DOCUMENT_PASSWORD_REQUIRED_TEMPLATE = 'myapp/document_password_required.html'
```

As above, but for password restrictions on documents. For more details, see the [Private pages](#) documentation.

WAGTAIL_FRONTEND_LOGIN_TEMPLATE

The basic login page can be customised with a custom template.

```
WAGTAIL_FRONTEND_LOGIN_TEMPLATE = 'myapp/login.html'
```

WAGTAIL_FRONTEND_LOGIN_URL

Or the login page can be a redirect to an external or internal URL.

```
WAGTAIL_FRONTEND_LOGIN_URL = '/accounts/login/'
```

For more details, see the [Setting up a login page](#) documentation.

Tags

TAGGIT_CASE_INSENSITIVE

```
TAGGIT_CASE_INSENSITIVE = True
```

Tags are case-sensitive by default ('music' and 'Music' are treated as distinct tags). In many cases the reverse behaviour is preferable.

TAG_SPACES_ALLOWED

```
TAG_SPACES_ALLOWED = False
```

Tags can only consist of a single word, no spaces allowed. The default setting is `True` (spaces in tags are allowed).

TAG_LIMIT

```
TAG_LIMIT = 5
```

Limit the number of tags that can be added to (django-taggit) Tag model. Default setting is `None`, meaning no limit on tags.

Usage for images, documents and snippets

WAGTAIL_USAGE_COUNT_ENABLED

```
WAGTAIL_USAGE_COUNT_ENABLED = True
```

When enabled Wagtail shows where a particular image, document or snippet is being used on your site. This is disabled by default because it generates a query which may run slowly on sites with large numbers of pages.

A link will appear on the edit page (in the rightmost column) showing you how many times the item is used. Clicking this link takes you to the "Usage" page, which shows you where the snippet, document or image is used.

The link is also shown on the delete page, above the "Delete" button.

Note: The usage count only applies to direct (database) references. Using documents, images and snippets within StreamFields or rich text fields will not be taken into account.

Static files

`WAGTAILADMIN_STATIC_FILE_VERSION_STRINGS`

```
WAGTAILADMIN_STATIC_FILE_VERSION_STRINGS = False
```

Static file URLs within the Wagtail admin are given a version-specific query string of the form `?v=1a2b3c4d`, to prevent outdated cached copies of JavaScript and CSS files from persisting after a Wagtail upgrade. To disable these, set `WAGTAILADMIN_STATIC_FILE_VERSION_STRINGS` to `False`.

API

For full documentation on API configuration, including these settings, see [Wagtail API v2 Configuration Guide](#) documentation.

`WAGTAILAPI_BASE_URL`

```
WAGTAILAPI_BASE_URL = 'http://api.example.com/'
```

Required when using frontend cache invalidation, used to generate absolute URLs to document files and invalidating the cache.

`WAGTAILAPI_LIMIT_MAX`

```
WAGTAILAPI_LIMIT_MAX = 500
```

Default is 20, used to change the maximum number of results a user can request at a time, set to `None` for no limit.

`WAGTAILAPI_SEARCH_ENABLED`

```
WAGTAILAPI_SEARCH_ENABLED = False
```

Default is true, setting this to false will disable full text search on all endpoints.

`WAGTAILAPI_USE_FRONTENDCACHE`

```
WAGTAILAPI_USE_FRONTENDCACHE = True
```

Requires `wagtailfrontendcache` app to be installed, indicates the API should use the frontend cache.

Frontend cache

For full documentation on frontend cache invalidation, including these settings, see [Frontend cache invalidator](#).

WAGTAILFRONTENDCACHE

```
WAGTAILFRONTENDCACHE = {
    'varnish': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.HTTPBackend',
        'LOCATION': 'http://localhost:8000',
    },
}
```

See documentation linked above for full options available.

Note: WAGTAILFRONTENDCACHE_LOCATION is no longer the preferred way to set the cache location, instead set the LOCATION within the WAGTAILFRONTENDCACHE item.

WAGTAILFRONTENDCACHE_LANGUAGES

```
WAGTAILFRONTENDCACHE_LANGUAGES = [l[0] for l in settings.LANGUAGES]
```

Default is an empty list, must be a list of languages to also purge the urls for each language of a purging url. This setting needs `settings.USE_I18N` to be True to work.

Redirects

WAGTAIL_REDIRECTS_FILE_STORAGE

```
WAGTAIL_REDIRECTS_FILE_STORAGE = 'tmp_file'
```

By default the redirect importer keeps track of the uploaded file as a temp file, but on certain environments (load balanced/cloud environments), you cannot keep a shared file between environments. For those cases you can use the built-in cache to store the file instead.

```
WAGTAIL_REDIRECTS_FILE_STORAGE = 'cache'
```

Form builder

WAGTAILFORMS_HELP_TEXT_ALLOW_HTML

```
WAGTAILFORMS_HELP_TEXT_ALLOW_HTML = True
```

When true, HTML tags in form field help text will be rendered unescaped (default: False).

Warning: Enabling this option will allow editors to insert arbitrary HTML into the page, such as scripts that could allow the editor to acquire administrator privileges when another administrator views the page. Do not enable this setting unless your editors are fully trusted.

Workflow

`WAGTAIL_MODERATION_ENABLED`

```
WAGTAIL_MODERATION_ENABLED = True
```

Changes whether the Submit for Moderation button is displayed in the action menu.

`WAGTAIL_WORKFLOW_ENABLED`

```
WAGTAIL_WORKFLOW_ENABLED = False
```

Specifies whether moderation workflows are enabled (default: True). When disabled, editors will no longer be given the option to submit pages to a workflow, and the settings areas for admins to configure workflows and tasks will be unavailable.

`WAGTAIL_WORKFLOW_REQUIRE_REAPPROVAL_ON_EDIT`

```
WAGTAIL_WORKFLOW_REQUIRE_REAPPROVAL_ON_EDIT = True
```

Moderation workflows can be used in two modes. The first is to require that all tasks must approve a specific page revision for the workflow to complete. As a result, if edits are made to a page while it is in moderation, any approved tasks will need to be re-approved for the new revision before the workflow finishes. This is the default, `WAGTAIL_WORKFLOW_REQUIRE_REAPPROVAL_ON_EDIT = True`. The second mode does not require reapproval: if edits are made when tasks have already been approved, those tasks do not need to be reapproved. This is more suited to a hierarchical workflow system. To use workflows in this mode, set `WAGTAIL_WORKFLOW_REQUIRE_REAPPROVAL_ON_EDIT = False`.

`WAGTAIL_FINISH_WORKFLOW_ACTION`

```
WAGTAIL_FINISH_WORKFLOW_ACTION = 'wagtail.workflows.publish_workflow_state'
```

This sets the function to be called when a workflow completes successfully - by default, `wagtail.workflows.publish_workflow_state`, which publishes the page. The function must accept a `WorkflowState` object as its only positional argument.

WAGTAIL_WORKFLOW_CANCEL_ON_PUBLISH

```
WAGTAIL_WORKFLOW_CANCEL_ON_PUBLISH = True
```

This determines whether publishing a page with an ongoing workflow will cancel the workflow (if true) or leave the workflow unaffected (false). Disabling this could be useful if your site has long, multi-step workflows, and you want to be able to publish urgent page updates while the workflow continues to provide less urgent feedback.

1.5.8 The project template

```
mysite/
    home/
        migrations/
            __init__.py
            0001_initial.py
            0002_create_homepage.py
        templates/
            home/
                home_page.html
            __init__.py
            models.py
    search/
        templates/
            search/
                search.html
        __init__.py
        views.py
mysite/
    settings/
        __init__.py
        base.py
        dev.py
        production.py
    static/
        css/
            mysite.css
        js/
            mysite.js
    templates/
        404.html
        500.html
        base.html
        __init__.py
        urls.py
        wsgi.py
Dockerfile
manage.py
requirements.txt
```

The “home” app

Location: `/mysite/home/`

This app is here to help get you started quicker by providing a `HomePage` model with migrations to create one when you first set up your app.

Default templates and static files

Location: `/mysite/mysite/templates/` and `/mysite/mysite/static/`

The templates directory contains `base.html`, `404.html` and `500.html`. These files are very commonly needed on Wagtail sites so they have been added into the template.

The static directory contains an empty JavaScript and CSS file.

Django settings

Location: `/mysite/mysite/settings/`

The Django settings files are split up into `base.py`, `dev.py`, `production.py` and `local.py`.

- `base.py` This file is for global settings that will be used in both development and production. Aim to keep most of your configuration in this file.
- `dev.py` This file is for settings that will only be used by developers. For example: `DEBUG = True`
- `production.py` This file is for settings that will only run on a production server. For example: `DEBUG = False`
- `local.py` This file is used for settings local to a particular machine. This file should never be tracked by a version control system.

Note: 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.

Dockerfile

Location: `/mysite/Dockerfile`

Contains configuration for building and deploying the site as a [Docker](#) container. To build and use the Docker image for your project, run:

```
docker build -t mysite .
docker run -p 8000:8000 mysite
```

1.5.9 Jinja2 template support

Wagtail supports Jinja2 templating for all front end features. More information on each of the template tags below can be found in the [Writing templates](#) documentation.

Configuring Django

Django needs to be configured to support Jinja2 templates. As the Wagtail admin is written using standard Django templates, Django has to be configured to use **both** templating engines. Add the Jinja2 template backend configuration to the `TEMPLATES` setting for your app as shown here:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.djangoproject.DjangoTemplates",
        # ... the rest of the existing Django template configuration ...
    },
    {
        'BACKEND': 'django.template.backends.jinja2.Jinja2',
        'APP_DIRS': True,
        'OPTIONS': {
            'extensions': [
                'wagtail.jinja2tags.core',
                'wagtail.admin.jinja2tags.userbar',
                'wagtail.images.jinja2tags.images',
            ],
        },
    },
]
```

Jinja templates must be placed in a `jinja2/` directory in your app. For example, the standard template location for an `EventPage` model in an `events` app would be `events/jinja2/events/event_page.html`.

By default, the Jinja environment does not have any Django functions or filters. The Django documentation has more information on `django.template.backends.jinja2.Jinja2` (configuring Jinja for Django).

`self` in templates

In Django templates, `self` can be used to refer to the current page, stream block, or field panel. In Jinja, `self` is reserved for internal use. When writing Jinja templates, use `page` to refer to pages, `value` for stream blocks, and `field_panel` for field panels.

Template tags, functions & filters

`pageurl()`

Generate a URL for a Page instance:

```
<a href="{{ pageurl(page.more_information) }}>More information</a>
```

See `pageurl` for more information

slugurl()

Generate a URL for a Page with a slug:

```
<a href="{{ slugurl('about') }}>About us</a>
```

See [slugurl](#) for more information

image()

Resize an image, and print an tag:

```
{# Print an image tag #}
{{ image(page.header_image, "fill-1024x200", class="header-image") }}

{# Resize an image #}
{% set background=image(page.background_image, "max-1024x1024") %}
<div class="wrapper" style="background-image: url('{{ background.url }}');">
```

See [How to use images in templates](#) for more information

|richtext

Transform Wagtail's internal HTML representation, expanding internal references to pages and images.

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

See [Rich text \(filter\)](#) for more information

wagtail_site

Returns the Site object corresponding to the current request.

```
{{ wagtail_site().site_name }}
```

See [wagtail_site](#) for more information

wagtailuserbar()

Output the Wagtail contextual flyout menu for editing pages from the front end

```
{{ wagtailuserbar() }}
```

See [Wagtail User Bar](#) for more information

```
{% include_block %}
```

Output the HTML representation for the stream content as a whole, as well as for each individual block.

Allows to pass template context (by default) to the StreamField template.

```
{% include_block page.body %}
{% include_block page.body with context %} {# The same as the previous #}
{% include_block page.body without context %}
```

See [StreamField template rendering](#) for more information.

Note: The `{% include_block %}` tag is designed to closely follow the syntax and behaviour of Jinja's `{% include %}`, so it does not implement the Django version's feature of only passing specified variables into the context.

1.5.10 Panel API

Panel

```
class wagtail.admin.panels.Panel(heading='', classname='', help_text='', base_form_class=None, icon='')
```

Defines part (or all) of the edit form interface for pages and other models within the Wagtail admin. Each model has an associated panel definition, consisting of a nested structure of Panel objects - this provides methods for obtaining a ModelForm subclass, with the field list and other parameters collated from all panels in the structure. It then handles rendering that form as HTML.

bind_to_model(model)

Create a clone of this panel definition with a `model` attribute pointing to the linked model class.

on_model_bound()

Called after the panel has been associated with a model class and the `self.model` attribute is available; panels can override this method to perform additional initialisation related to the model.

clone()

Create a clone of this panel definition. By default, constructs a new instance, passing the keyword arguments returned by `clone_kwargs`.

clone_kwargs()

Return a dictionary of keyword arguments that can be used to create a clone of this panel definition.

get_form_options()

Return a dictionary of attributes such as 'fields', 'formsets' and 'widgets' which should be incorporated into the form class definition to generate a form that this panel can use. This will only be called after binding to a model (i.e. `self.model` is available).

get_form_class()

Construct a form class that has all the fields and formsets named in the children of this edit handler.

get_bound_panel(instance=None, request=None, form=None, prefix='panel')

Return a BoundPanel instance that can be rendered onto the template as a component. By default, this creates an instance of the panel class's inner BoundPanel class, which must inherit from `Panel.BoundPanel`.

property clean_name

A name for this panel, consisting only of ASCII alphanumerics and underscores, suitable for use in identifiers. Usually generated from the panel heading. Note that this is not guaranteed to be unique or non-empty; anything making use of this and requiring uniqueness should validate and modify the return value as needed.

BoundPanel

class wagtail.admin.panels.Panel.BoundPanel(panel, instance, request, form, prefix)

A template component for a panel that has been associated with a model instance, form, and request.

In addition to the standard template component functionality (see [Creating components](#)), this provides the following attributes and methods:

panel

The panel definition corresponding to this bound panel

instance

The model instance associated with this panel

request

The request object associated with this panel

form

The form object associated with this panel

prefix

A unique prefix for this panel, for use in HTML IDs

id_for_label()

Returns an HTML ID to be used as the target for any label referencing this panel.

is_shown()

Whether this panel should be rendered; if false, it is skipped in the template output.

1.5.11 Viewsets

Viewsets are Wagtail's mechanism for defining a group of related admin views with shared properties, as a single unit. See [Generic views](#).

ViewSet

class wagtail.admin.viewsets.base.ViewSet(name, **kwargs)

Defines a viewset to be registered with the Wagtail admin.

Parameters

- **name** – A name for this viewset, used as the URL namespace.
- **url_prefix** – A URL path element, given as a string, that the URLs for this viewset will be found under. Defaults to the same as `name`.

All other keyword arguments will be set as attributes on the instance.

on_register()

Called when the viewset is registered; subclasses can override this to perform additional setup.

get_urlpatterns()

Returns a set of URL routes to be registered with the Wagtail admin.

get_url_name(view_name)

Returns the namespaced URL name for the given view.

ModelViewSet**class wagtail.admin.viewsets.model.ModelViewSet(name, **kwargs)**

A viewset to allow listing, creating, editing and deleting model instances.

model

Required; the model class that this viewset will work with.

form_fields

A list of model field names that should be included in the create / edit forms.

exclude_form_fields

Used in place of `form_fields` to indicate that all of the model's fields except the ones listed here should appear in the create / edit forms. Either `form_fields` or `exclude_form_fields` must be supplied (unless `get_form_class` is being overridden).

get_form_class(for_update=False)

Returns the form class to use for the create / edit forms.

icon = ''

The icon to use to represent the model within this viewset.

index_view_class = <class 'wagtail.admin.views.generic.models.IndexView'>

The view class to use for the index view; must be a subclass of `wagtail.admin.views.generic.IndexView`.

add_view_class = <class 'wagtail.admin.views.generic.models.CreateView'>

The view class to use for the create view; must be a subclass of `wagtail.admin.views.generic.CreateView`.

edit_view_class = <class 'wagtail.admin.views.generic.models.EditView'>

The view class to use for the edit view; must be a subclass of `wagtail.admin.views.generic.EditView`.

delete_view_class = <class 'wagtail.admin.views.generic.models.DeleteView'>

The view class to use for the delete view; must be a subclass of `wagtail.admin.views.generic.DeleteView`.

ChooserViewSet**class wagtail.admin.viewsets.chooser.ChooserViewSet(*args, **kwargs)**

A viewset that creates a chooser modal interface for choosing model instances.

model

Required; the model class that this viewset will work with.

```
icon = 'snippet'  
The icon to use in the header of the chooser modal, and on the chooser widget  
choose_one_text = 'Choose'  
Label for the 'choose' button in the chooser widget when choosing an initial item  
page_title = None  
Title text for the chooser modal (defaults to the same as choose_one_text)  
choose_another_text = 'Choose another'  
Label for the 'choose' button in the chooser widget, when an item has already been chosen  
edit_item_text = 'Edit'  
Label for the 'edit' button in the chooser widget  
per_page = 10  
Number of results to show per page  
choose_view_class = <class 'wagtail.admin.views.generic.chooser.ChooseView'>  
The view class to use for the overall chooser modal; must be a subclass of wagtail.admin.views.generic.chooser.ChooseView.  
choose_results_view_class = <class 'wagtail.admin.views.generic.chooser.ChooseResultsView'>  
The view class used to render just the results panel within the chooser modal; must be a subclass of wagtail.admin.views.generic.chooser.ChooseResultsView.  
chosen_view_class = <class 'wagtail.admin.views.generic.chooser.ChosenView'>  
The view class used after an item has been chosen; must be a subclass of wagtail.admin.views.generic.chooser.ChosenView.  
create_view_class = <class 'wagtail.admin.views.generic.chooser.CreateView'>  
The view class used to handle submissions of the 'create' form; must be a subclass of wagtail.admin.views.generic.chooser.CreateView.  
base_widget_class = <class 'wagtail.admin.widgets.chooser.BaseChooser'>  
The base Widget class that the chooser widget will be derived from.  
widget_class  
Returns the form widget class for this chooser.  
widget_teletype_adapter_class = None  
The adapter class used to map the widget class to its JavaScript implementation - see Form widget client-side API. Only required if the chooser uses custom JavaScript code.  
register_widget = True  
Defaults to True; if False, the chooser widget will not automatically be registered for use in admin forms.  
base_block_class = <class 'wagtail.blocks.field_block.ChooserBlock'>  
The base ChooserBlock class that the StreamField chooser block will be derived from.  
get_block_class(name=None, module_path=None)  
Returns a StreamField ChooserBlock class using this chooser.
```

Parameters

- **name** – Name to give to the class; defaults to the model name with “ChooserBlock” appended

- **module_path** – The dotted path of the module where the class can be imported from; used when deconstructing the block definition for migration files.

creation_form_class = None

Form class to use for the form in the “Create” tab of the modal.

form_fields = None

List of model fields that should be included in the creation form, if creation_form_class is not specified.

exclude_form_fields = None

List of model fields that should be excluded from the creation form, if creation_form_class. If none of creation_form_class, form_fields or exclude_form_fields are specified, the “Create” tab will be omitted.

create_action_label = 'Create'

Label for the submit button on the ‘create’ form

create_action_clicked_label = None

Alternative text to display on the submit button after it has been clicked

creation_tab_label = None

Label for the ‘create’ tab in the chooser modal (defaults to the same as create_action_label)

search_tab_label = 'Search'

Label for the ‘search’ tab in the chooser modal

1.6 Support

If you have any problems or questions about working with Wagtail, you are invited to visit any of the following support channels, where volunteer members of the Wagtail community will be happy to assist.

Please respect the time and effort of volunteers, by not asking the same question in multiple places. At best, you’ll be spamming the same set of people each time; at worst, you’ll waste the effort of volunteers who spend time answering a question unaware that it has already been answered elsewhere. If you absolutely must ask a question on multiple forums, post it on Stack Overflow first and include the Stack Overflow link in your subsequent posts.

1.6.1 Stack Overflow

Stack Overflow is the best place to find answers to your questions about working with Wagtail - there is an active community of Wagtail users and developers responding to questions there. When posting questions, please read Stack Overflow’s advice on [how to ask questions](#) and remember to tag your question with “wagtail”.

1.6.2 Mailing list

For topics and discussions that do not fit Stack Overflow’s question-and-answer format, there is a Wagtail Support mailing list at groups.google.com/d/forum/wagtail.

1.6.3 Slack

The Wagtail Slack workspace is open to all users and developers of Wagtail. To join, head to: <https://wagtail.org/slack/>. Please use the **#support** channel for support questions. Support is provided by members of the Wagtail community on a voluntary basis, and we cannot guarantee that questions will be answered quickly (or at all). If you want to see this resource succeed, please consider sticking around to help out! Also, please keep in mind that many of Wagtail's core and expert developers prefer to handle support queries on a non-realtime basis through Stack Overflow, and questions asked there may well get a better response.

1.6.4 GitHub discussions

Our [Github discussion boards](#) are open for sharing ideas and plans for the Wagtail project.

1.6.5 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/wagtail/wagtail/issues. If your bug or suggestion isn't there, raise a new issue, providing as much relevant context as possible.

If your bug report is a security issue, **do not** report it with an issue. Please read our guide to [reporting security issues](#).

1.6.6 Torchbox

Finally, if you have a query which isn't relevant for any of the above forums, feel free to contact the Wagtail team at Torchbox directly, on hello@wagtail.org or @wagtailcms.

1.7 Using Wagtail: an Editor's guide

This section of the documentation is written for the users of a Wagtail-powered site. That is, the content editors, moderators and administrators who will be running things on a day-to-day basis.

1.7.1 Introduction

Wagtail is an 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
- set up users, and provide them with specific roles to create a publishing workflow
- upload, edit and include images and documents
- ... and more!

1.7.2 Getting started

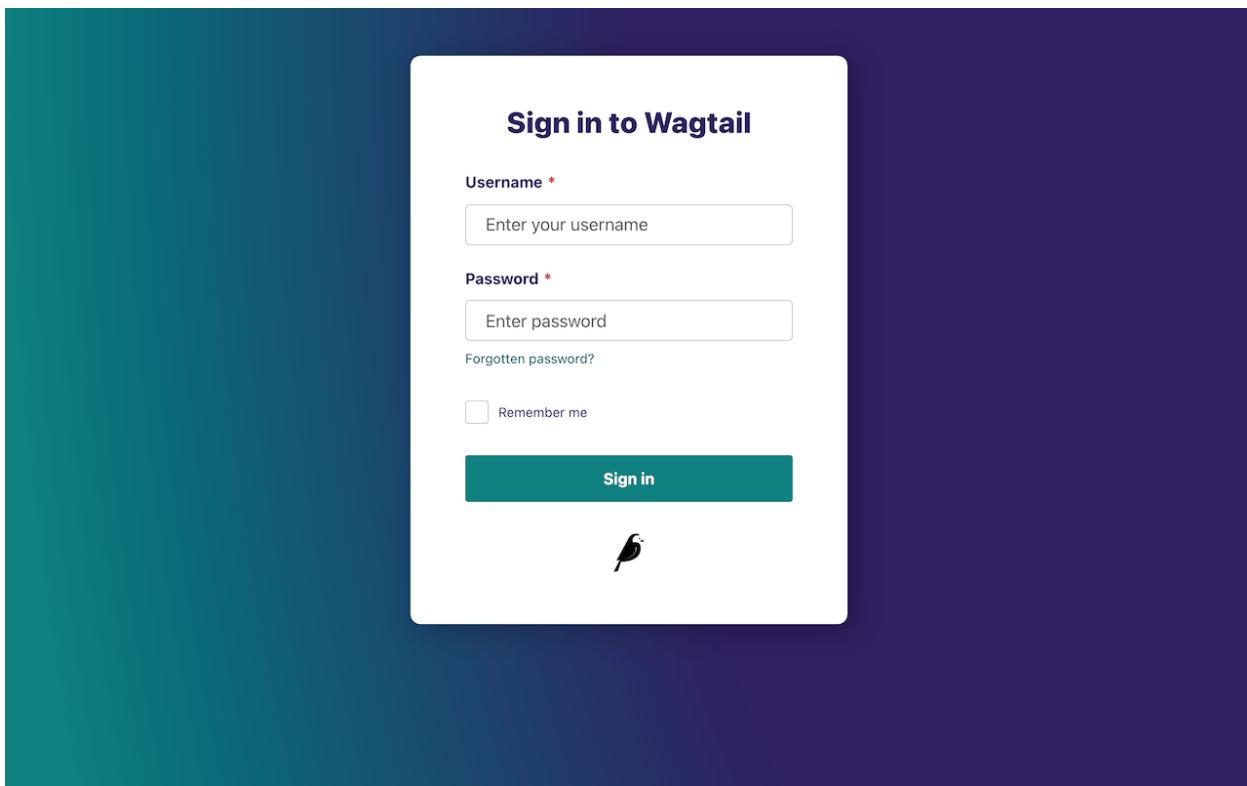
The Wagtail demo site

The examples in this document are based on [our bakery demo site](#). However, the instructions are general enough as to be applicable to any Wagtail site.

For the purposes of this documentation we will be using the URL, **www.example.com**, to represent the root (homepage) 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 (for example www.example.com/admin).
- Enter your username and password and click **Sign in**.



1.7.3 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)
- Any pages that you've locked (if your administrator has enabled *author-specific locking*)
- Your most recently edited pages

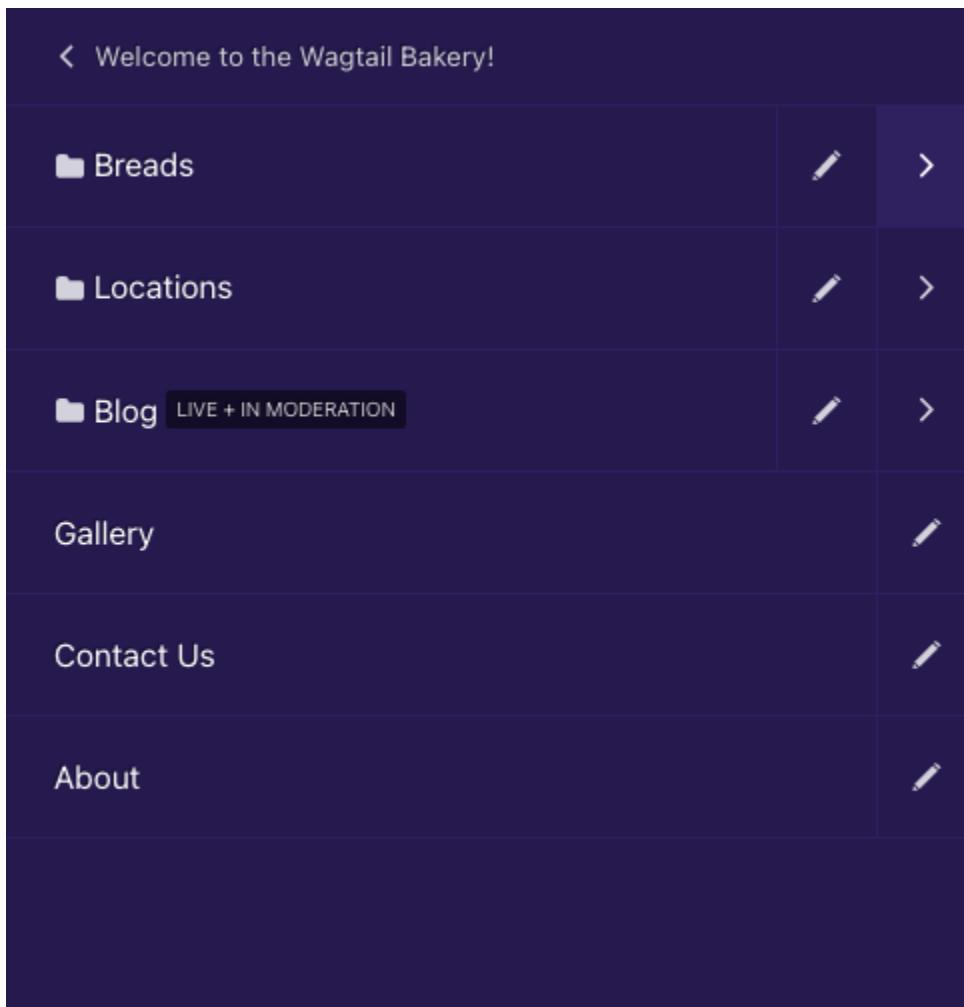
You can return to the Dashboard at any time by clicking the Wagtail logo in the top-left of the screen.

The screenshot shows the Wagtail CMS Dashboard with a dark purple header and sidebar. The sidebar on the left includes links for Search, Pages, Bread Categories, Bakery Misc, Images, Documents, Snippets, Forms, Reports, and Settings. The main content area features a welcome message and summary statistics: 30 Pages, 37 Images, and 0 Documents. Below this are several sections: 'Your pages in a workflow' (Blog page pending moderator approval), 'Awaiting your review' (Blog page pending moderator approval), 'Your most recent edits' (Bread and Circuses page live, Blog page live + in moderation), and 'Your locked pages' (Gallery page locked). A navigation bar at the bottom includes links for Admin, Logout, and Help.

- Clicking the logo returns you to your Dashboard.
- The stats at the top of the page describe the total amount of content on the CMS (just for fun!).
- The *Your pages in a workflow* panel shows you any pages in moderation that you own or submitted for moderation yourself, along with which moderation tasks they are awaiting, and how long they've been on that task.
- The *Awaiting your review* panel will only be displayed if you are able to perform moderation actions.
 - Clicking the name of a page will take you to the ‘Edit page’ interface for this page.

- Clicking approve or request changes will either progress the page to the next task in the moderation workflow (or publish if it's the final stage) or mark the page as needing changes and return it to normal editing. An email will be sent to the creator of the page giving the result of the overall workflow when it completes.
- The status indicator circles show the progress through the workflow: a tick for a completed task, or an empty circle for an incomplete one. Tooltips provide more details on the status of each.
- The time this page has been waiting for review on this task is also shown to the right.
- The *Your locked pages* panel shows the pages you've locked so that only you can edit them.
 - The date you locked the page is displayed to the right.
 - Clicking the name of a page will take you to the 'Edit page' interface for this page.
- The *Your most recent edits* table displays the five pages that you most recently edited.
 - The date that you edited the page is displayed. Hover your mouse over the date for a more exact time/date.
 - The current status of the page is displayed. A page will have one of a number of statuses:
 - * Live: Published and accessible to website visitors
 - * Draft: Not live on the website
 - * In Moderation: In the middle of a moderation workflow
 - * Scheduled: Not live, but has a publication date set
 - * Expired: Not live - this page was unpublished as it had an expiry date set which has now passed
 - * Live + Draft: A version of the page is live, but a newer version is in draft mode.
 - * Live + (another status): A version of the page is live, but a newer version has another status.

The Explorer menu



- Click the Pages button in the sidebar to open the explorer. This allows you to navigate through the sections of the site.
- Clicking the name of a page will take you to the Explorer page for that section.
- Clicking the edit icon for a page will take you to its edit screen.
- Clicking the arrow takes you to the sub-section.
- Clicking the section title takes you back to where you were.

Using search

The screenshot shows the Wagtail search interface. At the top, there is a search bar with the text "Search" and a magnifying glass icon, followed by a search input field containing "bread". Below the search bar, a message says "There are 6 matching pages". A navigation bar includes links for "Other searches", "Pages", "Images", "Documents", and "Users". Under "Page types", there are filters for "All (6)", "Blog page (4)", and "Bread page (2)". The main area displays a table of search results:

	Title	Parent	Updated	Type	Status
<input type="checkbox"/>	Bagel ENGLISH	Breads	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Black bread ENGLISH	Breads	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Bread and Circuses ENGLISH	Blog	24 minutes ago	Blog page	LIVE
<input type="checkbox"/>	The Great Icelandic Baking Show ENGLISH	Blog	24 minutes ago	Blog page	LIVE
<input type="checkbox"/>	The Joy of (Baking) Soda ENGLISH	Blog	24 minutes ago	Blog page	LIVE

- 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 of that page is.

The Explorer page

The Explorer page allows you to view 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 screenshot shows the Wagtail Explorer page for the 'Welcome to the Wagtail Bakery!' section. The page lists the following child pages:

Title	Last Updated	Type	Status	Action
Blog	24 minutes ago	Blog index page	LIVE + IN MODERATION	>
Breads	24 minutes ago	Breads index page	LIVE	>
About	24 minutes ago	Standard page	LIVE	
Locations	24 minutes ago	Locations index page	LIVE	>
Gallery	24 minutes ago	Gallery page	LIVE	
Contact Us	24 minutes ago	Form page	LIVE	

At the bottom of the page, it says "Page 1 of 1."

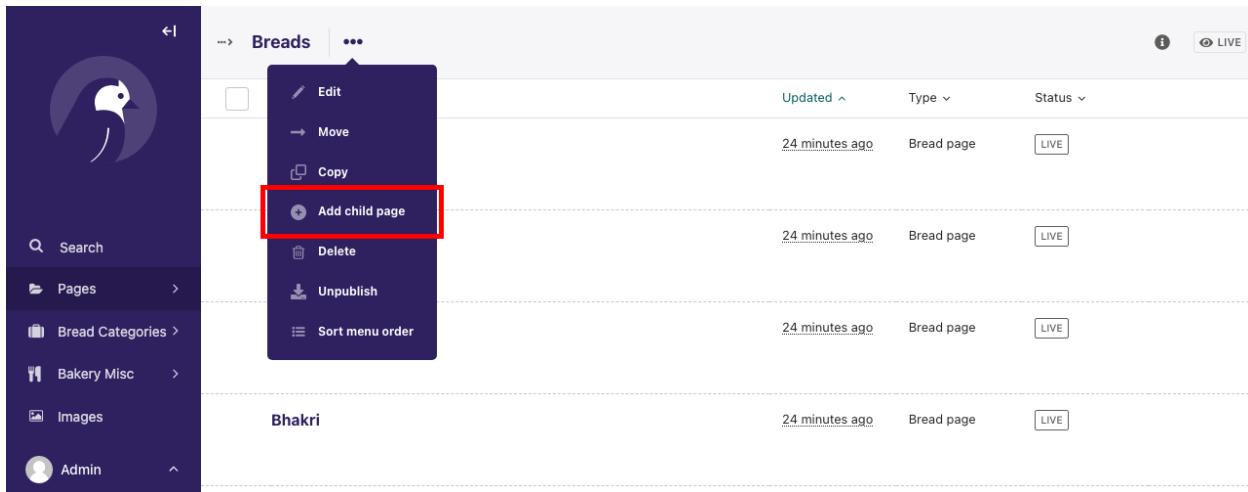
- The name of the section you are looking at is displayed at the top. Each section is also itself a page (in this case the homepage). Clicking the title of the section takes you to the Edit screen for the section page.
- As the heading suggests, below are the child pages of the section. Clicking the titles of each child page will take you to its Edit screen.
- Clicking the arrows will display a further level of child pages.

The screenshot shows the Wagtail Explorer page for the 'Breads' section. The breadcrumb trail shows the path: Root > Welcome to the Wagtail Bakery! > Breads. The page lists the following child pages:

Title	Last Updated	Type	Status	Action
Blog	24 minutes ago	Blog index page	LIVE + IN MODERATION	>
Breads	24 minutes ago	Breads index page	LIVE	>
About	24 minutes ago	Standard page	LIVE	
Locations	24 minutes ago	Locations index page	LIVE	>
Gallery	24 minutes ago	Gallery page	LIVE	
Contact Us	24 minutes ago	Form page	LIVE	

At the bottom of the page, it says "Page 1 of 1."

- 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.



- Clicking on the Actions dropdown will show a list of actions for the parent page, like Move, Copy, Delete, Unpublish, History. There will also be a Sort menu order button that will take you to the ordering page (see below)
- To add further child pages press the Add child page button. You can view the parent page on the live site by pressing the View live button. The Edit button will take you to the page edit screen.

This screenshot shows the Wagtail admin interface for the 'Breads' page, similar to the previous one but with more checkboxes selected. The 'Anpan' and 'Bazin' items have checkboxes checked, indicated by a green checkmark icon. A bulk action bar at the bottom of the list provides options: 'Move', 'Delete', 'Publish', and 'Unpublish', along with a summary '2 pages selected'.

- There is a checkbox on the left of each page row, hovering over the row will make it visible. There is also a checkbox at the top left of the header, used to select or deselect all other checkboxes.
- Selecting at least one checkbox will popup an action bar at the bottom, which will list all the available bulk actions for pages.
- Clicking on any action will take you to a separate view with all the selected pages, for confirmation
- Similar buttons are available for each child page. These are made visible on hover.

Reordering pages

The screenshot shows a list of bread pages under the 'Breads' section. The pages are listed vertically with their titles, last updated time, type, and status. The 'Status' column contains a 'LIVE' button for each page. At the top left of the list, there is a 'Sort' button with a dropdown arrow and a 'Title' dropdown menu.

Page	Last Updated	Type	Status
Anadama	24 minutes ago	Bread page	LIVE
Anpan	24 minutes ago	Bread page	LIVE
Appam	24 minutes ago	Bread page	LIVE
Arepá	24 minutes ago	Bread page	LIVE

- Clicking the “Sort menu order” button in the More dropdown will take you to a page with the reordering handles. This allows you to reorder the way that content displays in the main menu of your website.
- Reorder by dragging the pages by the handles on the far left (the icon made up of 6 dots).
- Your new order will be automatically saved each time you drag and drop an item.

1.7.4 Creating new pages

The screenshot shows a context menu for a bread page titled 'Bhakri'. The menu includes options like Edit, Move, Copy, Add child page (which is highlighted with a red box), Delete, Unpublish, and Sort menu order.

Create new pages by clicking the **Add child page** button. This creates a child page of the section you are currently in. In this case a child page of the ‘Breads’ page.

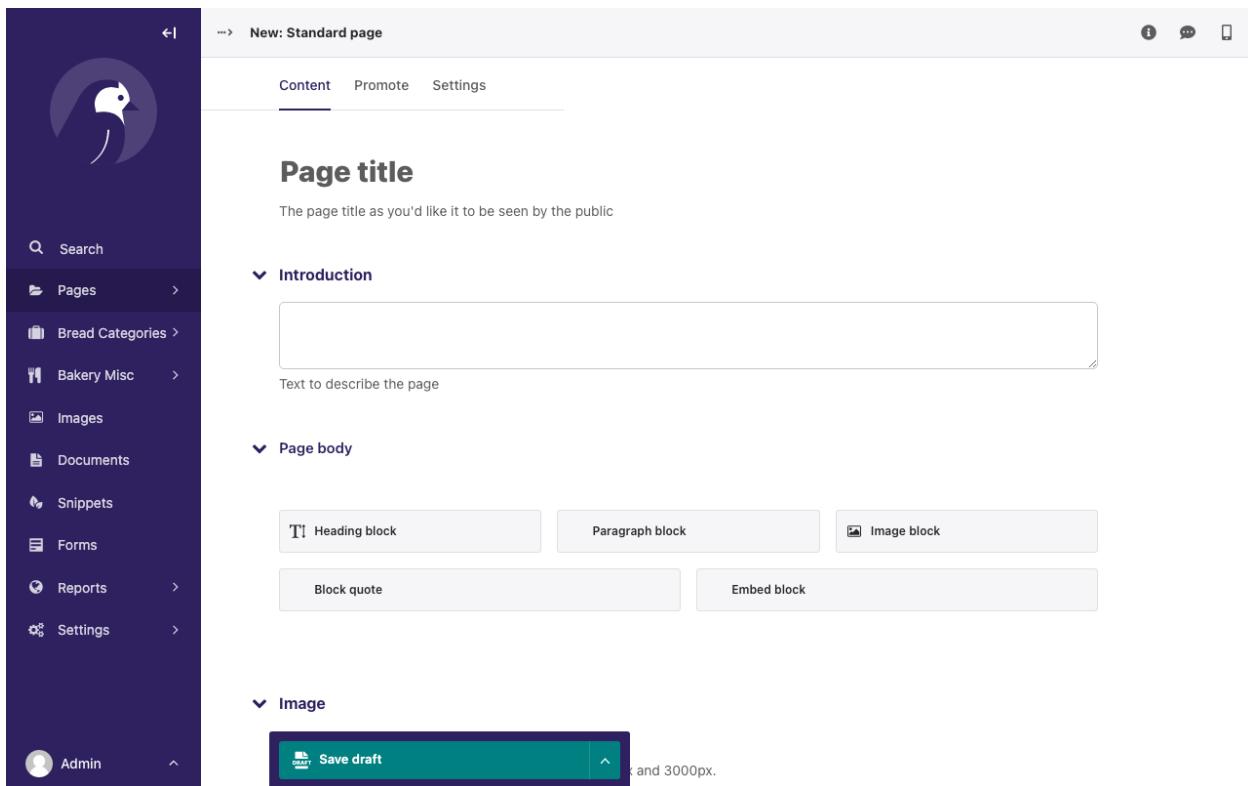
Selecting a page type

Create a page in Welcome to the Wagtail Bakery!

Choose which type of page you'd like to create.

 Blog index page	Pages using Blog index page
 Breads index page	Pages using Breads index page
 Form page	Pages using Form page
 Gallery page	Pages using Gallery page
 Home page	Pages using Home page
 Locations index page	Pages using Locations index page
 Standard page	Pages using Standard page

- On the left of the page chooser screen are listed all the types of pages that you can create. Clicking the page type name will take you to the Create new page screen for that page type (see below).
- Clicking the *Pages using ... Page* links on the right will display all the pages that exist on the website of this type. This is to help you judge what type of page you will need to complete your task.



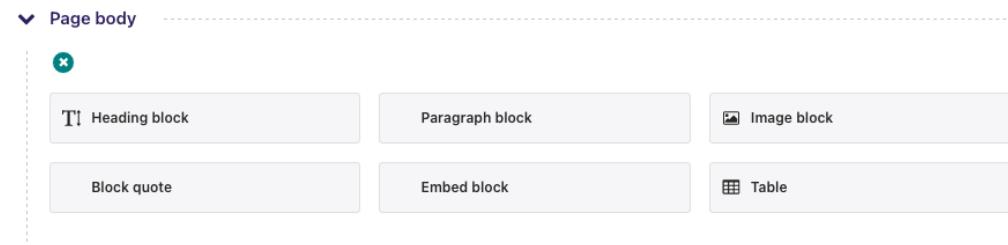
- Once you've selected a page type you will be presented with a blank New page screen.
- Click into the areas below each field's heading to start entering content.

Creating page body content

Wagtail supports a number of basic fields for creating content, as well as our unique StreamField feature which allows you to construct complex layouts by combining these basic fields in any order.

StreamField

StreamField allows you to create complex layouts of content on a page by combining a number of different arrangements of content, ‘blocks’, in any order.



When you first edit a page, you will be presented with the empty StreamField area, with the option to choose one of several block types. The block types on your website may be different from the screenshot here, but the principles are the same.

Click the block type, and the options will disappear, revealing the entry field for that block.

Depending on the block you chose, the field will display differently, and there might even be more than one field! There are a few common field types though that we will talk about here.

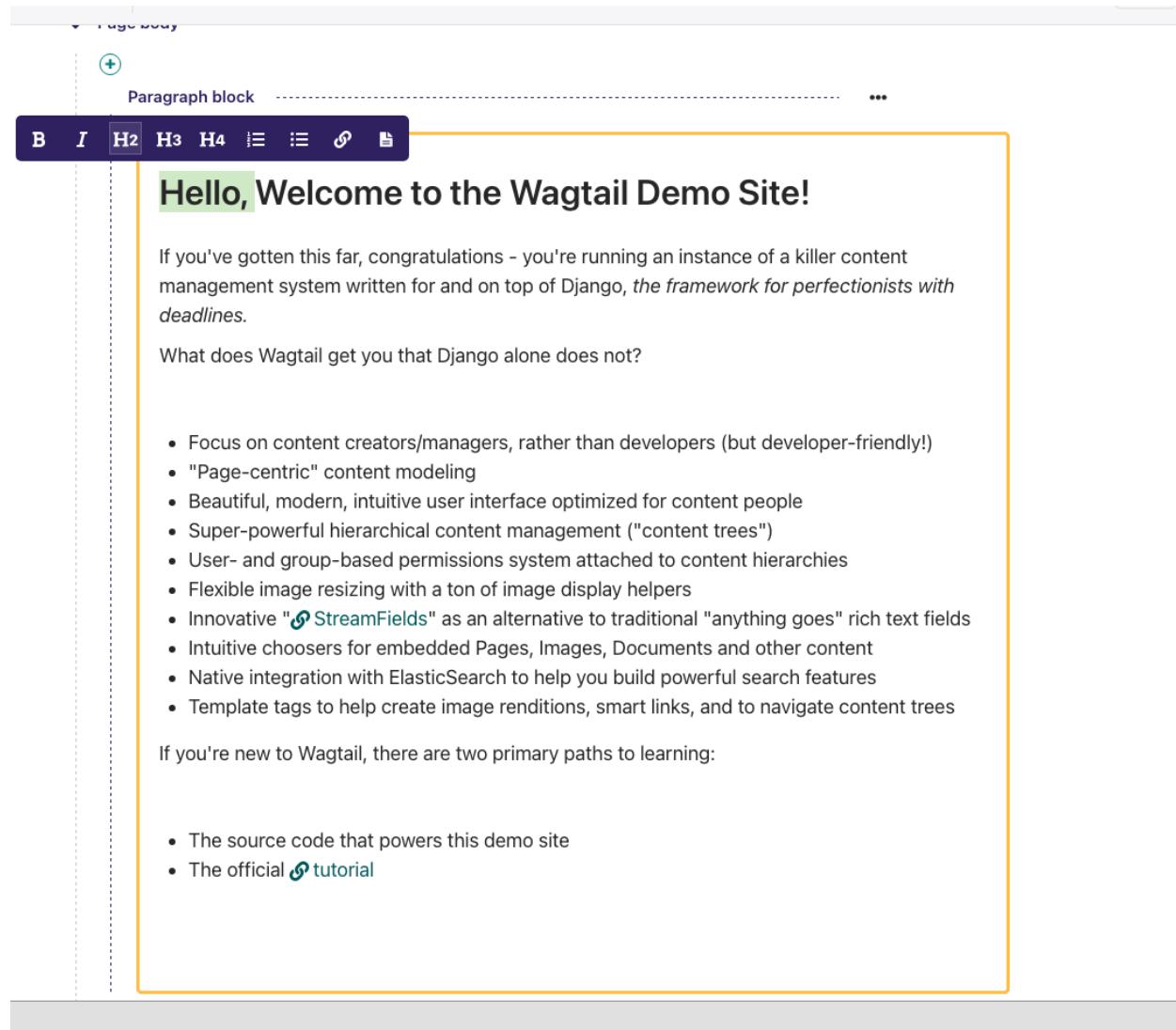
- Basic text field
- Rich text field
- Image field

Basic text field

Basic text fields have no formatting options. How these display will be determined by the style of the page in which they are being inserted. Just click into the field and type!

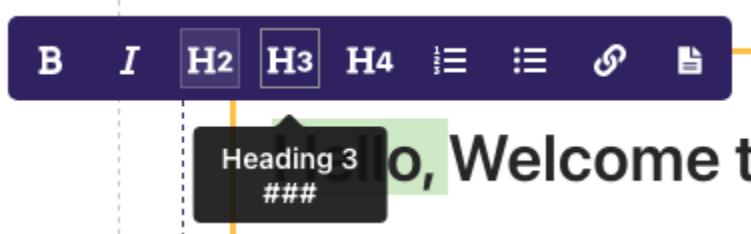
Rich text fields

Most of the time though, you need formatting options to create beautiful looking pages. Wagtail provides “rich text” fields, which have formatting options similar to those of word processors.



The screenshot shows the Wagtail rich text editor interface. At the top, there's a toolbar with icons for bold (B), italic (I), heading 2 (H2), heading 3 (H3), heading 4 (H4), and alignment tools. Below the toolbar, a heading 2 block is selected, indicated by a green background and the text "Hello, Welcome to the Wagtail Demo Site!". The text is bolded. To the left of the heading, there's a vertical dashed line, and to the right, a yellow border surrounds the content area. Below the heading, there's a paragraph of text: "If you've gotten this far, congratulations – you're running an instance of a killer content management system written for and on top of Django, *the framework for perfectionists with deadlines*. What does Wagtail get you that Django alone does not?". A bulleted list follows, detailing Wagtail's features: Focus on content creators/managers, rather than developers (but developer-friendly!), "Page-centric" content modeling, Beautiful, modern, intuitive user interface optimized for content people, Super-powerful hierarchical content management ("content trees"), User- and group-based permissions system attached to content hierarchies, Flexible image resizing with a ton of image display helpers, Innovative "StreamFields" as an alternative to traditional "anything goes" rich text fields, Intuitive choosers for embedded Pages, Images, Documents and other content, Native integration with ElasticSearch to help you build powerful search features, and Template tags to help create image renditions, smart links, and to navigate content trees. At the bottom, it says "If you're new to Wagtail, there are two primary paths to learning:" followed by another bulleted list: The source code that powers this demo site and The official [tutorial](#).

Those fields present 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. If you want to know more about a specific tool, hover your mouse on the corresponding button so the tooltip appears:



This tooltip shows a longer description of the tool, and displays its keyboard shortcut if there is one. If the keyboard shortcut does not start with CTRL or , it's a [Markdown](#) shortcut to type directly in the editor:

That's the gist of it! If you want more information about the editor, please have a look at its dedicated [user guide](#). It also contains a list of all of the available keyboard shortcuts, and some tricks and gotchas.

Adding further blocks in StreamField

A screenshot of the Wagtail StreamField block selection interface. At the top, there is a dropdown menu labeled "Body" with a red asterisk. Below it is a plus sign icon and a "Paragraph block" button. To the right is a three-dot menu icon. The main content area contains a text box with placeholder text: "We'd love to hear from you! Drop us a line to let us know what you liked or didn't like about your recent store visit, or if you have comments or questions about this site." Below this text box is another text block with placeholder text: "This form is for demo purposes only - email goes nowhere. Developers: Edit the 'To Address' field in the Form object in the Wagtail admin, and see the Email notes in the project readme to make your instance send live email." At the bottom, there is a row of block selection buttons: "Heading block" (with a red asterisk), "Paragraph block", "Image block", "Block quote", and "Embed block".

- To add new blocks, click the '+' icons above or below the existing blocks.
- You'll then be presented once again with the different blocks from which you may choose.
- You can cancel the addition of a new block by clicking the cross at the top of the block selection interface.
- You can duplicate a block (including the content) using the button towards the top right corner of an existing block.

Reordering and deleting content in StreamField

Paragraph block

We'd love to hear from you! Drop us a line to let us know what you liked or didn't like about your recent store visit, or if you have comments or questions about this site.

This form is for demo purposes only - email goes nowhere. Developers: Edit the "To Address" field in the Form object in the Wagtail admin, and see the Email notes in the project readme to make your instance send live email.

- Click the arrows on the right-hand side of each block to move blocks up and down in the StreamField order of content.
- The blocks will be displayed in the front-end in the order that they are placed in this interface.
- Click the rubbish bin on the far right to delete a field

Warning: Once a StreamField field is deleted it cannot be retrieved if the page has not been saved. Save your pages regularly so that if you accidentally delete a field you can reload the page to undo your latest edit.

Inserting images in a page

There will obviously be many instances in which you will want to add images to a page. There are two main ways to add images to pages, either via a specific image chooser field, or via the rich text field image button. Which of these you use will be dependent on the individual setup of your site.

Inserting images using the image chooser field

Often a specific image field will be used for a main image on a page, or for an image to be used when sharing the page on social media. For the standard page on the bakerydemo site, the former is used.

▼ **Image**

Choose an image

Landscape mode only; horizontal width between 1000px and 3000px.

- You insert an image by clicking the *Choose an image* button.

Choosing an image to insert

You have two options when selecting an image to insert:

1. Selecting an image from the existing image library, or...
2. Uploading a new image to the CMS

When you click the *Choose an image* button you will be presented with a pop-up with two tabs at the top. The first, *Search*, allows you to search and select from the library. The second, *Upload*, allows you to upload a new image.

Choosing an image from the image library

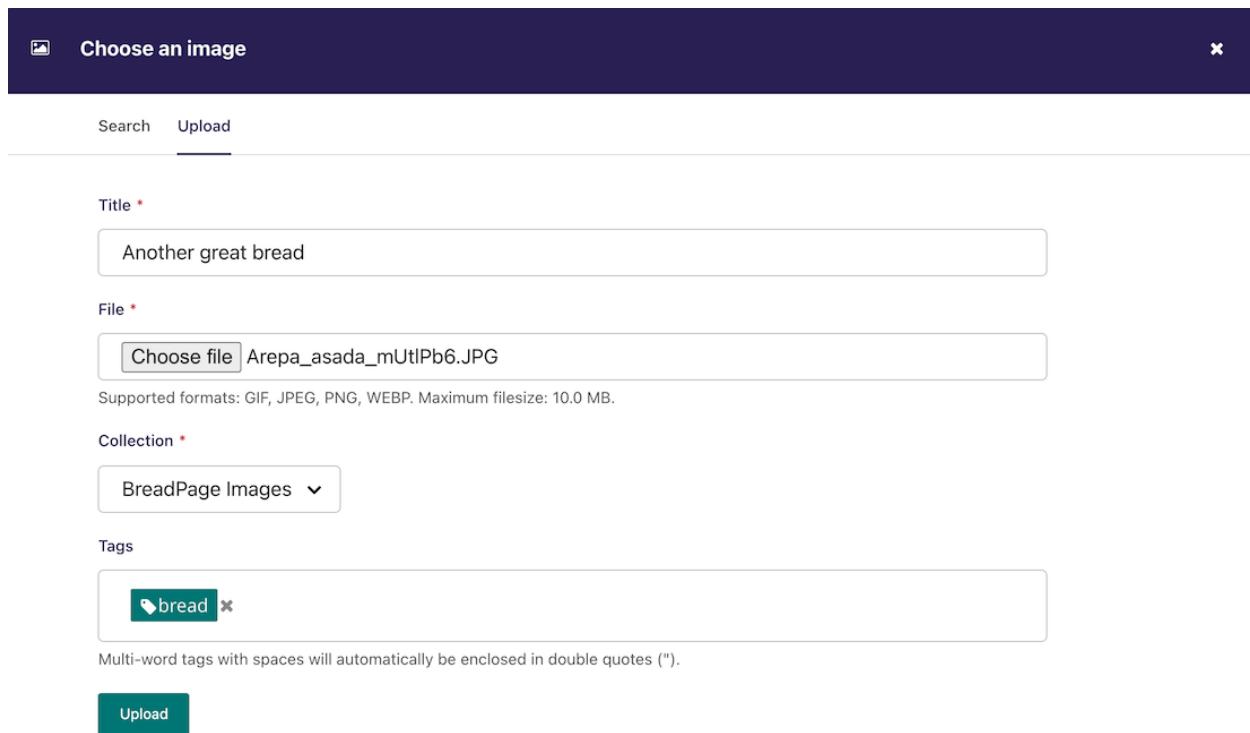
The image below demonstrates finding and inserting an image that is already present in the CMS image library.

The screenshot shows the 'Choose an image' modal window. At the top, there's a dark header bar with the title 'Choose an image' on the left and a close button 'x' on the right. Below the header, there are two tabs: 'Search' (which is active) and 'Upload'. The main content area contains several search filters and a grid of image thumbnails.

- Search term:** A text input field with the placeholder 'Search'.
- Collection:** A dropdown menu set to 'All collections'.
- Popular tags:** Three small buttons labeled 'bread', 'nature', and 'satellite'.
- Latest images:** A grid of four image thumbnails with labels below them:
 - Lightnin' Hopkins
 - Olivia Ava
 - Muddy Waters
 - Akranes
- Thumbnail preview row:** Below the latest images, there are four small thumbnail images: a landscape, a yellow gradient, a red gradient, and a blue gradient.

1. Typing into the search box will automatically display the results below.
2. Clicking one of the Popular tags will filter the search results by that tag.
3. Clicking an image will take you to the Choose a format window (see image below).

Uploading a new image to the CMS

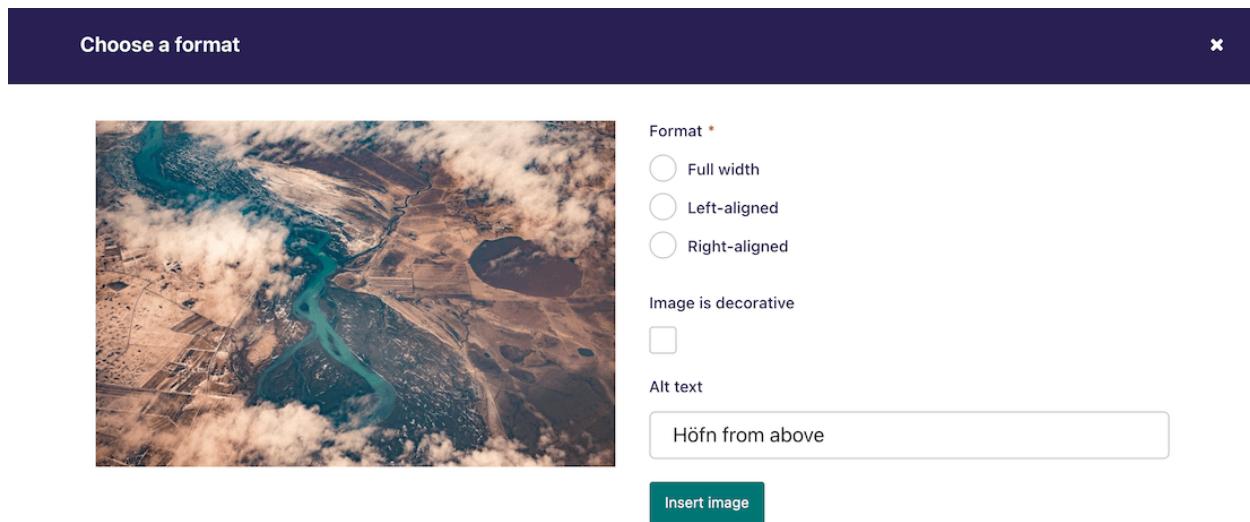


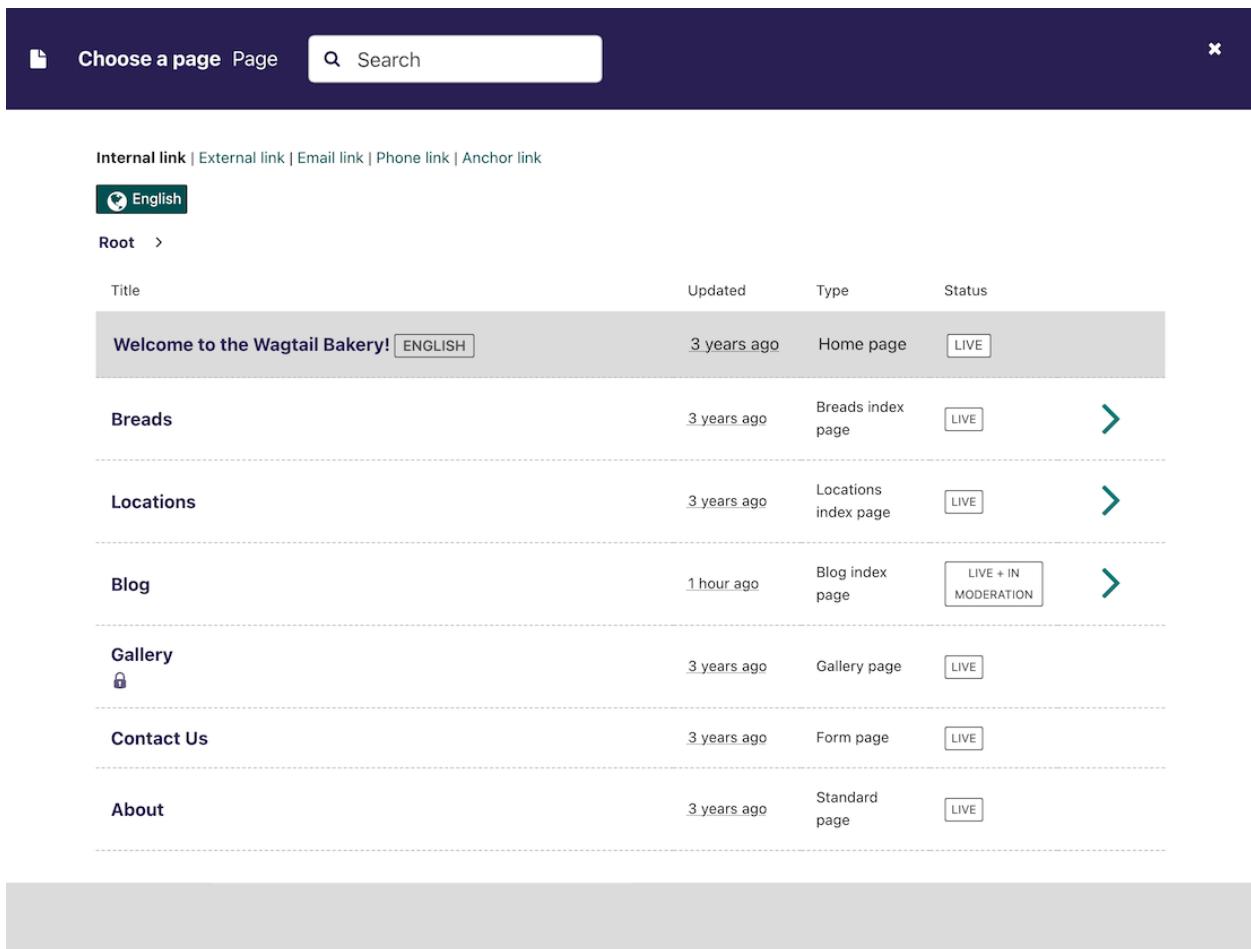
1. You must include an image title for your uploaded image
2. Click the *Choose file* button to choose an image from your computer.
3. *Tags* allows you to associate tags with the image you are uploading. This allows them to be more easily found when searching. Each tag should be separated by a space. Good practice for creating multiple word tags is to use an underscore between each word (for example `western_yellow_wagtail`).
4. Click *Upload* to insert the uploaded image into the carousel. The image will also be added to the main CMS image library for reuse in other content.

Inserting images using the rich text field

Images can also be inserted into the body text of a page via the rich text editor. When working in a rich text field, click the Image control. You will then be presented with the same options as for inserting images into the main carousel.

In addition, Wagtail allows you to choose the format of your image.





The screenshot shows the Wagtail CMS interface. At the top, there's a dark header with a 'Choose a page' button, a search bar containing 'Search', and a close button ('X'). Below the header, a navigation bar includes links for Internal link, External link, Email link, Phone link, and Anchor link. A language switcher shows 'English'. The main area is a table listing pages under the 'Root' category. The columns are 'Title', 'Updated', 'Type', and 'Status'. The table rows include:

Title	Updated	Type	Status
Welcome to the Wagtail Bakery! <small>ENGLISH</small>	3 years ago	Home page	<small>LIVE</small>
Breads	3 years ago	Breads index page	<small>LIVE</small> >
Locations	3 years ago	Locations index page	<small>LIVE</small> >
Blog	1 hour ago	Blog index page	<small>LIVE + IN MODERATION</small> >
Gallery <small>🔒</small>	3 years ago	Gallery page	<small>LIVE</small>
Contact Us	3 years ago	Form page	<small>LIVE</small>
About	3 years ago	Standard page	<small>LIVE</small>

- Search for an existing page to link to using the search bar at the top of the pop-up.
- Below the search bar you can select the type of link you want to insert. The following types are available:
 - Internal link: A link to an existing page within your website.
 - External link: A link to a page on another website.
 - Email link: A link that will open the user's default email client with the email address prepopulated.
 - Phone link: A link that will open the user's default client for initiating audio calls, with the phone number prepopulated.
 - Anchor link: A link that will scroll to a given hash id elsewhere on the same page.
- You can also navigate through the website to find an internal link via the explorer.

Inserting videos into body content

It is possible to embed media into the body text of a web page by clicking the *Embed* button in rich text toolbar.



- Copy and paste the web address for the media into the URL field and click Insert.

Heat is **gradually** transferred "from the surface of cakes, cookies, and breads to their centre. As heat travels through it transforms batters and doughs into baked goods with a firm dry crust and a softer centre".[2] Baking can be combined with grilling to produce a hybrid barbecue variant by using both methods simultaneously, or one after the other. Baking is related to barbecuing because the concept of the masonry oven is similar to that of a smoke pit.

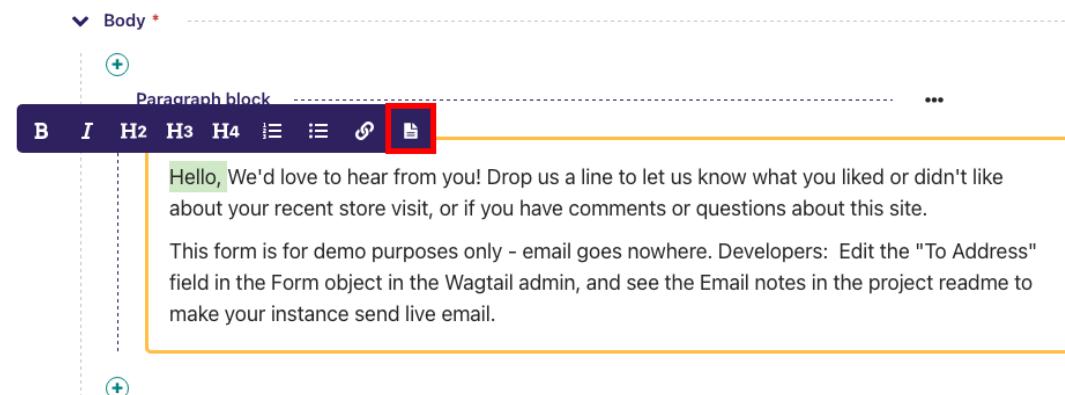


Write something or type '/' to insert a block

- A placeholder of the media will be inserted into the text area.

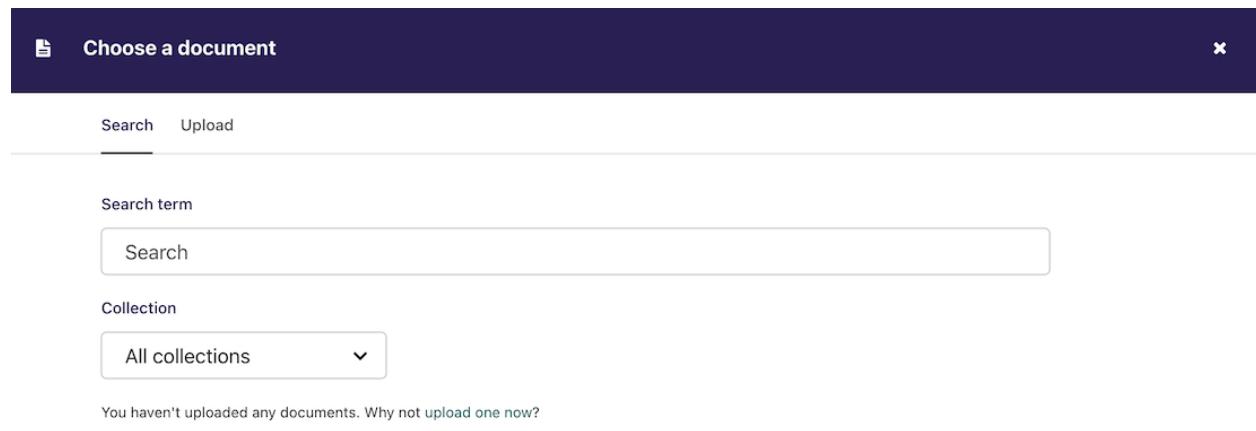
The embed button can be used to import media from a number of supported providers, you can see the [full list of supported providers](#) in Wagtail's source code.

Inserting links to documents into body text



It is possible to insert links to documents held in the CMS into the body text of a web page by clicking the button above in the rich text field.

The process for doing this is the same as when inserting an image. You are given the choice of either choosing a document from the CMS, or uploading a new document.



Adding multiple items

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

The screenshot shows a list of author entries under the 'Author(s)' section. Each entry includes a small profile icon, the author's name, and a trash can icon in the top right corner. A red box highlights the trash can icon for the second author entry. Below the list is a green 'Add author(s)' button.

- Whenever you see the “+ Add” button illustrated here with “Add author(s)”, 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 *Author(s)* items.
- You can delete an individual item by pressing the trash can in the top-right.
- You can add more items by clicking the Add button again.
- You can reorder your multiple items using the up and down arrows. Doing this will affect the order in which they are displayed 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 above each field that must be completed.

The screenshot shows a validation error for a required field. At the top, a red bar displays the message: “⚠ The page could not be created due to validation errors”. Below this, the page title is “New: Standard page”. The navigation tabs are Content (with 1 error), Promote (with 1 error), and Settings. The Content tab is active. A red box highlights the “Content” tab. In the main content area, a red box highlights the “Page title” field, which has a red error message: “⚠ This field is required.” Below the field is the placeholder text: “The page title as you'd like it to be seen by the public”. Further down, another red box highlights the “Introduction” field, which contains a large empty text area with the placeholder text: “Text to describe the page”.

Edit Page tabs

A common feature of the *Edit* pages for all page types is the three tabs at the top of the screen. The first, *Content*, is where you build the content of the page itself.

The Promote tab

The Promote tab is where you can configure a page's metadata, to help search engines find and index it. Below is a description of all the default fields under this tab.

For Search Engines

- **Slug:** The section of the URL that appears after your website's domain, for example `http://domain.com/blog/[my-slug]/`. This is automatically generated from the main page title, which is set in the Content tab. Slugs should be entirely lowercase, with words separated by hyphens (-). It is recommended that you don't change a page's slug once a page is published.
- **Title tag:** This is the bold headline that often shows up search engine results. This is one of the most significant elements of how search engines rank the page. The keywords used here should align with the keywords you wish to be found for. If you don't think this field is working, ask your developers to check they have configured the site to output the appropriate tags on the frontend.
- **Meta description:** This is the descriptive text displayed underneath a headline in search engine results. It is designed to explain what this page is about. It has no impact on how search engines rank your content, but it can impact on the likelihood that a user will click your result. Ideally 140 to 155 characters in length. If you don't think this field is working, ask your developers to check they have configured the site to output the appropriate tags on the frontend.

For Site Menus

- **Show in menus:** Ticking this box will ensure that the page is included in automatically generated menus on your site. Note: A page will only display in menus if all of its parent pages also have *Show in menus* ticked.

The screenshot shows the Wagtail Promote tab interface. At the top, there's a header with a back arrow, the text "New: Standard page", and three small icons. Below the header, there are three tabs: "Content", "Promote" (which is underlined), and "Settings".

For search engines

- Slug ***: A text input field containing "my-slug". Below it is a placeholder: "The name of the page as it will appear in URLs e.g http://domain.com/blog/[my-slug]/".
- Title tag**: A text input field.
- Meta description**: A text input field.

For site menus

- Show in menus**: A checkbox that is unchecked.

Below these fields is a teal "Save draft" button with a "DRAFT" icon and a small upward arrow icon to its right.

Note: You may see more fields than this in your promote tab. These are just the default fields, but you are free to add other fields to this section as necessary.

The Settings Tab

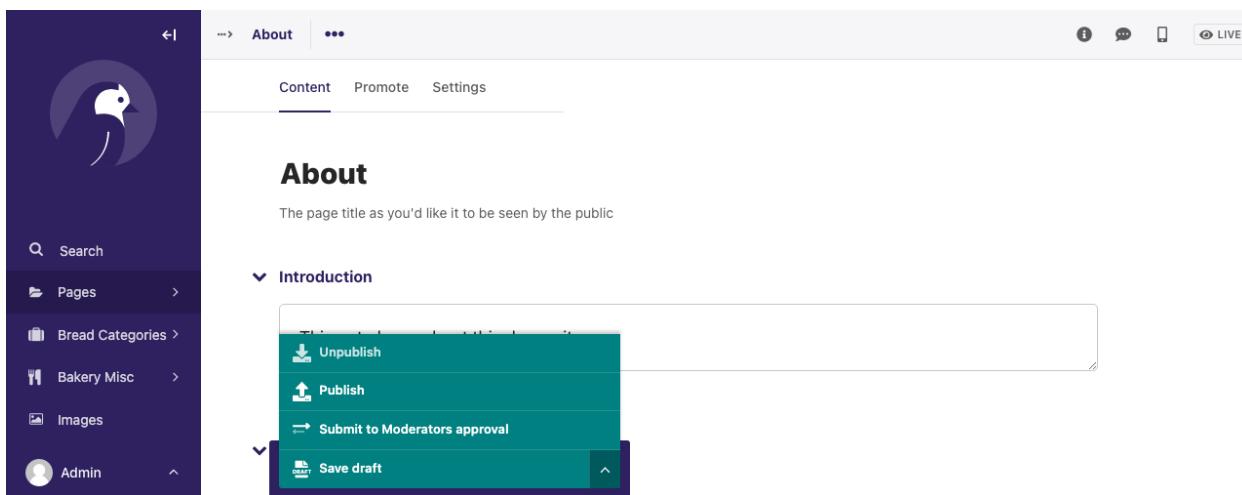
The *Settings* tab has three fields by default.

- **Go Live date/time:** Sets the date and time at which the changes should go live when published. See [Scheduled Publishing](#) for more details.
- **Expiry date/time:** Sets the date and time at which this page should be unpublished.
- **Privacy:** Sets restrictions for who can view the page on the frontend. Also applies to all child pages.

Previewing and submitting pages for moderation

The Save/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 to Moderators approval:** Saves your current changes and submits the page for moderation. The page will then enter a moderation workflow: a set of tasks which, when all are approved, will publish the page (by default, depending on your site [settings](#)). This button may be missing if the site administrator has [disabled moderation](#), or hasn't assigned a workflow to this part of the site. (all roles)
- **Publish/Unpublish:** Clicking the *Publish* button will publish this page. Clicking the *Unpublish* button will take you to a confirmation screen asking you to confirm that you wish to 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)



Other common page actions are available in the **Actions** dropdown, at the top of the screen.

Page previews

To access Wagtail's page preview, head over to the Preview side panel. A live-updating preview shows to the right of the page in Mobile mode, and it can be opened in a new tab displaying the page as it would look if published.

The screenshot shows the Wagtail edit interface for a page titled 'Bread and Circuses'. The top navigation bar includes 'Bread and Circuses', three dots, and icons for file operations and live preview. Below the title are tabs for 'Content', 'Promote', and 'Settings', with 'Content' selected.

The main content area contains:

- Bread and Circuses**: The page title as you'd like it to be seen by the public.
- Subtitle**: A field containing 'The art of baking'.
- Introduction**: A box containing text about baking: 'Baking is a method of cooking food that uses prolonged dry heat, normally in an oven, but also in hot ashes, or on hot stones. The most common baked item is bread but many other types of foods are baked.' Below this is a placeholder text 'Text to describe the page'.
- Image**: A thumbnail image of two Bedouins baking bread over an open fire, labeled 'Bedouins'.

At the bottom is a teal footer bar with a 'Save draft' button and a note about image dimensions: 'Image must be at least 100px wide and 3000px high'.

To the right, a preview card for 'The Wagtail Bakery' shows the page title, a breadcrumb trail ('Home > Blog > Bread and Circuses'), and a thumbnail image of the Bedouins baking.

Bread and Circuses

The art of baking

Baking is a method of cooking food that uses prolonged dry heat, normally in an oven, but also in hot ashes, or on hot stones. The most common baked item is bread but many other types of foods are baked.

Text to describe the page

Image



Bedouins

DRAFT Save draft

Image must be at least 100px wide and 3000px high

The Wagtail Bakery

Home > Blog > Bread and Circuses



1.7.5 Editing existing pages

Here is how 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.
- Clicking the *Edit* icon for a page in the explorer menu.

The screenshot shows the Wagtail admin interface for editing a page titled 'Bread and Circuses'. The left sidebar has a dark purple theme with a white penguin icon. The main content area has a light gray background. At the top, there's a breadcrumb navigation bar with 'Bread and Circuses' and three dots. Below it is a horizontal menu with 'Content' (which is underlined), 'Promote', and 'Settings'. In the top right corner, there are several small icons: a person, a speech bubble with a '1', a document, a circular arrow, and a 'LIVE' button.

Bread and Circuses

The page title as you'd like it to be seen by the public

Subtitle

The art of baking

Introduction

Baking is a method of cooking food that uses prolonged dry heat, normally in an oven, but also in hot ashes, or on hot stones. The most common baked item is bread but many other types of foods are baked.

Text to describe the page

Image

Bedouins

Save draft

and 3000px.

- When editing an existing page the title of the page being edited is displayed at the top of the page.
- You can perform various actions (such as copy, move or delete a page) by opening the secondary menu at the right of the top breadcrumb.
- The header has a link to the page history.
- In the Info side panel, you can find the page type and other important page information.
- If published, a link to the live version of the page is shown on the top right.
- You can toggle commenting mode by clicking the icon in the top right, which also shows the number of comments.
- You can change the title of the page by clicking into the title field.
- When you're typing into a field, help text is often displayed on the right-hand side of the screen.

Workflow

If the page is currently in a workflow, you'll see an additional indicator in the Info side panel, showing the current workflow task. Clicking the “View details” button will show more information about the page’s progress through the workflow, as well as any comments left by reviewers.

The screenshot shows the Wagtail admin interface with the 'Blog' page selected. The top navigation bar includes 'Blog' and three dots. The main content area shows a 'Blog' section with a title and a rich text editor containing 'Welcome to our blog'. Below it is a text input field with placeholder 'Text to describe the page'. A 'Image' section displays a thumbnail of a bread slice with the caption 'Breads 3'. A note at the bottom states 'Landscape mode only; horizontal width between 1000px and 3000px.' The right sidebar, titled 'I →', contains the following information:

- Live**
- In Moderation** (15 hours ago) [View details](#)
- Sent to Moderators approval** 15 hours ago
- English** (No other translations)
- Locked** (You can edit this page, but others may not.)
- Visible to all** (Once live anyone can view) [Change privacy](#)
- Blog index page** (First published 3 years ago)

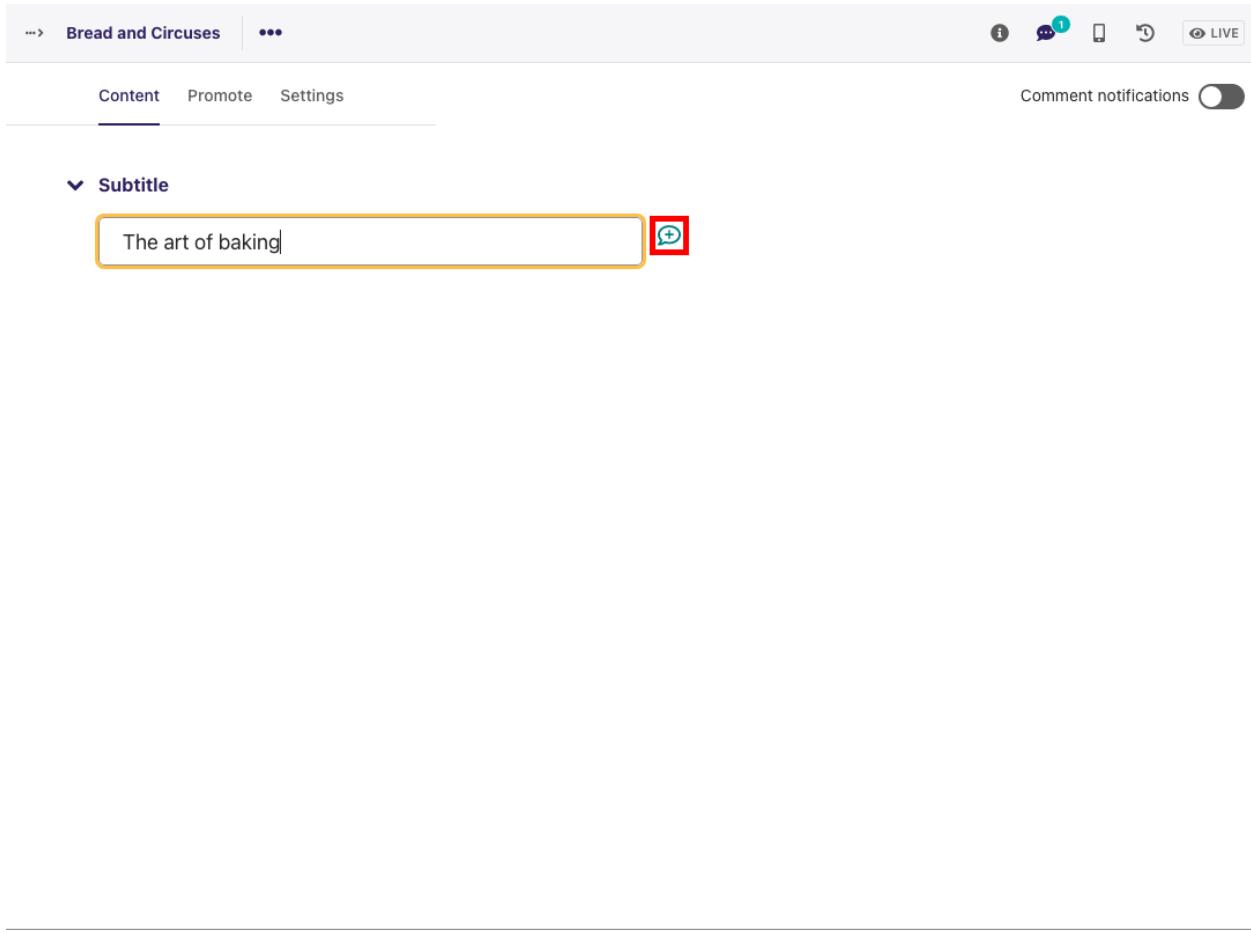
At the bottom of the sidebar, there is a teal button labeled 'Save draft' with a 'DRAFT' icon.

If you can perform moderation actions (for example, approval or requesting changes) on the current task, you’ll see additional options in the action menu at the bottom of the page.

Commenting

Use the comment icon at the top right to enable commenting mode. If there are comments on the page, the number of comments is shown alongside the icon.

When in commenting mode, hovering over a commentable field or block will reveal a comment button.



If there is no pre-existing comment on the field, you can use this to create a new comment.

The screenshot shows the Wagtail rich text editor with a comment dialog open. The text input field contains 'The art of baking'. To the right, a comment card is displayed for 'Admin User' from '6 Sept 2022, 11:36'. The card has a text input field 'Enter your comments...' and two buttons: 'Comment' and 'Cancel'.

If there is an existing comment, clicking either the field button or the comment will bring the comment thread into focus, allowing you to add new replies.

The screenshot shows the Wagtail rich text editor with an existing comment thread. The text input field contains 'Baking is a method of cooking food that uses'. To the right, a comment card is shown for 'Admin User' from 'Sep 6, 2022, 1:53 PM'. The card has the text 'Should we add examples?' and an 'Enter your reply...' input field.

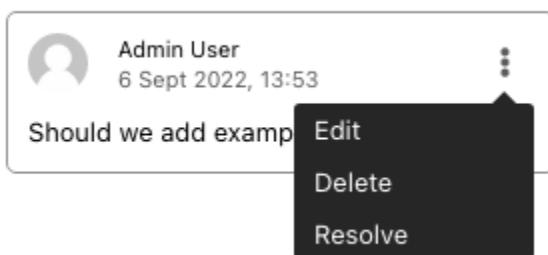
Within a rich text field, you can highlight text and click the comment button to add an inline comment.



Heat is **gradually** transferred "from the surface of cakes, cookies and breads to their centre. As it

All of these actions can also be performed using the comment shortcut, **Ctrl + Alt + M** / **Cmd + Alt + M**.

Clicking the three dots in the top right of an existing comment will open its menu. This allows you to resolve a comment, as well as to edit or delete any of your own comments.



Any comments or changes to comments will only be saved when the page is saved.

Note: Currently, fields inside `InlinePanels` and `ListBlocks` are uncommentable.

The arrow to the right of the comments icon shows the comment notifications panel, where you can enable or disable email notifications for other users' comments on the page.



Note: You will always receive email notifications for threads you are part of, unless you opt out of all comment notifications in your account settings. All participants in a thread will receive email notifications for new replies, even if they no longer have permission to edit the page.

1.7.6 Copying and creating aliases to existing pages

Copying Pages

Sometimes you might not need to create a new page from scratch. For example, you could have several pages that are very similar in terms of structure, but differ in content. In that case, you can copy an existing page and only change the required parts.

Here is how to copy an existing page:

- When you hover over a page in an *Explorer page*, click on *More* and then *Copy*.

The screenshot shows a list of bread pages in the Wagtail admin. The first page, 'Anadama', has a context menu open with the 'Copy' option highlighted. Other options in the menu include 'Move', 'Delete', 'Unpublish', 'History', and 'Sort menu order'. The menu is displayed over the list of pages.

	Title	Updated	Type	Status
<input type="checkbox"/>	Anadama	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Anadama	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Anpan	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Bazin	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Bhakri	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Black bread	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Appam	24 minutes ago	Bread page	LIVE
<input type="checkbox"/>	Arena	24 minutes ago	Bread page	LIVE

- You are then taken to a form where you can enter the copy's title, slug and choose the parent page (if you want to change it). You then get the option to publish the copied page right away and an option to mark the copy as an alias of the original page (see [Aliasing pages](#) for more information about aliasing pages).

Copy Anadama

New title *

New slug *

New parent page *

 Breads

This copy will be a child of this given parent page.

Publish copied page

This page is live. Would you like to publish its copy as well?

Alias

Keep the new pages updated with future changes

Copy this page

-
- Once you completed this form, press *Copy this page*. The page will then be copied and appear in the Explorer page.

Aliasing pages

When copying a page, you also have the option to mark it as an alias. The content of an aliased page will stay in sync with the original. This is particularly useful when a page is required to be available in multiple places. For example, let's say you have a page about Brioche in the Breads section. You then want to also make this page available in the Pastries section. One way to do this is to create a copy of the Brioche page and change the parent page to the Pastries page. However you now need to remember to update this copy each time the original is modified. If a copy is marked as an alias Wagtail will do this for you each time a modification to the original page is published.

Here is how to create an alias to an existing page:

- The first step is the same as it is for [Copying and creating aliases to existing pages](#). When you hover over a page in the [Explorer page](#), click on *More* and then *Copy*.
- When you get to the copy page form, you can then choose another page as the parent page. Click the *Choose another page* button.

Copy Anadama

New title *

New Anadama

New slug *

new-anadama-bread

New parent page *



Breads

Change

Edit

This copy will be a child of this given parent page.

Publish copied page



This page is live. Would you like to publish its copy as well?

Alias



Keep the new pages updated with future changes

Copy this page

- Select and click on the desired parent page.
- Make sure the *Alias* checkbox is ticked.
- You can then complete the page aliasing by clicking on the *Copy this page* button.

Copy Anadama

New title *

Anadama

New slug *

anadama-bread

New parent page *



Breads

This copy will be a child of this given parent page.

Publish copied page



This page is live. Would you like to publish its copy as well?

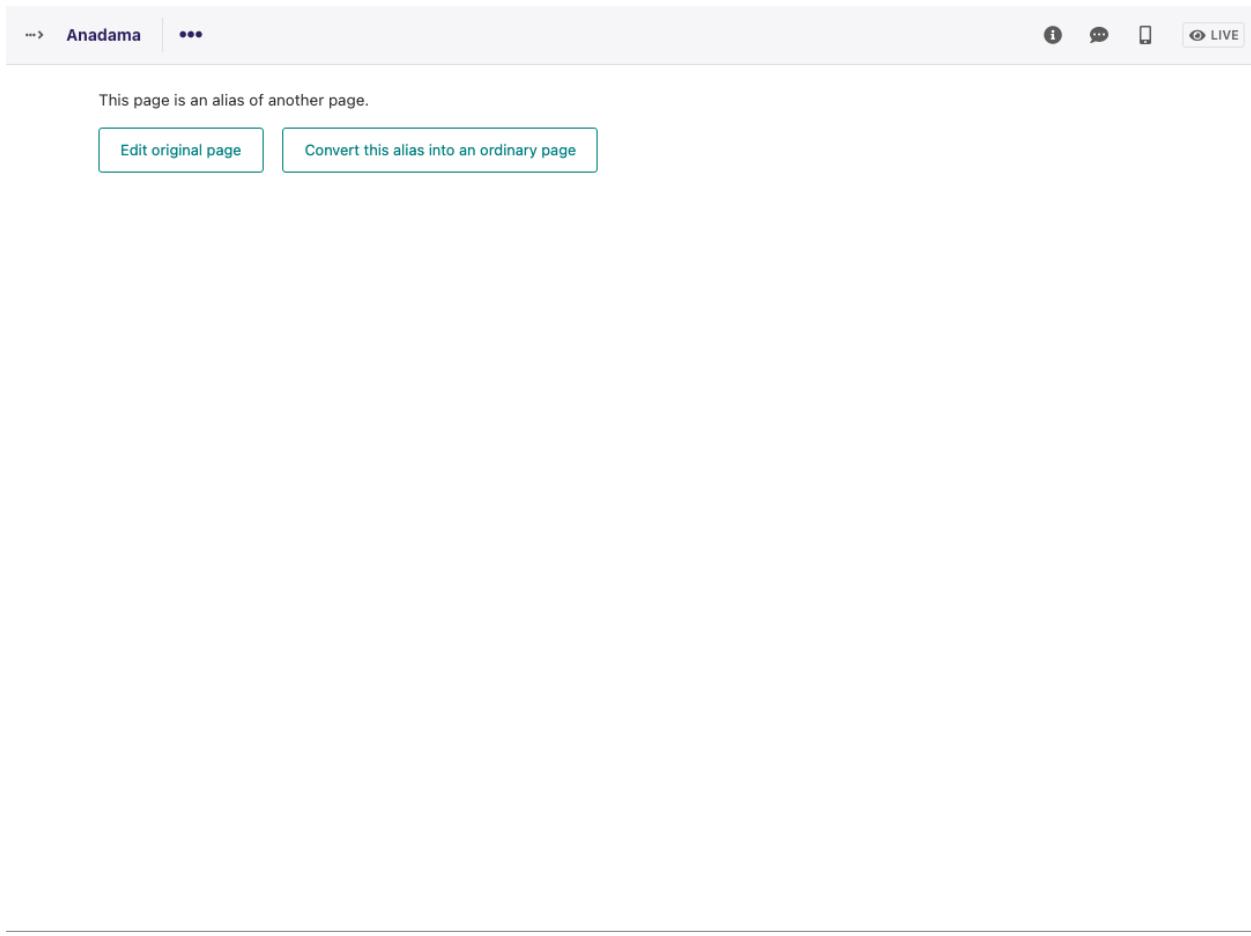
Alias



Keep the new pages updated with future changes

[Copy this page](#)

-
- The aliased page will then appear under the new parent in the Explorer.
 - When you try to edit the aliased page, you are notified that it is an alias of another page. You will then be offered two options: edit the original page (with changes appearing in both places) or convert the alias page into an ordinary page (a conventional copy, not updated when the original changes).



1.7.7 Managing documents, images, snippets and collections

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.

The screenshot shows the Wagtail CMS interface for managing documents. At the top, there's a header with the title 'Documents'. Below the header is a search bar with the placeholder 'Search documents'. To the right of the search bar is a green button labeled 'Add a document' with a plus sign icon. Underneath the header, there's a section titled 'Collection' with a dropdown menu set to 'All collections'. The main area displays a table of uploaded files:

Title	File	Collection	Created
TPS Report Full Guidelines	TPS_Report_Full_Guidelines.pdf	Other	24 minutes ago
Project risk analysis	Project_risk_analysis.pdf	Root	24 minutes ago
Full Accessibility Assessment 2022-02-10	Full_Accessibility_Assessment_2022-02-10.pdf	Root	24 minutes ago
2022 yearly report	2022_yearly_report.pdf	Root	24 minutes ago

- 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 *Collection* by selecting one from the collections dropdown above the documents list.

The screenshot shows the Wagtail Document index page. At the top, there is a header with a document icon, the text "Documents", a search bar containing "Search documents", and a green button with a plus sign and the text "Add a document". Below the header, there is a "Collection" dropdown set to "All collections". A table lists five documents:

<input type="checkbox"/>	Title	File	Collection	Created
<input type="checkbox"/>	TPS Report Full Guidelines	TPS_Report_Full_Guidelines.pdf	Other	24 minutes ago
<input checked="" type="checkbox"/>	Project risk analysis	Project_risk_analysis.pdf	Root	24 minutes ago
<input checked="" type="checkbox"/>	Full Accessibility Assessment 2022-02-10	Full_Accessibility_Assessment_2022-02-10.pdf	Root	24 minutes ago
<input type="checkbox"/>	2022 yearly report	2022_yearly_report.pdf	Root	24 minutes ago

Below the table, it says "Page 1 of 1.". At the bottom, there is a dark blue footer bar with icons for "Tag", "Add to collection", and "Delete", followed by the text "2 documents selected".

- Select multiple documents by checking the checkbox on the left of each row, then use the bulk actions bar at the bottom to perform an action on all selected documents.
- Edit the details of a document by clicking the document title.

Editing TPS Report Full Guidelines

Title *

File *

 [TPS_Report_Full_Guidelines.pdf](#)

Change document:

No file chosen

Collection *

▾

Tags

 Guides 

Multi-word tags with spaces will automatically be enclosed in double quotes (").

-
- 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.
 - Change the document's collection using the collection dropdown.
 - Add or remove tags using the Tags field.
 - Save or delete documents using the buttons at the bottom of the interface.

Warning: Deleted documents cannot be recovered.

Images

If you want to edit, add or remove images from the CMS outside of the individual pages you can do so from the Images interface. This is accessed from the left-hand menu.

The screenshot shows the Wagtail Images interface. At the top, there's a header with a camera icon, the word "Images", a search bar containing "Search images", and a green button with a plus sign and the text "Add an image". Below the header are three filter sections: "Collection" (dropdown set to "All collections"), "Sort by" (dropdown set to "Newest"), and "Entries per page" (dropdown set to "10"). Underneath these filters is a section titled "Popular Tags:" with three tags: "bread", "nature", and "satellite", each accompanied by a small icon. The main area displays a grid of eight images, each with a caption below it:

-  Lightnin' Hopkins
-  Olivia Ava
-  Muddy Waters
-  Akranes
-  Höfn from above
-  Old buildings above Vik
-  Glacier descending near Hof
-  Image by Sverrir Thorolfsson

- Select multiple images by checking the checkbox on the top left of each image block, then use the bulk actions bar at the bottom to perform an action on all selected images.

 **Images** + Add an image

Collection Sort by Entries per page

All collections Newest 10

Popular Tags:   

<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
			
Lightnin' Hopkins	Olivia Ava	Muddy Waters	Akranes
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
			
Höfn			Image by Sverrir Thorolfsson

Höfn Tag Add to collection Delete 5 images selected near Hof

- Clicking an image will allow you to edit the data associated with it. This includes the title, the focal point of the image and much more.

Editing Olivia Ava

Title *

File *

 olivia_ava.jpeg
(300x282)

Change image file:

No file chosen

Supported formats: GIF, JPEG, PNG, WEBP. Maximum filesize: 10.0 MB.

Collection *

Other

▼

Tags

Multi-word tags with spaces will automatically be enclosed in double quotes ("").



Focal point (optional)

To define this image's most important region, drag a box over the image above.

Max dimensions
300x282

Filesize
45.0 KB

Usage
Used 0 times

Changing the image

- When editing an image you can replace the file associated with that image record. This means you can update images without having to update the pages on which they are placed.

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

Focal area

- This interface allows you to select a focal area which can effect how your image displays to visitors on the front-end.
- If your images are cropped in some way to make them fit to a specific shape, then the focal area will define the centre point from which the image is cropped.
- To set the focal area, drag a marquee around the most important element of the image.
- To remove the focal area, hit the button below the image.
- If the feature is set up in your website, then on the front-end you will see the crop of this image focusing on your selection.

Snippets

Snippets allow you to create elements on a website once and reuse them in multiple places. Then, if you want to change something on the snippet, you only need to change it once, and it will change across all the occurrences of the snippet.

How snippets are used can vary widely between websites. Here are a few examples of things Torchbox have used snippets for on our clients' websites:

- For staff contact details, so that they can be added to many pages but managed in one place
- For Adverts, either to be applied sitewide or on individual pages
- To manage links in a global area of the site, for example in the footer
- For Calls to Action, such as Newsletter sign up blocks, that may be consistent across many different pages

The Snippets menu

Snippets

Name	Instances
Bread ingredients	5
Bread types	18
Countries of Origin	25
Footer Text	1
People	4

-
- You can access the Snippets menu by clicking on the ‘Snippets’ link in the left-hand menu bar.
 - To add or edit a snippet, click on the snippet type you are interested in (often help text will be included to help you in selecting the right type)
 - Click on an individual snippet to edit, or click ‘Add ...’ in the top right to add a new snippet
 - To delete snippets, select one or more snippets with the tickbox on the left and then click the delete button near the top right

Warning: Editing a snippet will change it on all of the pages on which it has been used. In the top-right of the Snippet edit screen you will see a label saying how many times the snippet has been used. Clicking this label will display a listing of all of these pages.

The screenshot shows the Wagtail Snippet edit interface for a snippet named 'Olivia Ava'. The top navigation bar shows the path '…> Olivia Ava'. The main form has sections for 'Name' (with fields for 'First name *' containing 'Olivia' and 'Last name *' containing 'Ava'), 'Job title *' (containing 'Director'), and a 'Save' button. A sidebar on the right displays the 'Usage' section, which shows 'Used 2 times' with a red box around it. There are also icons for info and refresh.

Adding snippets whilst editing a page

If you are editing a page, and you find yourself in need of a new snippet, do not fear! You can create a new one without leaving the page you are editing:

- Whilst editing the page, open the snippets interface in a new tab, either by Ctrl+click (cmd+click on Mac) or by right clicking it and selecting ‘Open in new tab’ from the resulting menu.
- Add the snippet in this new tab as you normally would.
- Return to your existing tab and reopen the Snippet chooser window.
- You should now see your new snippet, even though you didn’t leave the edit page.

Note: Even though this is possible, it is worth saving your page as a draft as often as possible, to avoid your changes being lost by navigating away from the edit page accidentally.

Collections

Access to specific sets of images and documents can be controlled by setting up ‘collections’. By default all images and documents belong to the ‘root’ collection, but users with appropriate permissions can create new collections through the **Settings -> Collections** area of the admin interface.



Add a collection

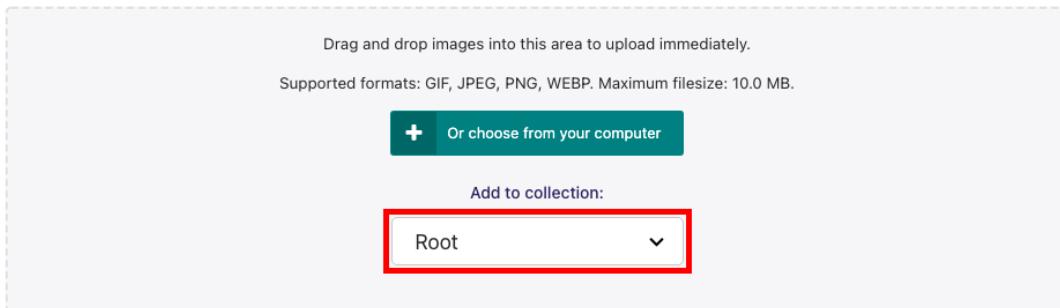
- Clicking the **add a collection** button will allow you to create a collection. Name your collection and click **save**.



Add images / documents to a collection

- Navigate to the **Images** or **Documents** section and select a collection from the dropdown menu.
- You can also select a collection as part of uploading multiple images or documents.

Add images



Drag and drop images into this area to upload immediately.

Supported formats: GIF, JPEG, PNG, WEBP. Maximum filesize: 10.0 MB.

+ Or choose from your computer

Add to collection:

Root ▾

- You can also edit an image or document directly by clicking on it to assign it to a collection.

🖼 Editing Olivia Ava

Title *

File *

 olivia_ava.jpeg
(300x282)

Change image file:

No file chosen

Supported formats: GIF, JPEG, PNG, WEBP. Maximum filesize: 10.0 MB.



Collection *

Tags

Multi-word tags with spaces will automatically be enclosed in double quotes ("").

Focal point (optional)
To define this image's most important region, drag a box over the image above.

Max dimensions
300x282

Filesize
45.0 KB

Usage
Used 0 times

Privacy settings

- To set permissions determining who is able to view documents within a collection, navigate to **Settings > Collections** and select a collection. Then click the privacy button above the collection name.

📄 Editing BreadPage Images

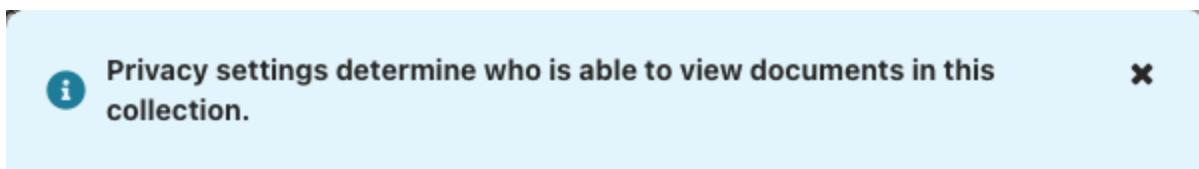
Privacy

Name *

Parent *

Select hierarchical position. Note: a collection cannot become a child of itself or one of its descendants.

- Within the privacy settings overlay, select the level of privacy for the collection.



Change privacy

- Public
- Private, accessible to logged-in users
- Private, accessible with the following password
- Private, accessible to users in specific groups

Save

Permissions set on a collection apply to that collection and all collections below it in the hierarchy. Therefore, if you make the ‘root’ collection private, all documents in the site will be private. Permissions set on other collections apply to that collection only.

Note: Although privacy settings are added to a collection, they are only enforced for documents within the collection. Privacy settings do not apply to images.

1.7.8 Managing Redirects

About redirects

When dealing with publishing and unpublishing pages you will eventually need to make redirects. A redirect ensures that when a page is no longer available (404), the visitor and search engines are sent to a new page. Therefore the visitor won’t end up in a breaking journey which would result in a page not found.

Wagtail considers two types of configurable redirects depending on whether *Permanent* is checked or not:

- Permanent redirect (checked by default)
- Temporary redirect

For both redirects the visitor won’t experience a difference when visiting a page, but the search engine will react to these two type of redirects differently.

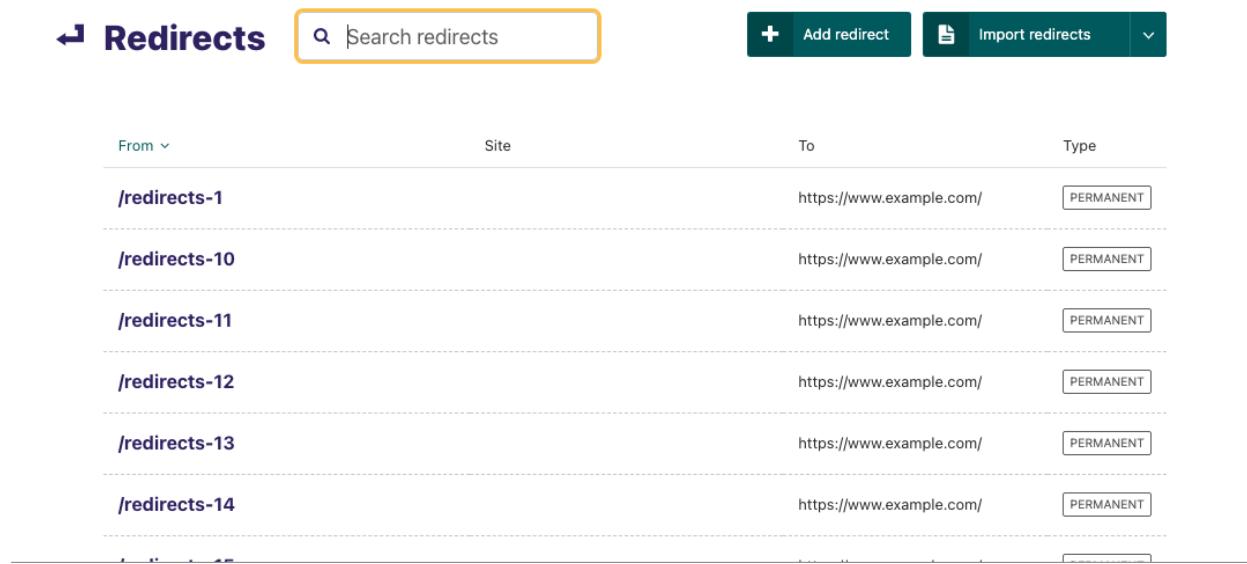
- In the case of a temporary redirect a search engine will keep track of your old page and will index the redirected page as well.

- With a permanent redirect, the search engine will mark the old page as obsolete and considers the new page as a replacement.

Note: As a best practice Wagtail will check redirects as permanent by default, in order to prevent the undermining of your search engine ranking.

Configuring redirects

To configure redirects head over to ‘Redirects’, which can be found in the Settings menu, accessible via the left-hand menu bar.



The screenshot shows the Wagtail Redirects list page. At the top, there is a header with a back arrow, the title 'Redirects', a search bar containing 'Search redirects', and buttons for 'Add redirect' and 'Import redirects'. Below the header is a table with four columns: 'From', 'Site', 'To', and 'Type'. The 'From' column contains URLs starting with '/redirects-', and the 'To' column contains 'https://www.example.com/'. All entries in the 'Type' column are labeled 'PERMANENT'. There are six rows in the table, each representing a redirect rule.

From	Site	To	Type
/redirects-1		https://www.example.com/	PERMANENT
/redirects-10		https://www.example.com/	PERMANENT
/redirects-11		https://www.example.com/	PERMANENT
/redirects-12		https://www.example.com/	PERMANENT
/redirects-13		https://www.example.com/	PERMANENT
/redirects-14		https://www.example.com/	PERMANENT

- Add a redirect by clicking the *Add redirect* button in the top-right.
- Search for redirects already configured by entering your search term in the search bar. The results will be automatically updated as you type.
- Edit the details of a redirect by clicking the URL path in the listing.

➡ Editing /redirects-1

Redirect from *

From site

All sites

Permanent

Recommended. Permanent redirects ensure search engines forget the old page (the 'Redirect from') and index the new page instead.

Redirect to a page

 Choose a page

Redirect to any URL

Save Delete redirect

- Set *Redirect from* to the URL pattern which is no longer available on your site.
- Set the *From site* if applicable (for eg: a multi-site environment).
- Check whether the redirect is *Permanent* or temporary (unchecked).

As a last step you can either redirect to a new page within Wagtail **or** you can redirect the page to a different domain outside of Wagtail.

- Select your page from the explorer for *Redirect to a page*.
- Set a full-domain and path for *Redirect to any URL*.

Note: Keep in mind a redirect will only be initiated if the page is not found. It will not be applied to existing pages (200) which will resolve on your site.

1.7.9 Administrator tasks

This section of the guide documents how to perform common tasks as an administrator of a Wagtail site.

Managing users and roles

As an administrator, a common task will be adding, modifying or removing user profiles.

This is done via the ‘Users’ interface, which can be found in the Settings menu, accessible via the left-hand menu bar.

In this interface you can see all of your users, their usernames, their ‘level’ of access (otherwise known as their ‘role’), and their status, either active or inactive.

You can sort this listing either via Name or Username.

Name	Username	Level	Status	Last Login
Olivia Ava	olivia.ava@example.com		ACTIVE	
Admin User	admin	Admin	ACTIVE	24 minutes ago
Muddy Waters	muddy.waters@example.com		ACTIVE	

Page 1 of 1.

- Select multiple users by checking the checkbox on the left of each row, then use the bulk actions bar at the bottom to perform an action on all selected users.

Name	Username	Level	Status	Last Login
Olivia Ava	olivia.ava@example.com		ACTIVE	
Admin User	admin	Admin	ACTIVE	24 minutes ago
Muddy Waters	muddy.waters@example.com		ACTIVE	

Delete Set active state Assign role 1 user selected

Clicking on a user's name will open their profile details. From here you can then edit that users details.

Note: It is possible to change user's passwords in this interface, but it is worth encouraging your users to use the 'Forgotten password' link on the login screen instead. This should save you some time!

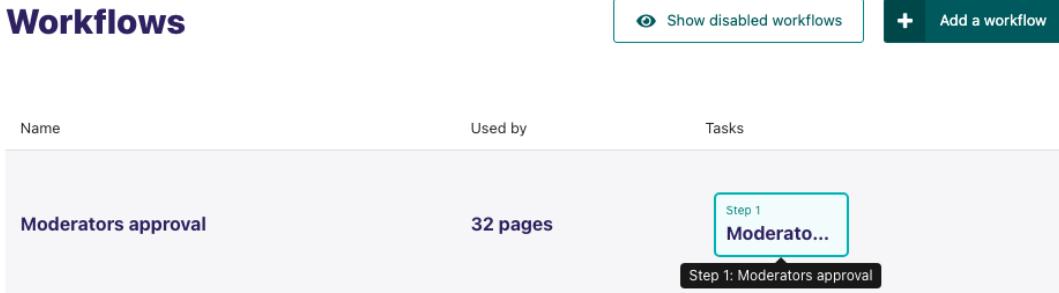
Click the 'Roles' tab to edit the level of access your users have. By default there are three roles:

Role	Create/view drafts	Publish content	Access Settings
Editor	Yes	No	No
Moderator	Yes	Yes	No
Administrator	Yes	Yes	Yes

Managing Workflows

Workflows allow you to configure how moderation works on your site. Workflows are sequences of tasks, all of which must be approved before the workflow completes (by default, this results in the publication of the page, but depends on your site *settings*).

The workflow management interface is accessed via the **Workflows** item in the **Settings** submenu, found in the left menu bar.



The screenshot shows the Wagtail Workflows interface. At the top, there is a header with a menu icon, the title 'Workflows', a button to 'Show disabled workflows', and a button to 'Add a workflow'. Below the header, there is a table with three columns: 'Name', 'Used by', and 'Tasks'. A single workflow row is visible, titled 'Moderators approval'. Under 'Used by', it says '32 pages'. Under 'Tasks', there is a box labeled 'Step 1' containing the text 'Moderato...'. A callout bubble points to this task with the text 'Step 1: Moderators approval'.

In this interface you can see all of the workflows on your site, and the order of tasks in each. You can click on a workflow to edit it or to assign it to part of the page tree, or use the **Add a workflow** button to create a new workflow. The number of pages each workflow applies to is also shown, and can be clicked for a complete listing.

Editing workflows

☰ Editing workflow Moderators approval

▼ Give your workflow a name *

Moderators approval

✚ Add tasks to your workflow

▼ Task

Task *



Moderators approval

...

+ Add task

─ Assign your workflow to pages

?

Workflows apply to child pages too. If you select a parent page here, its child pages will also use this workflow. See [the list of the pages](#) your workflow applies to.

Assigned pages

Page



Root

Delete

Save

^

Under **Add tasks to your workflow**, you can add, remove, or reorder tasks in a workflow. When adding a task to a workflow, you will be given the option to create a new task, or reuse an existing one.

Under **Assign your workflow to pages**, you can see a list of the pages to which the workflow has been assigned: any child pages will also have the same workflow, so if a workflow is assigned to the root page, it becomes the default workflow. You may remove it from pages using the **Delete** button to the right of each entry, or assign it to a page using the **Choose a page** button.

The action menu at the bottom allows you to save your changes, or to disable the workflow, which will cancel all pages currently in moderation on this workflow, and prevent others from starting it. On a previously disabled workflow, there is also the option to enable it again.

Creating and editing tasks

The screenshot shows the 'Workflow tasks' page. At the top right are two buttons: 'Show disabled tasks' (with a magnifying glass icon) and 'New workflow task' (with a plus sign icon). Below is a table with one row:

Name	Type	Used on
Moderators approval	Group approval task	Moderators approval

In the tasks interface, accessible via the `Workflow Tasks` item in the `Settings` submenu, you can see a list of the tasks currently available, as well as which workflows use each task. Similarly to workflows, you can click an existing task to edit it, or the `Add a task` button to create a new task.

When creating a task, if you have multiple task types available, these will be offered as options. By default, only `group approval tasks` are available. Creating a `group approval task`, you will be able to select one or multiple groups: members of any of these, as well as administrators, will be able to approve or reject moderation for this task.

>New workflow task

The form has the following fields:

- Name ***: An input field containing the placeholder text 'New task'.
- Groups ***: A section with two checkboxes:
 - Moderators
 - Editors
- A note below the groups section: 'Pages at this step in a workflow will be moderated or approved by these groups of users'.
- A large blue 'Create' button at the bottom.

When editing a task, you may find that some fields - including the name - are uneditable. This is to ensure workflow history remains consistent - if you find yourself needing to change the name, it is recommended that you disable the old task, and create a new one with the name you need. Disabling a task will cause any pages currently in moderation on that task to skip to the next task.

Promoted search results

Note: Promoted search results are an optional Wagtail feature. For details of how to enable them on a Wagtail installation, see [search_promotions](#)

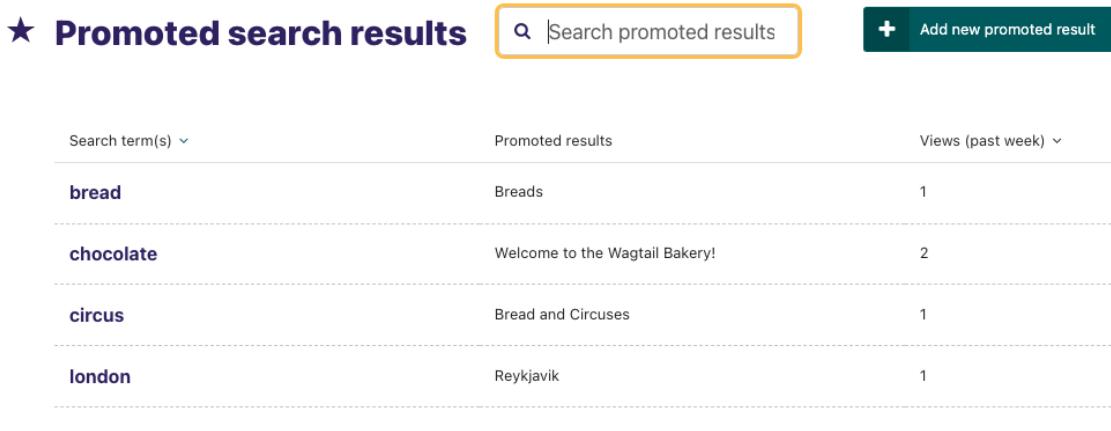
Wagtail allows you to promote certain search results dependent on the keyword or phrase entered by the user when searching. This can be particularly useful when users commonly refer to parts of your organisation via an acronym that isn't in official use, or if you want to direct users to a page when they enter a certain term related to the page but not included in the text of the page itself.

As a concrete example, one of our clients wanted to direct people who searched for ‘finances’ to their ‘Annual budget review’ page. The word ‘finances’ is not mentioned in either the title or the body of the target page, so they created a promoted search result for the word ‘finances’ that pushed the budget page to the very top of the results.

Note: The promoted result will only work if the user types *exactly* the phrase that you have set it up for. If you have variations of a phrase that you want to take into account, then you must create additional promoted results.

To set up the promoted search results, click on the ‘Promoted search results’ menu item in the ‘Settings’ menu.

Add a new promoted result from the button in the top right of the resulting screen, or edit an existing promoted result by clicking on it.



The screenshot shows the Wagtail Promoted search results interface. At the top, there is a header with a star icon, the text 'Promoted search results', a search bar containing 'Search promoted results', and a button labeled '+ Add new promoted result'. Below the header is a table with three columns: 'Search term(s) ▾', 'Promoted results', and 'Views (past week) ▾'. The table contains four rows of data:

Search term(s) ▾	Promoted results	Views (past week) ▾
bread	Breads	1
chocolate	Welcome to the Wagtail Bakery!	2
circus	Bread and Circuses	1
london	Reykjavik	1

At the bottom of the interface, the text 'Page 1 of 1.' is visible.

When adding a new promoted result, Wagtail provides you with a ‘Choose from popular search terms’ option. This will show you the most popular terms entered by users into your internal search. You can match this list against your existing promoted results to ensure that users are able to find what they are looking for.

Popular search terms X

Search term *

Search

Terms

Views (past week)

chocolate

2

london

1

circus

1

bread

1

Page 1 of 1.

You then add a the result itself by clicking ‘Add recommended page’. You can add multiple results, but be careful about adding too many, as you may end up hiding all of your organic results with promoted results, which may not be helpful for users who aren’t really sure what they are looking for.

★ Editing bread

Search term(s)/phrase *

[Choose from popular search terms](#)

Enter the full search string to match. An exact match is required for your Promoted Results to be displayed, wildcards are NOT allowed.

▼ Promoted search results

▼ Recommended page

...

Page *



Breads

Description

[Add recommended page](#)

Save

Delete

1.7.10 Browser issues

Some issues with Wagtail may come from the browser. Try [clearing the browser cache and cookies](#), or you can also try using Wagtail with [another browser](#) to see if the problem persists.

JavaScript is required to use Wagtail – make sure it is [enabled](#) in your browser.

Supported browsers

For the best experience and security, we recommend that you keep your browser up to date. Go to [Browse Happy](#) for more information.

IE11

Wagtail has officially removed support for the legacy Internet Explorer browser in 2.15.

- In Wagtail 2.11 (LTS), there was be a warning message displayed on the Wagtail dashboard for IE11 users with administrator role.
- In Wagtail 2.12, the message was displayed to all users regardless of their role.
- In Wagtail 2.13 and 2.14, the message was displayed at the top of all pages.
- Wagtail will no longer support IE11 as of version 2.15 and beyond, for any releases after 2.16 there will no longer be a warning to users.

If this affects you or your organisation, consider which alternative browsers you may be able to use. Wagtail is fully compatible with Microsoft Edge, Microsoft's replacement for Internet Explorer. You may consider using its IE mode to keep access to IE11-only sites, while other sites and apps like Wagtail can use modern browser capabilities.

Assistive technologies

We want Wagtail to be accessible for users of a wide range of assistive technologies, but are aware of many blockers currently. For an overview, see our [public accessibility audit](#).

1.8 Contributing to Wagtail

1.8.1 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/wagtail/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.

If your bug report is a security issue, **do not** report it with an issue. Please read our guide to [reporting security issues](#).

Issue tracking

We welcome bug reports, feature requests and pull requests through Wagtail's Github issue tracker.

Issues

An issue must always correspond to a specific action with a well-defined completion state: fixing a bug, adding a new feature, updating documentation, or cleaning up code. Open-ended issues where the end result is not immediately clear (“come up with a way of doing translations” or “Add more features to rich text fields.”) are better suited to [Github discussions](#), so that there can be feedback on clear way to progress the issue and identify when it has been completed through separate issues created from the discussion.

Do not use issues for support queries or other questions (“How do I do X?” - although “Implement a way of doing X” or “Document how to do X” could well be valid issues). These should be asked on [Stack Overflow](#) instead. For discussions that do not fit Stack Overflow’s question-and-answer format, see the other (Wagtail community support options)[<https://github.com/wagtail/wagtail#community-support>].

As soon as a ticket is opened - ideally within one day - a member of the core team will give it an initial classification, by either closing it due to it being invalid or updating it with the relevant labels. When a bug is opened, it will automatically be assigned the `type:Bug` and `status:Unconfirmed` labels, once confirmed the bug can have the unconfirmed status removed. A member of the team will potentially also add a release milestone to help guide the priority of this issue. Anyone is invited to help Wagtail with reproducing `status:Unconfirmed` bugs and commenting if it is a valid bug or not with additional steps to reproduce if needed.

Don’t be discouraged if you feel that your ticket has been given a lower priority than it deserves - this decision isn’t permanent. We will consider all feedback, and reassign or reopen tickets where appropriate. (From the other side, this means that the core team member doing the classification should feel free to make bold unilateral decisions - there’s no need to seek consensus first. If they make the wrong judgement call, that can always be reversed later.)

The possible milestones that it might be assigned to are as follows:

- **invalid** (closed): this issue doesn’t identify a specific action to be taken, or the action is not one that we want to take. For example - a bug report for something that’s working as designed, or a feature request for something that’s actively harmful.
- **real-soon-now**: no-one on the core team has resources allocated to work on this right now, but we know it’s a pain point, and it will be prioritised whenever we next get a chance to choose something new to work on. In practice, that kind of free choice doesn’t happen very often - there are lots of pressures determining what we work on from day to day - so if this is a feature or fix you need, we encourage you to work on it and contribute a pull request, rather than waiting for the core team to get round to it!
- A specific version number (for example **1.6**): the issue is important enough that it needs to be fixed in this version. There are resources allocated and/or plans to work on the issue in the given version.
- No milestone: the issue is accepted as valid once the `status:Unconfirmed` label is removed (when it’s confirmed as a report for a legitimate bug, or a useful feature request) but is not deemed a priority to work on (in the opinion of the core team). For example - a bug that’s only cosmetic, or a feature that would be kind of neat but not really essential. There are no resources allocated to it - feel free to take it on!

On some occasions it may take longer for the core team to classify an issue into a milestone. For example:

- It may require a non-trivial amount of work to confirm the presence of a bug. In this case, feedback and further details from other contributors, whether or not they can replicate the bug, would be particularly welcomed.
- It may require further discussion to decide whether the proposal is a good idea or not - if so, it will be tagged “design decision needed”.

We will endeavour to make sure that issues don't remain in this state for prolonged periods. Issues and PRs tagged "design decision needed" will be revisited regularly and discussed with at least two core contributors - we aim to review each ticket at least once per release cycle (= 6 weeks) as part of weekly core team meetings.

Pull requests

As with issues, the core team will classify pull requests as soon as they are opened, usually within one day. Unless the change is invalid or particularly contentious (in which case it will be closed or marked as "design decision needed"). It will generally be classified under the next applicable version - the next minor release for new features, or the next patch release for bugfixes - and marked as 'Needs review'.

- All contributors, core and non-core, are invited to offer feedback on the pull request.
- Core team members are invited to assign themselves to the pull request for review.
- More specific details on how to triage Pull Requests can be found on the [PR triage wiki page](#).

Subsequently (ideally within a week or two, but possibly longer for larger submissions) a core team member will merge it if it is ready to be merged, or tag it as requiring further work ('needs work' / 'needs tests' / 'needs docs'). Pull requests that require further work are handled and prioritised in the same way as issues - anyone is welcome to pick one up from the backlog, whether or not they were the original committer.

Rebasing / squashing of pull requests is welcome, but not essential. When doing so, do not squash commits that need reviewing into previous ones and make sure to preserve the sequence of changes. To fix mistakes in earlier commits, use `git commit --fixup` so that the final merge can be done with `git rebase -i --autosquash`.

Core team members working on Wagtail are expected to go through the same process with their own fork of the project.

Release schedule

We aim to release a new version every 2 months. To keep to this schedule, we will tend to 'bump' issues and PRs to a future release where necessary, rather than let them delay the present one. For this reason, an issue being tagged under a particular release milestone should not be taken as any kind of guarantee that the feature will actually be shipped in that release.

- See the [Release Schedule wiki page](#) for a full list of dates.
- See the [Roadmap wiki page](#) for a general guide of project planning.

1.8.2 Pull requests

If you're a Python or Django developer, [fork it](#) and read the [developing docs](#) to get stuck in! 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.org if you'd like us to suggest something!

For large-scale changes, we'd generally recommend breaking them down into smaller pull requests that achieve a single well-defined task and can be reviewed individually. If this isn't possible, we recommend opening a pull request on the [Wagtail RFCs](#) repository, so that there's a chance for the community to discuss the change before it gets implemented.

Development

Setting up a local copy of the Wagtail git repository is slightly more involved than running a release package of Wagtail, as it requires Node.js and npm for building JavaScript and CSS assets. (This is not required when running a release version, as the compiled assets are included in the release package.)

If you're happy to develop on a local virtual machine, the `docker-wagtail-develop` and `vagrant-wagtail-develop` setup scripts are the fastest way to get up and running. They will provide you with a running instance of the Wagtail Bakery demo site, with the Wagtail and bakerydemo codebases available as shared folders for editing on your host machine.

You can also set up a cloud development environment that you can work with in a browser-based IDE using the `gitpod-wagtail-develop` project.

(Build scripts for other platforms would be very much welcomed - if you create one, please let us know via the [Slack workspace!](#))

If you'd prefer to set up all the components manually, read on. These instructions assume that you're familiar with using pip and virtual environments to manage Python packages.

Setting up the Wagtail codebase

The preferred way to install the correct version of Node is to use [Node Version Manager \(nvm\)](#) or [Fast Node Manager \(fnn\)](#), which will always align the version with the supplied `.nvmrc` file in the root of the project. To ensure you are running the correct version of Node, run `nvm install` or `fnn install` from the project root. Alternatively, you can install [Node.js](#) directly, ensure you install the version as declared in the project's root `.nvmrc` file.

You will also need to install the `libjpeg` and `zlib` libraries, if you haven't done so already - see Pillow's [platform-specific installation instructions](#).

Clone a copy of the Wagtail codebase:

```
$ git clone https://github.com/wagtail/wagtail.git  
$ cd wagtail
```

With your preferred virtualenv activated, install the Wagtail package in development mode with the included testing and documentation dependencies:

```
$ pip install -e '[testing,docs]' -U
```

Install the tool chain for building static assets:

```
$ npm ci
```

Compile the assets:

```
$ npm run build
```

Any Wagtail sites you start up in this virtualenv will now run against this development instance of Wagtail. We recommend using the Wagtail Bakery demo site as a basis for developing Wagtail. Keep in mind that the setup steps for a Wagtail site may include installing a release version of Wagtail, which will override the development version you've just set up. In this case, you should install the site before running the `pip install -e` step, or re-run that step after the site is installed.

Testing

From the root of the Wagtail codebase, run the following command to run all the Python tests:

```
$ python runtests.py
```

Running only some of the tests

At the time of writing, Wagtail has well over 2500 tests, which takes a while to run. You can run tests for only one part of Wagtail by passing in the path as an argument to `runtests.py` or `tox`:

```
# Running in the current environment
$ python runtests.py wagtail

# Running in a specified Tox environment
$ tox -e py39-dj32-sqlite-noelasticsearch wagtail

# See a list of available Tox environments
$ tox -l
```

You can also run tests for individual TestCases by passing in the path as an argument to `runtests.py`

```
# Running in the current environment
$ python runtests.py wagtail.tests.test_blocks.TestIntegerBlock

# Running in a specified Tox environment
$ tox -e py39-dj32-sqlite-noelasticsearch wagtail.tests.test_blocks.TestIntegerBlock
```

Running migrations for the test app models

You can create migrations for the test app by running the following from the Wagtail root.

```
$ django-admin makemigrations --settings=wagtail.test.settings
```

Testing against PostgreSQL

Note: In order to run these tests, you must install the required modules for PostgreSQL as described in Django's [Databases documentation](#).

By default, Wagtail tests against SQLite. You can switch to using PostgreSQL by using the `--postgres` argument:

```
$ python runtests.py --postgres
```

If you need to use a different user, password, host or port, use the PGUSER, PGPASSWORD, PGHOST and PGPORT environment variables respectively.

Testing against a different database

Note: In order to run these tests, you must install the required client libraries and modules for the given database as described in Django's [Databases documentation](#) or 3rd-party database backend's documentation.

If you need to test against a different database, set the `DATABASE_ENGINE` environment variable to the name of the Django database backend to test against:

```
$ DATABASE_ENGINE=django.db.backends.mysql python runtests.py
```

This will create a new database called `test_wagtail` in MySQL and run the tests against it.

If you need to use different connection settings, use the following environment variables which correspond to the respective keys within Django's `DATABASES` settings dictionary:

- `DATABASE_ENGINE`
- `DATABASE_NAME`
- `DATABASE_PASSWORD`
- `DATABASE_HOST`
 - Note that for MySQL, this must be `127.0.0.1` rather than `localhost` if you need to connect using a TCP socket
- `DATABASE_PORT`

It is also possible to set `DATABASE_DRIVER`, which corresponds to the `driver` value within `OPTIONS` if an SQL Server engine is used.

Testing Elasticsearch

You can test Wagtail against Elasticsearch by passing the `--elasticsearch` argument to `runtests.py`:

```
$ python runtests.py --elasticsearch
```

Wagtail will attempt to connect to a local instance of Elasticsearch (`http://localhost:9200`) and use the index `test_wagtail`.

If your Elasticsearch instance is located somewhere else, you can set the `ELASTICSEARCH_URL` environment variable to point to its location:

```
$ ELASTICSEARCH_URL=http://my-elasticsearch-instance:9200 python runtests.py --  
  elasticsearch
```

Unit tests for JavaScript

We use [Jest](#) for unit tests of client-side business logic or UI components. From the root of the Wagtail codebase, run the following command to run all the front-end unit tests:

```
$ npm run test:unit
```

Integration tests

Our end-to-end browser testing suite also uses [Jest](#), combined with [Puppeteer](#). We set this up to be installed separately so as not to increase the installation size of the existing Node tooling. To run the tests, you will need to install the dependencies and, in a separate terminal, run the test suite's Django development server:

```
$ export DJANGO_SETTINGS_MODULE=wagtail.test.settings_ui
# Assumes the current environment contains a valid installation of Wagtail for local development.
$ ./wagtail/test/manage.py migrate
$ ./wagtail/test/manage.py createcachetable
$ DJANGO_SUPERUSER_EMAIL=admin@example.com DJANGO_SUPERUSER_USERNAME=admin DJANGO_SUPERUSER_PASSWORD=changeme ./wagtail/test/manage.py createsuperuser --noinput
$ ./wagtail/test/manage.py runserver 0:8000
# In a separate terminal:
$ npm --prefix client/tests/integration install
$ npm run test:integration
```

Integration tests target <http://localhost:8000> by default. Use the TEST_ORIGIN environment variable to use a different port, or test a remote Wagtail instance: TEST_ORIGIN=http://localhost:9000 npm run test:integration.

Browser and device support

Wagtail is meant to be used on a wide variety of devices and browsers. Supported browser / device versions include:

Browser	Device/OS	Version(s)
Mobile Safari	iOS Phone	Last 2
Mobile Safari	iOS Tablet	Last 2
Chrome	Android	Last 2
Chrome	Desktop	Last 2
MS Edge	Windows	Last 2
Firefox	Desktop	Latest
Firefox ESR	Desktop	Latest
Safari	macOS	Last 3

We aim for Wagtail to work in those environments, there are known support gaps for Safari 13 introduced in Wagtail 4.0 to provide better support for RTL languages. Our development standards ensure that the site is usable on other browsers **and will work on future browsers**.

IE 11 support has been officially dropped in 2.15 as it is gradually falling out of use. Features already known not to work include:

- Rich text copy-paste in the rich text editor.

- Sticky toolbar in the rich text editor.
- Focus outline styles in the main menu & explorer menu.
- Keyboard access to the actions in page listing tables.

Unsupported browsers / devices include:

Browser	Device/OS	Version(s)
Stock browser	Android	All
IE	Desktop	All
Safari	Windows	All

Accessibility targets

We want to make Wagtail accessible for users of a wide variety of assistive technologies. The specific standard we aim for is [WCAG2.1](#), AA level. Here are specific assistive technologies we aim to test for, and ultimately support:

- NVDA on Windows with Firefox ESR
- VoiceOver on macOS with Safari
- Windows Magnifier and macOS Zoom
- Windows Speech Recognition and macOS Dictation
- Mobile VoiceOver on iOS, or [TalkBack](#) on Android
- Windows [High-contrast mode](#)

We aim for Wagtail to work in those environments. Our development standards ensure that the site is usable with other assistive technologies. In practice, testing with assistive technology can be a daunting task that requires specialised training – here are tools we rely on to help identify accessibility issues, to use during development and code reviews:

- [react-axe](#) integrated directly in our build tools, to identify actionable issues. Logs its results in the browser console.
- [@wordpress/jest-puppeteer-axe](#) running Axe checks as part of integration tests.
- [Axe Chrome extension](#) for more comprehensive automated tests of a given page.
- [Accessibility Insights for Web](#) Chrome extension for semi-automated tests, and manual audits.

Known accessibility issues

Wagtail's administration interface isn't fully accessible at the moment. We actively work on fixing issues both as part of ongoing maintenance and bigger overhauls. To learn about known issues, check out:

- The [WCAG2.1 AA for CMS admin](#) issues backlog.
- Our [2021 accessibility audit](#).

The audit also states which parts of Wagtail have and haven't been tested, how issues affect WCAG 2.1 compliance, and the likely impact on users.

Compiling static assets

All static assets such as JavaScript, CSS, images, and fonts for the Wagtail admin are compiled from their respective sources by Webpack. The compiled assets are not committed to the repository, and are compiled before packaging each new release. Compiled assets should not be submitted as part of a pull request.

To compile the assets, run:

```
$ npm run build
```

This must be done after every change to the source files. To watch the source files for changes and then automatically recompile the assets, run:

```
$ npm start
```

Using the pattern library

Wagtail's UI component library is built with Storybook and [django-pattern-library](#). To run it locally,

```
$ export DJANGO_SETTINGS_MODULE=wagtail.test.settings_ui
# Assumes the current environment contains a valid installation of Wagtail for local development.
$ ./wagtail/test/manage.py migrate
$ ./wagtail/test/manage.py createcachetable
$ ./wagtail/test/manage.py runserver 0:8000
# In a separate terminal:
$ npm run storybook
```

The last command will start Storybook at `http://localhost:6006/`. It will proxy specific requests to Django at `http://localhost:8000` by default. Use the `TEST_ORIGIN` environment variable to use a different port for Django: `TEST_ORIGIN=http://localhost:9000` `npm run storybook`.

Compiling the documentation

The Wagtail documentation is built by Sphinx. To install Sphinx and compile the documentation, run:

```
$ cd /path/to/wagtail
# Install the documentation dependencies
$ pip install -e .[docs]
# or if using zsh as your shell:
#   pip install -e '.[docs]' -U
# Compile the docs
$ cd docs/
$ make html
```

The compiled documentation will now be in `docs/_build/html`. Open this directory in a web browser to see it. Python comes with a module that makes it very easy to preview static files in a web browser. To start this simple server, run the following commands:

```
$ cd docs/_build/html/
$ python -m http.server 8080
```

Now you can open <http://localhost:8080/> in your web browser to see the compiled documentation.

Sphinx caches the built documentation to speed up subsequent compilations. Unfortunately, this cache also hides any warnings thrown by unmodified documentation source files. To clear the built HTML and start fresh, so you can see all warnings thrown when building the documentation, run:

```
$ cd docs/  
$ make clean  
$ make html
```

Wagtail also provides a way for documentation to be compiled automatically on each change. To do this, you can run the following command to see the changes automatically at `localhost:4000`:

```
$ cd docs/  
$ make livehtml
```

Automatically lint and code format on commits

`pre-commit` is configured to automatically run code linting and formatting checks with every commit. To install `pre-commit` into your git hooks run:

```
$ pre-commit install
```

`pre-commit` should now run on every commit you make.

Committing code

This section is for the core team of Wagtail, or for anyone interested in the process of getting code committed to Wagtail.

Code should only be committed after it has been reviewed by at least one other reviewer or committer, unless the change is a small documentation change or fixing a typo. If additional code changes are made after the review, it is OK to commit them without further review if they are uncontroversial and small enough that there is minimal chance of introducing new bugs.

Most code contributions will be in the form of pull requests from Github. Pull requests should not be merged from Github, apart from small documentation fixes, which can be merged with the ‘Squash and merge’ option. Instead, the code should be checked out by a committer locally, the changes examined and rebased, the `CHANGELOG.txt` and release notes updated, and finally the code should be pushed to the `main` branch. This process is covered in more detail below.

Check out the code locally

If the code has been submitted as a pull request, you should fetch the changes and check them out in your Wagtail repository. A simple way to do this is by adding the following `git` alias to your `~/.gitconfig` (assuming `upstream` is `wagtail/wagtail`):

```
[alias]  
pr = !sh -c \"git fetch upstream pull/${1}/head:pr/${1} && git checkout pr/${1}\\"
```

Now you can check out pull request number `xxxx` by running `git pr xxxx`.

Rebase on to main

Now that you have the code, you should rebase the commits on to the `main` branch. Rebasing is preferred over merging, as merge commits make the commit history harder to read for small changes.

You can fix up any small mistakes in the commits, such as typos and formatting, as part of the rebase. `git rebase --interactive` is an excellent tool for this job.

Ideally, use this as an opportunity to squash the changes to a few commits, so each commit is making a single meaningful change (and not breaking anything). If this is not possible because of the nature of the changes, it's acceptable to either squash into a commit or leave all commits unsquashed, depending on which will be more readable in the commit history.

```
# Get the latest commits from Wagtail
git fetch upstream
git checkout main
git merge --ff-only upstream/main
# Rebase this pull request on to main
git checkout pr/xxxx
git rebase main
# Update main to this commit
git checkout main
git merge --ff-only pr/xxxx
```

Update CHANGELOG.txt and release notes

Note: This should only be done by core committers, once the changes have been reviewed and accepted.

Every significant change to Wagtail should get an entry in the `CHANGELOG.txt`, and the release notes for the current version.

The `CHANGELOG.txt` contains a short summary of each new feature, refactoring, or bug fix in each release. Each summary should be a single line. Bug fixes should be grouped together at the end of the list for each release, and be prefixed with “Fix:”. The name of the contributor should be added at the end of the summary, in brackets. For example:

```
* Fix: Tags added on the multiple image uploader are now saved correctly (Alex Smith)
```

The release notes for each version contain a more detailed description of each change. Backwards compatibility notes should also be included. Large new features or changes should get their own section, while smaller changes and bug fixes should be grouped together in their own section. See previous release notes for examples. The release notes for each version are found in `docs/releases/x.x.x.md`.

If the contributor is a new person, and this is their first contribution to Wagtail, they should be added to the `CONTRIBUTORS.rst` list. Contributors are added in chronological order, with new contributors added to the bottom of the list. Use their preferred name. You can usually find the name of a contributor on their Github profile. If in doubt, or if their name is not on their profile, ask them how they want to be named.

If the changes to be merged are small enough to be a single commit, amend this single commit with the additions to the `CHANGELOG.txt`, release notes, and contributors:

```
git add CHANGELOG.txt docs/releases/x.x.x.md CONTRIBUTORS.md
git commit --amend --no-edit
```

If the changes do not fit in a single commit, make a new commit with the updates to the `CHANGELOG.txt`, release notes, and contributors. The commit message should say `Release notes for #xxxx`:

```
git add CHANGELOG.txt docs/releases/x.x.x.md CONTRIBUTORS.md
git commit -m 'Release notes for #xxxx'
```

Push to main

The changes are ready to be pushed to `main` now.

```
# Check that everything looks OK
git log upstream/main..main --oneline
git push --dry-run upstream main
# Push the commits!
git push upstream main
git branch -d pr/xxxx
```

When you have made a mistake

It's ok! Everyone makes mistakes. If you realise that recent merged changes have a negative impact, create a new pull request with a revert of the changes and merge it without waiting for a review. The PR will serve as additional documentation for the changes, and will run through the CI tests.

Add commits to someone else's pull request

Github users with write access to `wagtail/wagtail` (core members) can add commits to the pull request branch of the contributor.

Given that the contributor username is `johndoe` and his pull request branch is called `foo`:

```
git clone git@github.com:wagtail/wagtail.git
cd wagtail
git remote add johndoe git@github.com:johndoe/wagtail.git
git fetch johndoe foo
git checkout johndoe/foo
# Make changes
# Commit changes
git push johndoe HEAD:foo
```

1.8.3 Translations

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

Translation work should be submitted through [Transifex](#).

1.8.4 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.

1.8.5 Developing packages for Wagtail

If you are developing packages for Wagtail, you can add the following PyPI classifiers:

- `Framework :: Wagtail`
- `Framework :: Wagtail :: 1`
- `Framework :: Wagtail :: 2`
- `Framework :: Wagtail :: 3`
- `Framework :: Wagtail :: 4`

You can also find a curated list of awesome packages, articles, and other cool resources from the Wagtail community at [Awesome Wagtail](#).

1.8.6 More information

UI Styleguide

Developers working on the Wagtail UI or creating new UI components may wish to test their work against our Styleguide, which is provided as the contrib module “`wagtailstyleguide`”.

To install the styleguide module on your site, add it to the list of `INSTALLED_APPS` in your settings:

```
INSTALLED_APPS = (
    # ...
    'wagtail.contrib.styleguide',
    # ...
)
```

This will add a ‘Styleguide’ item to the Settings menu in the admin.

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.

General coding guidelines

Language

British English is preferred for user-facing text; this text should also be marked for translation (using the `django.utils.translation.gettext` function and `{% trans %}` template tag, for example). However, identifiers within code should use American English if the British or international spelling would conflict with built-in language keywords; for example, CSS code should consistently use the spelling `color` to avoid inconsistencies like `background-color: $colour-red.`

Latin phrases and abbreviations

Try to avoid Latin phrases (such as `ergo` or `de facto`) and abbreviations (such as `i.e.` or `e.g.`), and use common English phrases instead. Alternatively find a simpler way to communicate the concept or idea to the reader. The exception is `etc.` which can be used when space is limited.

Examples:

Don't use this	Use this instead
<code>e.g.</code>	for example, such as
<code>i.e.</code>	that is
<code>viz.</code>	namely
<code>ergo</code>	therefore

File names

Where practical, try to adhere to the existing convention of file names within the folder where added.

Examples:

- Django templates - `lower_snake_case.html`
- Documentation - `lower_snake_case.md`

Python coding guidelines

PEP8

We ask that all Python contributions adhere to the [PEP8](#) style guide. All files should be formatted using the `black` auto-formatter. This will be run by `pre-commit` if that is configured.

- The project repository includes an `.editorconfig` file. We recommend using a text editor with [EditorConfig](#) support to avoid indentation and whitespace issues. Python and HTML files use 4 spaces for indentation.

In addition, import lines should be sorted according to `isort` 5.6.4 rules. If you have installed Wagtail's testing dependencies (`pip install -e '[testing]'`), you can check your code by running `make lint`. You can also just check python related linting by running `make lint-server`.

You can run all Python formatting with `make format`. Similar to linting you can format python/template only files by running `make format-server`.

Django compatibility

Wagtail is written to be compatible with multiple versions of Django. Sometimes, this requires running one piece of code for recent version of Django, and another piece of code for older versions of Django. In these cases, always check which version of Django is being used by inspecting `django.VERSION`:

```
import django

if django.VERSION >= (1, 9):
    # Use new attribute
    related_field = field.rel
else:
```

(continues on next page)

(continued from previous page)

```
# Use old, deprecated attribute
related_field = field.related
```

Always compare against the version using greater-or-equals (\geq), so that code for newer versions of Django is first.

Do not use a `try ... except` when seeing if an object has an attribute or method introduced in a newer versions of Django, as it does not clearly express why the `try ... except` is used. An explicit check against the Django version makes the intention of the code very clear.

```
# Do not do this
try:
    related_field = field.rel
except AttributeError:
    related_field = field.related
```

If the code needs to use something that changed in a version of Django many times, consider making a function that encapsulates the check:

```
import django

def related_field(field):
    if django.VERSION >= (1, 9):
        return field.rel
    else:
        return field.related
```

If a new function has been introduced by Django that you think would be very useful for Wagtail, but is not available in older versions of Django that Wagtail supports, that function can be copied over in to Wagtail. If the user is running a new version of Django that has the function, the function should be imported from Django. Otherwise, the version bundled with Wagtail should be used. A link to the Django source code where this function was taken from should be included:

```
import django

if django.VERSION >= (1, 9):
    from django.core.validators import validate_unicode_slug
else:
    # Taken from https://github.com/django/django/blob/1.9/django/core/validators.py#L230
    def validate_unicode_slug(value):
        # Code left as an exercise to the reader
        pass
```

Tests

Wagtail has a suite of tests, which we are committed to improving and expanding. See [Testing](#).

We run continuous integration 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.

UI Guidelines

Wagtail's user interface is built with:

- **HTML** using [Django templates](#)
- **CSS** using [Sass](#) and [Tailwind](#)
- **JavaScript** with [TypeScript](#)
- **SVG** for our icons, minified with [SVGO](#)

Linting and formatting

Here are the available commands:

- `make lint` will run all linting, `make lint-server` lints templates, `make lint-client` lints JS/CSS.
- `make format` will run all formatting and fixing of linting issues. There is also `make format-server` and `make format-client`.

Have a look at our `Makefile` tasks and `package.json` scripts if you prefer more granular options.

HTML guidelines

We use [djhtml](#) for formatting and [Curlylint](#) for linting.

- Write valid, semantic HTML.
- Follow ARIA authoring practices, in particular [No ARIA](#) is better than Bad ARIA.
- Use classes for styling, `data-` attributes for JavaScript behaviour, IDs for semantics only.
- For comments, use [Django template syntax](#) instead of HTML.

CSS guidelines

We use [Prettier](#) for formatting and [Stylelint](#) for linting.

- We follow [BEM](#) and [ITCSS](#), with a large amount of utilities created with [Tailwind](#).
- Familiarise yourself with our `stylelint-config-wagtail` configuration, which details our preferred code style.
- Use `rems` for `font-size`, because they offer absolute control over text. Additionally, unit-less `line-height` is preferred because it does not inherit a percentage value of its parent element, but instead is based on a multiplier of the `font-size`.
- Always use variables for design tokens such as colours or font sizes, rather than hard-coding specific values.
- We use the `w-` prefix for all styles intended to be reusable by Wagtail site implementers.

JavaScript guidelines

We use [Prettier](#) for formatting and [ESLint](#) for linting.

- We follow a somewhat relaxed version of the [Airbnb styleguide](#).
- Familiarise yourself with our [eslint-config-wagtail](#) configuration, which details our preferred code style.

Multilingual support

This is an area of active improvement for Wagtail, with ongoing discussions.

- Always use the `trimmed` attribute on `blocktrans` tags to prevent unnecessary whitespace from being added to the translation strings.

Documentation guidelines

- *Formatting recommendations*
- *Formatting to avoid*

Formatting recommendations

Wagtail's documentation uses a mixture of [Markdown](#) and [reStructuredText](#). We encourage writing documentation in Markdown first, and only reaching for more advanced reStructuredText formatting if there is a compelling reason.

Here are formats we encourage using when writing documentation for Wagtail.

Paragraphs

It all starts here. Keep your sentences short, varied in length.

Separate text with an empty line to create a new paragraph.

Heading levels

Use heading levels to create sections, and allow users to link straight to a specific section. Start documents with an `# h1`, and proceed with `## h2` and further sub-sections without skipping levels.

```
# Heading level 1  
## Heading level 2  
### Heading level 3
```

Lists

Use bullets for unordered lists, numbers when ordered. Prefer dashes – for bullets. Nest by indenting with 4 spaces.

```
- Bullet 1
- Bullet 2
  - Nested bullet 2
- Bullet 3

1. Numbered list 1
2. Numbered list 2
3. Numbered list 3
```

- Bullet 1
 - Bullet 2
 - Nested bullet 2
 - Bullet 3
1. Numbered list 1
 2. Numbered list 2
 3. Numbered list 3

Inline styles

Use **bold** and *italic* sparingly, inline code when relevant.

```
Use **bold** and _italic_ sparingly, inline `code` when relevant.
```

Code blocks

Make sure to include the correct language code for syntax highlighting, and to format your code according to our coding guidelines. Frequently used: `python`, `css`, `html`, `html+django`, `javascript`, `console`.

```
```python
INSTALLED_APPS = [
 ...
 "wagtail",
 ...
]
```
```

Links

Links are fundamental in documentation. Use internal links to tie your content to other docs, and external links as needed. Pick relevant text for links, so readers know where they will land.

Don't rely on `links over code`, as they are impossible to spot.

```
An [external link](https://www.example.com).  
An [internal link to another document](/reference/contrib/legacy_richtext).  
An auto generated link label to a page [](/getting_started/tutorial).  
A [link to a reference](register_reports_menu_item).
```

An external link. An *internal link to another document*. An auto generated link label to a page *Your first Wagtail site*. A *link to a reference*.

Reference links

Reference links (links to a target within a page) rely on the page having a reference created. Each reference must have a unique name and should use the `lower_snake_case` format. A reference can be added as follows:

```
(my_awesome_section)=  
  
##### Some awesome section title  
  
...
```

The reference can be linked to, with an optional label, using the Markdown link syntax as follows:

- Auto generated label (preferred) [](`my_awesome_section`)
- `[label for section]` (`my_awesome_section`)

Some awesome section title

...

- Auto generated label (preferred) *Some awesome section title*
- *label for section*

You can read more about other methods of linking to, and creating references in the MyST parser docs section on Targets and cross-referencing.

Note and warning call-outs

Use notes and warnings sparingly, as they rely on reStructuredText syntax which is more complicated for future editors.

```
```{note}  
Notes can provide complementary information.
```\n\n```{warning}  
Warnings can be scary.  
```
```

---

**Note:** Notes can provide complementary information.

---

**Warning:** Warnings can be scary.

These call-outs do not support titles, so be careful not to include them, titles will just be moved to the body of the call-out.

```
```{note} Title's here will not work correctly
Notes can provide complementary information.
```
```

## Images

Images are hard to keep up-to-date as documentation evolves, but can be worthwhile nonetheless. Here are guidelines when adding images:

- All images should have meaningful `alt` text unless they are decorative.
- Images are served as-is – pick the correct format, and losslessly compress all images.
- Use absolute paths for image files so they are more portable.

`![Screenshot of the workflow editing interface, with fields to change the workflow name, ↵ tasks, and assigned pages](_static/images/screen44_workflow_edit.png)`

## ☰ Editing workflow Moderators approval

▼ Give your workflow a name \*

Moderators approval

✚ Add tasks to your workflow

▼ Task

Task \*



Moderators approval

...

Add task

## ─ Assign your workflow to pages

?

Workflows apply to child pages too. If you select a parent page here, its child pages will also use this workflow. See [the list of the pages](#) your workflow applies to.

### Assigned pages

Page



Root

Delete

Save

^

## Autodoc

With its `autodoc` feature, Sphinx supports writing documentation in Python docstrings for subsequent integration in the project's documentation pages. This is a very powerful feature which we highly recommend using to document Wagtail's APIs.

```
```{eval-rst}
.. module:: wagtail.coreutils

.. autofunction:: cautious_slugify
```
```

`wagtail.coreutils.cautious_slugify(value)`

Convert a string to ASCII exactly as Django's `slugify` does, with the exception that any non-ASCII alphanumeric characters (that cannot be ASCIIified under Unicode normalisation) are escaped into codes like '`u0421`' instead of being deleted entirely.

This ensures that the result of slugifying (for example - Cyrillic) text will not be an empty string, and can thus be safely used as an identifier (albeit not a human-readable one).

## Tables

Only use tables when needed, using the GitHub Flavored Markdown table syntax.

| Browser       | Device/OS |
|---------------|-----------|
| Stock browser | Android   |
| IE            | Desktop   |
| Safari        | Windows   |

| Browser       | Device/OS |
|---------------|-----------|
| Stock browser | Android   |
| IE            | Desktop   |
| Safari        | Windows   |

## Tables of contents

`toctree` and `contents` can be used as reStructuredText directives.

```
```{toctree}
---
maxdepth: 2
titlesonly:
---
getting_started/index
topics/index
```

```{contents}
---
local:
depth: 1
---
```
```

## Version added, changed, deprecations

Sphinx offers release-metadata directives to generate this information consistently. Use as appropriate.

```
```{versionadded} 2.15
```

```{versionchanged} 2.15
```
```

New in version 2.15.

Changed in version 2.15.

## Progressive disclosure

We can add supplementary information in documentation with the HTML `<details>` element. This relies on HTML syntax, which can be hard to author consistently, so keep this type of formatting to a minimum.

```
<details>
 <summary>Supplementary information</summary>

 This will be visible when expanding the content.
</details>
```

Example:

This will be visible when expanding the content.

## Formatting to avoid

There is some formatting in the documentation which is technically supported, but we recommend avoiding unless there is a clear necessity.

### Call-outs

We only use `{note}` and `{warning}` call-outs. Avoid `{admonition}`, `{important}`, `{topic}`, and `{tip}`. If you find one of these, please replace it with `{note}`.

### Glossary

Sphinx glossaries (`.. glossary::`) generate definition lists. Use plain bullet or number lists instead, or sections with headings, or a table.

### Comments

Avoid documentation source comments in committed documentation.

### Figure

reStructuredText figures (`.. figure::`) only offer very marginal improvements over vanilla images. If your figure has a caption, add it as an italicised paragraph underneath the image.

### Other reStructuredText syntax and Sphinx directives

We generally want to favour Markdown over reStructuredText, to make it as simple as possible for newcomers to make documentation contributions to Wagtail. Always prefer Markdown, unless the document's formatting highly depends on reStructuredText syntax.

If you want to use a specific Sphinx directive, consult with core contributors to see whether its usage is justified, and document its expected usage on this page.

## Arbitrary HTML

While our documentation tooling offers some support for embedding arbitrary HTML, this is frowned upon. Only do so if there is a necessity, and if the formatting is unlikely to need updates.

## Writing documentation

Wagtail documentation is written in **four modes** of information delivery. Each type of information delivery has a purpose and targets a specific audience.

- *Tutorial*, learning-oriented
- *How-to guide*, goal-oriented
- *Reference*, information-oriented
- *Explanation*, understanding-oriented

We are following Daniele Procida's Diátaxis documentation framework.

### Choose a writing mode

Each page of the Wagtail documentation should be written in single mode of information delivery. Single pages with mixed modes are harder to understand. If you have documents that mix the types of information delivery, it's best to split them up. Add links to the first section of each document to cross reference other documents on the same topic.

Writing documentation in a specific mode will help our users to understand and quickly find what they are looking for.

### Tutorial

Tutorials are designed to be **learning-oriented** resources which guide newcomers through a specific topic. To help effective learning, tutorials should provide examples to illustrate the subjects they cover.

Tutorials may not necessarily follow best practices. They are designed to make it easier to get started. A tutorial is concrete and particular. It must be repeatable, instil confidence, and should result in success, every time, for every learner.

### Do

- Use conversational language
- Use contractions, speak in the first person plural, be reassuring. For example: “We’re going to do this.”
- Use pictures or concrete outputs of code to reassure people that they’re on the right track. For example: “Your new login page should look like this:” or “Your directory should now have three files”.

## Don't

- Tell people what they're going to learn. Instead, tell them what tasks they're going to complete.
- Use optionality in a tutorial. The word 'if' is a sign of danger! For example: "If you want to do this..." The expected actions and outcomes should be unambiguous.
- Assume that learners have a prior understanding of the subject.

[More about tutorials](#)

## How-to guide

A guide offers advice on how best to achieve a given task. How-to guides are **task-oriented** with a clear **goal or objective**.

## Do

- Name the guide well - ensure that the learner understands what exactly the guide does.
- Focus on actions and outcomes. For example: "If you do X, Y should happen."
- Assume that the learner has a basic understanding of the general concepts
- Point the reader to additional resources

## Don't

- Use an unnecessarily strict tone of voice. For example: "You must absolutely NOT do X."

[More about how-to guides](#)

## Reference

Reference material is **information-oriented**. A reference is well-structured and allows the reader to find information about a specific topic. They should be short and to the point. Boring is fine! Use an imperative voice. For example: "Inherit from the Page model".

Most references will be auto-generated based on doc-strings in the Python code.

[More about reference](#)

## Explanation

Explanations are **understanding-oriented**. They are high-level and offer context to concepts and design decisions. There is little or no code involved in explanations, which are used to deepen the theoretical understanding of a practical draft. Explanations are used to establish connections and may require some prior knowledge of the principles being explored.

[More about explanation](#)

## Reporting security issues

---

**Note:** Please report security issues **only** to [security@wagtail.org](mailto:security@wagtail.org).

---

Most normal bugs in Wagtail are reported as [GitHub issues](#), but due to the sensitive nature of security issues, we ask that they not be publicly reported in this fashion.

Instead, if you believe you've found something in Wagtail which has security implications, please send a description of the issue via email to [security@wagtail.org](mailto:security@wagtail.org). Mail sent to that address reaches a subset of the core team, who can forward security issues to other core team members for broader discussion if needed.

Once you've submitted an issue via email, you should receive an acknowledgement from a member of the security team within 48 hours, and depending on the action to be taken, you may receive further followup emails.

If you want to send an encrypted email (optional), the public key ID for [security@wagtail.org](mailto:security@wagtail.org) is `0xbed227b4daf93ff9`, and this public key is available from most commonly-used keyservers.

Django security issues should be reported directly to the Django Project, following [Django's security policies](#) (upon which Wagtail's own policies are based).

## Supported versions

At any given time, the Wagtail team provides official security support for several versions of Wagtail:

- The main development branch, hosted on GitHub, which will become the next release of Wagtail, receives security support.
- The two most recent Wagtail release series receive security support. For example, during the development cycle leading to the release of Wagtail 2.6, support will be provided for Wagtail 2.5 and Wagtail 2.4. Upon the release of Wagtail 2.6, Wagtail 2.4's security support will end.
- The latest long-term support release will receive security updates.

When new releases are issued for security reasons, the accompanying notice will include a list of affected versions. This list is comprised solely of supported versions of Wagtail: older versions may also be affected, but we do not investigate to determine that, and will not issue patches or new releases for those versions.

## How Wagtail discloses security issues

Our process for taking a security issue from private discussion to public disclosure involves multiple steps.

There is no fixed period of time by which a confirmed security issue will be resolved as this is dependent on the issue, however it will be a priority of the Wagtail team to issue a security release as soon as possible.

The reporter of the issue will receive notification of the date on which we plan to take the issue public. On the day of disclosure, we will take the following steps:

1. Apply the relevant patch(es) to Wagtail's codebase. The commit messages for these patches will indicate that they are for security issues, but will not describe the issue in any detail; instead, they will warn of upcoming disclosure.
2. Issue the relevant release(s), by placing new packages on [the Python Package Index](#), tagging the new release(s) in Wagtail's GitHub repository and updating Wagtail's [release notes](#).
3. Post a public entry on [Wagtail's blog](#), describing the issue and its resolution in detail, pointing to the relevant patches and new releases, and crediting the reporter of the issue (if the reporter wishes to be publicly identified).

4. Post a notice to the [Wagtail support forum](#) and Twitter feed (@WagtailCMS) that links to the blog post.

If a reported issue is believed to be particularly time-sensitive – due to a known exploit in the wild, for example – the time between advance notification and public disclosure may be shortened considerably.

## **CSV export security considerations**

In various places Wagtail provides the option to export data in CSV format, and several reporters have raised the possibility of a malicious user inserting data that will be interpreted as a formula when loaded into a spreadsheet package such as Microsoft Excel. We do not consider this to be a security vulnerability in Wagtail. CSV as defined by [RFC 4180](#) is purely a data format, and makes no assertions about how that data is to be interpreted; the decision made by certain software to treat some strings as executable code has no basis in the specification. As such, Wagtail cannot be responsible for the data it generates being loaded into a software package that interprets it insecurely, any more than it would be responsible for its data being loaded into a missile control system. This is consistent with [the Google security team's position](#).

Since the CSV format has no concept of formulae or macros, there is also no agreed-upon convention for escaping data to prevent it from being interpreted in that way; commonly-suggested approaches such as prefixing the field with a quote character would corrupt legitimate data (such as phone numbers beginning with '+') when interpreted by software correctly following the CSV specification.

Wagtail's data exports default to XLSX, which can be loaded into spreadsheet software without any such issues. This minimises the risk of a user handling CSV files insecurely, as they would have to explicitly choose CSV over the more familiar XLSX format.

## **Wagtail's release process**

### **Official releases**

Release numbering works as follows:

- Versions are numbered in the form A.B or A.B.C.
- A.B is the *feature release* version number. Each version will be mostly backwards compatible with the previous release. Exceptions to this rule will be listed in the release notes.
- C is the *patch release* version number, which is incremented for bugfix and security releases. These releases will be 100% backwards-compatible with the previous patch release. The only exception is when a security or data loss issue can't be fixed without breaking backwards-compatibility. If this happens, the release notes will provide detailed upgrade instructions.
- Before a new feature release, we'll make at least one release candidate release. These are of the form A.BrcN, which means the Nth release candidate of version A.B.

In git, each Wagtail release will have a tag indicating its version number. Additionally, each release series has its own branch, called `stable/A.B.x`, and bugfix/security releases will be issued from those branches.

#### **Feature release**

Feature releases (A.B, A.B+1, etc.) happen every three months – see [release schedule](#) for details. These releases will contain new features and improvements to existing features.

#### **Patch release**

Patch releases (A.B.C, A.B.C+1, etc.) will be issued as needed, to fix bugs and/or security issues.

These releases will be 100% compatible with the associated feature release, unless this is impossible for security reasons or to prevent data loss. So the answer to “should I upgrade to the latest patch release?” will always be “yes.”

## Long-term support release

Certain feature releases will be designated as long-term support (LTS) releases. These releases will get security and data loss fixes applied for a guaranteed period of time, typically six months.

## Release cadence

Wagtail uses a loose form of [semantic versioning](#). SemVer makes it easier to see at a glance how compatible releases are with each other. It also helps to anticipate when compatibility shims will be removed. It's not a pure form of SemVer as each feature release will continue to have a few documented backwards incompatibilities where a deprecation path isn't possible or not worth the cost.

## Deprecation policy

A feature release may deprecate certain features from previous releases. If a feature is deprecated in feature release A.B, it will continue to work in the following version but raise warnings. Features deprecated in release A.B will be removed in the A.B+2 release to ensure deprecations are done over at least 2 feature releases.

So, for example, if we decided to start the deprecation of a function in Wagtail 1.4:

- Wagtail 1.4 will contain a backwards-compatible replica of the function which will raise a `RemovedInWagtail16Warning`.
- Wagtail 1.5 will still contain the backwards-compatible replica.
- Wagtail 1.6 will remove the feature outright.

The warnings are silent by default. You can turn on display of these warnings with the `python -Wd` option.

## Supported versions

At any moment in time, Wagtail's developer team will support a set of releases to varying levels.

- The current development `main` will get new features and bug fixes requiring non-trivial refactoring.
- Patches applied to the `main` branch must also be applied to the last feature release branch, to be released in the next patch release of that feature series, when they fix critical problems:
  - Security issues.
  - Data loss bugs.
  - Crashing bugs.
  - Major functionality bugs in newly-introduced features.
  - Regressions from older versions of Wagtail.

The rule of thumb is that fixes will be backported to the last feature release for bugs that would have prevented a release in the first place (release blockers).

- Security fixes and data loss bugs will be applied to the current `main`, the last feature release branch, and any other supported long-term support release branches.
- Documentation fixes generally will be more freely backported to the last release branch. That's because it's highly advantageous to have the docs for the last release be up-to-date and correct, and the risk of introducing regressions is much less of a concern.

As a concrete example, consider a moment in time halfway between the release of Wagtail 1.6 and 1.7. At this point in time:

- Features will be added to `main`, to be released as Wagtail 1.7.
- Critical bug fixes will be applied to the `stable/1.6.x` branch, and released as 1.6.1, 1.6.2, etc.
- Security fixes and bug fixes for data loss issues will be applied to `main` and to the `stable/1.6.x` and `stable/1.4.x` (LTS) branches. They will trigger the release of 1.6.1, 1.4.8, etc.
- Documentation fixes will be applied to `main`, and, if easily backported, to the latest stable branch, `1.6.x`.

### Supported versions of Django

Each release of Wagtail declares which versions of Django it supports.

Typically, a new Wagtail feature release supports the last long-term support version and all following versions of Django.

For example, consider a moment in time before release of Wagtail 1.5 and after the following releases:

- Django 1.8 (LTS)
- Django 1.9
- Wagtail 1.4 (LTS) - Released before Django 1.10 and supports Django 1.8 and 1.9
- Django 1.10

Wagtail 1.5 will support Django 1.8 (LTS), 1.9, 1.10. Wagtail 1.4 will still support only Django 1.8 (LTS) and 1.9.

### Release schedule

Wagtail uses a [time-based release schedule](#), with feature releases every three months.

After each feature release, the release manager will announce a timeline for the next feature release.

### Release cycle

Each release cycle consists of three parts:

#### Phase one: feature proposal

The first phase of the release process will include figuring out what major features to include in the next version. This should include a good deal of preliminary work on those features – working code trumps grand design.

#### Phase two: development

The second part of the release schedule is the “heads-down” working period. Using the roadmap produced at the end of phase one, we’ll all work very hard to get everything on it done.

At the end of phase two, any unfinished features will be postponed until the next release.

At this point, the `stable/A.B.x` branch will be forked from `main`.

## Phase three: bugfixes

The last part of a release cycle is spent fixing bugs – no new features will be accepted during this time.

Once all known blocking bugs have been addressed, a release candidate will be made available for testing. The final release will usually follow two weeks later, although this period may be extended if further release blockers are found.

During this phase, committers will be more and more conservative with backports, to avoid introducing regressions. After the release candidate, only release blockers and documentation fixes should be backported.

Developers should avoid adding any new translatable strings after the release candidate - this ensures that translators have the full period between the release candidate and the final release to bring translations up to date. Translations will be re-imported immediately before the final release.

In parallel to this phase, `main` can receive new features, to be released in the A.B+1 cycle.

### Bug-fix releases

After a feature release A.B, the previous release will go into bugfix mode.

The branch for the previous feature release `stable/A.B-1.x` will include bugfixes. Critical bugs fixed on `main` must also be fixed on the bugfix branch; this means that commits need to cleanly separate bug fixes from feature additions. The developer who commits a fix to `main` will be responsible for also applying the fix to the current bugfix branch.

## 1.9 Release notes

### 1.9.1 Upgrading Wagtail

#### Version numbers

New feature releases of Wagtail are released every three months. These releases provide new features, improvements and bugfixes, and are marked by incrementing the second part of the version number (for example, 2.6 to 2.7).

Additionally, patch releases will be issued as needed, to fix bugs and security issues. These are marked by incrementing the third part of the version number (for example, 2.6 to 2.6.1). Wherever possible, these releases will remain fully backwards compatible with the corresponding feature and not introduce any breaking changes.

A feature release will usually stop receiving patch release updates when the next feature release comes out. However, selected feature releases are designated as Long Term Support (LTS) releases, and will continue to receive maintenance updates to address any security and data-loss related issues that arise. Typically, a Long Term Support release will happen once every four feature releases and receive updates for five feature releases, giving a support period of fifteen months with a three month overlap.

Also, Long Term Support releases will ensure compatibility with at least one Django Long Term Support release.

For dates of past and upcoming releases and support periods, see [Release Schedule](#).

## Deprecation policy

Sometimes it is necessary for a feature release to deprecate features from previous releases. This will be noted in the “Upgrade considerations” section of the release notes.

When a feature is deprecated, it will continue to work in that feature release and the one after it, but will raise a warning. The feature will then be removed in the subsequent feature release. For example, a feature marked as deprecated in version 1.8 will continue to work in versions 1.8 and 1.9, and be dropped in version 1.10.

## Upgrade process

We recommend upgrading one feature release at a time, even if your project is several versions behind the current one. This has a number of advantages over skipping directly to the newest release:

- If anything breaks as a result of the upgrade, you will know which version caused it, and will be able to troubleshoot accordingly;
- Deprecation warnings shown in the console output will notify you of any code changes you need to make before upgrading to the following version;
- Some releases make database schema changes that need to be reflected on your project by running `./manage.py makemigrations` - this is liable to fail if too many schema changes happen in one go.

Before upgrading to a new feature release:

- Check your project’s console output for any deprecation warnings, and fix them where necessary;
- Check the new version’s release notes, and the [Compatible Django / Python versions](#) table below, for any dependencies that need upgrading first;
- Make a backup of your database.

To upgrade:

- Update the `wagtail` line in your project’s `requirements.txt` file to specify the latest patch release of the version you wish to install. For example, to upgrade to version 1.8.x, the line should read:

```
wagtail>=1.8,<1.9
```

- Run:

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

- Make any necessary code changes as directed in the “Upgrade considerations” section of the release notes.
- Test that your project is working as expected.

Remember that the JavaScript and CSS files used in the Wagtail admin may have changed between releases - if you encounter erratic behaviour on upgrading, ensure that you have cleared your browser cache. When deploying the upgrade to a production server, be sure to run `./manage.py collectstatic` to make the updated static files available to the web server. In production, we recommend enabling `ManifestStaticFilesStorage` in the `STATICFILES_STORAGE` setting - this ensures that different versions of files are assigned distinct URLs.

## Compatible Django / Python versions

New feature releases frequently add support for newer versions of Django and Python, and drop support for older ones. We recommend always carrying out upgrades to Django and Python as a separate step from upgrading Wagtail.

The compatible versions of Django and Python for each Wagtail release are:

Wagtail release	Compatible Django versions	Compatible Python versions
4.1 LTS	3.2, 4.0, 4.1	3.7, 3.8, 3.9, 3.10
4.0	3.2, 4.0, 4.1	3.7, 3.8, 3.9, 3.10
3.0	3.2, 4.0	3.7, 3.8, 3.9, 3.10
2.16	3.2, 4.0	3.7, 3.8, 3.9, 3.10
2.15 LTS	3.0, 3.1, 3.2	3.6, 3.7, 3.8, 3.9, 3.10
2.14	3.0, 3.1, 3.2	3.6, 3.7, 3.8, 3.9
2.13	2.2, 3.0, 3.1, 3.2	3.6, 3.7, 3.8, 3.9
2.12	2.2, 3.0, 3.1	3.6, 3.7, 3.8, 3.9
2.11 LTS	2.2, 3.0, 3.1	3.6, 3.7, 3.8
2.10	2.2, 3.0, 3.1	3.6, 3.7, 3.8
2.9	2.2, 3.0	3.5, 3.6, 3.7, 3.8
2.8	2.1, 2.2, 3.0	3.5, 3.6, 3.7, 3.8
2.7 LTS	2.0, 2.1, 2.2	3.5, 3.6, 3.7, 3.8
2.6	2.0, 2.1, 2.2	3.5, 3.6, 3.7
2.5	2.0, 2.1, 2.2	3.4, 3.5, 3.6, 3.7
2.4	2.0, 2.1	3.4, 3.5, 3.6, 3.7
2.3 LTS	1.11, 2.0, 2.1	3.4, 3.5, 3.6
2.2	1.11, 2.0	3.4, 3.5, 3.6
2.1	1.11, 2.0	3.4, 3.5, 3.6
2.0	1.11, 2.0	3.4, 3.5, 3.6
1.13 LTS	1.8, 1.10, 1.11	2.7, 3.4, 3.5, 3.6
1.12 LTS	1.8, 1.10, 1.11	2.7, 3.4, 3.5, 3.6
1.11	1.8, 1.10, 1.11	2.7, 3.4, 3.5, 3.6
1.10	1.8, 1.10, 1.11	2.7, 3.4, 3.5, 3.6
1.9	1.8, 1.9, 1.10	2.7, 3.3, 3.4, 3.5
1.8 LTS	1.8, 1.9, 1.10	2.7, 3.3, 3.4, 3.5
1.7	1.8, 1.9, 1.10	2.7, 3.3, 3.4, 3.5
1.6	1.8, 1.9, 1.10	2.7, 3.3, 3.4, 3.5
1.5	1.8, 1.9	2.7, 3.3, 3.4, 3.5
1.4 LTS	1.8, 1.9	2.7, 3.3, 3.4, 3.5
1.3	1.7, 1.8, 1.9	2.7, 3.3, 3.4, 3.5
1.2	1.7, 1.8	2.7, 3.3, 3.4, 3.5
1.1	1.7, 1.8	2.7, 3.3, 3.4
1.0	1.7, 1.8	2.7, 3.3, 3.4
0.8 LTS	1.6, 1.7	2.6, 2.7, 3.2, 3.3, 3.4
0.7	1.6, 1.7	2.6, 2.7, 3.2, 3.3, 3.4
0.6	1.6, 1.7	2.6, 2.7, 3.2, 3.3, 3.4
0.5	1.6	2.6, 2.7, 3.2, 3.3, 3.4
0.4	1.6	2.6, 2.7, 3.2, 3.3, 3.4
0.3	1.6	2.6, 2.7
0.2	1.6	2.7
0.1	1.6	2.7

## 1.9.2 Wagtail 4.1 release notes - IN DEVELOPMENT

- [What's new](#)
- [Upgrade considerations](#)

Wagtail 4.1 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 12 months).

### What's new

### Other features

- Add basic keyboard control and screen reader support for page listing re-ordering (Paarth Agarwal, Thomas van der Hoeven)
- Add `PageQuerySet.private` method as an alias of `not_public` (Mehrdad Moradizadeh)
- Most images in the admin will now only load once they are visible on screen (Jake Howard)
- Allow setting default attributes on image tags [Adding default attributes to all images](#) (Jake Howard)
- Optimise the performance of the Wagtail userbar to remove duplicated queries, improving page loads when viewing live pages while signed in (Jake Howard)
- Remove legacy styling classes for buttons and refactor button styles to be more maintainable (Paarth Agarwal, LB (Ben Johnston))
- Add button variations to the pattern library (Paarth Agarwal)
- Provide a more accessible page title where the unique information is shown first and the CMS name is shown last (Mehrdad Moradizadeh)
- Pull out behaviour from `AbstractFormField` to `FormMixin` and `AbstractEmailForm` to `EmailFormMixin` to allow use with subclasses of `Page` [Using FormMixin or EmailFormMixin to use with other Page subclasses](#) (Mehrdad Moradizadeh, Kurt Wall)
- Add a `docs.wagtail.org/.well-known/security.txt` so that the security policy is available as per the specification on <https://securitytxt.org/> (Jake Howard)
- Add unit tests for the `classnames` Wagtail admin template tag (Mehrdad Moradizadeh)
- Show an inverse locked indicator when the page has been locked by the current user in reports and dashboard listings (Vaibhav Shukla, LB (Ben Johnston))
- Add clarity to the development documentation that `admonition` should not be used and titles for note are not supported, including clean up of some existing incorrect usage (LB (Ben Johnston))
- Unify the styling of delete/destructive button styles across the admin interface (Paarth Agarwal)
- Adopt new designs and unify the styling styles for `.button-secondary` buttons across the admin interface (Paarth Agarwal)
- Refine designs for disabled buttons throughout the admin interface (Paarth Agarwal)
- Update expanding formset add buttons to use `button` not `link` for behaviour and remove support for disabled as a class (LB (Ben Johnston))

- Add robust unit testing for authentication scenarios across the user management admin pages (Mehrdad Moradizadeh)
- Avoid assuming an integer PK named ‘id’ on multiple upload views (Matt Westcott)
- Add a toggle to collapse/expand all page panels at once (Helen Chapman)
- Improve the GitHub Workflows (CI) security (Alex (sashashura))

## Bug fixes

- Prevent `PageQuerySet.not_public` from returning all pages when no page restrictions exist (Mehrdad Moradizadeh)
- Ensure that duplicate block ids are unique when duplicating stream blocks in the page editor (Joshua Munn)
- Revise colour usage so that privacy & locked indicators can be seen in Windows High Contrast mode (LB (Ben Johnston))
- Ensure that disabled buttons have a consistent presentation on hover to indicate no interaction is available (Paarth Agarwal)
- Update the ‘Locked pages’ report menu title so that it is consistent with other pages reports and its own title on viewing (Nicholas Johnson)
- Support `formfield_callback` handling on `ModelForm.Meta` for future Django 4.2 release (Matt Westcott)
- Ensure that `ModelAdmin` correctly supports filters in combination with subsequent searches without clearing the applied filters (Stefan Hammer)
- Add missing translated values to site settings’ headers plus models presented in listings and audit report filtering labels (Stefan Hammer)
- Remove `capitalize()` calls to avoid issues with other languages or incorrectly presented model names for reporting and parts of site settings (Stefan Hammer)

## Upgrade considerations

### Button styling class changes

The `button-secondary` class is no longer compatible with either the `.serious` or `.no` classes, this partially worked previously but is no longer officially supported.

When adding custom buttons using the `ModelAdmin ButtonHelper` class, custom buttons will no longer include the `button-secondary` class by default in index listings.

If using the hook `register_user_listing_buttons` to register buttons along with the undocumented `UserListingButton` class, the `button-secondary` class will no longer be included by default.

Avoid using `disabled` as a class on `button` elements, instead use the `disabled` attribute as support for this as a class may be removed in a future version of Wagtail and is not accessible.

If using custom `expanding-formset` the add button will no longer support the `disabled` class but instead must require the `disabled` attribute to be set.

The following button classes have been removed, none of which were being used within the admin but may have been used by custom code or packages:

- `button-neutral`
- `button-strokeonhover`

- hover-no
- unbutton
- yes

### 1.9.3 Wagtail 4.0.2 release notes

*Unreleased*

- *What's new*

#### What's new

- Update all images and sections of the Wagtail Editor's guide to align with the new admin interface changes from Wagtail 3.0 and 4.0 (Thibaud Colas)
- Ensure all images in the documentation have a suitable alt text (Thibaud Colas)

#### Bug fixes

- Ensure tag autocomplete dropdown has a solid background (LB (Ben) Johnston)
- Allow inline panels to be ordered (LB (Ben) Johnston)
- Only show draft / live status tags on snippets that have `DraftStateMixin` applied (Sage Abdullah)
- Prevent JS error when initialising chooser modals with no tabs (LB (Ben) Johnston)
- Add missing vertical spacing between chooser modal header and body when there are no tabs (LB (Ben) Johnston)
- Reinstate specific labels for chooser buttons (for example ‘Choose another page’, ‘Edit this page’ not ‘Change’, ‘Edit’) so that it is clearer for users and non-English translations (Matt Westcott)
- Resolve issue where searches with a tag and a query param in the image listing would result in an `FilterFieldError` (Stefan Hammer)
- Add missing vertical space between header and content in embed chooser modal (LB (Ben) Johnston)
- Use the correct type scale for heading levels in rich text (Steven Steinwand)
- Update alignment and reveal logic of fields’ comment buttons (Steven Steinwand)
- Regression from Markdown conversion in documentation for API configuration - update to correctly use PEP-8 for example code (Storm Heg)
- Prevent ‘Delete’ link on page edit view from redirecting back to the deleted page (LB (Ben) Johnston)
- Prevent JS error on images index view when collections dropdown is omitted (Tidiane Dia)
- Prevent “Entries per page” dropdown on images index view from reverting to 10 (Tidiane Dia)
- Set related\_name on user revision relation to avoid conflict with django-reversion (Matt Westcott)
- Ensure the “recent edits” panel on the Dashboard (home) page works when page record is missing (Matt Westcott)
- Only add Translate buttons when the `simple_translation` app is installed (Dan Braghis)
- Ensure that `MultiFieldPanel` correctly outputs all child classnames in the template (Matt Westcott)

## 1.9.4 Wagtail 4.0.1 release notes

September 5, 2022

- *What's new*

### What's new

### Bug fixes

- On the Locked pages report, limit the “locked by” filter to just users who have locked pages (Stefan Hammer)
- Prevent JavaScript error when using StreamField on views without commenting support, such as snippets (Jacob Topp-Muggleton)
- Modify base template for new projects so that links opened from the preview panel open in a new window (Sage Abdullah)
- Prevent circular import error between custom document models and document chooser blocks (Matt Westcott)

## 1.9.5 Wagtail 4.0 release notes

August 31, 2022

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

### What's new

### Django 4.1 support

This release adds support for Django 4.1.

### Global settings models

The new `BaseGenericSetting` base model class allows defining a settings model that applies to all sites rather than just a single site.

See [the Settings documentation](#) for more information. This feature was implemented by Kyle Bayliss.

## **Image renditions can now be prefetched by filter**

When using a queryset to render a list of images, you can now use the `prefetch_renditions()` queryset method to prefetch the renditions needed for rendering with a single extra query, similar to `prefetch_related`. If you have many renditions per image, you can also call it with filters as arguments - `prefetch_renditions("fill-700x586", "min-600x400")` - to fetch only the renditions you intend on using for a smaller query. For long lists of images, this can provide a significant boost to performance. See [Prefetching image renditions](#) for more examples. This feature was developed by Tidiane Dia and Karl Hobley.

## **Page editor redesign**

Following from Wagtail 3.0, this release contains significant UI changes that affect all of Wagtail's admin, largely driven by the implementation of the new Page Editor. These include:

- Updating all widget styles across the admin UI, including basic form widgets, as well as choosers.
- Updating field styles across forms, with help text consistently under fields, error messages above, and comment buttons to the side.
- Making all sections of the page editing UI collapsible by default.
- New designs for StreamField and InlinePanel blocks, with better support for nested blocks.
- Updating the side panels to prevent overlap with form fields unless necessary.

Further updates to the page editor are expected in the next release. Those changes were implemented by Thibaud Colas. Development on this feature was sponsored by Google.

## **Rich text improvements**

As part of the page editor redesign project sponsored by Google, we have made a number of improvements to our rich text editor:

- **Inline toolbar:** The toolbar now shows inline, to avoid clashing with the page's header.
- **Command palette:** Start a block with a slash '/' to open the palette and change the text format.
- **Character count:** The character count is displayed underneath the editor, live-updating as you type. This counts the length of the text, not of any formatting.
- **Paste to auto-create links:** To add a link from your copy-paste clipboard, select text and paste the URL.
- **Text shortcuts undo:** The editor normally converts text starting with 1. to a list item. It's now possible to un-do this change and keep the text as-is. This works for all Markdown-style shortcuts.
- **RTL support:** The editor's UI now displays correctly in right-to-left languages.
- **Focus-aware placeholder:** The editor's placeholder text will now follow the user's focus, to make it easier to understand where to type in long fields.
- **Empty heading highlight:** The editor now highlights empty headings and list items by showing their type ("Heading 3") as a placeholder, so content is less likely to be published with empty headings.
- **Split and insert:** rich text fields can now be split while inserting a new StreamField block of the desired type.

## Live preview panel

Wagtail's page preview is now available in a side panel within the page editor. This preview auto-updates as users type, and can display the page in three different viewports: mobile, tablet, desktop. The existing preview functionality is still present, moved inside the preview panel rather than at the bottom of the page editor. The auto-update delay can be configured with the `WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL` setting. This feature was developed by Sage Abdullah.

## Admin colour themes

In Wagtail 2.12, we introduced theming support for Wagtail's primary brand colour. This has now been extended to almost all of Wagtail's colour palette. View our [Custom user interface colours](#) documentation for more information, an overview of Wagtail's customisable colour palette, and a live demo of the supported customisations. This was implemented by Thibaud Colas, under the page editor redesign project sponsored by Google.

## Windows High Contrast mode support improvements

In Wagtail 2.16, we introduced support for Windows High Contrast mode (WHCM). This release sees a lot of improvements to our support, thanks to our new contributor Anuja Verma, who has been working on this as part of the [Contrast Themes](#) Google Summer of Code project, with support from Jane Hughes, Scott Cranfill, and Thibaud Colas.

- Improve help block styles with less reliance on communication via colour alone in forced colors mode
- Add a bottom border to top messages so they stand out from the header in forced colors mode
- Make progress bars' progress visible in forced colors mode
- Make checkboxes visible in forced colors mode
- Display the correct color for icons in forced colors mode
- Add a border around modal dialogs so they can be identified in forced colors mode
- Ensure disabled buttons are distinguishable from active buttons in forced colors mode
- Ensure that the fields on login and password reset forms are visible in forced colors mode
- Missing a outline on dropdown content and malformed tooltip arrow in forced colors mode

## UX unification and consistency improvements

In Wagtail 3.0, a new Page Editor experience was introduced, this release brings many of the UX and UI improvements to other parts of Wagtail for a more consistent experience. The bulk of these enhancements have been from Paarth Agarwal, who has been doing the [UX Unification](#) internship project alongside other Google Summer of Code participants. This internship has been sponsored by Torchbox with mentoring support from LB (Ben Johnston), Thibaud Colas and Helen Chapman.

- **Login and password reset**
  - Refreshed design for login and password reset pages to better suit a wider range of device sizes
  - Better accessibility and screen reader support for the sign in form due to a more appropriate DOM structure and skip link improvements
  - Remove usage of inline script to focus on username field and instead use `autofocus`
  - Wagtail logo added with the ability to override this logo via [Custom branding](#)

- **Breadcrumbs**

- Add Breadcrumbs to the Wagtail pattern library
- Enhance new breadcrumbs so they can be added to any header or container element
- Adopt new breadcrumbs on the page explorer (listing) view and the page chooser modal, remove legacy breadcrumbs code for move page as no longer used
- Rename `explorer_breadcrumb` template tag to `breadcrumbs` as it is now used in multiple locations
- Remove legacy (non-next) breadcrumbs no longer used, remove `ModelAdmin` usage of breadcrumbs completely and adopt consistent ‘back’ link approach

- **Headers**

- Update classes and styles for the shared header templates to align with UI guidelines
- Ensure the shared header template is more reusable, add ability to include classes, extra content and other blocks
- Switch all report workflow, redirects, form submissions, site settings views to use Wagtail’s reusable header component
- Resolve issues throughout Wagtail where the sidebar toggle would overlap header content on small devices

- **Tabs**

- Add Tabs to the Wagtail pattern library
- Adopt new Tabs component in the workflow history report page, removing the bespoke implementation there

- **Dashboard (home) view**

- Migrate the dashboard (home) view header to the shared header template and update designs
- Refresh designs for Home (Dashboard) site summary panels, use theme spacing and colours
- Add support for RTL layouts and better support for small devices
- Add CSS support for more than three summary items if added via customisations

- **Page Listing view**

- Adopt the slim header in page listing views, with buttons moved under the “Actions” dropdown
- Add convenient access to the translate page button in the parent “more” button

## Previews, revisions and drafts for snippets

Snippets can now be given a previewable HTML representation, revision history, and draft / live states through the use of the mixins `PreviewableMixin`, `RevisionMixin`, and `DraftStateMixin`. For more details, see:

- *Making snippets previewable*
- *Saving revisions of snippets*
- *Saving draft changes of snippets*

These features were developed by Sage Abdullah.

## Documentation improvements

The documentation now has dark mode which will be turned on by default if set in your browser or OS preferences, it can also be toggled on and off manually. The colours and fonts of the documentation now align with the design updates introduced in Wagtail 3.0. These features were developed by Vince Salvino.

There are also many improvements to the documentation both under the hood and in the layout;

- Convert the rest of the documentation to Markdown, in place of RST, which will make it much easier for others to contribute to better documentation (Khanh Hoang, Vu Pham, Daniel Kirkham, LB (Ben) Johnston, Thiago Costa de Souza, Benedict Faw, Noble Mittal, Sævar Öfjörð Magnússon, Sandeep M A, Stefano Silvestri)
- Replace latin abbreviations (i.e. / e.g.) with common English phrases so that documentation is easier to understand (Dominik Lech)
- Improve organisation of the settings reference page with logical grouping and better internal linking (Akash Kumar Sen)
- Improve the accessibility of the documentation with higher contrast colours, consistent focus outline, better keyboard only navigation through the sidebar (LB (Ben) Johnston, Vince Salvino)
- Better sidebar scrolling behaviour, it is now sticky on larger devices and scrollable on its own (Paarth Agarwal, LB (Ben) Johnston)
- Fix links showing incorrectly in Safari (Tibor Leupold)
- See other features below for new feature specific documentation added.

## Other features

- Add clarity to confirmation when being asked to convert an external link to an internal one (Thijs Kramer)
- Add `base_url_path` to `ModelAdmin` so that the default URL structure of `app_label/model_name` can be overridden (Vu Pham, Khanh Hoang)
- Add `full_url` to the API output of `ImageRenditionField` (Paarth Agarwal)
- Use `InlinePanel`'s label when available for field comparison label (Sandil Ranasinghe)
- Drop support for Safari 13 by removing left/right positioning in favour of CSS logical properties (Thibaud Colas)
- Use `FormData` instead of jQuery's `form.serialize` when editing documents or images just added so that additional fields can be better supported (Stefan Hammer)
- Add informationalCodecov status checks for GitHub CI pipelines (Tom Hu)
- Make it possible to reuse and customise Wagtail's fonts with CSS variables (LB (Ben) Johnston)
- Add better handling and informative developer errors for cross linking URLs (e.g. success after add) in generic views `wagtail.admin.views.generic` (Matt Westcott)
- Introduce `wagtail.admin.widgets.chooser.BaseChooser` to make it easier to build custom chooser inputs (Matt Westcott)
- Introduce JavaScript chooser module, including a `SearchController` class which encapsulates the standard pattern of re-rendering the results panel in response to search queries and pagination (Matt Westcott)
- Migrate Image and Document choosers to new JavaScript chooser module (Matt Westcott)
- Add ability to select multiple items at once within bulk actions selections when holding shift on subsequent clicks (Hitansh Shah)

- Upgrade notification, shown to admins on the dashboard if Wagtail is out of date, will now link to the release notes for the closest minor branch instead of the latest patch (Tibor Leupold)
- Upgrade notification can now be configured to only show updates when there is a new LTS available via `WAGTAIL_ENABLE_UPDATE_CHECK = 'lts'` (Tibor Leupold)
- Implement redesign of the Workflow Status dialog, fixing accessibility issues (Steven Steinwand)
- Add the ability to change the number of images displayed per page in the image library (Tidiane Dia, with sponsorship from YouGov)
- Allow users to sort by different fields in the image library (Tidiane Dia, with sponsorship from YouGov)
- Add `prefetch_renditions` method to `ImageQueryset` for performance optimisation on image listings (Tidiane Dia, Karl Hobley)
- Add ability to define a custom `get_field_clean_name` method when defining `FormField` models that extend `AbstractFormField` (LB (Ben) Johnston)
- Migrate Home (Dashboard) view to use generic Wagtail class based view (LB (Ben) Johnston)
- Combine most of Wagtail's stylesheets into the global `core.css` file (Thibaud Colas)
- Update `ReportView` to extend from generic `wagtail.admin.views.generic.models.IndexView` (Sage Abdullah)
- Update pages `Unpublish` view to extend from generic `wagtail.admin.views.generic.models.UnpublishView` (Sage Abdullah)
- Introduce a `wagtail.admin.viewsets.chooser.ChooserViewSet` module to serve as a common base implementation for chooser modals (Matt Westcott)
- Add documentation for `wagtail.admin.viewsets.model.ModelViewSet` (Matt Westcott)
- Added *multi-site support* to the API (Sævar Öfjörð Magnússon)
- Add `add_to_admin_menu` option for `ModelAdmin` (Oliver Parker)
- Implement *Fuzzy matching* for Elasticsearch (Nick Smith)
- Cache model permission codenames in `PermissionHelper` (Tidiane Dia)
- Selecting a new parent page for moving a page now uses the chooser modal which allows searching (Viggo de Vries)
- Add clarity to the search indexing documentation for how `boost` works when using Postgres with the database search backend (Tibor Leupold)
- Updated `django-filter` version to support 23 (Yuekui)
- Use `.iterator()` in a few more places in the admin, to make it more stable on sites with many pages (Andy Babic)
- Migrate some simple React component files to TypeScript (LB (Ben) Johnston)
- Deprecate the usage and documentation of the `wagtail.contrib.modeladmin.menus.SubMenu` class, provide a warning if used directing developers to use `wagtail.admin.menu.Menu` instead (Matt Westcott)
- Replace human-readable-date hover pattern with accessible tooltip variant across all of admin (Bernd de Ridder)
- Added `WAGTAILADMIN_USER_PASSWORD_RESET_FORM` setting for overriding the admin password reset form (Michael Karamuth)
- Prefetch workflow states in edit page view to avoid queries in other parts of the view/templates that need it (Tidiane Dia)
- Remove the edit link from edit bird in previews to avoid confusion (Sævar Öfjörð Magnússon)

- Introduce new template fragment and block level enclosure tags for easier template composition (Thibaud Colas)
- Add a `classnames` template tag to easily build up classes from variables provided to a template (Paarth Agarwal)
- Clean up multiple eslint rules usage and configs to align better with the Wagtail coding guidelines (LB (Ben Johnston))
- Make `ModelAdmin InspectView` footer actions consistent with other parts of the UI (Thibaud Colas)
- Add support for Twitter and other text-only embeds in Draftail embed previews (Iman Syed, Paarth Agarwal)
- Use new modal dialog component for privacy settings modal (Sage Abdullah)
- Add `menu_item_name` to modify `MenuItem`'s name for `ModelAdmin` (Alexander Rogovskyy, Vu Pham)
- Add an extra confirmation prompt when deleting pages with a large number of child pages, see [WAGTAILADMIN\\_UNSAFE\\_PAGE\\_DELETION\\_LIMIT](#) (Jaspreet Singh)
- Add shortcut for accessing StreamField blocks by block name with new `blocks_by_name` and `first_block_by_name` methods on `StreamValue` (Tidiane Dia, Matt Westcott)
- Add HTML-aware `max_length` validation and character count on RichTextField and RichTextBlock (Matt Westcott, Thibaud Colas)
- Remove `is_parent` kwarg in various page button hooks as this approach is no longer required (Paarth Agarwal)
- Improve security of redirect imports by adding a file hash (signature) check for so that any tampering of file contents between requests will throw a `BadSignature` error (Jaap Roes)
- Allow generic chooser viewsets to support non-model data such as an API endpoint (Matt Westcott)
- Added `path` and `re_path` decorators to the `RoutablePageMixin` module which emulate their Django URL utils equivalent, redirect `re_path` to the original `route` decorator (Tidiane Dia)
- `BaseChooser` widget now provides a Telepath adapter that's directly usable for any subclasses that use the chooser widget and modal JS as-is with no customisations (Matt Westcott)
- Introduce new template fragment and block level enclosure tags for easier template composition (Thibaud Colas)
- Implement the new chooser widget styles as part of the page editor redesign (Thibaud Colas)
- Update base Draftail/TextField form designs as part of the page editor redesign (Thibaud Colas)
- Move commenting trigger to inline toolbar and move block splitting to the block toolbar and command palette only in Draftail (Thibaud Colas)
- Pages are now locked when they are scheduled for publishing (Karl Hobley)
- Simplify page chooser views by converting to class-based views (Matt Westcott)
- Add “Translate” button within pages’ Actions dropdown when editing pages (Sage Abdullah)
- Add translated labels to the bulk actions tags and collections bulk update fields (Stefan Hammer)
- Add support for bulk actions, including [Adding bulk actions to the snippets listing](#) (Shohan Dutta Roy)

## Bug fixes

- Fix issue where `ModelAdmin` index listings with export list enabled would show buttons with an incorrect layout (Josh Woodcock)
- Fix typo in `ResumeWorkflowActionFormatter` message (Stefan Hammer)
- Throw a meaningful error when saving an image to an unrecognised image format (Christian Franke)
- Remove extra padding for headers with breadcrumbs on mobile viewport (Steven Steinwand)
- Replace `PageRevision` with generic `Revision` model (Sage Abdullah)
- Ensure that custom document or image models support custom tag models (Matt Westcott)
- Ensure comments use translated values for their placeholder text (Stefan Hammer)
- Ensure the upgrade notification, shown to admins on the dashboard if Wagtail is out of date, content is translatable (LB (Ben) Johnston)
- Only show the re-ordering option to users that have permission to publish pages within the page listing (Stefan Hammer)
- Ensure default sidebar branding (bird logo) is not cropped in RTL mode (Steven Steinwand)
- Add an accessible label to the image focal point input when editing images (Lucie Le Frapper)
- Remove unused header search JavaScript on the redirects import page (LB (Ben) Johnston)
- Ensure non-square avatar images will correctly show throughout the admin (LB (Ben) Johnston)
- Ignore translations in test files and re-include some translations that were accidentally ignored (Stefan Hammer)
- Show alternative message when no page types are available to be created (Jaspreet Singh)
- Prevent error on sending notifications for the legacy moderation process when no user was specified (Yves Serrano)
- Ensure `aria-label` is not set on locale selection dropdown within page chooser modal as it was a duplicate of the button contents (LB (Ben) Johnston)
- Revise the `ModelAdmin` title column behaviour to only link to ‘edit’ if the user has the correct permissions, fallback to the ‘inspect’ view or a non-clickable title if needed (Stefan Hammer)
- Ensure that `DecimalBlock` preserves the `Decimal` type when retrieving from the database (Yves Serrano)
- When no snippets are added, ensure the snippet chooser modal would have the correct URL for creating a new snippet (Matt Westcott)
- `ngettext` in Wagtail’s internal JavaScript internationalisation utilities now works (LB (Ben) Johnston)
- Ensure the linting/formatting npm scripts work on Windows (Anuja Verma)
- Fix display of dates in exported `xlsx` files on macOS Preview and Numbers (Jaap Roes)
- Remove outdated reference to 30-character limit on usernames in help text (minusf)
- Resolve multiple form submissions index listing page layout issues including title not being visible on mobile and interaction with large tables (Paarth Agarwal)
- Ensure `ModelAdmin` single selection lists show correctly with Django 4.0 form template changes (Coen van der Kamp)
- Ensure icons within help blocks have accessible contrasting colours, and links have a darker colour plus underline to indicate they are links (Paarth Agarwal)
- Ensure consistent sidebar icon position whether expanded or collapsed (Scott Cranfill)

- Avoid redirects import error if the file had lots of columns (Jaap Roes)
- Resolve accessibility and styling issues with the expanding status panel (Sage Abdullah)
- Avoid 503 `AttributeError` when an empty search param `q=` is combined with other filters in the Images index view (Paritosh Kabra)
- Fix error with string representation of `FormSubmission` not returning a string (LB (Ben) Johnston)
- Ensure that bulk actions correctly support models with non-integer primary keys (`id`) (LB (Ben) Johnston)
- Make it possible to toggle collapsible panels in the edit UI with the keyboard (Thibaud Colas)
- Re-implement checkbox styles so the checked state is visible in forced colors mode (Thibaud Colas)
- Re-implement switch component styles so the checked state is visible in forced colors mode (Thibaud Colas)
- Always render select widgets consistently regardless of where they are in the admin (Thibaud Colas)
- Make sure input labels and always take up the available space (Thibaud Colas)
- Correctly style `BooleanBlock` within `StructBlock` (Thibaud Colas)
- Make sure comment icons can't overlap with help text (Thibaud Colas)
- Make it possible to scroll input fields in admin on safari mobile (Thibaud Colas)
- Stop rich text fields from overlapping with sidebar (Thibaud Colas)
- Prevent comment buttons from overlapping with fields (Thibaud Colas)
- Resolve MySQL search compatibility issue with Django 4.1 (Andy Chosak)
- Resolve layout issues with reports (including form submissions listings) on md device widths (Akash Kumar Sen, LB (Ben) Johnston)
- Resolve Layout issue with page explorer's inner header item on small device widths (Akash Kumar Sen)
- Ensure that `BaseSiteSetting` / `BaseGenericSetting` objects can be pickled (Andy Babic)
- Ensure `DocumentChooserBlock` can be deconstructed for migrations (Matt Westcott)
- Resolve frontend console error and unintended console logging issues (Matt Westcott, Paarth Agarwal)
- Resolve issue with sites that have not yet migrated away from `BaseSetting` when upgrading to Wagtail 4.0 (Stefan Hammer)
- Use correct classnames for showing/hiding edit button on chooser widget (Matt Westcott)
- Render `MultiFieldPanel`'s heading even when nested (Thibaud Colas)
- Make sure select widgets render correctly regardless of the Django field and widget type (Thibaud Colas)
- Consistently display boolean field labels above the widget so they render correctly (Thibaud Colas)
- Address form field label alignment issues by always displaying labels above the widget (Thibaud Colas)
- Make sure rich text URL editing tooltip is fully visible when displayed inside `InlinePanel` blocks (Thibaud Colas)
- Allow input fields to scroll horizontally in Safari iOS (Thibaud Colas)
- Ensure screen readers are made aware of page level messages added dynamically to the top of the page (Paarth Agarwal)
- Fix `updatemodulepaths` command for Python 3.7 (Matt Westcott)
- Only show locale filter in choosers when i18n is enabled in settings (Matt Westcott)
- Ensure that the live preview panel correctly clears the cache when a new page is created (Sage Abdullah)

- Ensure that there is a larger hoverable area for add block (+) within the Drafttail editor (Steven Steinwand)
- Resolve multiple header styling issues for modal, alignment on small devices, outside click handling target on medium devices, close button target size and hover styles (Paarth Agarwal)
- Fix issue where comments could not be added in StreamField that were already already saved (Jacob Topp-Muggleston)
- Remove outdated reference to Image.LoaderError (Matt Westcott)

### Upgrade considerations

#### Changes to `Page.serve()` and `Page.serve_preview()` methods

As part of making previews available to non-page models, the `serve_preview()` method has been decoupled from the `serve()` method and extracted into the `PreviewableMixin` class. If you have overridden the `serve()` method in your page models, you will likely need to override `serve_preview()`, `get_preview_template()`, and/or `get_preview_context()` methods to handle previews accordingly. Alternatively, you can also override the `preview_modes` property to return an empty list to disable previews.

#### Opening links within the live preview panel

The live preview panel utilises an iframe to display the preview in the editor page, which requires the page in the iframe to have the `X-Frame-Options` header set to `SAMEORIGIN` (or unset). If you click a link within the preview panel, you may notice that the iframe stops working. This is because the link is loaded within the iframe and the linked page may have the `X-Frame-Options` header set to `DENY`. To work around this problem, add the following `<base>` tag within your `<head>` element in your `base.html` template, before any `<link>` elements:

```
{% if request.in_preview_panel %}
 <base target="_blank">
{% endif %}
```

This will make all links in the live preview panel open in a new tab.

As of Wagtail 4.0.1, new Wagtail projects created through the `wagtail start` command already include this change in the base template.

#### `base_url_path` keyword argument added to `AdminURLHelper`

The `wagtail.contrib.modeladmin.helpers.AdminURLHelper` class now accepts a `base_url_path` keyword argument on its constructor. Custom subclasses of this class should be updated to accept this keyword argument.

#### Dropped support for Safari 13

Safari 13 will no longer be officially supported as of this release, this deviates the current support for the last 3 version of Safari by a few months and was required to add better support for RTL languages.

## PageRevision replaced with Revision

The `PageRevision` model has been replaced with a generic `Revision` model. If you use the `PageRevision` model in your code, make sure that:

- Creation of `PageRevision` objects should be updated to create `Revision` objects using the page's `id` as the `object_id`, the default `Page` model's content type as the `base_content_type`, and the page's specific content type as the `content_type`.
- Queries that use the `PageRevision.objects` manager should be updated to use the `Revision.page_revisions` manager.
- Revision queries that use `Page.id` should be updated to cast the `Page.id` to a string before using it in the query (e.g. by using `str()` or `Cast("page_id", output_field=CharField())`).
- Page queries that use `PageRevision.page_id` should be updated to cast the `Revision.object_id` to an integer before using it in the query (e.g. by using `int()` or `Cast("object_id", output_field=IntegerField())`).
- Access to `PageRevision.page` should be updated to `Revision.content_object`.

If you maintain a package across multiple Wagtail versions that includes a model with a `ForeignKey` to the `PageRevision` model, you can create a helper function to correctly resolve the model depending on the installed Wagtail version, for example:

```
from django.db import models
from wagtail import VERSION as WAGTAIL_VERSION

def get_revision_model():
 if WAGTAIL_VERSION >= (4, 0):
 return "wagtailcore.Revision"
 return "wagtailcore.PageRevision"

class MyModel(models.Model):
 # Before
 # revision = models.ForeignKey("wagtailcore.PageRevision")
 revision = models.ForeignKey(get_revision_model(), on_delete=models.CASCADE)
```

### `Page.get_latest_revision_as_page` renamed to `Page.get_latest_revision_as_object`

The `Page.get_latest_revision_as_page` method has been renamed to `Page.get_latest_revision_as_object`. The old name still exists for backwards-compatibility, but calling it will raise a `RemovedInWagtail50Warning`.

### `AdminChooser` replaced with `BaseChooser`

Custom choosers should no longer use `wagtail.admin.widgets.chooser.AdminChooser` which has been replaced with `wagtail.admin.widgets.chooser.BaseChooser`.

### `get_snippet_edit_handler` moved to `wagtail.admin.panels.get_edit_handler`

The `get_snippet_edit_handler` function in `wagtail.snippets.views.snippets` has been moved to `get_edit_handler` in `wagtail.admin.panels`.

### `explorerBreadcrumb` template tag has been renamed to `breadcrumbs`, `moveBreadcrumb` has been removed

The `explorerBreadcrumb` template tag is not documented, however if used it will need to be renamed to `breadcrumbs` and the `url_name` is now a required arg.

The `moveBreadcrumb` template tag is no longer used and has been removed.

### `wagtail.contrib.modeladmin.menus.SubMenu` is deprecated

The `wagtail.contrib.modeladmin.menus.SubMenu` class should no longer be used for constructing submenus of the admin sidebar menu. Instead, import `wagtail.admin.menu.Menu` and pass the list of menu items as the `items` keyword argument.

### Chooser widget JavaScript initialisers replaced with classes

The internal JavaScript functions `createPageChooser`, `createSnippetChooser`, `createDocumentChooser` and `createImageChooser` used for initialising chooser widgets have been replaced by classes, and user code that calls them needs to be updated accordingly:

- `createPageChooser(id)` should be replaced with `new PageChooser(id)`
- `createSnippetChooser(id)` should be replaced with `new SnippetChooser(id)`
- `createDocumentChooser(id)` should be replaced with `new DocumentChooser(id)`
- `createImageChooser(id)` should be replaced with `new ImageChooser(id)`

## URL route names for image, document and snippet apps have changed

If your code contains references to URL route names within the `wagtailimages`, `wagtailedocs` or `wagtailsnippets` namespaces, these should be updated as follows:

- `wagtailimages:chooser` is now `wagtailimages_chooser:choose`
- `wagtailimages:chooser_results` is now `wagtailimages_chooser:choose_results`
- `wagtailimages:image_chosen` is now `wagtailimages_chooser:chosen`
- `wagtailimages:chooser_upload` is now `wagtailimages_chooser:create`
- `wagtailimages:chooser_select_format` is now `wagtailimages_chooser:select_format`
- `wagtailedocs:chooser` is now `wagtailedocs_chooser:choose`
- `wagtailedocs:chooser_results` is now `wagtailedocs_chooser:choose_results`
- `wagtailedocs:document_chosen` is now `wagtailedocs_chooser:chosen`
- `wagtailedocs:chooser_upload` is now `wagtailedocs_chooser:create`
- `wagtailsnippets:list`, `wagtailsnippets:list_results`, `wagtailsnippets:add`,  
`wagtailsnippets:edit`, `wagtailsnippets:delete-multiple`, `wagtailsnippets:delete`,  
`wagtailsnippets:usage`, `wagtailsnippets:history`: These now exist in a separate  
`wagtailsnippets_{app_label}_{model_name}` namespace for each snippet model, and no longer  
take `app_label` and `model_name` as arguments.
- `wagtailsnippets:choose`, `wagtailsnippets:choose_results`, `wagtailsnippets:chosen`: These  
now exist in a separate `wagtailsnippetchoosers_{app_label}_{model_name}` namespace for each snippet  
model, and no longer take `app_label` and `model_name` as arguments.

## Auto-updating preview

As part of the introduction of the new live preview panel, we have changed the `WAGTAIL_AUTO_UPDATE_PREVIEW` setting to be on (True) by default. This can still be turned off by setting it to False. The `WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL` setting has been introduced for sites willing to reduce the performance cost of the live preview without turning it off completely.

## Slim header on page listings

The page explorer listings now use Wagtail's new slim header, replacing the previous large teal header. The parent page's metadata and related actions are now available within the "Info" side panel, while the majority of buttons are now available under the Actions dropdown in the header, identically to the page create/edit forms.

Customising which actions are available and adding extra actions is still possible, but has to be done with the `register_page_header_buttons` hook, rather than `register_page_listing_buttons` and `register_page_listing_more_buttons`. Those hooks still work as-is to define actions for each page within the listings.

### is\_parent removed from page button hooks

- The following hooks `construct_page_listing_buttons`, `register_page_listing_buttons`, `register_page_listing_more_buttons` no longer accept the `is_parent` keyword argument and this should be removed.
- `is_parent` was the previous approach for determining whether the buttons would show in the listing rows or the page's more button, this can be now achieved with discrete hooks instead.

### Changed CSS variables for admin colour themes

As part of our support for theming across all colors, we've had to rename or remove some of the pre-existing CSS variables. Wagtail's indigo is now customisable with `--w-color-primary`, and the teal is customisable as `--w-color-secondary`. See [Custom user interface colours](#) for an overview of all customisable colours. Here are replaced variables:

- `--color-primary` is now `--w-color-secondary`
- `--color-primary-hue` is now `--w-color-secondary-hue`
- `--color-primary-saturation` is now `--w-color-secondary-saturation`
- `--color-primary-lightness` is now `--w-color-secondary-lightness`
- `--color-primary-darker` is now `--w-color-secondary-400`
- `--color-primary-darker-hue` is now `--w-color-secondary-400-hue`
- `--color-primary-darker-saturation` is now `--w-color-secondary-400-saturation`
- `--color-primary-darker-lightness` is now `--w-color-secondary-400-lightness`
- `--color-primary-dark` is now `--w-color-secondary-600`
- `--color-primary-dark-hue` is now `--w-color-secondary-600-hue`
- `--color-primary-dark-saturation` is now `--w-color-secondary-600-saturation`
- `--color-primary-dark-lightness` is now `--w-color-secondary-600-lightness`
- `--color-primary-lighter` is now `--w-color-secondary-100`
- `--color-primary-lighter-hue` is now `--w-color-secondary-100-hue`
- `--color-primary-lighter-saturation` is now `--w-color-secondary-100-saturation`
- `--color-primary-lighter-lightness` is now `--w-color-secondary-100-lightness`
- `--color-primary-light` is now `--w-color-secondary-50`
- `--color-primary-light-hue` is now `--w-color-secondary-50-hue`
- `--color-primary-light-saturation` is now `--w-color-secondary-50-saturation`
- `--color-primary-light-lightness` is now `--w-color-secondary-50-lightness`

We've additionally removed all `--color-input-focus` and `--color-input-focus-border` variables, as Wagtail's form fields no longer have a different colour on focus.

`WAGTAILDOCS_DOCUMENT_FORM_BASE` and `WAGTAILIMAGES_IMAGE_FORM_BASE` must inherit from `BaseDocumentForm` / `BaseImageForm`

Previously, it was valid to specify an arbitrary model form as the `WAGTAILDOCS_DOCUMENT_FORM_BASE` / `WAGTAILIMAGES_IMAGE_FORM_BASE` settings. This is no longer supported; these forms must now inherit from `wagtail.documents.forms.BaseDocumentForm` and `wagtail.images.forms.BaseImageForm` respectively.

## Panel customisations

As part of the page editor redesign, we have removed support for the `classname="full"` customisation to panels. Existing title and collapsed customisations remain unchanged.

## Optional replacement for regex only route decorator for RoutablePageMixin

- This is an optional replacement, there are no immediate plans to remove the `route` decorator at this time.
- The `RoutablePageMixin` contrib module now provides a `path` decorator that behaves the same way as Django's `django.urls.path` function.
- `RoutablePageMixin`'s `route` decorator will now redirect to a new `re_path` decorator that emulates the behaviour of `django.urls.re_path`.

## BaseSetting model replaced by BaseSiteSetting

The `wagtail.contrib.settings.models.BaseSetting` model has been replaced by two new base models `BaseSiteSetting` and `BaseGenericSetting`, to accommodate settings that are shared across all sites. Existing setting models that inherit `BaseSetting` should be updated to use `BaseSiteSetting` instead:

```
from wagtail.contrib.settings.models import BaseSetting, register_setting

@register_setting
class SiteSpecificSocialMediaSettings(BaseSetting):
 facebook = models.URLField()
```

should become

```
from wagtail.contrib.settings.models import BaseSiteSetting, register_setting

@register_setting
class SiteSpecificSocialMediaSettings(BaseSiteSetting):
 facebook = models.URLField()
```

## **1.9.6 Wagtail 3.0.3 release notes**

*September 5, 2022*

- *What's new*

### **What's new**

#### **Bug fixes**

- On the Locked pages report, limit the “locked by” filter to just users who have locked pages (Stefan Hammer)
- Prevent JavaScript error when using StreamField on views without commenting support, such as snippets (Jacob Topp-Mugglestone)

## **1.9.7 Wagtail 3.0.2 release notes**

*August 30, 2022*

- *What's new*

### **What's new**

#### **Bug fixes**

- Ensure string representation of `FormSubmission` returns a string (LB (Ben Johnston))
- Fix `updatemodulepaths` command for Python 3.7 (Matt Westcott)
- Fix issue where comments could not be added in StreamField that were already saved (Jacob Topp-Mufflestone)
- Remove outdated reference to `Image.LoaderError` (Matt Westcott)

## **1.9.8 Wagtail 3.0.1 release notes**

*June 16, 2022*

- *What's new*

## What's new

### Other features

- Add warning when `WAGTAILADMIN_BASE_URL` is not configured (Matt Westcott)

### Bug fixes

- Ensure `TabbedInterface` will not show a tab if no panels are visible due to permissions (Paarth Agarwal)
- Specific snippets list language picker was not properly styled (Sage Abdullah)
- Ensure the upgrade notification request for the latest release, which can be disabled via the `WAGTAIL_ENABLE_UPDATE_CHECK` sends the referrer origin with `strict-origin-when-cross-origin` (Karl Hobley)
- Fix misaligned spinner icon on page action button (LB (Ben Johnston))
- Ensure radio buttons / checkboxes display vertically under Django 4.0 (Matt Westcott)
- Prevent failures when splitting blocks at the start or end of a block, or with highlighted text (Jacob Topp-Muggleton)
- Allow scheduled publishing to complete when the initial editor did not have publish permission (Matt Westcott)
- Stop emails from breaking when `WAGTAILADMIN_BASE_URL` is absent due to the request object not being available (Matt Westcott)
- Make try/except on sending email less broad so that legitimate template rendering errors are exposed (Matt Westcott)

## 1.9.9 Wagtail 3.0 release notes

May 16, 2022

- *What's new*
- *Upgrade considerations - changes affecting all projects*
- *Upgrade considerations - deprecation of old functionality*
- *Upgrade considerations - changes affecting Wagtail customisations*

## What's new

### Page editor redesign

This release contains significant UI changes that affect all of Wagtail's admin, largely driven by the implementation of the new Page Editor. These include:

- Fully remove the legacy sidebar, with slim sidebar replacing it for all users (Thibaud Colas)
- Add support for adding custom attributes for link menu items in the slim sidebar (Thibaud Colas)
- Convert all UI code to CSS logical properties for Right-to-Left (RTL) language support (Thibaud Colas)

- Switch the Wagtail branding font and monospace font to a system font stack (Steven Steinwand, Paarth Agarwal, Rishank Kanaparti)
- Remove most uppercased text styles from admin UI (Paarth Agarwal)
- Implement new tabs design across the admin interface (Steven Steinwand)

Other changes that are specific to the Page Editor include:

- Implement new slim page editor header with breadcrumb and secondary page menu (Steven Steinwand, Karl Hobley)
- Move page meta information from the header to a new status side panel component inside of the page editing UI (Steven Steinwand, Karl Hobley)

Further updates to the page editor are expected in the next release. Development on this feature was sponsored by Google.

## **Rich text block splitting**

Rich text blocks within StreamField now provide the ability to split a block at the cursor position, allowing new blocks to be inserted in between. This feature was developed by Jacob Topp-Mugglestone and sponsored by The Motley Fool.

## **Removal of special-purpose field panel types**

The panel types `StreamFieldPanel`, `RichTextFieldPanel`, `ImageChooserPanel`, `DocumentChooserPanel` and `SnippetChooserPanel` have been phased out, and can now be replaced with `FieldPanel`. Additionally, `PageChooserPanel` is only required when passing a `page_type` or `can_choose_root`, and can otherwise be replaced with `FieldPanel`. In all cases, `FieldPanel` will now automatically select the most appropriate form element. This feature was developed by Matt Westcott.

## **Permission-dependent FieldPanels**

`FieldPanel` now accepts a `permission` keyword argument to specify that the field should only be available to users with a given permission level. This feature was developed by Matt Westcott and sponsored by Google as part of Wagtail's page editor redevelopment.

## **Page descriptions**

With every Wagtail Page you are able to add a helpful description text, similar to a `help_text` model attribute. By adding `page_description` to your Page model you'll be adding a short description that can be seen in different places within Wagtail:

```
class LandingPage(Page):
 page_description = "Use this page for converting users"
```

## Image duplicate detection

Trying to upload an image that's a duplicate of one already in the image library will now lead to a confirmation step. This feature was developed by Tidiane Dia and sponsored by The Motley Fool.

## Image renditions can now be prefetched

When using a queryset to render a list of items with images, you can now make use of Django's built-in `prefetch_related()` queryset method to prefetch the renditions needed for rendering with a single extra query. For long lists of items, or where multiple renditions are used for each item, this can provide a significant boost to performance. This feature was developed by Andy Babic.

## Other features

- Upgrade ESLint and Stylelint configurations to latest shared Wagtail configs (Thibaud Colas, Paarth Agarwal)
- Major updates to frontend tooling; move Node tooling from Gulp to Webpack, upgrade to Node v16 and npm v8, eslint v8, stylelint v14 and others (Thibaud Colas)
- Change comment headers' date formatting to use browser APIs instead of requiring a library (LB (Ben Johnston))
- Lint with flake8-comprehensions and flake8-assertive, including adding a pre-commit hook for these (Mads Jensen, Dan Braghis)
- Add black configuration and reformat code using it (Dan Braghis)
- Remove UI code for legacy browser support: polyfills, IE11 workarounds, Modernizr (Thibaud Colas)
- Remove redirect auto-creation recipe from documentation as this feature is now supported in Wagtail core (Andy Babic)
- Remove IE11 warnings (Gianluca De Cola)
- Replace `content_json TextField` with `content JSONField` in `PageRevision` (Sage Abdullah)
- Remove `replace_text` management command (Sage Abdullah)
- Replace `data_json TextField` with `data JSONField` in `BaseLogEntry` (Sage Abdullah)
- Remove the legacy Hallo rich text editor as it has moved to an external package (LB (Ben Johnston))
- Increase the size of checkboxes throughout the UI, and simplify their alignment (Steven Steinwand)
- Adopt [MyST](#) for parsing documentation written in Markdown, replaces recommonmark (LB (Ben Johnston), Thibaud Colas)
- Installing docs extras requirements in CircleCI so issues with the docs requirements are picked up earlier (Thibaud Colas)
- Remove core usage of jinjalint and migrate to curlylint to resolve dependency incompatibility issues (Thibaud Colas)
- Switch focus outlines implementation to `:focus-visible` for cross-browser consistency (Paarth Agarwal)
- Migrate multiple documentation pages from RST to MD - including the editor's guide (Vibhakar Solanki, LB (Ben Johnston), Shwet Khatri)
- Add documentation for defining [`custom form validation`](#) on models used in Wagtail's `modelAdmin` (Serafeim Papastefanos)
- Update `README.md` logo to work for GitHub dark mode (Paarth Agarwal)

- Avoid an unnecessary page reload when pressing enter within the header search bar (Images, Pages, Documents) (Riley de Mestre)
- Removed unofficial length parameter on `If-Modified-Since` header in `sendfile_streaming_backend` which was only used by IE (Mariusz Felisiak)
- Add Pinterest support to the list of default oEmbed providers (Dharmik Gangani)
- Update Jinja2 template support for Jinja2 3.1 (Seb Brown)
- Add ability for `StreamField` to use `JSONField` to store data, rather than `TextField` (Sage Abdullah)
- Split up linting / formatting tasks in Makefile into client and server components (Hitansh Shah)
- Add support for embedding Instagram reels (Luis Nell)
- Use Django's JavaScript catalog feature to manage translatable strings in JavaScript (Karl Hobley)
- Add `trimmed` attribute to all blocktrans tags, so spacing is more reliable in translated strings (Harris Lapiroff)
- Add documentation that describes how to use `ModelAdmin` to manage Tags (Abdulmajeed Isa)
- Rename the setting `BASE_URL` (undocumented) to `WAGTAILADMIN_BASE_URL` and add to documentation, `BASE_URL` will be removed in a future release (Sandil Ranasinghe)
- Validate to and from email addresses within form builder pages when using `AbstractEmailForm` (Jake Howard)
- Add `WAGTAILIMAGES_RENDERING_STORAGE` setting to allow an alternative image rendition storage (Heather White)
- Add `wagtail_update_image_renditions management command` to regenerate image renditions or purge all existing renditions (Hitansh Shah, Onno Timmerman, Damian Moore)
- Add the ability for choices to be separated by new lines instead of just commas within the form builder, commas will still be supported if used (Abdulmajeed Isa)
- Add internationalisation UI to modeladmin (Andrés Martano)
- Support chunking in `PageQuerySet.specify()` to reduce memory consumption (Andy Babic)
- Add useful help text to Tag fields to advise what content is allowed inside tags, including when `TAG_SPACES_ALLOWED` is `True` or `False` (Abdulmajeed Isa)
- Change `AbstractFormSubmission`'s `form_data` to use `JSONField` to store form submissions (Jake Howard)

### Bug fixes

- Update django-treebeard dependency to 4.5.1 or above (Serafeim Papastefanos)
- When using `simple_translations` ensure that the user is redirected to the page edit view when submitting for a single locale (Mitchel Cabuloy)
- When previewing unsaved changes to Form pages, ensure that all added fields are correctly shown in the preview (Joshua Munn)
- When Documents (e.g. PDFs) have been configured to be served inline via `WAGTAILDOCS_CONTENT_TYPES` & `WAGTAILDOCS_INLINE_CONTENT_TYPES` ensure that the filename is correctly set in the Content-Disposition header so that saving the files will use the correct filename (John-Scott Atlakson)
- Improve the contrast of the “Remember me” checkbox against the login page’s background (Steven Steinwand)
- Group permission rows with custom permissions no longer have extra padding (Steven Steinwand)
- Make sure the focus outline of checkboxes is fully around the outer border (Steven Steinwand)

- Consistently set `aria-haspopup="menu"` for all sidebar menu items that have sub-menus (LB (Ben Johnston))
- Make sure `aria-expanded` is always explicitly set as a string in sidebar (LB (Ben Johnston))
- Use a button element instead of a link for page explorer menu item, for the correct semantics and behaviour (LB (Ben Johnston))
- Make sure the “Title” column label can be translated in the page chooser and page move UI (Stephanie Cheng Smith)
- Remove redundant `role="main"` attributes on `<main>` elements causing HTML validation issues (Luis Es-pinoza)
- Allow bulk publishing of pages without revisions (Andy Chosak)
- Stop skipping heading levels in Wagtail welcome page (Jesse Menn)
- Add missing `lang` attributes to `<html>` elements (James Ray)
- Add missing translation usage in Workflow templates (Anuja Verma, Saurabh Kumar)
- Avoid 503 server error when entering tags over 100chars and instead show a user facing validation error (Vu Pham, Khanh Hoang)
- Ensure `thumb_col_header_text` is correctly used by `ThumbnailMixin` within `ModelAdmin` as the column header label (Kyle J. Roux)
- Ensure page copy in Wagtail admin doesn’t ignore `exclude_fields_in_copy` (John-Scott Atlakson)
- Generate new translation keys for translatable `Orderables` when page is copied without being published (Kalob Taulien, Dan Braghis)
- Ignore `GenericRelation` when copying pages (John-Scott Atlakson)
- Implement ARIA tabs markup and keyboards interactions for admin tabs (Steven Steinwand)
- Ensure `wagtail.updatemodulepaths` works when system locale is not UTF-8 (Matt Westcott)
- Re-establish focus trap for Pages explorer in slim sidebar (Thibaud Colas)
- Ensure the icon font loads correctly when `STATIC_URL` is not `"/static/"` (Jacob Topp-Mugglestone)

## Upgrade considerations - changes affecting all projects

### Changes to module paths

Various modules of Wagtail have been reorganised, and imports should be updated as follows:

- The `wagtail.core.utils` module is renamed to `wagtail.coreutils`
- All other modules under `wagtail.core` can now be found under `wagtail` - for example, `from wagtail.core.models import Page` should be changed to `from wagtail.models import Page`
- The `wagtail.tests` module is renamed to `wagtail.test`
- `wagtail.admin.edit_handlers` is renamed to `wagtail.admin.panels`
- `wagtail.contrib.forms.edit_handlers` is renamed to `wagtail.contrib.forms.panels`

These changes can be applied automatically to your project codebase by running the following commands from the project root:

```
wagtail updatemodulepaths --list # list the files to be changed without updating them
wagtail updatemodulepaths --diff # show the changes to be made, without updating files
wagtail updatemodulepaths # actually update the files
```

## Removal of special-purpose field panel types

Within panel definitions on models, `StreamFieldPanel`, `RichTextFieldPanel`, `ImageChooserPanel`, `DocumentChooserPanel` and `SnippetChooserPanel` should now be replaced with `FieldPanel`. Additionally, `PageChooserPanel` can be replaced with `FieldPanel` if it does not use the `page_type` or `can_choose_root` arguments.

## `BASE_URL` setting renamed to `WAGTAILADMIN_BASE_URL`

References to `BASE_URL` in your settings should be updated to `WAGTAILADMIN_BASE_URL`. This setting was not previously documented, but was part of the default project template when starting a project with the `wagtail start` command, and specifies the full base URL for the Wagtail admin, for use primarily in email notifications.

## `use_json_field` argument added to `StreamField`

All uses of `StreamField` should be updated to include the argument `use_json_field=True`. After adding this, make sure to generate and run migrations. This converts the field to use `JSONField` as its internal type instead of `TextField`, which will allow you to use `JSONField` lookups and transforms on the field. This change is necessary to ensure that the database migration is applied; a future release will drop support for `TextField`-based `StreamFields`.

## `SQLite` now requires the `JSON1` extension enabled

Due to `JSONField` requirements, `SQLite` will only be supported with the `JSON1` extension enabled. See [Enabling JSON1 extension on SQLite](#) and [JSON1 extension](#) for details.

## Upgrade considerations - deprecation of old functionality

### Removed support for Internet Explorer (IE11)

IE11 support was officially dropped in Wagtail 2.15, and as of this release there will no longer be a warning shown to users of this browser. Wagtail is fully compatible with Microsoft Edge, Microsoft's replacement for Internet Explorer. You may consider using its [IE mode](#) to keep access to IE11-only sites, while other sites and apps like Wagtail can leverage modern browser capabilities.

## Hallo legacy rich text editor has moved to an external package

Hallo was deprecated in Wagtail v2.0 (February 2018) and has had only a minimal level of support since then. If you still require Hallo for your Wagtail installation, you will need to install the [Wagtail Hallo editor](#) legacy package. We encourage all users of the Hallo editor to take steps to migrate to the new Draftail editor as this external package is unlikely to have ongoing maintenance. `window.registerHalloPlugin` will no longer be created on the page editor load, unless the legacy package is installed.

## Removal of legacy `clean_name` on `AbstractFormField`

If you are upgrading a pre-2.10 project that uses the [\*Wagtail form builder\*](#), and has existing form submission data that needs to be preserved, you must first upgrade to a version between 2.10 and 2.16, and run migrations and start the application server, before upgrading to 3.0. This ensures that the `clean_name` field introduced in Wagtail 2.10 is populated. The mechanism for doing this (which had a dependency on the [Unidecode](#) package) has been dropped in Wagtail 3.0. Any new form fields created under Wagtail 2.10 or above use the [AnyAscii](#) library instead.

## Removed support for Jinja2 2.x

Jinja2 2.x is no longer supported as of this release; if you are using Jinja2 templating on your project, please upgrade to Jinja2 3.0 or above.

## Upgrade considerations - changes affecting Wagtail customisations

### API changes to panels (`EditHandlers`)

Various changes have been made to the internal API for defining panel types, previously known as edit handlers. As noted above, the module `wagtail.admin.edit_handlers` has been renamed to `wagtail.admin.panels`, and `wagtail.contrib.forms.edit_handlers` is renamed to `wagtail.contrib.forms.panels`.

Additionally, the base `wagtail.admin.edit_handlers.EditHandler` class has been renamed to `wagtail.admin.panels.Panel`, and `wagtail.admin.edit_handlers.BaseCompositeEditHandler` has been renamed to `wagtail.admin.panels.PanelGroup`.

Template paths have also been renamed accordingly - templates previously within `wagtailadmin/edit_handlers/` are now located under `wagtailadmin/panels/`, and `wagtailforms/edit_handlers/form_responses_panel.html` is now at `wagtailforms/panels/form_responses.html`.

Where possible, third-party packages that implement their own field panel types should be updated to allow using a plain `FieldPanel` instead, in line with Wagtail dropping its own special-purpose field panel types such as `StreamFieldPanel` and `ImageChooserPanel`. The steps for doing this will depend on the package's functionality, but in general:

- If the panel sets a custom template, your code should instead define a `Widget` class that produces your desired HTML rendering.
- If the panel provides a `widget_overrides` method, your code should instead call `register_form_field_override` so that the desired widget is always selected for the relevant model field type.
- If the panel provides a `get_comparison_class` method, your code should instead call `wagtail.admin.compare.register_comparison_class` to register the comparison class against the relevant model field type.

Within the `Panel` class, the methods `widget_overrides`, `required_fields` and `required_formsets` have been deprecated in favour of a new `get_form_options` method that returns a dict of configuration options to be passed on to the generated form class:

- Panels that define `required_fields` should instead return this value as a `fields` item in the dict returned from `get_form_options`
- Panels that define `required_formsets` should instead return this value as a `formsets` item in the dict returned from `get_form_options`
- Panels that define `widget_overrides` should instead return this value as a `widgets` item in the dict returned from `get_form_options`

The methods `on_request_bound`, `on_instance_bound` and `on_form_bound` are no longer used. In previous versions, over the course of serving a request an edit handler would have the attributes `request`, `model`, `instance` and `form` attached to it, with the corresponding `on_*_bound` method being called at that point. In the new implementation, only the `model` attribute and `on_model_bound` method are still available. This means it is no longer possible to vary or patch the form class in response to per-request information such as the user object. For permission checks, you should use the new `permission` option on `FieldPanel`; for other per-request customisations to the form object, use *a custom form class* with an overridden `__init__` method. (The current user object is available from the form as `self.for_user`.)

Binding to a request, instance and form object is now handled by a new class `Panel.BoundPanel`. Any initialisation logic previously performed in `on_request_bound`, `on_instance_bound` or `on_form_bound` can instead be moved to the constructor method of a subclass of `BoundPanel`:

```
class CustomPanel(Panel):
 class BoundPanel(Panel.BoundPanel):
 def __init__(self, **kwargs):
 super().__init__(**kwargs)
 # The attributes self.panel, self.request, self.instance and self.form
 # are available here
```

The template context for panels derived from `BaseChooserPanel` has changed. `BaseChooserPanel` is deprecated and now functionally identical to `FieldPanel`; as a result, the context variable `is_chosen`, and the variable name given by the panel's `object_type_name` property, are no longer available on the template. The only available variables are now `field` and `show_add_comment_button`. If your template depends on these additional variables, you will need to pass them explicitly by overriding the `BoundPanel.get_context_data` method.

## API changes to ModelAdmin

Some changes of behaviour have been made to `ModelAdmin` as a result of the panel API changes:

- When overriding the `get_form_class` method of a `ModelAdmin` `CreateView` or `EditView` to pass a custom form class, that form class must now inherit from `wagtail.admin.forms.models.WagtailAdminModelForm`. Passing a plain Django `ModelForm` subclass is no longer valid.
- The `ModelAdmin.get_form_fields_exclude` method is no longer passed a `request` argument. Subclasses that override this method should remove this from the method signature. If the `request` object is being used to vary the set of fields based on the user's permission, this can be replaced with the new `permission` option on `FieldPanel`.
- The `ModelAdmin.get_edit_handler` method is no longer passed a `request` or `instance` argument. Subclasses that override this method should remove this from the method signature.

### Replaced content\_json TextField with content JSONField in PageRevision

The `content_json` field in the `PageRevision` model has been renamed to `content`, and this field now internally uses `JSONField` instead of `TextField`. If you have a large number of `PageRevision` objects, running the migrations might take a while.

### Replaced data\_json TextField with data JSONField in BaseLogEntry

The `data_json` field in the `BaseLogEntry` model (and its subclasses `PageLogEntry` and `ModelLogEntry`) has been renamed to `data`, and this field now internally uses `JSONField` instead of `TextField`. If you have a large number of objects for these models, running the migrations might take a while.

### Replaced form\_data TextField with JSONField in AbstractFormSubmission

The `form_data` field in the `AbstractFormSubmission` model (and its subclasses `FormSubmission`) has been converted to `JSONField` instead of `TextField`. If you have customisations that programmatically add form submissions you will need to ensure that the `form_data` that is output is no longer a JSON string but instead a serialisable Python object. When interacting with the `form_data` you will now receive a Python object and not a string.

Example change

```
def process_form_submission(self, form):
 self.get_submission_class().objects.create(
 # form_data=json.dumps(form.cleaned_data, cls=DjangoJSONEncoder),
 form_data=form.cleaned_data, # new
 page=self, user=form.user
)
```

### Removed size argument from `wagtail.utils.sendfile_streaming_backend.was_modified_since`

The `size` argument of the undocumented `wagtail.utils.sendfile_streaming_backend.was_modified_since` function has been removed. This argument was used to add a length parameter to the HTTP header; however, this was never part of the HTTP/1.0 and HTTP/1.1 specifications see [RFC7232](#) and existed only as an unofficial implementation in IE browsers.

## 1.9.10 Wagtail 2.16.3 release notes

September 5, 2022

- *What's new*

## What's new

### Bug fixes

- Ensure the upgrade notification request for the latest release, which can be disabled via the `WAGTAIL_ENABLE_UPDATE_CHECK` sends the referrer origin with `strict-origin-when-cross-origin` (Karl Hobley)
- On the Locked pages report, limit the “locked by” filter to just users who have locked pages (Stefan Hammer)
- Ensure Python 3.10 compatibility when using Elasticsearch backend (Przemysław Buczkowski, Matt Westcott)

## 1.9.11 Wagtail 2.16.2 release notes

April 11, 2022

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

## What's new

### Bug fixes

- Update django-treebeard dependency to 4.5.1 or above (Serafeim Papastefanos)
- Fix permission error when sorting pages having page type restrictions (Thijs Kramer)
- Allow bulk publishing of pages without revisions (Andy Chosak)
- Ensure that all descendant pages are logged when deleting a page, not just immediate children (Jake Howard)
- Refactor `FormPagesListView` in `wagtail.contrib.forms` to avoid undefined `locale` variable when subclassing (Dan Braghis)
- Ensure page copy in Wagtail admin doesn't ignore `exclude_fields_in_copy` (John-Scott Atlakson)
- Generate new translation keys for translatable `Orderables` when page is copied without being published (Kalob Taulien, Dan Braghis)
- Ignore `GenericRelation` when copying pages (John-Scott Atlakson)
- Ensure ‘next’ links from image / document listings do not redirect back to partial AJAX view (Matt Westcott)
- Skip creation of automatic redirects when page cannot be routed (Matt Westcott)
- Prevent JS errors on locale switcher in page chooser (Matt Westcott)

## Upgrade considerations

### Jinja2 compatibility

Developers using Jinja2 templating should note that the template tags in this release (and earlier releases in the 2.15.x and 2.16.x series) are compatible with Jinja2 2.11.x and 3.0.x. Jinja2 2.11.x is unmaintained and requires `markupsafe` to be pinned to version <2.1 to work; Jinja2 3.1.x has breaking changes and is not compatible. We therefore recommend that you use Jinja2 3.0.x, or 2.11.x with fully pinned dependencies.

## 1.9.12 Wagtail 2.16.1 release notes

February 11, 2022

- *What's new*

### What's new

#### Bug fixes

- Ensure that correct sidebar submenus open when labels use non-Latin alphabets (Matt Westcott)
- Fix issue where invalid bulk action URLs would incorrectly trigger a server error (500) instead of a valid not found (404) (Ihor Marhitych)
- Fix issue where bulk actions would not work for object IDs greater than 999 when `USE_THOUSAND_SEPARATOR` (Dennis McGregor)
- Set cookie for sidebar collapsed state to “SameSite: lax” (LB (Ben Johnston))
- Prevent error on creating automatic redirects for sites with non-standard ports (Matt Westcott)
- Restore ability to customise admin UI colours via CSS (LB (Ben Johnston))

## 1.9.13 Wagtail 2.16 release notes

February 7, 2022

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

### What's new

#### Django 4.0 support

This release adds support for Django 4.0.

#### Slim sidebar

As part of a [wider redesign](#) of Wagtail's administration interface, we have replaced the sidebar with a slim, keyboard-friendly version. This re-implementation comes with significant accessibility improvements for keyboard and screen reader users, and will enable us to make navigation between views much snappier in the future. Please have a look at [upgrade considerations](#) for more details on differences with the previous version.

#### Automatic redirect creation

Wagtail projects using the `wagtail.contrib.redirects` app now benefit from 'automatic redirect creation' - which creates redirects for pages and their descendants whenever a URL-impacting change is made; such as a slug being changed, or a page being moved to a different part of the tree.

This feature should be beneficial to most 'standard' Wagtail projects and, in most cases, will have only a minor impact on responsiveness when making such changes. However, if you find this feature is not a good fit for your project, you can disabled it by adding the following to your project settings:

```
WAGTAILREDIRECTS_AUTO_CREATE = False
```

Thank you to The National Archives for kindly sponsoring this feature.

#### Other features

- Added persistent IDs for ListBlock items, allowing commenting and improvements to revision comparisons (Matt Westcott, Tidiane Dia, with sponsorship from [NHS](#))
- Added Aging Pages report (Tidiane Dia)
- Add more SketchFab oEmbed patterns for models (Tom Usher)
- Added `page_slug_changed` signal for Pages (Andy Babic)
- Add collapse option to `StreamField`, `StreamBlock`, and `ListBlock` which will load all sub-blocks initially collapsed (Matt Westcott)
- Private pages can now be fetched over the API (Nabil Khalil)
- Added `alias_of` field to the pages API (Dmitrii Faiazov)
- Add support for Azure CDN and Front Door front-end cache invalidation (Tomasz Knapik)
- Fixed `default_app_config` deprecations for Django  $\geq 3.2$  (Tibor Leupold)
- Removed WOFF fonts
- Improved styling of workflow timeline modal view (Tidiane Dia)
- Add secondary actions menu in edit page headers (Tidiane Dia)
- Add system check for missing core Page fields in `search_fields` (LB (Ben Johnston))

- Improve CircleCI frontend & backend build caches, add automated browser accessibility test suite in CircleCI (Thibaud Colas)
- Add a ‘remember me’ checkbox to the admin sign in form, if unticked (default) the auth session will expire if the browser is closed (Michael Karamuth, Jake Howard)
- When returning to image or document listing views after editing, filters (collection or tag) are now remembered (Tidiane Dia)
- Improve the visibility of field error messages, in Windows high-contrast mode and out (Jason Attwood)
- Improve implementations of visually-hidden text in explorer and main menu toggle (Martin Coote)
- Add locale labels to page listings (Dan Braghis)
- Add locale labels to page reports (Dan Braghis)
- Change release check domain to releases.wagtail.org (Jake Howard)
- Add the user who submitted a page for moderation to the “Awaiting your review” homepage summary panel (Tidiane Dia)
- When moving pages, default to the current parent section (Tidiane Dia)
- Add borders to TypedTableBlock to help visualize rows and columns (Scott Cranfill)
- Set default submit button label on generic create views to ‘Create’ instead of ‘Save’ (Matt Westcott)
- Improve display of image listing for long image titles (Krzysztof Jeziorny)
- Use SVG icons in admin home page site summary items (Jérôme Lebleu)
- Ensure site summary items wrap on smaller devices on the admin home page (Jérôme Lebleu)
- Rework Workflow task chooser modal to align with other chooser modals, using consistent pagination and leveraging class based views (Matt Westcott)
- Implemented a locale switcher on the forms listing page in the admin (Dan Braghis)
- Implemented a locale switcher on the page chooser modal (Dan Braghis)
- Implemented the wagtail\_site template tag for Jinja2 (Vladimir Tananko)
- Change webmaster to website administrator in the admin (Naomi Morduch Toubman)
- Added documentation for creating custom submenus in the admin menu (Sævar Öfjörð Magnússon)
- Choice blocks in StreamField now show label rather than value when collapsed (Jérôme Lebleu)
- Added documentation to clarify configuration of user-uploaded files (Cynthia Kiser)
- Change security contact address to security@wagtail.org (Jake Howard)

## Bug fixes

- Accessibility fixes for Windows high contrast mode; Dashboard icons colour and contrast, help/error/warning blocks for fields and general content, side comment buttons within the page editor, dropdown buttons (Sakshi Uppoor, Shariq Jamil, LB (Ben Johnston), Jason Attwood)
- Rename additional ‘spin’ CSS animations to avoid clashes with other libraries (Kevin Gutiérrez)
- Pages are refreshed from database on create before passing to hooks. Page aliases get correct first\_published\_date and last\_published\_date (Dan Braghis)
- Additional login form fields from WAGTAILADMIN\_USER\_LOGIN\_FORM are now rendered correctly (Michael Karamuth)

- Fix icon only button styling issue on small devices where height would not be set correctly (Vu Pham)
- Add padding to the Draftail editor to ensure ol items are not cut off (Khanh Hoang)
- Prevent opening choosers multiple times for Image, Page, Document, Snippet (LB (Ben Johnston))
- Ensure subsequent changes to styles files are picked up by Gulp watch (Jason Attwood)
- Ensure that programmatic page moves are correctly logged as ‘move’ and not ‘reorder’ in some cases (Andy Babic)

### Upgrade considerations

#### Removed support for Django 3.0 and 3.1

Django 3.0 and 3.1 are no longer supported as of this release; please upgrade to Django 3.2 or above before upgrading Wagtail.

#### Removed support for Python 3.6

Python 3.6 is no longer supported as of this release; please upgrade to Python 3.7 or above before upgrading Wagtail.

#### StreamField ListBlock now returns ListValue rather than a list instance

The data type returned as the value of a ListBlock is now a custom class, `ListValue`, rather than a Python `list` object. This change allows it to provide a `bound_blocks` property that exposes the list items as `BoundBlock objects` rather than plain values. `ListValue` objects are mutable sequences that behave similarly to lists, and so all code that iterates over them, accesses individual elements, or manipulates them should continue to work. However, code that specifically expects a `list` object (e.g. using `isinstance` or testing for equality against a list) may need to be updated. For example, a unit test that tests the value of a ListBlock as follows:

```
self.assertEqual(page.body[0].value, ['hello', 'goodbye'])
```

should be rewritten as:

```
self.assertEqual(list(page.body[0].value), ['hello', 'goodbye'])
```

#### Change to set method on tag fields

This release upgrades the `django-taggit` library to 2.x, which introduces one breaking change: the `TaggableManager.set` method now accepts a list of tags as a single argument, rather than a variable number of arguments. Code such as `page.tags.set('red', 'blue')` should be updated to `page.tags.set(['red', 'blue'])`.

## wagtail.admin.views.generic.DeleteView follows Django 4.0 conventions

The internal (undocumented) class-based view `wagtail.admin.views.generic.DeleteView` has been updated to align with [Django 4.0's DeleteView implementation](#), which uses `FormMixin` to handle POST requests. Any custom deletion logic in `delete()` handlers should be moved to `form_valid()`.

## Renamed admin/expanding-formset.js

`admin/expanding_formset.js` has been renamed to `admin/expanding-formset.js` as part of frontend code clean up work. Check for any customised admin views that are extending expanding formsets, or have overridden template and copied the previous file name used in an import as these may need updating.

## Deprecated sidebar capabilities

The new sidebar largely supports the same customisations as its predecessor, with a few exceptions:

- Top-level menu items should now always provide an `icon_name`, so they can be visually distinguished when the sidebar is collapsed.
- `MenuItem` and its sub-classes no longer supports customising arbitrary HTML attributes.
- `MenuItem` can no longer be sub-classed to customise its HTML output or load additional JavaScript

For sites relying on those capabilities, we provide a `WAGTAIL_SLIM_SIDEBAR = False` setting to switch back to the legacy sidebar. The legacy sidebar and this setting will be removed in Wagtail 2.18.

## 1.9.14 Wagtail 2.15.6 release notes

September 5, 2022

- [What's new](#)

### What's new

#### Bug fixes

- Ensure the upgrade notification request for the latest release, which can be disabled via the `WAGTAIL_ENABLE_UPDATE_CHECK` sends the referrer origin with `strict-origin-when-cross-origin` (Karl Hobley)
- On the Locked pages report, limit the “locked by” filter to just users who have locked pages (Stefan Hammer)
- Ensure Python 3.10 compatibility when using Elasticsearch backend (Przemysław Buczkowski, Matt Westcott)

## 1.9.15 Wagtail 2.15.5 release notes

April 11, 2022

- [What's new](#)
- [Upgrade considerations](#)

### What's new

### Bug fixes

- Allow bulk publishing of pages without revisions (Andy Chosak)
- Ensure that all descendant pages are logged when deleting a page, not just immediate children (Jake Howard)
- Generate new translation keys for translatable Orderable when page is copied without being published (Kalob Taulien, Dan Braghis)
- Ignore *GenericRelation* when copying pages (John-Scott Atlakson)

### Upgrade considerations

### Jinja2 compatibility

Developers using Jinja2 templating should note that the template tags in this release (and earlier releases in the 2.15.x series) are compatible with Jinja2 2.11.x and 3.0.x. Jinja2 2.11.x is unmaintained and requires `markupsafe` to be pinned to version <2.1 to work; Jinja2 3.1.x has breaking changes and is not compatible. We therefore recommend that you use Jinja2 3.0.x, or 2.11.x with fully pinned dependencies.

## 1.9.16 Wagtail 2.15.4 release notes

February 11, 2022

- [What's new](#)

### What's new

### Bug fixes

- Fix issue where invalid bulk action URLs would incorrectly trigger a server error (500) instead of a valid not found (404) (Ihor Marhitych)
- Fix issue where bulk actions would not work for object IDs greater than 999 when USE\_THOUSAND\_SEPARATOR (Dennis McGregor)
- Fix syntax when logging image rendition generation (Jake Howard)

## 1.9.17 Wagtail 2.15.3 release notes

January 26, 2022

- *What's new*

### What's new

### Bug fixes

- Implement correct check for SQLite installations without full-text search support (Matt Westcott)

## 1.9.18 Wagtail 2.15.2 release notes

January 18, 2022

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

### What's new

#### CVE-2022-21683: Comment reply notifications sent to incorrect users

This release addresses an information disclosure issue in Wagtail's commenting feature. Previously, when notifications for new replies in comment threads were sent, they were sent to all users who had replied or commented anywhere on the site, rather than only in the relevant threads. This meant that a user could listen in to new comment replies on pages they did not have editing access to, as long as they had left a comment or reply somewhere on the site.

Many thanks to Ihor Marhitych for reporting this issue. For further details, please see the [CVE-2022-21683 security advisory](#).

### Bug fixes

- Fixed transform operations in Filter.run() when image has been re-oriented (Justin Michalicek)
- Remove extraneous header action buttons when creating or editing workflows and tasks (Matt Westcott)
- Ensure that bulk publish actions pick up the latest draft revision (Matt Westcott)
- Ensure the `checkbox_aria_label` is used correctly in the Bulk Actions checkboxes (Vu Pham)
- Prevent error on MySQL search backend when searching three or more terms (Aldán Creo)
- Allow wagtail.search app migrations to complete on versions of SQLite without full-text search support (Matt Westcott)
- Update Pillow dependency to allow 9.x (Matt Westcott)

## Upgrade considerations

### Support for SQLite without full-text search support

This release restores the ability to run Wagtail against installations of SQLite that do not include the `fts5` extension for full-text search support. On these installations, the fallback search backend (without support for full-text queries) will be used, and the database table for storing indexed content will not be created.

If SQLite is subsequently upgraded to a version with `fts5` support, existing databases will still be missing this table, and full-text search will continue to be unavailable until it is created. To correct this, first make a backup copy of the database (since rolling back the migration could potentially reverse other schema changes), then run:

```
./manage.py migrate wagtailsearch 0005
./manage.py migrate
./manage.py update_index
```

Additionally, since the database search backend now needs to run a query on initialisation to check for the presence of this table, calling `wagtail.search.backends.get_search_backend` during application startup may now fail with a “Models aren’t loaded yet” error. Code that does this should be updated to only call `get_search_backend` at the point when a search query is to be performed.

## 1.9.19 Wagtail 2.15.1 release notes

*November 11, 2021*

- *What’s new*

### What’s new

### Bug fixes

- Fix syntax when logging image rendition generation (Jake Howard)
- Increase version range for django-filter dependency (Serafeim Papastefanos)
- Prevent bulk action checkboxes from displaying on page reports and other non-explorer listings (Matt Westcott)
- Fix errors on publishing pages via bulk actions (Matt Westcott)
- Fix `csrf_token` issue when using the Approve or Unlock buttons on pages on the Wagtail admin home (Matt Westcott)

## 1.9.20 Wagtail 2.15 release notes

*November 4, 2021*

- *What’s new*
- *Upgrade considerations*

Wagtail 2.15 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 12 months).

## What's new

### New database search backend

Wagtail has a new search backend that uses the full-text search features of the database in use. It supports SQLite, PostgreSQL, MySQL, and MariaDB.

This new search backend replaces both the existing generic database and PostgreSQL-specific search backends.

To switch to this new backend, see the upgrade considerations below.

This feature was developed by Aldán Creo as part of Google Summer of Code.

### Bulk actions

Bulk actions are now available for Page, User, Image and Document models in the Wagtail Admin, allowing users to perform actions like publication or deletion on groups of objects at once.

This feature was developed by Shohan Dutta Roy, mentored by Dan Braghis, Jacob Topp-Muggleton, and Storm Heg.

### Audit logging for all models

Audit logging has been extended so that all models (not just pages) can have actions logged against them. The Site History report now includes logs from all object types, and snippets and ModelAdmin provide a history view showing previous edits to an object. This feature was developed by Matt Westcott, and sponsored by [The Motley Fool](#).

### Collection management permissions

Permission for managing collections can now be assigned to individual subtrees of the collection hierarchy, allowing sub-teams within a site to control how their images and documents are organised. For more information, see [Collection management permissions](#). This feature was developed by Cynthia Kiser.

### Typed table block

A new `TypedTableBlock` block type is available for StreamField, allowing authors to create tables where the cell values are any StreamField block type, including rich text. For more information, see [Typed table block](#). This feature was developed by Matt Westcott, Coen van der Kamp and Scott Cranfill, and sponsored by YouGov.

## Windows high contrast support

As part of a broad push to improve the accessibility of the administration interface, Wagtail now supports [Windows high contrast mode](#). There are remaining known issues but we are confident Wagtail is now much more usable for people relying on this assistive technology.

Individual fixes were implemented by a large number of first-time and seasoned contributors:

- Comments icon now matches link colour (Dmitrii Faiazov, LB (Ben Johnston))
- Sidebar logo is now visible in high contrast mode (Dmitrii Faiazov, LB (Ben Johnston))
- Icons in links and buttons now use the appropriate “active control” colour (Dmitrii Faiazov, LB (Ben Johnston))
- Comments dropdown now has a border (Shariq Jamil, LB (Ben Johnston))
- Make StreamField block chooser menu buttons appear as buttons (Dmitrii Faiazov, LB (Ben Johnston))
- Add a separator to identify the search forms (Dmitrii Faiazov, LB (Ben Johnston))
- Update tab styles so the active tab can be identified (Dmitrii Faiazov, LB (Ben Johnston))
- Make hamburger menu a button for tab and high contrast accessibility (Amy Chan, Dan Braghis)
- Tag fields now have the correct background (Desai Akshata, LB (Ben Johnston))
- Added sidebar vertical separation with main content (Onkar Apte, LB (Ben Johnston))
- Added vertical separation between field panels (Chakita Muttaraju, LB (Ben Johnston))
- Switch widgets on/off states are now visually distinguishable (Sakshi Uppoor, Thibaud Colas)
- Checkbox widgets on/off states are now visually distinguishable (Thibaud Colas, Jacob Topp-Mugglestone, LB (Ben Johnston))

Particular thanks to LB, who reviewed almost all of those contributions, and Kyle Bayliss, who did the initial audit to identify High contrast mode issues.

## Other features

- Add the ability for the page chooser to convert external urls that match a page to internal links, see [WAGTAILADMIN\\_EXTERNAL\\_LINK\\_CONVERSION](#) (Jacob Topp-Mufflestone. Sponsored by The Motley Fool)
- Added “Extending Wagtail” section to documentation (Matt Westcott)
- Introduced [template components](#), a standard mechanism for renderable objects in the admin (Matt Westcott)
- Support `min_num` / `max_num` options on ListBlock (Matt Westcott)
- Implemented automatic tree synchronisation for `contrib.simple_translation` (Mitchel Cabuloy)
- Added a `background_position_style` property to renditions. This can be used to crop images using its focal point in the browser. See [Setting the background-position inline style based on the focal point](#) (Karl Hobley)
- Added a distinct `wagtail.copy_for_translation` log action type (Karl Hobley)
- Add a debug logger around image rendition generation (Jake Howard)
- Convert Documents and Images to class based views for easier overriding (Matt Westcott)
- Isolate admin URLs for Documents and Images search listing results with the name ‘`listing_results`’ (Matt Westcott)
- Removed `request.is_ajax()` usage in Documents, Image and Snippet views (Matt Westcott)

- Simplify generic admin view templates plus ensure `page_title` and `page_subtitle` are used consistently (Matt Westcott)
- Extend support for *collapsing edit panels* from just MultiFieldPanels to all kinds of panels (Fabien Le Frapper, Robbie Mackay)
- Add object count to header within modeladmin listing view (Jonathan “Yoni” Knoll)
- Add ability to return HTML in multiple image upload errors (Gordon Pendleton)
- Upgrade internal JS tooling; Node v14 plus other smaller package upgrades (LB (Ben Johnston))
- Add support for `non_field_errors` rendering in Workflow action modal (LB (Ben Johnston))
- Support calling `get_image_model` and `get_document_model` at import time (Matt Westcott)
- When copying a page, default the ‘Publish copied page’ field to false (Justin Slay)
- Open Preview and Live page links in the same tab, except where it would interrupt editing a Page (Sagar Agarwal)
- Added `ExcelDateFormatter` to `wagtail.admin.views.mixins` so that dates in Excel exports will appear in the locale’s `SHORT_DATETIME_FORMAT` (Andrew Stone)
- Add TIDAL support to the list of oEmbed providers (Wout De Puyseleir)
- Add `label_format` attribute to customise the label shown for a collapsed StructBlock (Matt Westcott)
- User Group permissions editing in the admin will now show all custom object permissions in one row instead of a separate table (Kamil Marut)
- Create `ImageFileMixin` to extract shared file handling methods from `AbstractImage` and `AbstractRendition` (Fabien Le Frapper)
- Add `before_delete_page` and `register_permissions` examples to Hooks documentation (Jane Liu, Daniel Fairhead)
- Add clarity to modeladmin template override behaviour in the documentation (Joe Howard, Dan Swain)
- Add section about CSV exports to security documentation (Matt Westcott)
- Add initial support for Django 4.0 deprecations (Matt Westcott, Jochen Wersdörfer)
- Translations in `n1_NL` are moved to the `n1.po` files. `n1_NL` translation files are deleted. Projects that use `LANGUAGE_CODE = 'n1-nl'` will automatically fallback to `n1`. (Loïc Teixeira, Coen van der Kamp)
- Add documentation for how to redirect to a separate page on Form builder submissions using `RoutablePageMixin` (Nick Smith)
- Refactored index listing views and made column sort-by headings more consistent (Matt Westcott)
- The title field on Image and Document uploads will now default to the filename without the file extension and this behaviour can be customised (LB Johnston)
- Add support for Python 3.10 (Matt Westcott)
- Introduce, `autocomplete`, a separate method which performs partial matching on specific autocomplete fields. This is useful for suggesting pages to the user in real-time as they type their query. (Karl Hobley, Matt Westcott)
- Use SVG icons in modeladmin headers and StreamField buttons/headers (Jérôme Lebleu)
- Add tags to existing Django registered checks (LB Johnston)
- Upgrade admin frontend JS libraries jQuery to 3.6.0 (Fabien Le Frapper)
- Added `request.preview_mode` so that template rendering can vary based on preview mode (Andy Chosak)

## Bug fixes

- Delete button is now correct colour on snippets and modeladmin listings (Brandon Murch)
- Ensure that StreamBlock / ListBlock-level validation errors are counted towards error counts (Matt Westcott)
- InlinePanel add button is now keyboard navigatable (Jesse Menn)
- Remove redundant ‘clear’ button from site root page chooser (Matt Westcott)
- Make ModelAdmin IndexView keyboard-navigable (Saptak Sengupta)
- Prevent error on refreshing page previews when multiple preview tabs are open (Alex Tomkins)
- Menu sidebar hamburger icon on smaller viewports now correctly indicates it is a button to screen readers and can be accessed via keyboard (Amy Chan, Dan Braghis)
- `blocks.MultipleChoiceBlock`, `forms.CheckboxSelectMultiple` and `ArrayField` checkboxes will now stack instead of display inline to align with all other checkboxes fields (Seb Brown)
- Screen readers can now access login screen field labels (Amy Chan)
- Admin breadcrumbs home icon now shows for users with access to a subtree only (Stefan Hammer)
- Add handling of invalid inline styles submitted to RichText so `ConfigException` is not thrown (Alex Tomkins)
- Ensure comment notifications dropdown handles longer translations without overflowing content (Krzysztof Jeziorny)
- Set `default_auto_field` in `postgres_search AppConfig` (Nick Moreton)
- Ensure admin tab JS events are handled on page load (Andrew Stone)
- `EmailNotificationMixin` and `send_notification` should only send emails to active users (Bryan Williams)
- Disable Task confirmation now shows the correct value for quantity of tasks in progress (LB Johnston)
- Page history now works correctly when it contains changes by a deleted user (Dan Braghis)
- Add `gettext_lazy` to `ModelAdmin` built in view titles so that language settings are correctly used (Matt Westcott)
- Tabbing and keyboard interaction on the Wagtail userbar now aligns with ARIA best practices (Storm Heg)
- Add full support for custom `edit_handler` usage by adding missing `bind_to` call to `PreviewOnEdit` view (Stefan Hammer)
- Only show active (not disabled) tasks in the workflow task chooser (LB Johnston)
- CSS build scripts now output to the correct directory paths on Windows (Vince Salvino)
- Capture log output from style fallback to avoid noise in unit tests (Matt Westcott)
- Nested InlinePanel usage no longer fails to save when creating two or more items (Indresh P, Rinish Sam, Anirudh V S)
- Changed relation name used for admin commenting from `comments` to `wagtail_admin_comments` to avoid conflicts with third-party commenting apps (Matt Westcott)
- CSS variables are now correctly used for the filtering menu in modeladmin (Noah H)
- Panel heading attribute is no longer ignored when nested inside a `MultiFieldPanel` (Jérôme Lebleu)

## Upgrade considerations

### Database search backends replaced

The following search backends (configured in `WAGTAILSEARCH_BACKENDS`) have been deprecated:

- `wagtail.search.backends.db` (the default if `WAGTAILSEARCH_BACKENDS` is not specified)
- `wagtail.contrib.postgres_search.backend`

Both of these backends have now been replaced by `wagtail.search.backends.database`. This new backend supports all of the features of the PostgreSQL backend, and also supports other databases. It will be made the default backend in Wagtail 3.0. To enable the new backend, edit (or add) the `WAGTAILSEARCH_BACKENDS` setting as follows:

```
WAGTAILSEARCH_BACKENDS = {
 'default': {
 'BACKEND': 'wagtail.search.backends.database',
 }
}
```

Also remove '`wagtail.contrib.postgres_search`' from `INSTALLED_APPS` if this was previously set.

After switching to this backend, you will need to run the `manage.py update_index` management command to populate the search index (see [update\\_index](#)).

If you have used the PostgreSQL-specific `SEARCH_CONFIG`, this will continue to work as before with the new backend. For example:

```
WAGTAILSEARCH_BACKENDS = {
 'default': {
 'BACKEND': 'wagtail.search.backends.database',
 'SEARCH_CONFIG': 'english',
 }
}
```

However, as a PostgreSQL specific feature, this will be ignored when using a different database.

### Admin homepage panels, summary items and action menu items now use components

Several Wagtail hooks provide a mechanism for passing Python objects to be rendered as HTML inside admin views, and the APIs for these objects have been updated to adopt a common [\*template components\*](#) pattern. The affected objects are:

- Homepage panels (as registered with the `construct_homepage_panels` hook)
- Homepage summary items (as registered with the `construct_homepage_summary_items` hook)
- Page action menu items (as registered with the `register_page_action_menu_item` and `construct_page_action_menu` hooks)
- Snippet action menu items (as registered with the `register_snippet_action_menu_item` and `construct_snippet_action_menu` hooks)

User code that creates these objects should be updated to follow the component API. This will typically require the following changes:

- Homepage panels should be made subclasses of `wagtail.admin.ui.components.Component`, and the `render(self)` method should be changed to `render_html(self, parent_context)`. (Alternatively, rather

than defining `render_html`, it may be more convenient to reimplement it with a template, as per [Creating components](#).)

- Summary item classes can continue to inherit from `wagtail.admin.site_summary.SummaryItem` (which is now a subclass of `Component`) as before, but:
  - Any `template` attribute should be changed to `template_name`;
  - Any place where the `render(self)` method is overridden should be changed to `render_html(self, parent_context)`;
  - Any place where the `get_context(self)` method is overridden should be changed to `get_context_data(self, parent_context)`.
- Action menu items for pages and snippets can continue to inherit from `wagtail.admin.action_menu.ActionMenuItem` and `wagtail.snippets.action_menu.ActionMenuItem` respectively - these are now subclasses of `Component` - but:
  - Any `template` attribute should be changed to `template_name`;
  - Any `get_context` method should be renamed to `get_context_data`;
  - The `get_url`, `is_shown`, `get_context_data` and `render_html` methods no longer accept a `request` parameter. The `request` object is available in the context dictionary as `context['request']`.

### Passing callables as messages in `register_log_actions` is deprecated

When defining new action types for [audit logging](#) with the `register_log_actions` hook, it was previously possible to pass a callable as the message. This is now deprecated - to define a message that depends on the log entry's data, you should now create a subclass of `wagtail.core.log_actions.LogFormatter`. For example:

```
from django.utils.translation import gettext_lazy as _
from wagtail.core import hooks

@hooks.register('register_log_actions')
def additional_log_actions(actions):

 def greeting_message(data):
 return _('Hello %(audience)s') % {
 'audience': data['audience'],
 }
 actions.register_action('wagtail_package.greet_audience', _('Greet audience'), greeting_message)
```

should now be rewritten as:

```
from django.utils.translation import gettext_lazy as _
from wagtail.core import hooks
from wagtail.core.log_actions import LogFormatter

@hooks.register('register_log_actions')
def additional_log_actions(actions):

 @actions.register_action('wagtail_package.greet_audience')
 class GreetingActionFormatter(LogFormatter):
 label = _('Greet audience')
```

(continues on next page)

(continued from previous page)

```
def format_message(self, log_entry):
 return _('Hello %(audience)s') % {
 'audience': log_entry.data['audience'],
 }
```

### PageLogEntry.objects.log\_action is deprecated

Audit logging is now supported on all model types, not just pages, and so the `PageLogEntry.objects.log_action` method for logging actions performed on pages is deprecated in favour of the general-purpose `log` function. Code that calls `PageLogEntry.objects.log_action` should now import the `log` function from `wagtail.core.log_actions` and call this instead (all arguments are unchanged).

Additionally, for logging actions on non-Page models, it is generally no longer necessary to subclass `BaseLogEntry`; see *Audit log* for further details.

### Removed support for Internet Explorer (IE11)

If this affects you or your organisation, consider which alternative browsers you may be able to use. Wagtail is fully compatible with Microsoft Edge, Microsoft's replacement for Internet Explorer. You may consider using its IE mode to keep access to IE11-only sites, while other sites and apps like Wagtail can leverage modern browser capabilities.

### search() method partial match future deprecation

Before the `autocomplete()` method was introduced, the search method also did partial matching. This behaviour is will be deprecated in a future release and you should either switch to the new `autocomplete()` method or pass `partial_match=False` into the search method to opt-in to the new behaviour. The partial matching in `search()` will be completely removed in a future release. See: [Searching QuerySets](#)

### Change of relation name for admin comments

The `related_name` of the relation linking the Page and User models to admin comments has been changed from `comments` to `wagtail_admin_comments`, to avoid conflicts with third-party apps that implement commenting. If you have any code that references the `comments` relation (including fixture files), this should be updated to refer to `wagtail_admin_comments` instead. If this is not feasible, the previous behaviour can be restored by adding `WAGTAIL_COMMENTS_RELATION_NAME = 'comments'` to your project's settings.

Reusable library code that needs to preserve backwards compatibility with previous Wagtail versions can find out the relation name as follows:

```
try:
 from wagtail.core.models import COMMENTS_RELATION_NAME
except ImportError:
 COMMENTS_RELATION_NAME = 'comments'
```

## Bulk action views not covered by existing hooks

Bulk action views provide alternative routes to actions like publishing or copying a page. If your site relies on hooks like `before_publish_page` or `before_copy_page` to perform checks, or add additional functionality, those hooks will not be called on the corresponding bulk action views. If you want to add this to the bulk action views as well, use the new bulk action hooks: `before_bulk_action` and `after_bulk_action`.

## 1.9.21 Wagtail 2.14.2 release notes

October 14, 2021

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

### What's new

#### Bug fixes

- Allow relation name used for admin commenting to be overridden to avoid conflicts with third-party commenting apps (Matt Westcott)
- Corrected badly-formed format strings in translations (Matt Westcott)
- Page history now works correctly when it contains changes by a deleted user (Dan Braghis)

#### Upgrade considerations

##### Customising relation name for admin comments

The admin commenting feature introduced in Wagtail 2.13 added a relation named `comments` to the Page and User models. This can cause conflicts with third-party apps that implement commenting functionality, and so this will be renamed to `wagtail_admin_comments` in Wagtail 2.15. Developers who are affected by this issue, and have thus been unable to upgrade to Wagtail 2.13 or above, can now “opt in” to the Wagtail 2.15 behaviour by adding the following line to their project settings:

```
WAGTAIL_COMMENTS_RELATION_NAME = 'wagtail_admin_comments'
```

This will allow third-party commenting apps to work in Wagtail 2.14.2 alongside Wagtail’s admin commenting functionality.

Reusable library code that needs to preserve backwards compatibility with previous Wagtail versions can find out the relation name as follows:

```
try:
 from wagtail.core.models import COMMENTS_RELATION_NAME
except ImportError:
 COMMENTS_RELATION_NAME = 'comments'
```

## 1.9.22 Wagtail 2.14.1 release notes

August 12, 2021

- [What's new](#)

### What's new

#### Bug fixes

- Prevent failure on Twitter embeds and others which return cache\_age as a string (Matt Westcott)
- Fix Uncaught ReferenceError when editing links in Hallo (Cynthia Kiser)

## 1.9.23 Wagtail 2.14 release notes

August 2, 2021

- [What's new](#)
- [Upgrade considerations](#)

### What's new

#### New features

- Added `ancestor_of` API filter. See [Filtering by tree position \(pages only\)](#). (Jaap Roes)
- Added support for customising group management views. See [Customising group edit/create views](#). (Jan Seifert)
- Added `full_url` property to image renditions (Shreyash Srivastava)
- Added locale selector when choosing translatable snippets (Karl Hobley)
- Added `WAGTAIL_WORKFLOW_ENABLED` setting for enabling / disabling moderation workflows globally (Matt Westcott)
- Allow specifying `max_width` and `max_height` on EmbedBlock (Petr Dlouhý)
- Add warning when StreamField is used without a StreamFieldPanel (Naomi Morduch Toubman)
- Added keyboard and screen reader support to Wagtail user bar (LB Johnston, Storm Heg)
- Added instructions on copying and aliasing pages to the editor's guide in documentation (Vlad Podgurschi)
- Add Google Data Studio to the list of oEmbed providers (Petr Dlouhý)
- Allow ListBlock to raise validation errors that are not attached to an individual child block (Matt Westcott)
- Use `DATETIME_FORMAT` for localization in templates (Andrew Stone)
- Added documentation on multi-site, multi instance and multi tenancy setups (Coen Van Der Kamp)
- Updated Facebook / Instagram oEmbed endpoints to v11.0 (Thomas Kremmel)
- Performance improvements for admin listing pages (Jake Howard, Dan Braghis, Tom Usher)

## Bug fixes

- Invalid filter values for foreign key fields in the API now give an error instead of crashing (Tidiane Dia)
- Ordering specified in the `construct_explorer_page_queryset` hook is now taken into account again by the page explorer API (Andre Fonseca)
- Deleting a page from its listing view no longer results in a 404 error (Tidiane Dia)
- The Wagtail admin urls will now respect the `APPEND_SLASH` setting (Tidiane Dia)
- Prevent “Forgotten password” link from overlapping with field on mobile devices (Helen Chapman)
- Snippet admin urls are now namespaced to avoid ambiguity with the primary key component of the url (Matt Westcott)
- Prevent error on copying pages with ClusterTaggableManager relations and multi-level inheritance (Chris Pollard)
- Prevent failure on root page when registering the Page model with ModelAdmin (Jake Howard)
- Prevent error when filtering page search results with a malformed `content_type` (Chris Pollard)
- Prevent multiple submissions of “update” form when uploading images / documents (Mike Brown)
- Ensure HTML title is populated on project template 404 page (Matt Westcott)
- Respect `cache_age` parameters on embeds (Gordon Pendleton)
- Page comparison view now reflects request-level customisations to edit handlers (Matt Westcott)
- Add `block.super` to remaining `extra_js` & `extra_css` blocks (Andrew Stone)
- Ensure that `editor` and `features` arguments on RichTextField are preserved by `clone()` (Daniel Fairhead)
- Rename ‘spin’ CSS animation to avoid clashes with other libraries (Kevin Gutiérrez)
- Prevent crash when copying a page from a section where the user has no publish permission (Karl Hobley)
- Ensure that rich text conversion correctly handles images / embeds inside links or inline styles (Matt Westcott)

## Upgrade considerations

### Removed support for Django 2.2

Django 2.2 is no longer supported as of this release; please upgrade to Django 3.0 or above before upgrading Wagtail.

### User bar with keyboard and screen reader support

The Wagtail user bar (“edit bird”) widget now supports keyboard and screen reader navigation. To make the most of this, we now recommend placing the widget near the top of the page `<body>`, so users can reach it without having to go through the whole page. See [Wagtail User Bar](#) for more information.

For implementers of custom user bar menu items, we also now require the addition of `role="menuitem"` on the `a` element to provide the correct semantics. See [construct\\_wagtail\\_userbar](#) for more information.

## Deprecation of Facebook / Instagram oEmbed product

As of June 2021, the procedure for setting up a Facebook app to handle Facebook / Instagram embedded content (see [Facebook and Instagram](#)) has changed. It is now necessary to activate the “oEmbed Read” feature on the app, and submit it to Facebook for review. Apps that activated the oEmbed Product before June 8, 2021 must be migrated to oEmbed Read by September 7, 2021 to continue working. No change to the Wagtail code or configuration is required.

## 1.9.24 Wagtail 2.13.5 release notes

October 14, 2021

- [What's new](#)
- [Upgrade considerations](#)

### What's new

#### Bug fixes

- Allow relation name used for admin commenting to be overridden to avoid conflicts with third-party commenting apps (Matt Westcott)
- Corrected badly-formed format strings in translations (Matt Westcott)
- Correctly handle non-numeric user IDs for deleted users in reports (Dan Braghis)

### Upgrade considerations

#### Customising relation name for admin comments

The admin commenting feature introduced in Wagtail 2.13 added a relation named `comments` to the Page and User models. This can cause conflicts with third-party apps that implement commenting functionality, and so this will be renamed to `wagtail_admin_comments` in Wagtail 2.15. Developers who are affected by this issue, and have thus been unable to upgrade to Wagtail 2.13 or above, can now “opt in” to the Wagtail 2.15 behaviour by adding the following line to their project settings:

```
WAGTAIL_COMMENTS_RELATION_NAME = 'wagtail_admin_comments'
```

This will allow third-party commenting apps to work in Wagtail 2.13.5 alongside Wagtail’s admin commenting functionality.

Reusable library code that needs to preserve backwards compatibility with previous Wagtail versions can find out the relation name as follows:

```
try:
 from wagtail.core.models import COMMENTS_RELATION_NAME
except ImportError:
 COMMENTS_RELATION_NAME = 'comments'
```

## 1.9.25 Wagtail 2.13.4 release notes

*July 13, 2021*

- *What's new*

### What's new

### Bug fixes

- Prevent embed thumbnail\_url migration from failing on URLs longer than 200 characters (Matt Westcott)

## 1.9.26 Wagtail 2.13.3 release notes

*July 5, 2021*

- *What's new*

### What's new

### Bug fixes

- Prevent error when using rich text on views where commenting is unavailable (Jacob Topp-Muggleton)
- Include form media on account settings page (Matt Westcott)
- Avoid error when rendering validation error messages on ListBlock children (Matt Westcott)
- Prevent comments CSS from overriding admin UI colour customisations (Matt Westcott)
- Avoid validation error when editing rich text content preceding a comment (Jacob Topp-Muggleton)

## 1.9.27 Wagtail 2.13.2 release notes

*June 17, 2021*

- *What's new*

## What's new

### CVE-2021-32681: Improper escaping of HTML ('Cross-site Scripting') in Wagtail StreamField blocks

This release addresses a cross-site scripting (XSS) vulnerability in StreamField. When the `{% include_block %}` template tag is used to output the value of a plain-text StreamField block (CharBlock, TextBlock or a similar user-defined block derived from FieldBlock), and that block does not specify a template for rendering, the tag output is not properly escaped as HTML. This could allow users to insert arbitrary HTML or scripting. This vulnerability is only exploitable by users with the ability to author StreamField content (i.e. users with 'editor' access to the Wagtail admin).

Site implementors who wish to retain the existing behaviour of allowing editors to insert HTML content in these blocks (and are willing to accept the risk of untrusted editors inserting arbitrary code) may disable the escaping by surrounding the relevant `{% include_block %}` tag in `{% autoescape off %}...{% endautoescape %}`.

Many thanks to Karen Tracey for reporting this issue. For further details, please see [the CVE-2021-32681 security advisory](#).

## 1.9.28 Wagtail 2.13.1 release notes

June 1, 2021

- *What's new*

## What's new

### Bug fixes

- Ensure comment notification checkbox is fully hidden when commenting is disabled (Karl Hobley)
- Prevent commenting from failing for user models with UUID primary keys (Jacob Topp-Muggleton)
- Fix incorrect link in comment notification HTML email (Matt Westcott)

## 1.9.29 Wagtail 2.13 release notes

May 12, 2021

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

## What's new

### StreamField performance and functionality updates

The StreamField editing interface has been rebuilt on a client-side rendering model, powered by the [telepath](#) library. This provides better performance, increased customisability and UI enhancements including the ability to duplicate blocks. For further background, see the blog post [Telepath - the next evolution of StreamField](#).

This feature was developed by Matt Westcott and Karl Hobley and sponsored by [YouGov](#), inspired by earlier work on [react-streamfield](#) completed by Bertrand Bordage through the [Wagtail's First Hatch](#) crowdfunder.

### Simple translation module

In Wagtail 2.12 we shipped the new localisation support, but in order to translate content an external library had to be used, such as [wagtail-localize](#).

In this release, a new contrib app has been introduced called [\*simple\\_translation\*](#). This allows you to create copies of pages and translatable snippets in other languages and translate them as regular Wagtail pages. It does not include any more advanced translation features such as using external services, PO files, or an interface that helps keep translations in sync with the original language.

This module was contributed by Coen van der Kamp.

### Commenting

The page editor now supports [\*leaving comments on fields and StreamField blocks\*](#), by entering commenting mode (using the button in the top right of the editor). Inline comments are available in rich text fields using the Draftail editor.

This feature was developed by Jacob Topp-Mugglestone, Karl Hobley and Simon Evans and sponsored by [The Motley Fool](#).

### Combined account settings

The “Account settings” section available at the bottom of the admin menu has been updated to include all settings on a single form. This feature was developed by Karl Hobley.

### Redirect export

The redirects module now includes support for exporting the list of redirects to XLSX or CSV. This feature was developed by Martin Sandström.

### Sphinx Wagtail Theme

The documentation now uses our brand new [Sphinx Wagtail Theme](#), with a search feature powered by [Algolia DocSearch](#).

Feedback and feature requests for the theme may be reported to the [sphinx\\_wagtail\\_theme](#) issue list, and to Wagtail’s issues for the search.

Thank you to Storm Heg, Tibor Leupold, Thibaud Colas, Coen van der Kamp, Olly Willans, Naomi Morduch Toubman, Scott Cranfill, and Andy Chosak for making this happen!

## Django 3.2 support

Django 3.2 is formally supported in this release. Note that Wagtail 2.13 will be the last release to support Django 2.2.

## Other features

- Support passing `min_num`, `max_num` and `block_counts` arguments directly to `StreamField` (Haydn Greatnews, Matt Westcott)
- Add the option to set rich text images as decorative, without alt text (Helen Chapman, Thibaud Colas)
- Add support for `__year` filter in Elasticsearch queries (Seb Brown)
- Add `PageQuerySet.defer_streamfields()` (Andy Babic)
- Utilize `PageQuerySet.defer_streamfields()` to improve efficiency in a few key places (Andy Babic)
- Support passing multiple models as arguments to `type()`, `not_type()`, `exact_type()` and `not_exact_type()` methods on `PageQuerySet` (Andy Babic)
- Update default attribute copying behaviour of `Page.get_specific()` and added the `copy_attrs_exclude` option (Andy Babic)
- Update `PageQueryset.specific(defer=True)` to only perform a single database query (Andy Babic)
- Switched `register_setting`, `register_settings_menu_item` to use SVG icons (Thibaud Colas)
- Add support to SVG icons for `SearchArea` subclasses in `register_admin_search_area` (Thibaud Colas)
- Add specialized `wagtail.reorder` page audit log action. This was previously covered by the `wagtail.move` action (Storm Heg)
- `get_settings` template tag now supports specifying the variable name with `{% get_settings as var %}` (Samir Shah)
- Reinstate submitter's name on moderation notification email (Matt Westcott)
- Add a new switch input widget as an alternative to checkboxes (Karl Hobley)
- Allow `{% pageurl %}` fallback to be a direct URL or an object with a `get_absolute_url` method (Andy Babic)
- Support slicing on `StreamField` / `StreamBlock` values (Matt Westcott)
- Switch Wagtail choosers to use SVG icons instead of font icon (Storm Heg)
- Save revision when restart workflow (Ihor Marhitych)
- Add a visible indicator of unsaved changes to the page editor (Jacob Topp-Muggleton)

## Bug fixes

- `StreamField` required status is now consistently handled by the `blank` keyword argument (Matt Westcott)
- Show ‘required’ asterisks for blocks inside required `StreamFields` (Matt Westcott)
- Make image chooser “Select format” fields translatable (Helen Chapman, Thibaud Colas)
- Fix pagination on ‘view users in a group’ (Sagar Agarwal)
- Prevent page privacy menu from being triggered by pressing enter on a char field (Sagar Agarwal)
- Validate host/scheme of return URLs on password authentication forms (Susan Dreher)

- Reordering a page now includes the correct user in the audit log (Storm Heg)
- Fix reverse migration errors in images and documents (Mike Brown)
- Make “Collection” and “Parent” form field labels translatable (Thibaud Colas)
- Apply enough chevron padding to all applicable select elements (Scott Cranfill)
- Reduce database queries in the page edit view (Ihor Marhitych)

### Upgrade considerations

#### End of Internet Explorer 11 support

Wagtail 2.13 will be the last Wagtail release to support IE11. Users accessing the admin with IE11 will be shown a warning message advising that support is being phased out.

#### Updated handling of non-required StreamFields

The rules for determining whether a StreamField is required (i.e. at least one block must be provided) have been simplified and made consistent with other field types. Non-required fields are now indicated by `blank=True` on the StreamField definition; the default is `blank=False` (the field is required). In previous versions, to make a field non-required, it was necessary to define [a top-level StreamBlock](#) with `required=False` (which applied the validation rule) as well as setting `blank=True` (which removed the asterisk from the form field). You should review your use of StreamField to check that `blank=True` is used on the fields you wish to make optional.

#### New client-side implementation for custom StreamField blocks

For the majority of cases, the new StreamField implementation in this release will be a like-for-like upgrade, and no code changes will be necessary - this includes projects where custom block types have been defined by extending `StructBlock`, `ListBlock` and `StreamBlock`. However, certain complex customisations may need to be reimplemented to work with the new client-side rendering model:

- When customising the form template for a `StructBlock` using the `form_template` attribute, the HTML of each child block must be enclosed in an element with a `data-contentpath` attribute equal to the block’s name. This attribute is used by the commenting framework to attach comments to the correct fields. See [Custom editing interfaces for StructBlock](#).
- If a `StructBlock` subclass overrides the `get_form_context` method as part of customising the form template, and that method contains logic that causes the returned context to vary depending on the block value, this will no longer work as intended. This is because `get_form_context` is now invoked once with the block’s default (blank) value in order to construct a template for the client-side rendering to use; previously it was called for each block in the stream. In the new implementation, any Python-side processing that needs to happen on a per-block-value basis can be performed in the block’s `get_form_state` method; the data returned from that method will then be available in the client-side `render` method.
- If `FieldBlock` is used to wrap a Django widget with non-standard client-side behaviour - such as requiring a JavaScript function to be called on initialisation, or combining multiple HTML elements such that it is not possible to read or write its data by accessing a single element’s `value` property - then you will need to supply a JavaScript handler object to define how the widget is rendered and populated, and how to extract data from it.
- Packages that replace the StreamField interface at a low level, such as `wagtail-react-streamfield`, are likely to be incompatible (but the new StreamField implementation will generally offer equivalent functionality).

For further details, see [How to build custom StreamField blocks](#).

## Switched register\_setting, register\_settings\_menu\_item to use SVG icons

Setting menu items now use SVG icons by default. For sites reusing built-in Wagtail icons, no changes should be required. For sites using custom font icons, update the menu items' definition to use the `classnames` attribute:

```
With register_setting,
Before:
@register_setting(icon='custom-cog')
After:
@register_setting(icon='', classnames='icon icon-custom-cog')

Or with register_settings_menu_item,
@hooks.register('register_settings_menu_item')
def register_frank_menu_item():
 # Before:
 return SettingMenuItem(CustomSetting, icon='custom-cog')
 # After:
 return SettingMenuItem(CustomSetting, icon='', classnames='icon icon-custom-cog')
```

## CommentPanel

`Page.settings_panels` now includes `CommentPanel`, which is used to save and load comments. If you are overriding page settings edit handlers without directly extending `Page.settings_panels` (ie `settings_panels = Page.settings_panels + [ FieldPanel('my_field') ]` would need no change here) and want to use the new commenting system, your list of edit handlers should be updated to include `CommentPanel`. For example:

```
from django.db import models

from wagtail.core.models import Page
from wagtail.admin.edit_handlers import CommentPanel

class HomePage(Page):
 settings_panels = [
 # My existing panels here
 CommentPanel(),
]
```

## Feedback

We would love to [receive your feedback](#) on this release.

## 1.9.30 Wagtail 2.12.6 release notes

*July 13, 2021*

- [\*What's new\*](#)

### What's new

#### Bug fixes

- Prevent embed thumbnail\_url migration from failing on URLs longer than 200 characters (Matt Westcott)

## 1.9.31 Wagtail 2.12.5 release notes

*June 17, 2021*

- [\*What's new\*](#)

### What's new

#### CVE-2021-32681: Improper escaping of HTML ('Cross-site Scripting') in Wagtail StreamField blocks

This release addresses a cross-site scripting (XSS) vulnerability in StreamField. When the `{% include_block %}` template tag is used to output the value of a plain-text StreamField block (CharBlock, TextBlock or a similar user-defined block derived from FieldBlock), and that block does not specify a template for rendering, the tag output is not properly escaped as HTML. This could allow users to insert arbitrary HTML or scripting. This vulnerability is only exploitable by users with the ability to author StreamField content (i.e. users with 'editor' access to the Wagtail admin).

Site implementors who wish to retain the existing behaviour of allowing editors to insert HTML content in these blocks (and are willing to accept the risk of untrusted editors inserting arbitrary code) may disable the escaping by surrounding the relevant `{% include_block %}` tag in `{% autoescape off %}...{% endautoescape %}`.

Many thanks to Karen Tracey for reporting this issue. For further details, please see the [CVE-2021-32681](#) security advisory.

## 1.9.32 Wagtail 2.12.4 release notes

April 19, 2021

- *What's new*

### What's new

#### CVE-2021-29434: Improper validation of URLs ('Cross-site Scripting') in rich text fields

This release addresses a cross-site scripting (XSS) vulnerability in rich text fields. When saving the contents of a rich text field in the admin interface, Wagtail did not apply server-side checks to ensure that link URLs use a valid protocol. A malicious user with access to the admin interface could thus craft a POST request to publish content with javascript: URLs containing arbitrary code. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to Kevin Breen for reporting this issue.

### Bug fixes

- Prevent reverse migration errors in images and documents (Mike Brown)
- Avoid wagtailembeds migration failure on MySQL 8.0.13+ (Matt Westcott)

## 1.9.33 Wagtail 2.12.3 release notes

March 5, 2021

- *What's new*

### What's new

### Bug fixes

- Un-pin django-treebeard following upstream fix for migration issue (Matt Westcott)
- Prevent crash when copying an alias page (Karl Hobley)
- Prevent errors on page editing after changing LANGUAGE\_CODE (Matt Westcott)
- Correctly handle model inheritance and ClusterableModel on copy\_for\_translation (Karl Hobley)

## 1.9.34 Wagtail 2.12.2 release notes

February 18, 2021

- *What's new*

### What's new

### Bug fixes

- Pin django-treebeard to <4.5 to prevent migration conflicts (Matt Westcott)

## 1.9.35 Wagtail 2.12.1 release notes

February 16, 2021

- *What's new*

### What's new

### Bug fixes

- Ensure aliases are published when the source page is published (Karl Hobley)
- Make page privacy rules apply to aliases (Karl Hobley)
- Prevent error when saving embeds that do not include a thumbnail URL (Cynthia Kiser)
- Ensure that duplicate embed records are deleted when upgrading (Matt Westcott)
- Prevent failure when running `manage.py dumpdata` with no arguments (Matt Westcott)

## 1.9.36 Wagtail 2.12 release notes

February 2, 2021

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

## What's new

### Image / document choose permission

Images and documents now support a distinct ‘choose’ permission type, to control the set of items displayed in the chooser interfaces when inserting images and documents into a page, and allow setting up collections that are only available for use by specific user groups. This feature was developed by Robert Rollins.

### In-place StreamField updating

StreamField values now formally support being updated in-place from Python code, allowing blocks to be inserted, modified and deleted rather than having to assign a new list of blocks to the field. For further details, see [Modifying StreamField data](#). This feature was developed by Matt Westcott.

### Admin colour themes

Wagtail’s admin now uses CSS custom properties for its primary teal colour. Applying brand colours for the whole user interface only takes a few lines of CSS, and third-party extensions can reuse Wagtail’s CSS variables to support the same degree of customisation. Read on [Custom user interface colours](#). This feature was developed by Joshua Marantz.

### Other features

- Added support for Python 3.9
- Added `WAGTAILIMAGES_IMAGE_FORM_BASE` and `WAGTAILDOCS_DOCUMENT_FORM_BASE` settings to customise the forms for images and documents (Dan Braghis)
- Switch pagination icons to use SVG instead of icon fonts (Scott Cranfill)
- Added string representation to image Format class (Andreas Nüßlein)
- Support `returning None` from `register_page_action_menu_item` and `register_snippet_action_menu_item` to skip registering an item (Vadim Karpenko)
- Fields on a custom image model can now be defined as required / `blank=False` (Matt Westcott)
- Add combined index for Postgres search backend (Will Giddens)
- Add `Page.specified_deferred` property for accessing specific page instance without up-front database queries (Andy Babic)
- Add hash lookup to embeds to support URLs longer than 255 characters (Coen van der Kamp)

### Bug fixes

- Stop menu icon overlapping the breadcrumb on small viewport widths in page editor (Karran Besen)
- Make sure document chooser pagination preserves the selected collection when moving between pages (Alex Sa)
- Gracefully handle oEmbed endpoints returning non-JSON responses (Matt Westcott)
- Fix unique constraint on WorkflowState for SQL Server compatibility (David Beitey)
- Reinstate chevron on collection dropdown (Mike Brown)

- Prevent delete button showing on collection / workflow edit views when delete permission is absent (Helder Correia)
- Move labels above the form field in the image format chooser, to avoid styling issues at tablet size (Helen Chapman)
- `{% include_block with context %}` now passes local variables into the block template (Jonny Scholes)

## Upgrade considerations

### Removed support for Elasticsearch 2

Elasticsearch version 2 is no longer supported as of this release; please upgrade to Elasticsearch 5 or above before upgrading Wagtail.

### **stream\_data on StreamField values is deprecated**

The `stream_data` property of `StreamValue` is commonly used to access the underlying data of a `StreamField`. However, this is discouraged, as it is an undocumented internal attribute and has different data representations depending on whether the value originated from the database or in memory, typically leading to errors on preview if this has not been properly accounted for. As such, `stream_data` is now deprecated.

The recommended alternative is to index the `StreamField` value directly as a list; for example, `page.body[0].block_type` and `page.body[0].value` instead of `page.body.stream_data[0]['type']` and `page.body.stream_data[0]['value']`. This has the advantage that it will return the same Python objects as when the `StreamField` is used in template code (such as `Page` instances for `PageChooserBlock`). However, in most cases, existing code using `stream_data` is written to expect the raw JSON-like representation of the data, and for this the new property `raw_data` (added in Wagtail 2.12) can be used as a drop-in replacement for `stream_data`.

## 1.9.37 Wagtail 2.11.9 release notes

January 24, 2022

- *What's new*

### What's new

### Bug fixes

- Update Pillow dependency to allow 9.x (Rizwan Mansuri)

## 1.9.38 Wagtail 2.11.8 release notes

June 17, 2021

- *What's new*

### What's new

#### CVE-2021-32681: Improper escaping of HTML ('Cross-site Scripting') in Wagtail StreamField blocks

This release addresses a cross-site scripting (XSS) vulnerability in StreamField. When the `{% include_block %}` template tag is used to output the value of a plain-text StreamField block (CharBlock, TextBlock or a similar user-defined block derived from FieldBlock), and that block does not specify a template for rendering, the tag output is not properly escaped as HTML. This could allow users to insert arbitrary HTML or scripting. This vulnerability is only exploitable by users with the ability to author StreamField content (i.e. users with 'editor' access to the Wagtail admin).

Site implementors who wish to retain the existing behaviour of allowing editors to insert HTML content in these blocks (and are willing to accept the risk of untrusted editors inserting arbitrary code) may disable the escaping by surrounding the relevant `{% include_block %}` tag in `{% autoescape off %}...{% endautoescape %}`.

Many thanks to Karen Tracey for reporting this issue. For further details, please see [the CVE-2021-32681 security advisory](#).

## 1.9.39 Wagtail 2.11.7 release notes

April 19, 2021

- *What's new*

### What's new

#### CVE-2021-29434: Improper validation of URLs ('Cross-site Scripting') in rich text fields

This release addresses a cross-site scripting (XSS) vulnerability in rich text fields. When saving the contents of a rich text field in the admin interface, Wagtail did not apply server-side checks to ensure that link URLs use a valid protocol. A malicious user with access to the admin interface could thus craft a POST request to publish content with javascript: URLs containing arbitrary code. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to Kevin Breen for reporting this issue.

## 1.9.40 Wagtail 2.11.6 release notes

March 5, 2021

- *What's new*

### What's new

#### Bug fixes

- Un-pin django-treebeard following upstream fix for migration issue (Matt Westcott)
- Prevent crash when copying an alias page (Karl Hobley)
- Prevent errors on page editing after changing LANGUAGE\_CODE (Matt Westcott)
- Correctly handle model inheritance and ClusterableModel on copy\_for\_translation (Karl Hobley)

## 1.9.41 Wagtail 2.11.5 release notes

February 18, 2021

- *What's new*

### What's new

#### Bug fixes

- Pin django-treebeard to <4.5 to prevent migration conflicts (Matt Westcott)

## 1.9.42 Wagtail 2.11.4 release notes

February 16, 2021

- *What's new*

### What's new

#### Bug fixes

- Prevent delete button showing on collection / workflow edit views when delete permission is absent (Helder Correia)
- Ensure aliases are published when the source page is published (Karl Hobley)
- Make page privacy rules apply to aliases (Karl Hobley)

## 1.9.43 Wagtail 2.11.3 release notes

December 10, 2020

- [What's new](#)
- [Upgrade considerations](#)

### What's new

### Bug fixes

- Updated project template migrations to ensure that initial homepage creation runs before addition of locale field (Dan Braghis)
- Restore ability to use translatable strings in LANGUAGES / WAGTAIL\_CONTENT\_LANGUAGES settings (Andreas Morgenstern)
- Allow `locale` / `translation_of` API filters to be used in combination with search (Matt Westcott)
- Prevent error on `create_log_entries_from_revisions` when checking publish state on a revision that cannot be restored (Kristin Riebe)

### Upgrade considerations

#### `run_before` declaration needed in initial homepage migration

The migration that creates the initial site homepage needs to be updated to ensure that will continue working under Wagtail 2.11. If you have kept the `home` app from the original project layout generated by the `wagtail start` command, this will be `home/migrations/0002_create_homepage.py`. Inside the `Migration` class, add the line `run_before = [('wagtailcore', '0053_locale_model')]` - for example:

```
...

class Migration(migrations.Migration):

 run_before = [
 ('wagtailcore', '0053_locale_model'), # added for Wagtail 2.11 compatibility
]

 dependencies = [
 ('home', '0001_initial'),
]

 operations = [
 migrations.RunPython(create_homepage, remove_homepage),
]
```

This fix applies to any migration that creates page instances programmatically. If you installed Wagtail into an existing Django project by following the instructions at [Integrating Wagtail into a Django project](#), you most likely created the initial homepage manually, and no change is required in this case.

**Further background:** Wagtail 2.11 adds a `locale` field to the `Page` model, and since the existing migrations in your project pre-date this, they are designed to run against a version of the `Page` model that has no `locale` field. As a result, they need to run before the new migrations that have been added to `wagtailcore` within Wagtail 2.11. However, in the old version of the homepage migration, there is nothing to ensure that this sequence is followed. The actual order chosen is an internal implementation detail of Django, and in particular is liable to change as you continue developing your project under Wagtail 2.11 and create new migrations that depend on the current state of `wagtailcore`. In this situation, a user installing your project on a clean database may encounter the following error when running `manage.py migrate`:

```
django.db.utils.IntegrityError: NOT NULL constraint failed: wagtailcore_page.locale_id
```

Adding the `run_before` directive will ensure that the migrations run in the intended order, avoiding this error.

## 1.9.44 Wagtail 2.11.2 release notes

*November 17, 2020*

- [What's new](#)

### What's new

#### Facebook and Instagram embed finders

Two new embed finders have been added for Facebook and Instagram, to replace the previous configuration using Facebook's public oEmbed endpoint which was retired in October 2020. These require a Facebook developer API key - for details of configuring this, see [Facebook and Instagram](#). This feature was developed by Cynthia Kiser and Luis Nell.

#### Bug fixes

- Improve performance of permission check on translations for edit page (Karl Hobley)
- Gracefully handle missing Locale records on `Locale.get_active` and `.localized` (Matt Westcott)
- Handle `get_supported_language_variant` returning a language variant not in `LANGUAGES` (Matt Westcott)
- Reinstate missing icon on settings edit view (Jérôme Lebleu)
- Avoid performance and pagination logic issues with a large number of languages (Karl Hobley)
- Allow deleting the default locale (Matt Westcott)

## 1.9.45 Wagtail 2.11.1 release notes

*November 6, 2020*

- [What's new](#)

## What's new

### Bug fixes

- Ensure that cached `wagtail_site_root_paths` structures from older Wagtail versions are invalidated (Sævar Öfjörð Magnússon)
- Avoid circular import between `wagtail.admin.auth` and custom user models (Matt Westcott)
- Prevent error on resolving page URLs when a locale outside of `WAGTAIL_CONTENT_LANGUAGES` is active (Matt Westcott)

## 1.9.46 Wagtail 2.11 release notes

November 2, 2020

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

Wagtail 2.11 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 12 months).

## What's new

### Multi-lingual content

With this release, Wagtail now has official support for authoring content for multiple languages/regions.

To find out more about this feature, see [Multi-language content](#).

We have also developed a new plugin for translating content between different locales called `wagtail-localize`. You can find details about `wagtail-localize` on its [GitHub page](#).

This feature was sponsored by The Mozilla Foundation and Torchbox.

### Page aliases

This release introduces support for creating page aliases.

Page aliases are exact copies of another page that sit in another part of the tree. They remain in sync with the original page until this link is removed by converting them into a regular page, or deleting the original page.

A page alias can be created through the “Copy Page” UI by selecting the “Alias” checkbox when creating a page copy.

This feature was sponsored by The Mozilla Foundation.

## Collections hierarchy

Collections (for organising images, documents or other media) can now be managed as a hierarchy rather than a flat list. This feature was developed by Robert Rollins.

## Other features

- Add `before_edit_snippet`, `before_create_snippet` and `before_delete_snippet` hooks and documentation (Karl Hobley. Sponsored by the Mozilla Foundation)
- Add `register_snippet_listing_buttons` and `construct_snippet_listing_buttons` hooks and documentation (Karl Hobley. Sponsored by the Mozilla Foundation)
- Add `wagtail --version` to available Wagtail CLI commands (Kalob Taulien)
- Add `hooks.register_temporarily` utility function for testing hooks (Karl Hobley. Sponsored by the Mozilla Foundation)
- Remove `unidecode` and use `anyascii` in for Unicode to ASCII conversion (Robbie Mackay)
- Add `render` helper to `RoutablePageMixin` to support serving template responses according to Wagtail conventions (Andy Babic)
- Specify minimum Python version in `setup.py` (Vince Salvino)
- Extend treebeard's `fix_tree` method with the ability to non-destructively fix path issues and add a `-full` option to apply path fixes (Matt Westcott)
- Add support for hierarchical/nested Collections (Robert Rollins)
- Show user's full name in report views (Matt Westcott)
- Improve Wagtail admin page load performance by caching SVG icons sprite in `localStorage` (Coen van der Kamp)
- Support SVG icons in ModelAdmin menu items (Scott Cranfill)
- Support SVG icons in admin breadcrumbs (Coen van der Kamp)
- Serve PDFs inline in the browser (Matt Westcott)
- Make `document content-type` and `content-disposition` configurable via `WAGTAILDOCS_CONTENT_TYPES` and `WAGTAILDOCS_INLINE_CONTENT_TYPES` (Matt Westcott)
- Slug generation no longer removes stopwords (Andy Chosak, Scott Cranfill)
- Add check to disallow StreamField block names that do not match Python variable syntax (François Poulain)
- The `BASE_URL` setting is now converted to a string, if it isn't already, when constructing API URLs (thenewguy)
- Preview from 'pages awaiting moderation' now opens in a new window (Cynthia Kiser)
- Add document extension validation if `WAGTAIL_DOCS_EXTENSIONS` is set to a list of allowed extensions (Meghana Bhange)
- Use `django-admin` command in place of `django-admin.py` (minusf)
- Add `register_snippet_action_menu_item` and `construct_snippet_action_menu` hooks to modify the actions available when creating / editing a snippet (Karl Hobley)
- Moved `generate_signature` and `verify_signature` functions into `wagtail.images.utils` (Noah H)
- Implement `bulk_to_python` on all structural StreamField block types (Matt Westcott)
- Add natural key support to `GroupCollectionPermission` (Jim Jazwiecki)

- Implement `prepopulated_fields` for `wagtail.contrib.modeladmin` (David Bramwell)
- Change `classname` keyword argument on basic StreamField blocks to `form_classname` (Meghana Bhange)
- Replace page explorer pushPage/popPage with gotoPage for more flexible explorer navigation (Karl Hobley)

## Bug fixes

- Make page-level actions accessible to keyboard users in page listing tables (Jesse Menn)
- `WAGTAILFRONTENDCACHE_LANGUAGES` was being interpreted incorrectly. It now accepts a list of strings, as documented (Karl Hobley)
- Update oEmbed endpoints to use https where available (Matt Westcott)
- Revise `edit_handler` bind order in ModelAdmin views and fix duplicate form instance creation (Jérôme Lebleu)
- Properly distinguish child blocks when comparing revisions with nested StreamBlocks (Martin Mena)
- Correctly handle Turkish ‘İ’ characters in client-side slug generation (Matt Westcott)
- Page chooser widgets now reflect custom `get_admin_display_title` methods (Saptak Sengupta)
- `Page.copy()` now raises an error if the page being copied is unsaved (Anton Zhyltsov)
- `Page.copy()` now triggers a `page_published` if the copied page is live (Anton Zhyltsov)
- The Elasticsearch URLs setting can now take a string on its own instead of a list (Sævar Öfjörð Magnússon)
- Avoid retranslating month / weekday names that Django already provides (Matt Westcott)
- Fixed padding around checkbox and radio inputs (Cole Maclean)
- Fix spacing around the privacy indicator panel (Sævar Öfjörð Magnússon, Dan Braghis)
- Consistently redirect to admin home on permission denied (Matt Westcott, Anton Zhyltsov)

## Upgrade considerations

### `run_before` declaration needed in initial homepage migration

The migration that creates the initial site homepage needs to be updated to ensure that will continue working under Wagtail 2.11. If you have kept the `home` app from the original project layout generated by the `wagtail start` command, this will be `home/migrations/0002_create_homepage.py`. Inside the `Migration` class, add the line `run_before = [('wagtailcore', '0053_locale_model')]` - for example:

```
...

class Migration(migrations.Migration):

 run_before = [
 ('wagtailcore', '0053_locale_model'), # added for Wagtail 2.11 compatibility
]

 dependencies = [
 ('home', '0001_initial'),
]
```

(continues on next page)

(continued from previous page)

```
operations = [
 migrations.RunPython(create_homepage, remove_homepage),
]
```

This fix applies to any migration that creates page instances programmatically. If you installed Wagtail into an existing Django project by following the instructions at [Integrating Wagtail into a Django project](#), you most likely created the initial homepage manually, and no change is required in this case.

**Further background:** Wagtail 2.11 adds a `locale` field to the `Page` model, and since the existing migrations in your project pre-date this, they are designed to run against a version of the `Page` model that has no `locale` field. As a result, they need to run before the new migrations that have been added to `wagtailcore` within Wagtail 2.11. However, in the old version of the homepage migration, there is nothing to ensure that this sequence is followed. The actual order chosen is an internal implementation detail of Django, and in particular is liable to change as you continue developing your project under Wagtail 2.11 and create new migrations that depend on the current state of `wagtailcore`. In this situation, a user installing your project on a clean database may encounter the following error when running `manage.py migrate`:

```
django.db.utils.IntegrityError: NOT NULL constraint failed: wagtailcore_page.locale_id
```

Adding the `run_before` directive will ensure that the migrations run in the intended order, avoiding this error.

## IE11 support being phased out

This release begins the process of phasing out support for Internet Explorer. Please see [IE11](#) for details of the timeline over which support will be dropped.

## SiteMiddleware moved to wagtail.contrib.legacy

The `SiteMiddleware` class (which provides the `request.site` property, and has been deprecated since Wagtail 2.9) has been moved to the `wagtail.contrib.legacy` namespace. On projects where this is still in use, the '`wagtail.core.middleware.SiteMiddleware`' entry in `MIDDLEWARE` should be changed to '`wagtail.contrib.legacy.sitemiddleware.SiteMiddleware`'.

## Collection model enforces alphabetical ordering

As part of the hierarchical collections support, the `path` field on the `Collection` model now enforces alphabetical ordering. Previously, collections were stored in the order in which they were created - and then sorted by name where displayed in the CMS. This change will be handled automatically through migrations when upgrading to Wagtail 2.11.

However, if your project creates new collections programmatically after migrations have run, and assigns the `path` field directly - for example, by loading from a fixture file - this code will need to be updated to insert them in alphabetical order. Otherwise, errors may occur when subsequently adding new collections through the Wagtail admin. This can be done as follows:

- Update paths to match alphabetical order. For example, if you have a fixture that creates the collections Zebras and Aardvarks with paths `00010001` and `00010002` respectively, these paths should be swapped.
- Alternatively, after creating the collections, run the Python code:

```
from wagtail.core.models import Collection
Collection.fix_tree(fix_paths=True)
```

or the management command:

```
python manage.py fixtree --full
```

### Site.get\_site\_root\_paths now returns language code

In previous releases, `Site.get_site_root_paths` returned a list of `(site_id, root_path, root_url)` tuples. To support the new internationalisation model, this has now been changed to a list of named tuples with the fields: `site_id`, `root_path`, `root_url` and `language_code`. Existing code that handled this as a 3-tuple should be updated accordingly.

### classname argument on StreamField blocks is now form\_classname

Basic StreamField block types such as `CharBlock` previously accepted a `classname` keyword argument, to specify a `class` attribute to appear on the page editing form. For consistency with `StructBlock`, this has now been changed to `form_classname`. The `classname` argument is still recognised, but deprecated.

## 1.9.47 Wagtail 2.10.2 release notes

September 25, 2020

- *What's new*

### What's new

#### Bug fixes

- Avoid use of `icon` class name on userbar icon to prevent clashes with front-end styles (Karran Besen)
- Prevent focused button labels from displaying as white on white (Karran Bessen)
- Avoid showing preview button on moderation dashboard for page types with preview disabled (Dino Perovic)
- Prevent oversized buttons in moderation dashboard panel (Dan Braghis)
- `create_log_entries_from_revisions` now handles revisions that cannot be restored due to foreign key constraints (Matt Westcott)

## 1.9.48 Wagtail 2.10.1 release notes

August 26, 2020

- *What's new*

## What's new

### Bug fixes

- Prevent `create_log_entries_from_revisions` command from failing when page model classes are missing (Dan Braghis)
- Prevent page audit log views from failing for user models without a `username` field (Vyacheslav Matyukhin)
- Fix icon alignment on menu items (Coen van der Kamp)
- Page editor header bar now correctly shows ‘Published’ or ‘Draft’ status when no revisions exist (Matt Westcott)
- Prevent page editor from failing when `USE_TZ` is false (Matt Westcott)
- Ensure whitespace between block-level elements is preserved when stripping tags from rich text for search indexing (Matt Westcott)

## 1.9.49 Wagtail 2.10 release notes

August 11, 2020

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

## What's new

### Moderation workflow

This release introduces a configurable moderation workflow system to replace the single-step “submit for moderation” feature. Workflows can be set up on specific subsections of the page tree and consist of any number of tasks to be completed by designated user groups. To support this, numerous UI improvements have been made to Wagtail’s page editor, including a new log viewer to track page history.

For further details, see [Managing Workflows](#) and [Adding new Task types](#).

This feature was developed by Jacob Topp-Mugglestone, Karl Hobley, Matt Westcott and Dan Braghis, and sponsored by [The Motley Fool](#).

### Django 3.1 support

This release adds support for Django 3.1. Compatibility fixes were contributed by Matt Westcott and Karl Hobley.

## Search query expressions

Search queries can now be constructed as structured expressions in the manner of the Django ORM’s `Q()` values, allowing for complex queries that combine individual terms, phrases and boosting. A helper function `parse_query_string` is provided to convert “natural” queries containing quoted phrases into these expressions. For complete documentation, see [Complex search queries](#). This feature was developed by Karl Hobley and sponsored by [The Motley Fool](#).

## Redirect importing

Redirects can now be imported from an uploaded CSV, TSV, XLS or XLSX file. This feature was developed by Martin Sandström.

## Accessibility and usability

This release contains a number of improvements to the accessibility and general usability of the Wagtail admin, fixing long-standing issues. Some of the changes come from our January 2020 sprint in Bristol, and some from our brand new [accessibility team](#):

- Remove sticky footer on small devices, so that content is not blocked and more easily editable (Saeed Tahmasebi)
- Add SVG icons to resolve accessibility and customisation issues and start using them in a subset of Wagtail’s admin (Coen van der Kamp, Scott Cranfill, Thibaud Colas, Dan Braghis)
- Switch userbar and header H1s to use SVG icons (Coen van der Kamp)
- Add skip link for keyboard users to bypass Wagtail navigation in the admin (Martin Coote)
- Add missing dropdown icons to image upload, document upload, and site settings screens (Andreas Bernacca)
- Prevent snippets’ bulk delete button from being present for screen reader users when it’s absent for sighted users (LB (Ben Johnston))

## Other features

- Added `webpquality` and `format-webp-lossless` image filters and `WAGTAILIMAGES_WEBP_QUALITY` setting. See [Output image format](#) and [Image quality](#) (Nikolay Lukyanov)
- Reorganised Dockerfile in project template to follow best practices (Tomasz Knapik, Jannik Wempe)
- Added filtering to locked pages report (Karl Hobley)
- Adds ability to view a group’s users via standalone admin URL and a link to this on the group edit view (Karran Besen)
- Redirect to previous url when deleting/copying/unpublish a page and modify this url via the relevant hooks (Ascani Carlo)
- Added `next_url` keyword argument on `register_page_listing_buttons` and `register_page_listing_more_buttons` hooks (Ascani Carlo, Matt Westcott, LB (Ben Johnston))
- `AbstractEmailForm` will use `SHORT_DATETIME_FORMAT` and `SHORT_DATE_FORMAT` Django settings to format date/time values in email (Haydn Greatnews)
- `AbstractEmailForm` now has a separate method (`render_email`) to build up email content on submission emails. See [Custom render\\_email method](#). (Haydn Greatnews)
- Add `pre_page_move` and `post_page_move` signals. (Andy Babic)

- Add ability to sort search promotions on listing page (Chris Ranjana, LB (Ben Johnston))
- Upgrade internal JS tooling; Node v10, Gulp v4 & Jest v23 (Jim Jazwiecki, Kim LaRocca, Thibaud Colas)
- Add `after_publish_page`, `before_publish_page`, `after_unpublish_page` & `before_unpublish_page` hooks (Jonatas Baldin, Coen van der Kamp)
- Add convenience `page_url` shortcut to improve how page URLs can be accessed from site settings in Django templates (Andy Babic)
- Show more granular error messages from Pillow when uploading images (Rick van Hattem)
- Add ordering to `Site` object, so that index page and Site switcher will be sorted consistently (Coen van der Kamp, Tim Leguijt)
- Add Reddit to oEmbed provider list (Luke Hardwick)
- Add ability to replace the default Wagtail logo in the userbar, via `branding_logo` block (Meteor0id)
- Add `alt` property to `ImageRenditionField` api representation (Liam Mullens)
- Add `purge_revisions` management command to purge old page revisions (Jacob Topp-Muggleton, Tom Dyson)
- Render the Wagtail User Bar on non Page views (Caitlin White, Coen van der Kamp)
- Add ability to define `form_classname` on `ListBlock` & `StreamBlock` (LB (Ben Johnston))
- Add documentation about how to use Rustface for image feature detection (Neal Todd)
- Improve performance of public/not\_public queries in `PageQuerySet` (Timothy Bautista)
- Add `add_redirect` static method to `Redirect` class for programmatic redirect creation (Brylie Christopher Oxley, Lacey Williams Henschel)
- Add reference documentation for `wagtail.contrib.redirects`. See [Redirects](#). (LB (Ben Johnston))
- `bulk_delete` page permission is no longer required to move pages, even if those pages have children (Robert Rollins, LB (Ben Johnston))
- Add `after_edit_snippet`, `after_create_snippet` and `after_delete_snippet` hooks and documentation (Kalob Taulien)
- Improve performance of empty search results by avoiding downloading the entire search index in these scenarios (Lars van de Kerkhof, Coen van der Kamp)
- Replace `gulp-sass` with `gulp-dart-sass` to improve core development across different platforms (Thibaud Colas)
- Remove markup around rich text rendering by default, provide a way to use old behaviour via `wagtail.contrib.legacy.richtext`. See [Legacy richtext](#). (Coen van der Kamp, Dan Braghis)
- Add `WAGTAIL_TIME_FORMAT` setting (Jacob Topp-Muggleton)
- Apply title length normalisation to improve ranking on PostgreSQL search (Karl Hobley)
- Allow omitting the default editor from `WAGTAILADMIN_RICH_TEXT_EDITORS` (Gassan Gousseinov)
- Disable password auto-completion on user creation form (Samir Shah)
- Upgrade jQuery to version 3.5.1 to reduce penetration testing false positives (Matt Westcott)
- Add ability to extend `EditHandler` without a `children` attribute (Seb Brown)
- `Page.objects_specific` now gracefully handles pages with missing specific records (Andy Babic)
- StreamField ‘add’ buttons are now disabled when maximum count is reached (Max Gabrielsson)

- Use underscores for form builder field names to allow use as template variables (Ashia Zawaduk, LB (Ben Johnston))
- Deprecate use of unidecode within form builder field names (Michael van Tellingen, LB (Ben Johnston))
- Improve error feedback when editing a page with a missing model class (Andy Babic)
- Change Wagtail tabs implementation to only allow slug-formatted tab identifiers, reducing false positives from security audits (Matt Westcott)
- Ensure errors during Postgres search indexing are left uncaught to assist troubleshooting (Karl Hobley)
- Add ability to edit images and embeds in rich text editor (Maylon Pedroso, Samuel Mendes, Gabriel Peracio)

## Bug fixes

- Ensure link to add a new user works when no users are visible in the users list (LB (Ben Johnston))
- `AbstractEmailForm` saved submission fields are now aligned with the email content fields, `form.cleaned_data` will be used instead of `form.fields` (Haydn Greatnews)
- Removed ARIA `role="table"` from TableBlock output (Thibaud Colas)
- Set Cache-Control header to prevent page preview responses from being cached (Tomas Walch)
- Accept unicode characters in slugs on the “copy page” form (François Poulain)
- Support IPv6 domain (Alex Gleason, Coen van der Kamp)
- Remove top padding when `FieldRowPanel` is used inside a `MultiFieldPanel` (Jérôme Lebleu)
- Add Wagtail User Bar back to page previews and ensure moderation actions are available (Coen van der Kamp)
- Fix issue where queryset annotations were lost (e.g. `.annotate_score()`) when using specific models in page query (Dan Bentley)
- Prevent date/time picker from losing an hour on losing focus when 12-hour times are in use (Jacob Topp-Muggleson)
- Strip out HTML tags from `RichTextField` & `RichTextBlock` search index content (Timothy Bautista)
- Avoid using null on string `Site.site_name` blank values to avoid different values for no name (Coen van der Kamp)
- Fix deprecation warnings on Elasticsearch 7 (Yngve Høiseth)
- Remove use of `Node.forEach` for IE 11 compatibility in admin menu items (Thibaud Colas)
- Fix incorrect method name in `SiteMiddleware` deprecation warning (LB (Ben Johnston))
- `wagtail.contrib.sitemaps` no longer depends on `SiteMiddleware` (Matt Westcott)
- Purge image renditions cache when renditions are deleted (Pascal Widdershoven, Matt Westcott)
- Image / document forms now display non-field errors such as `unique_together` constraints (Matt Westcott)
- Make “Site” chooser in site settings translatable (Andreas Bernacca)
- Fix group permission checkboxes not being clickable in IE11 (LB (Ben Johnston))

## Upgrade considerations

### Removed support for Python 3.5

Python 3.5 is no longer supported as of this release; please upgrade to Python 3.6 or above before upgrading Wagtail.

### Move to new configurable moderation system (workflow)

A new workflow system has been introduced for moderation. Task types are defined as models in code, and instances - tasks - are created in the Wagtail Admin, then chained together to form workflows: sequences of moderation stages through which a page must pass prior to publication.

Key points:

- Prior to 2.10, moderation in Wagtail was performed on a per-revision basis: once submitted, the moderator would approve or reject the submitted revision only, which would not include subsequent changes. Moderation is now performed per page, with moderators always seeing the latest revision.
- `PageRevision.submitted_for_moderation` will return `True` for revisions passing through the old moderation system, but not for the new system
- Pages undergoing moderation in the old system will not have their moderation halted, and can still be approved/rejected. As a result, you may see two sets of moderation dashboard panels until there are no longer any pages in moderation in the old system
- No pages can be submitted for moderation in the old system: “Submit for moderation” now submits to the new Workflow system
- You no longer need the publish permission to perform moderation actions on a page - actions available to each user are now configured per task. With the built in `GroupApprovalTask`, anybody in a specific set of groups can approve or reject the task.
- A data migration is provided to recreate your existing publish-permission based moderation workflow in the new system. If you have made no permissions changes, this should simply create a task approvable by anybody in the *Moderators* group, and assign a workflow with this task to the root page, creating a standard workflow for the entire page tree. However, if you have a complex nested set of publish page permissions, the created set of workflows will be more complex as well - you may wish to inspect the created workflows and tasks in the new `Settings/Workflows` admin area and potentially simplify them. See [Managing Workflows](#) for the administrator guide.

### <div class="rich-text"> wrappers removed from rich text

In previous releases, rich text values were enclosed in a `<div class="rich-text">` element when rendered; this element has now been removed. To restore the old behaviour, see [Legacy richtext](#).

## Prepopulating data for site history report

This release introduces logging of user actions, viewable through the “Site history” report. To pre-populate these logs with data from page revision history, run the management command: `./manage.py create_log_entries_from_revisions`.

## `clean_name` field added to form builder form field models

A `clean_name` field has been added to form field models that extend `AbstractForm`. This is used as the name attribute of the HTML form field, and the dictionary key that the submitted form data is stored under. Storing this on the model (rather than calculating it on-the-fly as was done previously) ensures that if the algorithm for generating the clean name changes in future, the existing data will not become inaccessible. A future version of Wagtail will drop the `unidecode` library currently used for this.

For forms created through the Wagtail admin interface, no action is required, as the new field will be populated on server startup. However, any process that creates form pages through direct insertion on the database (such as loading from fixtures) should now be updated to populate `clean_name`.

### New `next_url` keyword argument on `register_page_listing_buttons` and `register_page_listing_more_buttons` hooks

Functions registered through the hooks `register_page_listing_buttons` and `register_page_listing_more_buttons` now accept an additional keyword argument `next_url`. A hook function currently written as:

```
@hooks.register('register_page_listing_buttons')
def page_listing_more_buttons(page, page_perms, is_parent=False):
 yield wagtailadmin_widgets.Button(
 'My button', '/goes/to/a/url/', priority=60
)
```

should now become:

```
@hooks.register('register_page_listing_buttons')
def page_listing_more_buttons(page, page_perms, is_parent=False, next_url=None):
 yield wagtailadmin_widgets.Button(
 'My button', '/goes/to/a/url/', priority=60
)
```

The `next_url` argument specifies a URL to redirect back to after the action is complete, and can be passed as a query parameter to the linked URL, if the view supports it.

## 1.9.50 Wagtail 2.9.3 release notes

*July 20, 2020*

## CVE-2020-15118: HTML injection through form field help text

This release addresses an HTML injection vulnerability through help text in the `wagtail.contrib.forms` form builder app. When a form page type is made available to Wagtail editors, and the page template is built using Django's standard form rendering helpers such as `form.as_p` ([as directed in the documentation](#)), any HTML tags used within a form field's help text will be rendered unescaped in the page. Allowing HTML within help text is [an intentional design decision by Django](#); however, as a matter of policy Wagtail does not allow editors to insert arbitrary HTML by default, as this could potentially be used to carry out cross-site scripting attacks, including privilege escalation. This functionality should therefore not have been made available to editor-level users.

The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Site owners who wish to re-enable the use of HTML within help text (and are willing to accept the risk of this being exploited by editors) may set `WAGTAILFORMS_HELP_TEXT_ALLOW_HTML = True` in their configuration settings.

Many thanks to Timothy Bautista for reporting this issue.

## 1.9.51 Wagtail 2.9.2 release notes

*July 3, 2020*

- [What's new](#)

### What's new

### Bug fixes

- Prevent startup failure when `wagtail.contrib.sitemaps` is in `INSTALLED_APPS` (Matt Westcott)

## 1.9.52 Wagtail 2.9.1 release notes

*June 30, 2020*

- [What's new](#)

### What's new

### Bug fixes

- Fix incorrect method name in SiteMiddleware deprecation warning (LB (Ben Johnston))
- `wagtail.contrib.sitemaps` no longer depends on SiteMiddleware (Matt Westcott)
- Purge image renditions cache when renditions are deleted (Pascal Widdershoven, Matt Westcott)

## 1.9.53 Wagtail 2.9 release notes

May 4, 2020

- [What's new](#)
- [Upgrade considerations](#)

### What's new

#### Report data exports

Data from reports, form submissions and ModelAdmin can now be exported to both XLSX and CSV format. For ModelAdmin, this is enabled by specifying a `list_export` attribute on the ModelAdmin class. This feature was developed by Jacob Topp-Mugglestone and sponsored by [The Motley Fool](#).

#### CVE-2020-11037: Potential timing attack on password-protected private pages

This release addresses a potential timing attack on pages or documents that have been protected with a shared password through Wagtail's "Privacy" controls. This password check is performed through a character-by-character string comparison, and so an attacker who is able to measure the time taken by this check to a high degree of accuracy could potentially use timing differences to gain knowledge of the password. (This is understood to be feasible on a local network, but not on the public internet.)

Many thanks to Thibaud Colas for reporting this issue.

### Other features

- Added support for creating custom reports (Jacob Topp-Mufflestone)
- Skip page validation when unpublishing a page (Samir Shah)
- Added `MultipleChoiceBlock` block type for StreamField (James O'Toole)
- ChoiceBlock now accepts a `widget` keyword argument (James O'Toole)
- Reduced contrast of rich text toolbar (Jack Paine)
- Support the `rel` attribute on custom ModelAdmin buttons (Andy Chosak)
- Server-side page slug generation now respects `WAGTAIL_ALLOW_UNICODE_SLUGS` (Arkadiusz Michał Ryś)
- Wagtail admin no longer depends on SiteMiddleware, avoiding incompatibility with Django sites framework and redundant database queries (aritas1, timmymalls, Matt Westcott)
- Tag field autocompletion now handles custom tag models (Matt Westcott)
- `wagtail_serve` URL route can now be omitted for headless sites (Storm Heg)
- Allow free tagging to be disabled on custom tag models (Matt Westcott)
- Allow disabling page preview by setting `preview_modes` to an empty list (Casper Timmers)
- Add Vidyard to oEmbed provider list (Steve Lyall)
- Optimise compiling media definitions for complex StreamBlocks (pimarc)

- FieldPanel now accepts a ‘heading’ argument (Jacob Topp-Mufflestone)
- Replaced deprecated `ugettext` / `ungettext` calls with `gettext` / `ngettext` (Mohamed Feddad)
- ListBlocks now call child block `bulk_to_python` if defined (Andy Chosak)
- Site settings are now identifiable/cacheable by request as well as site (Andy Babic)
- Added `select_related` attribute to site settings to enable more efficient fetching of foreign key values (Andy Babic)
- Add caching of image renditions (Tom Dyson, Tim Kamanin)
- Add documentation for reporting security issues and internationalisation (Matt Westcott)
- Fields on a custom image model can now be defined as required `blank=False` (Matt Westcott)

### Bug fixes

- Added ARIA alert role to live search forms in the admin (Casper Timmers)
- Reordered login form elements to match expected tab order (Kjartan Sverrisson)
- Re-added ‘Close Explorer’ button on mobile viewports (Sævar Öfjörð Magnússon)
- Added a more descriptive label to Password reset link for screen reader users (Casper Timmers, Martin Coote)
- Improved Wagtail logo contrast by adding a background (Brian Edelman, Simon Evans, Ben Enright)
- Prevent duplicate notification messages on page locking (Jacob Topp-Mufflestone)
- Rendering of non field errors for InlinePanel items (Storm Heg)
- `{% image ... as var %}` now clears the context variable when passed None as an image (Maylon Pedroso)
- `refresh_index` method on Elasticsearch no longer fails (Lars van de Kerkhof)
- Document tags no longer fail to update when replacing the document file at the same time (Matt Westcott)
- Prevent error from very tall / wide images being resized to 0 pixels (Fidel Ramos)
- Remove excess margin when editing snippets (Quadric)
- Added `scope` attribute to table headers in TableBlock output (Quadric)
- Prevent KeyError when accessing a StreamField on a deferred queryset (Paulo Alvarado)
- Hide empty ‘view live’ links (Karran Besen)
- Mark up a few strings for translation (Luiz Boareto)
- Invalid `focal_point` attribute on image edit view (Michał (Quadric) Sieradzki)
- No longer expose the `.delete()` method on the default Page.objects manager (Nick Smith)
- `exclude_fields_in_copy` on Page models will now work for for modelcluster parental / many to many relations (LB (Ben Johnston))
- Response header (content disposition) now correctly handles filenames with non-ascii characters when using a storage backend (Rich Brennan)
- Improved accessibility fixes for `main`, `header` and `footer` elements in the admin page layout (Mitchel Cabuloy)
- Prevent version number from obscuring long settings menus (Naomi Morduch Toubman)
- Admin views using TemplateResponse now respect the user’s language setting (Jacob Topp-Mufflestone)
- Fixed incorrect language code for Japanese in language setting dropdown (Tomonori Tanabe)

## Upgrade considerations

### Removed support for Django 2.1

Django 2.1 is no longer supported as of this release; please upgrade to Django 2.2 or above before upgrading Wagtail.

### SiteMiddleware and request.site deprecated

Wagtail's `SiteMiddleware`, which makes the current site object available as the property `request.site`, is now deprecated as it clashes with Django's sites framework and makes unnecessary database queries on non-Wagtail views. References to `request.site` in your code should be removed; the recommended way of retrieving the current site is `Site.find_for_request(request)` in Python code, and the `{% wagtail_site %}` tag within Django templates.

For example:

```
old version

def get_menu_items(request):
 return request.site.root_page.get_children().live()

new version

from wagtail.core.models import Site

def get_menu_items(request):
 return Site.find_for_request(request).root_page.get_children().live()
```

```
{# old version #-}

<h1>Welcome to the {{ request.site.site_name }} website!</h1>

{# new version #-}
{% load wagtailcore_tags %}
{% wagtail_site as current_site %}

<h1>Welcome to the {{ current_site.site_name }} website!</h1>
```

Once these are removed, '`wagtail.core.middleware.SiteMiddleware`' can be removed from your project's `MIDDLEWARE` setting.

### Page / Collection managers no longer expose a delete method

For consistency with standard Django models, the `delete()` method is no longer available on the default Page and Collection managers. Code such as `Page.objects.delete()` should be changed to `Page.objects.all().delete()`.

## 1.9.54 Wagtail 2.8.2 release notes

May 4, 2020

### CVE-2020-11037: Potential timing attack on password-protected private pages

This release addresses a potential timing attack on pages or documents that have been protected with a shared password through Wagtail's "Privacy" controls. This password check is performed through a character-by-character string comparison, and so an attacker who is able to measure the time taken by this check to a high degree of accuracy could potentially use timing differences to gain knowledge of the password. (This is understood to be feasible on a local network, but not on the public internet.)

Many thanks to Thibaud Colas for reporting this issue.

## 1.9.55 Wagtail 2.8.1 release notes

April 14, 2020

### CVE-2020-11001: Possible XSS attack via page revision comparison view

This release addresses a cross-site scripting (XSS) vulnerability on the page revision comparison view within the Wagtail admin interface. A user with a limited-permission editor account for the Wagtail admin could potentially craft a page revision history that, when viewed by a user with higher privileges, could perform actions with that user's credentials. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to Vlad Gerasimenko for reporting this issue.

## 1.9.56 Wagtail 2.8 release notes

February 3, 2020

- [What's new](#)
- [Upgrade considerations](#)

### What's new

#### Django 3.0 support

This release is compatible with Django 3.0. Compatibility fixes were contributed by Matt Westcott and Mads Jensen.

## Improved page locking

The page locking feature has been revised so that the editor locking a page is given exclusive edit access to it, rather than it becoming read-only to everyone. A new Reports menu allows admin / moderator level users to see the currently locked pages, and unlock them if required.

This feature was developed by Karl Hobley and Jacob Topp-Mugglestone. Thanks to [The Motley Fool](#) for sponsoring this feature.

## Other features

- Removed leftover Python 2.x compatibility code (Sergey Fedoseev)
- Combine flake8 configurations (Sergey Fedoseev)
- Improve diffing behaviour for text fields (Aliosha Padovani)
- Improve contrast of disabled inputs (Nick Smith)
- Added `get_document_model_string` function (Andrey Smirnov)
- Added support for Cloudflare API tokens for frontend cache invalidation (Tom Usher)
- Cloudflare frontend cache invalidation requests are now sent in chunks of 30 to fit within API limits (Tom Usher)
- Added `ancestors` field to the pages endpoint in admin API (Karl Hobley)
- Removed Django admin management of Page & Site models (Andreas Bernacca)
- Cleaned up Django docs URLs in documentation (Pete Andrew)
- Add StreamFieldPanel to available panel types in documentation (Dan Swain)
- Add `{{ block.super }}` example to ModelAdmin customisation in documentation (Dan Swain)
- Add ability to filter image index by a tag (Benedikt Willi)
- Add partial experimental support for nested InlinePanels (Matt Westcott, Sam Costigan, Andy Chosak, Scott Cranfill)
- Added cache control headers when serving documents (Johannes Vogel)
- Use `sensitive_post_parameters` on password reset form (Dan Braghis)
- Add `WAGTAILEMBEDS_RESPONSIVE_HTML` setting to remove automatic addition of `responsive-object` around embeds (Kalob Taulien)

## Bug fixes

- Rename documents listing column ‘uploaded’ to ‘created’ (LB (Ben Johnston))
- Unbundle the i18n library as it was bundled to avoid installation errors which have been resolved (Matt Westcott)
- Prevent error when comparing pages that reference a model with a custom primary key (Fidel Ramos)
- Moved `get_document_model` location so it can be imported when Models are not yet loaded (Andrey Smirnov)
- Use correct HTML escaping of Jinja2 form templates for StructBlocks (Brady Moe)
- All templates with wagtailsettings and modeladmin now use `block.super` for `extra_js` & `extra_css` (Timothy Bautista)
- Layout issue when using `FieldRowPanel` with a heading (Andreas Bernacca)

- `file_size` and `file_hash` now updated when Document file changed (Andreas Bernacca)
- Fixed order of URLs in project template so that static / media URLs are not blocked (Nick Smith)
- Added `verbose_name_plural` to form submission model (Janneke Janssen)
- Prevent `update_index` failures and incorrect front-end rendering on blank `TableBlock` (Carlo Ascani)
- Dropdown initialisation on the search page after AJAX call (Eric Sherman)
- Make sure all modal chooser search results correspond to the latest search by cancelling previous requests (Esper Kuijs)

### Upgrade considerations

#### Removed support for Django 2.0

Django 2.0 is no longer supported as of this release; please upgrade to Django 2.1 or above before upgrading Wagtail.

#### Edit locking behaviour changed

The behaviour of the page locking feature in the admin interface has been changed. In past versions, the page lock would apply to all users including the user who locked the page. Now, the user who locked the page can still edit it but all other users cannot.

Pages that were locked before this release will continue to be locked in the same way as before, so this only applies to newly locked pages. If you would like to restore the previous behaviour, you can set the `WAGTAILADMIN_GLOBAL_PAGE_EDIT_LOCK` setting to `True`.

#### Responsive HTML for embeds no longer added by default

In previous versions of Wagtail, embedded media elements were given a class name of `responsive-object` and an `inline padding-bottom` style to assist in styling them responsively. These are no longer added by default. To restore the previous behaviour, add `WAGTAILEMBEDS_RESPONSIVE_HTML = True` to your project settings.

#### API endpoint classes have moved

For consistency with Django REST Framework, the `PagesAPIEndpoint`, `ImagesAPIEndpoint` and `DocumentsAPIEndpoint` classes have been renamed to `PagesAPIViewSet`, `ImagesAPIViewSet` and `DocumentsAPIViewSet` and moved to the `views` module in their respective packages. Projects using the Wagtail API should update their registration code accordingly.

Old code:

```
from wagtail.api.v2.endpoints import PagesAPIEndpoint
from wagtail.api.v2.router import WagtailAPIRouter
from wagtail.images.api.v2.endpoints import ImagesAPIEndpoint
from wagtail.documents.api.v2.endpoints import DocumentsAPIEndpoint

api_router = WagtailAPIRouter('wagtailapi')
api_router.register_endpoint('pages', PagesAPIEndpoint)
api_router.register_endpoint('images', ImagesAPIEndpoint)
api_router.register_endpoint('documents', DocumentsAPIEndpoint)
```

New code:

```
from wagtail.api.v2.views import PagesAPIViewSet
from wagtail.api.v2.router import WagtailAPIRouter
from wagtail.images.api.v2.views import ImagesAPIViewSet
from wagtail.documents.api.v2.views import DocumentsAPIViewSet

api_router = WagtailAPIRouter('wagtailapi')
api_router.register_endpoint('pages', PagesAPIViewSet)
api_router.register_endpoint('images', ImagesAPIViewSet)
api_router.register_endpoint('documents', DocumentsAPIViewSet)
```

### wagtail.documents.models.get\_document\_model has moved

The `get_document_model` function should now be imported from `wagtail.documents` rather than `wagtail.documents.models`. See [Custom document model](#).

### Removed Page and Site models from Django admin

The Page and Site models are no longer editable through the Django admin backend. If required these models can be re-registered within your own project using Django's `ModelAdmin`:

```
my_app/admin.py
from django.contrib import admin

from wagtail.core.models import Page, Site

admin.site.register(Site)
admin.site.register(Page)
```

## 1.9.57 Wagtail 2.7.4 release notes

*July 20, 2020*

### CVE-2020-15118: HTML injection through form field help text

This release addresses an HTML injection vulnerability through help text in the `wagtail.contrib.forms` form builder app. When a form page type is made available to Wagtail editors, and the page template is built using Django's standard form rendering helpers such as `form.as_p` ([as directed in the documentation](#)), any HTML tags used within a form field's help text will be rendered unescaped in the page. Allowing HTML within help text is [an intentional design decision by Django](#); however, as a matter of policy Wagtail does not allow editors to insert arbitrary HTML by default, as this could potentially be used to carry out cross-site scripting attacks, including privilege escalation. This functionality should therefore not have been made available to editor-level users.

The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Site owners who wish to re-enable the use of HTML within help text (and are willing to accept the risk of this being exploited by editors) may set `WAGTAILFORMS_HELP_TEXT_ALLOW_HTML = True` in their configuration settings.

Many thanks to Timothy Bautista for reporting this issue.

## Additional fixes

- Expand Pillow dependency range to include 7.x (Harris Lapiroff, Matt Westcott)

## 1.9.58 Wagtail 2.7.3 release notes

May 4, 2020

### CVE-2020-11037: Potential timing attack on password-protected private pages

This release addresses a potential timing attack on pages or documents that have been protected with a shared password through Wagtail’s “Privacy” controls. This password check is performed through a character-by-character string comparison, and so an attacker who is able to measure the time taken by this check to a high degree of accuracy could potentially use timing differences to gain knowledge of the password. (This is understood to be feasible on a local network, but not on the public internet.)

Many thanks to Thibaud Colas for reporting this issue.

## 1.9.59 Wagtail 2.7.2 release notes

April 14, 2020

### CVE-2020-11001: Possible XSS attack via page revision comparison view

This release addresses a cross-site scripting (XSS) vulnerability on the page revision comparison view within the Wagtail admin interface. A user with a limited-permission editor account for the Wagtail admin could potentially craft a page revision history that, when viewed by a user with higher privileges, could perform actions with that user’s credentials. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to Vlad Gerasimenko for reporting this issue.

## 1.9.60 Wagtail 2.7.1 release notes

January 8, 2020

- *What’s new*

## What’s new

## Bug fixes

- Management command startup checks under `ManifestStaticFilesStorage` no longer fail if `collectstatic` has not been run first (Alex Tomkins)

## 1.9.61 Wagtail 2.7 release notes

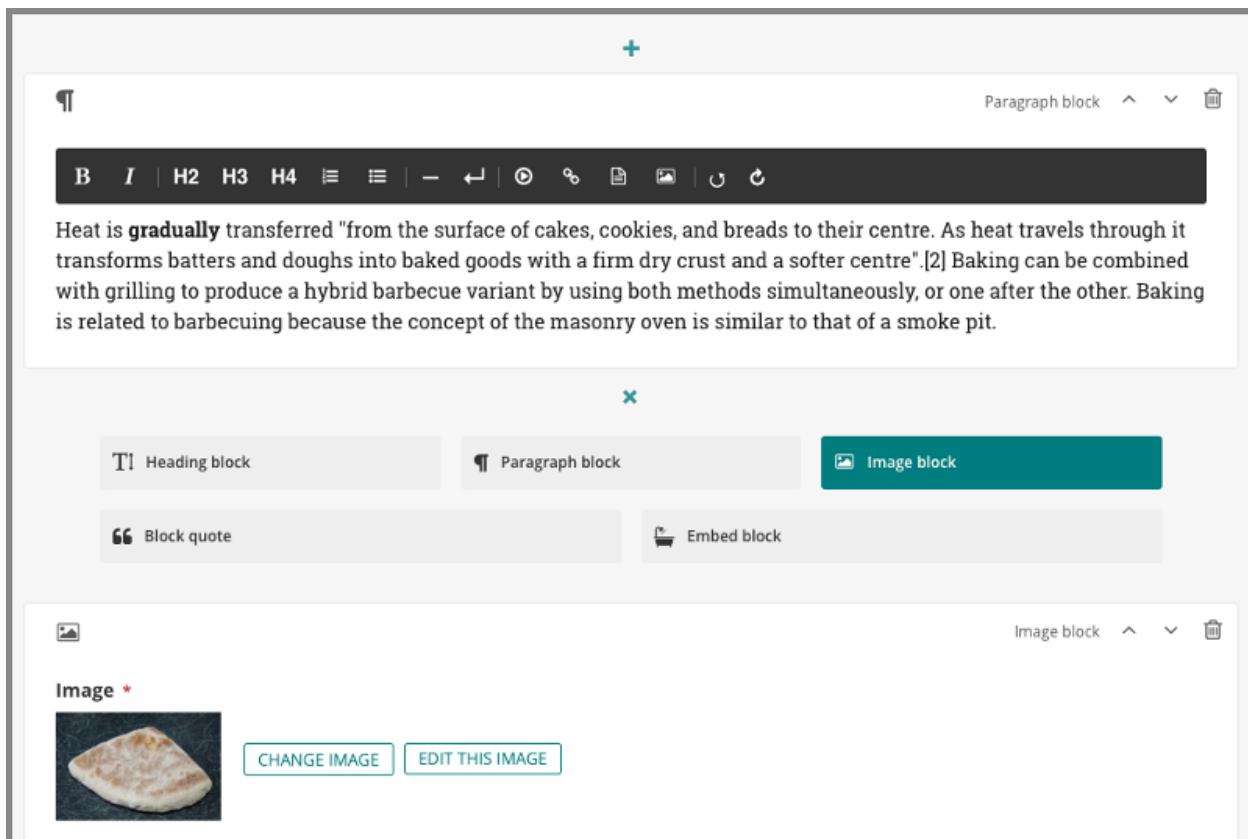
November 6, 2019

- [What's new](#)
- [Upgrade considerations](#)

Wagtail 2.7 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 12 months).

### What's new

#### Improved StreamField design



The design of the StreamField user interface has been updated to improve clarity and usability, including better handling of nested blocks. This work was completed by Bertrand Bordage as part of the [Wagtail's First Hatch](#) crowdfunding campaign. We would like to thank all supporters of the campaign.

### WebP image support

Images can now be uploaded and rendered in WebP format; see [Image file formats](#) for details. This feature was developed by frmdstryr, Karl Hobley and Matt Westcott.

### Other features

- Added Elasticsearch 7 support (pySilver)
- Added Python 3.8 support (John Carter, Matt Westcott)
- Added `construct_page_listing_buttons` hook (Michael van Tellingen)
- Added more detailed documentation and troubleshooting for installing OpenCV for feature detection (Daniele Procida)
- Move and refactor upgrade notification JS (Jonny Scholes)
- Remove need for Elasticsearch `update_all_types` workaround, upgrade minimum release to 6.4.0 or above (Jonathan Liuti)
- Add ability to insert internal anchor links/links with fragment identifiers in Draftail (rich text) fields (Iman Syed)
- Added Table Block caption for accessibility (Rahmi Pruitt)
- Add ability for users to change their own name via the account settings page (Kevin Howbrook)
- Add ability to insert telephone numbers as links in Draftail (rich text) fields (Mikael Engström and Liam Brenner)
- Increase delay before search in the snippet chooser, to prevent redundant search request round trips (Robert Rollins)
- Add `WAGTAIL_EMAIL_MANAGEMENT_ENABLED` setting to determine whether users can change their email address (Janne Alatalo)
- Recognise Soundcloud artist URLs as embeddable (Kiril Staikov)
- Add `WAGTAILDOCS_SERVE_METHOD` setting to determine how document downloads will be linked to and served (Tobias McNulty, Matt Westcott)
- Add `WAGTAIL_MODERATION_ENABLED` setting to enable / disable the ‘Submit for Moderation’ option (Jacob Topp-Muggleton) - thanks to [The Motley Fool](#) for sponsoring this feature
- Added settings to customise pagination page size for the Images admin area (Brian Whitton)
- Added ARIA role to TableBlock output (Matt Westcott)
- Added cache-busting query parameters to static files within the Wagtail admin (Matt Westcott)
- Allow `register_page_action_menu_item` and `construct_page_action_menu` hooks to override the default menu action (Rahmi Pruitt, Matt Westcott) - thanks to [The Motley Fool](#) for sponsoring review of this feature
- `WAGTAILIMAGES_MAX_IMAGE_PIXELS` limit now takes the number of animation frames into account (Karl Hobley)

## Bug fixes

- Added line breaks to long filenames on multiple image / document uploader (Kevin Howbrook)
- Added https support for Scribd oEmbed provider (Rodrigo)
- Changed StreamField group label colour so labels are visible (Catherine Farman)
- Prevented images with a very wide aspect ratio from being displayed distorted in the rich text editor (Iman Syed)
- Prevent exception when deleting a model with a protected One-to-one relationship (Neal Todd)
- Added labels to snippet bulk edit checkboxes for screen reader users (Martey Dodox)
- Middleware responses during page preview are now properly returned to the user (Matt Westcott)
- Default text of page links in rich text uses the public page title rather than the admin display title (Andy Chosak)
- Specific page permission checks are now enforced when viewing a page revision (Andy Chosak)
- `pageurl` and `slugurl` tags no longer fail when `request.site` is `None` (Samir Shah)
- Output form media on add/edit image forms with custom models (Matt Westcott)
- Output form media on add/edit document forms with custom models (Sergey Fedoseev)
- Fixes layout for the clear checkbox in default FileField widget (Mikalai Radchuk)
- Remove ASCII conversion from Postgres search backend, to support stemming in non-Latin alphabets (Pavel Denisov)
- Prevent tab labels on page edit view from being cut off on very narrow screens (Kevin Howbrook)
- Very long words in page listings are now broken where necessary (Kevin Howbrook)
- Language chosen in user preferences no longer persists on subsequent requests (Bojan Mihelac)
- Prevent new block IDs from being assigned on repeated calls to `StreamBlock.get_prep_value` (Colin Klein)
- Prevent broken images in notification emails when static files are hosted on a remote domain (Eduard Luca)
- Replace styleguide example avatar with default image to avoid issues when custom user model is used (Matt Westcott)
- `DraftailRichTextArea` is no longer treated as a hidden field by Django's form logic (Sergey Fedoseev)
- Replace `format()` placeholders in translatable strings with `%` formatting (Matt Westcott)
- Altering Django REST Framework's `DEFAULT_AUTHENTICATION_CLASSES` setting no longer breaks the page explorer menu and admin API (Matt Westcott)
- Regression - missing label for external link URL field in link chooser (Stefani Castellanos)

## Upgrade considerations

### Query strings added to static file URLs within the admin

To avoid problems caused by outdated cached JavaScript / CSS files following a Wagtail upgrade, URLs to static files within the Wagtail admin now include a version-specific query parameter of the form `?v=1a2b3c4d`. Under certain front-end cache configurations (such as [Cloudflare's 'No Query String' caching level](#)), the presence of this parameter may prevent the file from being cached at all. If you are using such a setup, and have some other method in place to expire outdated files (e.g. clearing the cache on deployment), you can disable the query parameter by setting `WAGTAILADMIN_STATIC_FILE_VERSION_STRINGS` to `False` in your project settings. (Note that this is automatically disabled when `ManifestStaticFilesStorage` is in use.)

## Page.dummy\_request is deprecated

The internal `Page.dummy_request` method (which generates an HTTP request object simulating a real page request, for use in previews) has been deprecated, as it did not correctly handle errors generated during middleware processing. Any code that calls this method to render page previews should be updated to use the new method `Page.make_preview_request(original_request=None, preview_mode=None)`, which builds the request and calls `Page.serve_preview` as a single operation.

## Changes to document serving on remote storage backends (Amazon S3 etc)

This release introduces a new setting `WAGTAILDOCS_SERVE_METHOD` to control how document downloads are served. On previous versions of Wagtail, document files would always be served through a Django view, to allow permission checks to be applied. When using a remote storage backend such as Amazon S3, this meant that the document would be downloaded to the Django server on every download request.

In Wagtail 2.7, the default behaviour on remote storage backends is to redirect to the storage's underlying URL after performing the permission check. If this is unsuitable for your project (for example, your storage provider is configured to block public access, or revealing its URL would be a security risk) you can revert to the previous behaviour by setting `WAGTAILDOCS_SERVE_METHOD` to '`'serve_view'`'.

## Template change for page action menu hooks

When customising the action menu on the page edit view through the `register_page_action_menu_item` or `construct_page_action_menu` hook, the `ActionMenuItem` object's `template` attribute or `render_html` method can be overridden to customise the menu item's HTML. As of Wagtail 2.7, the HTML returned from these should *not* include the enclosing `<li>` element.

Any add-on library that uses this feature and needs to preserve backward compatibility with previous Wagtail versions can conditionally reinsert the `<li>` wrapper through its `render_html` method - for example:

```
from django.utils.html import format_html
from wagtail import VERSION as WAGTAIL_VERSION
from wagtail.admin.action_menu import ActionMenuItem

class CustomMenuItem(ActionMenuItem):
 template = 'myapp/my_menu_item.html'

 def render_html(self, request, parent_context):
 html = super().render_html(request, parent_context)
 if WAGTAIL_VERSION < (2, 7):
 html = format_html('{}', html)
 return html
```

## wagtail.admin.utils and wagtail.admin.decorators modules deprecated

The modules `wagtail.admin.utils` and `wagtail.admin.decorators` have been deprecated. The helper functions defined here exist primarily for Wagtail's internal use; however, some of them (particularly `send_mail` and `permission_required`) may be found in user code, and import lines will need to be updated. The new locations for these definitions are as follows:

Definition	Old location	New location
<code>any_permission_required</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.auth</code>
<code>permission_denied</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.auth</code>
<code>permission_required</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.auth</code>
<code>PermissionPolicyChecker</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.auth</code>
<code>user_has_any_page_permission</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.auth</code>
<code>user_passes_test</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.auth</code>
<code>users_with_page_permission</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.auth</code>
<code>reject_request</code>	<code>wagtail.admin.decorators</code>	<code>wagtail.admin.auth</code>
<code>require_admin_access</code>	<code>wagtail.admin.decorators</code>	<code>wagtail.admin.auth</code>
<code>get_available_admin_languages</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.localization</code>
<code>get_available_admin_time_zones</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.localization</code>
<code>get_js_translation_strings</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.localization</code>
<code>WAGTAILADMIN_PROVIDED_LANGUAGES</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.localization</code>
<code>send_mail</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.mail</code>
<code>send_notification</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.mail</code>
<code>get_object_usage</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.models</code>
<code>popular_tags_for_model</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.models</code>
<code>get_site_for_user</code>	<code>wagtail.admin.utils</code>	<code>wagtail.admin.navigation</code>

## 1.9.62 Wagtail 2.6.3 release notes

October 22, 2019

- *What's new*

## What's new

### Bug fixes

- Altering Django REST Framework's DEFAULT\_AUTHENTICATION\_CLASSES setting no longer breaks the page explorer menu and admin API (Matt Westcott)

## 1.9.63 Wagtail 2.6.2 release notes

*September 19, 2019*

- *What's new*

## What's new

### Bug fixes

- Prevent search indexing failures on Postgres 9.4 and Django >= 2.2.1 (Matt Westcott)

## 1.9.64 Wagtail 2.6.1 release notes

*August 5, 2019*

- *What's new*

## What's new

### Bug fixes

- Prevent JavaScript errors caused by unescaped quote characters in translation strings (Matt Westcott)

## 1.9.65 Wagtail 2.6 release notes

*August 1, 2019*

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

## What's new

### Accessibility targets and improvements

Wagtail now has official accessibility support targets: we are aiming for compliance with WCAG2.1, AA level. WCAG 2.1 is the international standard which underpins many national accessibility laws.

Wagtail isn't fully compliant just yet, but we have made many changes to the admin interface to get there. We thank the UK Government (in particular the CMS team at the Department for International Trade), who commissioned many of these improvements.

Here are changes which should make Wagtail more usable for all users regardless of abilities:

- Increase font-size across the whole admin (Beth Menzies, Katie Locke)
- Improved text colour contrast across the whole admin (Beth Menzies, Katie Locke)
- Added consistent focus outline styles across the whole admin (Thibaud Colas)
- Ensured the ‘add child page’ button displays when focused (Helen Chapman, Katie Locke)

This release also contains many big improvements for screen reader users:

- Added more ARIA landmarks across the admin interface and welcome page for screen reader users to navigate the CMS more easily (Beth Menzies)
- Improved heading structure for screen reader users navigating the CMS admin (Beth Menzies, Helen Chapman)
- Make icon font implementation more screen-reader-friendly (Thibaud Colas)
- Removed buggy tab order customisations in the CMS admin (Jordan Bauer)
- Screen readers now treat page-level action dropdowns as navigation instead of menus (Helen Chapman)
- Fixed occurrences of invalid HTML across the CMS admin (Thibaud Colas)
- Add empty alt attributes to all images in the CMS admin (Andreas Bernacca)
- Fixed focus not moving to the pages explorer menu when open (Helen Chapman)

We've also had a look at how controls are labelled across the UI for screen reader users:

- Add image dimensions in image gallery and image choosers for screen reader users (Helen Chapman)
- Add more contextual information for screen readers in the explorer menu’s links (Helen Chapman)
- Make URL generator preview image alt translatable (Thibaud Colas)
- Screen readers now announce “Dashboard” for the main nav’s logo link instead of Wagtail’s version number (Thibaud Colas)
- Remove duplicate labels in image gallery and image choosers for screen reader users (Helen Chapman)
- Added a label to the modals’ “close” button for screen reader users (Helen Chapman, Katie Locke)
- Added labels to permission checkboxes for screen reader users (Helen Chapman, Katie Locke)
- Improve screen-reader labels for action links in page listing (Helen Chapman, Katie Locke)
- Add screen-reader labels for table headings in page listing (Helen Chapman, Katie Locke)
- Add screen reader labels for page privacy toggle, edit lock, status tag in page explorer & edit views (Helen Chapman, Katie Locke)
- Add screen-reader labels for dashboard summary cards (Helen Chapman, Katie Locke)
- Add screen-reader labels for privacy toggle of collections (Helen Chapman, Katie Locke)

Again, this is still a work in progress – if you are aware of other existing accessibility issues, please do [open an issue](#) if there isn’t one already.

### Other features

- Added support for `short_description` for field labels in modeladmin’s `InspectView` (Wesley van Lee)
- Rearranged SCSS folder structure to the client folder and split them approximately according to ITCSS. (Naomi Morduch Toubman, Jonny Scholes, Janneke Janssen, Hugo van den Berg)
- Added support for specifying cell alignment on `TableBlock` (Samuel Mendes)
- Added more informative error when a non-image object is passed to the `image` template tag (Deniz Dogan)
- Added `ButtonHelper` examples in the modelAdmin primer page within documentation (Kalob Taulien)
- Multiple clarifications, grammar and typo fixes throughout documentation (Dan Swain)
- Use correct URL in API example in documentation (Michael Bunsen)
- Move datetime widget initialiser JS into the widget’s form media instead of page editor media (Matt Westcott)
- Add form field prefixes for input forms in chooser modals (Matt Westcott)
- Removed version number from the logo link’s title. The version can now be found under the Settings menu (Thibaud Colas)
- Added “don’t delete” option to confirmation screen when deleting images, documents and modeladmin models (Kevin Howbrook)
- Added `branding_title` template block for the admin title prefix (Dillen Meijboom)
- Added support for custom search handler classes to modeladmin’s `IndexView`, and added a class that uses the default Wagtail search backend for searching (Seb Brown, Andy Babic)
- Update group edit view to expose the `Permission` object for each checkbox (George Hickman)
- Improve performance of Pages for Moderation panel (Fidel Ramos)
- Added `process_child_object` and `exclude_fields` arguments to `Page.copy()` to make it easier for third-party apps to customise copy behaviour (Karl Hobley)
- Added `Page.with_content_json()`, allowing revision content loading behaviour to be customised on a per-model basis (Karl Hobley)
- Added `construct_settings_menu` hook (Jordan Bauer, Quadric)
- Fixed compatibility of date / time choosers with wagtail-react-streamfield (Mike Hearn)
- Performance optimization of several admin functions, including breadcrumbs, home and index pages (Fidel Ramos)

## Bug fixes

- ModelAdmin no longer fails when filtering over a foreign key relation (Jason Dilworth, Matt Westcott)
- The Wagtail version number is now visible within the Settings menu (Kevin Howbrook)
- Scaling images now rounds values to an integer so that images render without errors (Adrian Brunyate)
- Revised test decorator to ensure TestPageEditHandlers test cases run correctly (Alex Tomkins)
- Wagtail bird animation in admin now ends correctly on all browsers (Deniz Dogan)
- Explorer menu no longer shows sibling pages for which the user does not have access (Mike Hearn)
- Admin HTML now includes the correct `dir` attribute for the active language (Andreas Bernacca)
- Fix type error when using `--chunk_size` argument on `./manage.py update_index` (Seb Brown)
- Avoid rendering entire form in EditHandler's `repr` method (Alex Tomkins)
- Add empty alt attributes to HTML output of Embedly and oEmbed embed finders (Andreas Bernacca)
- Clear pending AJAX request if error occurs on page chooser (Matt Westcott)
- Prevent text from overlapping in focal point editing UI (Beth Menzies)
- Restore custom “Date” icon for scheduled publishing panel in Edit page’s Settings tab (Helen Chapman)
- Added missing form media to user edit form template (Matt Westcott)
- `Page.copy()` no longer copies child objects when the accessor name is included in `exclude_fields_in_copy` (Karl Hobley)
- Clicking the privacy toggle while the page is still loading no longer loads the wrong data in the page (Helen Chapman)
- Added missing `is_stored_locally` method to `AbstractDocument` (jonny5532)
- Query model no longer removes punctuation as part of string normalisation (William Blackie)
- Make login test helper work with user models with non-default username fields (Andrew Miller)
- Delay dirty form check to prevent “unsaved changes” warning from being wrongly triggered (Thibaud Colas)

## Upgrade considerations

### Removed support for Python 3.4

Python 3.4 is no longer supported as of this release; please upgrade to Python 3.5 or above before upgrading Wagtail.

### Icon font implementation changes

The icon font implementation has been changed to be invisible for screen-reader users, by switching to using [Private Use Areas](#) Unicode code points. All of the icon classes (`icon-user`, `icon-search`, etc) should still work the same, except for two which have been removed because they were duplicates:

- `icon-picture` is removed. Use `icon-image` instead (same visual).
- `icon-file-text-alt` is removed. Use `icon-doc-full` instead (same visual).

For a list of all available icons, please see the [UI Styleguide](#).

## 1.9.66 Wagtail 2.5.2 release notes

August 1, 2019

- *What's new*

### What's new

### Bug fixes

- Delay dirty form check to prevent “unsaved changes” warning from being wrongly triggered (Thibaud Colas)

## 1.9.67 Wagtail 2.5.1 release notes

May 7, 2019

- *What's new*

### What's new

### Bug fixes

- Prevent crash when comparing StructBlocks in revision history (Adrian Turjak, Matt Westcott)

## 1.9.68 Wagtail 2.5 release notes

April 24, 2019

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

### What's new

### Django 2.2 support

This release is compatible with Django 2.2. Compatibility fixes were contributed by Matt Westcott and Andy Babic.

## New Markdown shortcuts in rich text

Wagtail's rich text editor now supports using Markdown shortcuts for inline formatting:

- \*\* for bold
- \_ for italic
- ~ for strikethrough (if enabled)
- ` for code (if enabled)

To learn other shortcuts, have a look at the [keyboard shortcuts](#) reference.

## Other features

- Added support for customising EditHandler-based forms on a per-request basis (Bertrand Bordage)
- Added more informative error message when `|richtext` filter is applied to a non-string value (mukesh5)
- Automatic search indexing can now be disabled on a per-model basis via the `search_auto_update` attribute (Karl Hobley)
- Improved diffing of StreamFields when comparing page revisions (Karl Hobley)
- Highlight broken links to pages and missing documents in rich text (Brady Moe)
- Preserve links when copy-pasting rich text content from Wagtail to other tools (Thibaud Colas)
- Rich text to contentstate conversion now prioritises more specific rules, to accommodate `<p>` and `<br>` elements with attributes (Matt Westcott)
- Added limit image upload size by number of pixels (Thomas Elliott)
- Added `manage.py wagtail_update_index` alias to avoid clashes with `update_index` commands from other packages (Matt Westcott)
- Renamed `target_model` argument on `PageChooserBlock` to `page_type` (Loic Teixeira)
- `edit_handler` and `panels` can now be defined on a `ModelAdmin` definition (Thomas Kremmel)
- Add Learn Wagtail to third-party tutorials in documentation (Matt Westcott)
- Add a Django setting `TAG_LIMIT` to limit number of tags that can be added to any taggit model (Mani)
- Added instructions on how to generate urls for `ModelAdmin` to documentation (LB (Ben Johnston), Andy Babic)
- Added option to specify a fallback URL on `{% pageurl %}` (Arthur Holzner)
- Add support for more rich text formats, disabled by default: `blockquote`, `superscript`, `subscript`, `strikethrough`, `code` (Md Arifin Ibne Matin)
- Added `max_count_per_parent` option on page models to limit the number of pages of a given type that can be created under one parent page (Wesley van Lee)
- `StreamField` field blocks now accept a `validators` argument (Tom Usher)
- Added edit / delete buttons to snippet index and “don’t delete” option to confirmation screen, for consistency with pages (Kevin Howbrook)
- Added name attributes to all built-in page action menu items (LB (Ben Johnston))
- Added validation on the filter string to the Jinja2 image template tag (Jonny Scholes)
- Changed the pages reordering UI toggle to make it easier to find (Katie Locke, Thibaud Colas)

- Added support for rich text link rewrite handlers for `external` and `email` links (Md Arifin Ibne Matin)
- Clarify installation instructions in documentation, especially regarding virtual environments. (Naomi Morduch Toubman)

### Bug fixes

- Set `SERVER_PORT` to 443 in `Page.dummy_request()` for HTTPS sites (Sergey Fedoseev)
- Include port number in `Host` header of `Page.dummy_request()` (Sergey Fedoseev)
- Validation error messages in `InlinePanel` no longer count towards `max_num` when disabling the ‘add’ button (Todd Dembrey, Thibaud Colas)
- Rich text to contentstate conversion now ignores stray closing tags (frmstryr)
- Escape backslashes in `postgres_search` queries (Hammy Goonan)
- Parent page link in page chooser search results no longer navigates away (Asanka Lihiniyagoda, Sævar Öfjörð Magnússon)
- `routablepageurl` tag now correctly omits domain part when multiple sites exist at the same root (Gassan Gousseinov)
- Added missing collection column specifier on document listing template (Sergey Fedoseev)
- Page Copy will now also copy ParentalManyToMany field relations (LB (Ben Johnston))
- Admin HTML header now includes correct language code (Matt Westcott)
- Unclear error message when saving image after focal point edit (Hugo van den Berg)
- Increase max length on `Embed.thumbnail_url` to 255 characters (Kevin Howbrook)
- `send_mail` now correctly uses the `html_message` kwarg for HTML messages (Tiago Requeijo)
- Page copying no longer allowed if page model has reached its `max_count` (Andy Babic)
- Don’t show page type on page chooser button when multiple types are allowed (Thijs Kramer)
- Make sure page chooser search results correspond to the latest search by cancelling previous requests (Esper Kuijs)
- Inform user when moving a page from one parent to another where there is an already existing page with the same slug (Casper Timmers)
- User add/edit forms now support form widgets with JS/CSS media (Damian Grinwis)
- Rich text processing now preserves non-breaking spaces instead of converting them to normal spaces (Wesley van Lee)
- Prevent autocomplete dropdowns from appearing over date choosers on Chrome (Kevin Howbrook)
- Prevent crash when logging HTTP errors on Cloudflare cache purging (Kevin Howbrook)
- Prevent rich text editor crash when filtering copy-pasted content and the last block is to be removed, e.g. unsupported image (Thibaud Colas)
- Removing rich text links / documents now also works when the text selection is backwards (Thibaud Colas)
- Prevent the rich text editor from crashing when copy-paste filtering removes all of its content (Thibaud Colas)
- Page chooser now respects custom `get_admin_display_title` methods on parent page and breadcrumb (Haydn Greatnews)
- Added consistent whitespace around sortable table headings (Matt Westcott)

- Moved locale names for Chinese (Simplified) and Chinese (Traditional) to zh\_Hans and zh\_Hant (Matt Westcott)

## Upgrade considerations

### EditHandler.bind\_to\_model and EditHandler.bind\_to\_instance deprecated

The internal `EditHandler` methods `bind_to_model` and `bind_to_instance` have been deprecated, in favour of a new combined `bind_to` method which accepts `model`, `instance`, `request` and `form` as optional keyword arguments. Any user code which calls `EditHandler.bind_to_model(model)` should be updated to use `EditHandler.bind_to(model=model)` instead; any user code which calls `EditHandler.bind_to_instance(instance, request, form)` should be updated to use `EditHandler.bind_to(instance=instance, request=request, form=form)`.

## Changes to admin pagination helpers

A number of changes have been made to pagination handling within the Wagtail admin; these are internal API changes, but may affect applications and third-party packages that add new paginated object listings, including chooser modals, to the admin. The `paginate` function in `wagtail.utils.pagination` has been deprecated in favour of the `django.core.paginator.Paginator.get_page` method introduced in Django 2.0 - a call such as:

```
from wagtail.utils.pagination import paginate

paginator, page = paginate(request, object_list, per_page=25)
```

should be replaced with:

```
from django.core.paginator import Paginator

paginator = Paginator(object_list, per_page=25)
page = paginator.get_page(request.GET.get('p'))
```

Additionally, the `is_ajax` flag on the template `wagtailadmin/shared/pagination_nav.html` has been deprecated in favour of a new template `wagtailadmin/shared/ajax_pagination_nav.html`:

```
{% include "wagtailadmin/shared/pagination_nav.html" with items=page_obj is_ajax=1 %}
```

should become:

```
{% include "wagtailadmin/shared/ajax_pagination_nav.html" with items=page_obj %}
```

## New rich text formats

Wagtail now has built-in support for new rich text formats, disabled by default:

- `blockquote`, using the `blockquote` Draft.js block type, saved as a `<blockquote>` tag.
- `superscript`, using the `SUPERSCRIPT` Draft.js inline style, saved as a `<sup>` tag.
- `subscript`, using the `SUBSCRIPT` Draft.js inline style, saved as a `<sub>` tag.
- `strikethrough`, using the `STRIKETHROUGH` Draft.js inline style, saved as a `<s>` tag.
- `code`, using the `CODE` Draft.js inline style, saved as a `<code>` tag.

Projects already using those exact Draft.js type and HTML tag combinations can safely replace their feature definitions with the new built-ins. Projects that use the same feature identifier can keep their existing feature definitions as overrides. Finally, if the Draft.js types / HTML tags are used but with a different combination, do not enable the new feature definitions to avoid conflicts in storage or editor behaviour.

### **register\_link\_type and register\_embed\_type methods for rich text tag rewriting have changed**

The `FeatureRegistry.register_link_type` and `FeatureRegistry.register_embed_type` methods, which define how links and embedded media in rich text are converted to HTML, now accept a handler class. Previously, they were passed an identifier string and a rewrite function. For details of updating your code to the new convention, see [Rewrite handlers](#).

### **Chinese language locales changed to zh\_Hans and zh\_Hant**

The translations for Chinese (Simplified) and Chinese (Traditional) are now available under the locale names `zh_Hans` and `zh_Hant` respectively, rather than `zh_CN` and `zh_TW`. Projects that currently use the old names for the `LANGUAGE_CODE` setting may need to update the settings file to use the new names.

## **1.9.69 Wagtail 2.4 release notes**

*December 19, 2018*

- [What's new](#)
- [Upgrade considerations](#)

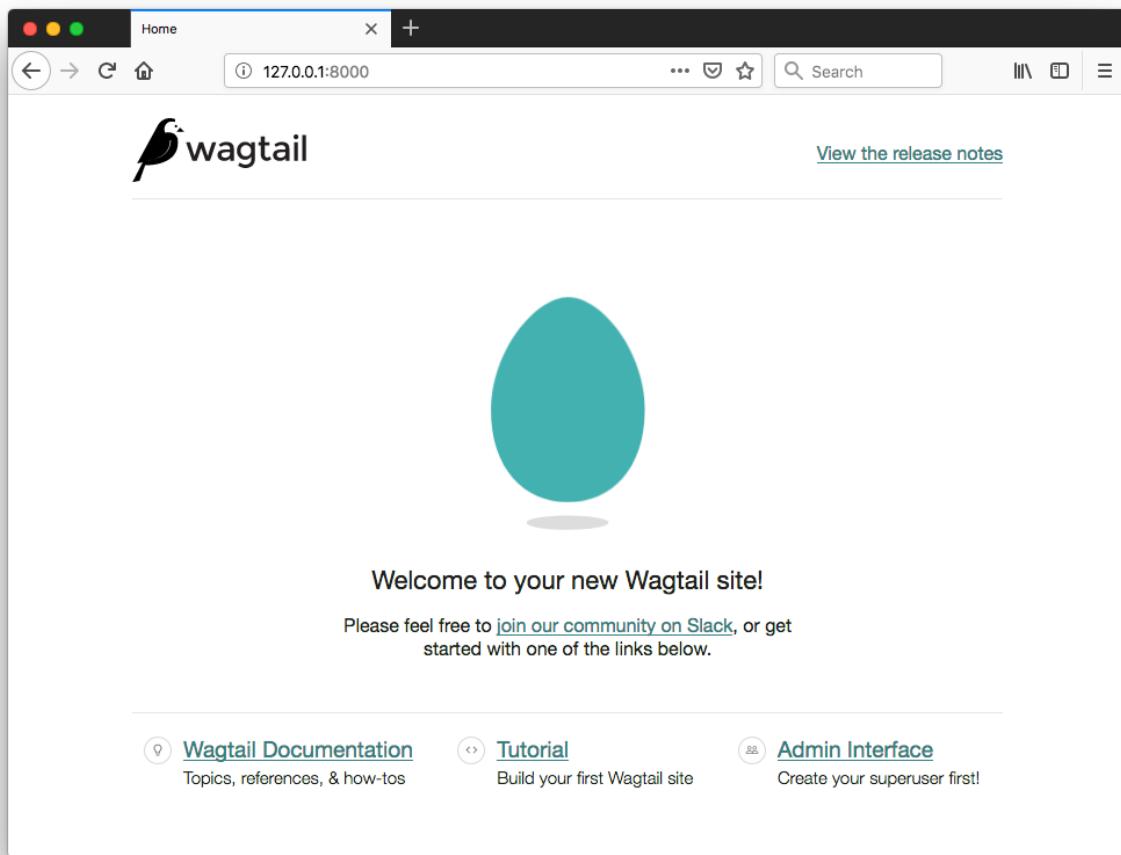
### **What's new**

#### **New “Welcome to your Wagtail site” Starter Page**

When using the `wagtail start` command to make a new site, users will now be greeted with a proper starter page. Thanks to Timothy Allen and Scott Cranfill for pulling this off!

### **Other features**

- Added support for Python 3.7 (Matt Westcott)
- Added `max_count` option on page models to limit the number of pages of a particular type that can be created (Dan Braghis)
- Document and image choosers now show the document / image’s collection (Alejandro Garza, Janneke Janssen)
- New `image_url` template tag allows to generate dynamic image URLs, so image renditions are being created outside the main request which improves performance. Requires extra configuration, see [Dynamic image serve view](#) (Yannick Chabbert, Dan Braghis).
- Added ability to run individual tests through tox (Benjamin Bach)
- Collection listings are now ordered by name (Seb Brown)



- Added `file_hash` field to documents (Karl Hobley, Dan Braghis)
- Added last login to the user overview (Noah B Johnson)
- Changed design of image editing page (Janneke Janssen, Ben Enright)
- Added Slovak character map for JavaScript slug generation (Andy Chosak)
- Make documentation links on welcome page work for prereleases (Matt Westcott)
- Allow overridden `copy()` methods in `Page` subclasses to be called from the page copy view (Robert Rollins)
- Users without a preferred language set on their profile now use language selected by Django's `LocaleMiddleware` (Benjamin Bach)
- Added hooks to customise the actions menu on the page create/edit views (Matt Westcott)
- Cleanup: Use `functools.partial()` instead of `django.utils.functional.curry()` (Sergey Fedoseev)
- Added `before_move_page` and `after_move_page` hooks (Maylon Pedroso)
- Bulk deletion button for snippets is now hidden until items are selected (Karl Hobley)

### Bug fixes

- Query objects returned from `PageQuerySet.type_q` can now be merged with `|` (Brady Moe)
- Add `rel="noopener noreferrer"` to target blank links (Anselm Bradford)
- Additional fields on custom document models now show on the multiple document upload view (Robert Rollins, Sergey Fedoseev)
- Help text does not overflow when using a combination of `BooleanField` and `FieldPanel` in page model (Dzianis Sheka)
- Document chooser now displays more useful help message when there are no documents in Wagtail document library (gmmoraes, Stas Rudakou)
- Allow custom logos of any height in the admin menu (Meteor0id)
- Allow nav menu to take up all available space instead of scrolling (Meteor0id)
- Users without the edit permission no longer see “Edit” links in list of pages waiting for moderation (Justin Focus, Fedor Selitsky)
- Redirects now return 404 when destination is unspecified or a page with no site (Hillary Jeffrey)
- Refactor all breakpoint definitions, removing style overlaps (Janneke Janssen)
- Updated `draftjs_exporter` to 2.1.5 to fix bug in handling adjacent entities (Thibaud Colas)
- Page titles consisting only of stopwords now generate a non-empty default slug (Andy Chosak, Janneke Janssen)
- Sitemap generator now allows passing a sitemap instance in the URL configuration (Mitchel Cabuloy, Dan Braghis)

## Upgrade considerations

### Removed support for Django 1.11

Django 1.11 is no longer supported in this release; please upgrade your project to Django 2.0 or 2.1 before upgrading to Wagtail 2.4.

### Custom image model migrations created on Wagtail <1.8 may fail

Projects with a custom image model (see [Custom image models](#)) created on Wagtail 1.7 or earlier are likely to have one or more migrations that refer to the (now-deleted) `wagtailimages.Filter` model. In Wagtail 2.4, the migrations that defined this model have been squashed, which may result in the error `ValueError: Related model 'wagtailimages.Filter' cannot be resolved` when bringing up a new instance of the database. To rectify this, check your project's migrations for `ForeignKey` references to `wagtailimages.Filter`, and change them to `IntegerField` definitions. For example, the line:

```
('filter', models.ForeignKey(blank=True, null=True, on_delete=django.db.models.deletion.
 ↪ CASCADE, related_name='+' , to='wagtailimages.Filter')),
```

should become:

```
('filter', models.IntegerField(blank=True, null=True)),
```

## 1.9.70 Wagtail 2.3 release notes

*October 23, 2018*

- [What's new](#)
- [Upgrade considerations](#)

Wagtail 2.3 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 8 months).

Note that Wagtail 2.3 will be the last release branch to support Django 1.11.

### What's new

#### Added Django 2.1 support

Wagtail is now compatible with Django 2.1. Compatibility fixes were contributed by Ryan Verner and Matt Westcott.

## Improved colour contrast

Colour contrast within the admin interface has been improved, and now complies with WCAG 2 level AA. This was completed by Coen van der Kamp and Naomi Morduch Toubman based on earlier work from Edd Baldry, Naa Marteki Reed and Ben Enright.

## Other features

- Added ‘scale’ image filter (Oliver Wilkerson)
- Added meta tag to prevent search engines from indexing admin pages (Karl Hobley)
- EmbedBlock now validates against recognised embed providers on save (Bertrand Bordage)
- Made cache control headers on Wagtail admin consistent with Django admin (Tomasz Knapik)
- Notification emails now include an “Auto-Submitted: auto-generated” header (Dan Braghis)
- Image chooser panels now show alt text as title (Samir Shah)
- Added `download_url` field to images in the API (Michael Harrison)
- Dummy requests for preview now preserve the HTTP Authorization header (Ben Dickinson)

## Bug fixes

- Respect next param on login (Loic Teixeira)
- InlinePanel now handles relations that specify a `related_query_name` (Aram Dulyan)
- `before_delete_page` / `after_delete_page` hooks now run within the same database transaction as the page deletion (Tomasz Knapik)
- Seek to the beginning of image files when uploading, to restore compatibility with django-storages Google Cloud and Azure backends (Mikalai Radchuk)
- Snippet chooser modal no longer fails on snippet models with UUID primary keys (Sævar Öfjörð Magnússon)
- Restored localisation in date/time pickers (David Moore, Thibaud Colas)
- Tag input field no longer treats ‘‘ on Russian keyboards as a comma (Michael Borisov)
- Disabled autocomplete dropdowns on date/time chooser fields (Janneke Janssen)
- Split up `wagtail.admin.forms` to make it less prone to circular imports (Matt Westcott)
- Disable linking to root page in rich text, making the page non-functional (Matt Westcott)
- Pages should be editable and save-able even if there are broken page or document links in rich text (Matt Westcott)
- Avoid redundant round-trips of JSON StreamField data on save, improving performance and preventing consistency issues on fixture loading (Andy Chosak, Matt Westcott)
- Users are not logged out when changing their own password through the Users area (Matt Westcott)

## Upgrade considerations

### wagtail.admin.forms reorganised

The `wagtail.admin.forms` module has been split up into submodules to make it less prone to producing circular imports, particularly when a custom user model is in use. The following (undocumented) definitions have now been moved to new locations:

Definition	New location
LoginForm	wagtail.admin.forms.auth
PasswordResetForm	wagtail.admin.forms.auth
URLOrAbsolutePathValidator	wagtail.admin.forms.choosers
URLOrAbsolutePathField	wagtail.admin.forms.choosers
ExternalLinkChooserForm	wagtail.admin.forms.choosers
EmailLinkChooserForm	wagtail.admin.forms.choosers
CollectionViewRestrictionForm	wagtail.admin.forms.collections
CollectionForm	wagtail.admin.forms.collections
BaseCollectionMemberForm	wagtail.admin.forms.collections
BaseGroupCollectionMemberPermissionFormSet	wagtail.admin.forms.collections
collection_member_permission_formset_factory	wagtail.admin.forms.collections
CopyForm	wagtail.admin.forms.pages
PageViewRestrictionForm	wagtail.admin.forms.pages
SearchForm	wagtail.admin.forms.search
BaseViewRestrictionForm	wagtail.admin.forms.view_restrictions

The following definitions remain in `wagtail.admin.forms`: FORM\_FIELD\_OVERRIDES, DIRECT\_FORM\_FIELD\_OVERRIDES, `formfield_for_dbfield`, WagtailAdminModelFormMetaclass, WagtailAdminModelForm and WagtailAdminPageForm.

## 1.9.71 Wagtail 2.2.2 release notes

August 29, 2018

- *What's new*

### What's new

### Bug fixes

- Seek to the beginning of image files when uploading, to restore compatibility with django-storages Google Cloud and Azure backends (Mikalai Radchuk)
- Respect next param on login (Loic Teixeira)

## 1.9.72 Wagtail 2.2.1 release notes

August 13, 2018

- [What's new](#)

### What's new

#### Bug fixes

- Pin BeautifulSoup to 4.6.0 due to further regressions in formatting empty elements (Matt Westcott)
- Prevent AppRegistryNotReady error when wagtail.contrib.sitemaps is in INSTALLED\_APPS (Matt Westcott)

## 1.9.73 Wagtail 2.2 release notes

August 10, 2018

- [What's new](#)
- [Upgrade considerations](#)

### What's new

#### Faceted search

Wagtail search now includes support for facets, allowing you to display search result counts broken down by a particular field value. For further details, see [Faceted search](#). This feature was developed by Karl Hobley.

#### Improved admin page search

The page search in the Wagtail admin now supports filtering by page type and ordering search results by title, creation date and status. This feature was developed by Karl Hobley.

#### Other features

- Added another valid AudioBoom oEmbed pattern (Bertrand Bordage)
- Added `annotate_score` support to PostgreSQL search backend (Bertrand Bordage)
- Pillow's image optimisation is now applied when saving PNG images (Dmitry Vasilev)
- JS / CSS media files can now be associated with Draftail feature definitions (Matt Westcott)
- The `{% slugurl %}` template tag is now site-aware (Samir Shah)
- Added `file_size` field to documents (Karl Hobley)
- Added `file_hash` field to images (Karl Hobley)

- Update documentation (configuring Django for Wagtail) to contain all current settings options (Matt Westcott, LB (Ben Johnston))
- Added `defer` flag to `PageQuerySet.specific` (Karl Hobley)
- Snippets can now be deleted from the listing view (LB (Ben Johnston))
- Increased max length of redirect URL field to 255 (Michael Harrison)
- Added documentation for new JS/CSS media files association with Draftail feature definitions (Ed Henderson)
- Added accessible colour contrast guidelines to the style guide (Catherine Farman)
- Admin modal views no longer rely on JavaScript `eval()`, for better CSP compliance (Matt Westcott)
- Update editor guide for embeds and documents in rich text (Kevin Howbrook)
- Improved performance of sitemap generation (Michael van Tellingen, Bertrand Bordage)
- Added an internal API for autocomplete (Karl Hobley)

## Bug fixes

- Handle all exceptions from `Image.get_file_size` (Andrew Plummer)
- Fix display of breadcrumbs in ModelAdmin (LB (Ben Johnston))
- Remove duplicate border radius of avatars (Benjamin Thurm)
- `Site.get_site_root_paths()` preferring other sites over the default when some sites share the same `root_page` (Andy Babic)
- Pages with missing model definitions no longer crash the API (Abdulmalik Abdulwahab)
- Rich text image chooser no longer skips format selection after a validation error (Matt Westcott)
- Null characters in URLs no longer crash the redirect middleware on PostgreSQL (Andrew Crewdson, Matt Westcott)
- Permission checks no longer prevent a non-live page from being unscheduled (Abdulmalik Abdulwahab)
- Copy-paste between Draftail editors now preserves all formatting/content (Thibaud Colas)
- Fix alignment of checkboxes and radio buttons on Firefox (Matt Westcott)

## Upgrade considerations

### JavaScript templates in modal workflows are deprecated

The `wagtail.admin.modal_workflow` module (used internally by Wagtail to handle modal popup interfaces such as the page chooser) has been updated to avoid returning JavaScript code as part of HTTP responses. User code that relies on this functionality can be updated as follows:

- Eliminate template tags from the `.js` template. Any dynamic data needed by the template can instead be passed in a dict to `render_modal_workflow`, as a keyword argument `json_data`; this data will then be available as the second parameter of the JavaScript function.
- At the point where you call the `ModalWorkflow` constructor, add an `onload` option - a dictionary of functions to be called on loading each step of the workflow. Move the code from the `.js` template into this dictionary. Then, on the call to `render_modal_workflow`, rather than passing the `.js` template name (which should now be replaced by `None`), pass a `step` item in the `json_data` dictionary to indicate the `onload` function to be called.

Additionally, if your code calls `loadResponseText` as part of a jQuery AJAX callback, this should now be passed all three arguments from the callback (the response data, status string and XMLHttpRequest object).

#### **Page.get\_sitemap\_urls() now accepts an optional request keyword argument**

The `Page.get_sitemap_urls()` method used by the `wagtail.contrib.sitemaps` module has been updated to receive an optional `request` keyword argument. If you have overridden this method in your page models, you will need to update the method signature to accept this argument (and pass it on when calling `super`, if applicable).

### **1.9.74 Wagtail 2.1.3 release notes**

*August 13, 2018*

- *What's new*

#### **What's new**

#### **Bug fixes**

- Pin Beautiful Soup to 4.6.0 due to further regressions in formatting empty elements (Matt Westcott)

### **1.9.75 Wagtail 2.1.2 release notes**

*August 6, 2018*

- *What's new*

#### **What's new**

#### **Bug fixes**

- Bundle the i18n package to avoid installation issues on systems with a non-Unicode locale (Matt Westcott)
- Mark Beautiful Soup 4.6.1 as incompatible due to bug in formatting empty elements (Matt Westcott)

### **1.9.76 Wagtail 2.1.1 release notes**

*July 4, 2018*

- *What's new*

## What's new

### Bug fixes

- Fix `Site.get_site_root_paths()` preferring other sites over the default when some sites share the same `root_page` (Andy Babic)
- Rich text image chooser no longer skips format selection after a validation error (Matt Westcott)
- Null characters in URLs no longer crash the redirect middleware on PostgreSQL (Matt Westcott)

## 1.9.77 Wagtail 2.1 release notes

May 22, 2018

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

## What's new

### New HelpPanel

A new panel type `HelpPanel` allows to easily add HTML within an edit form. This new feature was developed by Kevin Chung.

### Profile picture upload

Users can now upload profile pictures directly through the Account Settings menu, rather than using Gravatar. Gravatar is still used as a fallback if no profile picture has been uploaded directly; a new setting `WAGTAIL_GRAVATAR_PROVIDER_URL` has been added to specify an alternative provider, or disable the use of external avatars completely. This feature was developed by Daniel Chimeno, Pierre Geier and Matt Westcott.

### API lookup by page path

The API now includes an endpoint for finding pages by path; see *Finding pages by HTML path*. This feature was developed by Karl Hobley.

### User time zone setting

Users can now set their current time zone through the Account Settings menu, which will then be reflected in date / time fields throughout the admin (such as go-live / expiry dates). The list of available time zones can be configured via the `WAGTAIL_USER_TIME_ZONES` setting. This feature was developed by David Moore.

### Elasticsearch 6 support

Wagtail now supports Elasticsearch 6. See [Elasticsearch Backend](#) for configuration details. This feature was developed by Karl Hobley.

### Other features

- Persist tab hash in URL to allow direct navigation to tabs in the admin interface (Ben Weatherman)
- Animate the chevron icon when opening sub-menus in the admin (Carlo Ascani)
- Look through the target link and target page slug (in addition to the old slug) when searching for redirects in the admin (Michael Harrison)
- Remove support for IE6 to IE9 from project template (Samir Shah)
- Remove outdated X-UA-Compatible meta from admin template (Thibaud Colas)
- Add JavaScript source maps in production build for packaged Wagtail (Thibaud Colas)
- Removed assert statements from Wagtail API (Kim Chee Leong)
- Update *jquery-datetimepicker* dependency to make Wagtail more CSP-friendly (*unsafe-eval*) (Pomax)
- Added error notification when running the `wagtail` command on Python <3.4 (Matt Westcott)
- `update_index` management command now accepts a `--chunk_size` option to determine the number of items to load at once (Dave Bell)
- Added hook `register_account_menu_item` to add new account preference items (Michael van Tellingen)
- Added change email functionality from the account settings (Alejandro Garza, Alexs Mathilda)
- Add request parameter to edit handlers (Rajeev J Sebastian)
- ImageChooser now sets a default title based on filename (Coen van der Kamp)
- Added error handling to the Draftail editor (Thibaud Colas)
- Add new `wagtail_icon` template tag to facilitate making admin icons accessible (Sander Tuit)
- Set `ALLOWED_HOSTS` in the project template to allow any host in development (Tom Dyson)
- Expose reusable client-side code to build Draftail extensions (Thibaud Colas)
- Added `WAGTAILFRONTENDCACHE_LANGUAGES` setting to specify the languages whose URLs are to be purged when using `i18n_patterns` (PyMan Claudio Marinozzi)
- Added `extra_footer_actions` template blocks for customising the add/edit page views (Arthur Holzner)

### Bug fixes

- Status button on ‘edit page’ now links to the correct URL when live and draft slug differ (LB (Ben Johnston))
- Image title text in the gallery and in the chooser now wraps for long filenames (LB (Ben Johnston), Luiz Boaretto)
- Move image editor action buttons to the bottom of the form on mobile (Julian Gallo)
- StreamField icons are now correctly sorted into groups on the ‘append’ menu (Tim Heap)
- Draftail now supports features specified via the `WAGTAILADMIN_RICH_TEXT_EDITORS` setting (Todd Dembrey)
- Password reset form no longer indicates whether the email is recognised, as per standard Django behaviour (Bertrand Bordage)

- `UserAttributeSimilarityValidator` is now correctly enforced on user creation / editing forms (Tim Heap)
- Focal area removal not working in IE11 and MS Edge (Thibaud Colas)
- Rewrite password change feedback message to be more user-friendly (Casper Timmers)
- Correct dropdown arrow styling in Firefox, IE11 (Janneke Janssen, Alexs Mathilda)
- Password reset no indicates specific validation errors on certain password restrictions (Lucas Moeskops)
- Confirmation page on page deletion now respects custom `get_admin_display_title` methods (Kim Chee Leong)
- Adding external link with selected text now includes text in link chooser (Tony Yates, Thibaud Colas, Alexs Mathilda)
- Editing setting object with no site configured no longer crashes (Harm Zeinstra)
- Creating a new object with inlines while mandatory fields are empty no longer crashes (Bertrand Bordage)
- Localization of image and apps verbose names
- Draftail editor no longer crashes after deleting image/embed using DEL key (Thibaud Colas)
- Breadcrumb navigation now respects custom `get_admin_display_title` methods (Arthur Holzner, Wietze Helmantel, Matt Westcott)
- Inconsistent order of heading features when adding h1, h5 or h6 as default feature for Hallo RichText editor (Loic Teixeira)
- Add invalid password reset link error message (Coen van der Kamp)
- Bypass select/prefetch related optimisation on `update_index` for `ParentalManyToManyField` to fix crash (Tim Kamanin)
- ‘Add user’ is now rendered as a button due to the use of quotes within translations (Benoît Vogel)
- Menu icon no longer overlaps with title in Modeladmin on mobile (Coen van der Kamp)
- Background colour overflow within the Wagtail documentation (Sergey Fedoseev)
- Page count on homepage summary panel now takes account of user permissions (Andy Chosak)
- Explorer view now prevents navigating outside of the common ancestor of the user’s permissions (Andy Chosak)
- Generate URL for the current site when multiple sites share the same root page (Codie Roelf)
- Restored ability to use non-model fields with FieldPanel (Matt Westcott, LB (Ben Johnston))
- Stop revision comparison view from crashing when non-model FieldPanels are in use (LB (Ben Johnston))
- Ordering in the page explorer now respects custom `get_admin_display_title` methods when sorting <100 pages (Matt Westcott)
- Use index-specific Elasticsearch endpoints for bulk insertion, for compatibility with providers that lock down the root endpoint (Karl Hobley)
- Fix usage URL on the document edit page (Jérôme Lebleu)

## Upgrade considerations

### Image format `image_to_html` method has been updated

The internal API for rich text image format objects (see [Image Formats in the Rich Text Editor](#)) has been updated; the `Format.image_to_html` method now receives the `extra_attributes` keyword argument as a dictionary of attributes, rather than a string. If you have defined any custom format objects that override this method, these will need to be updated.

## 1.9.78 Wagtail 2.0.2 release notes

August 13, 2018

- [What's new](#)

### What's new

#### Bug fixes

- Restored ability to use non-model fields with FieldPanel (Matt Westcott, LB (Ben Johnston))
- Fix usage URL on the document edit page (Jérôme Lebleu)
- Pin Beautiful Soup to 4.6.0 due to further regressions in formatting empty elements (Matt Westcott)

## 1.9.79 Wagtail 2.0.1 release notes

April 4, 2018

- [What's new](#)

### What's new

- Added error notification when running the `wagtail` command on Python <3.4 (Matt Westcott)
- Added error handling to the Draftail editor (Thibaud Colas)

#### Bug fixes

- Draftail now supports features specified via the `WAGTAILADMIN_RICH_TEXT_EDITORS` setting (Todd Dembrey)
- Password reset form no longer indicates whether the email is recognised, as per standard Django behaviour (Bertrand Bordage)
- `UserAttributeSimilarityValidator` is now correctly enforced on user creation / editing forms (Tim Heap)
- Editing setting object with no site configured no longer crashes (Harm Zeinstra)
- Creating a new object with inlines while mandatory fields are empty no longer crashes (Bertrand Bordage)

## 1.9.80 Wagtail 2.0 release notes

February 28, 2018

- [What's new](#)
- [Upgrade considerations](#)

### What's new

#### Added Django 2.0 support

Wagtail is now compatible with Django 2.0. Compatibility fixes were contributed by Matt Westcott, Karl Hobley, LB (Ben Johnston) and Mads Jensen.

#### New rich text editor

Wagtail's rich text editor has now been replaced with [Draftail](#), a new editor based on [Draft.js](#), fixing numerous bugs and providing an improved editing experience, better support for current browsers, and more consistent HTML output. This feature was developed by Thibaud Colas, Loïc Teixeira and Matt Westcott.

#### Reorganised modules

The modules that make up Wagtail have been renamed and reorganised, to avoid the repetition in names like `wagtail.wagtailcore.models` (originally an artefact of app naming limitations in Django 1.6) and to improve consistency. While this will require some up-front work to upgrade existing Wagtail sites, we believe that this will be a long-term improvement to the developer experience, improving readability of code and reducing errors. This change was implemented by Karl Hobley and Matt Westcott.

#### Scheduled page revisions

The behaviour of scheduled publishing has been revised so that pages are no longer unpublished at the point of setting a future go-live date, making it possible to schedule updates to an existing published page. This feature was developed by Patrick Woods.

#### Other features

- Moved Wagtail API v1 implementation (`wagtail.contrib.api`) to an [external app](#) (Karl Hobley)
- The page chooser now searches all fields of a page, instead of just the title (Bertrand Bordage)
- Implement ordering by date in form submission view (LB (Ben Johnston))
- Elasticsearch scroll API is now used when fetching more than 100 search results (Karl Hobley)
- Added hidden field to the form builder (Ross Crawford-d'Heureuse)
- Usage count now shows on delete confirmation page when `WAGTAIL_USAGE_COUNT_ENABLED` is active (Kees Hink)
- Added usage count to snippets (Kees Hink)

- Moved usage count to the sidebar on the edit page (Kees Hink)
- Explorer menu now reflects customisations to the page listing made via the `construct_explorer_page_queryset` hook and `ModelAdmin.exclude_from_explorer` property (Tim Heap)
- “Choose another image” button changed to “Change image” to avoid ambiguity (Edd Baldry)
- Added hooks `before_create_user`, `after_create_user`, `before_delete_user`, `after_delete_user`, `before_edit_user`, `after_edit_user` (Jon Carmack)
- Added `exclude_fields_in_copy` property to Page to define fields that should not be included on page copy (LB (Ben Johnston))
- Improved error message on incorrect `{% image %}` tag syntax (LB (Ben Johnston))
- Optimized preview data storage (Bertrand Bordage)
- Added `render_landing_page` method to `AbstractForm` to be easily overridden and pass `form_submission` to landing page context (Stein Strindhaug)
- Added `heading` kwarg to `InlinePanel` to allow heading to be set independently of button label (Adrian Turjak)
- The value type returned from a `StructBlock` can now be customised. See [Additional methods and properties on StructBlock values](#) (LB (Ben Johnston))
- Added `bcolor` image operation (Karl Hobley)
- Added `WAGTAILADMIN_USER_LOGIN_FORM` setting for overriding the admin login form (Mike Dingjan)
- Snippets now support custom primary keys (Sævar Öfjörð Magnússon)
- Upgraded jQuery to version 3.2.1 (Janneke Janssen)
- Update autoprefixer configuration to better match browser support targets (Janneke Janssen)
- Update React and related dependencies to latest versions (Janneke Janssen, Hugo van den Berg)
- Remove Hallo editor `.richtext` CSS class in favour of more explicit extension points (Thibaud Colas)
- Updated documentation styling (LB (Ben Johnston))
- Rich text fields now take feature lists into account when whitelisting HTML elements (Matt Westcott)
- FormPage lists and Form submission lists in admin now use class based views for easy overriding (Johan Arensman)
- Form submission csv exports now have the export date in the filename and can be customised (Johan Arensman)
- FormBuilder class now uses bound methods for field generation, adding custom fields is now easier and documented (LB (Ben Johnston))
- Added `WAGTAILADMIN_NOTIFICATION_INCLUDE_SUPERUSERS` setting to determine whether superusers are included in moderation email notifications (Bruno Alla)
- Added a basic Dockerfile to the project template (Tom Dyson)
- StreamField blocks now allow custom `get_template` methods for overriding templates in instances (Christopher Bledsoe)
- Simplified edit handler API (Florent Osmont, Bertrand Bordage)
- Made ‘add/change/delete collection’ permissions configurable from the group edit page (Matt Westcott)
- Expose React-related dependencies as global variables for extension in the admin interface (Thibaud Colas)
- Added helper functions for constructing form data for use with `assertCanCreate`. See [Form data helpers](#) (Tim Heap, Matt Westcott)

## Bug fixes

- Do not remove stopwords when generating slugs from non-ASCII titles, to avoid issues with incorrect word boundaries (Sævar Öfjörð Magnússon)
- The PostgreSQL search backend now preserves ordering of the `QuerySet` when searching with `order_by_relevance=False` (Bertrand Bordage)
- Using `modeladmin_register` as a decorator no longer replaces the decorated class with `None` (Tim Heap)
- Fixed crash in XML sitemap generator when all pages on the site are private (Stein Strindhaug)
- The `{% routablepageurl %}` template tag no longer generates invalid URLs when the `WAGTAIL_APPEND_SLASH` setting was set to `False` (Venelin Stoykov)
- The “View live” button is no longer shown if the page doesn’t have a routable URL (Tim Heap)
- API listing views no longer fail when no site records are defined (Karl Hobley)
- Fixed rendering of border on dropdown arrow buttons on Chrome (Bertrand Bordage)
- Fixed incorrect z-index on userbar causing it to appear behind page content (Stein Strindhaug)
- Form submissions pagination no longer loses date filter when changing page (Bertrand Bordage)
- PostgreSQL search backend now removes duplicate page instances from the database (Bertrand Bordage)
- `FormSubmissionsPanel` now recognises custom form submission classes (LB (Ben Johnston))
- Prevent the footer and revisions link from unnecessarily collapsing on mobile (Jack Paine)
- Empty searches were activated when paginating through images and documents (LB (Ben Johnston))
- Summary numbers of pages, images and documents were not responsive when greater than 4 digits (Michael Palmer)
- Project template now has password validators enabled by default (Matt Westcott)
- Alignment options correctly removed from `TableBlock` context menu (LB (Ben Johnston))
- Fix support of `ATOMIC_REBUILD` for projects with Elasticsearch client `>=1.7.0` (Mikalai Radchuk)
- Fixed error on Elasticsearch backend when passing a `QuerySet` as an `__in` filter (Karl Hobley, Matt Westcott)
- `__isnull` filters no longer fail on Elasticsearch 5 (Karl Hobley)
- Prevented intermittent failures on Postgres search backend when a field is defined as both a `SearchField` and a `FilterField` (Matt Westcott)
- Alt text of images in rich text is no longer truncated on double-quote characters (Matt Westcott)
- Ampersands in embed URLs within rich text are no longer double-escaped (Matt Westcott)
- Using RGBA images no longer crashes with Pillow `>= 4.2.0` (Karl Hobley)
- Copying a page with PostgreSQL search enabled no longer crashes (Bertrand Bordage)
- Style of the page unlock button was broken (Bertrand Bordage)
- Admin search no longer floods browser history (Bertrand Bordage)
- Version comparison now handles custom primary keys on inline models correctly (LB (Ben Johnston))
- Fixed error when inserting chooser panels into `FieldRowPanel` (Florent Osmont, Bertrand Bordage)
- Reinstated missing error reporting on image upload (Matt Westcott)
- Only load Hallo CSS if Hallo is in use (Thibaud Colas)

- Prevent style leak of Wagtail panel icons in widgets using h2 elements (Thibaud Colas)

## Upgrade considerations

### Removed support for Python 2.7, Django 1.8 and Django 1.10

Python 2.7, Django 1.8 and Django 1.10 are no longer supported in this release. You are advised to upgrade your project to Python 3 and Django 1.11 before upgrading to Wagtail 2.0.

### Added support for Django 2.0

Before upgrading to Django 2.0, you are advised to review the [release notes](#), especially the [backwards incompatible changes](#) and [removed features](#).

### Wagtail module path updates

Many of the module paths within Wagtail have been reorganised to reduce duplication - for example, `wagtail.wagtailcore.models` is now `wagtail.core.models`. As a result, import lines and other references to Wagtail modules will need to be updated when you upgrade to Wagtail 2.0. A new command has been added to assist with this - from the root of your project's code base:

```
$ wagtail updatemodulepaths --list # list the files to be changed without updating them
$ wagtail updatemodulepaths --diff # show the changes to be made, without updating files
$ wagtail updatemodulepaths # actually update the files
```

Or, to run from a different location:

```
$ wagtail updatemodulepaths /path/to/project --list
$ wagtail updatemodulepaths /path/to/project --diff
$ wagtail updatemodulepaths /path/to/project
```

For the full list of command line options, enter `wagtail help updatemodulepaths`.

You are advised to take a backup of your project codebase before running this command. The command will perform a search-and-replace over all \*.py files for the affected module paths; while this should catch the vast majority of module references, it will not be able to fix instances that do not use the dotted path directly, such as `from wagtail import wagtailcore`.

The full list of modules to be renamed is as follows:

Old name	New name	Notes
wagtail.wagtailcore	wagtail.core	
wagtail.wagtailadmin	wagtail.admin	
wagtail.wagtailedocs	wagtail.documents	'documents' no longer abbreviated
wagtail.wagtailembeds	wagtail.embeds	
wagtail.wagtailimages	wagtail.images	
wagtail.wagtailsearch	wagtail.search	
wagtail.wagtailsites	wagtail.sites	
wagtail.wagtailsnippets	wagtail.snippets	
wagtail.wagtailusers	wagtail.users	
wagtail.wagtailforms	wagtail.contrib.forms	Moved into 'contrib'
wagtail.wagtailredirects	wagtail.contrib.redirects	Moved into 'contrib'
wagtail.contrib.wagtailapi	<i>removed</i>	API v1, removed in this release
wagtail.contrib.wagtailfrontendcache	wagtail.contrib.frontend_cache	Underscore added
wagtail.contrib.wagtailroutablepage	wagtail.contrib.routable_page	Underscore added
wagtail.contrib.wagtailsearchpromotions	wagtail.contrib.search_promotions	Underscore added
wagtail.contrib.wagtaileitemaps	wagtail.contrib.sitemaps	
wagtail.contrib.wagtailstyleguide	wagtail.contrib.styleguide	

Places these should be updated include:

- `import` lines
- Paths specified in settings, such as `INSTALLED_APPS`, `MIDDLEWARE` and `WAGTAILSEARCH_BACKENDS`
- Fields and blocks referenced within migrations, such as `wagtail.wagtailcore.fields.StreamField` and `wagtail.wagtailcore.blocks.RichTextBlock`

However, note that this only applies to dotted module paths beginning with `wagtail..`. App names that are *not* part of a dotted module path should be left unchanged - in this case, the `wagtail` prefix is still required to avoid clashing with other apps that might exist in the project with names such as `admin` or `images`. The following should be left unchanged:

- Foreign keys specifying a model as '`app_name.ModelName`', e.g. `models.ForeignKey('wagtailimages.Image', ...)`
- App labels used in database table names, content types or permissions
- Paths to templates and static files, e.g. when *overriding admin templates with custom branding*
- Template tag library names, e.g. `{% load wagtailcore_tags %}`

### Hallo.js customisations are unavailable on the Draftail rich text editor

The Draftail rich text editor has a substantially different API from Hallo.js, including the use of a non-HTML format for its internal data representation; as a result, functionality added through Hallo.js plugins will be unavailable. If your project is dependent on Hallo.js-specific behaviour, you can revert to the original Hallo-based editor by adding the following to your settings:

```
WAGTAILADMIN_RICH_TEXT_EDITORS = {
 'default': {
 'WIDGET': 'wagtail.admin.rich_text.HalloRichTextArea'
```

(continues on next page)

(continued from previous page)

```
 }
}
```

### Data format for rich text fields in assertCanCreate tests has been updated

The `assertCanCreate` test method (see [Testing your Wagtail site](#)) requires data to be passed in the same format that the page edit form would submit. The Draftail rich text editor posts this data in a non-HTML format, and so any existing `assertCanCreate` tests involving rich text fields will fail when Draftail is in use:

```
self.assertCanCreate(root_page, ContentPage, {
 'title': 'About us',
 'body': '<p>Lorem ipsum dolor sit amet</p>', # will not work
})
```

Wagtail now provides a set of helper functions for constructing form data: see [Form data helpers](#). The above assertion can now be rewritten as:

```
from wagtail.tests.utils.form_data import rich_text

self.assertCanCreate(root_page, ContentPage, {
 'title': 'About us',
 'body': rich_text('<p>Lorem ipsum dolor sit amet</p>'),
})
```

### Removed support for Elasticsearch 1.x

Elasticsearch 1.x is no longer supported in this release. Please upgrade to a 2.x or 5.x release of Elasticsearch before upgrading to Wagtail 2.0.

### Removed version 1 of the Wagtail API

Version 1 of the Wagtail API (`wagtail.contrib.wagtailapi`) has been removed from Wagtail.

If you're using version 1, you will need to migrate to version 2. Please see [Wagtail API v2 Configuration Guide](#) and [Wagtail API v2 Usage Guide](#).

If migrating to version 2 is not an option right now (if you have API clients that you don't have direct control over, such as a mobile app), you can find the implementation of the version 1 API in the new `wagtailapi_legacy` repository.

This repository has been created to provide a place for the community to collaborate on supporting legacy versions of the API until everyone has migrated to an officially supported version.

### `construct_whitelister_element_rules` hook is deprecated

The `construct_whitelister_element_rules` hook, used to specify additional HTML elements to be permitted in rich text, is deprecated. The recommended way of whitelisting elements is now to use rich text features. For example, a whitelist rule that was previously defined as:

```
from wagtail.core import hooks
from wagtail.core.whitelist import allow_without_attributes

@hooks.register('construct_whitelister_element_rules')
def whitelist_blockquote():
 return {
 'blockquote': allow_without_attributes,
 }
```

can be rewritten as:

```
from wagtail.admin.rich_text.converters.editor_html import WhitelistRule
from wagtail.core import hooks
from wagtail.core.whitelist import allow_without_attributes

@hooks.register('register_rich_text_features')
def blockquote_feature(features):

 # register a feature 'blockquote' which whitelists the <blockquote> element
 features.register_converter_rule('editorhtml', 'blockquote', [
 WhitelistRule('blockquote', allow_without_attributes),
])

 # add 'blockquote' to the default feature set
 features.default_features.append('blockquote')
```

Please note that the new Draftail rich text editor uses a different mechanism to process rich text content, and does not apply whitelist rules; they only take effect when the Hallo.js editor is in use.

### `wagtail.images.views.serve.generate_signature` now returns a string

The `generate_signature` function in `wagtail.images.views.serve`, used to build URLs for the *dynamic image serve view*, now returns a string rather than a byte string. This ensures that any existing user code that builds up the final image URL with `reverse` will continue to work on Django 2.0 (which no longer allows byte strings to be passed to `reverse`). Any code that expects a byte string as the return value of `generate_string` - for example, calling `decode()` on the result - will need to be updated. (Apps that need to preserve compatibility with earlier versions of Wagtail can call `django.utils.encoding.force_text` instead of `decode`.)

## Deprecated search view

Wagtail has always included a bundled view for frontend search. However, this view isn't easy to customise so defining this view per project is usually preferred. If you have used this bundled view (check for an import from `wagtail.wagtailsearch.urls` in your project's `urls.py`), you will need to replace this with your own implementation.

See the search view in Wagtail demo for a guide: <https://github.com/wagtail/wagtaildemo/blob/master/demo/views.py>

## New Hallo editor extension points

With the introduction of a new editor, we want to make sure existing editor plugins meant for Hallo only target Hallo editors for extension.

- The existing `.richtext` CSS class is no longer applied to the Hallo editor's DOM element.
- In JavaScript, use the `[data-hallo-editor]` attribute selector to target the editor, eg. `var $editor = $('[data-hallo-editor]');`.
- In CSS, use the `.halloeditor` class selector.

For example,

```
/* JS */
- var widget = $(elem).parent('.richtext').data('IKS-hallo');
+ var widget = $(elem).parent('[data-hallo-editor]').data('IKS-hallo');

[...]

/* Styles */
- .richtext {
+ .halloeditor {
 font-family: monospace;
}
```

## 1.9.81 Wagtail 1.13.4 release notes

August 13, 2018

- *What's new*

### What's new

### Bug fixes

- Pin BeautifulSoup to 4.6.0 due to further regressions in formatting empty elements (Matt Westcott)

## 1.9.82 Wagtail 1.13.3 release notes

August 13, 2018

- *What's new*

### What's new

### Bug fixes

- Pin django-taggit to <0.23 to restore Django 1.8 compatibility (Matt Westcott)
- Mark Beautiful Soup 4.6.1 as incompatible due to bug in formatting empty elements (Matt Westcott)

## 1.9.83 Wagtail 1.13.2 release notes

July 4, 2018

- *What's new*

### What's new

### Bug fixes

- Fix support of ATOMIC\_REBUILD for projects with Elasticsearch client >=1.7.0 (Mikalai Radchuk)
- Logging an indexing failure on an object with a non-ASCII representation no longer crashes on Python 2 (Aram Dulyan)
- Rich text image chooser no longer skips format selection after a validation error (Matt Westcott)
- Null characters in URLs no longer crash the redirect middleware on PostgreSQL (Matt Westcott)

## 1.9.84 Wagtail 1.13.1 release notes

November 17, 2017

- *What's new*

## What's new

### Bug fixes

- API listing views no longer fail when no site records are defined (Karl Hobley)
- Fixed crash in XML sitemap generator when all pages on the site are private (Stein Strindhaug)
- Fixed incorrect z-index on userbar causing it to appear behind page content (Stein Strindhaug)
- Fixed error in Postgres search backend when searching specific fields of a `specific()` Page QuerySet (Bertrand Bordage, Matt Westcott)
- Fixed error on Elasticsearch backend when passing a QuerySet as an `__in` filter (Karl Hobley, Matt Westcott)
- `__isnull` filters no longer fail on Elasticsearch 5 (Karl Hobley)
- Prevented intermittent failures on Postgres search backend when a field is defined as both a `SearchField` and a `FilterField` (Matt Westcott)

## 1.9.85 Wagtail 1.13 release notes

October 16, 2017

- *What's new*

Wagtail 1.13 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 8 months). Please note that Wagtail 1.13 will be the last LTS release to support Python 2.

## What's new

### New features

- Front-end cache invalidator now supports purging URLs as a batch - see [InValidating URLs](#) (Karl Hobley)
- *Custom document model* is now documented (Emily Horsman)
- Use minified versions of CSS in the admin by adding minification to the front-end tooling (Vincent Audebert, Thibaud Colas)
- Wagtailforms serve view now passes `request.FILES`, for use in custom form handlers (LB (Ben Johnston))
- Documents and images are now given new filenames on re-uploading, to avoid old versions being kept in cache (Bertrand Bordage)
- Added custom 404 page for admin interface (Jack Paine)
- Breadcrumb navigation now uses globe icon to indicate tree root, rather than home icon (Matt Westcott)
- Wagtail now uses React 15.6.2 and above, released under the MIT license (Janneke Janssen)
- User search in the Wagtail admin UI now works across multiple fields (Will Giddens)
- `Page.last_published_at` is now a filterable field for search (Mikalai Radchuk)
- Page search results and usage listings now include navigation links (Matt Westcott)

## Bug fixes

- “Open Link in New Tab” on a right arrow in page explorer should open page list (Emily Horsman)
- Using `order_by_relevance=False` when searching with PostgreSQL now works (Mitchel Cabuloy)
- Inline panel first and last sorting arrows correctly hidden in non-default tabs (Matt Westcott)
- `WAGTAILAPI_LIMIT_MAX` now accepts `None` to disable limiting (jcrony)
- In PostgreSQL, new default ordering when ranking of objects is the same (Bertrand Bordage)
- Fixed overlapping header elements on form submissions view on mobile (Jack Paine)
- Fixed avatar position in footer on mobile (Jack Paine)
- Custom document models no longer require their own post-delete signal handler (Gordon Pendleton)
- Deletion of image / document files now only happens when database transaction has completed (Gordon Pendleton)
- Fixed Node build scripts to work on Windows (Mikalai Radchuk)
- Stop breadcrumb home icon from showing as ellipsis in Chrome 60 (Matt Westcott)
- Prevent `USE_THOUSAND_SEPARATOR = True` from breaking the image focal point chooser (Sævar Öfjörð Magússon)
- Removed deprecated `SessionAuthenticationMiddleware` from project template (Samir Shah)
- Custom display page titles defined with `get_admin_display_title` are now shown in search results (Ben Sturmels, Matt Westcott)
- Custom PageManagers now return the correct `PageQuerySet` subclass (Matt Westcott)

## 1.9.86 Wagtail 1.12.6 release notes

August 13, 2018

- [What's new](#)

### What's new

## Bug fixes

- Pin BeautifulSoup to 4.6.0 due to further regressions in formatting empty elements (Matt Westcott)

## 1.9.87 Wagtail 1.12.5 release notes

August 13, 2018

- *What's new*

### What's new

### Bug fixes

- Pin django-taggit to <0.23 to restore Django 1.8 compatibility (Matt Westcott)
- Mark Beautiful Soup 4.6.1 as incompatible due to bug in formatting empty elements (Matt Westcott)

## 1.9.88 Wagtail 1.12.4 release notes

July 4, 2018

- *What's new*

### What's new

### Bug fixes

- Fix support of ATOMIC\_REBUILD for projects with Elasticsearch client >=1.7.0 (Mikalai Radchuk)
- Logging an indexing failure on an object with a non-ASCII representation no longer crashes on Python 2 (Aram Dulyan)
- Rich text image chooser no longer skips format selection after a validation error (Matt Westcott)
- Null characters in URLs no longer crash the redirect middleware on PostgreSQL (Andrew Crewdson, Matt Westcott)

## 1.9.89 Wagtail 1.12.3 release notes

November 17, 2017

- *What's new*

## What's new

### Bug fixes

- API listing views no longer fail when no site records are defined (Karl Hobley)
- Pinned Django REST Framework to <3.7 to restore Django 1.8 compatibility (Matt Westcott)
- Fixed crash in XML sitemap generator when all pages on the site are private (Stein Strindhaug)
- Fixed error in Postgres search backend when searching specific fields of a `specific()` Page QuerySet (Bertrand Bordage, Matt Westcott)
- Fixed error on Elasticsearch backend when passing a QuerySet as an `__in` filter (Karl Hobley, Matt Westcott)
- `__isnull` filters no longer fail on Elasticsearch 5 (Karl Hobley)
- Prevented intermittent failures on Postgres search backend when a field is defined as both a `SearchField` and a `FilterField` (Matt Westcott)

## 1.9.90 Wagtail 1.12.2 release notes

*September 18, 2017*

- *What's new*

## What's new

### Bug fixes

- Migration for addition of `Page.draft_title` field is now reversible (Venelin Stoykov)
- Fixed failure on application startup when `ManifestStaticFilesStorage` is in use and `collectstatic` has not yet been run (Matt Westcott)
- Fixed handling of Vimeo and other oEmbed providers with a `format` parameter in the endpoint URL (Mitchel Cabuloy)
- Fixed regression in rendering save button in `wagtail.contrib.settings` edit view (Matt Westcott)

## 1.9.91 Wagtail 1.12.1 release notes

*August 30, 2017*

- *What's new*

## What's new

### Bug fixes

- Prevent home page draft title from displaying as blank (Mikalai Radchuk, Matt Westcott)
- Fix regression on styling of preview button with more than one preview mode (Jack Paine)
- Enabled translations within date-time chooser widget (Lucas Moeskops)

## 1.9.92 Wagtail 1.12 release notes

August 21, 2017

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

Wagtail 1.12 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 8 months).

## What's new

### Configurable rich text features

The feature set provided by the rich text editor can now be configured on a per-field basis, by passing a `features` keyword argument; for example, a field can be configured to allow bold / italic formatting and links, but not headings or embedded images or media. For further information, see [Limiting features in a rich text field](#). This feature was developed by Matt Westcott.

### Improved embed configuration

New configuration options for embedded media have been added, to give greater control over how media URLs are converted to embeds, and to make it possible to specify additional media providers beyond the ones built in to Wagtail. For further information, see [Embedded content](#). This feature was developed by Karl Hobley.

## Other features

- The admin interface now displays a title of the latest draft (Mikalai Radchuk)
- `RoutablePageMixin` now has a default “index” route (Andreas Nüßlein, Matt Westcott)
- Added multi-select form field to the form builder (dwasy)
- Improved performance of sitemap generation (Levi Adler)
- `StreamField` now respects the `blank` setting; `StreamBlock` accepts a `required` setting (Loic Teixeira)
- `StreamBlock` now accepts `min_num`, `max_num` and `block_counts` settings to control the minimum and maximum numbers of blocks (Edwar Baron, Matt Westcott)

- Users can no longer remove their own active / superuser flags through Settings -> Users (Stein Strindhaug, Huub Bouma)
- The `process_form_submission` method of form pages now return the created form submission object (Christine Ho)
- Added `WAGTAILUSERS_PASSWORD_ENABLED` and `WAGTAILUSERS_PASSWORD_REQUIRED` settings to permit creating users with no Django-side passwords, to support external authentication setups (Matt Westcott)
- Added help text parameter to `DecimalBlock` and `RegexBlock` (Tomasz Knapik)
- Optimised caudal oscillation parameters on logo (Jack Paine)

## Bug fixes

- FieldBlocks in StreamField now call the field's `prepare_value` method (Tim Heap)
- Initial disabled state of InlinePanel add button is now set correctly on non-default tabs (Matthew Downey)
- Redirects with unicode characters now work (Rich Brennan)
- Prevent explorer view from crashing when page model definitions are missing, allowing the offending pages to be deleted (Matt Westcott)
- Hide the userbar from printed page representation (Eugene Morozov)
- Prevent the page editor footer content from collapsing into two lines unnecessarily (Jack Paine)
- StructBlock values no longer render HTML templates as their `str` representation, to prevent infinite loops in debugging / logging tools (Matt Westcott)
- Removed deprecated jQuery `load` call from TableBlock initialisation (Jack Paine)
- Position of options in mobile nav-menu (Jack Paine)
- Center page editor footer regardless of screen width (Jack Paine)
- Change the design of the navbar toggle icon so that it no longer obstructs page headers (Jack Paine)
- Document add/edit forms no longer render container elements for hidden fields (Jeffrey Chau)

## Upgrade considerations

### StreamField now defaults to `blank=False`

StreamField now respects the `blank` field setting; when this is false, at least one block must be supplied for the field to pass validation. To match the behaviour of other model fields, `blank` defaults to `False`; if you wish to allow a StreamField to be left empty, you must now add `blank=True` to the field.

When passing an explicit `StreamBlock` as the top-level block of a StreamField definition, note that the StreamField's `blank` keyword argument always takes precedence over the block's `required` property, including when it is left as the default value of `blank=False`. Consequently, setting `required=False` on a top-level `StreamBlock` has no effect.

### Old configuration settings for embeds are deprecated

The configuration settings `WAGTAILEMBEDS_EMBED_FINDER` and `WAGTAILEMBEDS_EMBEDLY_KEY` have been deprecated in favour of the new `WAGTAILEMBEDS_FINDERS` setting. Please see [Configuring embed “finders”](#) for the new configuration to use.

### Registering custom hallo.js plugins directly is deprecated

The ability to enable / disable `hallo.js` plugins by calling `registerHalloPlugin` or modifying the `halloPlugins` list has been deprecated, and will be removed in Wagtail 1.14. The recommended way of customising the `hallo.js` editor is now through [rich text features](#).

### Custom `get_admin_display_title` methods should use `draft_title`

This release introduces a new `draft_title` field on page models, so that page titles as used across the admin interface will correctly reflect any changes that exist in draft. If any of your page models override the `get_admin_display_title` method, to customise the display of page titles in the admin, it is recommended that you now update these to base their output on `draft_title` rather than `title`. Alternatively, to preserve backwards compatibility, you can invoke `super` on the method, for example:

```
def get_admin_display_title(self):
 return "%(title)s (%(lang)s)" % {
 'title': super(TranslatablePage, self).get_admin_display_title(),
 'lang': self.language_code,
 }
```

### Fixtures for loading pages should include `draft_title`

In most situations, the new `draft_title` field on page models will automatically be populated from the page title. However, this is not the case for pages that are created from fixtures. Projects that use fixtures to load initial data should therefore ensure that a `draft_title` field is specified.

### RoutablePageMixin now has a default index route

If you've used `RoutablePageMixin` on a Page model, you may have had to manually define an index route to serve the page at its main URL (`r'^$'`) so it behaves like a normal page. Wagtail now defines a default index route so this is no longer required.

## 1.9.93 Wagtail 1.11.1 release notes

*July 7, 2017*

- *What's new*

## What's new

### Bug fixes

- Custom display page titles defined with `get_admin_display_title` are now shown within the page explorer menu (Matt Westcott, Janneke Janssen)

## 1.9.94 Wagtail 1.11 release notes

June 30, 2017

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

## What's new

### Explorer menu built with the admin API and React

After more than a year of work, the new explorer menu has finally landed! It comes with the following improvements:

- View all pages - not just the ones with child pages.
- Better performance, no matter the number of pages or levels in the hierarchy.
- Navigate the menu via keyboard.
- View Draft pages, and go to page editing, directly from the menu.

Beyond features, the explorer is built with the new admin API and React components. This will facilitate further evolutions to make it even faster and user-friendly. This work is the product of 4 Wagtail sprints, and the efforts of 16 people, listed here by order of first involvement:

- Karl Hobley (Cape town sprint, admin API)
- Josh Barr (Cape town sprint, prototype UI)
- Thibaud Colas (Ede sprint, Reykjavík sprint)
- Janneke Janssen (Ede sprint, Reykjavík sprint, Wagtail Space sprint)
- Rob Moorman (Ede sprint, eslint-config-wagtail, ES6+React+Redux styleguide)
- Maurice Bartrnig (Ede sprint, i18n and bug fixes)
- Jonny Scholes (code review)
- Matt Westcott (Reykjavík sprint, refactorings)
- Sævar Ófjörð Magnússon (Reykjavík sprint)
- Eirikur Ingi Magnusson (Reykjavík sprint)
- Harris Lapiroff (Reykjavík sprint, tab-accessible navigation)
- Hugo van den Berg (testing, Wagtail Space sprint)
- Olly Willans (UI, UX, Wagtail Space sprint)
- Andy Babic (UI, UX)

- Ben Enright (UI, UX)
- Bertrand Bordage (testing, documentation)

### Privacy settings on documents

Privacy settings can now be configured on collections, to restrict access to documents either by shared password or by user account. See: [Private pages](#).

This feature was developed by Ulrich Wagner and Matt Westcott. Thank you to [Wharton Research Data Services](#) of [The Wharton School](#) for sponsoring this feature.

### Other features

- Optimised page URL generation by caching site paths in the request scope (Tobias McNulty, Matt Westcott)
- The current live version of a page is now tracked on the revision listing view (Matheus Bratfisch)
- Each block created in a `StreamField` is now assigned a globally unique identifier (Matt Westcott)
- Mixcloud oEmbed pattern has been updated (Alice Rose)
- Added `last_published_at` field to the Page model (Matt Westcott)
- Added `show_in_menus_default` flag on page models, to allow “show in menus” to be checked by default (LB (Ben Johnston))
- “Copy page” form now validates against copying to a destination where the user does not have permission (Henk-Jan van Hasselaar)
- Allows reverse relations in `RelatedFields` for Elasticsearch & PostgreSQL search backends (Lucas Moeskops, Bertrand Bordage)
- Added oEmbed support for Facebook (Mikalai Radchuk)
- Added oEmbed support for Tumblr (Mikalai Radchuk)

### Bug fixes

- Unauthenticated AJAX requests to admin views now return 403 rather than redirecting to the login page (Karl Hobley)
- `TableBlock` options `afterChange`, `afterCreateCol`, `afterCreateRow`, `afterRemoveCol`, `afterRemoveRow` and `contextMenu` can now be overridden (Loic Teixeira)
- The `lastmod` field returned by `wagtailsitemaps` now shows the last published date rather than the date of the last draft edit (Matt Westcott)
- Document chooser upload form no longer renders container elements for hidden fields (Jeffrey Chau)
- Prevented exception when visiting a preview URL without initiating the preview (Paul Kamp)

## Upgrade considerations

### Browser requirements for the new explorer menu

The new explorer menu does not support IE8, IE9, and IE10. The fallback experience is a link pointing to the explorer pages.

### Caching of site-level URL information throughout the request cycle

The `get_url_parts` and `relative_url` methods on `Page` now accept an optional `request` keyword argument. Additionally, two new methods have been added, `get_url` (analogous to the `url` property) and `get_full_url` (analogous to the `full_url`) property. Whenever possible, these methods should be used instead of the property versions, and the `request` passed to each method. For example:

```
page_url = my_page.url
```

would become:

```
page_url = my_page.get_url(request=request)
```

This enables caching of underlying site-level URL information throughout the request cycle, thereby significantly reducing the number of cache or SQL queries your site will generate for a given page load. A common use case for these methods is any custom template tag your project may include for generating navigation menus. For more information, please refer to [Page URLs](#).

Furthermore, if you have overridden `get_url_parts` or `relative_url` on any of your page models, you will need to update the method signature to support this keyword argument; most likely, this will involve changing the line:

```
def get_url_parts(self):
```

to:

```
def get_url_parts(self, *args, **kwargs):
```

and passing those through at the point where you are calling `get_url_parts` on `super` (if applicable).

See also: `wagtail.core.models.Page.get_url_parts()`, `wagtail.core.models.Page.get_url()`, `wagtail.core.models.Page.get_full_url()`, and `wagtail.core.models.Page.relative_url()`

### “Password required” template for documents

This release adds the ability to password-protect documents as well as pages. The template used for the “password required” form is distinct from the one used for pages; if you have previously overridden the default template through the `PASSWORD_REQUIRED_TEMPLATE` setting, you may wish to provide a corresponding template for documents through the setting `DOCUMENT_PASSWORD_REQUIRED_TEMPLATE`. See: [Private pages](#)

## **Elasticsearch 5.4 is incompatible with ATOMIC\_REBUILD**

While not specific to Wagtail 1.11, users of Elasticsearch should be aware that the ATOMIC\_REBUILD option is not compatible with Elasticsearch 5.4.x due to a bug in the handling of aliases. If you wish to use this feature, please use Elasticsearch 5.3.x or 5.5.x (when available).

## **1.9.95 Wagtail 1.10.1 release notes**

*May 19, 2017*

- *What's changed*

### **What's changed**

#### **Bug fixes**

- Fix admin page preview that was broken 24 hours after previewing a page (Martin Hill)
- Removed territory-specific translations for Spanish, Polish, Swedish, Russian and Chinese (Taiwan) that block more complete translations from being used (Matt Westcott)

## **1.9.96 Wagtail 1.10 release notes**

*May 3, 2017*

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

### **What's new**

#### **PostgreSQL search engine**

A new search engine has been added to Wagtail which uses PostgreSQL's built-in full-text search functionality. This means that if you use PostgreSQL to manage your database, you can now get a good quality search engine without needing to install Elasticsearch.

This feature was developed at the Arnhem sprint by Bertrand Bordage, Jaap Roes, Arne de Laat and Ramon de Jezus.

## Django 1.11 and Python 3.6 support

Wagtail is now compatible with Django 1.11 and Python 3.6. Compatibility fixes were contributed by Tim Graham, Matt Westcott, Mikalai Radchuk and Bertrand Bordage.

## User language preference

Users can now set their preferred language for the Wagtail admin interface under Account Settings → Language Preferences. The list of available languages can be configured via the `WAGTAILADMIN_PERMITTED_LANGUAGES` setting. This feature was developed by Daniel Chimeno.

## New admin preview

Previewing pages in Wagtail admin interface was rewritten to make it more robust. In previous versions, preview was broken in several scenarios so users often ended up on a blank page with an infinite spinner.

An additional setting was created: `WAGTAIL_AUTO_UPDATE_PREVIEW`. It allows users to see changes done in the editor by refreshing the preview tab without having to click again on the preview button.

This was developed by Bertrand Bordage.

## Other features

- Use minified versions of jQuery and jQuery UI in the admin. Total savings without compression 371 KB (Tom Dyson)
- Hooks can now specify the order in which they are run (Gagaro)
- Added a `submit_buttons` block to login template (Gagaro)
- Added `construct_image_chooser_queryset`, `construct_document_chooser_queryset` and `construct_page_chooser_queryset` hooks (Gagaro)
- The homepage created in the project template is now titled “Home” rather than “Homepage” (Karl Hobley)
- Signal receivers for custom `Image` and `Rendition` models are connected automatically (Mike Dingjan)
- `PageChooserBlock` can now accept a list/tuple of page models as `target_model` (Mikalai Radchuk)
- Styling tweaks for the `ModelAdmin`’s `IndexView` to be more inline with the Wagtail styleguide (Andy Babic)
- Added `.nvmrc` to the project root for Node versioning support (Janneke Janssen)
- Added `form_fields_exclude` property to `ModelAdmin` views (Matheus Bratfisch)
- User creation / edit form now enforces password validators set in `AUTH_PASSWORD_VALIDATORS` (Bertrand Bordage)
- Added support for displaying `non_field_errors` when validation fails in the page editor (Matt Westcott)
- Added `WAGTAILADMIN_RECENT_EDITS_LIMIT` setting to define the number of your most recent edits on the dashboard (Maarten Kling)
- Added link to the full Elasticsearch setup documentation from the Performance page (Matt Westcott)
- Tag input fields now accept spaces in tags by default, and can be overridden with the `TAG_SPACES_ALLOWED` setting (Kees Hink, Alex Gleason)
- Page chooser widgets now display the required page type where relevant (Christine Ho)

- Site root pages are now indicated with a globe icon in the explorer listing (Nick Smith, Huub Bouma)
- Draft page view is now restricted to users with edit / publish permission over the page (Kees Hink)
- Added the option to delete a previously saved focal point on a image (Maarten Kling)
- Page explorer menu item, search and summary panel are now hidden for users with no page permissions (Tim Heap)
- Added support for custom date and datetime formats in input fields (Bojan Mihelac)
- Added support for custom Django REST framework serialiser fields in `Page.api_fields` using a new `APIField` class (Karl Hobley)
- Added `classname` argument to `StreamFieldPanel` (Christine Ho)
- Added `group` keyword argument to StreamField blocks for grouping related blocks together in the block menu (Andreas Nüßlein)
- Update the sitemap generator to use the Django sitemap module (Michael van Tellingen, Mike Dingjan)

### Bug fixes

- Marked ‘Date from’ / ‘Date to’ strings in wagtailforms for translation (Vorlif)
- “File” field label on image edit form is now translated (Stein Strindhaug)
- Unreliable preview is now reliable by always opening in a new window (Kjartan Sverrisson)
- Fixed placement of `{{ block.super }}` in `snippets/type_index.html` (LB (Ben Johnston))
- Optimised database queries on group edit page (Ashia Zawaduk)
- Choosing a popular search term for promoted search results now works correctly after pagination (Janneke Janssen)
- IDs used in tabbed interfaces are now namespaced to avoid collisions with other page elements (Janneke Janssen)
- Page title not displaying page name when moving a page (Trent Holliday)
- The ModelAdmin module can now work without the wagtailimages and wagtaildocs apps installed (Andy Babic)
- Cloudflare error handling now handles non-string error responses correctly (hdnpl)
- Search indexing now uses a defined query ordering to prevent objects from being skipped (Christian Peters)
- Ensure that number localisation is not applied to object IDs within admin templates (Tom Hendrikx)
- Paginating with a search present was always returning the 1st page in Internet Explorer 10 & 11 (Ralph Jacobs)
- RoutablePageMixin and wagtailforms previews now set the `request.is_preview` flag (Wietze Helmantel)
- The save and preview buttons in the page editor are now mobile-friendly (Maarten Kling)
- Page links within rich text now respect custom URLs defined on specific page models (Gary Krige, Huub Bouma)
- Default avatar no longer visible when using a transparent gravatar image (Thijs Kramer)
- Scrolling within the datetime picker is now usable again for touchpads (Ralph Jacobs)
- List-based fields within form builder form submissions are now displayed as comma-separated strings rather than as Python lists (Christine Ho, Matt Westcott)
- The page type usage listing now have a translatable page title (Ramon de Jezus)
- Styles for submission filtering form now have a consistent height. (Thijs Kramer)

- Slicing a search result set no longer loses the annotation added by `annotate_score` (Karl Hobley)
- String-based primary keys are now escaped correctly in ModelAdmin URLs (Andreas Nüßlein)
- Empty search in the API now works (Morgan Aubert)
- RichTextBlock toolbar now correctly positioned within StructBlock (Janneke Janssen)
- Fixed display of ManyToMany fields and False values on the ModelAdmin inspect view (Andy Babic)
- Prevent pages from being recursively copied into themselves (Matheus Bratfisch)
- Specifying the full file name in documents URL is mandatory (Morgan Aubert)
- Reordering inline forms now works correctly when moving past a deleted form (Janneke Janssen)
- Removed erroneous `|safe` filter from search results template in project template (Karl Hobley)

## Upgrade considerations

### Django 1.9 and Python 3.3 support dropped

Support for Django 1.9 and Python 3.3 has been dropped in this release; please upgrade from these before upgrading Wagtail. Note that the Django 1.8 release series is still supported, as a Long Term Support release.

### Dropped support for generating static sites using django-medusa

Django-medusa is no longer maintained, and is incompatible with Django 1.8 and above. An alternative module based on the `django-bakery` package is available as a third-party contribution: <https://github.com/moorinteractive/wagtail-bakery>.

### Signals on custom Image and Rendition models connected automatically

Projects using `custom image models` no longer need to set up signal receivers to handle deletion of image files and image feature detection, as these are now handled automatically by Wagtail. The following lines of code should be removed:

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

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

Perform image feature detection (if enabled)
@receiver(pre_save, sender=CustomImage)
def image_feature_detection(sender, instance, **kwargs):
 if not instance.has_focal_point():
 instance.set_focal_point(instance.get_suggested_focal_point())
```

## Adding / editing users through Wagtail admin no longer sets `is_staff` flag

Previously, the `is_staff` flag (which grants access to the Django admin interface) was automatically set for superusers, and reset for other users, when creating and updating users through the Wagtail admin. This behaviour has now been removed, since Wagtail is designed to work independently of the Django admin. If you need to reinstate the old behaviour, you can set up a [pre\\_save signal handler](#) on the User model to set the flag appropriately.

## Specifying the full file name in documents URL is mandatory

In previous releases, it was possible to download a document using the primary key and a fraction of its file name, or even without file name. You could get the same document at the addresses `/documents/1/your-file-name.pdf`, `/documents/1/you & /documents/1/`.

This feature was supposed to allow shorter URLs but was not used in Wagtail. For security reasons, we removed it, so only the full URL works: `/documents/1/your-file-name.pdf`

If any of your applications relied on the previous behaviour, you will have to rewrite it to take this into account.

## 1.9.97 Wagtail 1.9.1 release notes

April 21, 2017

- [\*What's changed\*](#)

### What's changed

### Bug fixes

- Removed erroneous `|safe` filter from search results template in project template (Karl Hobley)
- Prevent pages from being recursively copied into themselves (Matheus Bratfisch)

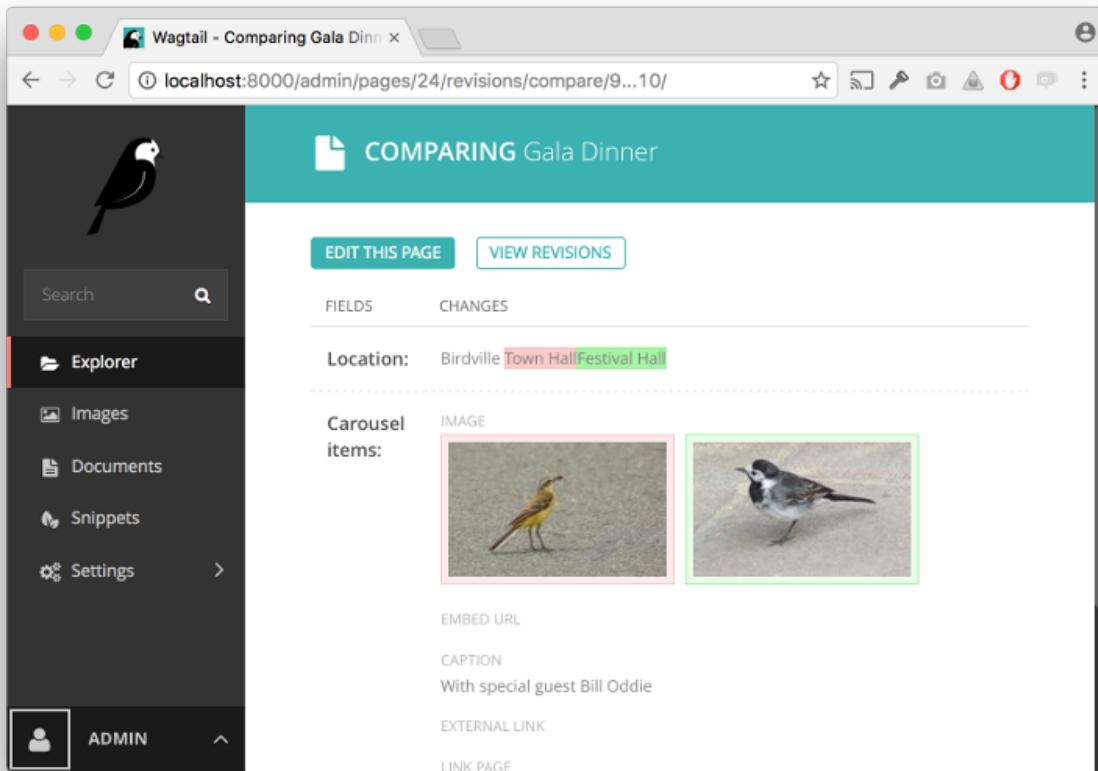
## 1.9.98 Wagtail 1.9 release notes

February 16, 2017

- [\*What's new\*](#)
- [\*Upgrade considerations\*](#)

## What's new

### Revision comparisons



Wagtail now provides the ability to view differences between revisions of a page, from the revisions listing page and when reviewing a page in moderation. This feature was developed by Karl Hobley, Janneke Janssen and Matt Westcott. Thank you to Blackstone Chambers for sponsoring this feature.

## Many-to-many relations on page models



Wagtail now supports a new field type `ParentalManyToManyField` that can be used to set up many-to-many relations on pages. For details, see the [Categories](#) section of the tutorial. This feature was developed by Thejaswi Puthraya and Matt Westcott.

## Bulk-deletion of form submissions

Form builder form submissions can now be deleted in bulk from the form submissions index page. This feature was sponsored by St John's College, Oxford and developed by Karl Hobley.

## Accessing parent context from StreamField block `get_context` methods

The `get_context` method on StreamField blocks now receives a `parent_context` keyword argument, consisting of the dict of variables passed in from the calling template. For example, this makes it possible to perform pagination logic within `get_context`, retrieving the current page number from `parent_context['request'].GET`. See [get\\_context on StreamField blocks](#). This feature was developed by Mikael Svensson and Peter Baumgartner.

## Welcome message customisation for multi-tenanted installations

The welcome message on the admin dashboard has been updated to be more suitable for multi-tenanted installations. Users whose page permissions lie within a single site will now see that site name in the welcome message, rather than the installation-wide `WAGTAIL_SITE_NAME`. As before, this message can be customised, and additional template variables have been provided for this purpose - see [Custom branding](#). This feature was developed by Jeffrey Chau.

## Other features

- Changed text of “Draft” and “Live” buttons to “View draft” and “View live” (Dan Braghis)
- Added `get_api_representation` method to streamfield blocks allowing the JSON representation in the API to be customised (Marco Fucci)
- Added `before_copy_page` and `after_copy_page` hooks (Matheus Bratfisch)
- View live / draft links in the admin now consistently open in a new window (Marco Fucci)
- `ChoiceBlock` now omits the blank option if the block is required and has a default value (Andreas Nüßlein)
- The `add_subpage` view now maintains a `next` URL parameter to specify where to redirect to after completing page creation (Robert Rollins)
- The `wagtailforms` module now allows to define custom form submission model, add custom data to CSV export and some other customisations. See [Form builder customisation](#) (Mikalai Radchuk)
- The Webpack configuration is now in a subfolder to declutter the project root, and uses environment-specific configurations for smaller bundles in production and easier debugging in development (Janneke Janssen, Thibaud Colas)
- Added page titles to title text on action buttons in the explorer, for improved accessibility (Matt Westcott)

## Bug fixes

- Help text for StreamField is now visible and does not cover block controls (Stein Strindhaug)
- “X minutes ago” timestamps are now marked for translation (Janneke Janssen, Matt Westcott)
- Avoid indexing unsaved field content on `save(update_fields=[...])` operations (Matt Westcott)
- Corrected ordering of arguments passed to `ModelAdmin get_extra_class_names_for_field_col / get_extra_attrs_for_field_col` methods (Andy Babic)
- `pageurl` / `slugurl` tags now function when `request.site` is not available (Tobias McNulty, Matt Westcott)

## Upgrade considerations

### **django-modelcluster and django-taggit dependencies updated**

Wagtail now requires version 3.0 or later of `django-modelcluster` and version 0.20 or later of `django-taggit`; earlier versions are unsupported. In normal circumstances these packages will be upgraded automatically when upgrading Wagtail; however, if your Wagtail project has a requirements file that explicitly specifies an older version, this will need to be updated.

### **get\_context methods on StreamField blocks need updating**

Previously, `get_context` methods on StreamField blocks returned a dict of variables which would be merged into the calling template’s context before rendering the block template. `get_context` methods now receive a `parent_context` dict, and are responsible for returning the final context dictionary with any new variables merged into it. The old calling convention is now deprecated, and will be phased out in Wagtail 1.11.

In most cases, the method will be calling `get_context` on the superclass, and can be updated by passing the new `parent_context` keyword argument to it:

```
class MyBlock(Block):

 def get_context(self, value):
 context = super(MyBlock, self).get_context(value)
 ...
 return context
```

becomes:

```
class MyBlock(Block):

 def get_context(self, value, parent_context=None):
 context = super(MyBlock, self).get_context(value, parent_context=parent_context)
 ...
 return context
```

Note that `get_context` methods on page models are unaffected by this change.

## 1.9.99 Wagtail 1.8.2 release notes

April 21, 2017

- *What's changed*

### What's changed

### Bug fixes

- Removed erroneous `|safe` filter from search results template in project template (Karl Hobley)
- Avoid indexing unsaved field content on `save(update_fields=[...])` operations (Matt Westcott)
- Prevent pages from being recursively copied into themselves (Matheus Bratfisch)

## 1.9.100 Wagtail 1.8.1 release notes

January 26, 2017

- *What's changed*

## What's changed

### Bug fixes

- Reduced Rendition.focal\_point\_key field length to prevent migration failure when upgrading to Wagtail 1.8 on MySQL with utf8 character encoding (Andy Chosak, Matt Westcott)

## 1.9.101 Wagtail 1.8 release notes

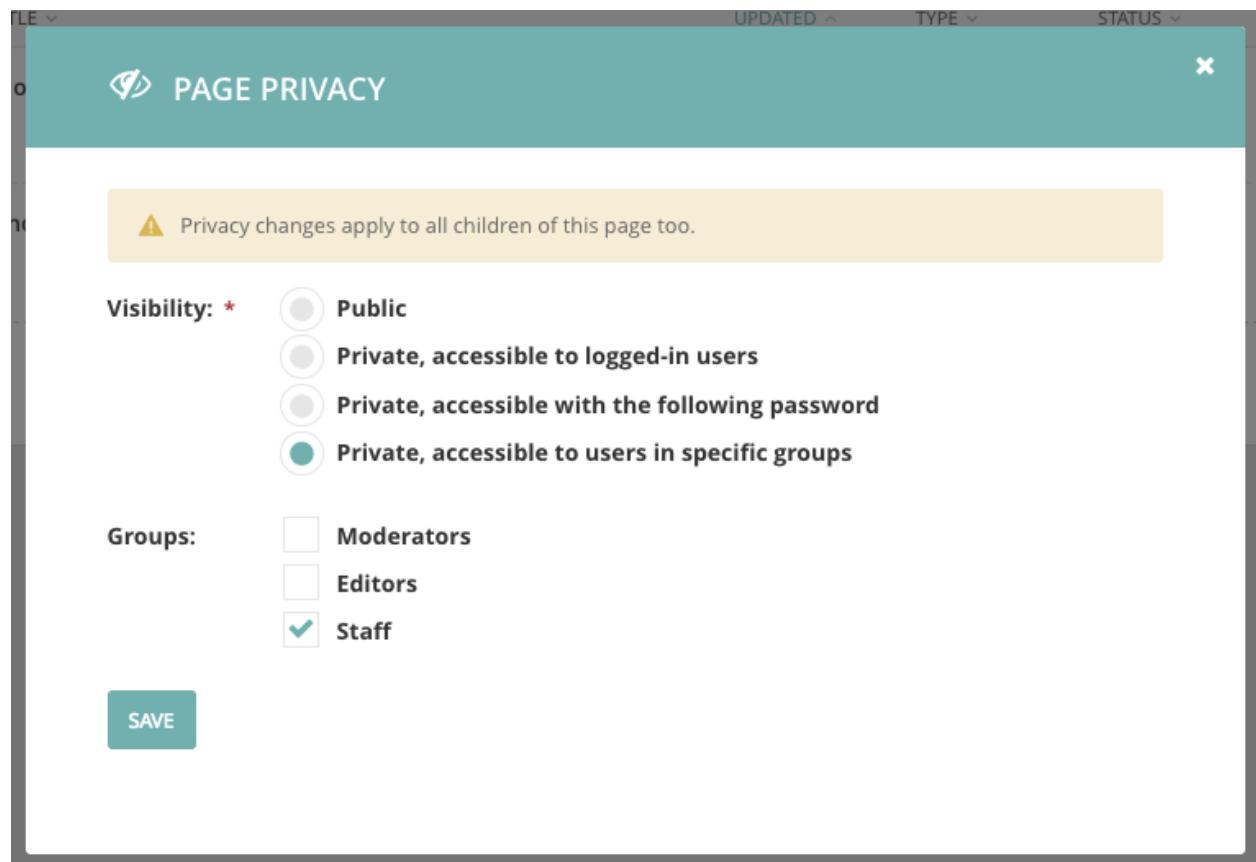
December 15, 2016

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

Wagtail 1.8 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 8 months).

### What's new

#### New page privacy options



Access to pages can now be restricted based on user accounts and group membership, rather than just through a shared password. This makes it possible to set up intranet-style sites via the admin, with no additional coding. This feature was developed by Shawn Makinson, Tom Miller, Luca Perico and Matt Westcott.

See: [Private pages](#)

### Restrictions on bulk-deletion of pages

Previously, any user with edit permission over a page and its descendants was able to delete them all as a single action, which led to the risk of accidental deletions. To guard against this, the permission rules have been revised so that a user with basic permissions can only delete pages that have no children; in order to delete a whole subtree, they must individually delete each child page first. A new “bulk delete” permission type has been added which allows a user to delete pages with children, as before; superusers receive this permission implicitly, and so there is no change of behaviour for them.

See: [Permissions](#)

This feature was developed by Matt Westcott.

### Elasticsearch 5 support

Wagtail now supports Elasticsearch 5. See [Elasticsearch Backend](#) for configuration details. This feature was developed by Karl Hobley.

### Permission-limited admin breadcrumb

Breadcrumb links within the admin are now limited to the portion of the page tree that covers all pages the user has permission over. As with the changes to the explorer sidebar menu in Wagtail 1.6, this is a step towards supporting full multi-tenancy (where multiple sites on the same Wagtail installation can be fully isolated from each other through permission configuration). This feature was developed by Jeffrey Chau, Robert Rollins and Matt Westcott.

### Updated tutorial

The “*Your first Wagtail site*” tutorial has been extensively updated to cover concepts such as dynamic page listings, template context variables, and tagging. This update was contributed by Scot Hacker, with additions from Matt Westcott.

### Other features

- Added support of a custom `edit_handler` for site settings. See [docs for the site settings module](#). (Axel Haustant)
- Added `get_landing_page_template` getter method to `AbstractForm` (Gagaro)
- Added `Page.get_admin_display_title` method to override how the title is displayed in the admin (Henk-Jan van Hasselaar)
- Added support for specifying custom HTML attributes for table rows on `ModelAdmin` index pages. See [ModelAdmin.get\\_extra\\_attrs\\_for\\_row\(\)](#) (Andy Babic)
- Added `first_common_ancestor` method to `PageQuerySet` (Tim Heap)
- Page chooser now opens at the deepest ancestor page that covers all the pages of the required page type (Tim Heap)

- `PageChooserBlock` now accepts a `target_model` option to specify the required page type (Tim Heap)
- `Modeladmin` forms now respect `fields` / `exclude` options passed on custom model forms (Thejaswi Puthraya)
- Added new StreamField block type `StaticBlock` for blocks that occupy a position in a stream but otherwise have no configuration; see [`StaticBlock`](#) (Benoît Vogel)
- Added new StreamField block type `BlockQuoteBlock` (Scot Hacker)
- Updated Cloudflare cache module to use the v4 API (Albert O'Connor)
- Added `exclude_from_explorer` attribute to the `ModelAdmin` class to allow hiding instances of a page type from Wagtail's explorer views (Andy Babic)
- Added `above_login`, `below_login`, `fields` and `login_form` customisation blocks to the login page template - see [`Customising admin templates`](#) (Tim Heap)
- `ChoiceBlock` now accepts a callable as the choices list (Mikalai Radchuk)
- Redundant action buttons are now omitted from the root page in the explorer (Nick Smith)
- Locked pages are now disabled from editing at the browser level (Edd Baldry)
- Added `wagtail.core.query.PageQuerySet.in_site()` method for filtering page QuerySets to pages within the specified site (Chris Rogers)
- Added the ability to override the default index settings for Elasticsearch. See [`Elasticsearch Backend`](#) (PyMan Claudio Marinoszzi)
- Extra options for the Elasticsearch constructor should be now defined with the new key `OPTIONS` of the `WAGTAILSEARCH_BACKENDS` setting (PyMan Claudio Marinoszzi)

## Bug fixes

- `AbstractForm` now respects custom `get_template` methods on the page model (Gagaro)
- Use specific page model for the parent page in the explore index (Gagaro)
- Remove responsive styles in embed when there is no ratio available (Gagaro)
- Parent page link in page search modal no longer disappears on hover (Dan Braghis)
- `ModelAdmin` views now consistently call `get_context_data` (Andy Babic)
- Header for search results on the redirects index page now shows the correct count when the listing is paginated (Nick Smith)
- `set_url_paths` management command is now compatible with Django 1.10 (Benjamin Bach)
- Form builder email notifications now output multiple values correctly (Sævar Öfjörð Magnússon)
- Closing ‘more’ dropdown on explorer no longer jumps to the top of the page (Ducky)
- Users with only publish permission are no longer given implicit permission to delete pages (Matt Westcott)
- `search_garbage_collect` management command now works when wagtailsearchpromotions is not installed (Morgan Aubert)
- `wagtail.contrib.settings` context processor no longer fails when `request.site` is unavailable (Diederik van der Boor)
- `TableBlock` content is now indexed for search (Morgan Aubert)
- `Page.copy()` is now marked as `alters_data`, to prevent template code from triggering it (Diederik van der Boor)

## Upgrade considerations

### unique\_together constraint on custom image rendition models needs updating

If your project is using a custom image model (see [Custom image models](#)), you will need to update the `unique_together` option on the corresponding Rendition model when upgrading to Wagtail 1.8. Change the line:

```
unique_together = (
 ('image', 'filter', 'focal_point_key'),
)
```

to:

```
unique_together = (
 ('image', 'filter_spec', 'focal_point_key'),
)
```

You will then be able to run `manage.py makemigrations` and `manage.py migrate` as normal.

Additionally, third-party code that accesses the Filter and Rendition models directly should note the following and make updates where applicable:

- Filter will no longer be a Django model as of Wagtail 1.9, and as such, ORM operations on it (such as `save()` and `Filter.objects`) are deprecated. It should be instantiated and used as an in-memory object instead - for example, `flt, created = Filter.objects.get_or_create(spec='fill-100x100')` should become `flt = Filter(spec='fill-100x100')`.
- The `filter` field of Rendition models is no longer in use; lookups should instead be performed on the `filter_spec` field, which contains a filter spec string such as '`fill-100x100`'.

### wagtail.wagtailimages.models.get\_image\_model has moved

The `get_image_model` function should now be imported from `wagtail.wagtailimages` rather than `wagtail.wagtailimages.models`. See [Referring to the image model](#).

### Non-administrators now need ‘bulk delete’ permission to delete pages with children

As a precaution against accidental data loss, this release introduces a new “bulk delete” permission on pages, which can be set through the Settings -> Groups area. Non-administrator users must have this permission in order to delete pages that have children; a user without this permission would have to delete each child individually before deleting the parent. By default, no groups are assigned this new permission. If you wish to restore the previous behaviour, and don’t want to configure permissions manually through the admin interface, you can do so with a data migration. Create an empty migration using `./manage.py makemigrations myapp --empty --name assign_bulk_delete_permission` (replacing `myapp` with the name of one of your project’s apps) and edit the migration file to contain the following:

```
from __future__ import unicode_literals

from django.db import migrations

def add_bulk_delete_permission(apps, schema_editor):
 """Find all groups with add/edit page permissions, and assign them bulk_delete_permission"""

```

(continues on next page)

(continued from previous page)

```

GroupPagePermission = apps.get_model('wagtailcore', 'GroupPagePermission')
for group_id, page_id in GroupPagePermission.objects.filter(
 permission_type__in=['add', 'edit']
).values_list('group', 'page').distinct():
 GroupPagePermission.objects.create(
 group_id=group_id, page_id=page_id, permission_type='bulk_delete'
)

def remove_bulk_delete_permission(apps, schema_editor):
 GroupPagePermission = apps.get_model('wagtailcore', 'GroupPagePermission')
 GroupPagePermission.objects.filter(permission_type='bulk_delete').delete()

class Migration(migrations.Migration):

 dependencies = [
 # keep the original dependencies line
]

 operations = [
 migrations.RunPython(add_bulk_delete_permission, remove_bulk_delete_permission),
]

```

### Cloudflare cache module now requires a ZONEID setting

The `wagtail.contrib.wagtailfrontendcache.backends.CloudflareBackend` module has been updated to use Cloudflare's v4 API, replacing the previous v1 implementation (which is [unsupported as of November 9th, 2016](#)). The new API requires users to supply a *zone identifier*, which should be passed as the `ZONEID` field of the `WAGTAILFRONTENDCACHE` setting:

```

WAGTAILFRONTENDCACHE = {
 'cloudflare': {
 'BACKEND': 'wagtail.contrib.wagtailfrontendcache.backends.CloudflareBackend',
 'EMAIL': 'your-cloudflare-email-address@example.com',
 'TOKEN': 'your cloudflare api token',
 'ZONEID': 'your cloudflare domain zone id',
 },
}

```

For details of how to obtain the zone identifier, see the [Cloudflare API documentation](#).

## Extra options for the Elasticsearch constructor should be now defined with the new key OPTIONS of the WAGTAILSEARCH\_BACKENDS setting

For the Elasticsearch backend, all extra keys defined in WAGTAILSEARCH\_BACKENDS are passed directly to the Elasticsearch constructor. All these keys now should be moved inside the new OPTIONS dictionary. The old behaviour is still supported, but deprecated.

For example, the following configuration changes the connection class that the Elasticsearch connector uses:

```
from elasticsearch import RequestsHttpConnection

WAGTAILSEARCH_BACKENDS = {
 'default': {
 'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch',
 'connection_class': RequestsHttpConnection,
 }
}
```

As connection\_class needs to be passed through to the Elasticsearch connector, it should be moved to the new OPTIONS dictionary:

```
from elasticsearch import RequestsHttpConnection

WAGTAILSEARCH_BACKENDS = {
 'default': {
 'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch',
 'OPTIONS': {
 'connection_class': RequestsHttpConnection,
 }
 }
}
```

## 1.9.102 Wagtail 1.7 release notes

*October 20, 2016*

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

### What's new

#### Elasticsearch 2 support

Wagtail now supports Elasticsearch 2. Note that you need to change backend in WAGTAILSEARCH\_BACKENDS, if you wish to switch to Elasticsearch 2. This feature was developed by Karl Hobley.

See: [Elasticsearch Backend](#)

## New image tag options for file type and JPEG compression level

The `{% image %}` tag now supports extra parameters for specifying the image file type and JPEG compression level on a per-tag basis. See [Output image format](#) and [Image quality](#). This feature was developed by Karl Hobley.

## AWS CloudFront support added to cache invalidation module

Wagtail's cache invalidation module can now invalidate pages cached in AWS CloudFront when they are updated or unpublished. This feature was developed by Rob Moorman.

See: [Amazon CloudFront](#)

## Unpublishing subpages

Unpublishing a page now gives the option to unpublish its subpages at the same time. This feature was developed by Jordi Joan.

## Minor features

- The `| embed` filter has been converted into a templatetag `{% embed %}` (Janneke Janssen)
- The `wagtailforms` module now provides a `FormSubmissionPanel` for displaying details of form submissions; see [Displaying form submission information](#) for documentation. (João Luiz Lorencetti)
- The Wagtail version number can now be obtained as a tuple using `from wagtail import VERSION` (Tim Heap)
- `send_mail` logic has been moved from `AbstractEmailForm.process_form_submission` into `AbstractEmailForm.send_mail`. Now it's easier to override this logic (Tim Leguijt)
- Added `before_create_page`, `before_edit_page`, `before_delete_page` hooks (Karl Hobley)
- Updated font sizes and colours to improve legibility of admin menu and buttons (Stein Strindhaug)
- Added pagination to “choose destination” view when moving pages (Nick Smith, Žan Anderle)
- Added ability to annotate search results with score - see [Annotating results with score](#) (Karl Hobley)
- Added ability to limit access to form submissions - see [filter\\_form\\_submissions\\_for\\_user](#) (Mikalai Radchuk)
- Added the ability to configure the number of days search logs are kept for, through the `WAGTAILSEARCH_HITS_MAX_AGE` setting (Stephen Rice)
- SnippetChooserBlock now supports passing the model name as a string (Nick Smith)
- Redesigned account settings / logout area in the sidebar for better clarity (Janneke Janssen)
- Pillow's image optimisation is now applied when saving JPEG images (Karl Hobley)

### Bug fixes

- Migrations for wagtailcore and project template are now reversible (Benjamin Bach)
- Migrations no longer depend on wagtailcore and taggit's `__latest__` migration, logically preventing those apps from receiving new migrations (Matt Westcott)
- The default image format label text ('Full width', 'Left-aligned', 'Right-aligned') is now localised (Mikalai Radchuk)
- Text on the front-end 'password required' form is now marked for translation (Janneke Janssen)
- Text on the page view restriction form is now marked for translation (Luiz Boaretto)
- Fixed toggle behaviour of userbar on mobile (Robert Rollins)
- Image rendition / document file deletion now happens on a `post_delete` signal, so that files are not lost if the deletion does not proceed (Janneke Janssen)
- "Your recent edits" list on dashboard no longer leaves out pages that another user has subsequently edited (Michael Cordover, Kees Hink, João Luiz Lorencetti)
- `InlinePanel` now accepts a `classname` parameter as per the documentation (emg36, Matt Westcott)
- Disabled use of escape key to revert content of rich text fields, which could cause accidental data loss (Matt Westcott)
- Setting `USE_THOUSAND_SEPARATOR = True` no longer breaks the rendering of numbers in JS code for `InlinePanel` (Mattias Loverot, Matt Westcott)
- Images / documents pagination now preserves GET parameters (Bojan Mihelac)
- Wagtail's `UserProfile` model now sets a `related_name` of `wagtail_userprofile` to avoid naming collisions with other user profile models (Matt Westcott)
- Non-text content is now preserved when adding or editing a link within rich text (Matt Westcott)
- Fixed preview when `SECURE_SSL_REDIRECT = True` (Aymeric Augustin)
- Prevent hang when truncating an image filename without an extension (Ricky Robinett)

### Upgrade considerations

#### Project template's initial migration should not depend on `wagtailcore.__latest__`

On projects created under previous releases of Wagtail, the `home/migrations/0001_initial.py` migration created by the `wagtail start` command contains the following dependency line:

```
dependencies = [
 ('wagtailcore', '__latest__'),
]
```

This may produce `InconsistentMigrationHistory` errors under Django 1.10 when upgrading Wagtail, since Django interprets this to mean that no new migrations can legally be added to wagtailcore after this migration is applied. This line should be changed to:

```
dependencies = [
 ('wagtailcore', '0029_unicode_slugfield_dj19'),
]
```

## Custom image models require a data migration for the new `filter_spec` field

The data model for image renditions will be changed in Wagtail 1.8 to eliminate `Filter` as a model. Wagtail sites using a custom image model (see [Custom image models](#)) need to have a schema and data migration in place prior to upgrading to Wagtail 1.8. To create these migrations:

- Run `manage.py makemigrations` to create the schema migration
- Run `manage.py makemigrations --empty myapp` (replacing `myapp` with the name of the app containing the custom image model) to create an empty migration
- Edit the created migration to contain:

```
from wagtail.wagtailimages.utils import get_fill_filter_spec_migrations
```

and, for the operations list:

```
forward, reverse = get_fill_filter_spec_migrations('myapp', 'CustomRendition')
operations = [
 migrations.RunPython(forward, reverse),
]
```

replacing `myapp` and `CustomRendition` with the app and model name for the custom rendition model.

## embed template filter is now a template tag

The `embed` template filter, used to translate the URL of a media resource (such as a YouTube video) into a corresponding embeddable HTML fragment, has now been converted to a template tag. Any template code such as:

```
{% load wagtailembeds_tags %}
...
{{ my_media_url|embed }}
```

should now be rewritten as:

```
{% load wagtailembeds_tags %}
...
{% embed my_media_url %}
```

## 1.9.103 Wagtail 1.6.3 release notes

*September 30, 2016*

- *What's changed*

## What's changed

### Bug fixes

- Restore compatibility with django-debug-toolbar 1.5 (Matt Westcott)
- Edits to StreamFields are no longer ignored in page edits on Django >=1.10.1 when a default value exists (Matt Westcott)

## 1.9.104 Wagtail 1.6.2 release notes

September 2, 2016

- *What's changed*

## What's changed

### Bug fixes

- Initial values of checkboxes on group permission edit form now are visible on Django 1.10 (Matt Westcott)

## 1.9.105 Wagtail 1.6.1 release notes

August 26, 2016

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

## What's new

### Minor features

- Added `WAGTAIL_ALLOW_UNICODE_SLUGS` setting to make Unicode support optional in page slugs (Matt Westcott)

### Bug fixes

- Wagtail's middleware classes are now compatible with Django 1.10's new-style middleware (Karl Hobley)
- The `can_create_at()` method is now checked in the create page view (Mikalai Radchuk)
- Fixed regression on Django 1.10.1 causing Page subclasses to fail to use PageManager (Matt Westcott)
- ChoiceBlocks with lazy translations as option labels no longer break Elasticsearch indexing (Matt Westcott)
- The page editor no longer fails to load JavaScript files with `ManifestStaticFilesStorage` (Matt Westcott)

- Django 1.10 enables client-side validation for all forms by default, but it fails to handle all the nuances of how forms are used in Wagtail. The client-side validation has been disabled for the Wagtail UI (Matt Westcott)

## Upgrade considerations

### Multi-level inheritance and custom managers

The inheritance rules for *Custom Page managers* have been updated to match Django's standard behaviour. In the vast majority of scenarios there will be no change. However, in the specific case where a page model with a custom objects manager is subclassed further, the subclass will be assigned a plain Manager instead of a PageManager, and will now need to explicitly override this with a PageManager to function correctly:

```
class EventPage(Page):
 objects = EventManager()

class SpecialEventPage(EventPage):
 # Previously SpecialEventPage.objects would be set to a PageManager automatically;
 # this now needs to be set explicitly
 objects = PageManager()
```

## 1.9.106 Wagtail 1.6 release notes

August 15, 2016

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

### What's new

#### Django 1.10 support

Wagtail is now compatible with Django 1.10. Thanks to Mikalai Radchuk and Paul J Stevens for developing this, and to Tim Graham for reviewing and additional Django core assistance.

#### {% include\_block %} tag for improved StreamField template inclusion

In previous releases, the standard way of rendering the HTML content of a StreamField was through a simple variable template tag, such as `{% page.body %}`. This had the drawback that any templates used in the StreamField rendering would not inherit variables from the parent template's context, such as `page` and `request`. To address this, a new template tag `{% include_block page.body %}` has been introduced as the new recommended way of outputting Streamfield content - this replicates the behaviour of Django's `{% include %}` tag, passing on the full template context by default. For full documentation, see *Template rendering*. This feature was developed by Matt Westcott, and additionally ported to Jinja2 (see: *Jinja2 template support*) by Mikalai Radchuk.

## Unicode page slugs

Page URL slugs can now contain Unicode characters, when using Django 1.9 or above. This feature was developed by Behzad Nategh.

## Permission-limited explorer menu

The explorer sidebar menu now limits the displayed pages to the ones the logged-in user has permission for. For example, if a user has permission over the pages MegaCorp / Departments / Finance and MegaCorp / Departments / HR, then their menu will begin at “Departments”. This reduces the amount of “drilling-down” the user has to do, and is an initial step towards supporting fully independent sites on the same Wagtail installation. This feature was developed by Matt Westcott and Robert Rollins, California Institute of Technology.

## Minor features

- Image upload form in image chooser now performs client side validation so that the selected file is not lost in the submission (Jack Paine)
- oEmbed URL for audioBoom was updated (Janneke Janssen)
- Remember tree location in page chooser when switching between Internal / External / Email link (Matt Westcott)
- `FieldRowPanel` now creates equal-width columns automatically if `col*` classnames are not specified (Chris Rogers)
- Form builder now validates against multiple fields with the same name (Richard McMillan)
- The ‘choices’ field on the form builder no longer has a maximum length (Johannes Spielmann)
- Multiple ChooserBlocks inside a StreamField are now prefetched in bulk, for improved performance (Michael van Tellingen, Roel Bruggink, Matt Westcott)
- Added new EmailBlock and IntegerBlock (Oktay Altay)
- Added a new FloatBlock, DecimalBlock and a RegexBlock (Oktay Altay, Andy Babic)
- Wagtail version number is now shown on the settings menu (Chris Rogers)
- Added a system check to validate that fields listed in `search_fields` are defined on the model (Josh Schneier)
- Added formal APIs for customising the display of StructBlock forms within the page editor - see [\*Custom editing interfaces for StructBlock\*](#) (Matt Westcott)
- `wagtailforms.models.AbstractEmailForm` now supports multiple email recipients (Serafeim Papastefanos)
- Added ability to delete users through Settings -> Users (Vincent Audebert; thanks also to Ludolf Takens and Tobias Schmidt for alternative implementations)
- Page previews now pass additional HTTP headers, to simulate the page being viewed by the logged-in user and avoid clashes with middleware (Robert Rollins)
- Added back buttons to page delete and unpublish confirmation screens (Matt Westcott)
- Recognise Flickr embed URLs using HTTPS (Danielle Madeley)
- Success message when publishing a page now correctly respects custom URLs defined on the specific page class (Chris Darko)
- Required blocks inside StreamField are now indicated with asterisks (Stephen Rice)

## Bug fixes

- Email templates and document uploader now support custom STATICFILES\_STORAGE (Jonny Scholes)
- Removed alignment options (deprecated in HTML and not rendered by Wagtail) from TableBlock context menu (Moritz Pfeiffer)
- Fixed incorrect CSS path on ModelAdmin’s “choose a parent page” view
- Prevent empty redirect by overnormalisation
- “Remove link” button in rich text editor didn’t trigger “edit” event, leading to the change to sometimes not be persisted (Matt Westcott)
- RichText values can now be correctly evaluated as booleans (Mike Dingjan, Bertrand Bordage)
- wagtailforms no longer assumes an .html extension when determining the landing page template filename (kakulukia)
- Fixed styling glitch on bi-colour icon + text buttons in Chrome (Janneke Janssen)
- StreamField can now be used in an InlinePanel (Gagarro)
- StreamField block renderings using templates no longer undergo double escaping when using Jinja2 (Aymeric Augustin)
- RichText objects no longer undergo double escaping when using Jinja2 (Aymeric Augustin, Matt Westcott)
- Saving a page by pressing enter key no longer triggers a “Changes may not be saved message” (Sean Muck, Matt Westcott)
- RoutablePageMixin no longer breaks in the presence of instance-only attributes such as those generated by FileFields (Fábio Macêdo Mendes)
- The --schema-only flag on update\_index no longer expects an argument (Karl Hobley)
- Added file handling to support custom user add/edit forms with images/files (Eraldo Energy)
- Placeholder text in modeladmin search now uses the correct template variable (Adriaan Tijsseling)
- Fixed bad SQL syntax for updating URL paths on Microsoft SQL Server (Jesse Legg)
- Added workaround for Django 1.10 bug <https://code.djangoproject.com/ticket/27037> causing forms with file upload fields to fail validation (Matt Westcott)

## Upgrade considerations

### Form builder FormField models require a migration

There are some changes in the `wagtailforms.models.AbstractFormField` model:

- The `choices` field has been changed from a `CharField` to a `TextField`, to allow it to be of unlimited length;
- The help text for the `to_address` field has been changed: it now gives more information on how to specify multiple addresses.

These changes require migration. If you are using the `wagtailforms` module in your project, you will need to run `python manage.py makemigrations` and `python manage.py migrate` after upgrading, in order to apply changes to your form page models.

### TagSearchable needs removing from custom image / document model migrations

The mixin class `wagtail.wagtailadmin.taggable.TagSearchable`, used internally by image and document models, has been deprecated. If you are using custom image or document models in your project, the migration(s) which created them will contain frozen references to `wagtail.wagtailadmin.taggable.TagSearchable`, which must now be removed. The line:

```
import wagtail.wagtailadmin.taggable
```

should be replaced by:

```
import wagtail.wagtailsearch.index
```

and the line:

```
bases=(models.Model, wagtail.wagtailadmin.taggable.TagSearchable),
```

should be updated to:

```
bases=(models.Model, wagtail.wagtailsearch.index.Indexed),
```

### render and render\_basic methods on StreamField blocks now accept a context keyword argument

The `render` and `render_basic` methods on `wagtail.wagtailcore.blocks.Block` have been updated to accept an optional `context` keyword argument, a template context to use when rendering the block. If you have defined any custom StreamField blocks that override either of these methods, the method signature now needs to be updated to include this keyword argument:

```
class MyBlock(Block):

 def render(self, value):
 ...

 def render_basic(self, value):
 ...
```

should now become:

```
class MyBlock(Block):

 def render(self, value, context=None):
 ...

 def render_basic(self, value, context=None):
 ...
```

## 1.9.107 Wagtail 1.5.3 release notes

*July 18, 2016*

- *What's changed*

### What's changed

#### Bug fixes

- Pin html5lib to version 0.999999 to prevent breakage caused by internal API changes (Liam Brenner)

## 1.9.108 Wagtail 1.5.2 release notes

*June 8, 2016*

- *What's new*

### What's new

#### Bug fixes

- Fixed regression in 1.5.1 on editing external links (Stephen Rice)

## 1.9.109 Wagtail 1.5.1 release notes

*June 7, 2016*

- *What's new*

### What's new

#### Bug fixes

- When editing a document link in rich text, the document ID is no longer erroneously interpreted as a page ID (Stephen Rice)
- Removing embedded media from rich text by mouse click action now gets correctly registered as a change to the field (Loic Teixeira)
- Rich text editor is no longer broken in InlinePanels (Matt Westcott, Gagaro)
- Rich text editor is no longer broken in settings (Matt Westcott)
- Link tooltip now shows correct urls for newly inserted document links (Matt Westcott)
- Now page chooser (in a rich text editor) opens up at the link's parent page, rather than at the page itself (Matt Westcott)

- Reverted fix for explorer menu scrolling with page content, as it blocked access to menus that exceed screen height
- Image listing in the image chooser no longer becomes unpaginated after an invalid upload form submission (Stephen Rice)
- Confirmation message on the ModelAdmin delete view no longer errors if the model's string representation depends on the primary key (Yannick Chabbert)
- Applied correct translation tags for ‘permanent’ / ‘temporary’ labels on redirects (Matt Westcott)

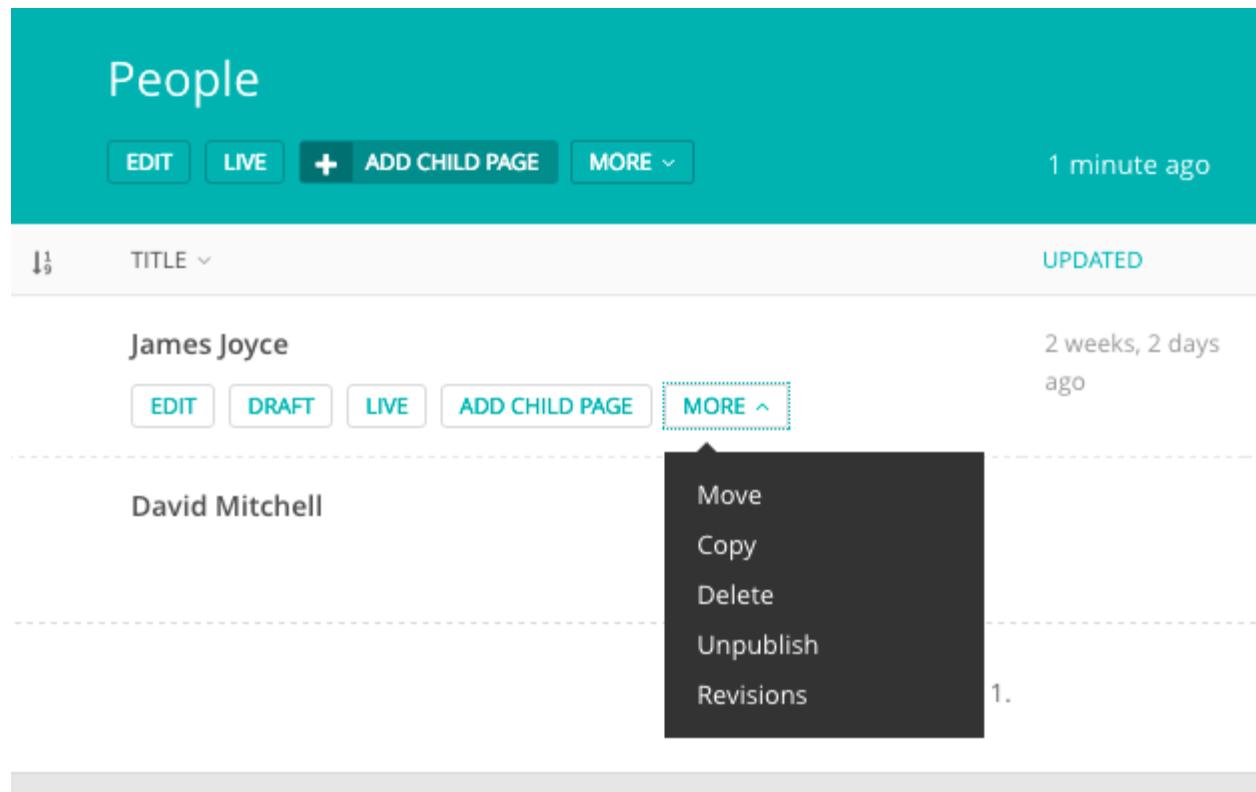
## 1.9.110 Wagtail 1.5 release notes

May 1, 2016

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

### What's new

#### Reorganised page explorer actions



The action buttons on the page explorer have been reorganised to reduce clutter, and lesser-used actions have been moved to a “More” dropdown. A new hook `register_page_listing_buttons` has been added for adding custom action buttons to the page explorer.

## ModelAdmin

Wagtail now includes an app `wagtail.contrib.modeladmin` (previously available separately as the `wagtailmodeladmin` package) which allows you to configure arbitrary Django models to be listed, added and edited through the Wagtail admin.

The screenshot shows the Wagtail admin interface for the 'COUNTRIES' model. On the left is a sidebar with a navigation menu: 'Search' (with a magnifying glass icon), 'Explorer', 'Images', 'Documents', 'Snippets', 'Countries' (which is selected and highlighted with a red border), and 'Settings'. Below the menu are user details: 'Matt Westcott' and a 'Log out' link. The main content area has a teal header with the title 'COUNTRIES' and a 'ADD COUNTRY' button. Below the header is a table with three rows of data:

NAME	CONTINENT	POPULATION
China	Asia	1376570000
India	Asia	1289020000
Indonesia	Asia	258705000

Each row has an 'EDIT' button (green) and a 'DELETE' button (red). To the right of the table is a 'FILTER' section titled 'By continent' with a dropdown menu containing the following options: ALL, AFRICA, ANTARCTICA, ASIA (which is selected and highlighted with a teal background), AUSTRALIA, EUROPE, NORTH AMERICA, and SOUTH AMERICA.

See [ModelAdmin](#) for full documentation. This feature was developed by Andy Babic.

## TableBlock

`TableBlock`, a new StreamField block type for editing table-based content, is now available through the `wagtail.contrib.table_block` module.

The screenshot shows the Wagtail rich text editor interface. A modal window titled "Table" is open. It contains several configuration options:

- "Row header": An unchecked checkbox.
- "Display the first row as a header": A tooltip explaining the function of the "Row header" option.
- "Column header": A checked checkbox.
- "Display the first column as a header": A tooltip explaining the function of the "Column header" option.
- "Table caption": A text input field containing "A heading that identifies the overall topic of the table, and is useful for screen reader users".
- A preview area showing a table with two rows and two columns. The first row has "From" and "To" headers. The data rows are: "redirect-1" and "https://www.example.com/"; "redirect-2" and "https://www.example.com/".

See [TableBlock](#) for documentation. This feature was developed by Moritz Pfeiffer, David Seddon and Brad Busenius.

### Improved link handling in rich text

The user experience around inserting, editing and removing links inside rich text areas has been greatly improved: link destinations are shown as tooltips, and existing links can be edited as well as unlinked. This feature was developed by Loic Teixeira.

### Improvements to the “Image serve view”

#### *Dynamic image serve view*

This view, which is used for requesting image thumbnails from an external app, has had some improvements made to it in this release.

- A “[redirect](#)” action has been added which will redirect the user to where the resized image is hosted rather than serving it from the app. This may be beneficial for performance if the images are hosted externally (eg, S3)
- It now takes an optional extra path component which can be used for appending a filename to the end of the URL
- The key is now configurable on the view so you don’t have to use your project’s SECRET\_KEY
- It’s been refactored into a class based view and you can now create multiple serve views with different image models and/or keys
- It now supports [serving image files using django-sendfile](#) (Thanks to Yannick Chabbert for implementing this)

## Minor features

- Password reset email now reminds the user of their username (Matt Westcott)
- Added jinja2 support for the `settings` template tag (Tim Heap)
- Added ‘revisions’ action to pages list (Roel Bruggink)
- Added a hook `insert_global_admin_js` for inserting custom JavaScript throughout the admin backend (Tom Dyson)
- Recognise instagram embed URLs with `www` prefix (Matt Westcott)
- The type of the `search_fields` attribute on Page models (and other searchable models) has changed from a tuple to a list (see upgrade consideration below) (Tim Heap)
- Use `PasswordChangeForm` when user changes their password, requiring the user to enter their current password (Matthijs Melissen)
- Highlight current day in date picker (Jonas Lergell)
- Eliminated the deprecated `register.assignment_tag` on Django 1.9 (Josh Schneier)
- Increased size of Save button on site settings (Liam Brenner)
- Optimised `Site.find_for_request` to only perform one database query (Matthew Downey)
- Notification messages on creating / editing sites now include the site name if specified (Chris Rogers)
- Added `--schema-only` option to `update_index` management command
- Added meaningful default icons to `StreamField` blocks (Benjamin Bach)
- Added title text to action buttons in the page explorer (Liam Brenner)
- Changed project template to explicitly import development settings via `settings.dev` (Tomas Olander)
- Improved L10N and I18N for revisions list (Roel Bruggink)
- The multiple image uploader now displays details of server errors (Nigel Fletton)
- Added `WAGTAIL_APPEND_SLASH` setting to determine whether page URLs end in a trailing slash - see *Append Slash* (Andrew Tork Baker)
- Added auto resizing text field, richtext field, and snippet chooser to styleguide (Liam Brenner)
- Support field widget media inside `StreamBlock` blocks (Karl Hobley)
- Spinner was added to Save button on site settings (Liam Brenner)
- Added success message after logout from Admin (Liam Brenner)
- Added `get_upload_to` method to `AbstractRendition` which, when overridden, allows control over where image renditions are stored (Rob Moggach and Matt Westcott)
- Added a mechanism to customise the add / edit user forms for custom user models - see *Custom user models* (Nigel Fletton)
- Added internal provision for swapping in alternative rich text editors (Karl Hobley)

### Bug fixes

- The currently selected day is now highlighted only in the correct month in date pickers (Jonas Lergell)
- Fixed crash when an image without a source file was resized with the “dynamic serve view”
- Registered settings admin menu items now show active correctly (Matthew Downey)
- Direct usage of `Document` model replaced with `get_document_model` function in `wagtail.contrib.wagtailmedusa` and in `wagtail.contrib.wagtailapi`
- Failures on sending moderation notification emails now produce a warning, rather than crashing the admin page outright (Matt Fozard)
- All admin forms that could potentially include file upload fields now specify `multipart/form-data` where appropriate (Tim Heap)
- REM units in Wagtailuserbar caused incorrect spacing (Vincent Audebert)
- Explorer menu no longer scrolls with page content (Vincent Audebert)
- `decorate_urlpatterns` now uses `functools.update_wrapper` to keep view names and docstrings (Mario César)
- StreamField block controls are no longer hidden by the StreamField menu when prepending a new block (Vincent Audebert)
- Removed invalid use of `__` alias that prevented strings getting picked up for translation (Juha Yrjölä)
- *Routable pages* without a main view no longer raise a `TypeError` (Bojan Mihelac)
- Fixed `UnicodeEncodeError` in `wagtailforms` when downloading a CSV for a form containing non-ASCII field labels on Python 2 (Mikalai Radchuk)
- Server errors during search indexing on creating / updating / deleting a model are now logged, rather than causing the overall operation to fail (Karl Hobley)
- Objects are now correctly removed from search indexes on deletion (Karl Hobley)

### Upgrade considerations

#### Buttons in admin now require `class="button"`

The Wagtail admin CSS has been refactored for maintainability, and buttons now require an explicit `button` class. (Previously, the styles were applied on all inputs of type “`submit`”, “`reset`” or “`button`”.) If you have created any apps that extend the Wagtail admin with new views / templates, you will need to add this class to all buttons.

#### The `search_fields` attribute on models should now be set to a list

On searchable models (eg, `Page` or custom `Image` models) the `search_fields` attribute should now be a list instead of a tuple.

For example, the following `Page` model:

```
class MyPage(Page):
 ...

 search_fields = Page.search_fields + (
```

(continues on next page)

(continued from previous page)

```
 indexed.SearchField('body'),
)
```

Should be changed to:

```
class MyPage(Page):
 ...

 search_fields = Page.search_fields + [
 indexed.SearchField('body'),
]
```

To ease the burden on third-party modules, adding tuples to `Page.search_fields` will still work. But this backwards-compatibility fix will be removed in Wagtail 1.7.

### Elasticsearch backend now defaults to verifying SSL certs

Previously, if you used the Elasticsearch backend, configured with the `URLS` property like:

```
WAGTAILSEARCH_BACKENDS = {
 'default': {
 'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch',
 'URLS': ['https://example.com/'],
 }
}
```

Elasticsearch would not be configured to verify SSL certificates for HTTPS URLs. This has been changed so that SSL certificates are verified for HTTPS connections by default.

If you need the old behaviour back, where SSL certificates are not verified for your HTTPS connection, you can configure the Elasticsearch backend with the `HOSTS` option, like so:

```
WAGTAILSEARCH_BACKENDS = {
 'default': {
 'BACKEND': 'wagtail.wagtailsearch.backends.elasticsearch',
 'HOSTS': [{
 'host': 'example.com',
 'use_ssl': True,
 'verify_certs': False,
 }],
 }
}
```

See the [Elasticsearch-py documentation](#) for more configuration options.

## Project template now imports settings.dev explicitly

In previous releases, the project template's `settings/__init__.py` file was set up to import the development settings (`settings/dev.py`), so that these would be picked up as the default (i.e. whenever a settings module was not specified explicitly). However, in some setups this meant that the development settings were being inadvertently imported in production mode.

For this reason, the import in `settings/__init__.py` has now been removed, and commands must now specify `myproject.settings.dev` or `myproject.settings.production` as appropriate; the supporting scripts (such as `manage.py`) have been updated accordingly. As this is a change to the project template, existing projects are not affected; however, if you have any common scripts or configuration files that rely on importing `myproject.settings` as the settings module, these will need to be updated in order to work on projects created under Wagtail 1.5.

## 1.9.111 Wagtail 1.4.6 release notes

*July 18, 2016*

- *What's changed*

### What's changed

#### Bug fixes

- Pin `html5lib` to version 0.999999 to prevent breakage caused by internal API changes (Liam Brenner)

## 1.9.112 Wagtail 1.4.5 release notes

*May 19, 2016*

- *What's changed*

### What's changed

#### Bug fixes

- Paste / drag operations done entirely with the mouse are now correctly picked up as edits within the rich text editor (Matt Fozard)
- Logic for cancelling the “unsaved changes” check on form submission has been fixed to work cross-browser (Stephen Rice)
- The “unsaved changes” confirmation was erroneously shown on IE / Firefox when previewing a page with validation errors (Matt Westcott)
- The up / down / delete controls on the “Promoted search results” form no longer trigger a form submission (Matt Westcott)
- Opening preview window no longer performs user-agent sniffing, and now works correctly on IE11 (Matt Westcott)

- Tree paths are now correctly assigned when previewing a newly-created page underneath a parent with deleted children (Matt Westcott)
- Added BASE\_URL setting back to project template
- Clearing the search box in the page chooser now returns the user to the browse view (Matt Westcott)
- The above fix also fixed an issue where Internet Explorer got stuck in the search view upon opening the page chooser (Matt Westcott)

## 1.9.113 Wagtail 1.4.4 release notes

May 10, 2016

- *What's changed*

### What's changed

#### Translations

- New translation for Slovenian (Mitja Pagon)

#### Bug fixes

- The wagtailuserbar template tag now gracefully handles situations where the request object is not in the template context (Matt Westcott)
- Meta classes on StreamField blocks now handle multiple inheritance correctly (Tim Heap)
- Now user can upload images / documents only into permitted collection from choosers
- Keyboard shortcuts for save / preview on the page editor no longer incorrectly trigger the “unsaved changes” message (Jack Paine / Matt Westcott)
- Redirects no longer fail when both a site-specific and generic redirect exist for the same URL path (Nick Smith, João Luiz Lorencetti)
- Wagtail now checks that Group is registered with the Django admin before unregistering it (Jason Morrison)
- Previewing inaccessible pages no longer fails with ALLOWED\_HOSTS = ['\*'] (Robert Rollins)
- The submit button ‘spinner’ no longer activates if the form has client-side validation errors (Jack Paine, Matt Westcott)
- Overriding MESSAGE\_TAGS in project settings no longer causes messages in the Wagtail admin to lose their styling (Tim Heap)
- Border added around explorer menu to stop it blending in with StreamField block listing; also fixes invisible explorer menu in Firefox 46 (Alex Gleason)

## 1.9.114 Wagtail 1.4.3 release notes

April 4, 2016

- *What's changed*

### What's changed

#### Bug fixes

- Fixed regression introduced in 1.4.2 which caused Wagtail to query the database during a system check (Tim Heap)

## 1.9.115 Wagtail 1.4.2 release notes

March 1, 2016

- *What's changed*

### What's changed

#### Bug fixes

- Streamfields no longer break on validation error
- Number of validation errors in each tab in the editor is now correctly reported again
- Userbar now opens on devices with both touch and mouse (Josh Barr)
- `wagtail.wagtailadmin.wagtail_hooks` no longer calls `static` during app load, so you can use `ManifestStaticFilesStorage` without calling the `collectstatic` command
- Fixed crash on page save when a custom Page edit handler has been specified using the `edit_handler` attribute (Tim Heap)

## 1.9.116 Wagtail 1.4.1 release notes

March 17, 2016

- *What's changed*

## What's changed

### Bug fixes

- Fixed erroneous rendering of up arrow icons (Rob Moorman)

## 1.9.117 Wagtail 1.4 release notes

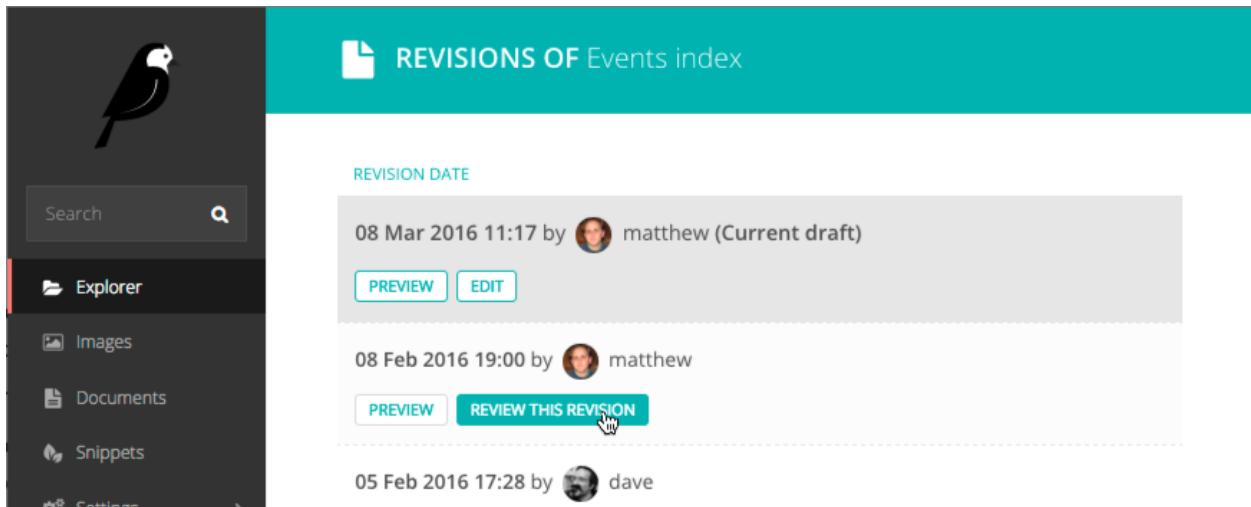
March 16, 2016

- [What's new](#)
- [Upgrade considerations](#)

Wagtail 1.4 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 8 months).

### What's new

#### Page revision management

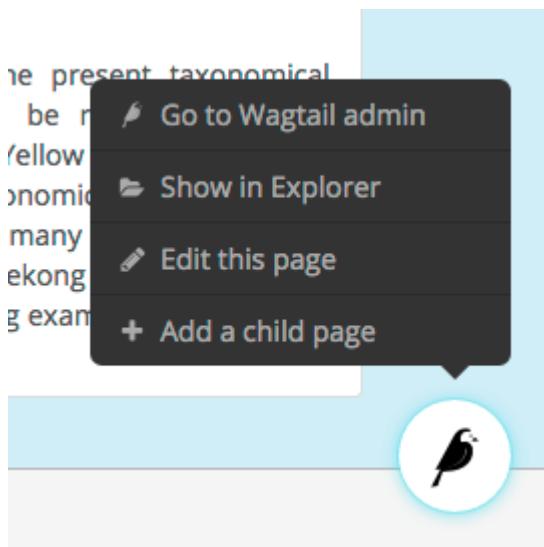


From the page editing interface, editors can now access a list of previous revisions of the page, and preview or roll back to any earlier revision.

## Collections for image / document organisation

Images and documents can now be organised into collections, set up by administrators through the Settings -> Collections menu item. User permissions can be set either globally (on the ‘Root’ collection) or on individual collections, allowing different user groups to keep their media items separated. Thank you to the University of South Wales for sponsoring this feature.

## Redesigned userbar



The Wagtail userbar (which gives editors quick access to the admin from the site frontend) has been redesigned, and no longer depends on an iframe. The new design allows more flexibility in label text, more configurable positioning to avoid overlapping with site navigation, and adds a new “Show in Explorer” option. This feature was developed by Thomas Winter and Gareth Price.

## Protection against unsaved changes

The page editor interface now produces a warning if the user attempts to navigate away while there are unsaved changes.

## Multiple document uploader

The “Add a document” interface now supports uploading multiple documents at once, in the same way as uploading images.

## Custom document models

The `Document` model can now be overridden using the new `WAGTAILDOCS_DOCUMENT_MODEL` setting. This works in the same way that `WAGTAILIMAGES_IMAGE_MODEL` works for `Image`.

## Removed django-compressor dependency

Wagtail no longer depends on the `django-compressor` library. While we highly recommend compressing and bundling the CSS and JavaScript on your sites, using `django-compressor` places additional installation and configuration demands on the developer, so this has now been optional.

## Minor features

- The page search interface now searches all fields instead of just the title (Kait Crawford)
- Snippets now support a custom `edit_handler` property; this can be used to implement a tabbed interface, for example. See [Customising the tabbed interface](#) (Mikalai Radchuk)
- Date/time pickers now respect the locale’s ‘first day of week’ setting (Peter Quade)
- Refactored the way forms are constructed for the page editor, to allow custom forms to be used
- Notification message on publish now indicates whether the page is being published now or scheduled for publication in future (Chris Rogers)
- Server errors when uploading images / documents through the chooser modal are now reported back to the user (Nigel Fletton)
- Added a hook `insert_global_admin_css` for inserting custom CSS throughout the admin backend (Tom Dyson)
- Added a hook `construct_explorer_page_queryset` for customising the set of pages displayed in the page explorer
- Page models now perform field validation, including testing slugs for uniqueness within a parent page, at the model level on saving
- Page slugs are now auto-generated at the model level on page creation if one has not been specified explicitly
- The `Page` model now has two new methods `get_site()` and `get_url_parts()` to aid with customising the page URL generation logic
- Upgraded jQuery to 2.2.1 (Charlie Choiniere)
- Multiple homepage summary items (`construct_homepage_summary_items` hook) now better vertically spaced (Nicolas Kuttler)
- Email notifications can now be sent in HTML format. See [WAGTAILADMIN\\_USER\\_PASSWORD\\_RESET\\_FORM](#) (Mike Dingjan)
- `StreamBlock` now has provision for throwing non-field-specific validation errors
- Wagtail now works with Willow 0.3, which supports auto-correcting the orientation of images based on EXIF data
- New translations for Hungarian, Swedish (Sweden) and Turkish

## Bug fixes

- Custom page managers no longer raise an error when used on an abstract model
- Wagtail’s migrations are now all reversible (Benjamin Bach)
- Deleting a page content type now preserves existing pages as basic Page instances, to prevent tree corruption
- The `Page.path` field is now explicitly given the “C” collation on PostgreSQL to prevent tree ordering issues when using a database created with the Slovak locale
- Wagtail’s compiled static assets are now put into the correct directory on Windows (Aarni Koskela)
- `ChooserBlock` now correctly handles models with primary keys other than `id` (alexpilot11)
- Fixed typo in Wistia oEmbed pattern (Josh Hurd)
- Added more accurate help text for the Administrator flag on user accounts (Matt Fozard)
- Tags added on the multiple image uploader are now saved correctly
- Documents created by a user are no longer deleted when the user is deleted
- Fixed a crash in `RedirectMiddleware` when a middleware class before `SiteMiddleware` returns a response (Josh Schneier)
- Fixed error retrieving the moderator list on pages that are covered by multiple moderator permission records (Matt Fozard)
- Ordering pages in the explorer by reverse ‘last updated’ time now puts pages with no revisions at the top
- WagtailTestUtils now works correctly on custom user models without a `username` field (Adam Bolfik)
- Logging in to the admin as a user with valid credentials but no admin access permission now displays an error message, rather than rejecting the user silently
- StreamBlock HTML rendering now handles non-ASCII characters correctly on Python 2 (Mikalai Radchuk)
- Fixed a bug preventing pages with a `OneToOneField` from being copied (Liam Brenner)
- SASS compilation errors during Wagtail development no longer cause exit of Gulp process, instead throws error to console and continues (Thomas Winter)
- Explorer page listing now uses specific page models, so that custom URL schemes defined on Page subclasses are respected
- Made settings menu clickable again in Firefox 46.0a2 (Juha Kujala)
- User management index view no longer assumes the presence of `username`, `first_name`, `last_name` and `email` fields on the user model (Eirik Krogstad)

## Upgrade considerations

### Removal of django-compressor

As Wagtail no longer installs django-compressor automatically as a dependency, you may need to make changes to your site’s configuration when upgrading. If your project is actively using django-compressor (that is, your site templates contain `{% compress %}` tags), you should ensure that your project’s requirements explicitly include django-compressor, rather than indirectly relying on Wagtail to install it. If you are not actively using django-compressor on your site, you should update your settings file to remove the line '`compressor`' from `INSTALLED_APPS`, and remove '`compressor.finders.CompressorFinder`' from `STATICFILES_FINDERS`.

## Page models now enforce field validation

In previous releases, field validation on Page models was only applied at the form level, meaning that creating pages directly at the model level would bypass validation. For example, if `NewsPage` is a Page model with a required `body` field, then code such as:

```
news_page = NewsPage(title="Hello", slug='hello')
parent_page = NewsIndex.objects.get()
parent_page.add_child(instance=news_page)
```

would create a page that does not comply with the validation rules. This is no longer possible, as validation is now enforced at the model level on `save()` and `save_revision()`; as a result, code that creates pages programmatically (such as unit tests, and import scripts) may need to be updated to ensure that it creates valid pages.

## 1.9.118 Wagtail 1.3.1 release notes

*January 5, 2016*

- *What's changed*

### What's changed

#### Bug fixes

- Applied workaround for failing `wagtailimages` migration on Django 1.8.8 / 1.9.1 with Postgres (see Django issue #26034)

## 1.9.119 Wagtail 1.3 release notes

*December 23, 2015*

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

### What's new

#### Django 1.9 support

Wagtail is now compatible with Django 1.9.

## Indexing fields across relations in Elasticsearch

Fields on related objects can now be indexed in Elasticsearch using the new `indexed.RelatedFields` declaration type:

```
class Book(models.Model, index.Indexed):
 ...

 search_fields = [
 index.SearchField('title'),
 index.FilterField('published_date'),

 index.RelatedFields('author', [
 index.SearchField('name'),
 index.FilterField('date_of_birth'),
]),
]

 # Search books where their author was born after 1950
 # Both the book title and the authors name will be searched
 >>> Book.objects.filter(author__date_of_birth__gt=date(1950, 1, 1)).search("Hello")
```

See: `index.RelatedFields`

## Cross-linked admin search UI

The search interface in the Wagtail admin now includes a toolbar to quickly switch between different search types - pages, images, documents and users. A new `register_admin_search_area` hook is provided for adding new search types to this toolbar.

## Minor features

- Added `WagtailPageTests`, a helper module to simplify writing tests for Wagtail sites. See [Testing your Wagtail site](#)
- Added system checks to check the `subpage_types` and `parent_page_types` attributes of page models
- Added `WAGTAIL_PASSWORD_RESET_ENABLED` setting to allow password resets to be disabled independently of the password management interface (John Draper)
- Submit for moderation notification emails now include the editor name (Denis Voskovitsov)
- Updated fonts for more comprehensive Unicode support
- Added `.alt` attribute to image renditions
- The default `src`, `width`, `height` and `alt` attributes can now be overridden by attributes passed to the `{% image %}` tag
- Added keyboard shortcuts for preview and save in the page editor
- Added Page methods `can_exist_under`, `can_create_at`, `can_move_to` for customising page type business rules
- `wagtailadmin.utils.send_mail` now passes extra keyword arguments to Django's `send_mail` function (Matthew Downey)

- `page_unpublish` signal is now fired for each page that was unpublished by a call to `PageQuerySet.unpublish()`
- Add `get_upload_to` method to `AbstractImage`, to allow overriding the default image upload path (Ben Emery)
- Notification emails are now sent per user (Matthew Downey)
- Added the ability to override the default manager on Page models
- Added an optional human-friendly `site_name` field to sites (Timo Rieber)
- Added a system check to warn developers who use a custom Wagtail build but forgot to build the admin css
- Added success message after updating image from the image upload view (Christian Peters)
- Added a `request.is_preview` variable for templates to distinguish between previewing and live (Denis Voskvtsov)
- ‘Pages’ link on site stats dashboard now links to the site homepage when only one site exists, rather than the root level
- Added support for chaining multiple image operations on the `{% image %}` tag (Christian Peters)
- New translations for Arabic, Latvian and Slovak

## Bug fixes

- Images and page revisions created by a user are no longer deleted when the user is deleted (Rich Atkinson)
- HTTP cache purge now works again on Python 2 (Mitchel Cabuloy)
- Locked pages can no longer be unpublished (Alex Bridge)
- Site records now implement `get_by_natural_key`
- Creating pages at the root level (and any other instances of the base `Page` model) now properly respects the `parent_page_types` setting
- Settings menu now opens correctly from the page editor and styleguide views
- `subpage_types / parent_page_types` business rules are now enforced when moving pages
- Multi-word tags on images and documents are now correctly preserved as a single tag (LKozlowski)
- Changed verbose names to start with lower case where necessary (Maris Serzans)
- Invalid images no longer crash the image listing (Maris Serzans)
- `MenuItem.url` parameter can now take a lazy URL (Adon Metcalfe, rayrayndwiga)
- Added missing translation tag to `InlinePanel` ‘Add’ button (jnns)
- Added missing translation tag to ‘Signing in...’ button text (Eugene MechanisM)
- Restored correct highlighting behaviour of rich text toolbar buttons
- Rendering a missing image through `ImageChooserBlock` no longer breaks the whole page (Christian Peters)
- Filtering by popular tag in the image chooser now works when using the database search backend

## Upgrade considerations

### Jinja2 template tag modules have changed location

Due to a change in the way template tags are imported in Django 1.9, it has been necessary to move the Jinja2 template tag modules from “templatetags” to a new location, “jinja2tags”. The correct configuration settings to enable Jinja2 templates are now as follows:

```
TEMPLATES = [
 # ...
 {
 'BACKEND': 'django.template.backends.jinja2.Jinja2',
 'APP_DIRS': True,
 'OPTIONS': {
 'extensions': [
 'wagtail.core.jinja2tags.core',
 'wagtail.wagtailadmin.jinja2tags.userbar',
 'wagtail.wagtailimages.jinja2tags.images',
],
 },
 },
]
```

See: [Jinja2 template support](#)

### ContentType-returning methods in wagtailcore are deprecated

The following internal functions and methods in `wagtail.wagtailcore.models`, which return a list of `ContentType` objects, have been deprecated. Any uses of these in your code should be replaced by the corresponding new function which returns a list of model classes instead:

- `get_page_types()` - replaced by `get_page_models()`
- `Page.clean_subpage_types()` - replaced by `Page.clean_subpage_models()`
- `Page.clean_parent_page_types()` - replaced by `Page.clean_parent_page_models()`
- `Page.allowed_parent_page_types()` - replaced by `Page.allowed_parent_page_models()`
- `Page.allowed_subpage_types()` - replaced by `Page.allowed_subpage_models()`

In addition, note that these methods now return page types that are marked as `is_creatable = False`, including the base `Page` class. (Abstract models are not included, as before.)

## 1.9.120 Wagtail 1.2 release notes

November 12, 2015

- [What's new](#)
- [Upgrade considerations](#)

## What's new

### Site settings module

Wagtail now includes a contrib module (previously available as the `wagtailsettings` package) to allow administrators to edit site-specific settings.

See: [Settings](#)

### Jinja2 support

The core templatetags (`pageurl`, `slugurl`, `image`, `richtext` and `wagtailuserbar`) are now compatible with Jinja2 so it's now possible to use Jinja2 as the template engine for your Wagtail site.

Note that the variable name `self` is reserved in Jinja2, and so Wagtail now provides alternative variable names where `self` was previously used: `page` to refer to page objects, and `value` to refer to StreamField blocks. All code examples in this documentation have now been updated to use the new variable names, for compatibility with Jinja2; however, users of the default Django template engine can continue to use `self`.

See: [Jinja2 template support](#)

### Site-specific redirects

You can now create redirects for a particular site using the admin interface.

### Search API improvements

Wagtail's image and document models now provide a `search` method on their QuerySets, making it easy to perform searches on filtered data sets. In addition, search methods now accept two new keyword arguments:

- `operator`, to determine whether multiple search terms will be treated as ‘or’ (any term may match) or ‘and’ (all terms must match);
- `order_by_relevance`, set to `True` (the default) to order by relevance or `False` to preserve the QuerySet’s original ordering.

See: [Searching](#)

### `max_num` and `min_num` parameters on inline panels

Inline panels now accept the optional parameters `max_num` and `min_num`, to specify the maximum / minimum number of child items that must exist in order for the page to be valid.

See: [Inline Panels and Model Clusters](#)

## get\_context on StreamField blocks

StreamField blocks now *provide a get\_context method* that can be overridden to pass additional variables to the block's template.

## Browsable API

The Wagtail API now incorporates the browsable front-end provided by Django REST Framework. Note that this must be enabled by adding '`rest_framework`' to your project's `INSTALLED_APPS` setting.

## Python 3.5 support

Wagtail now supports Python 3.5 when run in conjunction with Django 1.8.6 or later.

## Minor features

- WagtailRedirectMiddleware can now ignore the query string if there is no redirect that exactly matches it
- Order of URL parameters now ignored by redirect middleware
- Added SQL Server compatibility to image migration
- Added `class` attributes to Wagtail rich text editor buttons to aid custom styling
- Simplified `body_class` in default homepage template
- `page_published` signal now called with the revision object that was published
- Added a favicon to the admin interface, customisable by overriding the `branding_favicon` block (see *Custom branding*).
- Added spinner animations to long-running form submissions
- The `EMBEDLY_KEY` setting has been renamed to `WAGTAILEMBEDS_EMBEDLY_KEY`
- StreamField blocks are now added automatically, without showing the block types menu, if only one block type exists (Alex Gleason)
- The `first_published_at` and `latest_revision_created_at` fields on page models are now available as filter fields on search queries
- Wagtail admin now standardises on a single thumbnail image size, to reduce the overhead of creating multiple renditions
- Rich text fields now strip out HTML comments
- Page editor form now sets `enctype="multipart/form-data"` as appropriate, allowing `FileField` to be used on page models (Petr Vacha)
- Explorer navigation menu on a completely empty page tree now takes you to the root level, rather than doing nothing
- Added animation and fixed display issues when focusing a rich text field (Alex Gleason)
- Added a system check to warn if Pillow is compiled without JPEG / PNG support
- Page chooser now prevents users from selecting the root node where this would be invalid
- New translations for Dutch (Netherlands), Georgian, Swedish and Turkish (Turkey)

## Bug fixes

- Page slugs are no longer auto-updated from the page title if the page is already published
- Deleting a page permission from the groups admin UI does not immediately submit the form
- Wagtail userbar is shown on pages that do not pass a `page` variable to the template (e.g. because they override the `serve` method)
- `request.site` now set correctly on page preview when the page is not in the default site
- Project template no longer raises a deprecation warning (Maximilian Strauss)
- `PageManager.sibling_of(page)` and `PageManager.not_sibling_of(page)` now default to inclusive (i.e. page is considered a sibling of itself), for consistency with other sibling methods
- The “view live” button displayed after publishing a page now correctly reflects any changes made to the page slug (Ryan Pineo)
- API endpoints now accept and ignore the `_query` parameter used by jQuery for cache-busting
- Page slugs are no longer cut off when Unicode characters are expanded into multiple characters (Sævar Öfjörð Magnússon)
- Searching a specific page model while filtering it by either ID or tree position no longer raises an error (Ashia Zawaduk)
- Scrolling an over-long explorer menu no longer causes white background to show through (Alex Gleason)
- Removed jitter when hovering over StreamField blocks (Alex Gleason)
- Non-ASCII email addresses no longer throw errors when generating Gravatar URLs (Denis Voskvtsov, Kyle Stratis)
- Dropdown for `ForeignKey`s are now styled consistently (Ashia Zawaduk)
- Date choosers now appear on top of StreamField menus (Sergey Nikitin)
- Fixed a migration error that was raised when block-updating from 0.8 to 1.1+
- `Page.copy()` no longer breaks on models with a `ClusterTaggableManager` or `ManyToManyField`
- Validation errors when inserting an embed into a rich text area are now reported back to the editor

## Upgrade considerations

### `PageManager.sibling_of(page)` and `PageManager.not_sibling_of(page)` have changed behaviour

In previous versions of Wagtail, the `sibling_of` and `not_sibling_of` methods behaved inconsistently depending on whether they were called on a manager (e.g. `Page.objects.sibling_of(some_page)` or `EventPage.objects.sibling_of(some_page)`) or a QuerySet (e.g. `Page.objects.all().sibling_of(some_page)` or `EventPage.objects.live().sibling_of(some_page)`).

Previously, the manager methods behaved as *exclusive* by default; that is, they did not count the passed-in page object as a sibling of itself:

```
>>> event_1 = EventPage.objects.get(title='Event 1')
>>> EventPage.objects.sibling_of(event_1)
[<EventPage: Event 2>] # OLD behaviour: Event 1 is not considered a sibling of itself
```

This has now been changed to be *inclusive* by default; that is, the page is counted as a sibling of itself:

```
>>> event_1 = EventPage.objects.get(title='Event 1')
>>> EventPage.objects.sibling_of(event_1)
[<EventPage: Event 1>, <EventPage: Event 2>] # NEW behaviour: Event 1 is considered a
 ↳ sibling of itself
```

If the call to `sibling_of` or `not_sibling_of` is chained after another QuerySet method - such as `all()`, `filter()` or `live()` - behaviour is unchanged; this behaves as *inclusive*, as it did in previous versions:

```
>>> event_1 = EventPage.objects.get(title='Event 1')
>>> EventPage.objects.all().sibling_of(event_1)
[<EventPage: Event 1>, <EventPage: Event 2>] # OLD and NEW behaviour
```

If your project includes queries that rely on the old (exclusive) behaviour, this behaviour can be restored by adding the keyword argument `inclusive=False`:

```
>>> event_1 = EventPage.objects.get(title='Event 1')
>>> EventPage.objects.sibling_of(event_1, inclusive=False)
[<EventPage: Event 2>] # passing inclusive=False restores the OLD behaviour
```

### Image.search and Document.search methods are deprecated

The `Image.search` and `Document.search` methods have been deprecated in favour of the new QuerySet-based search mechanism - see [Searching Images, Documents and custom models](#). Code using the old `search` methods should be updated to search on QuerySets instead; for example:

```
Image.search("Hello", filters={'uploaded_by_user': user})
```

can be rewritten as:

```
Image.objects.filter(uploaded_by_user=user).search("Hello")
```

### Wagtail API requires adding rest\_framework to INSTALLED\_APPS

If you have the Wagtail API (`wagtail.contrib.wagtailapi`) enabled, you must now add '`rest_framework`' to your project's `INSTALLED_APPS` setting. In the current version the API will continue to function without this app, but the browsable front-end will not be available; this ability will be dropped in a future release.

### Page.get\_latest\_revision\_as\_page() now returns live page object when there are no draft changes

If you have any application code that makes direct updates to page data, at the model or database level, be aware that the way these edits are reflected in the page editor has changed.

Previously, the `get_latest_revision_as_page` method - used by the page editor to return the current page revision for editing - always retrieved data from the page's revision history. Now, it will only do so if the page has unpublished changes (i.e. the page is in `live + draft` state) - pages which have received no draft edits since being published will return the page's live data instead.

As a result, any changes made directly to a live page object will be immediately reflected in the editor without needing to update the latest revision record (but note, the old behaviour is still used for pages in `live + draft` state).

## 1.9.121 Wagtail 1.1 release notes

September 15, 2015

- [What's new](#)
- [Upgrade considerations](#)

### What's new

#### **specific() method on PageQuerySet**

Usually, an operation that retrieves a QuerySet of pages (such as `homepage.get_children()`) will return them as basic Page instances, which only include the core page data such as title. The `specific()` method (e.g. `homepage.get_children().specific()`) now allows them to be retrieved as their most specific type, using the minimum number of queries.

#### **“Promoted search results” has moved into its own module**

Previously, this was implemented in `wagtailsearch` but now has been moved into a separate module: `wagtail.contrib.wagtailsearchpromotions`

#### **Atomic rebuilding of Elasticsearch indexes**

The Elasticsearch search backend now accepts an experimental `ATOMIC_REBUILD` flag which ensures that the existing search index continues to be available while the `update_index` task is running. See [ATOMIC\\_REBUILD](#).

#### **The `wagtailapi` module now uses Django REST Framework**

The `wagtailapi` module is now built on Django REST Framework and it now also has a library of serialisers that you can use in your own REST Framework based APIs. No user-facing changes have been made.

We hope to support more REST framework features, such as a browsable API, in future releases.

#### **Permissions fixes in the admin interface**

A number of inconsistencies around permissions in the admin interface were fixed in this release:

- Removed all permissions for “User profile” (not used)
- Removed “delete” permission for Images and documents (not used)
- Users can now access images and documents when they only have the “change” permission (previously required “add” permission as well)
- Permissions for Users now taken from custom user model, if set (previously always used permissions on Django’s builtin User model)
- Groups and Users now respond consistently to their respective “add”, “change” and “delete” permissions

## Searchable snippets

Snippets that inherit from `wagtail.wagtailsearch.index.Indexed` are now given a search box on the snippet chooser and listing pages. See [Making snippets searchable](#).

## Minor features

- Implemented deletion of form submissions
- Implemented pagination in the page chooser modal
- Changed INSTALLED\_APPS in project template to list apps in precedence order
- The `{% image %}` tag now supports filters on the `image` variable, e.g. `{% image primary_img|default:secondary_img width=500 %}`
- Moved the style guide menu item into the Settings sub-menu
- Search backends can now be specified by module (e.g. `wagtail.wagtailsearch.backends.elasticsearch`), rather than a specific class (`wagtail.wagtailsearch.backends.elasticsearch.ElasticSearch`)
- Added `descendant_of` filter to the API
- Added optional directory argument to “wagtail start” command
- Non-superusers can now view/edit/delete sites if they have the correct permissions
- Image file size is now stored in the database, to avoid unnecessary filesystem lookups
- Page URL lookups hit the cache/database less often
- Updated URLs within the admin backend to use namespaces
- The `update_index` task now indexes objects in batches of 1000, to indicate progress and avoid excessive memory use
- Added database indexes on `PageRevision` and `Image` to improve performance on large sites
- Search in page chooser now uses Wagtail’s search framework, to order results by relevance
- `PageChooserPanel` now supports passing a list (or tuple) of accepted page types
- The snippet type parameter of `SnippetChooserPanel` can now be omitted, or passed as a model name string rather than a model class
- Added aliases for the `self` template variable to accommodate Jinja as a templating engine: `page` for pages, `field_panel` for field panels / edit handlers, and `value` for blocks
- Added signposting text to the explorer to steer editors away from creating pages at the root level unless they are setting up new sites
- “Clear choice” and “Edit this page” buttons are no longer shown on the page field of the group page permissions form
- Altered styling of stream controls to be more like all other buttons
- Added ability to mark page models as not available for creation using the flag `is_creatable`; pages that are abstract Django models are automatically made non-creatable
- New translations for Norwegian Bokmål and Icelandic

## Bug fixes

- Text areas in the non-default tab of the page editor now resize to the correct height
- Tabs in “insert link” modal in the rich text editor no longer disappear (Tim Heap)
- H2 elements in rich text fields were accidentally given a click() binding when put inside a collapsible multi field panel
- The `wagtailimages` module is now compatible with remote storage backends that do not allow reopening closed files
- Search no longer crashes when auto-indexing a model that doesn’t have an `id` field
- The `wagtailfrontendcache` module’s HTTP backend has been rewritten to reliably direct requests to the configured cache hostname
- Resizing single pixel images with the “fill” filter no longer raises `ZeroDivisionError` or “tile cannot extend outside image”
- The `QuerySet` returned from `search` operations when using the database search backend now correctly preserves additional properties of the original query, such as `prefetch_related` / `select_related`
- Responses from the external image URL generator are correctly marked as streaming and will no longer fail when used with Django’s cache middleware
- Page copy now works with pages that use multiple inheritance
- Form builder pages now pick up template variables defined in the `get_context` method
- When copying a page, IDs of child objects within page revision records were not remapped to the new objects; this would cause those objects to be lost from the original page when editing the new one
- Newly added redirects now take effect on all sites, rather than just the site that the Wagtail admin backend was accessed through
- Add user form no longer throws a hard error on validation failure

## Upgrade considerations

### “Promoted search results” no longer in `wagtailsearch`

This feature has moved into a contrib module so is no longer enabled by default.

To re-enable it, add `wagtail.contrib.wagtailsearchpromotions` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
 ...
 'wagtail.contrib.wagtailsearchpromotions',
 ...
]
```

If you have references to the `wagtail.wagtailsearch.models.EditorsPick` model in your project, you will need to update these to point to the `wagtail.contrib.wagtailsearchpromotions.models.SearchPromotion` model instead.

If you created your project using the `wagtail start` command with Wagtail 1.0, you will probably have references to this model in the `search/views.py` file.

### **is\_abstract flag on page models has been replaced by is\_creatable**

Previous versions of Wagtail provided an undocumented `is_abstract` flag on page models - not to be confused with Django's `abstract` Meta flag - to indicate that it should not be included in the list of available page types for creation. (Typically this would be used on model classes that were designed to be subclassed to create new page types, rather than used directly.) To avoid confusion with Django's distinct concept of abstract models, this has now been replaced by a new flag, `is_creatable`.

If you have used `is_abstract = True` on any of your models, you should now change this to `is_creatable = False`.

It is not necessary to include this flag if the model is abstract in the Django sense (i.e. it has `abstract = True` in the model's `Meta` class), since it would never be valid to create pages of that type.

## **1.9.122 Wagtail 1.0 release notes**

*July 16, 2015*

- *What's changed*
- *Upgrade considerations*

### **What's changed**

#### **StreamField - a field type for freeform content**

StreamField provides an editing model for freeform content such as blog posts and news stories, allowing diverse content types such as text, images, headings, video and more specialised types such as maps and charts to be mixed in any order. See [How to use StreamField for mixed content](#).

#### **Wagtail API - A RESTful API for your Wagtail site**

When installed, the new Wagtail API module provides a RESTful web API to your Wagtail site. You can use this for accessing your raw field content for your sites pages, images and documents in JSON format.

#### **MySQL support**

Wagtail now officially supports MySQL as a database backend.

#### **Django 1.8 support**

Wagtail now officially supports running under Django 1.8.

## Vanilla project template

The built-in project template is more like the Django built-in one with several Wagtail-specific additions. It includes bare minimum settings and two apps (home and search).

### Minor changes

- Dropped Django 1.6 support
- Dropped Python 2.6 and 3.2 support
- Dropped Elasticsearch 0.90.x support
- Removed dependency on `libsass`
- Users without usernames can now be created and edited in the admin interface
- Added new translations for Croatian and Finnish

### Core

- The Page model now records the date/time that a page was first published, as the field `first_published_at`
- Increased the maximum length of a page slug from 50 to 255 characters
- Added hooks `register_rich_text_embed_handler` and `register_rich_text_link_handler` for customising link / embed handling within rich text fields
- Page URL paths can now be longer than 255 characters

### Admin

#### UI

- Improvements to the layout of the left-hand menu footer
- Menu items of custom apps are now highlighted when being used
- Added thousands separator for counters on dashboard
- Added contextual links to admin notification messages
- When copying pages, it is now possible to specify a place to copy to
- Added pagination to the snippets listing and chooser
- Page / document / image / snippet choosers now include a link to edit the chosen item
- Plain text fields in the page editor now use auto-expanding text areas
- Added “Add child page” button to admin userbar
- Added update notifications (See: [Wagtail update notifications](#))

#### Page editor

- JavaScript includes in the admin backend have been moved to the HTML header, to accommodate form widgets that render inline scripts that depend on libraries such as jQuery
- The external link chooser in rich text areas now accepts URLs of the form ‘/some/local/path’, to allow linking to non-Wagtail-controlled URLs within the local site

- Bare text entered in rich text areas is now automatically wrapped in a paragraph element

### Edit handlers API

- `FieldPanel` now accepts an optional `widget` parameter to override the field's default form widget
- Page model fields without a `FieldPanel` are no longer displayed in the form
- No longer need to specify the base model on `InlinePanel` definitions
- Page classes can specify an `edit_handler` property to override the default Content / Promote / Settings tabbed interface. See [Customising the tabbed interface](#).

### Other admin changes

- SCSS files in wagtailadmin now use absolute imports, to permit overriding by user stylesheets
- Removed the dependency on `LOGIN_URL` and `LOGIN_REDIRECT_URL` settings
- Password reset view names namespaced to wagtailadmin
- Removed the need to add permission check on admin views (now automated)
- Reversing `django.contrib.auth.admin.login` will no longer lead to Wagtails login view (making it easier to have frontend login views)
- Added cache-control headers to all admin views. This allows Varnish/Squid/CDN to run on vanilla settings in front of a Wagtail site
- Date / time pickers now consistently use times without seconds, to prevent JavaSript behaviour glitches when focusing / unfocusing fields
- Added hook `construct_homepage_summary_items` for customising the site summary panel on the admin homepage
- Renamed the `construct_wagtail_edit_bird` hook to `construct_wagtail_userbar`
- ‘static’ template tags are now used throughout the admin templates, in place of `STATIC_URL`

### Docs

- Support for `django-sendfile` added
- Documents now served with correct mime-type
- Support for `If-Modified-Since` HTTP header

### Search

- Search view accepts “page” GET parameter in line with pagination
- Added `AUTO_UPDATE` flag to search backend settings to enable/disable automatically updating the search index on model changes

## Routable pages

- Added a new decorator-based syntax for RoutablePage, compatible with Django 1.8

## Bug fixes

- The document\_served signal now correctly passes the Document class as `sender` and the document as `instance`
- Image edit page no longer throws `OSSError` when the original image is missing
- Collapsible blocks stay open on any form error
- Document upload modal no longer switches tabs on form errors
- `with_metaclass` is now imported from Django's bundled copy of the `six` library, to avoid errors on Mac OS X from an outdated system copy of the library being imported

## Upgrade considerations

### Support for older Django/Python/Elasticsearch versions dropped

This release drops support for Django 1.6, Python 2.6/3.2 and Elasticsearch 0.90.x. Please make sure these are updated before upgrading.

If you are upgrading from Elasticsearch 0.90.x, you may also need to update the `elasticsearch` pip package to a version greater than 1.0 as well.

### Wagtail version upgrade notifications are enabled by default

Starting from Wagtail 1.0, the admin dashboard will (for admin users only) perform a check to see if newer releases are available. This also provides the Wagtail team with the hostname of your Wagtail site. If you'd rather not receive update notifications, or if you'd like your site to remain unknown, you can disable it by adding this line to your settings file:

```
WAGTAIL_ENABLE_UPDATE_CHECK = False
```

### InlinePanel definitions no longer need to specify the base model

In previous versions of Wagtail, inline child blocks on a page or snippet were defined using a declaration like:

```
InlinePanel(HomePage, 'carousel_items', label="Carousel items")
```

It is no longer necessary to pass the base model as a parameter, so this declaration should be changed to:

```
InlinePanel('carousel_items', label="Carousel items")
```

The old format is now deprecated; all existing `InlinePanel` declarations should be updated to the new format.

## Custom image models should now set the `admin_form_fields` attribute

Django 1.8 now requires that all the fields in a `ModelForm` must be defined in its `Meta.fields` attribute.

As Wagtail uses Django's `ModelForm` for creating image model forms, we've added a new attribute called `admin_form_fields` that should be set to a tuple of field names on the image model.

See [Custom image models](#) for an example.

## You no longer need `LOGIN_URL` and `LOGIN_REDIRECT_URL` to point to Wagtail admin.

If you are upgrading from an older version of Wagtail, you probably want to remove these from your project settings.

Previously, these two settings needed to be set to `wagtailadmin_login` and `wagtailadmin_dashboard` respectively or Wagtail would become very tricky to log in to. This is no longer the case and Wagtail should work fine without them.

## RoutablePage now uses decorator syntax for defining views

In previous versions of Wagtail, page types that used the `RoutablePageMixin` had endpoints configured by setting their `subpage_urls` attribute to a list of urls with view names. This will not work on Django 1.8 as view names can no longer be passed into a url (see: <https://docs.djangoproject.com/en/stable/releases/1.8/#django-conf-urls-patterns>).

Wagtail 1.0 introduces a new syntax where each view function is annotated with a `@route` decorator - see [RoutablePageMixin](#).

The old `subpage_urls` convention will continue to work on Django versions prior to 1.8, but this is now deprecated; all existing `RoutablePage` definitions should be updated to the decorator-based convention.

## Upgrading from the external `wagtailapi` module.

If you were previously using the external `wagtailapi` module (which has now become `wagtail.contrib.wagtailapi`). Please be aware of the following backwards-incompatible changes:

### 1. Representation of foreign keys has changed

Foreign keys were previously represented by just the value of their primary key. For example:

```
"feed_image": 1
```

This has now been changed to add some `meta` information:

```
"feed_image": {
 "id": 1,
 "meta": {
 "type": "wagtailimages.Image",
 "detail_url": "http://api.example.com/api/v1/images/1/"
 }
}
```

### 2. On the page detail view, the “parent” field has been moved out of meta

Previously, there was a “parent” field in the “meta” section on the page detail view:

```
{
 "id": 10,
 "meta": {
 "type": "demo.BlogPage",
 "parent": 2
 },

 ...
}
```

This has now been moved to the top level. Also, the above change to how foreign keys are represented applies to this field too:

```
{
 "id": 10,
 "meta": {
 "type": "demo.BlogPage"
 },
 "parent": {
 "id": 2,
 "meta": {
 "type": "demo.BlogIndexPage"
 }
 },

 ...
}
```

### Celery no longer automatically used for sending notification emails

Previously, Wagtail would try to use Celery whenever the `djcelery` module was installed, even if Celery wasn't actually set up. This could cause a very hard to track down problem where notification emails would not be sent so this functionality has now been removed.

If you would like to keep using Celery for sending notification emails, have a look at: [django-celery-email](#)

## **Login/Password reset views renamed**

It was previously possible to reverse the Wagtail login view using `django.contrib.auth.views.login`. This is no longer possible. Update any references to `wagtailadmin_login`.

Password reset view name has changed from `password_reset` to `wagtailadmin_password_reset`.

## **JavaScript includes in admin backend have been moved**

To improve compatibility with third-party form widgets, pages within the Wagtail admin backend now output their JavaScript includes in the HTML header, rather than at the end of the page. If your project extends the admin backend (through the `register_admin_menu_item` hook, for example) you will need to ensure that all associated JavaScript code runs correctly from the new location. In particular, any code that accesses HTML elements will need to be contained in an ‘onload’ handler (e.g. jQuery’s `$(document).ready()`).

## **EditHandler internal API has changed**

While it is not an official Wagtail API, it has been possible for Wagtail site implementers to define their own `EditHandler` subclasses for use in panel definitions, to customise the behaviour of the page / snippet editing forms. If you have made use of this facility, you will need to update your custom `EditHandlers`, as this mechanism has been refactored (to allow `EditHandler` classes to keep a persistent reference to their corresponding model). If you have only used Wagtail’s built-in panel types (`FieldPanel`, `InlinePanel`, `PageChooserPanel` and so on), you are unaffected by this change.

Previously, functions like `FieldPanel` acted as ‘factory’ functions, where a call such as `FieldPanel('title')` constructed and returned an `EditHandler` subclass tailored to work on a ‘title’ field. These functions now return an object with a `bind_to_model` method instead; the `EditHandler` subclass can be obtained by calling this with the model class as a parameter. As a guide to updating your custom `EditHandler` code, you may wish to refer to the relevant change to the Wagtail codebase.

## **chooser\_panel templates are obsolete**

If you have added your own custom admin views to the Wagtail admin (e.g. through the `register_admin_urls` hook), you may have used one of the following template includes to incorporate a chooser element for pages, documents, images or snippets into your forms:

- `wagtailadmin/edit_handlers/chooser_panel.html`
- `wagtailadmin/edit_handlers/page_chooser_panel.html`
- `wagtaidocs/edit_handlers/document_chooser_panel.html`
- `wagtailimages/edit_handlers/image_chooser_panel.html`
- `wagtailsnippets/edit_handlers/snippet_chooser_panel.html`

All of these templates are now deprecated. Wagtail now provides a set of Django form widgets for this purpose - `AdminPageChooser`, `AdminDocumentChooser`, `AdminImageChooser` and `AdminSnippetChooser` - which can be used in place of the `HiddenInput` widget that these form fields were previously using. The field can then be rendered using the regular `wagtailadmin/shared/field.html` or `wagtailadmin/shared/field_as_li.html` template.

## document\_served signal arguments have changed

Previously, the `document_served` signal (which is fired whenever a user downloads a document) passed the document instance as the `sender`. This has now been changed to correspond the behaviour of Django's built-in signals; `sender` is now the `Document` class, and the document instance is passed as the argument `instance`. Any existing signal listeners that expect to receive the document instance in `sender` must now be updated to check the `instance` argument instead.

## Custom image models must specify an `admin_form_fields` list

Previously, the forms for creating and editing images followed Django's default behaviour of showing all fields defined on the model; this would include any custom fields specific to your project that you defined by subclassing `AbstractImage` and setting `WAGTAILIMAGES_IMAGE_MODEL`. This behaviour is risky as it may lead to fields being unintentionally exposed to the user, and so Django has deprecated this, for removal in Django 1.8. Accordingly, if you create your own custom subclass of `AbstractImage`, you must now provide an `admin_form_fields` property, listing the fields that should appear on the image creation / editing form - for example:

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

class MyImage(AbstractImage):
 photographer = models.CharField(max_length=255)
 has_legal_approval = models.BooleanField()

 admin_form_fields = Image.admin_form_fields + ['photographer']
```

## construct\_wagtail\_edit\_bird hook has been renamed

Previously you could customise the Wagtail userbar using the `construct_wagtail_edit_bird` hook. The hook has been renamed to `construct_wagtail_userbar`.

The old hook is now deprecated; all existing `construct_wagtail_edit_bird` declarations should be updated to the new hook.

## IMAGE\_COMPRESSION\_QUALITY setting has been renamed

The `IMAGE_COMPRESSION_QUALITY` setting, which determines the quality of saved JPEG images as a value from 1 to 100, has been renamed to `WAGTAILIMAGES_JPEG_QUALITY`. If you have used this setting, please update your settings file accordingly.

## 1.9.123 Wagtail 0.8.10 release notes

*September 16, 2015*

- *What's changed*

## What's changed

### Bug fixes

- When copying a page, IDs of child objects within page revision records were not remapped to the new objects; this would cause those objects to be lost from the original page when editing the new one
- Search no longer crashes when auto-indexing a model that doesn't have an id field (Scot Hacker)
- Resizing single pixel images with the “fill” filter no longer raises `ZeroDivisionError` or “tile cannot extend outside image”

## 1.9.124 Wagtail 0.8.8 release notes

June 18, 2015

- *What's changed*

## What's changed

### Bug fixes

- Form builder no longer raises a `TypeError` when submitting unchecked boolean field
- Image upload form no longer breaks when using i10n thousand separators
- Multiple image uploader now escapes HTML in filenames
- Retrieving an individual item from a sliced `BaseSearchResults` object now properly takes the slice offset into account
- Removed dependency on `unicodecsv` which fixes a crash on Python 3
- Submitting unicode text in form builder form no longer crashes with `UnicodeEncodeError` on Python 2
- Creating a proxy model from a `Page` class no longer crashes in the system check
- Unrecognised embed URLs passed to the `|embed` filter no longer cause the whole page to crash with an `EmbedNotFoundException`
- Underscores no longer get stripped from page slugs

## 1.9.125 Wagtail 0.8.7 release notes

April 29, 2015

- *What's changed*

## What's changed

### Bug fixes

- wagtailfrontendcache no longer tries to purge pages that are not in a site
- The contents of <div> elements in the rich text editor were not being whitelisted
- Due to the above issue, embeds/images in a rich text field would sometimes be saved into the database in their editor representation
- RoutablePage now prevents `subpage_urls` from being defined as a property, which would cause a memory leak
- Added validation to prevent pages being created with only whitespace characters in their title fields
- Users are no longer logged out on changing password when SessionAuthenticationMiddleware (added in Django 1.7) is in use
- Added a workaround for a Python / Django issue that prevented documents with certain non-ASCII filenames from being served

## 1.9.126 Wagtail 0.8.6 release notes

March 10, 2015

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

## What's new

### Minor features

- Translations updated, including new translations for Czech, Italian and Japanese
- The “fixtree” command can now delete orphaned pages

### Bug fixes

- django-taggit library updated to 0.12.3, to fix a bug with migrations on SQLite on Django 1.7.2 and above (<https://github.com/alex/django-taggit/issues/285>)
- Fixed a bug that caused children of a deleted page to not be deleted if they had a different type

## Upgrade considerations

### Orphaned pages may need deleting

This release fixes a bug with page deletion introduced in 0.8, where deleting a page with child pages will result in those child pages being left behind in the database (unless the child pages are of the same type as the parent). This may cause errors later on when creating new pages in the same position. To identify and delete these orphaned pages, it is recommended that you run the following command (from the project root) after upgrading to 0.8.6:

```
$./manage.py fixtree
```

This will output a list of any orphaned pages found, and request confirmation before deleting them.

Since this now makes `fixtree` an interactive command, a `./manage.py fixtree --noinput` option has been added to restore the previous non-interactive behaviour. With this option enabled, deleting orphaned pages is always skipped.

## 1.9.127 Wagtail 0.8.5 release notes

February 17, 2015

- [What's new](#)

### What's new

#### Bug fixes

- On adding a new page, the available page types are ordered by the displayed verbose name
- Active admin submenus were not properly closed when activating another
- `get_sitemap_urls` is now called on the specific page class so it can now be overridden
- (Firefox and IE) Fixed preview window hanging and not refocusing when “Preview” button is clicked again
- Storage backends that return raw `ContentFile` objects are now handled correctly when resizing images
- Punctuation characters are no longer stripped when performing search queries
- When adding tags where there were none before, it is now possible to save a single tag with multiple words in it
- `richtext` template tag no longer raises `TypeError` if `None` is passed into it
- Serving documents now uses a streaming HTTP response and will no longer break Django’s cache middleware
- User admin area no longer fails in the presence of negative user IDs (as used by `django-guardian`’s default settings)
- Password reset emails now use the `BASE_URL` setting for the reset URL
- `BASE_URL` is now included in the project template’s default settings file

## 1.9.128 Wagtail 0.8.4 release notes

December 4, 2014

- [What's new](#)

### What's new

### Bug fixes

- It is no longer possible to have the explorer and settings menu open at the same time
- Page IDs in page revisions were not updated on page copy, causing subsequent edits to be committed to the original page instead
- Copying a page now creates a new page revision, ensuring that changes to the title/slug are correctly reflected in the editor (and also ensuring that the user performing the copy is logged)
- Prevent a race condition when creating Filter objects
- On adding a new page, the available page types are ordered by the displayed verbose name

## 1.9.129 Wagtail 0.8.3 release notes

November 18, 2014

- [What's new](#)
- [Upgrade considerations](#)

### What's new

### Bug fixes

- Added missing jQuery UI sprite files, causing collectstatic to throw errors (most reported on Heroku)
- Page system check for on\_delete actions of ForeignKeys was throwing false positives when page class descends from an abstract class (Alejandro Giacometti)
- Page system check for on\_delete actions of ForeignKeys now only raises warnings, not errors
- Fixed a regression where form builder submissions containing a number field would fail with a JSON serialisation error
- Resizing an image with a focal point equal to the image size would result in a divide-by-zero error
- Focal point indicator would sometimes be positioned incorrectly for small or thin images
- Fix: Focal point chooser background colour changed to grey to make working with transparent images easier
- Elasticsearch configuration now supports specifying HTTP authentication parameters as part of the URL, and defaults to ports 80 (HTTP) and 443 (HTTPS) if port number not specified
- Fixed a TypeError when previewing pages that use RoutablePageMixin

- Rendering image with missing file in rich text no longer crashes the entire page
- IOErrors thrown by underlying image libraries that are not reporting a missing image file are no longer caught
- Fix: Minimum Pillow version bumped to 2.6.1 to work around a crash when using images with transparency
- Fix: Images with transparency are now handled better when being used in feature detection

## **Upgrade considerations**

### **Port number must be specified when running Elasticsearch on port 9200**

In previous versions, an Elasticsearch connection URL in `WAGTAILSEARCH_BACKENDS` without an explicit port number (e.g. `http://localhost/`) would be treated as port 9200 (the Elasticsearch default) whereas the correct behaviour would be to use the default http/https port of 80/443. This behaviour has now been fixed, so sites running Elasticsearch on port 9200 must now specify this explicitly - e.g. `http://localhost:9200`. (Projects using the default settings, or the settings given in the Wagtail documentation, are unaffected.)

## **1.9.130 Wagtail 0.8.1 release notes**

*November 5, 2014*

- *What's new*

### **What's new**

### **Bug fixes**

- Fixed a regression where images would fail to save when feature detection is active

## **1.9.131 Wagtail 0.8 release notes**

*November 5, 2014*

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

Wagtail 0.8 is designated a Long Term Support (LTS) release. Long Term Support releases will continue to receive maintenance updates as necessary to address security and data-loss related issues, up until the next LTS release (typically a period of 8 months).

## What's new

### Minor features

- Page operations (creation, publishing, copying etc) are now logged via Python’s logging framework; to configure this, add a logger entry for ‘wagtail’ or ‘wagtail.core’ to the LOGGING setup in your settings file.
- The save button on the page edit page now redirects the user back to the edit page instead of the explorer
- Signal handlers for wagtail.wagtailsearch and wagtail.contrib.wagtailfrontendcache are now automatically registered when using Django 1.7 or above.
- Added a Django 1.7 system check to ensure that foreign keys from Page models are set to `on_delete=SET_NULL`, to prevent inadvertent (and tree-breaking) page deletions
- Improved error reporting on image upload, including ability to set a maximum file size via a new setting `WAGTAILIMAGES_MAX_UPLOAD_SIZE`
- The external image URL generator now keeps persistent image renditions, rather than regenerating them on each request, so it no longer requires a front-end cache.
- Added Dutch translation

### Bug fixes

- Replaced references of `.username` with `.get_username()` on users for better custom user model support
- Unpinned dependency versions for six and requests to help prevent dependency conflicts
- Fixed `TypeError` when getting embed HTML with oembed on Python 3
- Made HTML whitelisting in rich text fields more robust at catching disallowed URL schemes such as `jav\tascript:`
- `created_at` timestamps on page revisions were not being preserved on page copy, causing revisions to get out of sequence
- When copying pages recursively, revisions of sub-pages were being copied regardless of the `copy_revisions` flag
- Updated the migration dependencies within the project template to ensure that Wagtail’s own migrations consistently apply first
- The cache of site root paths is now cleared when a site is deleted
- Search indexing now prevents pages from being indexed multiple times, as both the base Page model and the specific subclass
- Search indexing now avoids trying to index abstract models
- Fixed references to “`username`” in login form help text for better custom user model support
- Later items in a model’s `search_field` list now consistently override earlier items, allowing subclasses to redefine rules from the parent
- Image uploader now accepts JPEG images that PIL reports as being in MPO format
- Multiple checkbox fields on form-builder forms did not correctly save multiple values
- Editing a page’s slug and saving it without publishing could sometimes cause the URL paths of child pages to be corrupted

- `latest_revision_created_at` was being cleared on page publish, causing the page to drop to the bottom of explorer listings
- Searches on `partial_match` fields were wrongly applying prefix analysis to the search query as well as the document (causing e.g. a query for “water” to match against “wagtail”)

## Upgrade considerations

### Corrupted URL paths may need fixing

This release fixes a bug in Wagtail 0.7 where editing a parent page’s slug could cause the URL paths of child pages to become corrupted. To ensure that your database does not contain any corrupted URL paths, it is recommended that you run `./manage.py set_url_paths` after upgrading.

### Automatic registration of signal handlers (Django 1.7+)

Signal handlers for the `wagtailsearch` core app and `wagtailfrontendcache` contrib app are automatically registered when using Django 1.7. Calls to `register_signal_handlers` from your `urls.py` can be removed.

### Change to search API when using database backend

When using the database backend, calling `search` (either through `Page.objects.search()` or on the backend directly) will now return a `SearchResults` object rather than a Django `QuerySet` to make the database backend work more like the Elasticsearch backend.

This change shouldn’t affect most people as `SearchResults` behaves very similarly to `QuerySet`. But it may cause issues if you are calling `QuerySet` specific methods after calling `.search()`. Eg: `Page.objects.search("Hello").filter(foo="Bar")` (in this case, `.filter()` should be moved before `.search()` and it would work as before).

### Removal of validate\_image\_format from custom image model migrations (Django 1.7+)

If your project is running on Django 1.7, and you have defined a custom image model (by extending the `wagtailimages.AbstractImage` class), the migration that creates this model will probably have a reference to `wagtail.wagtailimages.utils.validators.validate_image_format`. This module has now been removed, which will cause `manage.py migrate` to fail with an `ImportError` (even if the migration has already been applied). You will need to edit the migration file to remove the line:

```
import wagtail.wagtailimages.utils.validators
```

and the `validators` attribute of the ‘file’ field - that is, the line:

```
('file', models.ImageField(upload_to=wagtail.wagtailimages.models.get_upload_to,
 width_field='width', height_field='height',
 validators=[wagtail.wagtailimages.utils.validators.validate_image_format],
 verbose_name='File')),
```

should become:

```
('file', models.ImageField(upload_to=wagtail.wagtailimages.models.get_upload_to,
 width_field='width', height_field='height', verbose_name='File')),
```

## 1.9.132 Wagtail 0.7 release notes

October 9, 2014

- [What's new](#)
- [Upgrade considerations](#)

### What's new

#### New interface for choosing image focal point

##### Focal point (optional)

To define this image's most important region, drag a box over the image below. (Current focal point shown)



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

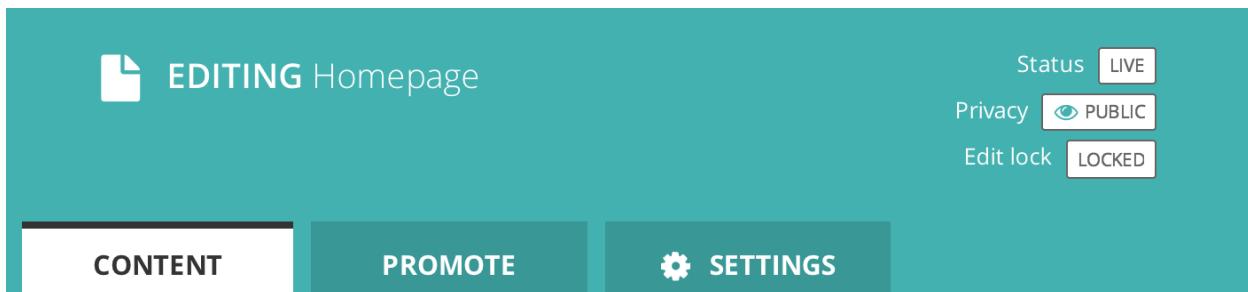
The main navigation menu has been reorganised, placing site configuration options in a ‘Settings’ submenu. This includes two new items, which were previously only available through the Django admin backend: ‘Groups’, for setting up user groups with a specific set of permissions, and ‘Sites’, for managing the list of sites served by this Wagtail instance.

The screenshot shows the Wagtail admin interface for creating a new user group named 'Editors'. The left sidebar has a dark theme with icons for Explorer, Images, Documents, and Settings. The main area has a teal header bar with the title 'EDITING Editors'. A form is displayed with the field 'Name: \* Editors'. Below this, the 'OBJECT PERMISSIONS' section lists permissions for Document, Image, Group, User, and User profile. The 'Document' row has checked boxes for ADD, CHANGE, and DELETE. The 'Image' row also has checked boxes for ADD, CHANGE, and DELETE. The 'User' and 'User profile' rows have empty boxes for all three columns. The 'OTHER PERMISSIONS' section lists 'Can access Wagtail admin' with a checked checkbox.

NAME	ADD	CHANGE	DELETE
Document	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Image	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Group	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
User	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
User profile	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

NAME	
Can access Wagtail admin	<input checked="" type="checkbox"/>

## 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 [How to use images in templates](#)
- Added support for invalidating Cloudflare caches. See [Frontend cache invalidator](#)
- Pages in the explorer can now be ordered by last updated time.

## Bug fixes

- The ‘wagtail start’ command now works on Windows and other environments where the `django-admin.py` executable is not readily accessible.
- The external image URL generator no longer stores generated images in Django’s cache; this was an unintentional side-effect of setting cache control headers.
- The Elasticsearch backend can now search QuerySets that have been filtered with an ‘in’ clause of a non-list type (such as a `ValuesListQuerySet`).
- Logic around the `has_unpublished_changes` flag has been fixed, to prevent issues with the ‘View draft’ button failing to show in some cases.

- It is now easier to move pages to the beginning and end of their section
- Image rendering no longer creates erroneous duplicate Rendition records when the focal point is blank.

### Upgrade considerations

#### Addition of wagtailsites app

The Sites administration interface is contained within a new app, `wagtailsites`. To enable this on an existing Wagtail project, add the line:

```
'wagtail.wagtailsites',
```

to the `INSTALLED_APPS` list in your project's settings file.

#### Title boost on search reduced to 2

Wagtail's search interface applies a 'boost' value to give extra weighting to matches on the title field. The original boost value of 100 was found to be excessive, and in Wagtail 0.7 this has been reduced to 2. If you have used comparable boost values on other fields, to give them similar weighting to title, you may now wish to reduce these accordingly. See [Indexing](#).

#### Addition of locked field to Page model

The page locking mechanism adds a `locked` field to `wagtailcore.Page`, defaulting to `False`. Any application code working with `Page` objects should be unaffected, but any code that creates page records using direct SQL, or within existing South migrations using South's frozen ORM, will fail as this code will be unaware of the new database column. To fix a South migration that fails in this way, add the following line to the '`wagtailcore.page`' entry at the bottom of the migration file:

```
'locked': ('django.db.models.fields.BooleanField', [], {'default': 'False'}),
```

#### Update to focal\_point\_key field on custom Rendition models

The `focal_point_key` field on `wagtailimages.Rendition` has been changed to `null=False`, to fix an issue with duplicate renditions being created. If you have defined a custom Rendition model in your project (by extending the `wagtailimages.AbstractRendition` class), you will need to apply a migration to make the corresponding change on your custom model. Unfortunately neither South nor Django 1.7's migration system are able to generate this automatically - you will need to customise the migration produced by `./manage.py schemamigration ./manage.py makemigrations`, using the `wagtailimages` migration as a guide:

- [https://github.com/wagtail/wagtail/blob/stable/0.7.x/wagtail/wagtailimages/south\\_migrations/0004\\_auto\\_\\_chg\\_field\\_rendition\\_focal\\_point\\_key.py](https://github.com/wagtail/wagtail/blob/stable/0.7.x/wagtail/wagtailimages/south_migrations/0004_auto__chg_field_rendition_focal_point_key.py) (for South / Django 1.6)
- [https://github.com/wagtail/wagtail/blob/stable/0.7.x/wagtail/wagtailimages/migrations/0004\\_make\\_focal\\_point\\_key\\_not\\_nullable.py](https://github.com/wagtail/wagtail/blob/stable/0.7.x/wagtail/wagtailimages/migrations/0004_make_focal_point_key_not_nullable.py) (for Django 1.7)

## 1.9.133 Wagtail 0.6 release notes

September 11, 2014

- [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`.
- `MenuItem`s can now have bundled JavaScript
- Added the `register_admin_menu_item` hook for registering menu items at startup. See [Hooks](#)
- Added a version indicator into the admin interface (hover over the wagtail to see it)
- Added Russian translation

### Bug fixes

- Page URL generation now returns correct URLs for sites that have the main ‘serve’ view rooted somewhere other than ‘/’.
- Search results in the page chooser now respect the `page_type` parameter on `PageChooserPanel`.
- Rendition filenames are now prevented from going over 60 chars, even with a large `focal_point_key`.
- Child relations that are defined on a model’s superclass (such as the base `Page` model) are now picked up correctly by the page editing form, page copy operations and the `replace_text` management command.
- Tags on images and documents are now committed to the search index immediately on saving.

## Upgrade considerations

### All features deprecated in 0.4 have been removed

See: *Deprecated features*

### Search signal handlers have been moved

If you have an import in your urls.py file like from wagtail.wagtailsearch import register\_signal\_handlers, this must now be changed to from wagtail.wagtailsearch.signal\_handlers import register\_signal\_handlers

## Deprecated features

- The wagtail.wagtailsearch.indexed module has been renamed to wagtail.wagtailsearch.index

## 1.9.134 Wagtail 0.5 release notes

August 1, 2014

- *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.

*Dynamic image serve view*

## RoutablePage

A RoutablePage model has been added to allow embedding Django-style URL routing within a page.

*RoutablePageMixin*

### Usage stats for images, documents and snippets

It's now easier to find where a particular image, document or snippet is being used on your site.

Set the `WAGTAIL_USAGE_COUNT_ENABLED` setting to True and an icon will appear on the edit page showing you which pages they have been used on.

## Copy Page action

The explorer interface now offers the ability to copy pages, with or without subpages.

## Minor features

### Core

- Hooks can now be defined using decorator syntax:

```
@hooks.register('construct_main_menu')
def construct_main_menu(request, menu_items):
 menu_items.append(
 MenuItem('Kittens!', '/kittens/',classnames='icon icon-folder-inverse',
order=1000)
)
```

- The lxml library (used for whitelisting and rewriting of rich text fields) has been replaced with the pure-python html5lib library, to simplify installation.
- A `page_unpublished` signal has been added.

### Admin

- Explorer nav now rendered separately and fetched with AJAX when needed.

This improves the general performance of the admin interface for large sites.

### Bug fixes

- Updates to tag fields are now properly committed to the database when publishing directly from the page edit interface.

## Upgrade considerations

### Urlconf entries for /admin/images/, /admin/embeds/ etc need to be removed

If you created a Wagtail project prior to the release of Wagtail 0.3, it is likely to contain the following entries in its `urls.py`:

```
TODO: some way of getting wagtailimages to register itself within wagtailadmin so that we
don't have to define it separately here
url(r'^admin/images/', include(wagtailimages_urls)),
url(r'^admin/embeds/', include(wagtailembeds_urls)),
url(r'^admin/documents/', include(wagtaildocs_admin_urls)),
url(r'^admin/snippets/', include(wagtailsnippets_urls)),
url(r'^admin/search/', include(wagtailsearch_admin_urls)),
url(r'^admin/users/', include(wagtailusers_urls)),
url(r'^admin/redirects/', include(wagtailredirects_urls)),
```

These entries (and the corresponding `from wagtail.wagtail* import ...` lines) need to be removed from `urls.py`. (The entry for `/admin/` should be left in, however.)

Since Wagtail 0.3, the `wagtailadmin` module automatically takes care of registering these URL subpaths, so these entries are redundant, and these urlconf modules are not guaranteed to remain stable and backwards-compatible in future. Leaving these entries in place will now cause an `ImproperlyConfigured` exception to be thrown.

### New fields on Image and Rendition models

Several new fields have been added to the Image and Rendition models to support [Feature Detection](#). These will be added to the database when you run `./manage.py migrate`. If you have defined a custom image model (by extending the `wagtailimages.AbstractImage` and `wagtailimages.AbstractRendition` classes and specifying `WAGTAILIMAGES_IMAGE_MODEL` in settings), the change needs to be applied to that model's database table too. Running the command:

```
$./manage.py schemamigration myapp --auto add_image_focal_point_fields
```

(with 'myapp' replaced with your app name) will generate the necessary migration file.

### South upgraded to 1.0

In preparation for Django 1.7 support in a future release, Wagtail now depends on South 1.0, and its migration files have been moved from `migrations` to `south_migrations`. Older versions of South will fail to find the migrations in the new location.

If your project's requirements file (most commonly `requirements.txt` or `requirements/base.txt`) references a specific older version of South, this must be updated to South 1.0.

## 1.9.135 Wagtail 0.4.1 release notes

July 14, 2014

### Bug fixes

- Elasticsearch backend now respects the backward-compatible URLs configuration setting, in addition to HOSTS
- Documentation fixes

## 1.9.136 Wagtail 0.4 release notes

July 10, 2014

- *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.core.models import Page
>>> Page.objects.live().descendant_of(events_index).search("Hello")
[<Page: Event 1>, <Page: Event 2>]
```

## Sitemap generation

A new module has been added (`wagtail.contrib.wagtailsitemaps`) which produces XML sitemaps for Wagtail sites.

*Sitemap generator*

## Front-end cache invalidation

A new module has been added (`wagtail.contrib.wagtailfrontendcache`) which invalidates pages in a frontend cache when they are updated or deleted in Wagtail.

*Frontend cache invalidator*

## Notification preferences

Users can now decide which notifications they receive from Wagtail using a new “Notification preferences” section located in the account settings.

## Minor features

### Core

- Any extra arguments given to `Page.serve` are now passed through to `get_context` and `get_template`
- Added `in_menu` and `not_in_menu` methods to `PageQuerySet`
- Added `search` method to `PageQuerySet`
- Added `get_next_siblings` and `get_prev_siblings` to `Page`
- Added `page_published` signal
- Added `copy` method to `Page` to allow copying of pages
- Added `construct_whitelister_element_rules` hook for customising the HTML whitelist used when saving `RichText` fields
- Support for setting a `subpage_types` property on `Page` models, to define which page types are allowed as subpages

### Admin

- Removed the “More” section from the menu
- Added pagination to page listings
- Added a new datetime picker widget
- Updated `hallo.js` to version 1.0.4
- Aesthetic improvements to preview experience
- Login screen redirects to dashboard if user is already logged in
- Snippets are now ordered alphabetically

- Added `init_new_page` signal

## Search

- Added a new way to configure searchable/filterable fields on models
- Added `get_indexed_objects` allowing developers to customise which objects get added to the search index
- Major refactor of Elasticsearch backend
- Use `match` instead of `query_string` queries
- Fields are now indexed in Elasticsearch with their correct type
- Filter fields are no longer included in `_all`
- Fields with partial matching are now indexed together into `_partials`

## Images

- Added `original` as a resizing rule supported by the `{% image %}` tag
- `image` tag now accepts extra keyword arguments to be output as attributes on the `img` tag
- Added an `attrs` property to image rendition objects to output `src`, `width`, `height` and `alt` attributes all in one go

## Other

- Added styleguide, for Wagtail developers

## Bug fixes

- Animated GIFs are now coalesced before resizing
- The Wand backend clones images before modifying them
- The admin breadcrumb is now positioned correctly on mobile
- The page chooser breadcrumb now updates the chooser modal instead of linking to Explorer
- Embeds - fixed crash when no HTML field is sent back from the embed provider
- Multiple sites with same hostname but different ports are now allowed
- It is no longer possible to create multiple sites with `is_default_site = True`

## Backwards-incompatible changes

### ElasticUtils replaced with elasticsearch-py

If you are using the Elasticsearch backend, you must install the `elasticsearch` module into your environment.

---

**Note:** If you are using an older version of Elasticsearch (< 1.0) you must install `elasticsearch` version 0.4.x.

---

### Addition of expired column may break old data migrations involving pages

The scheduled publishing mechanism adds an `expired` field to `wagtailcore.Page`, defaulting to `False`. Any application code working with `Page` objects should be unaffected, but any code that creates page records using direct SQL, or within existing South migrations using South's frozen ORM, will fail as this code will be unaware of the `expired` database column. To fix a South migration that fails in this way, add the following line to the '`wagtailcore.page`' entry at the bottom of the migration file:

```
'expired': ('django.db.models.fields.BooleanField', [], {'default': 'False'}),
```

## Deprecated features

### Template tag libraries renamed

The following template tag libraries have been renamed:

- `pageurl` => `wagtailcore_tags`
- `rich_text` => `wagtailcore_tags`
- `embed_filters` => `wagtailembeds_tags`
- `image_tags` => `wagtailimages_tags`

The old names will continue to work, but output a `DeprecationWarning` - you are advised to update any `{% load %}` tags in your templates to refer to the new names.

### New search field configuration format

`indexed_fields` is now deprecated and has been replaced by a new search field configuration format called `search_fields`. See [Indexing](#) for how to define a `search_fields` property on your models.

### Page.route method should now return a RouteResult

Previously, the `route` method called `serve` and returned an `HttpResponse` object. This has now been split up so `serve` is called separately and `route` must now return a `RouteResult` object.

If you are overriding `Page.route` on any of your page models, you will need to update the method to return a `RouteResult` object. The old method of returning an `HttpResponse` will continue to work, but this will throw a `DeprecationWarning` and bypass the `before_serve_page` hook, which means in particular that `Private pages` will not work on those page types. See [Adding Endpoints with Custom route\(\) Methods](#).

## **Wagtailadmin 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.core 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.admin.panels, 451  
wagtail.contrib.forms.panels, 265  
wagtail.contrib.frontend\_cache.utils, 348  
wagtail.contrib.legacy.richtext, 395  
wagtail.contrib.redirects, 393  
wagtail.contrib.redirects.models, 395  
wagtail.contrib.routable\_page, 348  
wagtail.contrib.routable\_page.models, 352  
wagtail.contrib.search\_promotions, 386  
wagtail.coreutils, 530  
wagtail.documents, 109  
wagtail.documents.edit\_handlers, 266  
wagtail.images, 99  
wagtail.images.edit\_handlers, 265  
wagtail.images.models, 96  
wagtail.models, 269  
wagtail.query, 297  
wagtail.snippets.edit\_handlers, 266  
wagtail.test.utils.form\_data, 166



# INDEX

## A

active (*wagtail.models.Task* attribute), 289  
active (*wagtail.models.Workflow* attribute), 287  
active\_workflows (*wagtail.models.Task* attribute), 290  
add\_page() (*wagtail.contrib.frontend\_cache.utils.PurgeBatch* method), 348  
add\_pages() (*wagtail.contrib.frontend\_cache.utils.PurgeBatch* method), 348  
add\_redirect() (*wagtail.contrib.redirects.models.Redirect* method), 395  
add\_url() (*wagtail.contrib.frontend\_cache.utils.PurgeBatch* method), 348  
add\_urls() (*wagtail.contrib.frontend\_cache.utils.PurgeBatch* method), 348  
add\_view\_class (*wagtail.admin.viewsets.model.ModelViewSet* attribute), 453  
alias\_of (*wagtail.models.Page* attribute), 270  
all\_pages() (*wagtail.models.Workflow* method), 288  
all\_tasks\_with\_status() (*wagtail.models.WorkflowState* method), 289  
ancestor\_of() (*wagtail.query.PageQuerySet* method), 299  
approve() (*wagtail.models.TaskState* method), 292  
approve\_moderation() (*wagtail.models.Revision* method), 286  
as\_object() (*wagtail.models.Revision* method), 286  
author\_name (*wagtail.embeds.models.Embed* attribute), 117

## B

base\_block\_class (*wagtail.admin.viewsets.chooser.ChooserViewSet* attribute), 454  
base\_content\_object (*wagtail.models.Revision* attribute), 286  
base\_content\_type (*wagtail.models.Revision* attribute), 284  
base\_form\_class (*wagtail.models.Page* attribute), 275  
base\_widget\_class (*wagtail.admin.viewsets.chooser.ChooserViewSet*

attribute), 454

BaseLogEntry (*class in wagtail.models*), 293  
bind\_to\_model() (*wagtail.admin.panels.Panel* method), 451  
BoundPanel (*class in wagtail.admin.panels.Panel*), 452  
built-in function  
log(), 229  
register\_form\_field\_override(), 217

## C

cached\_content\_type (*wagtail.models.Page* attribute), 272  
can\_create\_at() (*wagtail.models.Page* class method), 275  
can\_exist\_under() (*wagtail.models.Page* class method), 275  
can\_move\_to() (*wagtail.models.Page* method), 275  
cancel() (*wagtail.models.TaskState* method), 292  
cancel() (*wagtail.models.WorkflowState* method), 289  
cautious\_slugify() (*in module wagtail.coreutils*), 530  
child\_of() (*wagtail.query.PageQuerySet* method), 298  
children (*wagtail.admin.panels.FieldRowPanel* attribute), 264  
children (*wagtail.admin.panels.MultiFieldPanel* attribute), 263  
choose\_another\_text (*wagtail.admin.viewsets.chooser.ChooserViewSet* attribute), 454  
choose\_one\_text (*wagtail.admin.viewsets.chooser.ChooserViewSet* attribute), 454  
choose\_results\_view\_class (*wagtail.admin.viewsets.chooser.ChooserViewSet* attribute), 454  
choose\_view\_class (*wagtail.admin.viewsets.chooser.ChooserViewSet* attribute), 454  
ChooserViewSet (*class in wagtail.admin.viewsets.chooser*), 453  
chosen\_view\_class (*wagtail.admin.viewsets.chooser.ChooserViewSet* attribute), 454

**classname** (`wagtail.admin.panels.FieldPanel` attribute), 263  
**classname** (`wagtail.admin.panels.FieldRowPanel` attribute), 264  
**classname** (`wagtail.admin.panels.HelpPanel` attribute), 264  
**clean\_name** (`wagtail.admin.panels.Panel` property), 451  
**clone()** (`wagtail.admin.panels.Panel` method), 451  
**clone\_kwargs()** (`wagtail.admin.panels.Panel` method), 451  
**Comment** (`class` in `wagtail.models`), 295  
**comment** (`wagtail.models.BaseLogEntry` attribute), 294  
**comment** (`wagtail.models.CommentReply` attribute), 296  
**comment** (`wagtail.models.TaskState` attribute), 291  
**comment\_notifications** (`wagtail.models.PageSubscription` attribute), 296  
**CommentReply** (`class` in `wagtail.models`), 296  
**content** (`wagtail.admin.panels.HelpPanel` attribute), 264  
**content** (`wagtail.models.Revision` attribute), 285  
**content\_changed** (`wagtail.models.BaseLogEntry` attribute), 294  
**content\_object** (`wagtail.models.Revision` attribute), 284  
**content\_type** (`wagtail.models.BaseLogEntry` attribute), 293  
**content\_type** (`wagtail.models.Page` attribute), 269  
**content\_type** (`wagtail.models.Revision` attribute), 284  
**content\_type** (`wagtail.models.Task` attribute), 289  
**content\_type** (`wagtail.models.TaskState` attribute), 291  
**contentpath** (`wagtail.models.Comment` attribute), 295  
**context\_object\_name** (`wagtail.models.Page` attribute), 273  
**copy()** (`wagtail.models.TaskState` method), 292  
**copy\_approved\_task\_states\_to\_revision()** (`wagtail.models.WorkflowState` method), 289  
**copy\_for\_translation()** (`wagtail.models.Page` method), 274  
**copy\_for\_translation()** (`wagtail.models.TranslatableMixin` method), 280  
**create\_action\_clicked\_label** (`wagtail.admin.viewsets.chooser.ChooserViewSet` attribute), 455  
**create\_action\_label** (`wagtail.admin.viewsets.chooser.ChooserViewSet` attribute), 455  
**create\_alias()** (`wagtail.models.Page` method), 276  
**create\_rendition()** (`wagtail.images.models.AbstractImage` method), 96  
**create\_view\_class** (`wagtail.admin.viewsets.chooser.ChooserViewSet` attribute), 454  
**created\_at** (`wagtail.models.Comment` attribute), 295  
**created\_at** (`wagtail.models.CommentReply` attribute), 296  
**created\_at** (`wagtail.models.Revision` attribute), 285  
**created\_at** (`wagtail.models.WorkflowState` attribute), 288  
**creation\_form\_class** (`wagtail.admin.viewsets.chooser.ChooserViewSet` attribute), 455  
**creation\_tab\_label** (`wagtail.admin.viewsets.chooser.ChooserViewSet` attribute), 455  
**current\_task\_state** (`wagtail.models.WorkflowState` attribute), 288  
**current\_workflow\_state** (`wagtail.models.Page` attribute), 277  
**current\_workflow\_task** (`wagtail.models.Page` attribute), 277  
**current\_workflow\_task\_state** (`wagtail.models.Page` attribute), 277  
**custom\_field\_preprocess**, 220  
**custom\_value\_preprocess**, 220

## D

**data** (`wagtail.models.BaseLogEntry` attribute), 293  
**deactivate()** (`wagtail.models.Task` method), 290  
**deactivate()** (`wagtail.models.Workflow` method), 287  
**default\_preview\_mode** (`wagtail.models.PreviewableMixin` attribute), 281  
**defer\_streamfields()** (`wagtail.query.PageQuerySet` method), 301  
**delete\_view\_class** (`wagtail.admin.viewsets.model.ModelViewSet` attribute), 453  
**deleted** (`wagtail.models.BaseLogEntry` attribute), 294  
**descendant\_of()** (`wagtail.query.PageQuerySet` method), 298  
**disable\_comments** (`wagtail.admin.panels.FieldPanel` attribute), 263  
**DocumentChooserPanel** (`class` in `wagtail.documents.edit_handlers`), 266  
**draft\_title** (`wagtail.models.Page` attribute), 269  
**DraftStateMixin** (`class` in `wagtail.models`), 283

## E

**edit\_item\_text** (`wagtail.admin.viewsets.chooser.ChooserViewSet` attribute), 454  
**edit\_view\_class** (`wagtail.admin.viewsets.model.ModelViewSet` attribute), 453  
**exact\_type()** (`wagtail.query.PageQuerySet` method), 300

**E**

- exclude\_fields\_in\_copy (wagtail.models.Page attribute), 275
- exclude\_fields\_in\_copy (wagtail.models.TaskState attribute), 292
- exclude\_form\_fields (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 455
- exclude\_form\_fields (wagtail.admin.viewsets.model.ModelViewSet attribute), 453
- expand\_db\_attributes() (LinkHandler method), 238
- export\_headings, 220

**F**

- field\_name (wagtail.admin.panels.FieldPanel attribute), 263
- FieldPanel (class in wagtail.admin.panels), 263
- FieldRowPanel (class in wagtail.admin.panels), 264
- find\_existing\_rendition() (wagtail.images.models.AbstractImage method), 96
- find\_for\_request() (wagtail.models.Site static method), 278
- finish() (wagtail.models.WorkflowState method), 289
- finished\_at (wagtail.models.TaskState attribute), 291
- finished\_by (wagtail.models.TaskState attribute), 291
- first\_common\_ancestor() (wagtail.query.PageQuerySet method), 302
- first\_published\_at (wagtail.models.DraftStateMixin attribute), 283
- first\_published\_at (wagtail.models.Page attribute), 270
- focus() (built-in function), 315
- form (wagtail.admin.panels.Panel.BoundPanel attribute), 452
- form\_fields (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 455
- form\_fields (wagtail.admin.viewsets.model.ModelViewSet attribute), 453
- FormSubmissionsPanel (class in wagtail.contrib.forms.panels), 265
- full\_url (wagtail.models.Page attribute), 272

**G**

- generate\_rendition\_file() (wagtail.images.models.AbstractImage method), 96
- get\_actions() (wagtail.models.Task method), 290
- get\_active() (wagtail.models.Locale class method), 279
- get\_admin\_display\_title() (wagtail.models.Page method), 273
- get\_ancestors() (wagtail.models.Page method), 273
- get\_block\_class() (wagtail.admin.viewsets.chooser.ChooserViewSet method), 454
- get\_bound\_panel() (wagtail.admin.panels.Panel method), 451
- get\_comment() (wagtail.models.TaskState method), 292
- get\_context() (wagtail.models.Page method), 273
- get\_converter\_rule() (FeatureRegistry method), 242
- get\_default() (wagtail.models.Locale class method), 279
- get\_descendants() (wagtail.models.Page method), 273
- get\_description() (wagtail.models.Task class method), 291
- get\_display\_name() (wagtail.models.Locale method), 279
- get\_document\_model() (in module wagtail.documents), 109
- get\_document\_model\_string() (in module wagtail.documents), 109
- get\_editor\_plugin() (FeatureRegistry method), 241
- get\_form\_class() (wagtail.admin.panels.Panel method), 451
- get\_form\_class() (wagtail.admin.viewsets.model.ModelViewSet method), 453
- get\_form\_for\_action() (wagtail.models.Task method), 290
- get\_form\_options() (wagtail.admin.panels.Panel method), 451
- get\_full\_url() (wagtail.models.Page method), 272
- get\_image\_model() (in module wagtail.images), 99
- get\_image\_model\_string() (in module wagtail.images), 99
- get\_instance() (LinkHandler method), 238
- get\_latest\_revision\_as\_object() (wagtail.models.RevisionMixin method), 282
- get\_model() (LinkHandler method), 238
- get\_next\_task() (wagtail.models.WorkflowState method), 289
- get\_parent() (wagtail.models.Page method), 273
- get\_preview\_context() (wagtail.models.PreviewableMixin method), 281
- get\_preview\_template() (wagtail.models.PreviewableMixin method), 281
- get\_queryset(), 219
- get\_rendition() (wagtail.images.models.AbstractImage method), 96
- get\_route\_paths() (wagtail.models.Page method), 275
- get\_siblings() (wagtail.models.Page method), 273
- get\_site() (wagtail.models.Page method), 272
- get\_site\_root\_paths() (wagtail.models.Site static

```

 method), 278
get_specific() (wagtail.models.Page method), 271
get_subpage_urls() (wag-
 tail.contrib.routable_page.models.RoutablePageMixin
 class method), 352
get_task_states_user_can_moderate() (wag-
 tail.models.Task method), 290
get_template() (wagtail.models.Page method), 273
get_template_for_action() (wagtail.models.Task
 method), 290
get_translation() (wagtail.models.Page method),
 273
get_translation() (wag-
 tail.models.TranslatableMixin method), 280
get_translation_model() (wag-
 tail.models.TranslatableMixin class method),
 280
get_translation_or_none() (wagtail.models.Page
 method), 273
get_translation_or_none() (wag-
 tail.models.TranslatableMixin method), 280
get_translations() (wagtail.models.Page method),
 273
get_translations() (wag-
 tail.models.TranslatableMixin method), 280
get_url() (wagtail.models.Page method), 272
get_url_name() (wagtail.admin.viewsets.base.ViewSet
 method), 453
get_url_parts() (wagtail.models.Page method), 272
get_urlpatterns() (wag-
 tail.admin.viewsets.base.ViewSet
 method), 452
get_verbose_name() (wagtail.models.Task
 class
 method), 290
get_workflow() (wagtail.models.Page method), 277
getState() (built-in function), 315
getValue() (built-in function), 315
group (wagtail.models.GroupPagePermission attribute),
 286
GroupPagePermission (class in wagtail.models), 286

```

## H

```

has_translation() (wagtail.models.Page method),
 274
has_translation() (wag-
 tail.models.TranslatableMixin method), 280
has_unpublished_changes (wag-
 tail.models.DraftStateMixin attribute), 283
has_unpublished_changes (wagtail.models.Page at-
 tribute), 269
has_workflow (wagtail.models.Page attribute), 276
header_icon, 219
heading (wagtail.admin.panels.FieldPanel attribute),
 263

```

```

heading (wagtail.admin.panels.HelpPanel attribute),
 264
heading (wagtail.admin.panels.MultiFieldPanel
 attribute), 263
height (wagtail.embeds.models.Embed attribute), 117
HelpPanel (class in wagtail.admin.panels), 264
hostname (wagtail.models.Site attribute), 277
html (wagtail.embeds.models.Embed attribute), 117
|
icon (wagtail.admin.viewsets.chooser.ChooserViewSet
 attribute), 453
icon (wagtail.admin.viewsets.model.ModelViewSet
 attribute), 453
id_for_label() (wag-
 tail.admin.panels.Panel.BoundPanel method),
 452
identifier (LinkHandler attribute), 238
idForLabel (None attribute), 315
ImageChooserPanel (class in wag-
 tail.images.edit_handlers), 265
in_menu() (wagtail.query.PageQuerySet method), 297
in_site() (wagtail.query.PageQuerySet method), 298
index_view_class (wag-
 tail.admin.viewsets.model.ModelViewSet
 attribute), 453
inline_formset() (in module wag-
 tail.test.utils.form_data), 166
InlinePanel (class in wagtail.admin.panels), 264
instance (wagtail.admin.panels.Panel.BoundPanel
 attribute), 452
is_creatable (wagtail.models.Page attribute), 275
is_default_site (wagtail.models.Site attribute), 277
is_latest_revision() (wagtail.models.Revision
 method), 286
is_previewable() (wagtail.models.PreviewableMixin
 method), 281
is_shown(), 401
is_shown() (wagtail.admin.panels.Panel.BoundPanel
 method), 452

```

## L

```

label (wagtail.models.BaseLogEntry attribute), 293
language_code (wagtail.models.Locale attribute), 279
last_published_at (wagtail.models.DraftStateMixin
 attribute), 283
last_published_at (wagtail.models.Page attribute),
 270
last_updated (wagtail.embeds.models.Embed
 attribute), 118
latest_revision (wagtail.models.RevisionMixin
 attribute), 282
LinkHandler (built-in class), 238
list_export, 220

```

**live** (*wagtail.models.DraftStateMixin attribute*), 283  
**live** (*wagtail.models.Page attribute*), 269  
**live()** (*wagtail.query.PageQuerySet method*), 297  
**live\_revision** (*wagtail.models.DraftStateMixin attribute*), 283  
**locale** (*wagtail.models.Page attribute*), 271  
**locale** (*wagtail.models.TranslatableMixin attribute*), 280  
**localized** (*wagtail.models.Page attribute*), 274  
**localized** (*wagtail.models.TranslatableMixin attribute*), 280  
**localized\_draft** (*wagtail.models.Page attribute*), 274  
**locked** (*wagtail.models.Page attribute*), 270  
**locked\_at** (*wagtail.models.Page attribute*), 270  
**locked\_by** (*wagtail.models.Page attribute*), 270  
**log()**  
  built-in function, 229

**M**

**max\_count** (*wagtail.models.Page attribute*), 275  
**max\_count\_per\_parent** (*wagtail.models.Page attribute*), 275  
**max\_width** (*wagtail.embeds.models.Embed attribute*), 117  
**media**, 213  
**model** (*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 453  
**model** (*wagtail.admin.viewsets.model.ModelViewSet attribute*), 453  
**ModelViewSet** (*class in wagtail.admin.viewsets.model*), 453  
**module**  
  **wagtail.admin.panels**, 451  
  **wagtail.contrib.forms.panels**, 265  
  **wagtail.contrib.frontend\_cache.utils**, 348  
  **wagtail.contrib.legacy.richtext**, 395  
  **wagtail.contrib.redirects**, 393  
  **wagtail.contrib.redirects.models**, 395  
  **wagtail.contrib.routable\_page**, 348  
  **wagtail.contrib.routable\_page.models**, 352  
  **wagtail.contrib.search\_promotions**, 386  
  **wagtail.coreutils**, 530  
  **wagtail.documents**, 109  
  **wagtail.documents.edit\_handlers**, 266  
  **wagtail.images**, 99  
  **wagtail.images.edit\_handlers**, 265  
  **wagtail.images.models**, 96  
  **wagtail.models**, 269  
  **wagtail.query**, 297  
  **wagtail.snippets.edit\_handlers**, 266  
  **wagtail.test.utils.form\_data**, 166  
**MultiFieldPanel** (*class in wagtail.admin.panels*), 263

**N**

**name** (*wagtail.models.Task attribute*), 289  
**name** (*wagtail.models.Workflow attribute*), 287  
**nested\_form\_data()** (*in module wagtail.test.utils.form\_data*), 166  
**not\_ancestor\_of()** (*wagtail.query.PageQuerySet method*), 299  
**not\_child\_of()** (*wagtail.query.PageQuerySet method*), 299  
**not\_descendant\_of()** (*wagtail.query.PageQuerySet method*), 298  
**not\_exact\_type()** (*wagtail.query.PageQuerySet method*), 301  
**not\_in\_menu()** (*wagtail.query.PageQuerySet method*), 298  
**not\_live()** (*wagtail.query.PageQuerySet method*), 297  
**not\_page()** (*wagtail.query.PageQuerySet method*), 298  
**not\_parent\_of()** (*wagtail.query.PageQuerySet method*), 299  
**not\_public()** (*wagtail.query.PageQuerySet method*), 300  
**not\_sibling\_of()** (*wagtail.query.PageQuerySet method*), 299  
**not\_type()** (*wagtail.query.PageQuerySet method*), 300

**O**

**object\_id** (*wagtail.models.Revision attribute*), 285  
**object\_id()** (*wagtail.models.BaseLogEntry method*), 294  
**object\_verbose\_name** (*wagtail.models.BaseLogEntry attribute*), 294  
**objects** (*wagtail.models.Revision attribute*), 285  
**on\_action()** (*wagtail.models.Task method*), 290  
**on\_model\_bound()** (*wagtail.admin.panels.Panel method*), 451  
**on\_register()** (*wagtail.admin.viewsets.base.ViewSet method*), 452  
**order**, 401  
**Orderable** (*class in wagtail.models*), 287  
**owner** (*wagtail.models.Page attribute*), 269

**P**

**Page** (*class in wagtail.models*), 269  
**page** (*wagtail.models.Comment attribute*), 295  
**page** (*wagtail.models.GroupPagePermission attribute*), 286  
**page** (*wagtail.models.PageLogEntry attribute*), 294  
**page** (*wagtail.models.PageSubscription attribute*), 296  
**page** (*wagtail.models.PageViewRestriction attribute*), 287  
**page** (*wagtail.models.WorkflowPage attribute*), 293  
**page** (*wagtail.models.WorkflowState attribute*), 288  
**page()** (*wagtail.query.PageQuerySet method*), 298  
**page\_locked\_for\_user()** (*wagtail.models.Task method*), 290

```

page_revision (wagtail.models.TaskState attribute), register_converter_rule() (FeatureRegistry
 291 method), 242
page_revisions (wagtail.models.Revision attribute), register_editor_plugin() (FeatureRegistry
 285 method), 241
page_title (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), register_embed_type() (FeatureRegistry
 454 method), 240
page_type_display_name (wagtail.models.Page attribute), register_form_field_override()
 272 built-in function, 217
PageChooserPanel (class in wagtail.admin.panels), 265
PageLogEntry (class in wagtail.models), 294
PageQuerySet (class in wagtail.query), 297
PageSubscription (class in wagtail.models), 296
PageViewRestriction (class in wagtail.models), 287
Panel (class in wagtail.admin.panels), 451
panel (wagtail.admin.panels.Panel.BoundPanel attribute), 452
parent_of() (wagtail.query.PageQuerySet method), 299
parent_page_types (wagtail.models.Page attribute), 274
password (wagtail.models.PageViewRestriction attribute), 287
password_required_template (wagtail.models.Page attribute), 275
per_page (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 454
permission (wagtail.admin.panels.FieldPanel attribute), 263
permission_type (wagtail.models.GroupPagePermission attribute), 286
port (wagtail.models.Site attribute), 277
position (wagtail.models.Comment attribute), 295
prefix (wagtail.admin.panels.Panel.BoundPanel attribute), 452
preview_modes (wagtail.models.Page attribute), 273
preview_modes (wagtail.models.PreviewableMixin attribute), 281
PreviewableMixin (class in wagtail.models), 281
private() (wagtail.query.PageQuerySet method), 300
provider_name (wagtail.embeds.models.Embed attribute), 117
public() (wagtail.query.PageQuerySet method), 300
publish() (wagtail.models.DraftStateMixin method), 283
publish() (wagtail.models.Revision method), 286
purge() (wagtail.contrib.frontend_cache.utils.PurgeBatch method), 348
PurgeBatch (class in wagtail.contrib.frontend_cache.utils), 348
R
Redirect (class in wagtail.contrib.redirects.models), 395
register_converter_rule() (FeatureRegistry
 method), 242
register_editor_plugin() (FeatureRegistry
 method), 241
register_embed_type() (FeatureRegistry
 method), 240
register_form_field_override()
 built-in function, 217
register_link_type() (FeatureRegistry method), 239
register_widget (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 454
reject() (wagtail.models.TaskState method), 292
reject_moderation() (wagtail.models.Revision method), 286
relative_url() (wagtail.models.Page method), 272
render() (built-in function), 315
render() (wagtail.contrib.routable_page.models.RoutablePageMixin method), 352
render_html(), 213
request (wagtail.admin.panels.Panel.BoundPanel attribute), 452
requested_by (wagtail.models.WorkflowState attribute), 288
resolve_subpage() (wagtail.contrib.routable_page.models.RoutablePageMixin method), 352
resolved_at (wagtail.models.Comment attribute), 295
resolved_by (wagtail.models.Comment attribute), 295
resume() (wagtail.models.WorkflowState method), 289
reverse_subpage() (wagtail.contrib.routable_page.models.RoutablePageMixin method), 352
Revision (class in wagtail.models), 284
revision (wagtail.models.BaseLogEntry attribute), 293
revision_created (wagtail.models.Comment attribute), 295
RevisionMixin (class in wagtail.models), 282
revisions (wagtail.models.RevisionMixin attribute), 282
revisions() (wagtail.models.WorkflowState method), 289
rich_text() (in module wagtail.test.utils.form_data), 166
root_page (wagtail.models.Site attribute), 277
root_url (wagtail.models.Site attribute), 278
RoutablePageMixin (class in wagtail.contrib.routable_page.models), 352
routablepageurl() (in module wagtail.contrib.routable_page.templatetags.wagtailroutablepage_tags), 353
route() (wagtail.models.Page method), 273

```

**S**

`save()` (*wagtail.models.Page* method), 276  
`save_revision()` (*wagtail.models.RevisionMixin* method), 282  
`search()` (*wagtail.query.PageQuerySet* method), 300  
`search_description` (*wagtail.models.Page* attribute), 270  
`search_fields` (*wagtail.models.Page* attribute), 274  
`search_tab_label` (*wagtail.admin.viewsets.chooser.ChooserViewSet* attribute), 455  
`seo_title` (*wagtail.models.Page* attribute), 270  
`serve()` (*wagtail.models.Page* method), 273  
`serve_preview()` (*wagtail.models.Page* method), 273  
`serve_preview()` (*wagtail.models.PreviewableMixin* method), 281  
`setState()` (built-in function), 315  
`show_in_menus` (*wagtail.models.Page* attribute), 270  
`sibling_of()` (*wagtail.query.PageQuerySet* method), 299  
**Site** (class in *wagtail.models*), 277  
`site_name` (*wagtail.models.Site* attribute), 277  
`slug` (*wagtail.models.Page* attribute), 269  
**SnippetChooserPanel** (class in *wagtail.snippets.edit\_handlers*), 266  
`sort_order` (*wagtail.models.Orderable* attribute), 287  
`sort_order` (*wagtail.models.WorkflowTask* attribute), 292  
`specific` (*wagtail.models.Page* attribute), 271  
`specific` (*wagtail.models.Task* attribute), 290  
`specific` (*wagtail.models.TaskState* attribute), 292  
`specific()` (*wagtail.query.PageQuerySet* method), 301  
`specific_class` (*wagtail.models.Page* attribute), 272  
`specific_deferred` (*wagtail.models.Page* attribute), 272  
`start()` (*wagtail.models.Task* method), 290  
`start()` (*wagtail.models.Workflow* method), 287  
`started_at` (*wagtail.models.TaskState* attribute), 291  
`status` (*wagtail.models.TaskState* attribute), 291  
`status` (*wagtail.models.WorkflowState* attribute), 288  
`STATUS_CHOICES` (*wagtail.models.TaskState* attribute), 292  
`STATUS_CHOICES` (*wagtail.models.WorkflowState* attribute), 288  
`streamfield()` (in module *wagtail.test.utils.form\_data*), 166  
**StreamFieldPanel** (class in *wagtail.admin.panels*), 263  
`submitted_for_moderation` (*wagtail.models.Revision* attribute), 285  
`submitted_revisions` (*wagtail.models.Revision* attribute), 285  
`subpage_types` (*wagtail.models.Page* attribute), 274  
`SummaryItem()`, 401

**T**

`Task` (class in *wagtail.models*), 289  
`task` (*wagtail.models.TaskState* attribute), 291  
`task` (*wagtail.models.WorkflowTask* attribute), 292  
`task_state_class` (*wagtail.models.Task* attribute), 290  
`task_type_started_at` (*wagtail.models.TaskState* attribute), 292  
`tasks` (*wagtail.models.Workflow* attribute), 287  
**TaskState** (class in *wagtail.models*), 291  
`template` (*wagtail.admin.panels.HelpPanel* attribute), 264  
`template_name`, 219  
`text` (*wagtail.models.Comment* attribute), 295  
`text` (*wagtail.models.CommentReply* attribute), 296  
`thumbnail_url` (*wagtail.embeds.models.Embed* attribute), 117  
`timestamp` (*wagtail.models.BaseLogEntry* attribute), 294  
`title`, 219  
`title` (*wagtail.embeds.models.Embed* attribute), 117  
`title` (*wagtail.models.Page* attribute), 269  
**TranslatableMixin** (class in *wagtail.models*), 280  
`translation_key` (*wagtail.models.Page* attribute), 271  
`translation_key` (*wagtail.models.TranslatableMixin* attribute), 280  
`type` (*wagtail.embeds.models.Embed* attribute), 117  
`type()` (*wagtail.query.PageQuerySet* method), 300

**U**

`unpublish()` (*wagtail.models.DraftStateMixin* method), 284  
`unpublish()` (*wagtail.query.PageQuerySet* method), 301  
`update()` (*wagtail.models.WorkflowState* method), 288  
`update_aliases()` (*wagtail.models.Page* method), 276  
`updated_at` (*wagtail.models.Comment* attribute), 295  
`updated_at` (*wagtail.models.CommentReply* attribute), 296  
`url` (*wagtail.embeds.models.Embed* attribute), 117  
`user` (*wagtail.models.BaseLogEntry* attribute), 293  
`user` (*wagtail.models.Comment* attribute), 295  
`user` (*wagtail.models.CommentReply* attribute), 296  
`user` (*wagtail.models.PageSubscription* attribute), 296  
`user` (*wagtail.models.Revision* attribute), 285  
`user_can_access_editor()` (*wagtail.models.Task* method), 290  
`user_can_lock()` (*wagtail.models.Task* method), 290  
`user_can_unlock()` (*wagtail.models.Task* method), 290  
`user_display_name` (*wagtail.models.BaseLogEntry* attribute), 294

**V**

**ViewSet** (class in *wagtail.admin.viewsets.base*), 452

# W

wagtail.admin.forms.WagtailAdminModelForm  
    (built-in class), 148

wagtail.admin.forms.WagtailAdminPageForm  
    (built-in class), 148

wagtail.admin.panels  
    module, 451

wagtail.blocks.BlockQuoteBlock (built-in class),  
    308

wagtail.blocks.BooleanBlock (built-in class), 306

wagtail.blocks.CharBlock (built-in class), 304

wagtail.blocks.ChoiceBlock (built-in class), 309

wagtail.blocks.DateBlock (built-in class), 307

wagtail.blocks.DateTimeBlock (built-in class), 307

wagtail.blocks.DecimalBlock (built-in class), 305

wagtail.blocks.EmailBlock (built-in class), 305

wagtail.blocks.FloatBlock (built-in class), 305

wagtail.blocks.IntegerBlock (built-in class), 305

wagtail.blocks.ListBlock (built-in class), 312

wagtail.blocks.MultipleChoiceBlock (built-in  
    class), 309

wagtail.blocks.PageChooserBlock (built-in class),  
    310

wagtail.blocks.RawHTMLBlock (built-in class), 308

wagtail.blocks.RegexBlock (built-in class), 306

wagtail.blocks.RichTextBlock (built-in class), 308

wagtail.blocks.StaticBlock (built-in class), 311

wagtail.blocks.StreamBlock (built-in class), 312

wagtail.blocks.StructBlock (built-in class), 311

wagtail.blocks.TextBlock (built-in class), 304

wagtail.blocks.TimeBlock (built-in class), 307

wagtail.blocks.URLBlock (built-in class), 306

wagtail.contrib.forms.panels  
    module, 265

wagtail.contrib.frontend\_cache.utils  
    module, 348

wagtail.contrib.legacy.richtext  
    module, 395

wagtail.contrib.redirects  
    module, 393

wagtail.contrib.redirects.models  
    module, 395

wagtail.contrib.routable\_page  
    module, 348

wagtail.contrib.routable\_page.models  
    module, 352

wagtail.contrib.search\_promotions  
    module, 386

wagtail.coreutils  
    module, 530

wagtail.documents  
    module, 109

wagtail.documents.blocks.DocumentChooserBlock  
    (built-in class), 310

wagtail.documents.edit\_handlers  
    module, 266

wagtail.embeds.blocks.EmbedBlock (built-in  
    class), 310

wagtail.embeds.models.Embed (built-in class), 117

wagtail.fields.StreamField (built-in class), 303

wagtail.images  
    module, 99

wagtail.images.blocks.ImageChooserBlock  
    (built-in class), 310

wagtail.images.edit\_handlers  
    module, 265

wagtail.images.models  
    module, 96

wagtail.models  
    module, 269

wagtail.query  
    module, 297

wagtail.snippets.blocks.SnippetChooserBlock  
    (built-in class), 310

wagtail.snippets.edit\_handlers  
    module, 266

wagtail.test.utils.form\_data  
    module, 166

widget (wagtail.admin.panels.FieldPanel attribute), 263

widget\_class (wagtail.admin.viewsets.chooser.ChooserViewSet  
    attribute), 454

widget\_templatetags\_adapter\_class (wagtail.admin.viewsets.chooser.ChooserViewSet  
    attribute), 454

width (wagtail.embeds.models.Embed attribute), 117

with\_content\_json() (wagtail.models.DraftStateMixin method), 284

with\_content\_json() (wagtail.models.Page method),  
    276

with\_content\_json() (wagtail.models.RevisionMixin  
    method), 282

Workflow (class in wagtail.models), 287

workflow (wagtail.models.WorkflowPage attribute), 293

workflow (wagtail.models.WorkflowState attribute), 288

workflow (wagtail.models.WorkflowTask attribute), 292

workflow\_in\_progress (wagtail.models.Page attribute),  
    277

workflow\_state (wagtail.models.TaskState attribute),  
    291

WorkflowPage (class in wagtail.models), 293

workflows (wagtail.models.Task attribute), 290

WorkflowState (class in wagtail.models), 288

WorkflowTask (class in wagtail.models), 292