
Wagtail Documentation

Release 6.4

Wagtail core team and contributors

Feb 20, 2025

CONTENTS

1 Index	3
1.1 Getting started	3
1.2 Tutorial	35
1.3 Usage guide	63
1.4 Advanced	141
1.5 Extending	291
1.6 Reference	363
1.7 Deployment & hosting	569
1.8 Support	583
1.9 Editor's guide	584
1.10 Contributing	584
1.11 Release notes	640
Python Module Index	985
Index	987

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*](#)
- [*Tutorial*](#)

- **Using Wagtail**

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

- **For editors**

- Editors guide (separate site)

1.1 Getting started

1.1.1 Your first Wagtail site

This tutorial shows you how to build a blog using Wagtail. Also, the tutorial gives you hands-on experience with some of Wagtail's features.

To complete this tutorial, we recommend that you have some basic programming knowledge, as well as an understanding of web development concepts. A basic understanding of Python and the Django framework ensures a more grounded understanding of this tutorial, but it's not mandatory.

Note

If you want to add Wagtail to an existing Django project instead, see [Integrating Wagtail into a Django project](#).

Install and run Wagtail

Install dependencies

View the [compatible versions of Python](#) that Wagtail supports.

To check if you have an appropriate version of Python 3, run the following command:

```
python --version
# Or:
python3 --version
# **On Windows** (cmd.exe, with the Python Launcher for Windows):
py --version
```

If none of the preceding commands return a version number, or return a version lower than 3.9, then install Python 3.

Create and activate a virtual environment

This tutorial recommends using a virtual environment, which isolates installed dependencies from other projects. This tutorial uses `venv`, which is packaged with Python 3. On Ubuntu, it may be necessary to run `sudo apt install python3-venv` to install it.

On Windows (cmd.exe), run the following command to create a virtual environment:

```
py -m venv mysite\env
```

Activate this virtual environment using:

```
mysite\env\Scripts\activate.bat

# if mysite\env\Scripts\activate.bat doesn't work, run:

mysite\env\Scripts\activate
```

On GNU/Linux or MacOS (bash):

Create the virtual environment using:

```
python -m venv mysite/env
```

Activate the virtual environment using:

```
source mysite/env/bin/activate
```

Upon activation, your command line will show `(env)` to indicate that you're now working within this virtual environment.

For other shells see the `venv` documentation.

Note

If you're using version control such as git, then `mysite` is the directory for your project. You must exclude the `env` directory from any version control.

Install Wagtail

To install Wagtail and its dependencies, use pip, which is packaged with Python:

```
pip install wagtail
```

Generate your site

Wagtail provides a `start` command similar to `django-admin startproject`. Running `wagtail start mysite` in your project generates 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
```

Here is the generated project structure:

```
mysite/
├── .dockerignore
├── Dockerfile
├── home/
├── manage.py*
├── mysite/
└── requirements.txt
    └── search/
```

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 that you've just created. The `requirements.txt` file contains all the dependencies needed to run the project.

Create the database

By default, your database is SQLite. To match your database tables with your project's models, run the following command:

```
python manage.py migrate
```

This command ensures that the tables in your database match the models in your project. Every time you alter your model, then you must run the `python manage.py migrate` command to update the database. For example, if you add a field to a model, then you must run the command.

Create an admin user

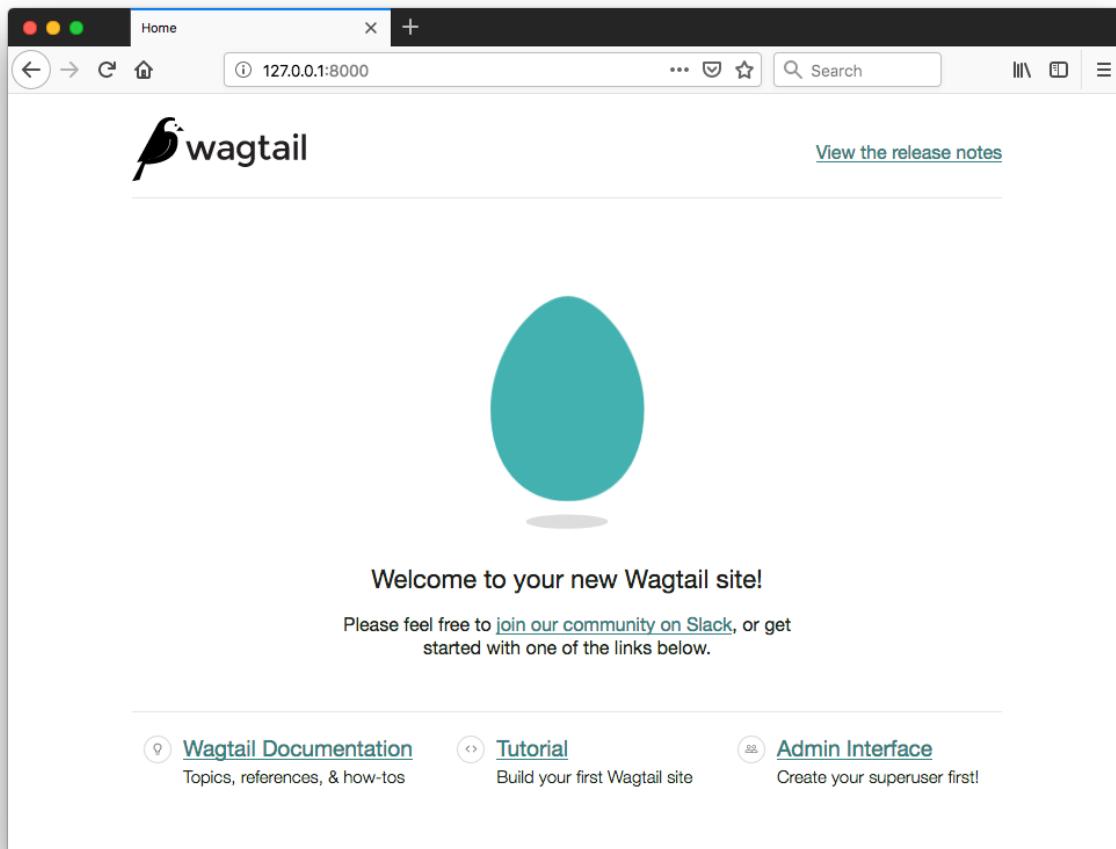
```
python manage.py createsuperuser
```

This prompts you to create a new admin user account with full permissions. It's important to note that for security reasons, the password text won't be visible while typing.

Start the server

```
python manage.py runserver
```

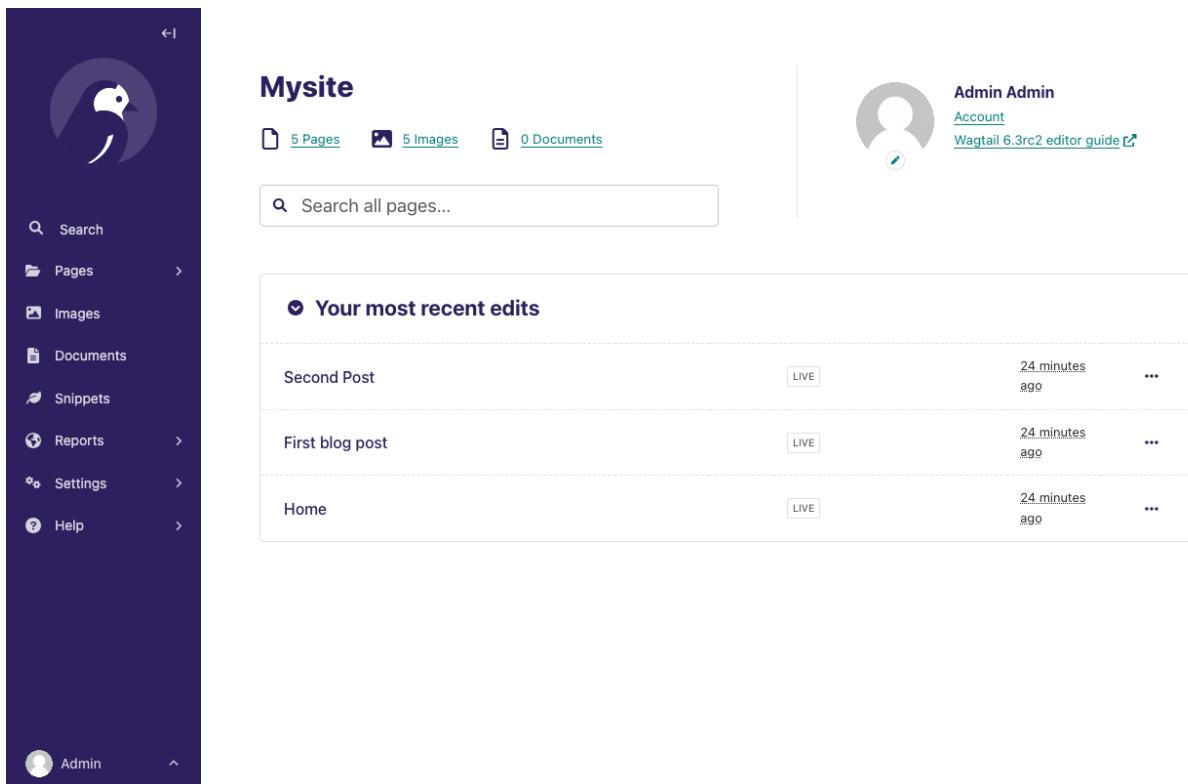
After the server starts, go to <http://127.0.0.1:8000> to see Wagtail's welcome page:



Note

This tutorial uses `http://127.0.0.1:8000` as the URL for your development server but depending on your setup, this could be a different IP address or port. Please read the console output of `manage.py runserver` to determine the correct URL for your local site.

You can now access the [admin interface](#) by logging into `http://127.0.0.1:8000/admin` with the username and password that you entered while creating an admin user with `createsuperuser`.



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

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

    content_panels = Page.content_panels + ["body"]
```

`body` is a `RichTextField`, a special Wagtail field. When `blank=True`, it means the field isn’t mandatory and you can leave it empty. You can use any of the [Django core fields](#). `content_panels` define the capabilities and the layout of the editing interface. Adding fields to `content_panels` enables you to edit them in the Wagtail [admin interface](#). You can read more about this on [Page models](#).

Run:

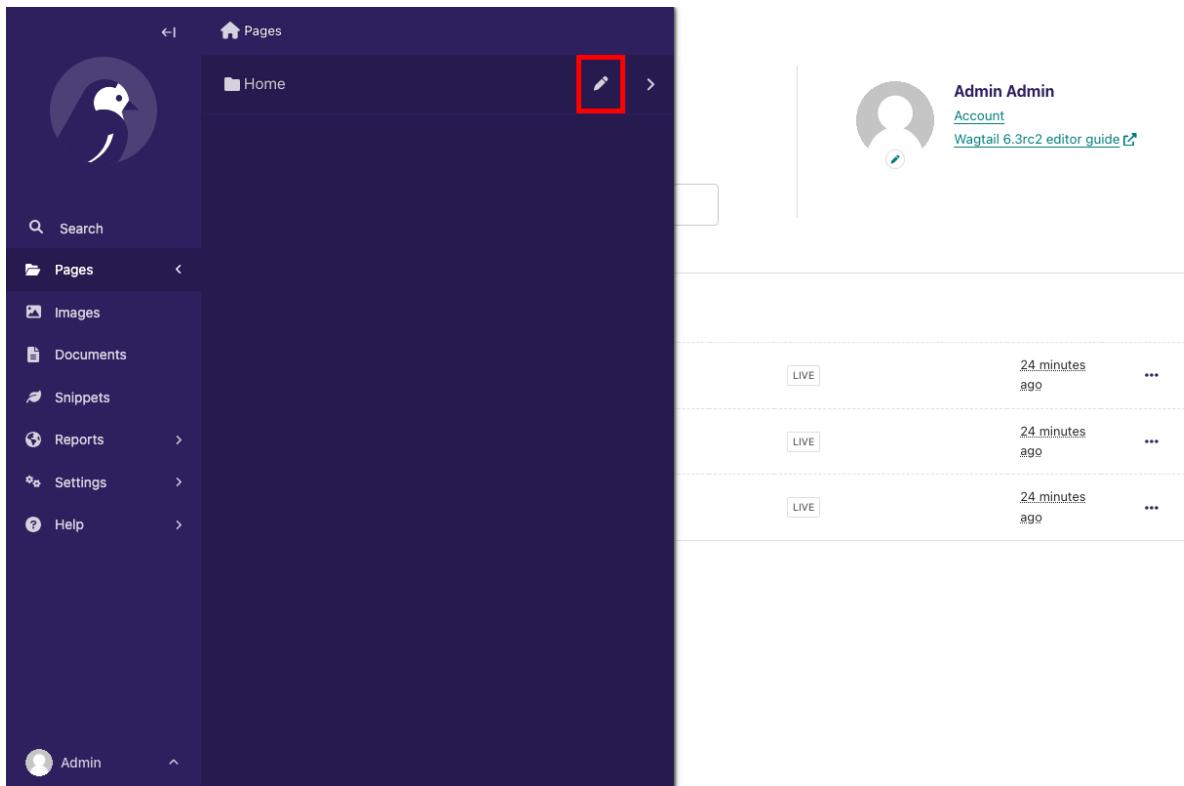
```
# Creates the migrations file.
python manage.py makemigrations

# Executes the migrations and updates the database with your model changes.
python manage.py migrate
```

You must run the preceding commands each time you make changes to the model definition. Here is the expected output from the terminal:

```
Migrations for 'home':
  home/migrations/0003_homepage_body.py
    - Add field body to homepage
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, home, sessions, taggit, wagtailadmin, wagtailcore, wagtailembeds, wagtailforms, wagtailimages, wagtailredirects, wagtailsearch, wagtailusers
Running migrations:
  Applying home.0003_homepage_body... OK
```

You can now edit the homepage within the Wagtail **admin** interface. On your **Sidebar**, go to **Pages** and click edit beside **Home** to see the new body field.



Enter the text “Welcome to our new site!” into the body field, and publish the page by selecting **Publish** at the bottom of the page editor, rather than **Save Draft**.

You must update the page template to reflect the changes made to the model. Wagtail uses normal Django templates to render each page type. By default, it looks 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 that Django’s [template rules](#) recognize. Conventionally, you can place it within a `templates` folder within the app.

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

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

<!-- load wagtailcore_tags by adding this: --&gt;
{% load wagtailcore_tags %}</pre>
```

(continues on next page)

(continued from previous page)

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

<!-- replace everything below with: -->
{% block content %}
  {{ page.body|richtext }}
{% endblock %}
```

base.html refers to a parent template. It must always be the first template tag that you use 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 can override specific content within the parent template.

Also, you must load wagtailcore_tags 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 you can load by including `{% load wagtailcore_tags %}` at the top of your template file.

This tutorial uses the `richtext` filter to escape and print the contents of a RichTextField:

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

Produces:

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

Note: You must include `{% load wagtailcore_tags %}` in each template that uses Wagtail's tags. If the tags aren't loaded, Django throws a `TemplateSyntaxError`.

A basic blog

You are now ready to create a blog, use the following command line to create a new app in your Wagtail project.

```
python manage.py startapp blog
```

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

```
INSTALLED_APPS = [
    "blog", # <- Our new blog app.
    "home",
    "search",
    "wagtail.contrib.forms",
    "wagtail.contrib.redirects",
    "wagtail.embeds",
    "wagtail.sites",
    "wagtail.users",
    #... other packages
]
```

Note

You must register all apps within the `INSTALLED_APPS` section of the `base.py` file in the `mysite/settings` directory. Look at this file to see how the `start` command lists your project's apps.

Blog index and posts

Start with creating a simple index page for your blog. Edit `blog/models.py` to include:

```
from django.db import models

# Add these:
from wagtail.models import Page
from wagtail.fields import RichTextField

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

    content_panels = Page.content_panels + ["intro"]
```

Since you added a new model to your app, you must create and run a database migration:

```
python manage.py makemigrations
python manage.py migrate
```

Also, since the model name is `BlogIndexPage`, the default template name, unless you override it, is `blog_index_page.html`. Django looks for a template whose name matches the name of your `Page` model within the `templates` directory in your blog app folder. You can override this default behavior if you want to. To create a template for the `BlogIndexPage` model, create a file at the location `blog/templates/blog/blog_index_page.html`.

Note

You need to create the folders `templates/blog` within your `blog` app folder.

In your `blog_index_page.html` file enter the following content:

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

{% load wagtailcore_tags %}

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

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

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

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

{% endblock %}
```

Other than using `get_children`, the preceding `blog_index_page.html` template is similar to your previous work with the `home_page.html` template. You will learn about the use of `get_children` later in the tutorial.

If you have a Django background, then you will notice that the `pageurl` tag is similar to Django's `url` tag, but takes a Wagtail Page object as an additional argument.

Now that this is complete, here is how you can create a page from the Wagtail admin interface:

1. Go to <http://127.0.0.1:8000/admin> and sign in with your admin user details.
2. In the Wagtail admin interface, go to Pages, then click Home.
3. Add a child page to the Home page by clicking the + icon (Add child page) at the top of the screen.
4. Choose **Blog index page** from the list of the page types.
5. Use “Blog” as your page title, make sure it has the slug “blog” on the Promote tab, and publish it.

You can now access the URL, <http://127.0.0.1:8000/blog> on your site. Note how the slug from the Promote tab defines the page URL.

Now create a model and template for your blog posts. Edit `blog/models.py` to include:

```
from django.db import models
from wagtail.models import Page
from wagtail.fields import RichTextField

# add this:
from wagtail.search import index

# keep the definition of BlogIndexPage model, and add the BlogPage model:

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

(continues on next page)

(continued from previous page)

```
body = RichTextField(blank=True)

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

content_panels = Page.content_panels + ["date", "intro", "body"]
```

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

You have to migrate your database again because of the new changes in your `models.py` file:

```
python manage.py makemigrations
python manage.py migrate
```

Create a new template file at the location `blog/templates/blog/blog_page.html`. Now add the following content to your `blog_page.html` file:

```
{% 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>

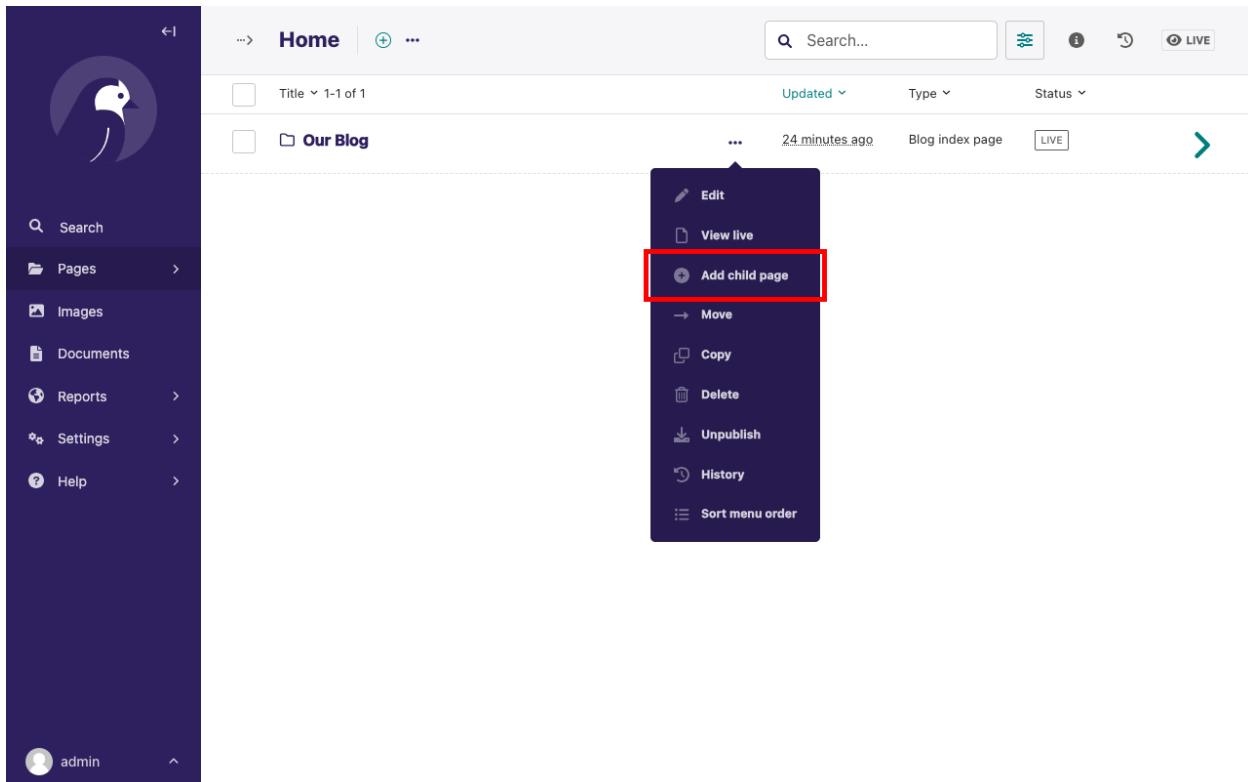
    {{ page.body|richtext }}

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

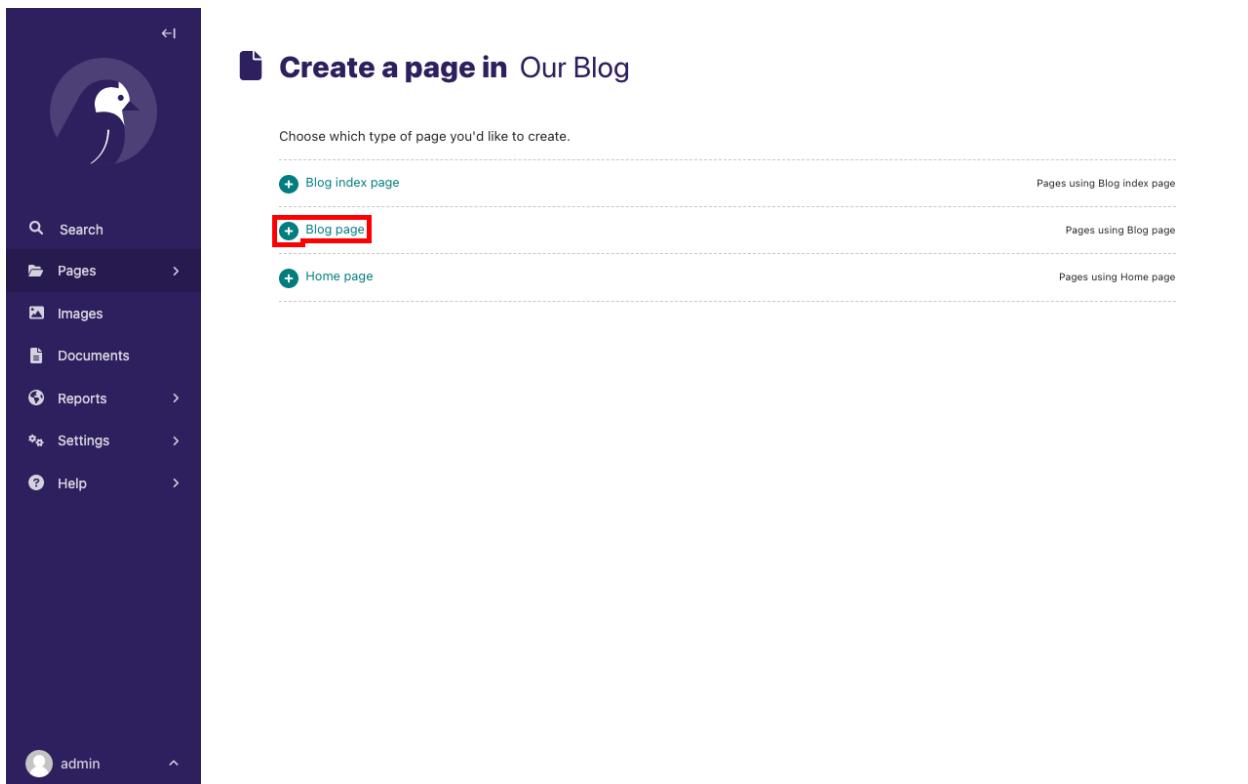
Note the use of Wagtail's built-in `get_parent()` method to obtain the URL of the blog this post is a part of.

Now, go to your `admin interface` and create a few blog posts as children of `BlogIndexPage` by following these steps:

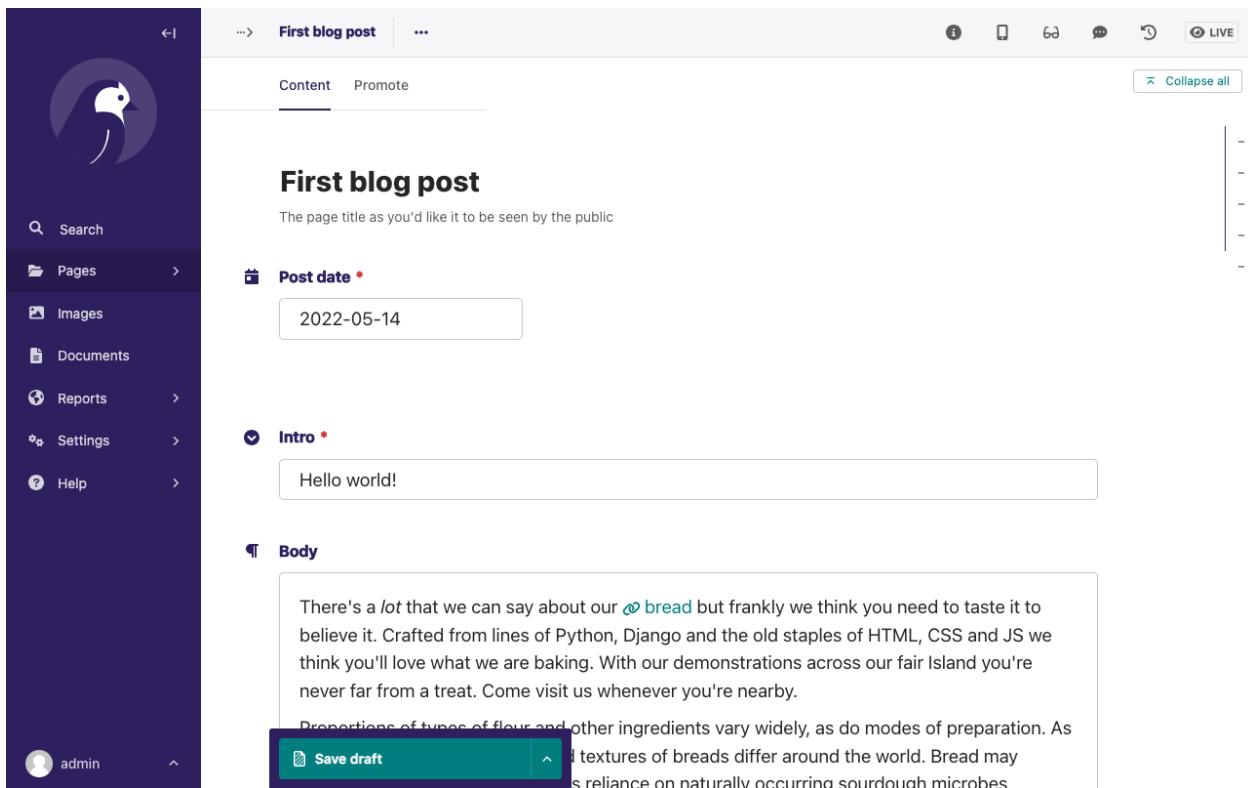
1. Click **Pages** from the Wagtail **Sidebar**, and then click **Home**
2. Hover on **Blog** and click **Add child page**.



Select the page type, **Blog page**:



Populate the fields with the content of your choice:



To add a link from your rich text **Body** field, highlight the text you want to attach the link to. You can now see a pop-up modal which has several actions represented by their icons. Click on the appropriate icon to add a link. You can also click the **+** icon, which appears at the left-hand side of the field to get similar actions as those shown in the pop-up modal.

To add an image, press enter to move to the next line in the field. Then click the **+** icon and select **Image** from the list of actions to add an image.

Note

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

Publish each blog post when you are done editing.

Congratulations! You now have the beginnings of a working blog. If you go to <http://127.0.0.1:8000/blog> in your browser, you can see all the posts that you created by following the preceding steps:

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 in Wagtail revolves around the concept of *hierarchical tree structures* consisting of nodes and leaves. You can read more on this [Theory](#). In this case, the `BlogIndexPage` serves as a *node*, and individual `BlogPage` instances represent 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 position in the hierarchy. But why do you have to specify `post.specific.intro` rather than `post.intro`? This has to do with the way you define your model, `class BlogPage(Page)`. The `get_children()` method gets you a list of instances of the `Page` base class. When you 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 you need `.specific` to access it.

You can simplify the template code by using the Django `with` tag. Now, modify your `blog_index_page.html`:

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

(continues on next page)

(continued from previous page)

```
    {{ post.body|richtext }}
{%
  endwith
%}
{%
  endfor
%}
```

When you start writing more customized 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

With a keen eye, you may have noticed problems with the blog index page:

1. Posts are in chronological order. Generally blogs display content in *reverse* chronological order.
2. Posts drafts are visible. You want to make sure that it displays only *published* content.

To accomplish these, you need to do more than grab the index page's children in the template. Instead, you want to modify the QuerySet in the model definition. Wagtail makes this possible via the overridable `get_context()` method.

Modify your `BlogIndexPage` model:

```
class BlogIndexPage(Page):
    intro = RichTextField(blank=True)
    # add the get_context method:
    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

    # ...
```

Here is a quick breakdown of the changes that you made:

1. You retrieved the original context.
2. You created a custom QuerySet modifier.
3. You added the custom QuerySet modifier to the retrieved context.
4. You returned the modified context to the view.

You also need to modify your `blog_index_page.html` template slightly. Change:

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

Now, unpublish one of your posts. The unpublished post should disappear from your blog's index page. Also, the remaining posts should now be sorted with the most recently published posts coming first.

Images

The next feature that you need to add is the ability to attach an image gallery to your blog posts. While it's possible to simply insert images into the rich text `body` field, there are several advantages to setting up your gallery images as a new dedicated object type within the database. This way, you have full control over the layout and styling of the images on the template, rather than having to lay them out in a particular way within the field. It also makes it possible for you to use the images elsewhere, independently of the blog text. For example, displaying a thumbnail on the blog's index page.

Now modify your `BlogPage` model and add a new `BlogPageGalleryImage` model to `blog/models.py`:

```
# New imports added for ParentalKey, Orderable

from modelcluster.fields import ParentalKey

from wagtail.models import Page, Orderable
from wagtail.fields import RichTextField
from wagtail.search import index

# ... Keep the definition of BlogIndexPage, update the content_panels of BlogPage, ↴
# and add a new BlogPageGalleryImage model:

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 + [
        "date", "intro", "body",

        # Add this
        "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 = ["image", "caption"]
```

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

There are a few new concepts here:

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

2. 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.
3. image is a ForeignKey to Wagtail’s built-in Image model, which stores the actual images. This appears in the page editor as a pop-up interface for choosing an existing image or uploading a new one. This way, you allow an image to exist in multiple galleries. This creates a many-to-many relationship between pages and images.
4. Specifying `on_delete=models.CASCADE` on the foreign key means that deleting the image from the system also deletes the gallery entry. In other situations, it might be appropriate to leave the gallery entry in place. For example, if an “our staff” page includes a list of people with headshots, and you delete one of those photos, but prefer to leave the person in place on the page without a photo. In this case, you must set the foreign key to `blank=True, null=True, on_delete=models.SET_NULL`.
5. Finally, adding the `InlinePanel` to `BlogPage.content_panels` makes the gallery images available on the editing interface for `BlogPage`.

After editing your `blog/models.py`, you should see **Images** in your **Sidebar** and a **Gallery images** field with the option to upload images and provide a caption for it in the **Edit Screen** of your blog posts.

Edit your blog page template `blog_page.html` to include the images section:

```
<!-- Load the wagtailimages_tags: -->
{%- 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 }}

    <!-- Add this: -->
    {%- for item in page.gallery_images.all %}
        <div style="float: inline-start; 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 %}
```

Make sure to upload some images when editing the blog page on your Wagtail admin if you want to display them after editing your blog page template.

Here, you 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 resize and crop the image 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 your gallery images are database objects in their own right, you can now query and re-use them independently of the blog post body. Now, define a `main_image` method in your `BlogPage` model, 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)
    # Add the main_image method:
    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 + ["date", "intro", "body", "gallery_images"]
```

This method is now available from your templates. Update `blog_index_page.html` to load the `wagtailimages_tags` library and include the main image as a thumbnail alongside each post:

```
<!-- Load wagtailimages_tags: -->
{%- load wagtailcore_tags wagtailimages_tags %}

<!-- Modify this: -->
```

(continues on next page)

(continued from previous page)

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

    <!-- Add this: -->
    {% 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 %}
```

Authors

You probably want your blog posts to have authors, which is an essential feature of blogs. The way to go about this is to have a fixed list, managed by the site owner through a separate area of the [admin interface](#).

First, define an Author model. This model isn't a page in its own right. You have to define it as a standard Django `models.Model` rather than inheriting from `Page`. Wagtail introduces the concept of **Snippets** for reusable pieces of content which don't exist as part of the page tree themselves. You can manage snippets through the [admin interface](#). You can register a model as a snippet by adding the `@register_snippet` decorator. Also, you can use all the fields types that you've used so far on pages on snippets too.

To create Authors and give each author an author image as well as a name, add the following to `blog/models.py`:

```
# Add this to the top of the file
from wagtail.snippets.models import register_snippet

# ... Keep BlogIndexPage, BlogPage, BlogPageGalleryImage models, and then add the
# →Author model:
@register_snippet
class Author(models.Model):
    name = models.CharField(max_length=255)
    author_image = models.ForeignKey(
        'wagtailimages.Image', null=True, blank=True,
        on_delete=models.SET_NULL, related_name='+')

    panels = ["name", "author_image"]

    def __str__(self):
        return self.name

    class Meta:
        verbose_name_plural = 'Authors'
```

Note

Note that you 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. So there is no need to distinguish between 'content panels' and 'promote panels'.

Migrate this change by running `python manage.py makemigrations` and `python manage.py migrate`. Create a few authors through the **Snippets** area which now appears in your Wagtail admin interface.

You can now add authors to the `BlogPage` model, as a many-to-many field. The field type to use for this is `ParentalManyToManyField`. This field is a variation of the standard Django `ManyToManyField` that ensures the selected objects are properly associated with the page record in the revision history. It operates in a similar manner to how `ParentalKey` replaces `ForeignKey` for one-to-many relations. To add authors to the `BlogPage`, modify `models.py` in your blog app folder:

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

from modelcluster.fields import ParentalKey, ParentalManyToManyField
from wagtail.models import Page, Orderable
from wagtail.fields import RichTextField
from wagtail.admin.panels import MultiFieldPanel
from wagtail.search import index
from wagtail.snippets.models import register_snippet

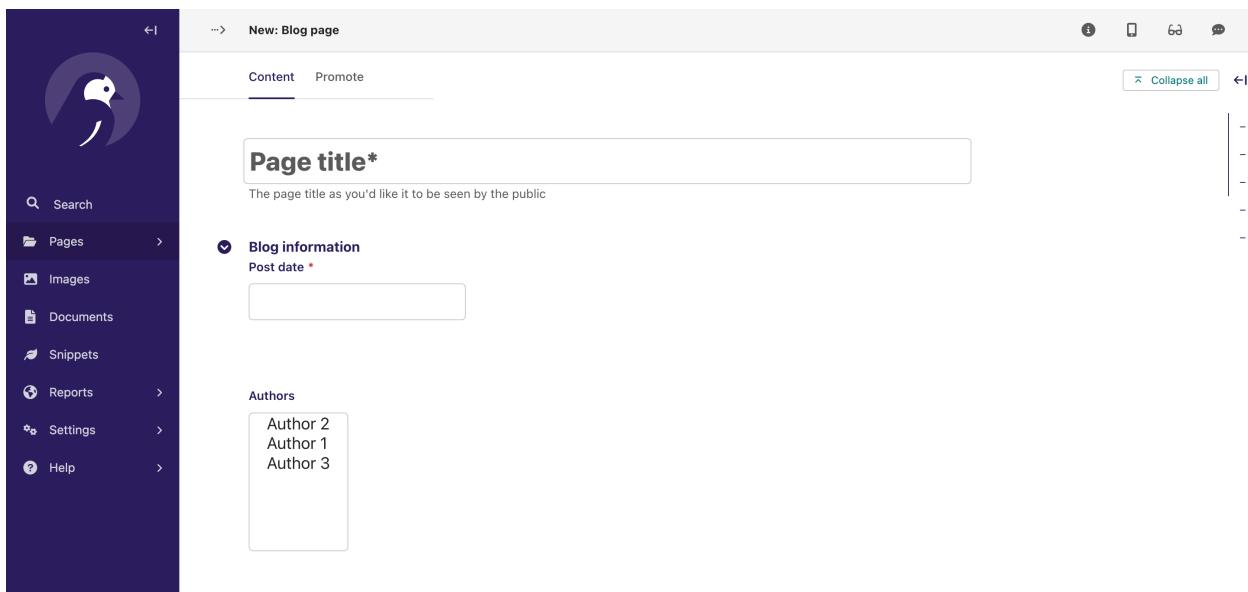
class BlogPage(Page):
    date = models.DateField("Post date")
    intro = models.CharField(max_length=250)
    body = RichTextField(blank=True)

    # Add this:
    authors = ParentalManyToManyField('blog.Author', blank=True)

    # ... Keep the main_image method and search_fields definition. Modify your
    # content_panels:
    content_panels = Page.content_panels + [
        MultiFieldPanel(["date", "authors"], heading="Blog information"),
        "intro", "body", "gallery_images"
    ]
```

Here you have used the `MultiFieldPanel` in `content_panels` to group the `date` and `authors` fields together for readability. By doing this, you are creating a single panel object that encapsulates multiple fields within a list or tuple into a single heading string. This feature is particularly useful for organizing related fields in the admin interface, making the UI more intuitive for content editors.

Migrate your database by running `python manage.py makemigrations` and `python manage.py migrate`, and then go to your [admin interface](#). Notice that the list of authors is presented as a multiple select box. This is the default representation for a multiple choice field - however, users often find a set of checkboxes to be more familiar and easier to work with.



You can do this by replacing the definition of "authors" in `content_panels` with a `FieldPanel` object. `FieldPanel("authors")` is equivalent to writing "`authors`", but allows passing additional optional arguments such as `widget`:

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

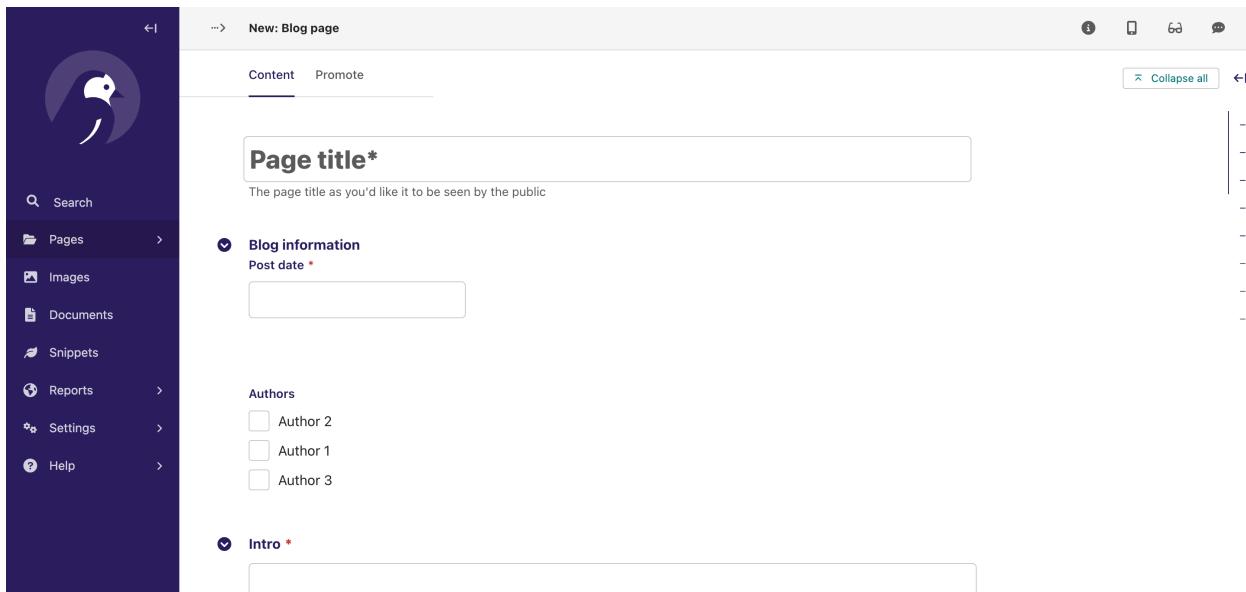
from modelcluster.fields import ParentalKey, ParentalManyToManyField
from wagtail.models import Page, Orderable
from wagtail.fields import RichTextField
from wagtail.admin.panels import FieldPanel, MultiFieldPanel
from wagtail.search import index
from wagtail.snippets.models import register_snippet

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

    authors = ParentalManyToManyField('blog.Author', blank=True)

    content_panels = Page.content_panels + [
        MultiFieldPanel([
            "date",
            # Change this:
            FieldPanel("authors", widget=forms.CheckboxSelectMultiple),
        ], heading="Blog information"),
        "intro", "body", "gallery_images"
    ]
```

In the preceding model modification, you used the `widget` keyword argument on the `FieldPanel` definition to specify a more user-friendly checkbox-based widget instead of the default list. Now go to your admin interface and you should see the author list displayed as a checklist.



Update the `blog_page.html` template to display the authors:

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

<!-- Add this: -->
{%- with authors=page.authors.all %}
  {%- if authors %}
    <h3>Posted by:</h3>
    <ul>
      {%- for author in authors %}
        <li style="display: inline">
          {%- image author.author_image fill-40x60 style="vertical-align: middle" %}
          {{ author.name }}
        </li>
      {%- endfor %}
    </ul>
  {%- endif %}
{%- endwith %}

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

{{ page.body|richtext }}

{%- for item in page.gallery_images.all %}
  <div style="float: inline-start; 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 %}
```

Add some authors to your blog posts, and publish them. Clicking on your blog posts from your blog index page should

now give you a page similar to this image:

Second Post

Sept. 3, 2022

Posted by:



Olivia Ava



Roberta Johnson

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](#)



Tag posts

Let's say you want to let editors "tag" their posts, so that readers can, for example, view all bicycle-related content together. For this, you have 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, you'll also need a working tag-specific URL view as well.

First, alter `models.py` once more:

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

# New imports added for ClusterTaggableManager, TaggedItemBase

from modelcluster.fields import ParentalKey, ParentalManyToManyField
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 model and add a new BlogPageTag model
class BlogPageTag(TaggedItemBase):
    content_object = ParentalKey(
```

(continues on next page)

(continued from previous page)

```

        'BlogPage',
        related_name='tagged_items',
        on_delete=models.CASCADE
    )

# Modify the BlogPage model:
class BlogPage(Page):
    date = models.DateField("Post date")
    intro = models.CharField(max_length=250)
    body = RichTextField(blank=True)
    authors = ParentalManyToManyField('blog.Author', blank=True)

    # Add this:
    tags = ClusterTaggableManager(through=BlogPageTag, blank=True)

    # ... Keep the main_image method and search_fields definition. Then modify the
→content_panels:
    content_panels = Page.content_panels + [
        MultiFieldPanel([
            "date",
            FieldPanel("authors", widget=forms.CheckboxSelectMultiple),

            # Add this:
            "tags",
        ], heading="Blog information"),
            "intro", "body", "gallery_images"
    ]
]

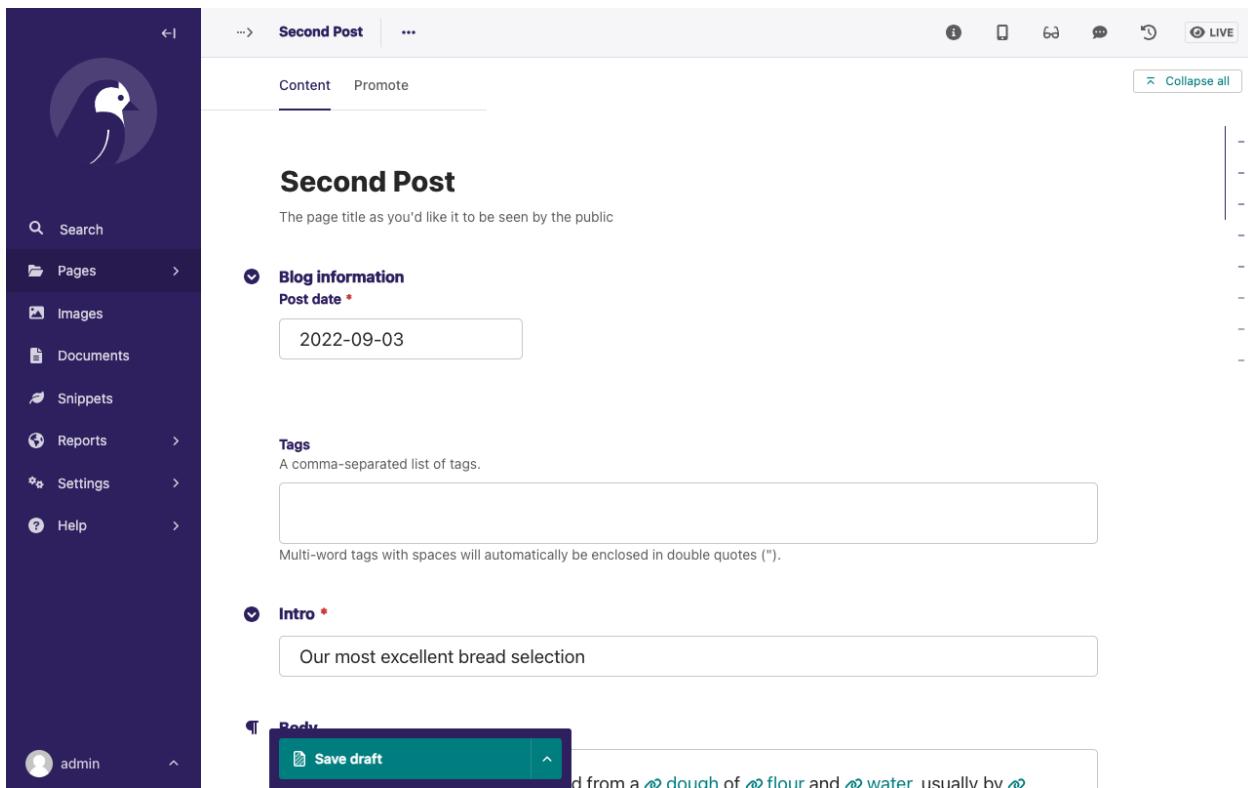
```

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

The changes you made can be summarized as follows:

- New `modelcluster` and `taggit` imports
- Addition of a new `BlogPageTag` model, and a `tags` field on `BlogPage`.

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`:

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

<!-- Add this: -->
{%
    with tags=page.tags.all %
        {% if tags %}
            <div class="tags">
                <h3>Tags</h3>
                {% for tag in tags %}
                    <a href="{% slugurl 'tags' %}?tag={{ tag }}><button type="button">{{ tag }}</button></a>
                {% endfor %}
            </div>
        {% endif %}
    {% endwith %}
}
```

Notice that you’re linking to pages here with the builtin `slugurl` tag rather than `pageurl`, which you 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’s unambiguous and avoids extra database lookups. But in the case of this loop, the Page object isn’t readily available, so you fall back on the less-preferred `slugurl` tag.

With the modifications that you’ve made so far, visiting a blog post with tags displays a series of linked buttons at the bottom, one for each tag associated with the post. However, clicking on a button will result in a **404** error page, as you are yet to define a “tags” view.

Return to `blog/models.py` and add a new `BlogTagIndexPage` model:

```
class BlogTagIndexPage(Page):

    def get_context(self, request):
```

(continues on next page)

(continued from previous page)

```
# 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. You can also override its `get_context()` method to add a `QuerySet` to the context dictionary, making it available to the template.

Migrate this by running `python manage.py makemigrations` and then `python manage.py migrate`. After migrating the new changes, create a new `BlogTagIndexPage` in the admin interface. To create the `BlogTagIndexPage`, follow the same process you followed in creating the `BlogIndexPage` and give it the slug “tags” on the Promote tab. This means the `BlogTagIndexPage` is a child of the home page and parallel to `Blog` in the admin interface.

Access `/tags` and Django will tell you what you probably already knew. You need to create the template, `blog/templates/blog/blog_tag_index_page.html` and add the following content to it:

```
{% 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 />
      <small>Revised: {{ blogpage.latest_revision_created_at }}</small><br />
    </p>

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

  {% endblock %}
```

In the preceding `blog_tag_index_page.html` template, you’re calling the built-in `latest_revision_created_at` field on the `Page` model. It’s handy to know this is always available.

Clicking the tag button at the bottom of a blog post renders a page 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.



Congratulations!

You completed this tutorial . Applaud yourself, and get yourself a cookie!

Thank you for reading and welcome to the Wagtail community!

Where next

- Read our [full tutorial](#) to transform your blog site into a fully deployable portfolio site.
- 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](#)

1.1.2 Quick install

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](#).

Dependencies needed for installation

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

Install Wagtail

Run the following commands in a virtual environment of your choice:

```
pip install wagtail
```

Once installed, Wagtail provides a `wagtail start` command to generate a new project:

```
wagtail start mysite
```

Running the command creates a new folder `mysite`, which is a template containing everything you need to get started. More information on this 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
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver
```

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

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

There are a few optional packages that are not installed by default. You can install them to improve performance or add features to Wagtail. These optional packages include:

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

Common quick install issues

Python is not available in path

```
python
> command not found: python
```

For detailed guidance, see this guide on [how to add Python to your path](#).

python3 not available

```
python3 -m pip install --upgrade pip
> command not found: python3
```

If this error occurs, the `python3` can be replaced with `py`.

1.1.3 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 the development of Wagtail itself.

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

1.1.4 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.

Note

We highly recommend working through the [Getting Started tutorial](#), even if you are not planning to create a standalone Wagtail project. This will ensure you have a good understanding of Wagtail concepts.

Wagtail is currently compatible with Django 4.2, 5.0 and 5.1. 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.py` 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 doesn't have one already:

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

Add MEDIA_ROOT and MEDIA_URL settings, if your project doesn't have these already:

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

Set the DATA_UPLOAD_MAX_NUMBER_FIELDS setting to 10000 or higher. This specifies the maximum number of fields allowed in a form submission, and it is recommended to increase this from Django's default of 1000, as particularly complex page models can exceed this limit within Wagtail's page editor:

```
DATA_UPLOAD_MAX_NUMBER_FIELDS = 10_000
```

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

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

Add a WAGTAILADMIN_BASE_URL - this is the base URL used by the Wagtail admin site. It is typically used for generating URLs to include in notification emails:

```
WAGTAILADMIN_BASE_URL = 'http://example.com'
```

If this setting is not present, Wagtail will fall back to `request.site.root_url` or to the hostname of the request. Although this setting is not strictly required, it is highly recommended because leaving it out may produce unusable URLs in notification emails.

Add a WAGTAILDOCS_EXTENSIONS setting to specify the file types that Wagtail will allow to be uploaded as documents. This can be omitted to allow all file types, but this may present a security risk if untrusted users are allowed to upload documents - see [User Uploaded Files](#).

```
WAGTAILDOCS_EXTENSIONS = ['csv', 'docx', 'key', 'odt', 'pdf', 'pptx', 'rtf', 'txt',
                           ↪'xlsx', 'zip']
```

Various other settings are available to configure Wagtail's behavior - 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)),
    ...
]
```

You can alter URL paths here 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 host the Wagtail admin at `/admin/`, but if this clashes with your project's existing admin backend then you can use an alternative path, such as `/cms/`.

Wagtail serves your document files from the location, `wagtaildocs_urls`. You can omit this if you do not intend to use Wagtail's document management features.

Wagtail serves your pages from the `wagtail_urls` location. In the above example, Wagtail handles URLs under `/pages/`, leaving your Django project to handle the root URL and other paths as normal. If you want Wagtail to handle the entire URL space including the root URL, then place `path('', include(wagtail_urls))` at the end of the `urlpatterns` list. Placing `path('', include(wagtail_urls))` at the end of the `urlpatterns` ensures that it doesn't override more specific URL patterns.

Finally, you need to set up your project 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 have 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 `python manage.py migrate` to create the database tables used by Wagtail.

User accounts

Wagtail uses Django's default user model by default. Superuser accounts receive automatic access to the Wagtail [admin interface](#); use `python manage.py createsuperuser` if you don't already have one. Wagtail supports custom user models with some restrictions. Wagtail uses an extension of Django's permissions framework, so your user model must at minimum inherit from `AbstractBaseUser` and `PermissionsMixin`.

Define page models and start developing

Before you can create pages, you must define one or more page models, as described in [Your first Wagtail site](#). The `wagtail` start project template provides a `home` app containing an initial `HomePage` model - when integrating Wagtail into an existing project, you will need to create this app yourself through `python manage.py startapp`. (Remember to add it to `INSTALLED_APPS` in your `settings.py` file.)

The initial “Welcome to your new Wagtail site!” page is a placeholder using the base `Page` model, and is not directly usable. After defining your own home page model, you should create a new page at the root level through the Wagtail admin interface, and set this as the site's homepage (under `Settings / Sites`). You can then delete the placeholder page.

1.1.5 The Zen of Wagtail

Wagtail is 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 find that Wagtail is in that sweet spot. However, as a piece of software, Wagtail can only take that mission so far. It's 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. You should expect to write code.

Always wear the right hat.

The key to using Wagtail effectively is to recognize 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 does the bulk of their work through the Wagtail [admin interface](#). A developer or designer spends 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 are probably loudly clamouring for that. However, 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 you can better achieve 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, and 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 allows you to 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 in 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. 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. Then you can 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:

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

While the sales pitch may make it appear appealing, the truth is that the average user of a content management system (CMS) would find it practically impossible to make such a fundamental change. This holds especially true for a live website, as it requires a deep understanding of programming and an awareness of the potential consequences of the change. As such, it will always be the programmer's job to negotiate that point-and-click interface. All you've done is take 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 recognizes that most programming tasks are best done by writing code, and doesn't 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 interface](#) 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 a CRM system or payment processor for instance. 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.

Wagtail is an open-source content management system (CMS) that's built on [Django](#), a popular Python web framework. It has gained popularity among developers and content editors for its powerful features and intuitive interface, providing a streamlined editing experience. Wagtail offers a comprehensive toolkit for content creation and management, including a rich text editor with formatting options, image and document management, version control, workflows, and content scheduling.

Developers appreciate Wagtail's highly customizable and modular architecture, which includes built-in support for Django's app structure. This allows them to easily create and integrate custom functionality, making Wagtail suitable for projects of any size. Wagtail excels in handling complex content structures, offering features like hierarchical page organization, robust search capabilities, and content localization.

Wagtail stands out as the preferred choice for tens of thousands of organizations globally, including renowned names like Google, the National Aeronautics and Space Administration (NASA), and the British National Health Service (NHS). It has proven scalability and is capable of handling high volumes of traffic from millions of visitors every month. What

sets Wagtail apart is its ability to extend beyond traditional content management, providing seamless integration with data tools and rich data visualizations.

For more information about Wagtail and the guiding principles for building websites with it, read [The Zen of Wagtail](#).

1.2 Tutorial

1.2.1 Customize your home page

Heads'up! Make sure you have completed [Your first Wagtail site](#) before going through this extended tutorial.

When building your portfolio website, the first step is to set up and personalize your homepage. The homepage is your chance to make an excellent first impression and convey the core message of your portfolio. So your homepage should include the following features:

1. **Introduction:** A concise introduction captures visitors' attention.
2. **Biography:** Include a brief biography that introduces yourself. This section should mention your name, role, expertise, and unique qualities.
3. **Hero Image:** This may be a professional headshot or other image that showcases your work and adds visual appeal.
4. **Call to Action (CTA):** Incorporate a CTA that guides visitors to take a specific action, such as “View Portfolio,” “Hire Me,” or “Learn More”.
5. **Resume:** This is a document that provides a summary of your education, work experience, achievements, and qualifications.

In this section, you'll learn how to add features **1** through **4** to your homepage. You'll add your resume or CV later in the tutorial.

Now, modify your `home/models.py` file to include the following:

```
from django.db import models

from wagtail.models import Page
from wagtail.fields import RichTextField

# import MultiFieldPanel:
from wagtail.admin.panels import FieldPanel, MultiFieldPanel


class HomePage(Page):
    # add the Hero section of HomePage:
    image = models.ForeignKey(
        "wagtailimages.Image",
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name="+",
        help_text="Homepage image",
    )
    hero_text = models.CharField(
        blank=True,
        max_length=255, help_text="Write an introduction for the site"
    )
    hero_cta = models.CharField(
        blank=True,
```

(continues on next page)

(continued from previous page)

```
verbose_name="Hero CTA",
max_length=255,
help_text="Text to display on Call to Action",
)
hero_cta_link = models.ForeignKey(
    "wagtailcore.Page",
    null=True,
    blank=True,
    on_delete=models.SET_NULL,
    related_name="+",
    verbose_name="Hero CTA link",
    help_text="Choose a page to link to for the Call to Action",
)

body = RichTextField(blank=True)

# modify your content_panels:
content_panels = Page.content_panels + [
    MultiFieldPanel(
        [
            FieldPanel("image"),
            FieldPanel("hero_text"),
            FieldPanel("hero_cta"),
            FieldPanel("hero_cta_link"),
        ],
        heading="Hero section",
    ),
    FieldPanel('body'),
]
```

You might already be familiar with the different parts of your `HomePage` model. The `image` field is a `ForeignKey` referencing Wagtail's built-in `Image` model for storing images. Similarly, `hero_cta_link` is a `ForeignKey` to `wagtailcore.Page`. The `wagtailcore.Page` is the base class for all other page types in Wagtail. This means all Wagtail pages inherit from `wagtailcore.Page`. For instance, your class `HomePage(Page)` inherits from `wagtailcore.Page`.

Using `on_delete=models.SET_NULL` ensures that if you remove an image or hero link from your admin interface, the `image` or `hero_cta_link` fields on your `Homepage` will be set to null, but the rest of the data will be preserved. Read the [Django documentation on the `on_delete` attribute](#) for more details.

By default, Django creates a reverse relation between the models when you have a `ForeignKey` field within your model. Django also generates a name for this reverse relation using the model name and the `_set` suffix. You can use the default name of the reverse relation to access the model with the `ForeignKey` field from the referenced model.

You can override this default naming behavior and provide a custom name for the reverse relationship by using the `related_name` attribute. For example, if you want to access your `HomePage` from `wagtailimages.Image`, you can use the value you provided for your `related_name` attribute.

However, when you use `related_name="+"`, you create a connection between models without creating a reverse relation. In other words, you're instructing Django to create a way to access `wagtailimages.Image` from your `Homepage` but not a way to access `HomePage` from `wagtailimages.Image`.

While `body` is a `RichTextField`, `hero_text` and `hero_cta` are `CharField`, a Django string field for storing short text.

The [Your First Wagtail Tutorial](#) already explained `content_panels`. `FieldPanel` and `MultiPanel` are types of Wagtail built-in `Panels`. They're both subclasses of the base `Panel` class and accept all of Wagtail's `Panel` parameters in addition to their own. While the `FieldPanel` provides a widget for basic Django model fields, `MultiFieldPanel` helps

you decide the structure of the editing form. For example, you can group related fields.

Now that you understand the different parts of your `HomePage` model, migrate your database by running `python manage.py makemigrations` and then `python manage.py migrate`

After migrating your database, start your server by running `python manage.py runserver`.

Add content to your homepage

To add content to your homepage through the admin interface, follow these steps:

1. Log in to your [admin interface](#), with your admin username and password.
2. Click Pages.
3. Click the **pencil** icon beside **Home**.
4. Choose an image, choose a page, and add data to the input fields.

Note

You can choose your home page or blog index page to link to your Call to Action. You can choose a more suitable page later in the tutorial.

5. Publish your Home page.

You have all the necessary data for your Home page now. You can visit your Home page by going to `http://127.0.0.1:8000` in your browser. You can't see all your data, right? That's because you must modify your `Homepage` template to display the data.

Replace the content of your `home/templates/home/home_page.html` file with the following:

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

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

{% block content %}
<div>
    <h1>{{ page.title }}</h1>
    {% image page.image fill-480x320 %}
    <p>{{ page.hero_text }}</p>
    {% if page.hero_cta_link %}
        <a href="{% pageurl page.hero_cta_link %}">
            {% firstof page.hero_cta page.hero_cta_link.title %}
        </a>
    {% endif %}
</div>

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

In your `Homepage` template, notice the use of `firstof` in line 13. It's helpful to use this tag when you have created a series of fallback options, and you want to display the first one that has a value. So, in your template, the `firstof` template tag displays `page.hero_cta` if it has a value. If `page.hero_cta` doesn't have a value, then it displays `page.hero_cta_link.title`.

Congratulations! You've completed the first stage of your Portfolio website .

1.2.2 Create a footer for all pages

The next step is to create a footer for all pages of your portfolio site. You can display social media links and other information in your footer.

Add a base app

Now, create a general-purpose app named `base`. To generate the `base` app, run the command:

```
python manage.py startapp base
```

After generating the `base` app, you must install it on your site. In your `mysite/settings/base.py` file, add "`base`" to the `INSTALLED_APPS` list.

Create navigation settings

Now, go to your `base/models.py` file and add the following lines of code:

```
from django.db import models
from wagtail.admin.panels import (
    FieldPanel,
    MultiFieldPanel,
)
from wagtail.contrib.settings.models import (
    BaseGenericSetting,
    register_setting,
)

@register_setting
class NavigationSettings(BaseGenericSetting):
    linkedin_url = models.URLField(verbose_name="LinkedIn URL", blank=True)
    github_url = models.URLField(verbose_name="GitHub URL", blank=True)
    mastodon_url = models.URLField(verbose_name="Mastodon URL", blank=True)

    panels = [
        MultiFieldPanel(
            [
                FieldPanel("linkedin_url"),
                FieldPanel("github_url"),
                FieldPanel("mastodon_url"),
            ],
            "Social settings",
        )
    ]
```

In the preceding code, the `register_setting` decorator registers your `NavigationSettings` models. You used the `BaseGenericSetting` base model class to define a settings model that applies to all web pages rather than just one page.

Now, migrate your database by running the commands `python manage.py makemigrations` and `python manage.py migrate`. After migrating your database, reload your `admin` interface. You'll get the error '`wagtailsettings`' is not a registered namespace. This is because you haven't installed the `wagtail.contrib.settings` module.

The `wagtail.contrib.settings` module defines models that hold common settings across all your web pages. So, to successfully import the `BaseGenericSetting` and `register_setting`, you must install the `wagtail`.

`contrib.settings` module on your site. To install `wagtail.contrib.settings`, go to your `mysite/settings/base.py` file and add "`wagtail.contrib.settings`" to the `INSTALLED_APPS` list:

```
INSTALLED_APPS = [
    # ...
    # Add this line to install wagtail.contrib.settings:
    "wagtail.contrib.settings",
]
```

Also, you have to register the `settings` context processor. Registering `settings` context processor makes site-wide settings accessible in your templates. To register the `settings` context processor, modify your `mysite/settings/base.py` file as follows:

```
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",

            # Add this to register the _settings_ context processor:
            "wagtail.contrib.settings.context_processors.settings",
        ],
    },
},
]
```

Add your social media links

To add your social media links, reload your admin interface and click **Settings** from your [Sidebar](#). You can see your **Navigation Settings**. Clicking the **Navigation Settings** gives you a form to add your social media account links.

Display social media links

You must provide a template to display the social media links you added through the admin interface.

Create an `includes` folder in your `mysite/templates` folder. Then in your newly created `mysite/templates/includes` folder, create a `footer.html` file and add the following to it:

```
<footer>
    <p>Built with Wagtail</p>

    {%- with linkedin_url=settings.base.NavigationSettings.linkedin_url github_
    url=settings.base.NavigationSettings.github_url mastodon_url=settings.base.
    NavigationSettings.mastodon_url %}
        {%- if linkedin_url or github_url or mastodon_url %}
            <p>
                Follow me on:

```

(continues on next page)

(continued from previous page)

```
{% if github_url %}
    <a href="{{ github_url }}>GitHub</a>
{% endif %}
{% if linkedin_url %}
    <a href="{{ linkedin_url }}>LinkedIn</a>
{% endif %}
{% if mastodon_url %}
    <a href="{{ mastodon_url }}>Mastodon</a>
{% endif %}
</p>
{% endif %}
{% endwith %}
</footer>
```

Now, go to your `mysite/templates/base.html` file and modify it as follows:

```
<body class="{% block body_class %}{% endblock %}>
    {% wagtailuserbar %}

    {% block content %}{% endblock %}

    {# Add this to the file: #}
    {% include "includes/footer.html" %}

    {# Global javascript #}
    <script type="text/javascript" src="{% static 'js/mysite.js' %}"></script>

    {% block extra_js %}
        {# Override this in templates to add extra javascript #}
    {% endblock %}
</body>
```

Now, reload your [homepage](#). You'll see your social media links at the bottom of your homepage.

1.2.3 Create editable footer text with Wagtail Snippets

Having only your social media links in your portfolio footer isn't ideal. You can add other items, like site credits and copyright notices, to your footer. One way to do this is to use the Wagtail [snippet](#) feature to create an editable footer text in your admin interface and display it in your site's footer.

To add a footer text snippet to your admin interface, modify your `base/models.py` file as follows:

```
from django.db import models
from wagtail.admin.panels import (
    FieldPanel,
    MultiFieldPanel,

    # import PublishingPanel:
    PublishingPanel,
)

# import RichTextField:
from wagtail.fields import RichTextField

# import DraftStateMixin, PreviewableMixin, RevisionMixin, TranslatableMixin:
```

(continues on next page)

(continued from previous page)

```

from wagtail.models import (
    DraftStateMixin,
    PreviewableMixin,
    RevisionMixin,
    TranslatableMixin,
)

from wagtail.contrib.settings.models import (
    BaseGenericSetting,
    register_setting,
)

# import register_snippet:
from wagtail.snippets.models import register_snippet

# ...keep the definition of the NavigationSettings model and add the FooterText model:
@register_snippet
class FooterText(
    DraftStateMixin,
    RevisionMixin,
    PreviewableMixin,
    TranslatableMixin,
    models.Model,
):
    body = RichTextField()

    panels = [
        FieldPanel("body"),
        PublishingPanel(),
    ]

    def __str__(self):
        return "Footer text"

    def get_preview_template(self, request, mode_name):
        return "base.html"

    def get_preview_context(self, request, mode_name):
        return {"footer_text": self.body}

    class Meta(TranslatableMixin.Meta):
        verbose_name_plural = "Footer Text"

```

In the preceding code, the `FooterText` class inherits from several Mixins, the `DraftStateMixin`, `RevisionMixin`, `PreviewableMixin`, and `TranslatableMixin`. In Django, Mixins are reusable pieces of code that define additional functionality. They are implemented as Python classes, so you can inherit their methods and properties.

Since your `FooterText` model is a Wagtail snippet, you must manually add Mixins to your model. This is because snippets aren't Wagtail Pages in their own right. Wagtail Pages don't require Mixins because they already have them.

`DraftStateMixin` is an abstract model that you can add to any non-page Django model. You can use it for drafts or unpublished changes. The `DraftStateMixin` requires `RevisionMixin`.

`RevisionMixin` is an abstract model that you can add to any non-page Django model to save revisions of its instances. Every time you edit a page, Wagtail creates a new `Revision` and saves it in your database. You can use `Revision` to find the history of all the changes that you make. `Revision` also provides a place to keep new changes before they go

live.

`PreviewableMixin` is a Mixin class that you can add to any non-page Django model to preview any changes made. `TranslatableMixin` is an abstract model you can add to any non-page Django model to make it translatable.

Also, with Wagtail, you can set publishing schedules for changes you made to a Snippet. You can use a Publishing-Panel to schedule revisions in your `FooterText`.

The `__str__` method defines a human-readable string representation of an instance of the `FooterText` class. It returns the string “Footer text”.

The `get_preview_template` method determines the template for rendering the preview. It returns the template name “`base.html`”.

The `get_preview_context` method defines the context data that you can use to render the preview template. It returns a key “`footer_text`” with the content of the body field as its value.

The `Meta` class holds metadata about the model. It inherits from the `TranslatableMixin.Meta` class and sets the `verbose_name_plural` attribute to “*Footer Text*”.

Now, migrate your database by running `python manage.py makemigrations` and `python manage.py migrate`. After migrating, restart your server and then reload your admin interface. You can now find **Snippets** in your Sidebar.

Add footer text

To add your footer text, go to your [admin interface](#). Click **Snippets** in your Sidebar and add your footer text.

Display your footer text

In this tutorial, you’ll use a custom template tag to display your footer text.

In your `base` folder, create a `templatetags` folder. Within your new `templatetags` folder, create the following files:

- `__init__.py`
- `navigation_tags.py`

Leave your `base/templatetags/__init__.py` file blank and add the following to your `base/templatetags/navigation_tags.py` file:

```
from django import template

from base.models import FooterText

register = template.Library()

@register.inclusion_tag("base/includes/footer_text.html", takes_context=True)
def get_footer_text(context):
    footer_text = context.get("footer_text", "")

    if not footer_text:
        instance = FooterText.objects.filter(live=True).first()
        footer_text = instance.body if instance else ""

    return {
```

(continues on next page)

(continued from previous page)

```

    "footer_text": footer_text,
}

```

In the preceding code, you imported the `template` module. You can use it to create and render template tags and filters. Also, you imported the `FooterText` model from your `base/models.py` file.

`register = template.Library()` creates an instance of the `Library` class from the `template` module. You can use this instance to register custom template tags and filters.

`@register.inclusion_tag("base/includes/footer_text.html", takes_context=True)` is a decorator that registers an inclusion tag named `get_footer_text`. `"base/includes/footer_text.html"` is the template path that you'll use to render the inclusion tag. `takes_context=True` indicates that the context of your `footer_text.html` template will be passed as an argument to your inclusion tag function.

The `get_footer_text` inclusion tag function takes a single argument named `context`. `context` represents the template context where you'll use the tag.

`footer_text = context.get("footer_text", "")` tries to retrieve a value from the context using the key `footer_text`. The `footer_text` variable stores any retrieved value. If there is no `footer_text` value within the context, then the variable stores an empty string `""`.

The `if` statement in the `get_footer_text` inclusion tag function checks whether the `footer_text` exists within the context. If it doesn't, the `if` statement proceeds to retrieve the first published instance of the `FooterText` from the database. If a published instance is found, the statement extracts the `body` content from it. However, if there's no published instance available, it defaults to an empty string.

Finally, the function returns a dictionary containing the `"footer_text"` key with the value of the retrieved `footer_text` content. You'll use this dictionary as context data when rendering your `footer_text` template.

To use the returned dictionary, create a `templates/base/includes` folder in your `base` folder. Then create a `footer_text.html` file in your `base/templates/base/includes/` folder and add the following to it:

```

{%- load wagtailcore_tags %}



{{ footer_text|richtext }}


```

Add your `footer_text` template to your footer by modifying your `mysite/templates/includes/footer.html` file:

```

{# Load navigation_tags at the top of the file: #}
{%- load navigation_tags %}

<footer>
    <p>Built with Wagtail</p>

    {%- with linkedin_url=settings.base.NavigationSettings.linkedin_url github_
    url=settings.base.NavigationSettings.github_url mastodon_url=settings.base.
    NavigationSettings.mastodon_url %}
        {%- if linkedin_url or github_url or mastodon_url %}
            <p>
                Follow me on:
                {%- if github_url %}
                    <a href="{{ github_url }}>GitHub</a>
                {%- endif %}
                {%- if linkedin_url %}
                    <a href="{{ linkedin_url }}>LinkedIn</a>
                {%- endif %}
            </p>
        {%- endif %}
    {%- endwith %}

```

(continues on next page)

(continued from previous page)

```

    {%- endif %}
    {%- if mastodon_url %}
        <a href="{{ mastodon_url }}>Mastodon</a>
    {%- endif %}
    </p>
    {%- endif %}
{%- endwith %}

{# Add footer_text: #}
{{ get_footer_text }}
</footer>
```

Now, restart your server and reload your [homepage](#). For more information on how to render your Wagtail snippets, read [Rendering snippets](#).

Well done! You now have a footer across all pages of your portfolio site. In the next section of this tutorial, you'll learn how to set up a site menu for linking to your homepage and other pages as you add them.

1.2.4 Set up site menu for linking to the homepage and other pages

This section of the tutorial will teach you how to create a site menu to link to your homepage and other pages as you add them. The site menu will appear across all pages of your portfolio website, just like your footer.

Start by creating a template tag in your `base/templatetags/navigation_tags.py` file:

```

from django import template

# import site:
from wagtail.models import Site

from base.models import FooterText

register = template.Library()

# ... keep the definition of get_footer_text and add the get_site_root template tag:
@register.simple_tag(takes_context=True)
def get_site_root(context):
    return Site.find_for_request(context["request"]).root_page
```

In the preceding code, you created the `get_site_root` template tag to retrieve the root page of your site, which is your `HomePage` in this case.

Now, create `mysite/templates/includes/header.html` and add the following to it:

```

{%- load wagtailcore_tags navigation_tags %}

<header>
    {%- get_site_root as site_root %}
    <nav>
        <p>
            <a href="{% pageurl site_root %}">Home</a> |
            {%- for menuitem in site_root.get_children.live.in_menu %}
                <a href="{% pageurl menuitem %}">{{ menuitem.title }}</a>
            {%- endfor %}
        </p>
```

(continues on next page)

(continued from previous page)

```
</nav>
</header>
```

In the preceding template you loaded the `wagtailcore_tags` and `navigation_tags`. With these tags, you can generate navigation menus for your Wagtail project.

`{% get_site_root as site_root %}` retrieves your HomePage and assigns it to the variable `site_root`. `Home` | creates a link to your HomePage by using the `pageurl` template tag with `site_root` as an argument. It generates a link to your HomePage, with the label **Home**, followed by a pipe symbol |, to separate the menu items.

`{% for menuitem in site_root.get_children.live.in_menu %}` is a loop that iterates through the child pages of your HomePage that are live and included in the menu.

Finally, add your `header` template to your base template by modifying your `mysite/templates/base.html` file:

```
<body class="{% block body_class %}{% endblock %}>
  {% wagtailuserbar %}

  {% # Add your header template to your base template: %}
  {% include "includes/header.html" %}

  {% block content %}{% endblock %}

  {% include "includes/footer.html" %}

  {% # Global javascript %}

  <script type="text/javascript" src="{% static 'js/mysite.js' %}"></script>

  {% block extra_js %}
    {% # Override this in templates to add extra javascript %}
  {% endblock %}
</body>
```

Now, if you restart your server and reload your homepage, you'll see your site menu with a link to your homepage labeled as **Home**.

Add pages to your site menu

You can add any top-level, like your Home page, Blog index page, Portfolio page, and Form page to the site menu by doing the following:

1. Go to your admin interface.
2. Go to any top-level page.
3. Click **Promote**.
4. Check the **Show in menus** checkbox.

In the next section of this tutorial, we'll show you how to style your site and improve its user experience.

1.2.5 Style and improve user experience

In this tutorial, you'll add a basic site theme to your portfolio site and improve its user experience.

Add styles

To style your site, navigate to your `mysite/static/css/mysite.css` file and add the following:

```
*,
::before,
::after {
    box-sizing: border-box;
}

html {
    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", system-ui, Roboto,
    "Helvetica Neue", Arial, sans-serif, Apple Color Emoji, "Segoe UI Emoji", "Segoe UI",
    "Symbol", "Noto Color Emoji";
}

body {
    min-height: 100vh;
    max-width: 800px;
    margin: 0 auto;
    padding: 10px;
    display: grid;
    gap: 3vw;
    grid-template-rows: min-content 1fr min-content;
}

a {
    color: currentColor;
}

footer {
    border-top: 2px dotted;
    text-align: center;
}

header {
    border-bottom: 2px dotted;
}

.template-homepage main {
    text-align: center;
}
```

Now, reload your portfolio site to reflect the styles.

Note

If your webpage's styles do not update after reloading, then you may need to clear your browser cache.

Improve user experience

There are several ways to improve the user experience of your portfolio site.

Start by modifying your `mysite/templates/base.html` file as follows:

```

{# Remove wagtailuserbar: #}
{%- load static wagtailcore_tags %}

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>
            {%- block title %}{{ page.seo_title }}{{ page.title }}{%- endblock %}
            {%- if page.seo_title %}{{ page.seo_title }}{%- else %}{{ page.title }}{%- endif %}
            {%- endblock %}
            {%- block title_suffix %}
                {%- wagtail_site as current_site %}
                {%- if current_site and current_site.site_name %}- {{ current_site.site_name }}{%- endif %}
                {%- endblock %}
        </title>
        {%- if page.search_description %}
        <meta name="description" content="{{ page.search_description }}" />
        {%- endif %}
        <meta name="viewport" content="width=device-width, initial-scale=1" />

        {# Force all links in the live preview panel to be opened in a new tab #}
        {%- if request.in_preview_panel %}
        <base target="_blank">
        {%- endif %}

        {# Add supported color schemes: #}
        <meta name="color-scheme" content="light dark">

        {# Add a favicon with inline SVG: #}
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 100 100%22><text y=%22.9em%22 font-size=%2290%22>%</text></svg>" />

        {# Global stylesheets #}
        <link rel="stylesheet" type="text/css" href="{% static 'css/mysite.css' %}">

        {%- block extra_css %}
        {# Override this in templates to add extra stylesheets #}
        {%- endblock %}
    </head>

    <body class="{%- block body_class %}{%- endblock %}">
        {# Remove wagtailuserbar: #}

        {%- include "includes/header.html" %}

        {# Wrap your block content within a <main> HTML5 tag: #}
        <main>
            {%- block content %}{%- endblock %}

```

(continues on next page)

(continued from previous page)

```
</main>

{%
    include "includes/footer.html"
}

{# Global javascript #}

<script type="text/javascript" src="{% static 'js/mysite.js' %}"></script>

{%
    block extra_js
    # Override this in templates to add extra javascript #
    %endblock%
}
</body>
</html>
```

In the preceding template, you made the following modifications:

1. You removed `wagtailuserbar` from your base template. You'll add the `wagtailuserbar` to your header template later in the tutorial. This change improves the user experience for keyboard and screen reader users.
2. You Added `<meta name="color-scheme" content="light dark">` to inform the browser about the supported color schemes for your site. This makes your site adapt to both dark and light themes.
3. You used the `<link>` tag to add a favicon to your portfolio site using inline SVG.
4. You wrapped the `{% block content %}` and `{% endblock %}` tags with a `<main>` HTML5 tag. The `<main>` tag is a semantic HTML5 tag used to indicate the main content of a webpage.

Also, you should dynamically get your HomePage's title to use in your site menu instead of hardcoding it in your template. You should include the child pages of the Home page in your site menu if they have their 'Show in menus' option checked. Finally, you want to ensure that you add the `wagtailuserbar` that you removed from your base template to your header template. This will improve users' experience for keyboard and screen reader users.

To make the improvements mentioned in the preceding paragraph, modify your `mysite/templates/includes/header.html` file as follows:

```
{# Load wagtailuserbar: #}
{%
    load wagtailcore_tags navigation_tags wagtailuserbar
}

<header>
    {% get_site_root as site_root %}
    <nav>
        <p>
            <a href="{% pageurl site_root %}">{{ site_root.title }}</a> |
            {% for menuitem in site_root.get_children.live.in_menu %}
                {# Add the child pages of your HomePage that have their 'Show in menu' checked #}
                <a href="{% pageurl menuitem %}">{{ menuitem.title }}</a>{% if not forloop.last %} | {% endif %}
            {% endfor %}
        </p>
    </nav>
    {# Add wagtailuserbar: #}
    {% wagtailuserbar "top-right" %}
</header>
```

Another way you can improve user experience is by adding a skip link for keyboard users. A skip link is a web accessibility

feature that enhances the browsing experience for keyboard navigators and screen readers. The skip link will let your users jump directly to the main content.

To add a skip-link, add the following styles to your `mysite/static/css/mysite.css` file:

```
.skip-link {
    position: absolute;
    top: -30px;
}

.skip-link:focus-visible {
    top: 5px;
}
```

After adding the styles, go to your `mysite/templates/base.html` file and add a unique identifier:

```
{% include "includes/header.html" %}

{# Add a unique identifier: #}
<main id="main">
    {% block content %}{% endblock %}
</main>
```

Finally, go to your `mysite/templates/includes/header.html` file and modify it as follows:

```
{% load wagtailcore_tags navigation_tags wagtailuserbar %}

<header>
    {# Add this: #}
    <a href="#main" class="skip-link">Skip to content</a>

    {% get_site_root as site_root %}
    <nav>
        <p>
            <a href="{% pageurl site_root %}">{{ site_root.title }}</a> |
            {% for menuitem in site_root.get_children.live.in_menu %}
                <a href="{% pageurl menuitem %}">{{ menuitem.title }}</a>{% if not forloop.
            -last %} | {% endif %}
            {% endfor %}
        </p>
    </nav>
    {% wagtailuserbar "top-right" %}
</header>
```

In the preceding template, you added an `<a>` (anchor) element to create a *Skip to content* link. You set the `href` attribute to `#main`. The internal anchor links to your base template's `main` element.

Well done! Now, you know how to style a Wagtail site. The next section will teach you how to create a contact page for your portfolio site.

1.2.6 Create contact page

Having a contact page on your portfolio site will help you connect with potential clients, employers, or other professionals who are interested in your skills.

In this section of the tutorial, you'll add a contact page to your portfolio site using Wagtail forms.

Start by modifying your `base/models.py` file:

```
from django.db import models

# import ParentalKey:
from modelcluster.fields import ParentalKey

# import FieldRowPanel and InlinePanel:
from wagtail.admin.panels import (
    FieldPanel,
    FieldRowPanel,
    InlinePanel,
    MultiFieldPanel,
    PublishingPanel,
)

from wagtail.fields import RichTextField
from wagtail.models import (
    DraftStateMixin,
    PreviewableMixin,
    RevisionMixin,
    TranslatableMixin,
)

# import AbstractEmailForm and AbstractFormField:
from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField

# import FormSubmissionsPanel:
from wagtail.contrib.forms.panels import FormSubmissionsPanel
from wagtail.contrib.settings.models import (
    BaseGenericSetting,
    register_setting,
)
from wagtail.snippets.models import register_snippet


# ... keep the definition of NavigationSettings and FooterText. Add FormField and
# FormPage:
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 + [
        FormSubmissionsPanel(),
        FieldPanel('intro'),
        InlinePanel('form_fields', label="Form fields"),
    ]
```

(continues on next page)

(continued from previous page)

```

FieldPanel('thank_you_text'),
MultiFieldPanel([
    FieldRowPanel([
        FieldPanel('from_address'),
        FieldPanel('to_address'),
    ]),
    FieldPanel('subject'),
], "Email"),
]

```

In the preceding code, your `FormField` model inherits from `AbstractFormField`. With `AbstractFormField`, you can define any form field type of your choice in the admin interface. `page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields')` defines a parent-child relationship between the `FormField` and `FormPage` models.

On the other hand, your `FormPage` model inherits from `AbstractEmailForm`. Unlike `AbstractFormField`, `AbstractEmailForm` offers a form-to-email capability. Also, it defines the `to_address`, `from_address`, and `subject` fields. It expects a `form_fields` to be defined.

After defining your `FormField` and `FormPage` models, you must create `form_page` and `form_page_landing` templates. The `form_page` template differs from a standard Wagtail template because it's passed a variable named `form` containing a Django Form object in addition to the usual `Page` variable. The `form_page_landing.html`, on the other hand, is a standard Wagtail template. Your site displays the `form_page_landing.html` after a user makes a successful form submission.

Now, create a `base/templates/base/form_page.html` file and add the following to it:

```

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

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

{% block content %}
<h1>{{ page.title }}</h1>
<div>{{ page.intro|richtext }}</div>

<form class="page-form" action="{% pageurl page %}" method="POST">
    {% csrf_token %}
    {{ form.as_div }}
    <button type="Submit">Submit</button>
</form>
{% endblock content %}

```

Also, create a `base/templates/base/form_page_landing.html` file and add the following to it:

```

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

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

{% block content %}
<h1>{{ page.title }}</h1>
<div>{{ page.thank_you_text|richtext }}</div>
{% endblock content %}

```

Now, you've added all the necessary lines of code and templates that you need to create a contact page on your portfolio website.

Now, migrate your database by running `python manage.py makemigrations` and then `python manage.py migrate`.

Add your contact information

To add contact information to your portfolio site, follow these steps:

1. Create a **Form page** as a child page of **Home** by following these steps:
 - a. Restart your server.
 - b. Go to your admin interface.
 - c. Click **Pages** in your **Sidebar**.
 - d. Click **Home**.
 - e. Click the + icon (Add child page) at the top of the resulting page.
 - f. Click **Form page**.
2. Add the necessary data.
3. Publish your Form Page.

Style your contact page

To style your contact page, add the following CSS to your `mysite/static/css/mysite.css` file:

```
.page-form label {  
    display: block;  
    margin-top: 10px;  
    margin-bottom: 5px;  
}  
  
.page-form :is(textarea, input, select) {  
    width: 100%;  
    max-width: 500px;  
    min-height: 40px;  
    margin-top: 5px;  
    margin-bottom: 10px;  
}  
  
.page-form .helptext {  
    font-style: italic;  
}
```

In the next section of this tutorial, you'll learn how to add a portfolio page to your site.

1.2.7 Create a portfolio page

A portfolio page is a web page that has your resume or Curriculum Vitae (CV). The page will give potential employers a chance to review your work experience.

This tutorial shows you how to add a portfolio page to your portfolio site using the Wagtail StreamField.

First, let's explain what StreamField is.

What is StreamField?

StreamField is a feature that was created to balance the need for developers to have well-structured data and the need for content creators to have editorial flexibility in how they create and organize their content.

In traditional content management systems, there's often a compromise between structured content and giving editors the freedom to create flexible layouts. Typically, Rich Text fields are used to give content creators the tools they need to make flexible and versatile content. Rich Text fields can provide a WYSIWYG editor for formatting. However, Rich Text fields have limitations.

One of the limitations of Rich Text fields is the loss of semantic value. Semantic value in content denotes the underlying meaning or information conveyed by the structure and markup of content. When content lacks semantic value, it becomes more difficult to determine its intended meaning or purpose. For example, when editors use Rich Text fields to style text or insert multimedia, the content might not be semantically marked as such.

So, StreamField gives editors more flexibility and addresses the limitations of Rich Text fields. StreamField is a versatile content management solution that treats content as a sequence of blocks. Each block represents different content types like paragraphs, images, and maps. Editors can arrange and customize these blocks to create complex and flexible layouts. Also, StreamField can capture the semantic meaning of different content types.

Create reusable custom blocks

Now that you know what StreamField is, let's guide you through using it to add a portfolio page to your site.

Start by adding a new app to your portfolio site by running the following command:

```
python manage.py startapp portfolio
```

Install your new portfolio app to your site by adding “*portfolio*” to the `INSTALLED_APPS` list in your `mysite/settings/base.py` file.

Now create a `base/blocks.py` file and add the following lines of code to it:

```
from wagtail.blocks import (
    CharBlock,
    ChoiceBlock,
    RichTextBlock,
    StreamBlock,
    StructBlock,
)
from wagtail.embeds.blocks import EmbedBlock
from wagtail.images.blocks import ImageBlock

class CaptionedImageBlock(StructBlock):
    image = ImageBlock(required=True)
    caption = CharBlock(required=False)
    attribution = CharBlock(required=False)

    class Meta:
        icon = "image"
        template = "base/blocks/captioned_image_block.html"

class HeadingBlock(StructBlock):
    heading_text = CharBlock(classname="title", required=True)
    size = ChoiceBlock()
```

(continues on next page)

(continued from previous page)

```
choices=[  
    ("", "Select a heading size"),  
    ("h2", "H2"),  
    ("h3", "H3"),  
    ("h4", "H4"),  
,  
    blank=True,  
    required=False,  
)  
  
class Meta:  
    icon = "title"  
    template = "base/blocks/heading_block.html"  
  
class BaseStreamBlock(StreamBlock):  
    heading_block = HeadingBlock()  
    paragraph_block = RichTextBlock(icon="pilcrow")  
    image_block = CaptionedImageBlock()  
    embed_block = EmbedBlock()  
        help_text="Insert a URL to embed. For example, https://www.youtube.com/watch?  
→v=SGJFWirQ3ks",  
        icon="media",  
)
```

In the preceding code, you created reusable Wagtail custom blocks for different content types in your general-purpose app. You can use these blocks across your site in any order. Let's take a closer look at each of these blocks.

First, `CaptionedImageBlock` is a block that editors can use to add images to a StreamField section.

```
class CaptionedImageBlock(StructBlock):  
    image = ImageBlock(required=True)  
    caption = CharBlock(required=False)  
    attribution = CharBlock(required=False)  
class Meta:  
    icon = "image"  
    template = "base/blocks/captioned_image_block.html"
```

`CaptionedImageBlock` inherits from `StructBlock`. With `StructBlock`, you can group several child blocks together under a single parent block. Your `CaptionedImageBlock` has three child blocks. The first child block, `Image`, uses the `ImageBlock` field block type. With `ImageBlock`, editors can select an existing image or upload a new one. Its `required` argument has a value of `true`, which means that you must provide an image for the block to work. The `caption` and `attribution` child blocks use the `CharBlock` field block type, which provides single-line text inputs for adding captions and attributions to your images. Your `caption` and `attribution` child blocks have their `required` attributes set to `false`. That means you can leave them empty in your `admin interface` if you want to.

Just like `CaptionedImageBlock`, your `HeadingBlock` also inherits from `StructBlock`. It has two child blocks. Let's look at those.

```
class HeadingBlock(StructBlock):  
    heading_text = CharBlock(classname="title", required=True)  
    size = ChoiceBlock()  
        choices=[  
            ("", "Select a heading size"),  
            ("h2", "H2"),  
            ("h3", "H3"),  
            ("h4", "H4"),  
,
```

(continues on next page)

(continued from previous page)

```

        ],
        blank=True,
        required=False,
    )
    class Meta:
        icon = "title"
        template = "base/blocks/heading_block.html"

```

The first child block, `heading_text`, uses `CharBlock` for specifying the heading text, and it's required. The second child block, `size`, uses `ChoiceBlock` for selecting the heading size. It provides options for `h2`, `h3`, and `h4`. Both `blank=True` and `required=False` make the heading text optional in your `admin` interface.

Your `BaseStreamBlock` class inherits from `StreamBlock`. `StreamBlock` defines a set of child block types that you would like to include in all of the `StreamField` sections across a project. This class gives you a baseline collection of common blocks that you can reuse and customize for all the different page types where you use `StreamField`. For example, you will definitely want editors to be able to add images and paragraph text to all their pages, but you might want to create a special pull quote block that is only used on blog pages.

```

class BaseStreamBlock(StreamBlock):
    heading_block = HeadingBlock()
    paragraph_block = RichTextBlock(icon="pilcrow")
    image_block = CaptionedImageBlock()
    embed_block = EmbedBlock(
        help_text="Insert a URL to embed. For example, https://www.youtube.com/watch?v=SGJFWirQ3ks",
        icon="media",
    )

```

Your `BaseStreamBlock` has four child blocks. The `heading_block` uses the previously defined `HeadingBlock`. `paragraph_block` uses `RichTextBlock`, which provides a WYSIWYG editor for creating formatted text. `image_block` uses the previously defined `CaptionedImageBlock` class. `embed_block` is a block for embedding external content like videos. It uses the Wagtail `EmbedBlock`. To discover more field block types that you can use, read the [documentation on Field block types](#).

Also, you defined a `Meta` class within your `CaptionedImageBlock` and `HeadingBlock` blocks. The `Meta` classes provide metadata for the blocks, including icons to visually represent them in the `admin` interface. The `Meta` classes also include custom templates for rendering your `CaptionedImageBlock` and `HeadingBlock` blocks.

Note

Wagtail provides built-in templates to render each block. However, you can override the built-in template with a custom template.

Finally, you must add the custom templates that you defined in the `Meta` classes of your `CaptionedImageBlock` and `HeadingBlock` blocks.

To add the custom template of your `CaptionedImageBlock`, create a `base/templates/base/blocks/captioned_image_block.html` file and add the following to it:

```

{%- load wagtailimages_tags %}

<figure>
    {%- image self.image fill-600x338 loading="lazy" %}
    <figcaption>{{ self.caption }} - {{ self.attribution }}</figcaption>
</figure>

```

To add the custom template of your HeadingBlock block, create a base/templates/base/blocks/heading_block.html file and add the following to it:

```
{% if self.size == 'h2' %}  
  <h2>{{ self.heading_text }}</h2>  
{% elif self.size == 'h3' %}  
  <h3>{{ self.heading_text }}</h3>  
{% elif self.size == 'h4' %}  
  <h4>{{ self.heading_text }}</h4>  
{% endif %}
```

Note

You can also create a custom template for a child block. For example, to create a custom template for embed_block, create a base/templates/base/blocks/embed_block.html file and add the following to it:

```
{{ self }}
```

Use the blocks you created in your portfolio app

You can use the reusable custom blocks you created in your general-purpose base app across your site. However, it's conventional to define the blocks you want to use in a blocks.py file of the app you intend to use them in. Then you can import the blocks from your app's blocks.py file to use them in your models.py file.

Now create a portfolio/blocks.py file and import the block you intend to use as follows:

```
from base.blocks import BaseStreamBlock  
  
class PortfolioStreamBlock(BaseStreamBlock):  
    pass
```

The preceding code defines a custom block named PortfolioStreamBlock, which inherits from BaseStreamBlock. The pass statement indicates a starting point. Later in the tutorial, you'll add custom block definitions and configurations to the PortfolioStreamBlock.

Now add the following to your portfolio/models.py file:

```
from wagtail.models import Page  
from wagtail.fields import StreamField  
from wagtail.admin.panels import FieldPanel  
  
from portfolio.blocks import PortfolioStreamBlock  
  
class PortfolioPage(Page):  
    parent_page_types = ["home.HomePage"]  
  
    body = StreamField(  
        PortfolioStreamBlock(),  
        blank=True,  
        use_json_field=True,  
        help_text="Use this section to list your projects and skills.",  
    )  
  
    content_panels = Page.content_panels + [
```

(continues on next page)

(continued from previous page)

```
    FieldPanel("body"),
]
```

In the preceding code, you defined a Wagtail Page named `PortfolioPage`. `parent_page_types = ["home.HomePage"]` specifies that your Portfolio page can only be a child page of Home Page. Your `body` field is a StreamField, which uses the `PortfolioStreamBlock` custom block that you imported from your `portfolio/blocks.py` file. `blank=True` indicates that you can leave this field empty in your admin interface. `help_text` provides a brief description of the field to guide editors.

Your next step is to create a template for your `PortfolioPage`. To do this, create a `portfolio/templates/portfolio/portfolio_page.html` file and add the following to it:

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

{% load wagtailcore_tags wagtailimages_tags %}

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

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

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

Now migrate your database by running `python manage.py makemigrations` and then `python manage.py migrate`.

Add more custom blocks

To add more custom blocks to your `PortfolioPage`'s body, modify your `portfolio/blocks.py` file:

```
# import CharBlock, ListBlock, PageChooserBlock, PageChooserBlock, RichTextBlock, and
# StructBlock:
from wagtail.blocks import (
    CharBlock,
    ListBlock,
    PageChooserBlock,
    RichTextBlock,
    StructBlock,
)

# import ImageBlock:
from wagtail.images.blocks import ImageBlock

from base.blocks import BaseStreamBlock

# add CardBlock:
class CardBlock(StructBlock):
    heading = CharBlock()
    text = RichTextBlock(features=["bold", "italic", "link"])
    image = ImageBlock(required=False)

    class Meta:
        icon = "form"
        template = "portfolio/blocks/card_block.html"
```

(continues on next page)

(continued from previous page)

```
# add FeaturedPostsBlock:
class FeaturedPostsBlock(StructBlock):
    heading = CharBlock()
    text = RichTextBlock(features=["bold", "italic", "link"], required=False)
    posts = ListBlock(PageChooserBlock(page_type="blog.BlogPage"))

    class Meta:
        icon = "folder-open-inverse"
        template = "portfolio/blocks/featured_posts_block.html"

class PortfolioStreamBlock(BaseStreamBlock):
    # delete the pass statement

    card = CardBlock(group="Sections")
    featured_posts = FeaturedPostsBlock(group="Sections")
```

In the preceding code, CardBlock has three child blocks, heading, text and image. You are already familiar with the field block types used by the child pages.

However, in your FeaturedPostsBlock, one of the child blocks, posts, uses ListBlock. ListBlock is a structural block type that you can use for multiple sub-blocks of the same type. You used it with PageChooserBlock to select only the Blog Page type pages. To better understand structural block types, read the [Structural block types documentation](#).

Furthermore, icon = "form" and icon = "folder-open-inverse" define custom block icons to set your blocks apart in the admin interface. For more information about block icons, read the [documentation on block icons](#).

You used group="Sections" in card = CardBlock(group="Sections") and featured_posts = FeaturedPostsBlock(group="Sections") to categorize your card and featured_posts child blocks together within a category named section.

You probably know what your next step is. You have to create templates for your CardBlock and FeaturedPostsBlock.

To create a template for CardBlock, create a portfolio/templates/portfolio/blocks/card_block.html file and add the following to it:

```
{% load wagtailcore_tags wagtailimages_tags %}


<h3>{{ self.heading }}</h3>
    <div>{{ self.text|richtext }}</div>
    {% if self.image %}
        {% image self.image width-480 %}
    {% endif %}
</div>


```

To create a template for featured_posts_block, create a portfolio/templates/portfolio/blocks/featured_posts_block.html file and add the following to it:

```
{% load wagtailcore_tags %}


<h2>{{ self.heading }}</h2>
    {% if self.text %}
        <p>{{ self.text|richtext }}</p>
    {% endif %}


```

(continues on next page)

(continued from previous page)

```
<div class="grid">
    {%
        for page in self.posts %}
            <div class="card">
                <p><a href="{% pageurl page %}">{{ page.title }}</a></p>
                <p>{{ page.specific.date }}</p>
            </div>
    {% endfor %}
</div>
</div>
```

Finally, migrate your changes by running `python manage.py makemigrations` and then `python manage.py migrate`.

Add your resume

To add your resume to your portfolio site, follow these steps:

1. Create a **Portfolio Page** as a child page of **Home** by following these steps:
 - a. Restart your server.
 - b. Go to your admin interface.
 - c. Click **Pages** in your **Sidebar**.
 - d. Click **Home**.
 - e. Click the + icon (Add child page) at the top of the resulting page.
 - f. Click **Portfolio Page**.
2. Add your resume data by following these steps:
 - a. Use “Resume” as your page title.
 - b. Click + to expand your body section.
 - c. Click **Paragraph block**.
 - d. Copy and paste the following text in your new **Paragraph block**:

I'm a Wagtail Developer with a proven track record of developing and maintaining complex web applications. I have experience writing custom code to extend Wagtail applications, collaborating with other developers, and integrating third-party services and APIs.

- e. Click + below your preceding Paragraph block, and then click **Paragraph block** to add a new Paragraph Block.
- f. Type “/” in the input field of your new Paragraph block and then click **H2 Heading 2**.
- g. Use “Work Experience” as your Heading 2.
- h. Type “/” below your Heading 2 and click **H3 Heading 3**.
- i. Use the following as your Heading 3:

Wagtail developer at Birdwatchers Inc, United Kingdom

- j. Type the following after your Heading 3:

January 2022 to November 2023

- Developed and maintained a complex web application using Wagtail, resulting in a 25% increase in user engagement and a 20% increase in revenue within the first year.
- Wrote custom code to extend Wagtail applications, resulting in a 30% reduction in development time and a 15% increase in overall code quality.
- Collaborated with other developers, designers, and stakeholders to integrate third-party services and APIs, resulting in a 40% increase in application functionality and user satisfaction.
- Wrote technical documentation and participated in code reviews, providing feedback to other developers and improving overall code quality by 20%.

Note

By starting your sentences with “-”, you’re writing out your work experience as a Bulleted list. You can achieve the same result by typing “/” in the input field of your Paragraph block and then clicking **Bulleted list**.

k. Click + below your Work experience. l. Click **Paragraph block** to add another Paragraph block. m. Type “/” in the input field of your new Paragraph block and then click **H2 Heading 2**. n. Use “Skills” as the Heading 2 of your new Paragraph block. o. Copy and paste the following after your Heading 2:

```
Python, Django, Wagtail, HTML, CSS, Markdown, Open-source management, Trello, Git,  
→ GitHub
```

3. Publish your Portfolio Page.

Congratulations!  You now understand how to create complex flexible layouts with Wagtail StreamField. In the next section of this tutorial, you’ll learn how to add search functionality to your site.

1.2.8 Add search to your site

Using the Wagtail `start` command to start your project gives you a built-in search app. This built-in search app provides a simple search functionality for your site.

However, you can customize your search template to suit your portfolio site. To customize your search template, go to your `search/templates/search.html` file and modify it as follows:

```
{% extends "base.html" %}  
{% load static wagtailcore_tags %}  
  
{% block body_class %}template-searchresults{% endblock %}  
  
{% block title %}Search{% endblock %}  
  
{% block content %}  
<h1>Search</h1>  
  
<form action="{% url 'search' %}" method="get">  
    <input type="text" name="query" {% if search_query %} value="{{ search_query }}" {%  
    endif %}>  
    <input type="submit" value="Search" class="button">  
</form>  
  
{% if search_results %}  
  
    # Add this paragraph to display the details of results found:  
    <p>You searched{% if search_query %} for "{{ search_query }}"{% endif %}, {{ search_  
    results.paginator.count }} result{{ search_results.paginator.count|pluralize }}  
    →found.</p>  
  
    # Replace the <ul> HTML element with the <ol> html element:  
<ol>  
    {% for result in search_results %}  
        <li>  
            <h4><a href="{{ pageurl result }}>{{ result }}</a></h4>  
            {% if result.search_description %}  
                {{ result.search_description }}  
            {% endif %}  
        </li>  
    {% endfor %}  
</ol>
```

(continues on next page)

(continued from previous page)

```

    {%- endif %}
  </li>
  {%- endfor %}
</ol>

{# Improve pagination by adding: #}
{%- if search_results.paginator.num_pages > 1 %}
  <p>Page {{ search_results.number }} of {{ search_results.paginator.num_pages }},_
  showing {{ search_results|length }} result{{ search_results|pluralize }} out of {{_
  search_results.paginator.count }}</p>
{%- endif %}

{%- if search_results.has_previous %}
Previous</a>
{%- endif %}

{%- if search\_results.has\_next %}
Next</a>
{%- endif %}

{%- elif search\\_query %}
No results found
{%- endif %}
{%- endblock %}

```

Now, let's explain the customizations you made in the preceding template:

1. You used `<p>You searched{{ if search_query %} for “{{ search_query }}”{{ endif %}}, {{ search_results.paginator.count }} result{{ search_results.paginator.count|pluralize }} found.</p>` to display the search query, the number of results found. You also used it to display the plural form of “result” if more than one search result is found.
2. You replaced the `` HTML element with the `` HTML element. The `` HTML element contains a loop iterating through each search result and displaying them as list items. Using `` gives you numbered search results.
3. You improved the pagination in the template. `{% if search_results.paginator.num_pages > 1 %}` checks if there is more than one page of search results. If there is more than one page of search results, it displays the current page number, the total number of pages, the number of results on the current page, and the total number of results. `{% if search_results.has_previous %}` and `{% if search_results.has_next %}` checks if there are previous and next pages of search results. If they exist, it displays “Previous” and “Next” links with appropriate URLs for pagination.

Now, you want to display your search across your site. One way to do this is to add it to your header. Go to your `mysite/templates/includes/header.html` file and modify it as follows:

```

{%- load wagtailcore_tags navigation_tags wagtailuserbar %}

<header>
  <a href="#main" class="skip-link">Skip to content</a>
  {%- get_site_root as site_root %}
  <nav>
    <p>
      <a href="{% pageurl site_root %}">{{ site_root.title }}</a> |
      {%- for menuitem in site_root.get_children.live.in_menu %}

```

(continues on next page)

(continued from previous page)

```
<a href="#"><% pageurl menuitem %}>{{ menuitem.title }}</a>{%
  if not
  forloop.last %} | {%
  endif %}
  {%
  endfor %}

  {# Display your search by adding this: #}
  | <a href="/search/">Search</a>
</p>
</nav>

{%
  wagtailuserbar "top-right" %
}
</header>
```

Well done! You now have a fully deployable portfolio site. The next section of this tutorial will walk you through how to deploy your site.

1.2.9 Deploy your site

So far, you've been accessing your site locally. Now, it's time to deploy it. Deployment makes your site publicly accessible on the Internet by hosting it on a production server.

To deploy your site, you'll first need to choose a *hosting provider*, then follow their Wagtail deployment guide.

If you're unsure, we have a tutorial for using *Fly.io with Backblaze*.

Congratulations! You made it to the end of the tutorial!

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*

Congratulations on completing *Your first Wagtail site* tutorial! Now that you've completed the beginner tutorial and built a blog site from scratch, you should have a solid understanding of the basic building blocks of a Wagtail website. We hope you enjoyed learning all about Wagtail.

Now that you can build a blog site with Wagtail, why stop there? We created this extended tutorial to help you grow your Wagtail knowledge.

In this tutorial, you'll transform your blog site into a fully deployable portfolio site. You must first complete the *Your First Wagtail Site* tutorial before you begin this extended tutorial.

You'll learn the following in this tutorial:

- How to add pagination to your Wagtail website
- How to use Wagtail StreamField
- How to use Wagtail documents
- How to use snippets across multiple web pages
- How to use Wagtail forms
- How to implement the search feature in a Wagtail website

- How to deploy a Wagtail website

Now, let's dive in.

1.3 Usage guide

1.3.1 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,
        on_delete=models.SET_NULL,
        related_name='+'
    )

    # Search index configuration
```

(continues on next page)

(continued from previous page)

```
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](#)

Structural

These are used for structuring fields in the interface.

- [MultiFieldPanel](#)
- [InlinePanel](#)
- [FieldRowPanel](#)

Customizing the page editor interface

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

Parent page / subpage type rules

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

Both parent and subpage types 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 behavior.

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 %}` or `{% fullpageurl %}` template tags. Since it’s called from a template, these automatically includes the optimizations mentioned below.

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

Customizing 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

You can call the `Page.get_url(request)` method whenever you need a page URL. 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, it would return a full URL including the protocol and domain. Whenever possible, you should include the optional `request` argument 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 uses `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\(\)`](#).

To retrieve the full URL (including the protocol and domain), use `Page.get_full_url(request)`. 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.

Customizing 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:
```

(continues on next page)

(continued from previous page)

```
        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 allows the nesting of other models within a 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.

An inline model must have a `ParentalKey` pointing to the parent model. It can also inherit from `wagtail.models.Orderable` to allow reordering of items in the admin interface.

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`. For a brief description of parameters taken by `InlinePanel`, see [InlinePanel](#).

Re-using inline models across multiple page types

In the above example, related links are defined as a child object on the `BlogPage` page type. Often, the same kind of inline child object will appear on several page types, and in these cases, it's undesirable to repeat the entire model definition. This can be avoided by refactoring the common fields into an abstract model:

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

# The abstract model for related links, complete with panels
class RelatedLink(models.Model):
    name = models.CharField(max_length=255)
    url = models.URLField()

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

    class Meta:
        abstract = True

# The real model which extends the abstract model with a ParentalKey relation back to
# the page model.
# This can be repeated for each page type where the relation is to be added
# (for example, NewsPageRelatedLink, PublicationPageRelatedLink and so on).
class BlogPageRelatedLink(Orderable, RelatedLink):
    page = ParentalKey(BlogPage, on_delete=models.CASCADE, related_name='related_links')
```

Alternatively, if `RelatedLink` is going to appear on a significant number of the page types defined in your project, it may be more appropriate to set up a single `RelatedLink` model pointing to the base `wagtailcore.Page` model:

```
class RelatedLink(Orderable):
    page = ParentalKey("wagtailcore.Page", on_delete=models.CASCADE, related_name=
    'related_links')
    name = models.CharField(max_length=255)
    url = models.URLField()
    panels = [
        FieldPanel('name'),
        FieldPanel('url'),
    ]
```

This will then make `related_links` available as a relation across all page types, although it will still only be editable on page types that include the `InlinePanel` in their panel definitions - for other page types, the set of related links will remain empty.

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:<br/>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_
<br/>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()
```

1.3.2 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 of (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` model, 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/
```

(continues on next page)

(continued from previous page)

```
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.

Images in multiple formats

The `picture` tag works like `image`, but allows specifying multiple formats to generate a `<picture>` element with `<source>` elements and a fallback ``.

For example:

```
{% load wagtailimages_tags %}  
...  
{% picture page.photo format-{avif,webp,jpeg} width-400 %}
```

See [Multiple formats](#) for full documentation.

Images in multiple sizes

The `srcset_image` tag works like `image`, but allows specifying multiple sizes to generate a `srcset` attribute and leverage [responsive image rules](#).

For example:

```
{% load wagtailimages_tags %}  
...  
{% srcset_image page.photo width-{400,800} sizes="(max-width: 600px) 400px, 80vw" %}
```

This can also be done with `picture`, to generate multiple formats and sizes at once:

```
{% picture page.photo format-{avif,webp,jpeg} width-{400,800} sizes="80vw" %}
```

See [Responsive images](#) for full documentation.

Rich text (filter)

The `richtext` filter takes a chunk of HTML content and renders it as safe HTML on 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 %}
```

fullpageurl

Takes a Page object and returns its absolute URL (`http://example.com/foo/bar/`).

```
{% load wagtailcore_tags %}  
...  
<meta property="og:url" content="{% fullpageurl page %}" />
```

Much like `pageurl`, a `fallback` keyword argument may be provided.

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)

The `static` tag is used to load anything from your static files directory. Use of this tag avoids rewriting all static paths if hosting arrangements change, such as when moving from development to a live environment.

```
{% 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

The `wagtailuserbar` 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 to allow keyboard users to 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 user bar #
    <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 user bar in each corner of the screen:

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

The user bar 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::part(userbar) {
    bottom: 30px;
}
```

To customize the items shown in the user bar, you can use the `construct_wagtail_userbar` hook.

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() {
        (continues on next page)
```

(continued from previous page)

```
...  
</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*.

Template fragment caching

Django supports [template fragment caching](#), which allows caching portions of a template. Using Django's `{% cache %}` tag natively with Wagtail can be [dangerous](#) as it can result in preview content being shown to end users. Instead, Wagtail provides 2 extra template tags which can be loaded from `wagtail_cache`:

Preview-aware caching

The `{% wagtailcache %}` tag functions similarly to Django's `{% cache %}` tag, but will neither cache nor serve cached content when previewing a page (or other model) in Wagtail.

```
{% load wagtail_cache %}  
  
{% wagtailcache 500 sidebar %}  
    <!-- sidebar -->  
{% endwagtailcache %}
```

Much like `{% cache %}`, you can use `make_template_fragment_key` to obtain the cache key.

Page-aware caching

`{% wagtailpagecache %}` is an extension of `{% wagtailcache %}`, but is also aware of the current page and site, and includes those as part of the cache key. This makes it possible to easily add caching around parts of the page without worrying about the page it's on. `{% wagtailpagecache %}` intentionally makes assumptions - for more customization it's recommended to use `{% wagtailcache %}`.

```
{% load wagtail_cache %}  
  
{% wagtailpagecache 500 hero %}  
    <!-- hero -->  
{% endwagtailpagecache %}
```

This is identical to:

```
{% wagtail_site as current_site %}  
  
{% wagtailcache 500 hero page.cache_key current_site.id %}  
    <!-- hero -->  
{% endwagtailcache %}
```

Note the use of the page's [cache key](#), which ensures that when a page is updated, the cache is automatically invalidated.

If you want to obtain the cache key, you can use `make_wagtail_template_fragment_key` (based on Django's `make_template_fragment_key`):

```
from django.core.cache import cache
from wagtail.coreutils import make_wagtail_template_fragment_key

key = make_wagtail_template_fragment_key("hero", page, site)
cache.delete(key) # invalidates cached template fragment
```

1.3.3 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: --&gt;
{% image page.photo width-400 %}

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

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.

Multiple formats

To render an image in multiple formats, you can use the `picture` tag:

```
{% picture page.photo format-{avif,webp,jpeg} width-400 %}
```

Compared to `image`, this will render a `<picture>` element with a fallback `` within and one `<source>` element per extra format. The browser picks the first format it supports, or defaults to the fallback `` element. For example, the above will render HTML similar to:

```
<picture>
  <source srcset="/media/images/pied-wagtail.width-400.avif" type="image/avif">
  <source srcset="/media/images/pied-wagtail.width-400.webp" type="image/webp">
  
</picture>
```

In this case, if the browser supports the AVIF format it will load the AVIF file. Otherwise, if the browser supports the WebP format, it will try to load the WebP file. If none of those formats are supported, the browser will load the JPEG image. The order of the provided formats isn't configurable – Wagtail will always output source elements in the following order: AVIF, WebP, JPEG, PNG, and GIF. This ensures the most optimized format is provided whenever possible.

The `picture` tag can also be used with multiple image resize rules to generate responsive images.

Responsive images

Wagtail provides `picture` and `srcset_image` template tags which can generate image elements with `srcset` attributes. This allows browsers to select the most appropriate image file to load based on responsive image rules.

The syntax for `srcset_image` is the same as `image`, with two exceptions:

```
{% srcset_image [image] [resize-rule-with-brace-expansion] sizes="[my source sizes]" %}
```

- The resize rule should be provided with multiple sizes in a brace-expansion pattern, like `width-{200, 400}`. This will generate the `srcset` attribute, with as many URLs as there are sizes defined in the resize rule, and one width descriptor per URL. The first provided size will always be used as the `src` attribute, and define the image's width and height attributes, as a fallback.
- The `sizes` attribute is essential. This tells the browser how large the image will be displayed on the page, so that it can select the most appropriate image to load.

Here is an example of `srcset_image` in action, generating a `srcset` attribute:

```
{% srcset_image page.photo width-{400, 800} sizes="(max-width: 600px) 400px, 80vw" %}
```

This outputs:

```

```

Here is an example with the `picture` tag:

```
{% picture page.photo format-{avif, webp, jpeg} width-{400, 800} sizes="80vw" %}
```

This outputs:

```
<picture>
  <source sizes="80vw" srcset="/media/images/pied-wagtail.width-400.avif 400w, /media/images/pied-wagtail.width-800.avif 800w" type="image/avif">
  <source sizes="80vw" srcset="/media/images/pied-wagtail.width-400.webp 400w, /media/images/pied-wagtail.width-800.webp 800w" type="image/webp">
  
</picture>
```

Available resizing methods

The available resizing methods are as follows:

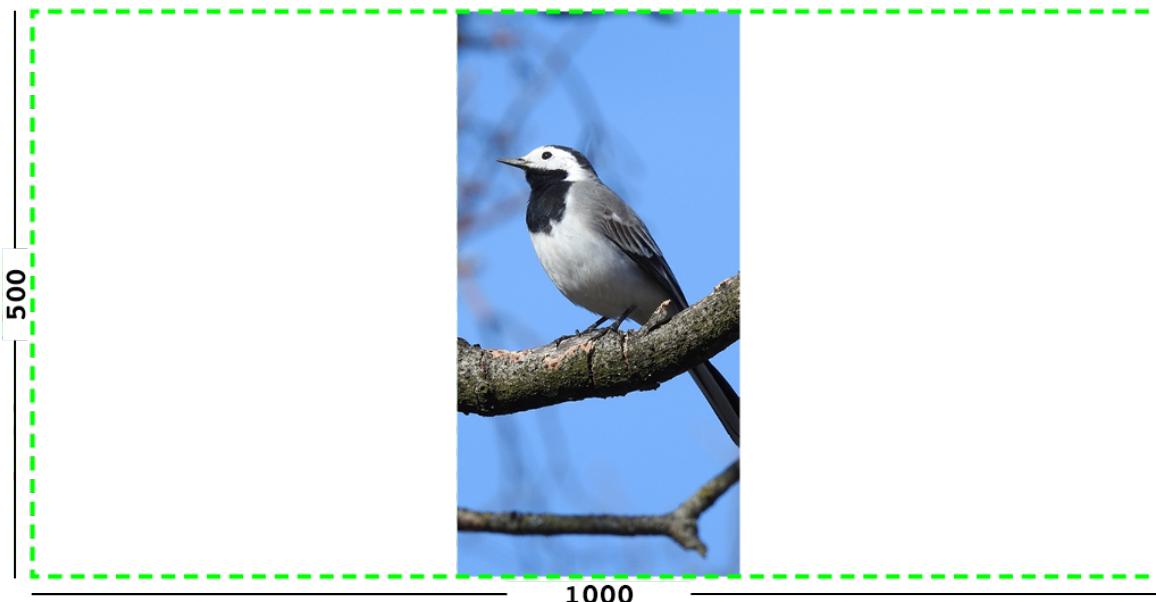
`max`

(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.

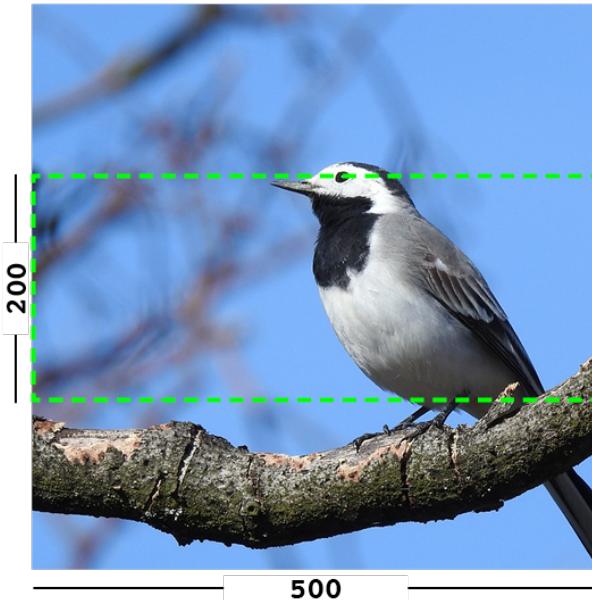
`min`

(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 a percentage)

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

Resize the image to the percentage specified.

fill

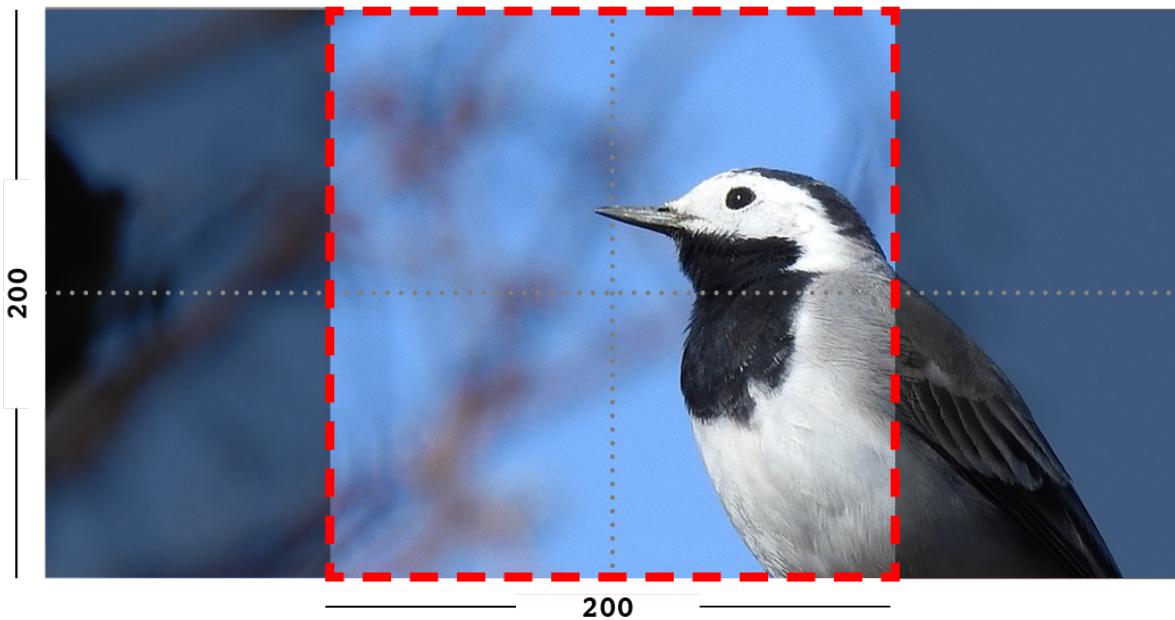
(takes two dimensions and an optional `-c` parameter)

```
{% 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 center 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.

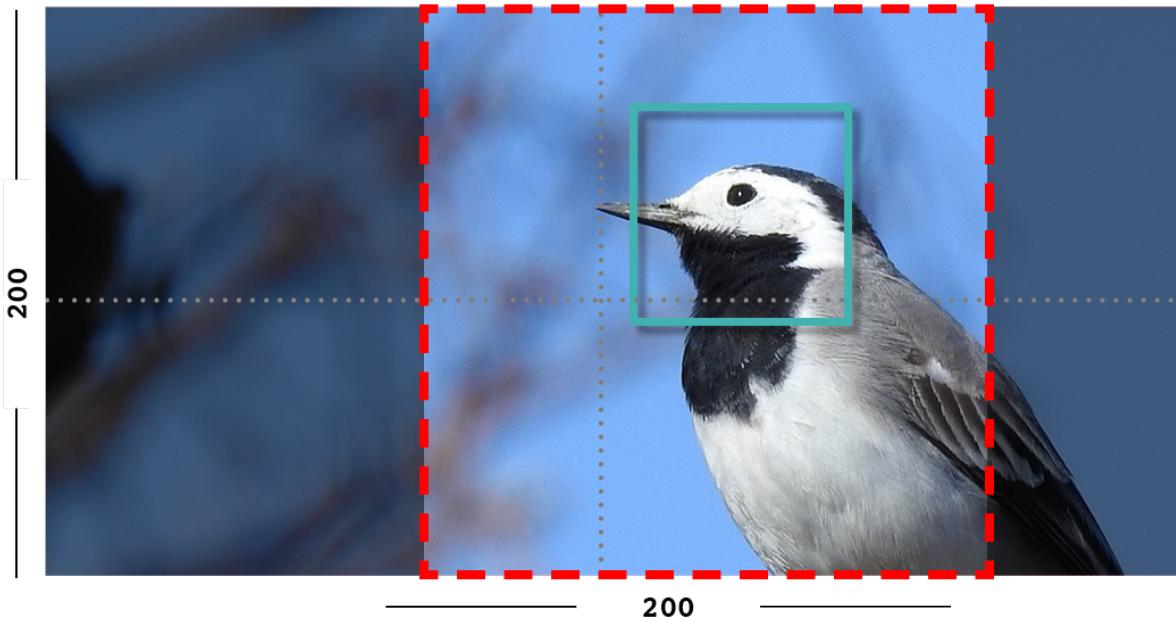
In some cases (for example thumbnails), it may be preferable to crop closer to the focal point, so that the subject of the image is more prominent.

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`:

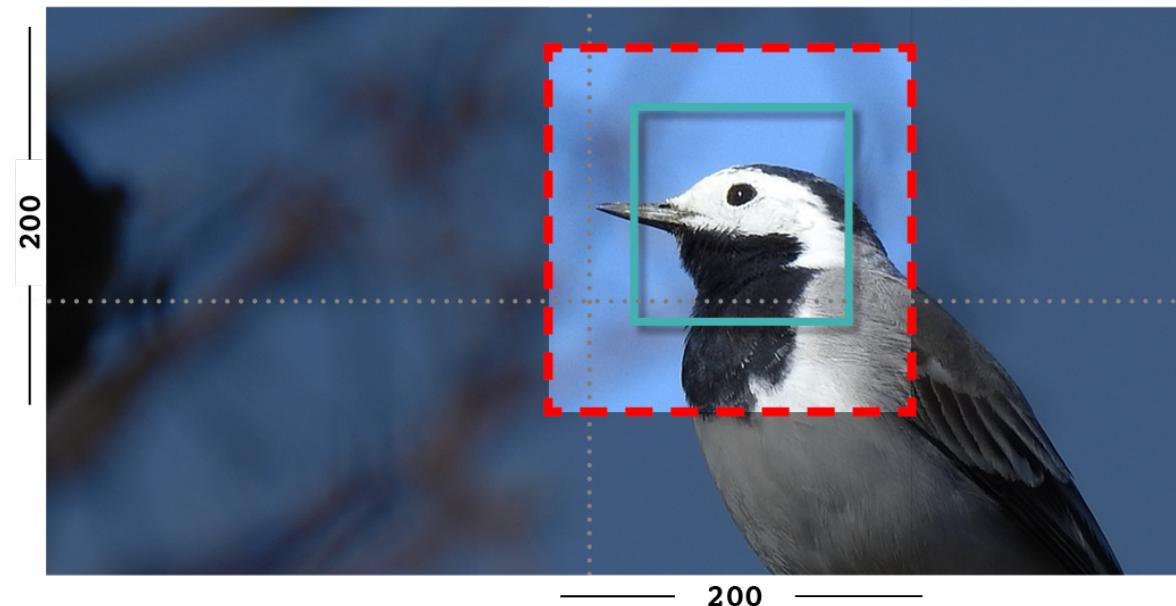
```
{% image page.photo fill-200x200-c100 %}
```

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 center 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 the `-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, to access the underlying image Rendition (`tmp_photo`):

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


```

This is also possible with the `srcset_image` tag, to retrieve multiple size renditions:

```
{% srcset_image page.photo width={200,400} as tmp_photo %}


```

(continues on next page)

(continued from previous page)

```
    class="my-custom-class"
/>>
```

And with the picture tag, to retrieve multiple formats:

```
{% picture page.photo format-{avif,jpeg} as tmp_photo %}

{{ tmp_photo.formats.avif.0.url }}
{{ tmp_photo.formats.jpeg.0.url }}
```

Note

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

Renditions contain 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, contextual alt text or `default_alt_text` if not.

`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="og: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) is **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 `CustomImagesAppConfig`:

```
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 several 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 the format
- GIF images without animation are converted to PNGs
- AVIF images 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`, `format-gif` or `format-ico`.

Lossless AVIF and WebP

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

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

Favicon generation

You can save images as a `.ico` file using `format-ico`, which is especially useful when managing a site's favicon through the Admin.

```
{% image favicon_image format-ico as favicon_image_formatted %}
<link rel="icon" type="image/x-icon" href="{{ favicon_image_formatted.url }}"/>
```

Background color

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 color.

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

This filter takes a single argument, which is a CSS 3 or 6-digit hex code representing the color 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 image quality settings default to 85 (which is quite high). AVIF and WebP default to 80. This can be changed either globally or on a per-tag basis.

Changing globally

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

```
# settings.py

# Make low-quality but small images
WAGTAILIMAGES_AVIF_QUALITY = 50
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 AVIF, JPEG, and WebP qualities on individual tags by using `avifquality`, `jpegquality`, and `webpquality` filters. This will always override the default setting:

```
{% image page.photo_avif width=400 avifquality=40 %}
{% image page.photo_jpeg width=400 jpegquality=40 %}
{% image page.photo_webp width=400 webpquality=50 %}
```

Note that this will not affect PNG or GIF files. If you want all images to be low quality, you can use this filter with `format-avif`, `format-jpeg`, or `format-webp` (which forces all images to output in AVIF, JPEG, or WebP format):

```
{% image page.photo width=400 format-avif avifquality=40 %}
{% 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](#).

SVG images

Wagtail supports the use of Scalable Vector Graphics alongside raster images. To allow Wagtail users to upload and use SVG images, add “svg” to the list of allowed image extensions by configuring WAGTAILIMAGES_EXTENSIONS:

```
WAGTAILIMAGES_EXTENSIONS = ["gif", "jpg", "jpeg", "png", "webp", "svg"]
```

SVG images can be included in templates via the `image` template tag, as with raster images. However, operations that require SVG images to be rasterized are not currently supported. This includes direct format conversion, e.g. `format-webp`, and `bgcolor` directives. Crop and resize operations do not require rasterization, so may be used freely (see [Available resizing methods](#)).

The `image` tag’s `preserve-svg` positional argument may be used to restrict the operations applied to an SVG image to only those that do not require rasterization. This may be useful in situations where a single `image` tag declaration is applied to multiple source images, for example:

```
{% for picture in pictures %}
  {% image picture fill-400x400 format-webp preserve-svg %}
{% endfor %}
```

In this example, any of the `image` objects that are SVGs will only have the `fill-400x400` operation applied to them, while raster images will have both the `fill-400x400` and `format-webp` operations applied. If the `preserve-svg` argument is not used in this example, an error will be raised when attempting to convert SVG images to `webp`, as this is not possible without a rasterization library.

Security considerations

Wagtail’s underlying image library, Willow, is configured to mitigate known XML parser exploits (e.g. billion laughs, quadratic blowup) by rejecting suspicious files.

When including SVG images in templates via the `image` tag, they will be rendered as HTML `img` elements. In this case, `script` elements in SVGs will not be executed, mitigating XSS attacks.

If a user navigates directly to the URL of the SVG file embedded scripts may be executed, depending on server/storage configuration. This can be mitigated by setting appropriate Content-Security-Policy or Content-Disposition headers for SVG responses:

- setting Content-Security-Policy: `default-src 'none'` will prevent scripts from being loaded or executed (as well as other resources - a more relaxed policy of `script-src 'none'` may also be suitable); and
- setting Content-Disposition: `attachment` will cause the file to be downloaded rather than being immediately rendered in the browser, meaning scripts will not be executed (note: this will not prevent scripts from running if a user downloads and subsequently opens the SVG file in their browser).

The steps required to set headers for specific responses will vary, depending on how your Wagtail application is deployed. For the built-in [Dynamic image serve view](#), a Content-Security-Policy header is automatically set for you.

HEIC / HEIF images

HEIC / HEIF images are not widely supported on the web, but may be encountered when exporting images from Apple devices. Wagtail does not allow upload of these by default, but this can be enabled by adding "heic" to WAGTAIL_IMAGES_EXTENSIONS:

```
WAGTAILIMAGES_EXTENSIONS = ["gif", "jpg", "jpeg", "png", "webp", "heic"]
```

Note that to upload HEIC / HEIF images, the file extension must be .heic and not .heif or other extensions.

These images will be automatically converted to JPEG format when rendered (see [Customizing output formats](#)).

1.3.4 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 to 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 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 the 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.

Disabling model indexing

Indexing of a model can be disabled completely by setting `search_fields = []` within the model. This will disable index updates by the signal handler and by the `update_index` management command.

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

- **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 a 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'), (4.333333333333333, '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 that match partial words. For example, a page titled Hello World! will be found if the user only types He_l into the search box.

This takes the same options as `index.SearchField`.

Note

`index.AutoCompleteField` should only be used on fields that are displayed in the search results. This allows users to see any words that were partial-matched.

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 field 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'),
        ]),
    ]
```

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', [
```

(continues on next page)

(continued from previous page)

```
        index.SearchField('title'),
    ],
]
```

Filtering on `index.RelatedFields`

It's not possible to filter on any `index.FilterFields` within `index.RelatedFields` using the `QuerySet` API. Placing `index.FilterField` inside `index.RelatedFields` is valid, and will cause the appropriate field data to be stored at indexing time, but the `QuerySet` API does not currently support filters that span relations, and so there is no way to access these fields. However, 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 [InlinePanel](#), to index each BookPage by the titles of its related_links:

```
class BookPage(Page):
    # ...
    def get_related_link_titles(self):
        # Get list of titles and concatenate them
        return '\n'.join(self.related_links.all().values_list('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', boost=10),
        index.AutocompleteField('title', 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 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 Page-QuerySet. 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()`.

The standard behavior of the `search` method is to only return matches for complete words; for example, a search for “hel” will not return results containing the word “hello”. The exception to this is the fallback database search backend, used when the database does not have full-text search extensions available, and no alternative backend has been specified. This performs basic substring matching and will return results containing the search term ignoring all word boundaries.

Autocomplete searches

Wagtail provides a separate method which performs partial matching on specific autocomplete fields. This is primarily useful for suggesting pages to the user in real-time as they type their query - it is not recommended for ordinary searches, as the autocomplete will tend to add unwanted results beyond the specific term being searched for.

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

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 the 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
# 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 behavior

Search operator

The search operator specifies how the 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 requires 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.

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

Phrase searching

Phrase searching is used for finding whole sentences or phrases 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

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>, <Page: Hello>]
```

Fuzzy matching is supported by the Elasticsearch search backend only.

The `operator` keyword argument is also supported on `Fuzzy` objects, defaulting to “or”:

```
>>> Page.objects.search(Fuzzy("Hallo wurld", operator="and"))
[<Page: Hello World>]
```

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 the 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 to 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 a `QueryDict` 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 key:value1 key:value2', operator='and')

# Alternatively..
# filters, query = parse_query_string("my query string 'this is a phrase' this_is_
->a:filter key:test1 key:test2", operator='and')

>>> filters
<QueryDict: {'this_is_a': ['filter'], 'key': ['value1', 'value2']}>>

# Get a list of values associated with a particular key using the getlist method
>>> filters.getlist('key')
['value1', 'value2']

# Get a dict representation using the dict method
# NOTE: The dict method will reduce multiple values for a particular key to the last_
->assigned value
>>> filters.dict()
{
    'this_is_a': 'filter',
    'key': 'value2'
}

>>> 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)

```

(continues on next page)

(continued from previous page)

```
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)
...
("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.contrib.search_promotions.models import Query
```

(continues on next page)

(continued from previous page)

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

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

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

And here's a template to go with it:

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

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

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

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

Promoted search results

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

This functionality is provided by the `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`

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 roll back if an error occurs.

Setting the `ATOMIC_REBUILD` setting to `True` makes Wagtail rebuild into a separate index while keeping 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 7 and 8 are supported. Use the appropriate backend for your version:

- `wagtail.search.backends.elasticsearch7` (Elasticsearch 7.x)
- `wagtail.search.backends.elasticsearch8` (Elasticsearch 8.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>=7.0.0,<8.0.0" # for Elasticsearch 7.x
```

```
pip install "elasticsearch>=8.0.0,<9.0.0" # for Elasticsearch 8.x
```

The backend is configured in settings:

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.search.backends.elasticsearch8',
        'URLS': ['https://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 a case-sensitive keyword argument (for example `'max_retries': 1`).

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

```
WAGTAILSEARCH_BACKENDS = {
    'default': {
        ...
        'URLS': ['https://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 `ElasticsearchBackend` class in the module `wagtail/wagtail/search/backends/elasticsearch7.py`. Any new key is added and any existing key, if not a dictionary, is replaced with the new value. Here's a sample of how to configure the number of shards and set 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](#), which offers 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`

OpenSearch

OpenSearch is a community-driven search engine originally created as a fork of Elasticsearch 7. Wagtail supports OpenSearch through the `wagtail.search.backends.elasticsearch7` backend and version 7.13.4 of the [Elasticsearch Python library](#). Later versions of the library only permit connecting to Elastic-branded servers, and are not compatible with OpenSearch.

Amazon AWS OpenSearch

The Elasticsearch backend is compatible with [Amazon OpenSearch 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': {
```

(continues on next page)

(continued from previous page)

```
'BACKEND': 'wagtail.search.backends.elasticsearch7',
'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](#).

1.3.5 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 or function.

By default, 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.

Registering snippets

Snippets can be registered using `register_snippet` as a decorator or as a function. The latter is recommended, but the decorator is provided for convenience and backward compatibility.

Using `@register_snippet` as a decorator

Snippets can be registered using the `@register_snippet` decorator on the Django model. 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: `url` and `text`. The editing interface is very close to that provided for Page-derived models, with fields assigned in the `panels` (or `edit_handler`) property. Unless configured further, 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.

Using `register_snippet` as a function

While the `@register_snippet` decorator is convenient, the recommended way to register snippets is to use `register_snippet` as a function in your `wagtail_hooks.py` file. For example:

```
# myapp/wagtail_hooks.py
from wagtail.snippets.models import register_snippet

from myapp.models import Advert

register_snippet(Advert)
```

Registering snippets in this way allows you to add further customizations using a custom `SnippetViewSet` class later. This also provides a better separation between your Django model and Wagtail-specific concerns. For example, instead of defining the `panels` or `edit_handler` on the model class, they can be defined on the `SnippetViewSet` class:

```
# myapp/wagtail_hooks.py
from wagtail.snippets.models import register_snippet
from wagtail.snippets.views.snippets import SnippetViewSet

from myapp.models import Advert

class AdvertViewSet(SnippetViewSet):
    model = Advert

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

# Instead of using @register_snippet as a decorator on the model class,
# register the snippet using register_snippet as a function and pass in
# the custom SnippetViewSet subclass.
register_snippet(AdvertViewSet)
```

If you would like to do more customizations of the panels, you can override the `get_edit_handler()` method. Further customizations will be explained later in [Customizing admin views for snippets](#).

Rendering snippets

As Django models, snippets can be rendered in Django templates using a custom template tag. Alternatively, they can also be included as part of a Wagtail page's rendering process.

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 %
        ...
    {% 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(

```

(continues on next page)

(continued from previous page)

```
'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'),  
    ]  
  
    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.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>
```

(continues on next page)

(continued from previous page)

```

<a href="{{ advert_placement.advert.url }}>
    {{ advert_placement.advert.text }}
</a>
</p>
{ % endfor %}

```

Optional features

By default, snippets lack many of the features of pages, such as previews, revisions, and workflows. These features can individually be added to each snippet model by inheriting from the appropriate mixin classes.

Making snippets previewable

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
# ...


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 customize 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 customize 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
# ...


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
# ...


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'),
        index.AutocompleteField('text'),
    ]
```

Saving revisions of snippets

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 highly recommended to define a `GenericRelation` to the `Revision` model as the `revisions` attribute so that you can do related queries. Defining the `GenericRelation` is also necessary to ensure that `Revision` instances are automatically deleted when the snippet instance is deleted. If you need to customize how the revisions are fetched (for example, to handle the content type to use for models with multi-table inheritance), you can define the `revisions` as a property.

For example, the `Advert` snippet could be made revisable as follows:

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


class Advert(RevisionMixin, models.Model):
    url = models.URLField(null=True, blank=True)
    text = models.CharField(max_length=255)
    # If no custom logic is required, this can be defined as `revisions` directly
    _revisions = GenericRelation("wagtailcore.Revision", related_query_name="advert")

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

    @property
    def revisions(self):
        # Some custom logic here if necessary
        return self._revisions
```

If your snippet model defines relations using Django's `ManyToManyField`, you need to change the model class to inherit from `modelcluster.models.ClusterableModel` instead of `django.models.Model` and replace the `ManyToManyField` with `ParentalManyToManyField`. Inline models should continue to use `ParentalKey` instead of `ForeignKey`. This is necessary in order to allow the relations to be stored in the revisions. See the *Authors* section of the tutorial for more details. For example:

```
# ...
from django.db import models
from modelcluster.fields import ParentalKey, ParentalManyToManyField
from modelcluster.models import ClusterableModel
from wagtail.models import RevisionMixin
# ...


class ShirtColour(models.Model):
    name = models.CharField(max_length=255)

    panels = [FieldPanel("name")]


class ShirtCategory(ClusterableModel):
    name = models.CharField(max_length=255)
```

(continues on next page)

(continued from previous page)

```
panels = [FieldPanel("name")]

class Shirt(RevisionMixin, ClusterableModel):
    name = models.CharField(max_length=255)
    colour = models.ForeignKey("shirts.ShirtColour", on_delete=models.SET_NULL,_
→blank=True, null=True)
    categories = ParentalManyToManyField("shirts.ShirtCategory", blank=True)
    revisions = GenericRelation("wagtailcore.Revision", related_query_name="shirt")

    panels = [
        FieldPanel("name"),
        FieldPanel("colour"),
        FieldPanel("categories", widget=forms.CheckboxSelectMultiple),
        InlinePanel("images", label="Images"),
    ]

class ShirtImage(models.Model):
    shirt = ParentalKey("shirts.Shirt", related_name="images")
    image = models.ForeignKey("wagtailimages.Image", on_delete=models.CASCADE,_
→related_name="+")
    caption = models.CharField(max_length=255, blank=True)
    panels = [
        FieldPanel("image"),
        FieldPanel("caption"),
    ]
```

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

If a snippet model inherits from `DraftStateMixin`, Wagtail will automatically add a live/draft status column to the listing view, change the “Save” action menu to “Save draft”, and add a new “Publish” action menu in the editor. Any changes you save in the snippets admin will be saved as revisions and will not be reflected in the “live” snippet instance until you publish the changes.

As the `DraftStateMixin` works by saving draft changes as revisions, inheriting from this mixin also requires inheriting from `RevisionMixin`. See [Saving revisions of snippets](#) above for more details.

Wagtail will also allow you to set publishing schedules for instances of the model if there is a `PublishingPanel` in the model’s `panels` definition.

For example, the `Advert` snippet could save draft changes and publishing schedules by defining it as follows:

```
# ...
from django.contrib.contenttypes.fields import GenericRelation
from wagtail.admin.panels import PublishingPanel
from wagtail.models import DraftStateMixin, RevisionMixin
# ...


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'),
        PublishingPanel(),
    ]

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

The `DraftStateMixin` includes additional fields that need 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.

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.

If you use the scheduled publishing feature, make sure that you run the `publish_scheduled` management command periodically. For more details, see [Scheduled publishing](#).

Publishing a snippet instance requires `publish` permission on the snippet model. For models with `DraftStateMixin` applied, Wagtail automatically creates the corresponding `publish` permissions and displays them in the ‘Groups’ area of the Wagtail admin interface. For more details on how to configure the permission, see [Permissions](#).

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 (for example, by checking its `live` field). We are planning to improve this in the future.

Locking snippets

If a snippet model inherits from `LockableMixin`, Wagtail will automatically add the ability to lock instances of the model. When editing, Wagtail will show the locking information in the “Status” side panel, and a button to lock/unlock the instance if the user has the permission to do so.

If the model is also configured to have scheduled publishing (as shown in [Saving draft changes of snippets](#) above), Wagtail will lock any instances that are scheduled for publishing.

Similar to pages, users who locked a snippet can still edit it, unless `WAGTAILADMIN_GLOBAL_EDIT_LOCK` is set to `True`.

For example, instances of the `Advert` snippet could be locked by defining it as follows:

```
# ...
from wagtail.models import LockableMixin
# ...


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

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

If you use the other mixins, make sure to apply `LockableMixin` after the other mixins, but before the `RevisionMixin` (in left-to-right order). For example, with `DraftStateMixin` and `RevisionMixin`, the correct inheritance of the model would be `class MyModel(DraftStateMixin, LockableMixin, RevisionMixin)`. There is a system check to enforce the ordering of the mixins.

The `LockableMixin` includes additional fields that need 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.

Locking and unlocking a snippet instance requires `lock` and `unlock` permissions on the snippet model, respectively. For models with `LockableMixin` applied, Wagtail automatically creates the corresponding `lock` and `unlock` permissions and displays them in the ‘Groups’ area of the Wagtail admin interface. For more details on how to configure the permission, see [Permissions](#).

Enabling workflows for snippets

If a snippet model inherits from `WorkflowMixin`, Wagtail will automatically add the ability to assign a workflow to the model. With a workflow assigned to the snippet model, a “Submit for moderation” and other workflow action menu items will be shown in the editor. The status side panel will also show the information on the current workflow.

Since the `WorkflowMixin` utilizes revisions and publishing mechanisms in Wagtail, inheriting from this mixin also requires inheriting from `RevisionMixin` and `DraftStateMixin`. It is also recommended to enable locking by inheriting from `LockableMixin`, so that the snippet instance can be locked and only editable by reviewers when it is in a workflow. See the above sections for more details.

In addition to inheriting the mixins, it is highly recommended to define a `GenericRelation` to the `WorkflowState` model so that you can do related queries and that the workflow-related data is properly cleaned up when the snippet instance is deleted.

For example, workflows (with locking) can be enabled for the `Advert` snippet by defining it as follows:

```
# ...
from wagtail.models import DraftStateMixin, LockableMixin, RevisionMixin,
    WorkflowMixin
# ...


class Advert(WorkflowMixin, DraftStateMixin, LockableMixin, RevisionMixin, models.
    Model):
    url = models.URLField(null=True, blank=True)
    text = models.CharField(max_length=255)
```

(continues on next page)

(continued from previous page)

```

_revisions = GenericRelation("wagtailcore.Revision", related_query_name="advert")
workflow_states = GenericRelation(
    "wagtailcore.WorkflowState",
    content_type_field="base_content_type",
    object_id_field="object_id",
    related_query_name="advert",
    for_concrete_model=False,
)

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

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

```

The other mixins required by `WorkflowMixin` includes additional fields that need 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.

After enabling the mixin, you can assign a workflow to the snippet models through the workflow settings. For more information, see how to [configure workflows for moderation](#).

The admin dashboard and workflow reports will also show you snippets (alongside pages) that have been submitted to workflows.

Tagging snippets

Adding tags to snippets is very similar to adding tags to pages. The only difference is that if `RevisionMixin` is not applied, then `taggit.manager.TaggableManager` should be used in the place of `modelcluster.contrib.taggit.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')


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.

Inline models within snippets

Similar to pages, you can nest other models within a snippet. This requires the snippet model to inherit from `modelcluster.models.ClusterableModel` instead of `django.models.Model`.

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

class BandMember(Orderable):
    band = ParentalKey("music.Band", related_name="members", on_delete=models.CASCADE)
    name = models.CharField(max_length=255)

@register_snippet
class Band(ClusterableModel):
    name = models.CharField(max_length=255)
    panels = [
        FieldPanel("name"),
        InlinePanel("members")
    ]
```

The [documentation on how to use inline models with pages](#) provides more information that is also applicable to snippets.

Customizing admin views for snippets

Additional customizations to the admin views for each snippet model can be achieved through a custom `SnippetViewSet` class. The `SnippetViewSet` is a subclass of `ModelViewSet`, with snippets-specific properties provided by default. Hence, it supports the same customizations provided by `ModelViewSet` such as customizing the listing view (e.g. adding custom columns, and filters), creating a custom menu item, and more.

Before proceeding, ensure that you register the snippet model using `register_snippet` as a function instead of a decorator, as described in [Registering snippets](#).

For demonstration, consider the following `Member` model and a `MemberFilterSet` class:

```
# models.py
from django.db import models
from wagtail.admin.filters import WagtailFilterSet

class Member(models.Model):
    class ShirtSize(models.TextChoices):
        SMALL = "S", "Small"
        MEDIUM = "M", "Medium"
        LARGE = "L", "Large"
        EXTRA_LARGE = "XL", "Extra Large"

    name = models.CharField(max_length=255)
    shirt_size = models.CharField(max_length=5, choices=ShirtSize.choices, ↴
                                default=ShirtSize.MEDIUM)
```

(continues on next page)

(continued from previous page)

```

def get_shirt_size_display(self):
    return self.ShirtSize(self.shirt_size).label

get_shirt_size_display.admin_order_field = "shirt_size"
get_shirt_size_display.short_description = "Size description"

class MemberFilterSet(WagtailFilterSet):
    class Meta:
        model = Member
        fields = ["shirt_size"]

```

And the following is the snippet's corresponding SnippetViewSet subclass:

```

# wagtail_hooks.py
from wagtail.admin.panels import FieldPanel, ObjectList, TabbedInterface
from wagtail.admin.ui.tables import UpdatedAtColumn
from wagtail.snippets.models import register_snippet
from wagtail.snippets.views.snippets import SnippetViewSet

from myapp.models import Member, MemberFilterSet

class MemberViewSet(SnippetViewSet):
    model = Member
    icon = "user"
    list_display = ["name", "shirt_size", "get_shirt_size_display", UpdatedAtColumn()]
    list_per_page = 50
    copy_view_enabled = False
    inspect_view_enabled = True
    admin_url_namespace = "member_views"
    base_url_path = "internal/member"
    filterset_class = MemberFilterSet
    # alternatively, you can use the following instead of filterset_class
    # list_filter = ["shirt_size"]
    # or
    # list_filter = {"shirt_size": ["exact"], "name": ["icontains"]}

    edit_handler = TabbedInterface([
        ObjectList([FieldPanel("name")], heading="Details"),
        ObjectList([FieldPanel("shirt_size")], heading="Preferences"),
    ])

register_snippet(MemberViewSet)

```

Icon

You can define an `icon` attribute on the `SnippetViewSet` to specify the icon that is used across the admin for this snippet type. The `icon` needs to be [registered in the Wagtail icon library](#). If `icon` is not set, the default "snippet" icon is used.

URL namespace and base URL path

The `url_namespace` property can be overridden to use a custom URL namespace for the URL patterns of the views. If unset, it defaults to `wagtailsnippets_{app_label}_{model_name}`. Meanwhile, overriding `url_prefix` allows you to customize the base URL path relative to the Wagtail admin URL. If unset, it defaults to `snippets/app_label/model_name`.

Similar URL customizations are also possible for the snippet chooser views through `chooser_admin_url_namespace`, `chooser_base_url_path`, `get_chooser_admin_url_namespace()`, and `get_chooser_admin_base_path()`.

Listing view

You can customize the listing view to add custom columns, filters, pagination, etc. via various attributes available on the `SnippetViewSet`. Refer to [the listing view customizations for `ModelViewSet`](#) for more details.

Additionally, you can customize the base queryset for the listing view by overriding the `get_queryset()` method.

Copy view

The copy view is enabled by default and will be accessible by users with the 'add' permission on the model. To disable it, set `copy_view_enabled` to `False`. Refer to [the copy view customizations for `ModelViewSet`](#) for more details.

Inspect view

The inspect view is disabled by default, as it's not often useful for most models. To enable it, set `inspect_view_enabled` to `True`. Refer to [the inspect view customizations for `ModelViewSet`](#) for more details.

Templates

Template customizations work the same way as for `ModelViewSet`, except that the `template_prefix` defaults to `wagtailsnippets/snippets/`. Refer to [the template customizations for `ModelViewSet`](#) for more details.

Menu item

By default, registering a snippet model will add a “Snippets” menu item to the sidebar menu. However, you can configure a snippet model to have its own top-level menu item in the sidebar menu by setting `add_to_admin_menu` to True. Refer to [the menu customizations for ModelViewSet](#) for more details.

An example of a custom SnippetViewSet subclass with `add_to_admin_menu` set to True:

```
from wagtail.snippets.views.snippets import SnippetViewSet

class AdvertViewSet(SnippetViewSet):
    model = Advert
    icon = "crosshairs"
    menu_label = "Advertisements"
    menu_name = "adverts"
    menu_order = 300
    add_to_admin_menu = True
```

Multiple snippet models can also be grouped under a single menu item using a `SnippetViewSetGroup`. You can do this by setting the `model` attribute on the SnippetViewSet classes and then registering the SnippetViewSetGroup subclass instead of each individual model or viewset:

```
from wagtail.snippets.views.snippets import SnippetViewSet, SnippetViewSetGroup

class AdvertViewSet(SnippetViewSet):
    model = Advert
    icon = "crosshairs"
    menu_label = "Advertisements"
    menu_name = "adverts"

class ProductViewSet(SnippetViewSet):
    model = Product
    icon = "desktop"
    menu_label = "Products"
    menu_name = "banners"

class MarketingViewSetGroup(SnippetViewSetGroup):
    items = (AdvertViewSet, ProductViewSet)
    menu_icon = "folder-inverse"
    menu_label = "Marketing"
    menu_name = "marketing"

# When using a SnippetViewSetGroup class to group several SnippetViewSet classes_
# together,
# only register the SnippetViewSetGroup class. You do not need to register each_
# snippet
# model or viewset separately.
register_snippet(MarketingViewSetGroup)
```

If all snippet models have their own menu items, the “Snippets” menu item will not be shown.

Various additional attributes are available to customize the viewset - see [SnippetViewSet](#).

1.3.6 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 specialized 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 ImageBlock

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', ImageBlock()),
    ])

    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](#).

Note

While block definitions look similar to model fields, they are not the same thing. Blocks are only valid within a StreamField - using them in place of a model field will not work.

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', ImageBlock(required=False)),
        ('biography', blocks.RichTextBlock()),
    ])),
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageBlock()),
])
```

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 = ImageBlock(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', ImageBlock()),
])
```

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', ImageBlock(required=False)),
        ('biography', blocks.RichTextBlock()),
    ], icon='user')),
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageBlock()),
])
```

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

    class Meta:
        icon = 'user'
```

For a list of icons available out of the box, see our [icons overview](#). Project-specific icons are also displayed in the [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(ImageBlock())),
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageBlock()),
])
```

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>
            {%
              endif %
            }
          </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', ImageBlock()),
        ('video', EmbedBlock()),
    ])),
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageBlock()),
])
```

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 = ImageBlock()
    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 = ImageBlock()

class BlogPage(Page):
    body = StreamField(CommonContentBlock())
```

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>
```

Limiting block counts

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 a maximum number of blocks:

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

Or equivalently:

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

    class Meta:
        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 `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', ImageBlock()),
], block_counts={
    'heading': {'min_num': 1, 'max_num': 3},
})
```

Or equivalently:

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

    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', ImageBlock(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 = ImageBlock(required=False)
    biography = blocks.RichTextBlock()

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

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

```
{% load wagtailimages_tags %}

<div class="person">
```

(continues on next page)

(continued from previous page)

```
{% 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 }}
        </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.

Similarly, a `get_template` method can be defined to dynamically select a template based on the block value:

```
import datetime

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

    def get_template(self, value, context=None):
        if value["date"] > datetime.date.today():
            return "myapp/blocks/future_event.html"
        else:
            return "myapp/blocks/event.html"
```

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](#).

Configuring block previews

Added in version 6.4: The ability to have previews for StreamField blocks was added.

StreamField blocks can have previews that will be shown inside the block picker when you add a block in the editor. To enable this feature, you must configure the preview value and template. You can also add a description to help users pick the right block for their content.

You can do so by *passing the keyword arguments* `preview_value`, `preview_template`, and `description` when instantiating a block:

```
("quote", blocks.StructBlock(
    [
        ("text", blocks.TextBlock()),
        ("source", blocks.CharBlock()),
    ],
    preview_value={"text": "This is the coolest CMS ever.", "source": "Willie Wagtail"
    },
    preview_template="myapp/previews/blocks/quote.html",
    description="A quote with attribution to the source, rendered as a blockquote."
))
```

You can also set `preview_value`, `preview_template`, and `description` as attributes in the `Meta` class of the block. For example:

```
class QuoteBlock(blocks.StructBlock):
    text = blocks.TextBlock()
    source = blocks.CharBlock()

    class Meta:
        preview_value = {"text": "This is the coolest CMS ever.", "source": "Willie Wagtail"}
        preview_template = "myapp/previews/blocks/quote.html"
        description = "A quote with attribution to the source, rendered as a blockquote."
```

For more details on the preview options, see the corresponding `get_preview_value()`, `get_preview_template()`, and `get_description()` methods, as well as the `get_preview_context()` method.

In particular, the `get_preview_value()` method can be overridden to provide a dynamic preview value, such as from the database:

```
from myapp.models import Quote

class QuoteBlock(blocks.StructBlock):
    ...

    def get_preview_value(self, value):
        quote = Quote.objects.first()
        return {"text": quote.text, "source": quote.source}
```

Overriding the global preview template

In many cases, you likely want to use the block's real template that you already configure via `template` or `get_template` as described in [Per-block templates](#). However, such templates are only an HTML fragment for the block, whereas the preview requires a complete HTML document as the template.

To avoid having to specify `preview_template` for each block, Wagtail provides a default preview template for all blocks. This template makes use of the `{% include_block %}` tag (as described in [Template rendering](#)), which will reuse your block's specific template.

Note that the default preview template does not include any static assets that may be necessary to render your blocks properly. If you only need to add static assets to the default preview template, you can skip specifying `preview_template` for each block and instead override the default template globally. You can do so by creating a `wagtailcore/shared/block_preview.html` template inside one of your `templates` directories (with a higher precedence than the `wagtail` app) with the following content:

```
{% extends "wagtailcore/shared/block_preview.html" %}
{% load static %}

{% block css %}
    {{ block.super }}
    <link rel="stylesheet" href="{% static 'css/my-styles.css' %}">
{% endblock %}

{% block js %}
    {{ block.super }}
    <script src="{% static 'js/my-script.js' %}"></script>
{% endblock %}
```

For more details on overriding templates, see Django's guide on [How to override templates](#).

The global `wagtailcore/shared/block_preview.html` override will be used for all blocks by default. If you want to use a different template for a particular block, you can still specify `preview_template`, which will take precedence.

Customizations

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 customizations 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
```

(continues on next page)

(continued from previous page)

```
from wagtail.rich_text import RichText
my_page.body.append('paragraph', RichText("<p>And they all lived happily ever after.
→</p>"))

# Save the updated data back to the database
my_page.save()
```

If a block extending a StructBlock is to be used inside of the StreamField's value, the value of this block can be provided as a Python dict (similar to what is accepted by the block's `.to_python` method).

```
from wagtail import blocks

class UrlWithTextBlock(blocks.StructBlock):
    url = blocks.URLBlock()
    text = blocks.TextBlock()

    # using this block inside the content

data = {
    'url': 'https://github.com/wagtail/',
    'text': 'A very interesting and useful repo'
}

# append the new block to the stream as a tuple with the defined index for this block
→type
my_page.body.append('url', data)
my_page.save()
```

Retrieving blocks by name

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>
```

Search considerations

Like any other field, content in a StreamField can be made searchable by adding the field to the model's search_fields definition - see [Indexing extra fields](#). By default, all text content from the stream will be added to the search index. If you wish to exclude certain block types from being indexed, pass the keyword argument `search_index=False` as part of the block's definition. For example:

```
body = StreamField([
    ('normal_text', blocks.RichTextBlock()),
    ('pull_quote', blocks.RichTextBlock(search_index=False)),
    ('footnotes', blocks.ListBlock(blocks.CharBlock(), search_index=False)),
])
```

Custom validation

Custom validation logic can be added to blocks by overriding the block's `clean` method. For more information, see [StreamField validation](#).

Migrations

Since StreamField data is stored as a single JSON field, rather than being arranged in a formal database structure, it will often be necessary to write data migrations when changing the data structure of a StreamField or converting to or from other field types. For more information on how StreamField interacts with Django's migration system, and a guide to migrating rich text to StreamFields, see [StreamField migrations](#).

1.3.7 Permissions

Wagtail adapts and extends [the Django permission system](#) to cater to 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'.

Note

Whilst Wagtail supports a number of user roles and permissions, the Wagtail Admin should still be restricted to trusted users.

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 the 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 add 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. The 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 this page (and any pages underneath it) for editing, preventing other users from making any further edits to it.
- **Unlock** - grants the ability to unlock this page (and any pages underneath it), even if the page was locked by another user. Without this permission, only the user who locked the page (and superusers) can unlock the page.

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 in 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 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.

Adding custom permissions

See Django's documentation on [custom permissions](#) for details on how to set permissions up.

Permissions for models registered with Wagtail will automatically show up in the Wagtail admin Group edit form. For other models, you can also add the permissions using the `register_permissions` hook (see [register_permissions](#)).

To add a custom permission to be used in the Wagtail admin without relating to a specific model, you can create it using the content type of the `wagtail.admin.models.Admin` model. For example:

```
from django.contrib.auth.models import Permission
from django.contrib.contenttypes.models import ContentType
from wagtail.admin.models import Admin

content_type = ContentType.objects.get_for_model(Admin)
permission = Permission.objects.create(
    content_type=content_type,
    codename="can_do_something",
    name="Can do something",
)
```

After registering the permission using the `register_permissions` hook, it will be displayed in the Wagtail admin Group edit form under the 'Other permissions' section, alongside the 'Can access Wagtail admin' permission.

FieldPanel and PanelGroup permissions

Permissions can be used to restrict access to fields within the editor interface. See permission on [FieldPanel](#).

Permissions can be used to restrict groups of panels via the `permission` keyword argument on `PanelGroup` classes (`TabbedInterface`, `ObjectList`, `FieldRowPanel`, `MultiFieldPanel`). See how `PanelGroup` usage can be customized [Panels](#).

1.4 Advanced

1.4.1 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`. Hence, 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](#)

Generating multiple renditions for an image

You can generate multiple renditions of the same image from Python using the native `get_renditions()` method. It will accept any number of ‘specification’ strings or `Filter` instances, and will generate a set of matching renditions much more efficiently than generating each one individually. For example:

```
image.get_renditions('width-600', 'height-400', 'fill-300x186|jpegquality-60')
```

The return value is a dictionary of renditions keyed by the specifications that were provided to the method. The return value from the above example would look something like this:

```
{
    "width-600": <Rendition: Rendition object (7)>,
    "height-400": <Rendition: Rendition object (8)>,
    "fill-300x186|jpegquality-60": <Rendition: Rendition object (9)>,
}
```

Caching image renditions

Wagtail will cache image rendition lookups, which can improve the performance of pages which include many images.

By default, Wagtail will try to use the cache called “renditions”. If no such cache exists, it will fall back to using the default cache.

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.

Regenerating existing renditions

You can also directly use the image management command from the console to regenerate the renditions:

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

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

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 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

The following `AbstractImage` model methods are involved in finding and generating renditions. If using a custom image model, you can customize the behavior of either of these methods by overriding them on your model:

`class wagtail.images.models.AbstractImage`

`get_rendition(filter: Filter | str) → 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: Filter) → 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: Filter) → 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.

`get_renditions(*filters: Filter | str) → dict[str, AbstractRendition]`

Returns a dict of Rendition instances with image files reflecting the supplied `filters`, keyed by filter spec patterns.

Note: If using custom image models, instances of the custom rendition model will be returned.

`find_existing_renditions(*filters: Filter) → dict[Filter, AbstractRendition]`

Returns a dictionary of existing Rendition instances with `file` values (images) reflecting the supplied `filters` and the focal point values from this object.

Filters for which an existing rendition cannot be found are omitted from the return value. If none of the requested renditions have been created before, the return value will be an empty dict.

`create_renditions(*filters: Filter) → dict[Filter, AbstractRendition]`

Creates multiple Rendition instances with image files reflecting the supplied `filters`, and returns them as a dict keyed by the relevant `Filter` instance. Where suitable renditions already exist in the database, they will be returned instead, so as not to create duplicates.

This method is usually called by `Image.get_renditions()`, 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: Filter, *, source: django.core.files.File = None) → django.core.files.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()`.

If the contents of `self.file` has already been read into memory, the `source` keyword can be used to provide a reference to the in-memory `File`, bypassing the need to reload the image contents from storage.

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-avif as image_avif %}
    <source srcset="{{ image_avif.url }}" type="image/avif">

    {% 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>
```

Customizing output formats

By default, all `avif`, `bmp` and `webp` images are converted to the `png` format when no image output format is given, and `heic` images are converted to `jpeg`.

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 = {
    'avif': 'avif',
    'bmp': 'jpeg',
    'webp': 'webp',
}
```

will convert `bmp` images to `jpeg` and disable the default `avif` and `webp` to `png` conversion.

Custom image models

The `Image` model can be customized, 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',
    )

@property
def default_alt_text(self):
    # Force editors to add specific alt text if description is empty.
    # Do not use image title which is typically derived from file name.
    return getattr(self, "description", None)

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 `wagtail.images.models.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.

Overriding the upload location

The following methods can be overridden on your custom `Image` or `Rendition` models to customize how the original and rendition image files get stored.

```
class wagtail.images.models.AbstractImage
    def get_upload_to(self, filename)
        Generates a file path in the "original_images" folder. Ensuring ASCII characters and limiting length to prevent
        filesystem issues during uploads.
```

```
class wagtail.images.models.AbstractRendition
    def get_upload_to(self, filename)
        Generates a file path within the "images" folder by combining the folder name and the validated filename.
```

Refer to the Django `FileField.upload_to` function to further understand how the function works.

Changing rich text representation

The HTML representation of an image in rich text can be customized - 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', 'image-classes_'
                          'object-contain', 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):
    
```

(continues on next page)

(continued from previous page)

```
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)  
  
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 `libsmbus`, `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), however, 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 behavior 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 first 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  
  
urlpatterns = [  
    ...  
  
    re_path(r'^images/([^/]*)/(\d*)/([^/]*)/[^/]*$', SendFileView.as_view(), name=  
        ↪'wagtailimages_serve'),  
]
```

You can customize 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 customize 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_tags %}

{% image page.image 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 customize 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 customize this [same behavior for documents](#).

You can customize 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 behavior 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 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

For each example below, create the specified external JavaScript file in your app's static directory, such as static/js/, and reference it in the wagtail_hooks.py file.

Removing any url unsafe characters from the title

```
# wagtail_hooks.py
from django.templatetags.static import static
from django.utils.html import format_html

from wagtail import hooks

@hooks.register("insert_global_admin_js")
def get_global_admin_js():
    script_url = static('js/wagtail_admin.js')
    return format_html('<script src="{}"></script>', script_url)
```

```
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:images-upload', function (event) {
        const newTitle = (event.detail.data.title || '').replace(
            /[^\u00a1-\u00d7\u00d9\u00d8]/g,
            '',
        );
        event.detail.data.title = newTitle;
    });
});
```

Changing generated titles on the page editor only to remove dashes/underscores

Use 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.templatetags.static import static
from django.utils.html import format_html

from wagtail import hooks

@hooks.register("insert_editor_js")
def get_global_admin_js():
    def insert_remove_dashes_underscores_js():
        script_url = static('js/remove_dashes_underscores.js')
        return format_html('<script src="{}"></script>', script_url)
```

```
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:images-upload', function (event) {
        // Replace dashes/underscores with a space
        const newTitle = (event.detail.data.title || '').replace(
            /(\s|_|-)/g,
            ' ',
        );
        event.detail.data.title = newTitle;
    });
});
```

Stopping pre-filling of title based on filename

```
# wagtail_hooks.py
from django.templatetags.static import static
from django.utils.html import format_html

from wagtail import hooks

@hooks.register("insert_global_admin_js")
def get_global_admin_js():
    script_url = static('js/stop_prefill.js')
    return format_html('<script src="{}"></script>', script_url)
```

```
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:images-upload', function (event) {
        // Stop title pre-fill on single file uploads
        // Set the multiple upload title to the filename (with extension)
        event.preventDefault();
    });
});
```

1.4.2 Documents

Documents overview

This page provides an overview of the basics of using the `'wagtail.documents'` app in your Wagtail project.

Including `'wagtail.documents'` in `INSTALLED_APPS`

To use the `wagtail.documents` app, you need to include it in the `INSTALLED_APPS` list in your Django project's settings. Simply add it to the list like this:

```
# settings.py

INSTALLED_APPS = [
    # ...
    'wagtail.documents',
    # ...
]
```

Setting up URLs

Next, you need to set up URLs for the `wagtail.documents` app. You can include these URLs in your project's main `urls.py` file. To do this, add the following lines:

```
# urls.py

from wagtail.documents import urls as wagtaildocs_urls

urlpatterns = [
    # ...
    path('documents/', include(wagtaildocs_urls)),
    # ...
]
```

New documents saved are stored in the `reference index` by default.

Using documents in a Page

To include a document file in a Wagtail page, you can use `FieldPanel` in your page model.

Here's an example:

```
# models.py

from wagtail.admin.panels import FieldPanel
from wagtail.documents import get_document_model


class YourPage(Page):
    # ...
    document = models.ForeignKey(
        get_document_model(),
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
    )

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

This allows you to select a document file when creating or editing a page, and link to it in your page template.

Here's an example template to access the document field and render it:

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

{% block content %}
    {% if page.document %}
        <h2>Document: {{ page.document.title }}</h2>
        <p>File Type: {{ page.document.file_extension }}</p>
        <a href="{{ page.document.url }}" target="_blank">View Document</a>
    {% else %}
        <p>No document attached to this page.</p>
    {% endif %}
    <div>{{ page.body }}</div>
{% endblock %}
```

Using documents within RichTextFields

Links to documents can be made in pages using the `RichTextField`. By default, Wagtail will include the features for adding links to documents see [Limiting features in a rich text field](#).

You can either exclude or include these by passing the `features` to your `RichTextField`. In the example below we create a `RichTextField` with only documents and basic formatting.

```
# models.py
from wagtail.fields import RichTextField


class BlogPage(Page):
    # ...other fields
```

(continues on next page)

(continued from previous page)

```
document_footnotes = RichTextField(
    blank=True,
    features=["bold", "italic", "ol", "document-link"]
)

panels = [
    # ...other panels
    FieldPanel("document_footnotes"),
]
```

Using documents within StreamField

StreamField provides a content editing model suitable for pages that do not follow a fixed structure. To add links to documents using StreamField, include it in your models and also include the DocumentChooserBlock.

Create a Page model with a StreamField named doc and a DocumentChooserBlock named doc_link inside the field:

```
# models.py

from wagtail.fields import StreamField
from wagtail.documents.blocks import DocumentChooserBlock


class BlogPage(Page):
    # ... other fields

    documents = StreamField([
        ('document', DocumentChooserBlock())
    ],
    null=True,
    blank=True,
    use_json_field=True,
)

panels = [
    # ... other panels
    FieldPanel("documents"),
]
```

In blog_page.html, add the following block of code to display the document link in the page:

```
{% for block in page.documents %}
<a href="{{ block.value.url }}>{{ block.value.title }}</a>
{% endfor %}
```

Working documents and collections

Documents in Wagtail can be organized within **collections**. Collections provide a way to group related documents. You can cross-link documents between collections and make them accessible through different parts of your site.

Here's an example:

```
from wagtail.documents import get_document_model

class PageWithCollection(Page):
    collection = models.ForeignKey(
        "wagtailcore.Collection",
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name='+',
        verbose_name='Document Collection',
    )

    content_panels = Page.content_panels + [
        FieldPanel("collection"),
    ]

    def get_context(self, request):
        context = super().get_context(request)
        documents = get_document_model().objects.filter(collection=self.collection)
        context['documents'] = documents
        return context
```

Here's an example template to access the document collection and render it:

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

{% block content %}
    {% if documents %}
        <h3>Documents:</h3>
        <ul>
            {% for document in documents %}
                <li>
                    <a href="{{ document.url }}" target="_blank">{{ document.title }}</a>
                </li>
            {% endfor %}
        </ul>
    {% endif %}
{% endblock %}
```

Making documents private

If you want to restrict access to certain documents, you can place them in [private collections](#).

Private collections are not publicly accessible, and their contents are only available to users with the appropriate permissions.

API access

Documents in Wagtail can be accessed through the API via the `wagtail.documents.api.v2.views.DocumentAPIViewSet`. This allows you to programmatically interact with documents, retrieve their details, and perform various operations.

For more details, you can refer to the [API section](#) that provides additional information and usage examples.

Custom document model

An alternate Document model can be used to add custom behavior 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,
        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 built-in 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](#).

Templates that reference the built-in document model will 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 `wagtail.documents.models.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.

Custom document upload form

Wagtail provides a way to use a custom document form by modifying the `WAGTAILDOCS_DOCUMENT_FORM_BASE` setting. This setting allows you to extend the default document form with your custom fields and logic.

Here's an example:

```
# settings.py
WAGTAILDOCS_DOCUMENT_FORM_BASE = 'myapp.forms.CustomDocumentForm'
```

```
# myapp/forms.py
from django import forms

from wagtail.documents.forms import BaseDocumentForm

class CustomDocumentForm(BaseDocumentForm):
    terms_and_conditions = forms.BooleanField(
        label="I confirm that this document was not created by AI.",
        required=True,
    )

    def clean(self):
        cleaned_data = super().clean()
        if not cleaned_data.get("terms_and_conditions"):
            raise forms.ValidationError(
                "You must confirm the document was not created by AI."
            )
        return cleaned_data
```

Note

Any custom document form should extend the built-in `BaseDocumentForm` class.

Storing and serving

Wagtail follows Django's conventions for managing uploaded files. For configuration of `FileSystemStorage` and more information on handling user uploaded files, see [User Uploaded Files](#).

File storage location

Wagtail uses the `STORAGES["default"]` setting to determine where and how user-uploaded files are stored. By default, Wagtail stores files in the local filesystem.

Serving documents

Document serving is controlled by the `WAGTAILDOCS_SERVE_METHOD` method. It provides a number of serving methods which trade some of the strictness of the permission check that occurs when normally handling a document request for performance.

The serving methods provided are `direct`, `redirect` and `serve_view`, with `redirect` method being the default when `WAGTAILDOCS_SERVE_METHOD` is unspecified or set to `None`. For example:

```
WAGTAILDOCS_SERVE_METHOD = "redirect"
```

Content types

Wagtail provides the `WAGTAILDOCS_CONTENT_TYPES` setting to specify which document content types are allowed to be uploaded. For example:

```
WAGTAILDOCS_CONTENT_TYPES = {
    'pdf': 'application/pdf',
    'txt': 'text/plain',
}
```

Inline content types

Inline content types can be specified using `WAGTAILDOCS_INLINE_CONTENT_TYPES`, are displayed within the rich text editor.

For example:

```
WAGTAILDOCS_INLINE_CONTENT_TYPES = ['application/pdf', 'text/plain']
```

File extensions

Wagtail allows you to specify the permitted file extensions for document uploads using the `WAGTAILDOCS_EXTENSIONS` setting.

It also validates the extensions using Django's `FileExtensionValidator`. For example:

```
WAGTAILDOCS_EXTENSIONS = ['pdf', 'docx']
```

Document password required template

Wagtail provides the `WAGTAILDOCS_PASSWORD_REQUIRED_TEMPLATE` setting to use a custom template when a password is required to access a protected document. Read more about [Private pages](#).

Here's an example:

```
WAGTAILDOCS_PASSWORD_REQUIRED_TEMPLATE = 'myapp/document_password_required.html'
```

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 customize 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 customize this [same behavior for images](#).

You can customize 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 an event listener will work.

DOM event

The event name to listen to 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 behavior 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

For each example below, create the specified external JavaScript file in your app's static directory, such as `static/js/`, and reference it in the `wagtail_hooks.py` file.

Adding the file extension to the start of the title

```
# wagtail_hooks.py
from django.templatetags.static import static
from django.utils.html import format_html

from wagtail import hooks

@hooks.register("insert_global_admin_js")
def get_global_admin_js():
    script_url = static('js/title_with_extension.js')
    return format_html('<script src="{}"></script>', script_url)
```

```
// title_with_extension.js
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:documents-upload', function (event) {
        const extension = (event.detail.filename.match(
            /\.([^.]*?)\?=\?|#|\$/,
        ) || [''])[1];
        const newTitle = `(${extension.toUpperCase()}) ${event.detail.data.title} || ''`;
        event.detail.data.title = newTitle;
    });
});
```

Changing generated titles on the page editor only to remove dashes/underscores

Use the `insert_editor_js` hook instead so that this script will run only on page editors and not on the Document.

```
# wagtail_hooks.py
from django.templatetags.static import static
from django.utils.html import format_html

from wagtail import hooks

@hooks.register("insert_editor_js")
def get_editor_js():
    script_url = static('js/remove_dashes_underscores.js')
    return format_html('<script src="{}"></script>', script_url)
```

```
// remove_dashes_underscores.js
window.addEventListener('DOMContentLoaded', function () {
    document.addEventListener('wagtail:documents-upload', function (event) {
        // Replace dashes/underscores with a space
        const newTitle = (event.detail.data.title || '').replace(
            /(\s|-)/g,
            ' ',
        );
});
```

(continues on next page)

(continued from previous page)

```

        event.detail.data.title = newTitle;
    });
});

```

Stopping pre-filling of title based on filename

```

# wagtail_hooks.py
from django.templatetags.static import static
from django.utils.html import format_html

from wagtail import hooks

@hooks.register("insert_global_admin_js")
def insert_stop_prefill_js():
    script_url = static('js/stop_title_prefill.js')
    return format_html('<script src="{}"></script>', script_url)

```

Save the following code as static/js/stop_title_prefill.js

```

// stop_title_prefill.js
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();
    });
});

```

1.4.3 Icons

Wagtail comes with an SVG icon set. The icons are used throughout the admin interface.

Elements that use icons are:

- *Register Admin Menu Item*
- *Client-side React components*
- *Rich text editor toolbar buttons*
- *Snippets*
- *StreamField blocks*

This document describes how to choose, add and customize icons.

Add a custom icon

Draw or download an icon and save it in a template folder:

```
# app/templates/app_name/toucan.svg

<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 800 800" id="icon-toucan">
    <!--! CC0 license. https://creativecommons.org/publicdomain/zero/1.0/ -->
    <path d="M321 662v1a41 41 0 1 1-83-2V470c0-129 71-221 222-221 122 0 153-42 153-93 0-
    ↵34-18-60-53-72v-4c147 23 203 146 203 257 0 107-80 247-277 247v79a41 41 0 1 1-82-
    ↵1v46a41 41 0 0 1-83 0v-46z"/>
    <path d="M555 136a23 23 0 1 0-46 0 23 23 0 0 0 46 0Zm-69-57H175c-60 0-137 36-137-
    ↵14519-8 367 6 72 18V79z"/>
</svg>
```

The `svg` tag should:

- Set the `id="icon-<name>"` attribute, icons are referenced by this name. The name should only contain lowercase letters, numbers, and hyphens.
- Set the `xmlns="http://www.w3.org/2000/svg"` attribute.
- Set the `viewBox="..."` attribute, and no `width` and `height` attributes.
- If the icon should be mirrored in right-to-left (RTL) languages, set the `class="icon--directional"` attribute.
- Include license / source information in a `<!--! -->` HTML comment, if applicable.

Set `fill="currentColor"` or remove `fill` attributes so the icon color changes according to usage.

Add the icon with the `register_icons` hook.

```
@hooks.register("register_icons")
def register_icons/icons:
    return icons + ['app_name/toucan.svg']
```

The majority of Wagtail's default icons are drawn on a 16x16 viewBox, sourced from the FontAwesome v6 free icons set.

Icon template tag

Use an icon in a custom template:

```
{% load wagtailadmin_tags %}
{% icon name="toucan" classname="..." title="..." %}
```

Changing icons via hooks

```
@hooks.register("register_icons")
def register_icons/icons:
    icons.remove("wagtailadmin/icons/time.svg") # Remove the original icon
    icons.append("path/to/time.svg") # Add the new icon
    return icons
```

Changing icons via template override

When several applications provide different versions of the same template, the application listed first in `INSTALLED_APPS` has precedence.

Place your app before any Wagtail apps in `INSTALLED_APPS`.

Wagtail icons live in `wagtail/admin/templates/wagtailadmin/icons/`. Place your own SVG files in `<your_app>/templates/wagtailadmin/icons/`.

Available icons

Enable the [styleguide](#) to view the available icons and their names for any given project.

Here are all available icons out of the box:

1.4.4 Embedded content

Wagtail supports generating embed code from URLs to content on external providers such as YouTube or Reddit. 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 [Embedly](#) 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

Customizing 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
# - '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],
    }
]
```

Customizing an individual provider

Multiple finders can be chained together. This can be used for customizing 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'}
    },
]
```

(continues on next page)

(continued from previous page)

```
# 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 doesn't return an embed.

Facebook and Instagram

As of October 2020, Meta 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 Meta 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 Meta. 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,
        }
```

(continues on next page)

(continued from previous page)

```
        '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's 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.

In general, whenever you make changes to embed settings you are recommended to clear out Embed objects using `purge_embeds command`.

1.4.5 Tagging

Wagtail provides tagging capabilities through the combination of two Django modules.

1. `django-taggit` - Which provides a general-purpose tagging implementation.
2. `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.

Adding tags to a page model

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')

class BlogPage(Page):
    ...
    tags = ClusterTaggableManager(through=BlogPageTag, blank=True)
```

(continues on next page)

(continued from previous page)

```
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](#). 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 recognize 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 recognized as an existing tag will be created automatically. To disable this behavior, 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 as snippets

To manage all the tags used in a project, you can register the Tag model as a snippet to be managed via 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 of your project's apps:

```
from wagtail.admin.panels import FieldPanel
from wagtail.snippets.models import register_snippet
from wagtail.snippets.views.snippets import SnippetViewSet
from taggit.models import Tag

class TagsSnippetViewSet(SnippetViewSet):
    panels = [FieldPanel("name")] # only show the name field
    model = Tag
    icon = "tag" # change as required
    add_to_admin_menu = True
    menu_label = "Tags"
    menu_order = 200 # will put in 3rd place (000 being 1st, 100 2nd)
    list_display = ["name", "slug"]
    search_fields = ("name",)

register_snippet(TagsSnippetViewSet)
```

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 automatically populated when the name field is filled and you don't need to enter the same name in both fields.

1.4.6 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:

```
mypoint/
  mypoint/
    __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.

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 [InlinePanel](#) 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),

    path('admin/', include(wagtailadmin_urls)),
```

(continues on next page)

(continued from previous page)

```

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',
    'wagtail.admin',
    'wagtail',

    'taggit',
]

```

(continues on next page)

(continued from previous page)

```
'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': {  
        '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  
    }  
}
```

(continues on next page)

(continued from previous page)

```

}

# 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'

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 customize your logging configuration.
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'filters': {

```

(continues on next page)

(continued from previous page)

```
'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.elasticsearch8',
#        'INDEX': 'myapp'
#    }
#}

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

# Wagtail email notification format
# WAGTAILADMIN_NOTIFICATION_USE_HTML = True

# Allowed file extensions for documents in the document library.
# This can be omitted to allow all files, but note that this may present a security
# risk
# if untrusted users are allowed to upload files -
# see https://docs.wagtail.org/en/stable/advanced_topics/deploying.html#user-uploaded-
# files
WAGTAILDOCS_EXTENSIONS = ['csv', 'docx', 'key', 'odt', 'pdf', 'pptx', 'rtf', 'txt',
    'xlsx', 'zip']

# 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'))
    ]

```

1.4.7 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.

We have tried to minimize 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',
        }
    }
}
```

To use a different cache backend for [caching image renditions](#), configure the “renditions” backend:

```
CACHES = {
    'default': {...},
    'renditions': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
        'TIMEOUT': 600,
        'OPTIONS': {
            'MAX_ENTRIES': 1000
        }
    }
}
```

Image URLs

If all you need is the URL to an image (such as for use in meta tags or other tag attributes), it is likely more efficient to use the `image serve view` and `{% image_url %}` tag:

```
<meta property="og:image" content="{% image_url page.hero_image width-600 %}" />
```

Rather than finding or creating the rendition in the page request, the image serve view offloads this to a separate view, which only creates the rendition when the user requests the image (or returning an existing rendition if it already exists). This can drastically speed up page loads with many images. This may increase the number of requests handled by Wagtail if you’re using an external storage backend (for example Amazon S3).

Another side benefit is it prevents errors during conversion from causing page errors. If an image is too large for Willow to handle (the size of an image can be constrained with `WAGTAILIMAGES_MAX_IMAGE_PIXELS`), Willow may crash. As the resize is done outside the page load, the image will be missing, but the rest of the page content will remain.

The same can be achieved in Python using `generate_image_url`.

Prefetch image rendition

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.

Frontend caching proxy

Many websites use a frontend cache such as [Varnish](#), [Squid](#), [Cloudflare](#) or [CloudFront](#) to support high volumes of traffic with excellent response times. 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.

Wagtail supports being [integrated](#) with many CDNs, so it can inform them when a page changes, so the cache can be cleared immediately and users see the changes sooner.

If you have multiple frontends configured (eg Cloudflare for one site, CloudFront for another), it's recommended to set the [`HOSTNAMES`](#) key to the list of hostnames the backend can purge, to prevent unnecessary extra purge requests.

Page URLs

To fully resolve the URL of a page, Wagtail requires information from a few different sources.

The methods used to get the URL of a Page such as `Page.get_url` and `Page.get_full_url` optionally accept extra arguments for `request` and `current_site`. Passing these arguments enable much of underlying site-level URL information to be reused for the current request. In situations such as navigation menu generation, plus any links that appear in page content, providing `request` or `current_site` can result in a drastic reduction in the number of cache or database queries your site will generate for a given page load.

When using the `{% pageurl %}` or `{% fullpageurl %}` template tags, the request is automatically passed in, so no further optimization is needed.

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, MySQL and MariaDB. It may work on some third-party database backends as well, but this is not guaranteed.

We recommend PostgreSQL for production use, however, the choice of database ultimately depends on a combination of factors, including personal preference, team expertise, and specific project requirements. The most important aspect is to ensure that your selected database can meet the performance and scalability requirements of your project.

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 optimization is already handled for you for images in the admin site.

Template fragment caching

Django supports [template fragment caching](#), which allows caching portions of a template. Using Django's `{% cache %}` tag natively with Wagtail can be [dangerous](#) as it can result in preview content being shown to end users. Instead, Wagtail provides 2 extra template tags: `{% wagtailcache %}` and `{% wagtailpagecache %}` which both avoid these issues.

Page cache key

It's often necessary to cache a value based on an entire page, rather than a specific value. For this, `cache_key` can be used to get a unique value for the state of a page. Should something about the page change, so will its cache key. You can also use the value to create longer, more specific cache keys when using Django's caching framework directly. For example:

```
from django.core.cache import cache

result = page.expensive_operation()
cache.set("expensive_result_" + page.cache_key, result, 3600)

# Later...
cache.get("expensive_result_" + page.cache_key)
```

To modify the cache key, such as including a custom model field value, you can override `get_cache_key_components`:

```
def get_cache_key_components(self):
    components = super().get_cache_key_components()
    components.append(self.external_slug)
    return components
```

Manually updating a page might not result in a change to its cache key, unless the default component field values are modified directly. To be sure of a change in the cache key value, try saving the changes to a `Revision` instead, and then publishing it.

Django

Wagtail is built on Django. Many of the [performance tips](#) set out by Django are also applicable to Wagtail.

1.4.8 Internationalization

- *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 internationalization*
 - * *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 internationalized sites*
 - * *Language/region selector*
 - * *API filters for headless sites*
 - * *Translatable snippets*
 - *Translation workflow*
 - * *Wagtail Localize*
- *Alternative internationalization 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 internationalization of content managed by Wagtail. For information on how to translate static content in template files, JavaScript code, etc, refer to the [Django internationalization 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 internationalization 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 exists 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 internationalization utilities such as `i18n_patterns` and `LocaleMiddleware`. This means that Wagtail should work seamlessly with any other internationalized 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 internationalization

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 internationalization 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 internationalization, 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 internationalization](#)
- [Configuring available languages](#)
- [Enabling the locale management UI \(optional\)](#)
- [Adding a language prefix to URLs](#)

- *User language auto-detection*
- *Custom routing/language detection*

Enabling internationalization

To enable internationalization 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 localization 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 localization (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)"),
```

(continues on next page)

(continued from previous page)

```
('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 = [
```

(continues on next page)

(continued from previous page)

```
path('django-admin/', admin.site.urls),  
  
    path('admin/', include(wagtailadmin_urls)),  
    path('documents/', include(wagtaildocs_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 internationalized sites

Language/region selector

Perhaps the most important bit of internationalization-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#stickiness> 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 wagtailcore_tags %}

{%- if page %}
    {%- for translation in page.get_translations.live %}
        <a href="{% pageurl translation %}" rel="alternate" hreflang="{{ translation.
        locale.language_code }}">
            {{ translation.locale.language_name_local }}
    {%- endfor %}
{%- endif %}
```

(continues on next page)

(continued from previous page)

```
</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.

```
<a href="{% pageurl translation %}" rel="alternate" hreflang="{{ translation.locale.
  language_code }}>
  {{ translation.locale.language_name_local }}
</a>
```

This adds a link to the translation. We use `{{ translation.locale.language_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. `translation.locale` is an instance of the [Locale model](#).

Alternatively, a built-in tag from Django that gets info about the language of the translation. For more information, see `{% get_language_info %}` in the [Django docs](#).

```
{%
  load i18n %}

{%
  get_language_info for translation.locale.language_code as lang %}
```

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',
      ],
    },
  },
]
```

(continues on next page)

(continued from previous page)

```
    },
]
```

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 internationalized 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


@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 first 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
```

(continues on next page)

(continued from previous page)

```
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 internationalization 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 internationalization plugins

Before official multi-language support was added into Wagtail, site implementers 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_MIN_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.

1.4.9 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 subpages. Several different kinds of restrictions are available:

- **Accessible to any logged-in users:** The user must log in to view the page. All user accounts are granted access, regardless of permission level.
- **Accessible with a shared password:** The user must enter the given shared 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.

⚠ Warning

Shared passwords should not be used to protect sensitive content, as the password is shared between all users, and stored in plain text in the database. Where possible, it's recommended to require users log in to access private page content.

You can disable shared password for pages using `WAGTAIL_PRIVATE_PAGE_OPTIONS`.

```
WAGTAIL_PRIVATE_PAGE_OPTIONS = {"SHARED_PASSWORD": False}
```

Any existing shared password usage will remain active but will not be viewable by the user within the admin, these can be removed in the Django shell as follows.

```
from wagtail.models import Page

for page in Page.objects.private():
    page.get_view_restrictions().filter(restriction_type='password').delete()
```

Private collections (restricting documents)

Similarly, documents can be made private by placing them in a collection with appropriate privacy settings (see: [Image / document permissions](#)).

You can also disable shared password for collections (which will impact document links) using `WAGTAILDOCS_PRIVATE_COLLECTION_OPTIONS`.

```
WAGTAILDOCS_PRIVATE_COLLECTION_OPTIONS = {"SHARED_PASSWORD": False}
```

Any existing shared password usage will remain active but will not be viewable within the admin, these can be removed in the Django shell as follows.

```
from wagtail.models import Collection

for collection in Collection.objects.all():
    collection.get_view_restrictions().filter(restriction_type='password').delete()
```

Setting up a login page

Private pages and collections (restricting documents) work on Wagtail out of the box - the site implementer does not need to do anything to set them up.

However, the default “login” and “password required” forms are only bare-bones HTML pages, and site implementers may wish to replace them with a page customized to their site design.

The basic login page can be customized 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 `WAGTAIL_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):

```
WAGTAIL_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. Several 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 `WAGTAIL_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.
      {% if user.is_authenticated %}To proceed, please log in with an account_
      ↪that has access.{% endif %}
    </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 `WAGTAIL_DOCS_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'
```

1.4.10 Customizing Wagtail

Customizing the editing interface

Customizing the tabbed interface

As standard, Wagtail organizes panels for pages into two tabs: ‘Content’ and ‘Promote’. For snippets, Wagtail puts all panels into one page. Depending on the requirements of your site, you may wish to customize 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, TitleFieldPanel, ObjectList

class BlogPage(Page):
    # field definitions omitted

    content_panels = [
        TitleFieldPanel('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'), # The default settings
        # are now displayed in the sidebar but need to be in the `TabbedInterface`.
    ])
)
```

Permissions can be set using `permission` on the `ObjectList` to restrict entire groups of panels to specific users.

```
from wagtail.admin.panels import TabbedInterface, TitleFieldPanel, ObjectList

class FundingPage(Page):
    # field definitions omitted

    shared_panels = [
```

(continues on next page)

(continued from previous page)

```

        TitleFieldPanel('title', classname="title"),
        FieldPanel('date'),
        FieldPanel('body'),
    ]
private_panels = [
    FieldPanel('approval'),
]

edit_handler = TabbedInterface([
    ObjectList(shared_panels, heading='Details'),
    ObjectList(private_panels, heading='Admin only', permission="superuser"),
    ObjectList(Page.promote_panels, heading='Promote'),
    ObjectList(Page.settings_panels, heading='Settings'), # The default settings
    ↪are now displayed in the sidebar but need to be in the `TabbedInterface`.
])

```

For more details on how to work with `Panel` and `PanelGroup` classes, see [Panels](#).

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:

- h2, h3, h4 - 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 a 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:

- h1, h5, h6 - heading elements
- 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 customize the formatting options shown to the editor when inserting images in the `RichTextField` editor.

As an example, add a “thumbnail” format:

```
# image_formats.py
from wagtail.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.

classname

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 frontend CSS code. Specifying a `classname` 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 when viewing or editing pages that reference them.

Customizing 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 TitleFieldPanel, 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')
```

(continues on next page)

(continued from previous page)

```
    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()
    location = models.CharField(max_length=255)

    content_panels = [
        TitleFieldPanel('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.

Customizing the generated copy page form

```
class wagtail.admin.forms.CopyForm
```

When copying a page, Wagtail will generate a form to allow the user to modify the copied page. By default, this form subclasses `CopyForm`. A custom base form class can be configured by setting the `copy_form_class` attribute on any model. Custom forms must subclass `CopyForm`.

This can be used to specify alterations to the copied form on a per-model basis.

For example, auto-incrementing the slug field:

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

from wagtail.admin.forms.pages import CopyForm
from wagtail.admin.panels import FieldPanel
from wagtail.models import Page

class CustomCopyForm(CopyForm):
    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```

"""
Override the default copy form to auto-increment the slug.
"""

super().__init__(*args, **kwargs)
suffix = 2 # set initial_slug as incremented slug
parent_page = self.page.get_parent()
if self.page.slug:
    try:
        suffix = int(self.page.slug[-1])+1
        base_slug = self.page.slug[:-2]

    except ValueError:
        base_slug = self.page.slug

    new_slug = base_slug + f"-{suffix}"
    while not Page._slug_is_available(new_slug, parent_page):
        suffix += 1
        new_slug = f"{base_slug}-{suffix}"

    self.fields["new_slug"].initial = new_slug

class BlogPage(Page):
    copy_form_class = CustomCopyForm # Set the custom copy form for all EventPage
    ↪models

    introduction = models.TextField(blank=True)
    body = RichTextField()

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

```

Custom page listings

Normally, editors navigate through the Wagtail admin interface by following the structure of the page tree. However, this can make it slow to locate a specific page for editing, especially on large sites where pages are organised into a deep hierarchy.

Custom page listings are a way to present a flat list of all pages of a given type, accessed from a menu item in the Wagtail admin menu, with the ability for editors to search and filter this list to find the pages they are interested in. To define a custom page listing, create a subclass of `PageListingViewSet` and register it using the `register_admin_viewset` hook.

For example, if your site implemented the page type `BlogPage`, you could provide a “Blog pages” listing in the Wagtail admin by adding the following definitions to a `wagtail_hooks.py` file within the app:

```

# myapp/wagtail_hooks.py
from wagtail import hooks
from wagtail.admin.viewsets.pages import PageListingViewSet

from myapp.models import BlogPage

class BlogPageListingViewSet(PageListingViewSet):

```

(continues on next page)

(continued from previous page)

```
icon = "globe"
menu_label = "Blog Pages"
add_to_admin_menu = True
model = BlogPage

blog_page_listing_viewset = BlogPageListingViewSet("blog_pages")
@hooks.register("register_admin_viewset")
def register_blog_page_listing_viewset():
    return blog_page_listing_viewset
```

The columns of the listing can be customized by overriding the `columns` attribute on the viewset. This should be a list of `wagtail.admin.ui.tables.Column` instances:

```
from wagtail import hooks
from wagtail.admin.ui.tables import Column
from wagtail.admin.viewsets.pages import PageListingViewSet

from myapp.models import BlogPage

class BlogPageListingViewSet(PageListingViewSet):
    # ...
    columns = PageListingViewSet.columns + [
        Column("blog_category", label="Category", sort_key="blog_category"),
    ]
```

The filtering options for the listing can be customized by overriding the `filterset_class` attribute on the viewset:

```
from wagtail import hooks
from wagtail.admin.viewsets.pages import PageListingViewSet

from myapp.models import BlogPage

class BlogPageFilterSet(PageListingViewSet.filterset_class):
    class Meta:
        model = BlogPage
        fields = ["blog_category"]

class BlogPageListingViewSet(PageListingViewSet):
    # ...
    filterset_class = BlogPageFilterSet
```

Customizing 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 = (
    # ...
```

(continues on next page)

(continued from previous page)

```
'dashboard',
'wagtail',
'wagtail.admin',
# ...
)
```

Custom branding

The template blocks that are available to customize 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 on 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 overwrites 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 profile avatar

To render a user avatar other than the one sourced from the UserProfile model or from gravatar, you can use the `get_avatar_url` hook and resolve the avatar’s image url as you see fit.

For example, you might have an avatar on a Profile model in your own application that is keyed to the auth.User model in the familiar pattern. In that case, you could register your hook as the in following example, and the Wagtail admin avatar will be replaced with your own Profile avatar accordingly.

```
@hooks.register('get_avatar_url')
def get_profile_avatar(user, size):
    return user.profile.avatar
```

Additionally, you can use the default `size` parameter that is passed in to the hook if you need to attach it to a request or do any further processing on your image.

Custom user interface fonts

To customize 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 colors

⚠ Warning

The default Wagtail colors conform to the WCAG2.1 AA level color contrast requirements. When customizing the admin colors you should test the contrast using tools like [Axe](#).

To customize the colors 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. Color variables are reused across both the light and dark themes of the admin interface. To change the colors of a specific theme, use:

- `:root, .w-theme-light` for the light theme.
- `.w-theme-dark` for the dark theme.
- `@media (prefers-color-scheme: light) { .w-theme-system { [...] } }` for the light theme via system settings.
- `@media (prefers-color-scheme: dark) { .w-theme-system { [...] } }` for the dark theme via system settings.

There are two ways to customize Wagtail's color scheme:

- Set static color variables, which are then reused in both light and dark themes across a wide number of UI components.
- Set semantic colors, which are more numerous but allow customizing specific UI components.

For static colors, either set each color 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 customized at once. For example, setting `--w-color-secondary-hue: 180;` will customize all of the secondary shades at once.

Custom UI information density

To customize information density of the admin user interface, inject a CSS file using the hook `insert_global_admin_css`. Set the `--w-density-factor` CSS variable to increase or reduce the UI density. The default value is 1, the “snug” UI theming uses 0.5. Here are example overrides:

```
:root,
:host {
    /* Reduce the UI density by 20% for users of the default theme. */
    --w-density-factor: 0.8;
}
```

(continues on next page)

(continued from previous page)

```
:root,  
:host {  
    /* Increase the UI density by 20% for users of the default theme. */  
    --w-density-factor: 1.2;  
}  
  
.w-density-snug {  
    /* For snug theme users, set a UI density even lower than vanilla Wagtail. */  
    --w-density-factor: 0.25;  
}
```

UI components which have been designed to use the `--w-density-factor` will increase in size or spacing accordingly.

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 customize the branding in the admin page. These can be useful for customizing the dashboard on a multi-tenanted 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 customize 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 }}
{% endblock %}
```

Extending client-side JavaScript

Wagtail provides multiple ways to *extend client-side JavaScript*.

Custom user models

This page shows how to configure Wagtail to accommodate a custom user model.

Creating a custom user model

This example uses a custom user model that adds a text field and foreign key field.

The custom user model 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).

```
# myapp/models.py
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 `myapp` and the model is `User`.

```
AUTH_USER_MODEL = 'myapp.User'
```

Creating custom user forms

Now we need to configure Wagtail's user forms to allow the custom fields' values to be updated. Create your custom user 'create' and 'edit' forms in your app:

```
# myapp/forms.py
from django import forms
from django.utils.translation import gettext_lazy as _

from wagtail.users.forms import UserEditForm, UserCreationForm

from myapp.models import MembershipStatus

class CustomUserEditForm(UserEditForm):
    status = forms.ModelChoiceField(queryset=MembershipStatus.objects, required=True,
    ↴label=_('Status'))

    # Use ModelForm's automatic form fields generation for the model's `country` field,
    # but use an explicit custom form field for `status`.
    class Meta(UserEditForm.Meta):
        fields = UserEditForm.Meta.fields | {"country", "status"}

class CustomUserCreationForm(UserCreationForm):
    status = forms.ModelChoiceField(queryset=MembershipStatus.objects, required=True,
    ↴label=_('Status'))
```

(continues on next page)

(continued from previous page)

```
# Use ModelForm's automatic form fields generation for the model's `country` ↴
# but use an explicit custom form field for `status`.
class Meta(UserCreationForm.Meta):
    fields = UserCreationForm.Meta.fields | {"country", "status"}
```

Extending the create and edit templates

Extend the Wagtail user ‘create’ and ‘edit’ templates. These extended templates should be placed in a template directory `wagtailusers/users`. Using a custom template directory is possible and will be explained later.

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. There is a `fields` block that allows appending fields to the end or beginning of the existing fields or to allow all the fields to be redefined.

Creating a custom `UserViewSet`

To make use of the custom forms, create a `UserViewSet` subclass.

```
# myapp/viewsets.py
from wagtail.users.views.users import UserViewSet as WagtailUserViewSet

from .forms import CustomUserCreationForm, CustomUserEditForm

class UserViewSet(WagtailUserViewSet):
    def get_form_class(self, for_update=False):
        if for_update:
            return CustomUserEditForm
        return CustomUserCreationForm
```

Then, 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:

```
# myproject/apps.py
from wagtail.users.apps import WagtailUsersAppConfig

class CustomUsersAppConfig(WagtailUsersAppConfig):
    user_viewset = "myapp.viewsets.UserViewSet"
```

Replace `wagtail.users` in `settings.INSTALLED_APPS` with the path to `CustomUsersAppConfig`.

```
INSTALLED_APPS = [
    ...
    # Make sure you have two separate entries for the following:
    "myapp", # an app that contains the custom user model
    "myproject.apps.CustomUsersAppConfig", # a custom app config for the wagtail.
    ↪users app
    # "wagtail.users", # this should be removed in favour of the custom app config
    ...
]
```

Warning

You can also place the `WagtailUsersAppConfig` subclass inside the same `apps.py` file of your custom user model's app (instead of in a `myproject/apps.py` file), but you need to be careful. Make sure to use two separate config classes instead of turning your existing `AppConfig` subclass into a `WagtailUsersAppConfig` subclass, as that would cause Django to pick up your custom user model as being part of `wagtail.users`. You may also need to set `default` to `True` in your own app's `AppConfig`, unless you already use a dotted path to the app's `AppConfig` subclass in `INSTALLED_APPS`.

The `UserViewSet` class is a subclass of `ModelViewSet` and thus it supports most of *the customizations available for ModelViewSet*. For example, you can use a custom directory for the templates by setting `template_prefix`:

```
class UserViewSet(WagtailUserViewSet):
    template_prefix = "myapp/users/"
```

or customize the create and edit templates specifically:

```
class UserViewSet(WagtailUserViewSet):
    create_template_name = "myapp/users/create.html"
    edit_template_name = "myapp/users/edit.html"
```

The group forms and views can be customized in a similar way – see *Customizing group edit/create views*.

How to build custom StreamField blocks

Custom editing interfaces for StructBlock

To customize 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()
```

(continues on next page)

(continued from previous page)

```
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_global_admin_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 customizations 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:
        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 customized 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" classname="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 behavior 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 initialized.

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 recognized 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);
            }
        };
        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 on 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 behavior, 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')
        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 customized client-side behavior 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 customize 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.

1.4.11 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.

- [Create a blog in Django with Wagtail \(video\)](#) (20 January 2025)
- [Finding \(StreamField\) Blocks Across a Wagtail Site](#) (6 December 2024)
- [Blog Site with Django + Wagtail CMS - Beginner Friendly Guide \(video\)](#) (25 November 2024)
- [Setting up Wagtail on Docker with PostgreSQL](#) (24 November 2024)
- [Importing and Synchronizing Pages with Wagtail and Wagtail Localize](#) (20 November 2024)
- [How to Build No-Code Modal Components for Wagtail CMS Content Editors](#) (17 November 2024)
- [How to Set Up GDPR-Compliant Analytics in Wagtail CMS: Cookie Consent with Clarity and Google Analytics](#) (10 November 2024)
- [Adding Wagtail to a Django project to support content](#) (29 September 2024)
- [A simple block pattern for Wagtail CMS](#) (29 August 2024)
- [Django and Wagtail site building comparison tutorial video](#) (9 July 2024)
- [An introduction to Wagtail tutorial](#) (17 June 2024)
- [Unleashing the Power of Custom Wagtail Models](#) (9 June 2024)
- [Deploying Wagtail on Divio \(~June 2024\)](#)
- [How to Deploy Wagtail To AWS EC2 and Digital Ocean](#) (12 May 2024)
- [Upgrading Wagtail \(from 2.5 to 6.0\)](#) (18 April 2024)
- [Using Wagtail Form Templates in Software Development Projects](#) (9 April 2024)
- [Build an Intuitive Link StructBlock in Wagtail: Simplifying Link Management for Content Editors](#) (9 March 2024)
- [Improving Wagtail RichText Block Revision Differing](#) (6 March 2024)
- [Wagtail StreamField - Propagating the `required` Attribute on Nested Blocks](#) (20 February 2024)
- [An overview of a Wagtail website stack](#) (15 February 2024)
- [Efficient Cascading Choices in Wagtail Admin: A Smart Chooser Panel Solution](#) (27 January 2024)
- [How to add an edit link to `wagtail-autocomplete` items](#) (30 January 2024)
- [Programmatically Creating a Wagtail page with StreamField](#) (19 December 2023)
- [Adding reCAPTCHA V3 to Wagtail's Form Builder Pages](#) (19 December 2023)
- [Guide for managing a complex multi-tenancy setup with Wagtail](#) (1 November 2023)
- [Wagtail tutorial video series, getting started through to e-commerce integration with PayPal](#) (1 November 2023)
- [Integrating Next.js and Wagtail: Building a Headless, Content-Driven Website](#) (21 October 2023)
- [Wagtail tutorial video series, building a blog](#) (1 October 2023)
- [Deploy Wagtail CMS to PythonAnywhere using git push to a bare repo](#) (27 September 2023)
- [Building a custom Django command to check all admin pages are loading correctly in Wagtail](#) (1 September 2023)

- [Integrating ChatGPT with Wagtail for efficient content generation](#) (15 August 2023)
- [Creating Wagtail StreamField StructBlocks with a Customised Editor Interface](#) (9 July 2023)
- [Wagtail on Cloud Run](#) (26 June 2023)
- [How to create a custom Wagtail CMS page type?](#) (29 June 2023)
- [Create Stylish Wagtail Pages with Tailwind CSS](#) (15 June 2023)
- [Backup and Restore a Wagtail CMS website](#) (26 May 2023)
- [A guide for updating Wagtail CMS](#) (22 May 2023)
- [Creating Custom Choosers with Viewsets](#) (18 April 2023)
- [Build a Website Without Any Coding with Traleor \(& Wagtail\) in Just 10 Minutes \(video\)](#) (19 March 2023)
- [Getting Started with Wagtail: A Beginner's Installation Guide \(Windows\)](#) (9 March 2023)
- [Introduction to Stimulus in Wagtail for contributors \(video\)](#) (28 February 2023)
- [How to pick a good Wagtail package](#) (1 February 2023)
- [Dockerized Wagtail 4 + NuxtJS 3 + Graphene 3 + Vuetify 3 \(with template\)](#) (26 January 2023)
- [Wagtail: Extending the Draftail Editor Part 4 - Custom Lists](#) (29 December 2022)
- [Making Wagtail pages more SEO friendly with Wagtail Metadata](#) (24 December 2022)
- [Configuring a Dynamic Sitemap on Wagtail](#) (22 December 2022)
- [Deploying Wagtail to Google's Cloud Run](#) (7 December 2022)
- [Tutorial: Build a Wagtail blog in 20 minutes](#) (5 December 2022)
- [Headless Wagtail and Next.js preview mode](#) (25 November 2022)
- [A Step-by-Step Guide on Installing the Wagtail Codebase on a Local Machine for Contribution \(video\)](#) (19 November 2022)
- [How we created the new Wagtail.org](#) (16 November 2022)
- [Build a Blog With Wagtail CMS \(4.0.0\) Released](#) (7 November 2022)
- [Create a custom Wagtail Image filter for Thumbnails with Preserved Edges](#) (4 November 2022)
- [Static-Dynamic Content With In-Memory SQLite using Wagtail](#) (3 November 2022)
- [A Step-by-Step Guide for Manually Setting up Bakery Demo with Wagtail](#) (18 November 2022)
- [Integrating Sa11y accessibility checker into a Wagtail website \(video\)](#) (26 October 2022)
- [Wagtail: Extending the Draftail Editor Part 3 - Dynamic Text](#) (21 October 2022)
- [What's this? A new website? - Explainer for building a new website with Wagtail](#) (10 October 2022)
- [Guide to integrate Wagtail CRX with a Snipcart storefront](#) (9 October 2022)
- [Adding featured events to the HomePage with Wagtail 4.0 \(video\)](#) (6 October 2022)
- [Wagtail: Extending the Draftail Editor Part 2 - Block Styles](#) (5 October 2022)
- [Wagtail: Extending the Draftail Editor Part 1 - Inline Styles](#) (5 October 2022)
- [Creating an interactive event budgeting tool within Wagtail](#) (4 October 2022)
- [Configuring Rich Text Blocks for Your Wagtail Site](#) (26 September 2022)
- [Deploy Django Wagtail to Render](#) (23 September 2022)

- Using a migration to apply permissions to Wagtail snippets (7 September 2022)
- Deploying a Wagtail site to Fly.io - Part 1 of 5 (30 August 2022)
- Django Wagtail CMS | Building A Blog In 20 Minutes (video) (12 August 2022)
- 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)
- Deploying Wagtail on Divio (~September 2021)

- [How to Install Wagtail on Shared Hosting without Root \(cPanel\)](#) (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 \(Video\)](#) (28 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)
- [Deploying Wagtail to Heroku](#) (15 July 2015)
- [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 share your Wagtail HOWTOs, development notes, or site launches in the [Wagtail Slack workspace](#) in #watercooler, or feel free to reach out directly via [email](#).

1.4.12 Testing your Wagtail site

Wagtail comes with some utilities that simplify writing tests for your site.

WagtailPageTestCase

`class wagtail.test.utils.WagtailPageTestCase` WagtailPageTestCase 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 WagtailPageTestCase
from myapp.models import MyPage

class MyPageTests(WagtailPageTestCase):
    def test_can_create_a_page(self):
        ...
```

`assertPageIsRoutable(page, route_path="/", msg=None)`

Asserts that page can be routed to without raising a `Http404` error.

For page types with multiple routes, you can use `route_path` to specify an alternate route to test.

This assertion is great for getting coverage on custom routing logic for page types. Here is an example:

```
from wagtail.test.utils import WagtailPageTestCase
from myapp.models import EventListPage

class EventListPageRoutabilityTests(WagtailPageTestCase):
    @classmethod
    def setUpTestData(cls):
        # create page(s) for testing
        ...

    def test_default_route(self):
        self.assertPageIsRoutable(self.page)

    def test_year_archive_route(self):
        # NOTE: Despite this page type raising a 404 when no events exist for
        # the specified year, routing should still be successful
        self.assertPageIsRoutable(self.page, "archive/year/1984/")
```

```
assertPageIsRenderable(page, route_path="/", query_data=None, post_data=None, user=None, accept_404=False, accept_redirect=False, msg=None)
```

Asserts that page can be rendered without raising a fatal error.

For page types with multiple routes, you can use `route_path` to specify a partial path to be added to the page's regular url.

When `post_data` is provided, the test makes a POST request with `post_data` in the request body. Otherwise, a GET request is made.

When supplied, `query_data` is always converted to a querystring and added to the request URL.

When `user` is provided, the test is conducted with them as the active user.

By default, the assertion will fail if the request to the page URL results in a 301, 302 or 404 HTTP response. If you are testing a page/route where a 404 response is expected, you can use `accept_404=True` to indicate this, and the assertion will pass when encountering a 404 response. Likewise, if you are testing a page/route where a redirect response is expected, you can use `accept_redirect=True` to indicate this, and the assertion will pass when encountering 301 or 302 response.

This assertion is great for getting coverage on custom rendering logic for page types. Here is an example:

```
def test_default_route_rendering(self):
    self.assertPageIsRenderable(self.page)

def test_year_archive_route_with_zero_matches(self):
    # NOTE: Should raise a 404 when no events exist for the specified year
    self.assertPageIsRenderable(self.page, "archive/year/1984/", accept_404=True)

def test_month_archive_route_with_zero_matches(self):
    # NOTE: Should redirect to year-specific view when no events exist for the
    # specified month
    self.assertPageIsRenderable(self.page, "archive/year/1984/07/", accept_
    # redirect=True)
```

```
assertPageIsEditable(page, post_data=None, user=None, msg=None)
```

Asserts that the page edit view works for page without raising a fatal error.

When `user` is provided, the test is conducted with them as the active user. Otherwise, a superuser is created and used for the test.

After a successful GET request, a POST request is made with field data in the request body. If `post_data` is provided, that will be used for this purpose. If not, this data will be extracted from the GET response HTML.

This assertion is great for getting coverage on custom fields, panel configuration and custom validation logic. Here is an example:

```
def test_editability(self):
    self.assertPageIsEditable(self.page)

def test_editability_on_post(self):
    self.assertPageIsEditable(
        self.page,
        post_data={
            "title": "Fabulous events",
            "slug": "events",
            "show_featured": True,
            "show_expired": False,
            "action-publish": ""})
```

(continues on next page)

(continued from previous page)

```

        }
    )
}
```

assertPageIsPreviewable(*page*, *mode*='', *post_data*=None, *user*=None, *msg*=None)

Asserts that the page preview view can be loaded for *page* without raising a fatal error.

For page types that support different preview modes, you can use *mode* to specify the preview mode to be tested.

When *user* is provided, the test is conducted with them as the active user. Otherwise, a superuser is created and used for the test.

To load the preview, the test client needs to make a POST request including all required field data in the request body. If *post_data* is provided, that will be used for this purpose. If not, the method will attempt to extract this data from the page edit view.

This assertion is great for getting coverage on custom preview modes, or getting reassurance that custom rendering logic is compatible with Wagtail's preview mode. Here is an example:

```

def test_general_previewability(self):
    self.assertPageIsPreviewable(self.page)

def test_archive_previewability(self):
    self.assertPageIsPreviewable(self.page, mode="year-archive")
```

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, *publish*=True) 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.

publish specifies whether the page being created should be published or not - default is True.

```

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'),

```

(continues on next page)

(continued from previous page)

```
    ])
}))
```

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})

    # 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': '',  
# }
```

Creating Page objects within tests

If you want to create page objects within tests, you will need to go through some steps before actually creating the page you want to test.

- Pages can't be created directly with `MyPage.objects.create()` as you would do with a regular Django model, they need to be added as children to a parent page with `parent.add_child(instance=child)`.
- To start the page tree, you need a root page that can be created with `Page.get_first_root_node()`.
- You also need a Site set up with the correct hostname and a `root_page`.

```
from wagtail.models import Page, Site
from wagtail.rich_text import RichText
from wagtail.test.utils import WagtailPageTestCase

from home.models import HomePage, MyPage

class MyPageTest(WagtailPageTestCase):
    @classmethod
    def setUpTestData(cls):
        root = Page.get_first_root_node()
        Site.objects.create(
            hostname="testserver",
            root_page=root,
            is_default_site=True,
            site_name="testserver",
        )
        home = HomePage(title="Home")
        root.add_child(instance=home)
        cls.page = MyPage(
            title="My Page",
            slug="mypage",
        )
        home.add_child(instance=cls.page)

    def test_get(self):
        response = self.client.get(self.page.url)
        self.assertEqual(response.status_code, 200)
```

Working with Page content

You will likely want to test the content of your page. If it includes a StreamField, you will need to set its content as a list of tuples with the block's name and content. For RichTextBlock, the content has to be an instance of RichText.

```
...
from wagtail.rich_text import RichText

class MyPageTest(WagtailPageTestCase):
    @classmethod
    def setUpTestData(cls):
        ...
        # Create page instance here
        cls.page.body.extend(
            [
                ("heading", "Just a CharField Heading"),
            ]
        )
```

(continues on next page)

(continued from previous page)

```
        ("paragraph", RichText("<p>First paragraph</p>")),
        ("paragraph", RichText("<p>Second paragraph</p>")),
    ]
)
cls.page.save()

def test_page_content(self):
    response = self.client.get(self.page.url)
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, "Just a CharField Heading")
    self.assertContains(response, "<p>First paragraph</p>")
    self.assertContains(response, "<p>Second paragraph</p>")
```

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 customized 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",
            "content_type": ["website", "homepage"],
            "depth": 2
        }
    },
    {
        "model": "website.homepage",
        "pk": 3,
        "fields": {}
    }
]
```

Treebeard fields

Filling in the `path` / `numchild` / `depth` fields is necessary 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.

1.4.13 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 stabilization 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 multiple 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`
- Redirects `wagtail.contrib.redirects.api.RedirectsAPIViewSet` see [API](#)

You can subclass any of these endpoint classes to customize their functionality. For example, in this case, if you need to change the `APIViewSet` by setting a desired renderer class:

```
from rest_framework.renderers import JSONRenderer

# ...

class CustomPagesAPIViewSet(PagesAPIViewSet):
    renderer_classes = [JSONRenderer]
    name = "pages"

api_router.register_endpoint("pages", CustomPagesAPIViewSet)
```

Or changing the desired model to use for page results.

```
from rest_framework.renderers import JSONRenderer

# ...

class PostPagesAPIViewSet(PagesAPIViewSet):
    model = models.BlogPage

api_router.register_endpoint("posts", PostPagesAPIViewSet)
```

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 = [
```

(continues on next page)

(continued from previous page)

```

    ...
    path('api/v2/', api_router.urls),
    ...

# Ensure that the api_router line appears above the default Wagtail page serving
route
    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`.

Adding form fields to the API

If you have a FormBuilder page called `FormPage` this is an example of how you would expose the form fields to the API:

```
from wagtail.api import APIField

class FormPage(AbstractEmailForm):
    ...
    api_fields = [
        APIField('form_fields'),
    ]
```

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):
    ...

    api_fields = [
        # Adds information about the source image (eg, title) into the API
        APIField('feed_image'),

        # Adds a URL to a rendered thumbnail of the image to the API
        APIField('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.

For cases where the source image set may contain SVGs, the `ImageRenditionField` constructor takes a `preserve_svg` argument. The behavior of `ImageRenditionField` when `preserve_svg` is `True` is as described for the `image` template tag's `preserve-svg` argument (see the documentation on [SVG images](#)).

Authentication

To protect the access to your API, you can implement an `authentication` method provided by the Django REST Framework, for example the `Token Authentication`:

```
# api.py

from rest_framework.permissions import IsAuthenticated

# ...

class CustomPagesAPIViewSet(PageViewSet):
    name = "pages"
    permission_classes = (IsAuthenticated,)

api_router.register_endpoint("pages", CustomPagesAPIViewSet)
```

Extend settings with

```
# settings.py

INSTALLED_APPS = [
    ...
    'rest_framework.authtoken',
    ...
]

REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "rest_framework.authentication.TokenAuthentication"
    ],
}
```

Don't forget to run the app's migrations.

Your API endpoint will be accessible only with the Authorization header containing the generated `Token exampleS-
secretToken123xyz`. Tokens can be generated in the Django admin under Auth Token or using the `manage.py command drf_create_token`.

Note: If you use `TokenAuthentication` in production you must ensure that your API is only available over `https`.

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 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*
 - * *Multiple 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": {
```

(continues on next page)

(continued from previous page)

```

        "type": "app_name.ModelName",
        "detail_url": "http://api.example.com/api/v2/endpoint_name/1/"
    },
    "field": "value"
},
{
    "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.

Multiple ordering

Multiple fields can be passed into the `?order` for consecutive ordering.

```
GET /api/v2/pages/?order=title,-slug

HTTP 200 OK
Content-Type: application/json

{
    "meta": {
        "total_count": 50
    },
    "items": [
        pages will be ordered by title and for all matching titles (a-z), then sorted
        (continues on next page)
    ]
}
```

(continued from previous page)

```
    ↵by slug (z-a).  
    ]  
}
```

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": {  
        "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",  
            }  
        }  
    ]  
}
```

(continues on next page)

(continued from previous page)

```

        "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 the 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 filters the list to only include descendants of that page (children, grandchildren, etc.).

Filtering pages by site

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 internationalization](#) 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

{
    "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"
            },
        }
    ]
}
```

(continues on next page)

(continued from previous page)

```
        "title": "British page"
    },
    {
        "id": 12,
        "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 the 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 customizing 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 `page` 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

1.4.14 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 to 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 behavior 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():
        response = wagtail_serve(request, path)
```

(continues on next page)

(continued from previous page)

```
# Render template responses now while AMP mode is still active
if isinstance(response, SimpleTemplateResponse):
    response.render()

return response
```

Then we need to create an `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 an `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 = [
{}
```

(continues on next page)

(continued from previous page)

```
...
'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 to 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

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()
```

(continues on next page)

(continued from previous page)

```
    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
```

1.4.15 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

Wherever an image is used, the content editor should be able to mark the image as decorative or provide a context-specific text alternative. The image embed in our rich text editor supports this behavior. Wagtail 6.3 added *ImageBlock* to provide this behavior for images within StreamFields.

Wagtail 6.3 also added an optional `description` field to the Wagtail image model and to custom image models inheriting from `wagtail.images.models.AbstractImage`. Text in that field will be offered as the default alt text when inserting images in rich text or using `ImageBlock`. If the `description` field is empty, the `title` field will be used instead. If you would like to customize this behavior, `override the default_alt_text property` in your image model.

Note

Important considerations

- Alt text should be written based on the context the image is displayed in.
- When specifying alt text fields, make sure they are optional so editors can choose to not write any alt text for decorative images. An image might be decorative in some cases but not in others. For example, thumbnails in page listings can often be considered decorative.

- If the alt text's content is already part of the rest of the page, ideally the image should not repeat the same content.
- Take the time to provide `help_text` with appropriate guidance. For example, linking to established resources on alt text.

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 often a problem with OEmbed providers. 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 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

Screen readers will use row and column headers to announce the context of each table cell. Please encourage editors to set row headers and/or column headers as appropriate for their table.

Always add a Caption, so screen reader users navigating the site's tables get an overview of the table content before it is read.

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 modeling](#) section above. Additionally, make sure to [customize 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 easy for editors to create a heading block but not add any content to it. The [built-in accessibility checker](#) will highlight empty headings so editors can find and fix them. If you need stricter enforcement:

- 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.

Alternately, 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

A number of built-in tools and additional resources are available to help create accessible content.

Built-in accessibility checker

Wagtail includes an accessibility checker built into the [user bar](#) and editing views supporting previews. The checker can help authors create more accessible websites following best practices and accessibility standards like [WCAG](#).

The checker is based on the [Axe](#) testing engine and scans the loaded page for errors.

By default, the checker includes the following rules to find common accessibility issues in authored content:

- `button-name`: `<button>` elements must always have a text label.
- `empty-heading`: This rule checks for headings with no text content. Empty headings are confusing to screen readers users and should be avoided.
- `empty-table-header`: Table header text should not be empty
- `frame-title`: `<iframe>` elements must always have a text label.
- `heading-order`: This rule checks for incorrect heading order. Headings should be ordered in a logical and consistent manner, with the main heading (`h1`) followed by subheadings (`h2`, `h3`, etc.).
- `input-button-name`: `<input>` button elements must always have a text label.
- `link-name`: `<a>` link elements must always have a text label.
- `p-as-heading`: This rule checks for paragraphs that are styled as headings. Paragraphs should not be styled as headings, as they don't help users who rely on headings to navigate content.
- `alt-text-quality`: A custom rule ensures that image alt texts don't contain anti-patterns like file extensions and underscores.

To customize how the checker is run (such as what rules to test), you can define a custom subclass of [`AccessibilityItem`](#) and override the attributes to your liking. Then, swap the instance of the default `AccessibilityItem` with an instance of your custom class via the [`construct_wagtail_userbar`](#) hook.

For example, Axe's `p-as-heading` rule evaluates combinations of font weight, size, and italics to decide if a paragraph is acting as a heading visually. Depending on your heading styles, you might want Axe to rely only on font weight to flag short, bold paragraphs as potential headings.

```
from wagtail.admin.userbar import AccessibilityItem

class CustomAccessibilityItem(AccessibilityItem):
    def get_axe_custom_checks(self, request):
        checks = super().get_axe_custom_checks(request)
        # Flag heading-like paragraphs based only on font weight compared to
        # surroundings.
        checks.append(
            {
                "id": "p-as-heading",
                "options": {
                    "margins": [
                        { "weight": 150 },
                    ],
                    "passLength": 1,
                    "failLength": 0.5
                }
            }
        )
        return checks
```

(continues on next page)

(continued from previous page)

```
        },
    },
)
return checks

@hooks.register('construct_wagtail_userbar')
def replace_userbar_accessibility_item(request, items, page):
    items[:] = [CustomAccessibilityItem() if isinstance(item, AccessibilityItem) else_
    item for item in items]
```

The checks you run in production should be restricted to issues your content editors can fix themselves; warnings about things out of their control will only teach them to ignore all warnings. However, it may be useful for you to run additional checks in your development environment.

```
from wagtail.admin.userbar import AccessibilityItem

class CustomAccessibilityItem(AccessibilityItem):
    # Run all Axe rules with these tags in the development environment
    axe_rules_in_dev = [
        "wcag2a",
        "wcag2aa",
        "wcag2aaa",
        "wcag21a",
        "wcag21aa",
        "wcag22aa",
        "best-practice",
    ]
    # Except for the color-contrast-enhanced rule
    axe_rules = {
        "color-contrast-enhanced": {"enabled": False},
    }

    def get_axe_run_only(self, request):
        if env.bool('DEBUG', default=False):
            return self.axe_rules_in_dev
        else:
            # In production, run Wagtail's default accessibility rules for authored_
            # content only
            return self.axe_run_only

@hooks.register('construct_wagtail_userbar')
def replace_userbar_accessibility_item(request, items, page):
    items[:] = [CustomAccessibilityItem() if isinstance(item, AccessibilityItem) else_
    item for item in items]
```

AccessibilityItem reference

The following is the reference documentation for the `AccessibilityItem` class:

class wagtail.admin.userbar.AccessibilityItem

A userbar item that runs the accessibility checker.

axe_include = ['body']

A list of CSS selector(s) to test specific parts of the page. For more details, see [Axe documentation](#).

axe_exclude = []

A list of CSS selector(s) to exclude specific parts of the page from testing. For more details, see [Axe documentation](#).

axe_run_only

A list of `axe-core` tags or a list of `axe-core` rule IDs (not a mix of both). Setting this to a falsy value (e.g. `None`) will omit the `runOnly` option and make Axe run with all non-experimental rules enabled.

axe_rules = {}

A dictionary that maps `axe-core` rule IDs to a dictionary of rule options, commonly in the format of `{"enabled": True/False}`. This can be used in conjunction with `axe_run_only` to enable or disable specific rules. For more details, see [Axe documentation](#).

axe_custom_rules

A list to add custom Axe rules or override their properties, alongside with `axe_custom_checks`. Includes Wagtail's custom rules. For more details, see [Axe documentation](#).

axe_custom_checks

A list to add custom Axe checks or override their properties. Should be used in conjunction with `axe_custom_rules`. For more details, see [Axe documentation](#).

axe_messages

A dictionary that maps `axe-core` rule IDs to custom translatable strings to use as the error messages. If an enabled rule does not exist in this dictionary, Axe's error message for the rule will be used as fallback.

The above attributes can also be overridden via the following methods to allow per-request customization. When overriding these methods, be mindful of the mutability of the class attributes above. To avoid unexpected behavior, you should always return a new object instead of modifying the attributes directly in the methods.

get_axe_include(request)

get_axe_exclude(request)

get_axe_run_only(request)

get_axe_rules(request)

get_axe_custom_rules(request)

get_axe_custom_checks(request)

get_axe_messages(request)

For more advanced customization, you can also override the following methods:

get_axe_context(request)

Returns the `context` object to be passed as the `context` parameter for `axe.run`.

`get_axe_options (request)`

Returns the options object to be passed as the `options` parameter for `axe.run`.

`get_axe_spec (request)`

Returns spec for Axe, including custom rules and custom checks

wagtail-accessibility

`wagtail-accessibility` is a third-party package which adds `total1y` 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.

prefers-reduced-motion

Some users, such as those with vestibular disorders, may prefer a more static version of your site. You can respect this preference by using the `prefers-reduced-motion` media query in your CSS.

```
@media (prefers-reduced-motion) {
    /* styles to apply if a user's device settings are set to reduced motion */
    /* for example, disable animations */
    *
    animation: none !important;
    transition: none !important;
}
}
```

Note that `prefers-reduced-motion` is only applied for users who enabled this setting in their operating system or browser. This feature is supported by Chrome, Safari and Firefox. For more information on reduced motion, see the MDN Web Docs.

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

1.4.16 Sustainability considerations

Here are guidelines and resources we recommend for projects with sustainability goals relating to climate action, such as the UN's [Sustainable Development Goal 13: Climate action](#), and SBTi's [Corporate Net-Zero Standard](#).

Standards

To account for the emissions of websites and track their reduction, we recommend the following:

- ITU [L.1420](#) and [L.1430](#)
- GHG Protocol [Product Life Cycle Accounting and Reporting Standard](#) (Scope 3), and its additional [ICT Sector Guidance](#).

Those are the same standards used to assess the sustainability of Wagtail.

Guidelines

Here are the guidelines we would recommend applying to Wagtail websites:

- Sustainable Web Design W3C Interest Group working draft of the [Web Sustainability Guidelines](#)
- Sustainable Web Design
- [GR491](#)
- Green Design Principles by Microsoft (PDF)
- Green Software Foundation Patterns

Quantifying emissions

To quantify the emissions of a Wagtail website, we recommend three different approaches:

- The [Sustainable Web Design](#) model, which uses page weight as a metric of energy efficiency, and page views as a metric of site utilization. This model has clear [known limitations](#), but is nonetheless ideal to provide high-level figures for a wide range of websites or pages.
- Infrastructure-based calculators such as [Cloud Carbon Footprint](#), a measurement and analysis tools.
- Measurement orchestration tools such as [Green Metrics](#), [GreenFrame](#), [Scaphandre](#).

We are working on those considerations as part of Wagtail's development process. An example of this is the two [Google Summer of Code](#) internships focusing on sustainability, in partnership with the [Green Web Foundation](#) and [Green Coding Berlin](#).

1.4.17 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 `IntegerField`, 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()),
        # ...
    ])
```

```
{% 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.

1.4.18 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 a 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/acom.py

from base import * # noqa

ALLOWED_HOSTS = ['a.com']
DATABASES["NAME"] = "acom"
DATABASES["PASSWORD"] = "password-for-acom"
MEDIA_DIR = BASE_DIR / "acom-media"
```

Each site can be started with its own settings file. In development `./manage.py runserver --settings settings.acom`. In production, for example with uWSGI, specify the correct settings with `env = DJANGO_SETTINGS_MODULE=settings.acom`.

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 differentiating 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 with 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 customize 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.

1.4.19 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

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:
```

(continues on next page)

(continued from previous page)

```

        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)

```

1.4.20 StreamField migrations

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 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`):

```

import json

from django.core.serializers.json import DjangoJSONEncoder
from django.db import migrations

import wagtail.blocks
import wagtail.fields


def convert_to_streamfield(apps, schema_editor):
    BlogPage = apps.get_model("demo", "BlogPage")
    for page in BlogPage.objects.all():
        page.body = json.dumps(
            [{"type": "rich_text", "value": page.body}],
            cls=DjangoJSONEncoder
        )
        page.save()

def convert_to_richtext(apps, schema_editor):
    BlogPage = apps.get_model("demo", "BlogPage")
    for page in BlogPage.objects.all():
        if page.body:

```

(continues on next page)

(continued from previous page)

```
stream = json.loads(page.body)
page.body = "".join([
    child["value"] for child in stream
    if child["type"] == "rich_text"
])
page.save()

class Migration(migrations.Migration):

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

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

        # leave the generated AlterField intact!
        migrations.AlterField(
            model_name="BlogPage",
            name="body",
            field=wagtail.fields.StreamField(
                [("rich_text", wagtail.blocks.RichTextBlock())],
            ),
        ),
    ],
]
```

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:

```
import json

from django.contrib.contenttypes.models import ContentType
from django.core.serializers.json import DjangoJSONEncoder
from django.db import migrations

import wagtail.blocks
import wagtail.fields

def page_to_streamfield(page):
    changed = False
    try:
        json.loads(page.body)
    except ValueError:
        page.body = json.dumps(
            [{"type": "rich_text", "value": page.body}],
        )
        changed = True
    else:
        # It's already valid JSON. Leave it.
        pass
```

(continues on next page)

(continued from previous page)

```

    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": body,
                        "type": "rich_text"
                    },
                    cls=DjangoJSONEncoder)
            changed = True
    else:
        # It's already valid JSON. Leave it.
        pass
    return revision_data, changed

def page_to_richtext(page):
    changed = False
    if page.body:
        try:
            body_data = json.loads(page.body)
        except ValueError:
            # It's not apparently a StreamField. Leave it.
            pass
        else:
            page.body = "".join([
                child["value"] for child in body_data
                if child["type"] == "rich_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"] for child in body_data
                if child["type"] == "rich_text"
            ])
            revision_data["body"] = raw_text
            changed = True

```

(continues on next page)

(continued from previous page)

```
return revision_data, changed

def convert(apps, schema_editor, page_converter, pagerevision_converter):
    BlogPage = apps.get_model("demo", "BlogPage")
    content_type = ContentType.objects.get_for_model(BlogPage)
    Revision = apps.get_model("wagtailcore", "Revision")

    for page in BlogPage.objects.all():

        page, changed = page_converter(page)
        if changed:
            page.save()

        for revision in Revision.objects.filter(
            content_type_id=content_type.pk, object_id=page.pk
        ):
            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)

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"),
        ("wagtailcore", "0076_modellogentry_revision"),
    ]

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

        # leave the generated AlterField intact!
        migrations.AlterField(
            model_name="BlogPage",
            name="body",
            field=wagtail.fields.StreamField(
                [("rich_text", wagtail.blocks.RichTextBlock())],
            ),
        ),
    ],
]
```

StreamField data migrations

Wagtail provides a set of utilities for creating data migrations on StreamField data. These are exposed through the modules:

- `wagtail.blocks.migrations.migrate_operation`
- `wagtail.blocks.migrations.operations`
- `wagtail.blocks.migrations.utils`

 Note

An add-on package `wagtail-streamfield-migration-toolkit` is available, additionally providing limited support for auto-generating migrations.

Why are data migrations necessary?

If you change the block definition of a StreamField on a model that has existing data, you may have to manually alter that data to match the new format.

A StreamField is stored as a single column of JSON data in the database. Blocks are stored as structures within the JSON, and can be nested. However, as far as Django is concerned when generating schema migrations, everything inside this column is just a string of JSON data. The database schema doesn't change - regardless of the content/structure of the StreamField - since it is the same field type before and after any change to the StreamField's blocks. Therefore whenever changes are made to StreamFields, any existing data must be changed into the new required structure, typically by defining a data migration. If the data is not migrated, even a simple change like renaming a block can result in old data being lost.

Generally, data migrations are performed manually by making an empty migration file and writing the forward and backward functions for a `RunPython` command. These functions handle the logic for taking the previously saved JSON representation and converting it into the new JSON representation needed. While this is fairly straightforward for simple changes (such as renaming a block), this can easily get very complicated when nested blocks, multiple fields, and revisions are involved.

To reduce boilerplate, and the potential for errors, `wagtail.blocks.migrations` provides the following:

- utilities to recurse through stream data structures and apply changes; and
- operations for common use cases like renaming, removing and altering values of blocks.

Basic usage

Suppose we have a `BlogPage` model in an app named `blog`, defined as follows:

```
class BlogPage(Page):
    content = StreamField([
        ("stream1", blocks.StreamBlock([
            ("field1", blocks.CharBlock())
        ])),
    ])
```

After running the initial migrations and populating the database with some records, we decide to rename `field1` to `block1`.

```
class BlogPage(Page):
    content = StreamField([
        ("stream1", blocks.StreamBlock([
            ("block1", blocks.CharBlock())
        ])),
    ])
```

Even though we changed the name to `block1` in our `StreamField` definition, the actual data in the database will not reflect this. To update existing data, we need to create a data migration.

First we create an empty migration file within the app. We can use Django's `makemigrations` command for this:

```
python manage.py makemigrations --empty blog
```

which will generate an empty migration file which looks like this:

```
# Generated by Django 4.0.3 on 2022-09-09 21:33

from django.db import migrations

class Migration(migrations.Migration):

    dependencies = [...]

    operations = [
    ]
```

We need to make sure that either this migration or one of the migrations it depends on has the Wagtail core migrations as a dependency, since the utilities need the migrations for the `Revision` models to be able to run.

```
dependencies = [
    ('wagtailcore', '0069_log_entry_jsonfield'),
    ...
]
```

(if the project started off with Wagtail 4, '0076_modellogentry_revision' would also be fine)

Next we need a migration operation which Django will run to make our changes. If we weren't using the provided utilities, we would use a `migrations.RunPython` operation, and we would define what data (model, field etc.) we want and how we want to change that data in its forward (function) argument.

Instead, we have a `migrate_operation.MigrateStreamData` operation which will handle accessing the relevant data for us. We need to specify the app name, model name and field name for the relevant `StreamField` as shown below.

```
from django.db import migrations

from wagtail.blocks.migrations.migrate_operation import MigrateStreamData

class Migration(migrations.Migration):

    dependencies = [...]

    operations = [
        MigrateStreamData(
            app_name="blog",
            model_name="BlogPage",
        )
    ]
```

(continues on next page)

(continued from previous page)

```

        field_name="content",
        operations_and_block_paths=[...]
    ),
]

```

In a StreamField, accessing just the field is not enough, since we will typically need to operate on specific block types. For this, we define a block path which points to that specific block path within the StreamField definition to obtain the specific data we need. Finally, we define an operation to update that data. As such we have an (`IntraFieldOperation()`, 'block_path') tuple. We can have as many as these as we like in our `operations_and_block_paths`, but for now we'll look at a single one for our rename operation.

In this case the block that we are operating on is `stream1`, the parent of the block being renamed (refer to [RenameStreamChildrenOperation](#) - for rename and remove operations we always operate on the parent block). In that case our block path will be `stream1`. Next we need a function that will update our data. For this, the `wagtail.blocks.operations` module has a set of commonly used intra-field operations available (and it is possible to write [custom operations](#)). Since this is a rename operation that operates on a StreamField, we will use `wagtail.blocks.operations.RenameStreamChildrenOperation` which accepts two arguments as the old block name and the new block name. As such our operation and block path tuple will look like this:

```
(RenameStreamChildrenOperation(old_name="field1", new_name="block1"), "stream1")
```

And our final code will be:

```

from django.db import migrations

from wagtail.blocks.migrations.migrate_operation import MigrateStreamData
from wagtail.blocks.migrations.operations import RenameStreamChildrenOperation

class Migration(migrations.Migration):

    dependencies = [
        ...
    ]

    operations = [
        MigrateStreamData(
            app_name="blog",
            model_name="BlogPage",
            field_name="content",
            operations_and_block_paths=[
                (RenameStreamChildrenOperation(old_name="field1", new_name="block1"),
                 "stream1"),
            ]
        ),
    ]

```

Using operations and block paths properly

The `MigrateStreamData` class takes a list of operations and corresponding block paths as a parameter `operations_and_block_paths`. Each operation in the list will be applied to all blocks that match the corresponding block path.

```
operations_and_block_paths=[  
    (operation1, block_path1),  
    (operation2, block_path2),  
    ...  
]
```

Block path

The block path is a `'.'`-separated list of names of the block types from the top level `StreamBlock` (the container of all the blocks in the `StreamField`) to the nested block type which will be matched and passed to the operation.

Note

If we want to operate directly on the top level `StreamBlock`, then the block path must be an empty string `" "`.

For example, if our stream definition looks like this:

```
class MyDeepNestedBlock(StreamBlock):  
    foo = CharBlock()  
    date = DateBlock()  
  
class MyNestedBlock(StreamBlock):  
    char1 = CharBlock()  
    deepnested1 = MyDeepNestedBlock()  
  
class MyStreamBlock(StreamBlock):  
    field1 = CharBlock()  
    nested1 = MyNestedBlock()  
  
class MyPage(Page):  
    content = StreamField(MyStreamBlock)
```

If we want to match all “`field1`” blocks, our block path will be `"field1"`:

```
[  
    { "type": "field1", ... }, # this is matched  
    { "type": "field1", ... }, # this is matched  
    { "type": "nested1", "value": [...] },  
    { "type": "nested1", "value": [...] },  
    ...  
]
```

If we want to match all “`deepnested1`” blocks, which are a direct child of “`nested1`”, our block path will be `"nested1.deepnested1"`:

```
[  
    { "type": "field1", ... },
```

(continues on next page)

(continued from previous page)

```

{ "type": "field1", ... },
{ "type": "nested1", "value": [
    { "type": "char1", ... },
    { "type": "deepnested1", ... }, # This is matched
    { "type": "deepnested1", ... }, # This is matched
    ...
] },
{ "type": "nested1", "value": [
    { "type": "char1", ... },
    { "type": "deepnested1", ... }, # This is matched
    ...
] },
...
]

```

When the path contains a ListBlock child, ‘item’ must be added to the block path as the name of said child. For example, if we consider the following stream definition:

```

class MyStructBlock(StructBlock):
    char1 = CharBlock()
    char2 = CharBlock()

class MyStreamBlock(StreamBlock):
    list1 = ListBlock(MyStructBlock())

```

Then if we want to match “char1”, which is a child of the StructBlock which is the direct list child, we have to use `block_path_str="list1.item.char1"` instead of `block_path_str="list1.char1"`. We can also match the ListBlock child with `block_path_str="list1.item"`.

Rename and remove operations

The following operations are available for renaming and removing blocks.

- *RenameStreamChildrenOperation*
- *RenameStructChildrenOperation*
- *RemoveStreamChildrenOperation*
- *RemoveStructChildrenOperation*

Note that all of these operations operate on the value of the parent block of the block which must be removed or renamed. Hence make sure that the block path you are passing points to the parent block when using these operations (see the example in [basic usage](#)).

Alter block structure operations

The following operations allow you to alter the structure of blocks in certain ways.

- *StreamChildrenToListBlockOperation*: operates on the value of a StreamBlock. Combines all child blocks of type `block_name` as children of a single ListBlock which is a child of the parent StreamBlock.
- *StreamChildrenToStreamBlockOperation*: operates on the value of a StreamBlock. Note that `block_names` here is a list of block types and not a single block type unlike `block_name` in the previous operation. Combines each child block of a type in `block_names` as children of a single StreamBlock which is a child of the parent StreamBlock.

- *StreamChildrenToStructBlockOperation*: moves each StreamBlock child of the given type inside a new StructBlock

A new StructBlock will be created as a child of the parent StreamBlock for each child block of the given type, and then that child block will be moved from the parent StreamBlock's children inside the new StructBlock as a child of that StructBlock.

For example, consider the following StreamField definition:

```
mystream = StreamField([("char1", CharBlock()) ...], ...)
```

Then the stream data would look like the following:

```
[  
    { "type": "char1", "value": "Value1", ... },  
    { "type": "char1", "value": "Value2", ... },  
    ...  
]
```

And if we define the operation like this:

```
StreamChildrenToStructBlockOperation("char1", "struct1")
```

Our altered stream data would look like this:

```
[  
    ...  
    { "type": "struct1", "value": { "char1": "Value1" } },  
    { "type": "struct1", "value": { "char1": "Value2" } },  
    ...  
]
```

Note

Block ids are not preserved here since the new blocks are structurally different than the previous blocks.

Other operations

- *AlterBlockValueOperation*

Making custom operations

Basic usage

While this package comes with a set of operations for common use cases, there may be many instances where you need to define your own operation for mapping data. Making a custom operation is fairly straightforward. All you need to do is extend the `BaseBlockOperation` class and define the required methods,

- `apply`
This applies the actual changes to the existing block value and returns the new block value.
- `operation_name_fragment`
(@property) Returns a name to be used for generating migration names.

(**NOTE:** `BaseBlockOperation` inherits from `abc.ABC`, so all of the required methods mentioned above have to be defined on any class inheriting from it.)

For example, if we want to truncate the string in a `CharBlock` to a given length,

```
from wagtail.blocks.migrations.operations import BaseBlockOperation

class MyBlockOperation(BaseBlockOperation):
    def __init__(self, length):
        super().__init__()
        # we will need to keep the length as an attribute of the operation
        self.length = length

    def apply(self, block_value):
        # block value is the string value of the CharBlock
        new_block_value = block_value[:self.length]
        return new_block_value

    @property
    def operation_name_fragment(self):
        return "truncate_{}".format(self.length)
```

block_value

Note that depending on the type of block we're dealing with, the `block_value` which is passed to `apply` may take different structures.

For non-structural blocks, the value of the block will be passed directly. For example, if we're dealing with a `CharBlock`, it will be a string value.

The value passed to `apply` when the matched block is a `StreamBlock` would look like this,

```
[  
  { "type": "...", "value": "...", "id": "..." },  
  { "type": "...", "value": "...", "id": "..." },  
  ...  
]
```

The value passed to `apply` when the matched block is a `StructBlock` would look like this,

```
{  
  "type1": "...",  
  "type2": "...",  
  ...  
}
```

The value passed to `apply` when the matched block is a `ListBlock` would look like this,

```
[  
  { "type": "item", "value": "...", "id": "..." },  
  { "type": "item", "value": "...", "id": "..." },  
  ...  
]
```

Making structural changes

When making changes involving the structure of blocks (changing the block type for example), it may be necessary to operate on the block value of the parent block instead of the block to which the change is made, since only the value of a block is changed by the `apply` operation.

Take a look at the implementation of `RenameStreamChildrenOperation` for an example.

Old list format

Prior to Wagtail version 2.16, `ListBlock` children were saved as just a normal Python list of values. However, for newer versions of Wagtail, list block children are saved as `ListValues`. When handling raw data, the changes would look like the following:

Old format

```
[  
    value1,  
    value2,  
    ...  
]
```

New format

```
[  
    { "type": "item", "id": "...", "value": value1 },  
    { "type": "item", "id": "...", "value": value2 },  
    ...  
]
```

When defining an operation that operates on a `ListBlock` value, in case you have old data which is still in the old format, it is possible to use `wagtail.blocks.migrations.utils.formatted_list_child_generator` to obtain the children in the new format like so:

```
def apply(self, block_value):  
    for child_block in formatted_list_child_generator(list_block_value):  
        ...
```

1.4.21 StreamField validation

All `StreamField` blocks implement a `clean` method which accepts a block value and returns a cleaned version of that value, or raises a `ValidationError` if the value fails validation. Built-in validation rules, such as checking that a `URLBlock` value is a correctly-formatted URL, are implemented through this method. Additionally, for blocks that act as containers for other blocks, such as `StructBlock`, the `clean` method recursively calls the `clean` methods of its child blocks and handles raising validation errors back to the caller as required.

The `clean` method can be overridden on block subclasses to implement custom validation logic. For example, a `StructBlock` that requires either one of its child blocks to be filled in could be implemented as follows:

```
from django.core.exceptions import ValidationError  
from wagtail.blocks import StructBlock, PageChooserBlock, URLBlock  
  
class LinkBlock(StructBlock):  
    page = PageChooserBlock(required=False)
```

(continues on next page)

(continued from previous page)

```
url = URLBlock(required=False)

def clean(self, value):
    result = super().clean(value)
    if not(result['page'] or result['url']):
        raise ValidationError("Either page or URL must be specified")
    return result
```

Note

The validation of the blocks in the StreamField happens through the form field (`wagtail.blocks.base.BlockField`), not the model field (`wagtail.fields.StreamField`).

This means that calling validation methods on your page instance (such as `my_page.full_clean()`) won't catch invalid blocks in the StreamField data.

This should only be relevant when the data in the StreamField is added programmatically, through other paths than the form field.

Controlling where error messages are rendered

In the above example, an exception of type `ValidationError` is raised, which causes the error to be attached and rendered against the StructBlock as a whole. For more control over where the error appears, the exception class `wagtail.blocks.StructBlockValidationError` can be raised instead. The constructor for this class accepts the following arguments:

- `non_block_errors` - a list of error messages or `ValidationError` instances to be raised against the StructBlock as a whole
- `block_errors` - a dict of `ValidationError` instances to be displayed against specific child blocks of the StructBlock, where the key is the child block's name

The following example demonstrates raising a validation error attached to the 'description' block within the StructBlock:

```
from django.core.exceptions import ValidationError
from wagtail.blocks import CharBlock, StructBlock, StructBlockValidationError, TextBlock

class TopicBlock(StructBlock):
    keyword = CharBlock()
    description = TextBlock()

    def clean(self, value):
        result = super().clean(value)
        if result["keyword"] not in result["description"]:
            raise StructBlockValidationError(block_errors={
                "description": ValidationError("Description must contain the keyword")
            })
        return result
```

`ListBlock` and `StreamBlock` also have corresponding exception classes `wagtail.blocks.ListBlockValidationError` and `wagtail.blocks.StreamBlockValidationError`, which work similarly, except that the keys of the `block_errors` dict are the numeric indexes of the blocks where the errors are to be attached:

```
from django.core.exceptions import ValidationError
from wagtail.blocks import ListBlock, ListBlockValidationError

class AscendingListBlock(ListBlock):
    # example usage:
    # price_list = AscendingListBlock(FloatBlock())

    def clean(self, value):
        result = super().clean(value)
        errors = {}
        for i in range(1, len(result)):
            if result[i] < result[i - 1]:
                errors[i] = ValidationError("Values must be in ascending order")

        if errors:
            raise ListBlockValidationError(block_errors=errors)

    return result
```

1.4.22 Manage the reference index

Wagtail maintains a reference index, which records references between objects whenever those objects are saved. The index allows Wagtail to efficiently report the usage of images, documents and snippets within pages, including within StreamField and rich text fields.

Configuration

By default, the index will store references between objects managed within the Wagtail admin, specifically:

- all Page types
- Images
- Documents
- models registered as *Snippets*
- models registered with *ModelViewSet*

The reference index does not require any further configuration. However there are circumstances where it may be necessary to add or remove models from the index.

Registering a Model for Indexing

A model can be registered for reference indexing by adding code to `apps.py` in the app where the model is defined:

```
from django.apps import AppConfig

class Sprocket AppConfig(AppConfig):
    ...
    def ready(self):
        from wagtail.models.reference_index import ReferenceIndex
        from .models import SprocketController
```

(continues on next page)

(continued from previous page)

```
ReferenceIndex.register_model(SprocketController)
```

Preventing Indexing of models and fields

The `wagtail_reference_index_ignore` attribute can be used to prevent indexing with a particular model or model field.

- set the `wagtail_reference_index_ignore` attribute to `True` within any model class where you want to prevent indexing of all fields in the model; or
- set the `wagtail_reference_index_ignore` attribute to `True` within any model field, to prevent that field or the related model field from being indexed:

```
class CentralPage(Page):  
    ...  
    reference = models.ForeignKey(  
        "doc",  
        on_delete=models.SET_NULL,  
        related_name="page_ref",  
    )  
    reference.wagtail_reference_index_ignore = True  
    ...
```

Maintenance

The index can be rebuilt with the `rebuild_references_index` management command. This will repopulate the references table and ensure that reference counts are displayed accurately. This should be done if models are manipulated outside of Wagtail, or after an upgrade.

A summary of the index can be shown with the `show_references_index` management command. This shows the number of objects indexed against each model type, and can be useful to identify which models are being indexed without rebuilding the index itself.

1.4.23 Headless support

Wagtail has good support for headless sites, but there are some limitations developers should take into account when using Wagtail as a headless CMS. This page covers most topics related to headless sites, and tries to identify where you might run into issues using (good support), (workarounds needed or incomplete support) and (lacking support).

Wagtail maintains a current list of issues tagged with `#headless` on [GitHub](#)

API

There are generally two popular options for API when using Wagtail as a headless CMS, REST and GraphQL.

REST

REST (or REpresentational State Transfer) was introduced in 2000 as a simpler approach to machine-to-machine communication using the HTTP protocol. Since REST was introduced, RESTful APIs have proliferated across the web to the point where they're essentially the default standard for modern APIs. Many headless content management systems use either RESTful architecture or GraphQL for their APIs. Both options work with headless Wagtail, so let's explore the upsides and downsides of choosing REST.

Upsides of a REST API

- Requests can be sent using common software like cURL or through web browsers.
- The REST standards are open source and relatively simple to learn.
- REST uses standard HTTP actions like GET, POST, and PUT.
- REST operations require less bandwidth than other comparable technologies (such as SOAP).
- REST is stateless on the server-side, so each request is processed independently.
- Caching is manageable with REST.
- REST is more common currently and there are many more tools available to support REST.
- The REST API is a native feature of Wagtail with some functionality already built in.

Downsides of a REST API

- Sometimes, multiple queries are required to return the necessary data.
- REST isn't always efficient if a query requires access to multiple endpoints.
- Requests to REST APIs can return extra data that's not needed.
- REST depends on fixed data structures that can be somewhat difficult to update.

Note

If you don't want to use Wagtail's built-in REST API, you can build your own using the [Django REST framework](#). Remember, Wagtail is just Django.

GraphQL

GraphQL is a newer API technology than REST. Unlike REST, GraphQL isn't an architecture; it's a data query language that helps simplify API requests. GraphQL was developed by Facebook (now Meta) and open sourced in 2015. It's a newer technology that was designed to provide more flexibility and efficiency than REST. Besides REST, GraphQL is currently the only other API technology that is recommended for headless Wagtail. Let's have a look at the current upsides and downsides of choosing GraphQL.

Upsides of GraphQL

- Changes can be made more rapidly on the client-side of a project without substantial backend updates.
- Queries can be more precise and efficient without over- or under-fetching data.
- You can use fewer queries to retrieve data that would require multiple endpoints in REST.
- GraphQL APIs use fewer resources with fewer queries.
- GraphQL provides options for analytics and performance monitoring.

Downsides of GraphQL

- GraphQL is not natively supported in Wagtail.
- You will need to install a library package to use GraphQL.
- There are currently fewer tools and resources available for supporting GraphQL.
- Fewer developers are familiar with GraphQL.
- GraphQL can introduce additional performance and security considerations due to its flexibility.

GraphQL libraries compatible with Wagtail

- [wagtail-grapple](#) by GrappleGQL
- [strawberry-wagtail](#) by Patrick Arminio

Functionality

Page preview

Previews need a workaround currently.

There currently isn't a way to request a draft version of a page using the public API. We typically recommend [wagtail-headless-preview](#), a mature and widely used third-party package.

When autosave is released in Wagtail, generating previews will likely be less of an obstacle since the API would be serving up the latest changes in all circumstances. This can be achieved using a [workaround](#).

?

Images

Additional image considerations are needed for headless Wagtail.

On traditional sites, Wagtail has a template tag that makes it easy for a frontend developer to request an image of a particular size. Currently, the Wagtail API provides two solutions:

- Add an *ImageRenditionField* to the model, that allows an image in a particular placement on a page to be requested at a pre-defined size. This is the approach we recommend in most cases.
- Use the *dynamic image serve* view, which allows any image to be rendered at any size. Note that this approach may require extra work, since a key is required and you'll need a secure way to pass the key back and forth. Without this, there's a higher risk of crashing your site, by an attacker requesting the same image in millions of subtly different ways.

Neither of these solutions are easy for a frontend developer. They may not have the access or skills to add an `ImageRenditionField`, and crafting a URL to the dynamic image serve view is tricky because it needs to be signed and there currently isn't a library or code snippet to do this from JavaScript. Hashes also need to be generated and the current JS version is complex.

?

Page URL routing

Headless Wagtail requires different routing.

A different approach to routing is needed for headless Wagtail projects. Unlike the traditional routing for Wagtail, the URL patterns on a headless site are usually configured in the frontend framework (such as [Next.js](#) or [Gatsby](#)). Wagtail, by default, resolves URLs to pages using their slugs and location in the page tree.

Because of this default, the “View Live” links in the administration view of Wagtail may resolve to the wrong URL if the URL patterns configured in the frontend framework don’t match the page structure. If rich text is rendered server-side, this will also affect any internal links in rich text fields.

The current recommended approach to routing on a headless Wagtail project is to stick with using the Wagtail routes rather than creating custom routes. Creating custom routes will require more frequent updates and maintenance.

Routes need to be built each time a new site is created and we’d like better documentation to explain this process. One long-term solution for supporting routing in headless Wagtail may be to manage it through a JS library or a plugin.

?

Rich text

There are broadly two approaches to handling rich text in headless Wagtail:

Rendering on the backend

Wagtail stores rich text internally in a HTML-like format with some custom elements to support internal page links and image embeds. On a traditional Wagtail site, those custom tags are converted into standard HTML, but this doesn’t happen in the built-in API. The API returns unprocessed rich text content, which means that users need to either parse the HTML on the frontend and convert the custom tags or they need implement a custom serializer to render the RichText fields on the backend. Currently, [wagtail-grapple](#) pre-renders the HTML. So one solution could be to update the built-in API to also pre-render the HTML.

Rendering on the backend is currently the easier approach to user for managing rich text. Note that using this approach requires page URLs to follow Wagtail’s conventions, so custom routing isn’t possible without some complex configuration.

Rendering on the frontend

Pre-rendering the HTML on the backend may be more convenient if you're happy with the way Wagtail renders it, but it's still difficult to customize the rendering on the frontend. Other headless CMSs provide Rich Text as a sequence of blocks in JSON format. This approach makes it easier to customize the rendering of the blocks without having to find a way to parse the HTML fragment.

This approach is currently the harder approach for managing rich text.

?

Multi-site support

Multi-site works differently in headless Wagtail.

The notion of a “site” is different for headless Wagtail. In traditional Wagtail, the domain or port of a Wagtail site are where Wagtail will serve content and the location the end user visits to find the site.

But in headless Wagtail, the domain or port that the end user uses and domain or port that Wagtail serves content on will be different. For example, the end user may visit www.wagtail.org, and the website could be a Next.js app that queries a Wagtail instance running at api.wagtail.org.

Wagtail's current API implementation will check the host header and port to find the site so that it only returns pages under that site. This means that your site record must be set to api.wagtail.org. However, when Wagtail generates URLs, these URLs need to be generated for www.wagtail.org.

The Wagtail API only allows requests from one site at a time to make sure any site listings are isolated from other sites by default. But the API could be improved in the following ways:

- Allow the site to be specified in the API request.
- Allow all pages across all sites to be queried on an opt-in basis.

With these approaches, the site record in the Wagtail admin of headless Wagtail would be set to the domain or port that the end user sees so URLs could be reversed correctly. All API requests would specify the site as a GET parameter.

?

Form submissions

There's currently no official API for a headless site to use to submit data to a Wagtail form.

?

Password-protected pages

There currently isn't a way to view a password-protected page from a headless frontend. The API currently excludes all password-protected pages from queries.

Frontend

There are a few options to build your frontend for Wagtail.

?

Next.js

Next.js is a [popular open source JavaScript framework](#) you can choose for building the frontend of your headless Wagtail website.

There's no specific support for Next.js in headless Wagtail, but you could take a look at Wagtail's self-paced [guide](#) to Next.js and Wagtail or projects using Wagtail and Next.js on [Github](#), for inspiration and exploration.

?

Nuxt.js

Nuxt.js is an [open source JavaScript framework](#) you can use to build a frontend for your headless Wagtail project. Several high profile sites run a combination of Wagtail and Nuxt.js, including NASA's [Jet Propulsion Laboratory](#). While there is currently no specific support for Nuxt.js, Wagtail's built-in API makes this a straightforward option. Several projects are available on [GitHub](#) for inspiration and exploration.

?

Gatsby

Gatsby is a frontend JavaScript framework for generating static websites that you could use for your headless Wagtail site.

There is currently no specific support for Gatsby in headless Wagtail. There is a plugin available called `gatsby-source-wagtail` you can use for connecting Wagtail to a Gatsby frontend. Choosing to use that plugin means committing to using a GraphQL library for your API, since it only works with the `wagtail-grapple` library.

Supporting platforms

There are many platforms you can use to host your frontend site, here are some that have been used in combination with Wagtail.

?

Vercel

Vercel is a frontend platform for developer teams that uses Next.js.

Currently, there is no plugin available to use Vercel with headless Wagtail. Most of the backend server rendering will generate new content anyway, so you can proceed without a plugin if you want.

?

Netlify

Netlify is a [platform for publishing static websites](#) that can be used to create a frontend for your headless Wagtail site.

There is a plugin available currently that automatically pings Netlify to build a new version of your headless Wagtail site every time you publish called `wagtail-netlify`.

Additional resources

- [Official Wagtail documentation on building a public-facing API](#)
- Wagtail API tutorial from [LearnWagtail.com](#)
- Using Wagtail, NuxtJS and Vuetify to build a fast and secure static site
- Going Headless with Wagtail, Nuxt.js and GraphQL (PDF)

1.5 Extending

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.

Note

The features described in this section and their corresponding reference documentation are not subject to the same level of stability described in our [Deprecation policy](#). Any backwards-incompatible changes to these features will be called out in the upgrade considerations of the [Release notes](#).

1.5.1 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. Using [Snippets](#), Wagtail provides ready-made views for listing, creating, and editing Django models with minimal configuration.

For other kinds of admin views 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	8	9	10	11	12	13	14
11	12	13	14	15	16	17	8	9	10	11	12	13	14	15	16	17	18	19	20	21
18	19	20	21	22	23	24	15	16	17	18	19	20	21	22	23	24	25	26	27	28
25	26	27	28	29	30	31	22	23	24	25	26	27	28	29	30	31				
April							May							June						
Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
5	6	7	8	9	10	11	3	4	5	6	7	8	9	1	2	3	4	5	6	
12	13	14	15	16	17	18	10	11	12	13	14	15	16	14	15	16	17	18	19	20
19	20	21	22	23	24	25	17	18	19	20	21	22	23	21	22	23	24	25	26	27
26	27	28	29	30			24	25	26	27	28	29	30	28	29	30				31
July							August							September						
Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun
5	6	7	8	9	10	11	2	3	4	5	6	7	8	1	2	3	4	5		
12	13	14	15	16	17	18	9	10	11	12	13	14	15	13	14	15	16	17	18	19
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
October							November							December						
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	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						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 a stable part of Wagtail's API and may change in future releases.

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` and `calendar.css` as follows:

```
{% extends "wagtailadmin/base.html" %}
{% load static %}

{% block titletag %}{{ current_year }} calendar{% endblock %}

{% block extra_css %}
    {{ block.super }}
    <link rel="stylesheet" href="{% static 'css/calendar.css' %}">
{% endblock %}

{% block content %}
    {% include "wagtailadmin/shared/header.html" with title="Calendar" icon="date" %}

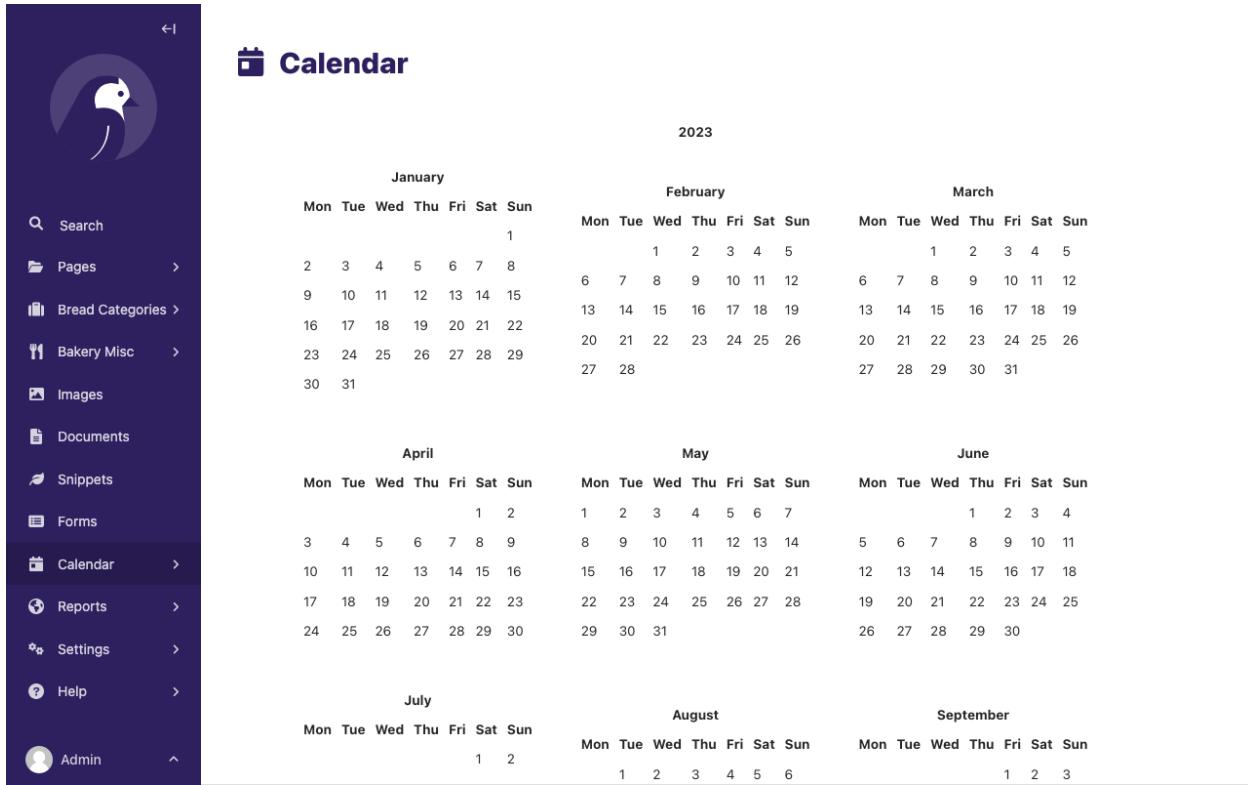
    <div class="nice-padding">
        {{ calendar_html|safe }}
    </div>
{% endblock %}
```

```
/* calendar.css */
table.month {
    margin: 20px;
}

table.month td,
table.month th {
    padding: 5px;
}
```

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 recognized 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

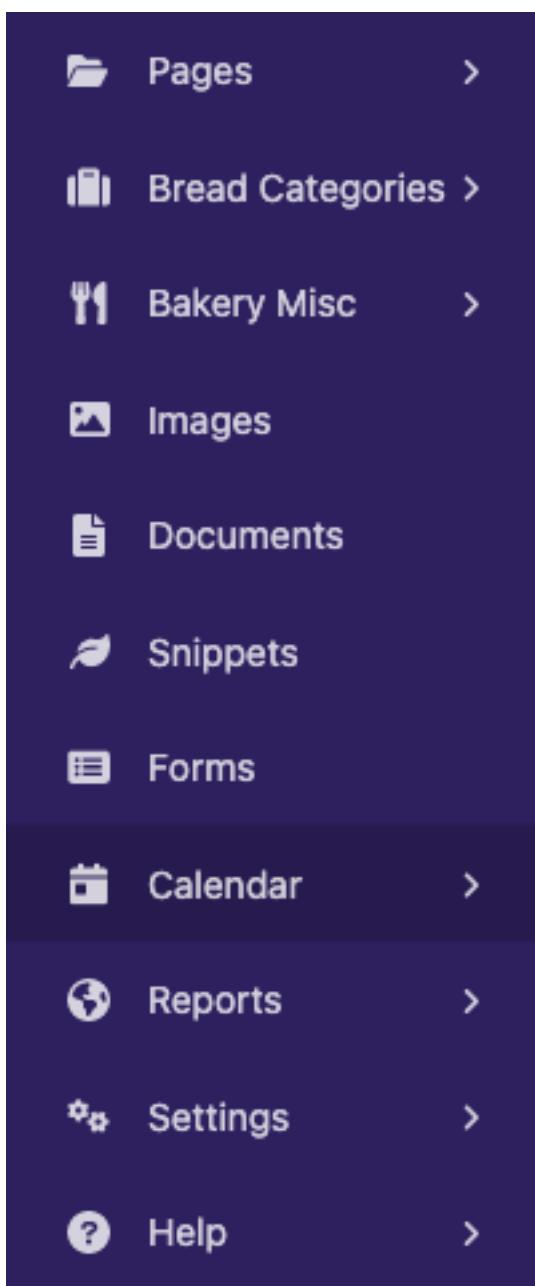
from wagtail.admin.menu import MenuItem
from wagtail import hooks

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 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
```

(continues on next page)

(continued from previous page)

```
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

May 2023						
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	31				

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'),
    ]

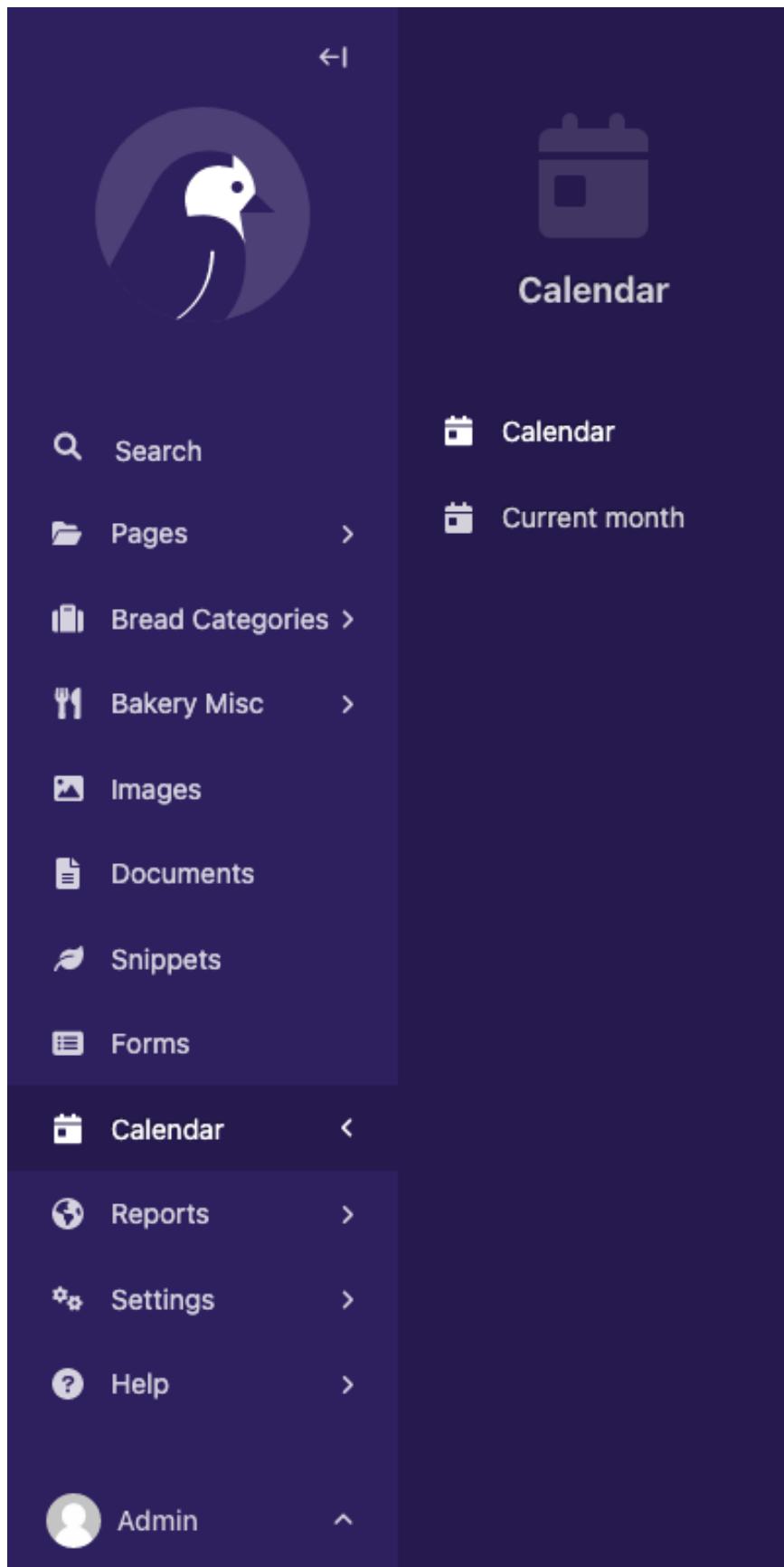
@hooks.register('register_admin_menu_item')
def register_calendar_menu_item():
    submenu = Menu(items=[
        MenuItem('Calendar', reverse('calendar'), icon_name='date'),
        MenuItem('Current month', reverse('calendar-month'), icon_name='date'),
    ])
```

(continues on next page)

(continued from previous page)

```
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.



Using `ViewSet` to group custom admin views

Registering admin views along with their URLs and menu items is a common pattern in Wagtail. This often involves several related views with shared properties such as the model that we're working with, and its associated icon. To support the pattern, Wagtail implements the concept of a `viewset`, which allows a bundle of views and their URLs to be defined collectively, along with a menu item to be registered with the admin app as a single operation through the `register_admin_viewset` hook.

For example, you can group the calendar views from the previous example into a single menu item by creating a `ViewSet` subclass in `views.py`:

```
from wagtail.admin.viewsets.base import ViewSet

...

class CalendarViewSet(ViewSet):
    add_to_admin_menu = True
    menu_label = "Calendar"
    icon = "date"
    # The `name` will be used for both the URL prefix and the URL namespace.
    # They can be customized individually via `url_prefix` and `url_namespace`.
    name = "calendar"

    def get_urlpatterns(self):
        return [
            # This can be accessed at `/admin/calendar/`
            # and reverse-resolved with the name `calendar:index`.
            # This first URL will be used for the menu item, but it can be
            # customized by overriding the `menu_url` property.
            path('', index, name='index'),

            # This can be accessed at `/admin/calendar/month/`
            # and reverse-resolved with the name `calendar:month`.
            path('month/', month, name='month'),
        ]
```

Then, remove the `register_admin_urls` and `register_admin_menu_item` hooks in `wagtail_hooks.py` in favor of registering the `ViewSet` subclass with the `register_admin_viewset` hook:

```
from .views import CalendarViewSet

@hooks.register("register_admin_viewset")
def register_viewset():
    return CalendarViewSet()
```

Compared to the previous example with the two separate hooks, this will result in a single menu item “Calendar” that takes you to the `/admin/calendar/` URL. The second URL will not have its own menu item, but it will still be accessible at `/admin/calendar/month/`. This is useful for grouping related views together, that may not necessarily need their own menu items.

For further customizations, refer to the `ViewSet` documentation.

Combining multiple ViewSets using a ViewSetGroup

The `ViewSetGroup` class can be used to group multiple ViewSets inside a top-level menu item. For example, if you have a different viewset e.g. `EventViewSet` that you want to group with the `CalendarViewSet` from the previous example, you can do so by creating a `ViewSetGroup` subclass in `views.py`:

```
from wagtail.admin.viewsets.base import ViewSetGroup

...
class AgendaViewSetGroup(ViewSetGroup):
    menu_label = "Agenda"
    menu_icon = "table"
    # You can specify instances or subclasses of `ViewSet` in `items`.
    items = (CalendarViewSet(), EventViewSet)
```

Then, remove `add_to_admin_menu` from the viewsets and update the `register_admin_viewset` hook in `wagtail_hooks.py` to register the `ViewSetGroup` instead of the individual viewsets:

```
from .views import AgendaViewSetGroup

@hooks.register("register_admin_viewset")
def register_viewset():
    return AgendaViewSetGroup()
```

This will result in a top-level menu item “Agenda” with the two viewsets’ menu items as sub-items, e.g. “Calendar” and “Events”.

For further customizations, refer to the `ViewSetGroup` documentation.

Adding links in admin views

Snippets

We will use `BreadTypeSnippet` from the Wagtail Bakery demo as an example.

The snippet URL names follow the following pattern: `wagtailsnippets_{app_label}_{model_name}:{list/edit/inspect/copy/delete}` by default.

In Python, you can use `get_url_name()` to get the name of the snippet view URL. (e.g. `BreadTypeSnippet.get_url_name("list")`)

So the `BreadTypeSnippet` URLs would look as follows, when used in templates:

```
{% url 'wagtailsnippets_breads_breadtype:list' %}
{% url 'wagtailsnippets_breads_breadtype:edit' object.id %}
{% url 'wagtailsnippets_breads_breadtype:inspect' object.id %}
{% url 'wagtailsnippets_breads_breadtype:copy' object.id %}
{% url 'wagtailsnippets_breads_breadtype:delete' object.id %}
```

Pages

New page

```
{% url 'wagtailadmin_pages:add' content_type_app_name content_type_model_name parent_
→id %}
```

Page usage

```
{% url 'wagtailadmin_pages:usage' page_id %}
```

Edit page

```
{% url 'wagtailadmin_pages:edit' page_id %}
```

Delete page

```
{% url 'wagtailadmin_pages:delete' page_id %}
```

Copy page

```
{% url 'wagtailadmin_pages:copy' page_id %}
```

Images

Images list

```
{% url 'wagtailimages:index' %}
```

Edit image

```
{% url 'wagtailimages:edit' image_id %}
```

Delete image

```
{% url 'wagtailimages:delete' image_id %}
```

New image

```
{% url 'wagtailimages:add' %}
```

Image usage

```
{% url 'wagtailimages:image_usage' image_id %}
```

AdminURLFinder

To find the url for any model in the admin the AdminURLFinder class can be used.

```
from wagtail.admin.admin_url_finder import AdminURLFinder

finder = AdminURLFinder()

finder.get_edit_url(model_instance)
```

1.5.2 Generic views

Wagtail provides several generic views for handling common tasks such as creating / editing model instances and chooser modals. For convenience, these views are bundled in [viewsets](#).

ModelViewSet

The `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"
    add_to_admin_menu = True
    copy_view_enabled = False
    inspect_view_enabled = True

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
```

(continues on next page)

(continued from previous page)

```
@hooks.register("register_admin_viewset")
def register_viewset():
    return person_viewset
```

The viewset can be further customized by overriding other attributes and methods.

Icon

You can define an `icon` attribute on the `ModelViewSet` to specify the icon that is used across the views in the viewset. The `icon` needs to be [registered in the Wagtail icon library](#).

URL prefix and namespace

The `url_prefix` and `url_namespace` properties can be overridden to use a custom URL prefix and namespace for the views. If unset, they default to the model's `model_name`.

Menu item

By default, registering a `ModelViewSet` will not register a main menu item. To add a menu item, set `add_to_admin_menu` to `True`. Alternatively, if you want to add the menu item inside the “Settings” menu, you can set `add_to_settings_menu` to `True`. Unless `menu_icon` is specified, the menu will use the same `icon` used for the views. The `menu_url` property can be overridden to customize the menu item’s link, which defaults to the listing view for the model.

Unless specified, the menu item will be labeled after the model’s verbose name. You can customize the menu item’s label, name, and order by setting the `menu_label`, `menu_name`, and `menu_order` attributes respectively. If you would like to customize the `MenuItem` instance completely, you could override the `get_menu_item()` method.

You can group multiple `ModelViewSets`’ menu items inside a single top-level menu item using the `ModelViewSet-Group` class. It is similar to `ViewSetGroup`, except it takes the `app_label` of the first viewset’s model as the default `menu_label`. Refer to [the examples for `ViewSetGroup`](#) for more details.

Listing view

The `list_display` attribute can be set to specify the columns shown on the listing view. To customize the number of items to be displayed per page, you can set the `list_per_page` attribute. Additionally, the `ordering` attribute can be used to override the `default_ordering` configured in the listing view.

You can add the ability to filter the listing view by defining a `list_filter` attribute and specifying the list of fields to filter. Wagtail uses the `django-filter` package under the hood, and this attribute will be passed as `django-filter`’s `FilterSet.Meta.fields` attribute. This means you can also pass a dictionary that maps the field name to a list of lookups.

If you would like to make further customizations to the filtering mechanism, you can also use a custom `wagtail.admin.filters.WagtailFilterSet` subclass by overriding the `filterset_class` attribute. The `list_filter` attribute is ignored if `filterset_class` is set. For more details, refer to [django-filter’s documentation](#).

You can add the ability to export the listing view to a spreadsheet by setting the `list_export` attribute to specify the columns to be exported. The `export_filename` attribute can be used to customize the file name of the exported spreadsheet.

Create and edit views

You can define a `panels` or `edit_handler` attribute on the `ModelViewSet` or your Django model to use Wagtail's panels mechanism. For more details, see [Panels](#).

If neither `panels` nor `edit_handler` is defined and the `get_edit_handler()` method is not overridden, the form will be rendered as a plain Django form. You can customize the form by setting the `form_fields` attribute to specify the fields to be shown on the form. Alternatively, you can set the `exclude_form_fields` attribute to specify the fields to be excluded from the form. If panels are not used, you must define `form_fields` or `exclude_form_fields`, unless `get_form_class()` is overridden.

Copy view

The copy view is enabled by default and will be accessible by users with the ‘add’ permission on the model. To disable it, set `copy_view_enabled` to `False`.

The view’s form will be generated in the same way as create or edit forms. To use a custom form, override the `copy_view_class` and modify the `form_class` property on that class.

Inspect view

The inspect view 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 enable the inspect view by setting `inspect_view_enabled` on your `ModelViewSet` class.

When inspect view is enabled, an ‘Inspect’ button will automatically appear for each row on the listing view, which takes you to a view that shows a list of field values for that particular instance.

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 customize what values are displayed by specifying the `inspect_view_fields` or the `inspect_view_fields_exclude` attributes on your `ModelViewSet` class.

Templates

If `template_prefix` is set, Wagtail will look for the views’ templates in the following directories within your project or app, before resorting to the defaults:

1. `templates/{template_prefix}/{app_label}/{model_name}/`
2. `templates/{template_prefix}/{app_label}/`
3. `templates/{template_prefix}/`

To override the template used by the `IndexView` for example, you could create a new `index.html` template and put it in one of those locations. For example, given `custom/campaign` as the `template_prefix` and a `Shirt` model in a `merch` app, you could add your custom template as `templates/custom/campaign/merch/shirt/index.html`.

For some common views, Wagtail also allows you to override the template used by overriding the `{view_name}_template_name` property on the viewset. The following is a list of customization points for the views:

- `IndexView: index.html` or `index_template_name`
 - For the results fragment used in AJAX responses (e.g. when searching), customize `index_results.html` or `index_results_template_name`

- CreateView: `create.html` or `create_template_name`
- EditView: `edit.html` or `edit_template_name`
- DeleteView: `delete.html` or `delete_template_name`
- HistoryView: `history.html` or `history_template_name`
- InspectView: `inspect.html` or `inspect_template_name`

Other customizations

By default, the model registered with a `ModelViewSet` will also be registered to the `reference index`. You can turn off this behavior by setting `add_to_reference_index` to `False`.

Various additional attributes are available to customize the viewset - see the `ModelViewSet` documentation.

ChooserViewSet

The `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(
    name="PersonChooserBlock", module_path="myapp.blocks"
)
```

Limits choices via linked fields

Chooser viewsets provide a mechanism for limiting the options displayed in the chooser according to another input field on the calling page. For example, suppose the `person` model has a `country` field - we can then set up a `page` model with a country dropdown and a person chooser, where an editor first selects a country from the dropdown and then opens the person chooser to be presented with a list of people from that country.

To set this up, define a `url_filter_parameters` attribute on the `ChooserViewSet`. This specifies a list of URL parameters that will be recognized for filtering the results - whenever these are passed in the URL, a `filter` clause on the correspondingly-named field will be applied to the queryset. These parameters should also be listed in the `preserve_url_parameters` attribute, so that they are preserved in the URL when navigating through the chooser (such as when following pagination links). The following definition will allow the person chooser to be filtered by country:

```
class PersonChooserViewSet(ChooserViewSet):
    model = "myapp.Person"
    url_filter_parameters = ["country"]
    preserve_url_parameters = ["multiple", "country"]
```

The chooser widget now needs to be configured to pass these URL parameters when opening the modal. This is done by passing a `linked_fields` dictionary to the widget's constructor, where the keys are the names of the URL parameters to be passed, and the values are CSS selectors for the corresponding input fields on the calling page. For example, suppose we have a `page` model with a country dropdown and a person chooser:

```
class BlogPage(Page):
    country = models.ForeignKey(Country, null=True, blank=True, on_delete=models.SET_NULL)
    author = models.ForeignKey(Person, null=True, blank=True, on_delete=models.SET_NULL)

    content_panels = Page.content_panels + [
        FieldPanel('country'),
        FieldPanel('author', widget=PersonChooserWidget(linked_fields={
            # pass the country selected in the id_country input to the person chooser
            # as a URL parameter 'country'
            'country': '#id_country',
        })),
    ]
```

A number of other lookup mechanisms are available:

```
PersonChooserWidget(linked_fields={
    'country': {'selector': '#id_country'} # equivalent to 'country': '#id_country'
})

# Look up by ID
PersonChooserWidget(linked_fields={
    'country': {'id': 'id_country'}
})

# Regexp match, for use in StreamFields and InlinePanels where IDs are dynamic:
# 1) Match the ID of the current widget's form element (the PersonChooserWidget)
#     against the regexp '^id_blog_person_relationship-\d+-'
# 2) Append 'country' to the matched substring
# 3) Retrieve the input field with that ID
PersonChooserWidget(linked_fields={
    'country': {'match': r'^id_blog_person_relationship-\d+-', 'append': 'country'}
})
```

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 through the use of the `queryish` library, which allows any data source to be wrapped in a Django QuerySet-like interface. This can then be passed to ChooserViewSet like a normal model. For example, the Pokemon example from the `queryish` documentation could be made into a chooser as follows:

```
# views.py

import re
from queryish.rest import APIModel
from wagtail.admin.viewsets.chooser import ChooserViewSet

class Pokemon(APIModel):
    class Meta:
        base_url = "https://pokeapi.co/api/v2/pokemon/"
        detail_url = "https://pokeapi.co/api/v2/pokemon/%s/"
        fields = ["id", "name"]
        pagination_style = "offset-limit"
        verbose_name_plural = "pokemon"

    @classmethod
    def from_query_data(cls, data):
        return cls(
            id=int(re.match(r'https://pokeapi.co/api/v2/pokemon/(\d+)/', data['url']).group(1)),
            name=data['name'],
        )

    @classmethod
    def from_individual_data(cls, data):
        return cls(
            id=data['id'],
            name=data['name'],
        )
```

(continues on next page)

(continued from previous page)

```

def __str__(self):
    return self.name

class PokemonChooserViewSet(ChooserViewSet):
    model = Pokemon

    choose_one_text = "Choose a pokemon"
    choose_another_text = "Choose another pokemon"

pokemon_chooser_viewset = PokemonChooserViewSet("pokemon_chooser")

# wagtail_hooks.py

from wagtail import hooks

from .views import pokemon_chooser_viewset

@hooks.register("register_admin_viewset")
def register_pokemon_chooser_viewset():
    return pokemon_chooser_viewset

```

1.5.3 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).

Note

Starting with version 6.0, Wagtail uses the [Laces](#) library to provide all the component related implementations.

The Laces library was extracted from Wagtail to make the concept of “template components” available to the wider Django ecosystem.

All import paths shown below continue to work, but they are only references to the implementations in Laces.

“Template components” are not restricted to extensions of the Wagtail admin. You can use the concepts and tools below in your user-facing code as well.

You can find more information on the use of components in the [Laces documentation](#).

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 %}
```

You can pass additional context variables to the component using the keyword `with`:

```
{% component panel with username=request.user.username %}
```

To render the component with only the variables provided (and no others from the calling template's context), use `only`:

```
{% component panel with username=request.user.username only %}
```

To store the component's rendered output in a variable rather than outputting it immediately, use `as` followed by the variable name:

```
{% component panel as panel_html %}

{{ panel_html }}
```

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(),
    ]

    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 %}
```

1.5.4 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
```

(continues on next page)

(continued from previous page)

```
class FeaturedImage(models.Model):
    date = models.DateField()
    image = models.ForeignKey(Image, on_delete=models.CASCADE)

class FeaturedImageForm(WagtailAdminModelForm):
    class Meta:
        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 and snippets 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 [site settings](#) contrib module.

See [Panels](#) 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, snippets can define their panels via the `SnippetViewSet` class.
- If the `PanelsGroups` permissions do not allow a user to see this panel, then nothing more will be done.
 - This can be modified using the `permission` keyword argument, see examples of this usage in [Customizing the tabbed interface](#) and [Permissions](#).
- 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 initialization 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](#).

1.5.5 Adding reports

Reports are views with listings of pages or any non-page model (such as snippets) matching a specific query. Reports 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 with two base classes provided:

- `wagtail.admin.views.reports.ReportView` - Provides the basic listing (with a single column) and spreadsheet export functionality.
- `wagtail.admin.views.reports.PageReportView` - Extends the `ReportView` and provides a default set of fields suitable for page listings.

Reporting reference

`get_queryset`

The most important attributes and methods to customize to define your report are:

`get_queryset(self)`

This retrieves the queryset of pages or other models for your report, two examples below.

```
# <project>/views.py

from wagtail.admin.views.reports import ReportView, PageReportView
from wagtail.models import Page

from .models import MySnippetModel


class UnpublishedChangesReportView(PageReportView):
    # includes common page fields by default

    def get_queryset(self):
        return Page.objects.filter(has_unpublished_changes=True)


class CustomModelReport(ReportView):
    # includes string representation as a single column only

    def get_queryset(self):
        return MySnippetModel.objects.all()
```

Other attributes

`template_name`

(string)

The template used to render your report view, defaults to "wagtailadmin/reports/base_report.html". Note that this template only provides the skeleton of the view, not the listing table itself. The listing table should be implemented in a separate template specified by `results_template_name` (see below), to then be rendered via `{% include %}`. Unless you want to customize the overall view, you will rarely need to change this template. To customize the listing, change the `results_template_name` instead.

`results_template_name`

(string)

The template used to render the listing table. For `ReportView`, this defaults to "wagtailadmin/reports/base_report_results.html", which provides support for using the `wagtail.admin.ui.tables` framework. For `PageReportView`, this defaults to "wagtailadmin/reports/base_page_report_results.html", which provides a default table layout 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.

`page_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'.

index_url_name

(string)

The name of the URL pattern registered for the report view.

index_results_url_name

(string)

The name of the URL pattern registered for the results view (the report view with `.as_view(results_only=True)`).

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`.

Example report for pages with unpublished changes

For this example, we'll add a report which shows any pages with unpublished changes. We will register this view using the `unpublished_changes_report` name for the URL pattern.

```
# <project>/views.py
from wagtail.admin.views.reports import PageReportView

class UnpublishedChangesReportView(PageReportView):
    index_url_name = "unpublished_changes_report"
    index_results_url_name = "unpublished_changes_report_results"
```

Customizing 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_results.html`, and `wagtailadmin/reports/listing/_list_page_report.html`.

```
{# <project>/templates/reports/unpublished_changes_report_results.html #}

{% extends 'wagtailadmin/reports/base_page_report_results.html' %}

{% block results %}
    {% include 'reports/include/_list_unpublished_changes.html' %}
{% endblock %}

{% block no_results_message %}
    <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.results_template_name` to this new template: `'reports/unpublished_changes_report_results.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'), icon_name=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'),
        # Add a results-only view to add support for AJAX-based filtering
        path('reports/unpublished-changes/results/', UnpublishedChangesReportView.as_-
→view(results_only=True), name='unpublished_changes_report_results'),
    ]
```

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`.

Setting up permission restriction

Even with the menu item hidden, it would still be possible for any user to visit the report's URL directly, and so it is necessary to set up a permission restriction on the report view itself. This can be done by adding a `dispatch` method to the existing `UnpublishedChangesReportView` view:

```
# add the below dispatch method to the existing UnpublishedChangesReportView view
def dispatch(self, request, *args, **kwargs):
    if not self.request.user.is_superuser:
        return permission_denied(request)
    return super().dispatch(request, *args, **kwargs)
```

The full code

```
# <project>/views.py

from wagtail.admin.auth import permission_denied
from wagtail.admin.views.reports import PageReportView
from wagtail.models import Page

class UnpublishedChangesReportView(PageReportView):
    index_url_name = "unpublished_changes_report"
    index_results_url_name = "unpublished_changes_report_results"
    header_icon = 'doc-empty-inverse'
    results_template_name = 'reports/unpublished_changes_report_results.html'
    page_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)

    def dispatch(self, request, *args, **kwargs):
        if not self.request.user.is_superuser:
            return permission_denied(request)
        return super().dispatch(request, *args, **kwargs)
```

```
# <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'), icon_name=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'),
        path('reports/unpublished-changes/results/', UnpublishedChangesReportView.as_.
        ↪view(results_only=True), name='unpublished_changes_report_results'),
    ]
```

```
{# <project>/templates/reports/unpublished_changes_report_results.html #}

{% extends 'wagtailadmin/reports/base_page_report_results.html' %}

{% block results %}
    {% include 'reports/include/_list_unpublished_changes.html' %}
```

(continues on next page)

(continued from previous page)

```
{% endblock %}

{% block no_results_message %}
    <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 %}
```

1.5.6 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 implement your own task types. Instances of your custom task can then be created in the Workflow tasks section of the Wagtail Admin.

Task models

All custom tasks must be models inheriting from `wagtailcore.Task`.

If you need to customize the behavior of the built-in `GroupApprovalTask`, create a custom task which inherits from `AbstractGroupApprovalTask` and add your customizations there. See below for more details on how to customize behavior.

In this set of examples, we'll set up a task that 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
from wagtail.models import Task

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']
```

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,
                           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 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,
                           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 an object 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
```

Customizing behavior

Both `Task` and `TaskState` have a number of methods that can be overridden to implement custom behavior. Here are some of the most useful:

```
Task.user_can_access_editor(obj, user), Task.user_can_lock(obj, user), Task.user_can_unlock(obj, user):
```

These methods determine if users usually without permission can access the editor, and lock, or unlock the object, 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, obj, user):
    return user == self.user
```

```
Task.locked_for_user(obj, user):
```

This returns `True` if the object should be locked and uneditable by the user. It is used by `GroupApprovalTask` to lock the object to any users not in the approval group.

```
def locked_for_user(self, obj, user):
    return user != self.user
```

Task.get_actions(obj, 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, obj, 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(obj, 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: canceling 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) that the given user can moderate - this is currently used to select objects 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 canceled and then resumed -
            # when object 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['object'] = task_state.workflow_state.content_object
        context['task'] = task_state.task.specific
        return context

    def get_recipient_users(self, task_state, **kwargs):
        # Send emails to the user assigned to the task
```

(continues on next page)

(continued from previous page)

```

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()

```

1.5.7 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 provides 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 the 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(
        instance=page, action='mypackage.custom_action', data=data
    )
```

Log actions provided by Wagtail

Action	Notes
wagtail.create	The object was created
wagtail.edit	The object was edited (for pages, saved as a draft)
wagtail.delete	The object was deleted. Will only surface in the Site History for administrators
wagtail.publish	The page was published
wagtail.publish.schedule	The draft is scheduled for publishing
wagtail.publish.scheduled	Draft published via <code>publish_scheduled</code> management command
wagtail.schedule.cancel	Draft scheduled for publishing canceled via “Cancel scheduled publish”
wagtail.unpublish	The page was unpublished
wagtail.unpublish.scheduled	Page unpublished via <code>publish_scheduled</code> management command
wagtail.lock	Page was locked
wagtail.unlock	Page was unlocked
wagtail.rename	A page was renamed
wagtail.revert	The page was reverted to a previous draft
wagtail.copy	The page was copied to a new location
wagtail.copy_for_translation	The page was copied into a new locale for translation
wagtail.move	The page was moved to a new location
wagtail.reorder	The order of the page under its parent was changed
wagtail.view_restriction.create	The page was restricted
wagtail.view_restriction.edit	The page restrictions were updated
wagtail.view_restriction.delete	The page restrictions were removed
wagtail.workflow.start	The page was submitted for moderation in a Workflow
wagtail.workflow.approve	The draft was approved at a Workflow Task
wagtail.workflow.reject	The draft was rejected, and changes were requested at a Workflow Task
wagtail.workflow.resume	The draft was resubmitted to the workflow
wagtail.workflow.cancel	The workflow was canceled

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)
```

1.5.8 Customizing the user account settings form

This document describes how to customize the user account settings form which 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 that 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 = [...]
```

```
# wagtail_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"
```

(continues on next page)

(continued from previous page)

```
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

class CustomProfileSettingsForm(forms.ModelForm):

    class Meta:
        model = UserProfile
        fields = [...]
```

```
# wagtail_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:

```
# wagtail_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

Customizing the template

You can provide a custom template for the panel by specifying a template name:

```
# wagtail_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 %}
```

1.5.9 Customizing 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 customize the behavior of the group management interface.

Custom edit/create forms

This example shows how to customize 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:

```
# myapp/models.py
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)

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:

```
# myapp/forms.py
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()
```

(continues on next page)

(continued from previous page)

```
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.

```
# myapp/viewsets.py
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 %}

{% 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:

```
# myproject/apps.py
from wagtail.users.apps import WagtailUsersAppConfig

class CustomUsersAppConfig(WagtailUsersAppConfig):
    group_viewset = "myapp.viewsets.GroupViewSet"
```

Replace `wagtail.users` in `settings.INSTALLED_APPS` with the path to `CustomUsersAppConfig`.

```
INSTALLED_APPS = [
    ...,
    "myproject.apps.CustomUsersAppConfig",
    # "wagtail.users",
    ...,
]
```

The `GroupViewSet` class is a subclass of `ModelViewSet` and thus it supports most of *the customizations available for ModelViewSet*.

The user forms and views can be customized in a similar way - see [Creating a custom UserViewSet](#).

Customizing the group editor permissions ordering

The order in which object types appear in the group editor’s “Object permissions” and “Other permissions” sections can be configured by registering that order in one or more AppConfig definitions. The order value is typically an integer between 0 and 999, although this is not enforced.

```
from django.apps import AppConfig

class MyProjectAdminAppConfig(AppConfig):
    name = "myproject_admin"
    verbose_name = "My Project Admin"

    def ready(self):
        from wagtail.users.permission_order import register

        register("gadgets.SprocketType", order=150)
        register("gadgets.ChainType", order=151)
        register("site_settings.Settings", order=160)
```

A model class can also be passed to register().

Any object types that are not explicitly given an order will be sorted in alphabetical order by app_label and model, and listed after all of the object types *with* a configured order.

1.5.10 Custom image filters

Wagtail comes with *various image operations*. To add custom image operation, add register_image_operations hook to your wagtail_hooks.py file.

In this example, the willow.image is a Pillow Image instance. If you use another image library, or like to support multiple image libraries, you need to update the filter code accordingly. See the Willow documentation for more information.

```
from PIL import ImageFilter

from wagtail import hooks
from wagtail.images.image_operations import FilterOperation

class BlurOperation(FilterOperation):
    def construct(self, radius):
        self.radius = int(radius)

    def run(self, willow, image, env):
        willow.image = willow.image.filter(ImageFilter.GaussianBlur(radius=self.
        ↪radius))
        return willow

@hooks.register("register_image_operations")
def register_image_operations():
    return [
        ("blur", BlurOperation),
    ]
```

Use the filter in a template, like so:

```
{% load wagtailimages_tags %}

{% image page.photo width-400 blur-7 %}
```

If your custom image filter depends on fields within the `Image`, for instance those defining the focal point, add a `vary_fields` property listing those field names to the subclassed `FilterOperation`. This ensures that a new rendition is created whenever the focal point is changed:

```
class BlurOutsideFocusPointOperation(FilterOperation):
    vary_fields = (
        "focal_point_width",
        "focal_point_height",
        "focal_point_x",
        "focal_point_y",
    )
    # ...
```

1.5.11 Extending client-side behavior

Many kinds of common customizations can be done without reaching into JavaScript, but depending on what parts of the client-side interaction you want to leverage or customize, you may need to employ React, Stimulus, or plain (vanilla) JS.

[React](#) is used for more complex parts of Wagtail, such as the sidebar, commenting system, and the Draftail rich-text editor. For basic JavaScript-driven interaction, Wagtail is migrating towards [Stimulus](#).

You don't need to know or use these libraries to add your custom behavior to elements, and in many cases, simple JavaScript will work fine, but Stimulus is the recommended approach for more complex use cases.

You don't need to have Node.js tooling running for your custom Wagtail installation for many customizations built on these libraries, but in some cases, such as building packages, it may make more complex development easier.

Note

Avoid using jQuery and undocumented jQuery plugins, as they will be removed in a future version of Wagtail.

Adding custom JavaScript

Within Wagtail's admin interface, there are a few ways to add JavaScript.

The simplest way is to add global JavaScript files via hooks, see [`insert_editor_js`](#) and [`insert_global_admin_js`](#).

For JavaScript added when a specific Widget is used you can add an inner `Media` class to ensure that the file is loaded when the widget is used, see [Django's docs on their form Media class](#).

In a similar way, Wagtail's [Template components](#) provide a `media` property or `Media` class to add scripts when rendered.

These will ensure the added files are used in the admin after the core JavaScript admin files are already loaded.

Extending with DOM events

When approaching client-side customizations or adopting new components, try to keep the implementation simple first, you may not need any knowledge of Stimulus, React, JavaScript Modules, or a build system to achieve your goals.

The simplest way to attach behavior to the browser is via [DOM Events](#) and plain (vanilla) JavaScript.

Wagtail's custom DOM events

Wagtail supports some custom behavior via listening or dispatching custom DOM events.

- See [Images title generation on upload](#).
- See [Documents title generation on upload](#).
- See [InlinePanel DOM events](#).

Extending with Stimulus

Wagtail uses [Stimulus](#) as a way to provide lightweight client-side interactivity or custom JavaScript widgets within the admin interface.

The key benefit of using Stimulus is that your code can avoid the need for manual initialization when widgets appear dynamically, such as within modals, `InlinePanel`, or `StreamField` panels.

The [Stimulus handbook](#) is the best source on how to work with and understand Stimulus.

Adding a custom Stimulus controller

Wagtail exposes two client-side globals for using Stimulus.

1. `window.wagtail.app` the core admin Stimulus application instance.
2. `window.StimulusModule` Stimulus module as exported from `@hotwired/stimulus`.

First, create a custom [Stimulus controller](#) that extends the base `window.StimulusModule.Controller` using [JavaScript class inheritance](#). If you are using a build tool you can import your base controller via `import { Controller } from '@hotwired/stimulus';`.

Once you have created your custom controller, you will need to register your Stimulus controllers manually via the `window.wagtail.app.register` method.

A simple controller example

First, create your HTML so that appears somewhere within the Wagtail admin.

```
<!-- Will log 'My controller has connected: hi' to the console -->
<div data-controller="my-controller">Hi</div>
<!-- Will log 'My controller has connected: hello' to the console, with the span
element-->
<div data-controller="my-controller">
    Hello <span data-my-controller-target="label"></span>
</div>
```

Second, create a JavaScript file that will contain your controller code. This controller logs a simple message on connect, which is once the controller has been created and connected to an HTML element with the matching data-controller attribute.

```
// myapp/static/js/example.js

class MyController extends window.StimulusModule.Controller {
    static targets = ['label'];
    connect() {
        console.log(
            'My controller has connected:',
            this.element.innerText,
            this.labelTargets,
        );
    }
}

window.wagtail.app.register('my-controller', MyController);
```

Finally, load the JavaScript file into Wagtail's admin with a hook.

```
# myapp/wagtail_hooks.py
from django.templatetags.static import static
from django.utils.html import format_html

from wagtail import hooks

@hooks.register('insert_global_admin_js')
def global_admin_js():
    return format_html(
        f'<script src="{static("js/example.js")}"></script>',
    )
```

You should now be able to refresh your admin that was showing the HTML and see two logs in the console.

A more complex controller example

Now we will create a WordCountController that adds a small output element next to the controlled input element that shows a count of how many words have been entered.

```
// myapp/static/js/word-count-controller.js
class WordCountController extends window.StimulusModule.Controller {
    static values = { max: { default: 10, type: Number } };

    connect() {
        this.setupOutput();
        this.updateCount();
    }

    setupOutput() {
        if (this.output) return;
        const template = document.createElement('template');
        template.innerHTML = `<output name='word-count' for='${this.element.id}' class='output-label'></output>`;
        const output = template.content.firstChild;
        this.element.insertAdjacentElement('beforebegin', output);
    }
}
```

(continues on next page)

(continued from previous page)

```

        this.output = output;
    }

    updateCount(event) {
        const value = event ? event.target.value : this.element.value;
        const words = (value || '').split(' ');
        this.output.textContent = `${words.length} / ${this maxValue} words`;
    }

    disconnect() {
        this.output && this.output.remove();
    }
}
window.wagtail.app.register('word-count', WordCountController);

```

This lets the data attribute `data-word-count-max-value` determine the ‘configuration’ of this controller and the data attribute actions to determine the ‘triggers’ for the updates to the output element.

```

# models.py
from django import forms

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


class BlogPage(Page):
    # ...
    content_panels = Page.content_panels + [
        FieldPanel('subtitle', classname="full"),
        FieldPanel(
            'introduction',
            classname="full",
            widget=forms.TextInput(
                attrs={
                    'data-controller': 'word-count',
                    # allow the max number to be determined with attributes
                    # note we can use Python values here, Django will handle the
                    # string conversion (including escaping if applicable)
                    'data-word-count-max-value': 5,
                    # decide when you want the count to update with data-action
                    # (e.g. 'blur->word-count#updateCount' will only update when
                    # field loses focus)
                    'data-action': 'word-count#updateCount paste->word-count
                    #updateCount',
                }
            ),
        ),
    #...

```

This next code snippet shows a more advanced version of the `insert_editor_js` hook usage which is set up to append additional scripts for future controllers.

```

# wagtail_hooks.py
from django.utils.html import format_html_join
from django.templatetags.static import static

```

(continues on next page)

(continued from previous page)

```
from wagtail import hooks

@hooks.register('insert_editor_js')
def editor_js():
    # add more controller code as needed
    js_files = ['js/word-count-controller.js',]
    return format_html_join('\n', '<script src="{0}"></script>',
        ((static(filename),) for filename in js_files)
    )
```

You should be able to see that on your Blog Pages, the introduction field will now have a small `output` element showing the count and max words being used.

A more complex widget example

For more complex widgets we can now integrate additional libraries whenever the widget appears in the rendered HTML, either on initial load or dynamically without the need for any inline `script` elements.

In this example, we will build a color picker widget using the `Coloris` JavaScript library with support for custom widget options.

First, let's start with the HTML, building on the `Django widgets` system that Wagtail supports for `FieldPanel` and `FieldBlock`. Using the `build_attrs` method, we build up the appropriate Stimulus data attributes to support common data structures being passed into the controller.

Observe that we are using `json.dumps` for complex values (a list of strings in this case), Django will automatically escape these values when rendered to avoid common causes of insecure client-side code.

```
# myapp/widgets.py
import json

from django.forms import Media, TextInput

from django.utils.translation import gettext as _

class ColorWidget(TextInput):
    """
    See https://coloris.js.org/
    """

    def __init__(self, attrs=None, swatches=[], theme='large'):
        self.swatches = swatches
        self.theme = theme
        super().__init__(attrs=attrs);

    def build_attrs(self, *args, **kwargs):
        attrs = super().build_attrs(*args, **kwargs)
        attrs['data-controller'] = 'color'
        attrs['data-color-theme-value'] = self.theme
        attrs['data-color-swatches-value'] = json.dumps(swatches)
        return attrs

    @property
    def media(self):
        return Media(
```

(continues on next page)

(continued from previous page)

```

js=[  
    # load the UI library  
    "https://cdn.jsdelivr.net/gh/mdbassit/Coloris@latest/dist/coloris.min.  
→js",  
    # load controller JS  
    "js/color-controller.js",  
,  
    css={"all": ["https://cdn.jsdelivr.net/gh/mdbassit/Coloris@latest/dist/  
→coloris.min.css"]},  
)

```

For the Stimulus controller, we pass the values through to the JavaScript library, including a reference to the controlled element via `this.element.id`.

```

// myapp/static/js/color-controller.js

class ColorController extends window.StimulusModule.Controller {  
    static values = { swatches: Array, theme: String };  
  
    connect() {  
        // create  
        Coloris({ el: `#${this.element.id}` });  
  
        // set options after initial creation  
        setTimeout(() => {  
            Coloris({ swatches: this.swatchesValue, theme: this.themeValue });  
        });
    }
}  
  
window.wagtail.app.register('color', ColorController);

```

Now we can use this widget in any FieldPanel or any FieldBlock for StreamFields, it will automatically instantiate the JavaScript to the field's element.

```

# blocks.py  
  
# ... other imports  
from django import forms  
from wagtail.blocks import FieldBlock  
  
from .widgets import ColorWidget  
  
class ColorBlock(FieldBlock):  
    def __init__(self, *args, **kwargs):  
        swatches = kwargs.pop('swatches', [])  
        theme = kwargs.pop('theme', 'large')  
        self.field = forms.CharField(widget=ColorWidget(swatches=swatches,  
→theme=theme))  
        super().__init__(*args, **kwargs)

```

```

# models.py  
  
# ... other imports  
from django import forms

```

(continues on next page)

(continued from previous page)

```
from wagtail.admin.panels import FieldPanel

from .blocks import ColorBlock
from .widgets import ColorWidget

BREAD_COLOR_PALETTE = ["#CFAC89", "#C68C5F", "#C47647", "#98644F", "#42332E"]

class BreadPage(Page):
    body = StreamField([
        # ...
        ('color', ColorBlock(swatches=BREAD_COLOR_PALETTE)),
        # ...
    ], use_json_field=True)
    color = models.CharField(blank=True, max_length=50)

    # ... other fields

    content_panels = Page.content_panels + [
        # ... other panels
        FieldPanel("body"),
        FieldPanel("color", widget=ColorWidget(swatches=BREAD_COLOR_PALETTE)),
    ]
```

Using a build system

You will need ensure your build output is ES6/ES2015 or higher. You can use the exposed global module at `window.StimulusModule` or provide your own using the npm module `@hotwired/stimulus`.

```
// myapp/static/js/word-count-controller.js
import { Controller } from '@hotwired/stimulus';

class WordCountController extends Controller {
    // ... the same as above
}

window.wagtail.app.register('word-count', WordCountController);
```

You may want to avoid bundling Stimulus with your JavaScript output and treat the global as an external/alias module, refer to your build system documentation for instructions on how to do this.

Extending with React

To customize or extend the `React` 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'
window.ReactDOM;
```

(continues on next page)

(continued from previous page)

```
// '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;
```

Extending Draftail

- *Extending the Draftail editor*

Extending StreamField

- *Form widget client-side API*
- *Additional JavaScript on StructBlock forms*

1.5.12 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')
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)

Optional. 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 ``.

Either this method or `expand_db_attributes_many` must be defined in a custom rewrite handler.

expand_db_attributes_many (attrs_list)

Optional. The `expand_db_attributes_many` method works similarly to `expand_db_attributes` but instead takes a list of attribute dictionaries and returns a list of HTML tags. This method is used by rewrite handlers to work in bulk, for example leveraging the ability to make one database query instead of multiple.

Either this method or `expand_db_attributes` must be defined in a custom rewrite handler. If not defined, the default implementation of `expand_db_attributes_many` works by making a series of calls to `expand_db_attributes`.

`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 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.

This method should raise an exception if the provided attributes cannot be used to retrieve a Django model instance, for example if the provided `id` attribute is invalid.

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.

`get_many(attrs_list)`

Optional. The classmethod `get_many` method works similarly to `get_instance` but instead takes a list of attribute dictionaries and returns a list of Django model instances.

Any instances that cannot be retrieved will be represented by `None` in the returned list.

Below is an example custom rewrite handler that implements some of 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):
    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 recognized 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 recognize, and adapt its behavior 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 recognize.

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 standardized way for `register_rich_text_features` hooks to define plugins to be pulled into 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 behavior 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.

1.5.13 Extending the Draftail editor

Wagtail's rich text editor is built with [Draftail](#), which supports different types of extensions.

Formatting extensions

Draftail supports three types of formatting:

- **Inline styles** – To format a portion of a line, for example `bold`, `italic` or `monospace`. Text can have as many inline styles as needed – for example `bold` and `italic` at the same time.
- **Blocks** – To indicate the structure of the content, for example, `blockquote`, `ol`. Any given text can only be of one block type.
- **Entities** – To enter additional data/metadata, for example, `link` (with a URL) or `image` (with a file). Text can only have one entity applied at a time.

All of these extensions 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 ...
from wagtail.admin.rich_text.editors.draftail.features import InlineStyleElementHandler
from wagtail import hooks

# 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 = {
```

(continues on next page)

(continued from previous page)

```
'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`.
- `icon` is an icon name *registered in the Wagtail icon library* - for example, `'icon': '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
from wagtail import hooks

@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 `<div class="help-text">` tag.

```

(continues on next page)

(continued from previous page)

```

"""
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.

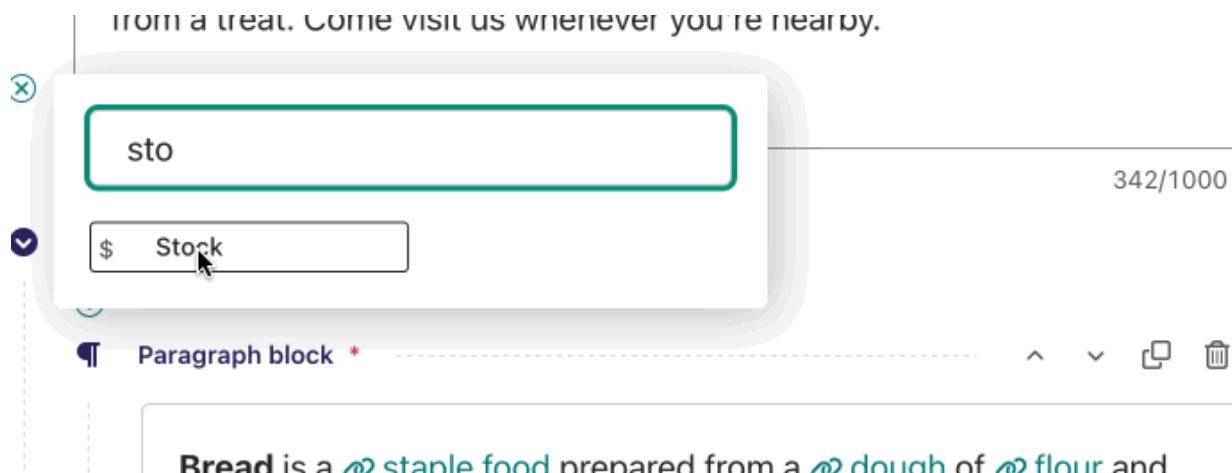
Here are the main requirements to create a new entity feature:

- As 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 serialized 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 client-side React 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 “\$NEE” 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 = {
```

(continues on next page)

(continued from previous page)

```
'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 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):
    """
    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:

```
// Not a real React component - just creates the entities as soon as it is rendered.
class StockSource extends window.React.Component {
    componentDidMount() {
        const { editorState, entityType, onComplete } = this.props;

        const content = editorState.getCurrentContent();
        const selection = editorState.getSelection();

        const demoStocks = ['AMD', 'AAPL', 'NEE', 'FSLR'];
        const randomStock = demoStocks[Math.floor(Math.random() * demoStocks.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 = window.DraftJS.Modifier.replaceText(
            content,
            selection,
            text,
            null,
            entityKey,
        );
        const nextState = window.DraftJS.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 client-side React components](#).

We then create the decorator component:

```
const Stock = (props) => {
    const { entityKey, contentState } = props;
    const data = contentState.getEntity(entityKey).getData();

    return window.React.createElement(
        'a',
        {
            role: 'button',
```

(continues on next page)

(continued from previous page)

```
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:

```
// Register the plugin directly on script execution so the editor loads it when
// initializing.
window.draftail.registerPlugin({
  type: 'STOCK',
  source: StockSource,
  decorator: Stock,
}, 'entityTypes');
```

And that's it! All of this setup will finally produce the following HTML on the site's front-end:

```
<p>
  Anyone following NextEra technology <span data-stock="NEE">$NEE</span> should
  also look into <span data-stock="FSLR">$FSLR</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}<img alt="Stock price sparkline" data-stock="${elt.dataset.stock}">`;
  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.

Other editor extensions

Draftail has additional APIs for more complex customizations:

- **Controls** – To add arbitrary UI elements to editor toolbars.
- **Decorators** – For arbitrary text decorations/highlighting.
- **Plugins** – For direct access to all Draft.js APIs.

Custom toolbar controls

To add an arbitrary new UI element to editor toolbars, Draftail comes with a [controls API](#). Controls can be arbitrary React components, which can get and set the editor state. Note controls update on *every keystroke* in the editor – make sure they render fast!

Here is an example with a simple sentence counter – first, registering the editor feature in a `wagtail_hooks.py`:

```
from wagtail.admin.rich_text.editors.draftail.features import ControlFeature
from wagtail import hooks

@hooks.register('register_rich_text_features')
def register_sentences_counter(features):
    feature_name = 'sentences'
    features.default_features.append(feature_name)

    features.register_editor_plugin(
        'draftail',
        feature_name,
        ControlFeature({
            'type': feature_name,
        },
        js=['draftail_sentences.js'],
    ),
)
```

Then, `draftail_sentences.js` declares a React component that will be rendered in the “meta” bottom toolbar of the editor:

```
const countSentences = (str) =>
  str ? (str.match(/[^!.\r\n]+/g) || []).length + 1 : 0;

const SentenceCounter = ({ getEditorState }) => {
  const editorState = getEditorState();
  const content = editorState.getCurrentContent();
  const text = content.getPlainText();

  return window.React.createElement('div', {
    className: 'w-inline-block w-tabular-nums w-help-text w-mr-4',
  }, `Sentences: ${countSentences(text)}`);
}

window.draftail.registerPlugin({
  type: 'sentences',
  meta: SentenceCounter,
}, 'controls');
```

Note

Remember to include this feature in any custom Draft configs set up in the WAGTAILADMIN_RICH_TEXT_EDITORS setting. So that this new ‘sentences’ feature is available.

For example:

```
WAGTAILADMIN_RICH_TEXT_EDITORS = {
    'default': {
        'WIDGET': 'wagtail.admin.rich_text.DraftailRichTextArea',
        'OPTIONS': {
            'features': ['bold', 'italic', 'link', 'sentences'], # Add 'sentences' here
        },
    },
}
```

Text decorators

The decorators API is how Draftail / Draft.js supports highlighting text with special formatting in the editor. It uses the CompositeDecorator API, with each entry having a strategy function to determine what text to target, and a component function to render the decoration.

There are two important considerations when using this API:

- Order matters: only one decorator can render per character in the editor. This includes any entities that are rendered as decorations.
- For performance reasons, Draft.js only re-renders decorators that are on the currently focused line of text.

Here is an example with highlighting of problematic punctuation – first, registering the editor feature in a wagtail_hooks.py:

```
from wagtail.admin.rich_text.editors.draftail.features import DecoratorFeature
from wagtail import hooks


@hooks.register('register_rich_text_features')
def register_punctuation_highlighter(features):
    feature_name = 'punctuation'
    features.default_features.append(feature_name)

    features.register_editor_plugin(
        'draftail',
        feature_name,
        DecoratorFeature({
            'type': feature_name,
        },
        js=['draftail_punctuation.js'],
    ),
)
```

Then, draftail_punctuation.js defines the strategy and the highlighting component:

```
const PUNCTUATION = /(\.\.\.\|!|\?)/g;
```

(continues on next page)

(continued from previous page)

```

const punctuationStrategy = (block, callback) => {
  const text = block.getText();
  let matches;
  while ((matches = PUNCTUATION.exec(text)) !== null) {
    callback(matches.index, matches.index + matches[0].length);
  }
};

const errorHighlight = {
  color: 'var(--w-color-text-error)',
  outline: '1px solid currentColor',
}

const PunctuationHighlighter = ({ children }) => (
  window.React.createElement('span', { style: errorHighlight, title: 'refer to our
→ styleguide' }, children)
);

window.draftail.registerPlugin({
  type: 'punctuation',
  strategy: punctuationStrategy,
  component: PunctuationHighlighter,
}, 'decorators');

```

Arbitrary plugins



This is an advanced feature. Please carefully consider whether you really need this.

Draftail supports plugins following the [Draft.js Plugins](#) architecture. Such plugins are the most advanced and powerful type of extension for the editor, offering customization capabilities equal to what would be possible with a custom Draft.js editor.

A common scenario where this API can help is to add bespoke copy-paste processing. Here is a simple example, automatically converting URL anchor hash references to links. First, let's register the extension in Python:

```

@hooks.register('register_rich_text_features')
def register_anchorify(features):
  feature_name = 'anchorify'
  features.default_features.append(feature_name)

  features.register_editor_plugin(
    'draftail',
    feature_name,
    PluginFeature({
      'type': feature_name,
    },
    js=['draftail_anchorify.js'],
  ),
)

```

Then, in `draftail_anchorify.js`:

```
const anchorifyPlugin = {
    type: 'anchorify',

    handlePastedText(text, html, editorState, { setEditorState }) {
        let nextState = editorState;

        if (text.match(/^#[a-zA-Z0-9_-]+$/ig)) {
            const selection = nextState.getSelection();
            let content = nextState.getCurrentContent();
            content = content.createEntity("LINK", "MUTABLE", { url: text });
            const entityKey = content.getLastCreatedEntityKey();

            if (selection.isCollapsed()) {
                content = window.DraftJS.Modifier.insertText(
                    content,
                    selection,
                    text,
                    undefined,
                    entityKey,
                )
                nextState = window.DraftJS.EditorState.push(
                    nextState,
                    content,
                    "insert-fragment",
                );
            } else {
                nextState = window.DraftJS.RichUtils.toggleLink(nextState, selection, entityKey);
            }
            setEditorState(nextState);
            return "handled";
        }

        return "not-handled";
    },
};

window.draftail.registerPlugin(anchorifyPlugin, 'plugins');
```

Integration of the Draftail widgets

To further customize 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 that contains the data, and `[data-draftail-editor-wrapper]` for the element that wraps the editor.
- The `editor` instance is bound to the `input` field for imperative access. Use `document.querySelector('[data-draftail-input]').draftailEditor`.
- In CSS, use the classes prefixed with `Draftail-`.

1.5.14 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
    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 (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 class names that will be used on the button in the user interface

An example of 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 %}{% blocktranslate trimmed count counter=items|length %}Delete 1
→item{{ plural }}Delete {{ counter }} items{% endblocktranslate %}{% 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 %}
```

(continues on next page)

(continued from previous page)

```

{%
    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="noreferrer">{{ item.item.title }}</a>
                    </li>
                {%
                    endfor %
                </ul>
            {%
                endif %
            %}
        {% endblock items_with_access %}
    {%
        block items_with_no_access %

            {% blocktranslate trimmed asvar 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{%
            endblocktranslate %

            {% 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_

```

(continues on next page)

(continued from previous page)

```
→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 customized 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.bulk_actions.snippet_bulk_action.SnippetBulkAction` instead of `wagtail.admin.views.bulk_action.BulkAction`

Basic example

```
from wagtail.snippets.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]
    # ...
```

1.6 Reference

1.6.1 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 [Page](#), and all of your page-level content will inherit basic webpage-related properties from it. But for the most part, you will be defining content yourself, through the construction of Django models using Wagtail's `Page` as a base.

Wagtail organizes content created from your models in a tree, which can have any structure and combination of model objects in it. Wagtail doesn't prescribe ways to organize and interrelate your content, but here we've sketched out some strategies for organizing your models.

The presentation of your content, the actual webpages, includes the normal use of the Django template system. We'll cover additional functionality that Wagtail provides at the template level later on.

Theory

Introduction to trees

If you're unfamiliar with trees as an abstract data type, you might want to [review the concepts involved](#).

As a web developer, though, you probably already have a good understanding of trees as filesystem directories or paths. Wagtail pages can create the same structure, as each page in the tree has its own URL path, like so:

```
/  
  people/  
    nien-nunb/  
    laura-roslin/  
  events/  
    captain-picard-day/  
    winter-wrap-up/
```

The Wagtail admin interface uses the tree to organize content for editing, letting you navigate up and down levels in the tree through its Explorer menu. This method of organization is a good place to start in thinking about your own Wagtail models.

Nodes and leaves

It might be handy to think of the `Page`-derived models you want to create as being one of two node types: parents and leaves. Wagtail isn't prescriptive in this approach, but it's a good place to start if you're not experienced in structuring your own content types.

Nodes

Parent nodes on the Wagtail tree probably want to organize and display a browseable 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 the 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 that make sense, specifically ones that 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 consumable page, and might just consist of a bunch of properties. A blog page leaf might have some body text and an image. A person's 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 the 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 that 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 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/kwarg`s to be passed to `serve()`.
5. Wagtail calls `serve()`, which constructs a context using `get_context()`
6. `serve()` finds a template to pass it to using `get_template()`
7. A response object is returned by `serve()` and Django responds to the requester.

You can apply custom behavior to this process by overriding `Page` class methods such as `route()` and `serve()` in your own models. For examples, see [Recipes](#).

Scheduled publishing

Page publishing can be scheduled through the *Set schedule* feature in the *Status* side panel of the *Edit* page. This allows you to set up initial page publishing or a page update in advance. For pages to go live at the scheduled time, you should set up the `publish_scheduled` management command.

Basic workflow for scheduled publishing

- Scheduling is done by setting the *go-live at* field of the page and clicking *Publish*.
- 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 the revision will be published when the time comes.
- If the page has a scheduled revision and you set another revision to publish immediately (i.e. clicking *Publish* with the *go-live at* field unset), the scheduled revision will be unscheduled.
- If the page has a scheduled revision and you schedule another revision to publish (i.e. clicking *Publish* with the *go-live at* field set), the existing scheduled revision will be unscheduled and the new revision will be scheduled instead.

Note

You must click *Publish* after setting the *go-live at* field for the revision to be scheduled. Saving a draft revision with the *go-live at* field without clicking *Publish* will not schedule it to be published.

Viewing and managing scheduled revisions

The *History* view for a given page will show which revision is scheduled and when it is scheduled. A scheduled revision in the list will also provide an *Unschedule* button to cancel it.

Scheduled unpublishing

In addition to scheduling a page to be published, it is also possible to schedule a page to be unpublished by setting the *expire at* field. However, unlike with publishing, the unpublishing schedule is applied to the live page instance rather than a specific revision. This means that any change to the *expire at* field will only be effective once the associated revision is published (i.e. when the changes are applied to the live instance). To illustrate:

Basic workflow for scheduled unpublishing

- Scheduling is done by setting the *expire at* field of the page and clicking *Publish*. If the *go-live at* field is also set, then the unpublishing schedule will only be applied after the revision goes live.
- Consider a live page that is scheduled to be unpublished on e.g. 14 June. Then sometime before the schedule, consider that a new revision is scheduled to be published on a date that's **earlier** than the unpublishing schedule, e.g. 9 June. When the new revision goes live on 9 June, the *expire at* field contained in the new revision will replace the existing unpublishing schedule. This means:
 - If the new revision contains a different *expire at* field (e.g. 17 June), the new revision will go live on 9 June and the page will not be unpublished on 14 June but will be unpublished on 17 June.

- If the new revision has the *expire at* field unset, the new revision will go live on 9 June and the unpublishing schedule will be unset, thus the page will not be unpublished.
- Consider another live page that is scheduled to be unpublished on e.g. 14 June. Then sometime before the schedule, consider that a new revision is scheduled to be published on a date that's *later* than the unpublishing schedule, e.g. 21 June. The new revision will not take effect until it goes live on 21 June, so the page will still be unpublished on 14 June. This means:
 - If the new revision contains a different *expire at* field (e.g. 25 June), the page will be unpublished on 14 June, the new revision will go live on 21 June and the page will be unpublished again on 25 June.
 - If the new revision has the *expire at* field unset, the page will be unpublished on 14 June and the new revision will go live on 21 June.

Note

The same scheduling mechanism also applies to snippets with `DraftStateMixin` applied. For more details, see [Saving draft changes of snippets](#).

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 that 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(...):
    ...

    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' ↴
            # keyword argument
            return RouteResult(self, kwargs={'path_components': path_components})
        else:
            if self.live:
                # tell Wagtail to call self.serve() with no further args
                return RouteResult(self)
            else:
                raise Http404

    def serve(self, path_components=[]):
        return render(request, self.template, {
            'page': 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='rariant')

```

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.

search(query, fields=None, operator=None, order_by_relevance=True, backend='default')

This runs a search query on all the items in the QuerySet

See: [Searching QuerySets](#)

Example:

```
# Search future events
results = EventPage.objects.live().filter(date__gt=timezone.now()).search(
    "Hello")
```

type(*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)
```

(continues on next page)

(continued from previous page)

```
# 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 items for the queryset, using the minimum number of queries.

When the “defer” keyword argument is set to True, only generic 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.

select_related(**fields*, *for_specific_subqueries: bool = False*)

Overrides Django's native `select_related()` to allow related objects to be fetched by the subqueries made when a specific queryset is evaluated.

When *for_specific_subqueries* is `False` (the default), the method functions exactly like the original method. However, when `True`, *fields* are **required**, and must match names of `ForeignKey` fields on all specific models that might be included in the result (which can include fields inherited from concrete

parents). Unlike when `for_specific_subqueries` is `False`, no validation is applied to fields when the method is called. Rather, that when the method is called. Instead, that validation is applied for each individual subquery when the queryset is evaluated. This difference in behaviour should be taken into account when experimenting with `for_specific_subqueries=True`.

As with Django's native implementation, you chain multiple applications of `select_related()` with `for_specific_subqueries=True` to progressively add to the list of fields to be fetched. For example:

```
# Fetch 'author' when retrieving specific page data
queryset = Page.objects.specific().select_related("author", for_specific_
    ↴subqueries=True)

# We're rendering cards with images, so fetch the listing image too
queryset = queryset.select_related("listing_image", for_specific_
    ↴subqueries=True)

# Fetch some key taxonomy data too
queryset = queryset.select_related("topic", "target_audience", for_specific_
    ↴subqueries=True)
```

As with Django's native implementation, `None` can be supplied in place of fields to negate a previous application of `select_related()`. By default, this will only work for cases where `select_related()` was called without `for_specific_subqueries`, or with `for_specific_subqueries=False`. However, you can use `for_specific_subqueries=True` to negate subquery-specific applications too. For example:

```
# Fetch 'author' and 'listing_image' when retrieving specific page data
queryset = Page.objects.specific().select_related(
    "author",
    "listing_image",
    for_specific_subqueries=True
)

# I've changed my mind. Do not fetch any additional data
queryset = queryset.select_related(None, for_specific_subqueries=True)
```

`prefetch_related(*lookups, for_specific_subqueries: bool = False)`

Overrides Django's native `prefetch_related()` implementation to allow related objects to be fetched alongside the subqueries made when a specific queryset is evaluated.

When `for_specific_subqueries` is `False` (the default), the method functions exactly like the original method. However, when `True`, lookups are **required**, and must match names of related fields on all specific models that might be included in the result (which can include relationships inherited from concrete parents). Unlike when `for_specific_subqueries` is `False`, no validation is applied to lookups when the method is called. Instead, that validation is applied for each individual subquery when the queryset is evaluated. This difference in behaviour should be taken into account when experimenting with `for_specific_subqueries=True`.

As with Django's native implementation, you chain multiple applications of `prefetch_related()` with `for_specific_subqueries=True` to progressively add to the list of lookups to be made. For example:

```
# Fetch 'contributors' when retrieving specific page data
queryset = Page.objects.specific().prefetch_related("contributors", for_
    ↴specific_subqueries=True)

# We're rendering cards with images, so prefetch listing image renditions too
```

(continues on next page)

(continued from previous page)

```
queryset = queryset.prefetch_related("listing_image_renditions", for_
    ↪specific_subqueries=True)

# Fetch some key taxonomy data also
queryset = queryset.prefetch_related("tags", for_specific_subqueries=True)
```

As with Django's native implementation, `None` can be supplied in place of `lookups` to negate a previous application of `prefetch_related()`. By default, this will only work for cases where `prefetch_related()` was called without `for_specific_subqueries`, or with `for_specific_subqueries=False`. However, you can use `for_specific_subqueries=True` to negate subquery-specific applications too. For example:

```
# Fetch 'contributors' and 'listing_image' renditions when retrieving_
    ↪specific page data
queryset = Page.objects.specific().prefetch_related(
    "contributors",
    "listing_image_renditions",
    for_specific_subqueries=True
)

# I've changed my mind. Do not make any additional queries
queryset = queryset.prefetch_related(None, for_specific_subqueries=True)
```

Performance considerations

Typical usage of `prefetch_related()` results in an additional database query being executed for each of the provided `lookups`. However, when combined with `for_specific_subqueries=True`, this additional number of database queries is multiplied for each specific type in the result. If you are only fetching a small number of objects, or the type-variance of results is likely to be high, the additional overhead of making these additional queries could actually have a negative impact on performance.

Using `prefetch_related()` with `for_specific_subqueries=True` should be reserved for cases where a large number of results is needed, or the type-variance is restricted in some way. For example, when rendering a list of child pages where `allow_subtypes` is set on the parent, limiting the results to a small number of page types. Or, where the `type()` or `not_type()` filters have been applied to restrict the queryset to a small number of specific types.

1.6.2 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.

Note

While block definitions look similar to model fields, they are not the same thing. Blocks are only valid within a `StreamField` - using them in place of a model field will not work.

```
class wagtail.fields.StreamField(blocks, 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.
- **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.

```
body = StreamField([
    ('heading', blocks.CharBlock(form_classname="title")),
    ('paragraph', blocks.RichTextBlock()),
    ('image', ImageBlock()),
], block_counts={
    'heading': {'min_num': 1},
    'image': {'max_num': 5},
})
```

Block options and methods

All block definitions accept the following optional keyword arguments or Meta class attributes:

- **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 editor. For more details, see our [icons overview](#).
- **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.

[StreamField blocks can have previews](#) that will be shown inside the block picker. To accommodate the feature, all block definitions also accept the following options:

- **preview_value**
 - The placeholder value that will be used for rendering the preview. See [get_preview_value\(\)](#) for more details.

- `preview_template`
 - The template that is used to render the preview. See `get_preview_template()` for more details.
- `description`
 - The description of the block to be shown to editors. See `get_description()` for more details.

Added in version 6.4: The `preview_value`, `preview_template`, and `description` keyword arguments were added.

All block definitions have the following methods that can be overridden:

```
class wagtail.blocks.Block(*args, **kwargs)
```

`get_context(value, parent_context=None)`

Return a dict of context variables (derived from the block `value` and combined with the `parent_context`) to be used as the template context when rendering this value through a template. See [the usage example](#) for more details.

`get_template(value=None, context=None)`

Return the template to use for rendering the block if specified. This method allows for dynamic templates based on the block instance and a given `value`. See [the usage example](#) for more details.

`get_preview_value()`

Return the placeholder value that will be used for rendering the block's preview. By default, the value is the `preview_value` from the block's options if provided, otherwise the `default` is used as fallback. This method can be overridden to provide a dynamic preview value, such as from the database.

`get_preview_context(value, parent_context=None)`

Return a dict of context variables to be used as the template context when rendering the block's preview. The `value` argument is the value returned by `get_preview_value()`. The `parent_context` argument contains the following variables:

- `request`: The current request object.
- `block_def`: The block instance.
- `block_class`: The block class.
- `bound_block`: A `BoundBlock` instance representing the block and its value.

If [the global preview template](#) is used, the block will be rendered as the main content using `{% include_block %}`, which in turn uses `get_context()`. As a result, the context returned by this method will be available as the `parent_context` for `get_context()` when the preview is rendered.

`get_preview_template(value, context=None)`

Return the template to use for rendering the block's preview. The `value` argument is the value returned by `get_preview_value()`. The `context` argument contains the variables listed for the `parent_context` argument of `get_preview_context()` above (and not the context returned by that method itself).

Note that the preview template is used to render a complete HTML page of the preview, not just an HTML fragment for the block. The method returns the `preview_template` attribute from the block's options if provided, and falls back to [the global preview template](#) otherwise.

If the global preview template is used, the block will be rendered as the main content using `{% include_block %}`, which in turn uses `get_template()`.

`get_description()`

Return the description of the block to be shown to editors as part of the preview. For [field block types](#), it will fall back to `help_text` if not provided.

Field block types

```
class wagtail.blocks.FieldBlock(*args, **kwargs)
```

Bases: *Block*

A block that wraps a Django form field

The parent class of all StreamField field block types.

```
class wagtail.blocks.CharBlock(*args, **kwargs)
```

Bases: *FieldBlock*

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.
- **search_index** – If false (default true), the content of this block will not be indexed for searching.
- **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(*args, **kwargs)
```

Bases: *FieldBlock*

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.
- **search_index** – If false (default true), the content of this block will not be indexed for searching.
- **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(*args, **kwargs)
```

Bases: *FieldBlock*

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(*args, **kwargs)
```

Bases: [*FieldBlock*](#)

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(*args, **kwargs)
```

Bases: [*FieldBlock*](#)

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(*args, **kwargs)
```

Bases: [*FieldBlock*](#)

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(*args, **kwargs)
```

Bases: [FieldBlock](#)

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(*args, **kwargs)
```

Bases: [FieldBlock](#)

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(*args, **kwargs)
```

Bases: [FieldBlock](#)

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(*args, **kwargs)
```

Bases: *FieldBlock*

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 recognized 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(*args, **kwargs)
```

Bases: *FieldBlock*

A time picker. The following keyword arguments are accepted in addition to the standard ones:

Parameters

- **format** – Time format. This must be one of the recognized formats listed in the TIME_INPUT_FORMATS setting. If not specified Wagtail will use the WAGTAIL_TIME_FORMAT setting with fallback to "%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.DateTimeBlock(*args, **kwargs)
```

Bases: *FieldBlock*

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 recognized 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(*args, **kwargs)
```

Bases: [FieldBlock](#)

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.
- **search_index** – If false (default true), the content of this block will not be indexed for searching.
- **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(*args, **kwargs)
```

Bases: [FieldBlock](#)

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(*args, **kwargs)
```

Bases: [TextBlock](#)

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(*args, **kwargs)
```

Bases: `BaseChoiceBlock`

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.
- **search_index** – If false (default true), the content of this block will not be indexed for searching.
- **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(*args, **kwargs)
```

Bases: BaseChoiceBlock

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.
- **search_index** – If false (default true), the content of this block will not be indexed for searching.
- **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(*args, **kwargs)
```

Bases: ChooserBlock

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 specialized 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(*args, **kwargs)
```

Bases: BaseDocumentChooserBlock

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.ImageBlock(*args, **kwargs)
```

Bases: `StructBlock`

An usage of ImageChooserBlock with support for alt text. For backward compatibility, this block overrides necessary methods to change the StructValue to be an Image model instance, making it compatible in places where ImageChooserBlock was used.

An accessibility-focused control to allow the editor to select an existing image, or upload a new one. This has provision for adding alt text and indicating whether images are purely decorative, and is the Wagtail-recommended approach to uploading images. The following additional keyword argument is accepted:

Parameters

required – If true (the default), the field cannot be left blank.

`ImageBlock` incorporates backwards compatibility with `ImageChooserBlock`. A block initially defined as `ImageChooserBlock` can be directly replaced with `ImageBlock` - existing data created with `ImageChooserBlock` will be handled automatically and changed to `ImageBlock`'s data format when the field is resaved.

Added in version 6.3: The `ImageBlock` block type was added. Blocks previously defined as `ImageChooserBlock` can be directly replaced with `ImageBlock` to benefit from the alt text support, with no data migration or template changes required.

```
class wagtail.images.blocks.ImageChooserBlock(*args, **kwargs)
```

Bases: `ChooserBlock`

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(*args, **kwargs)
```

Bases: `ChooserBlock`

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(*args, **kwargs)
```

Bases: `URLBlock`

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.:param 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(*args, **kwargs)
```

Bases: *Block*

A block that just ‘exists’ and has no fields.

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 that 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 customized 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(*args, **kwargs)
```

Bases: *BaseStructBlock*

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', ImageBlock(required=False)),  
        ('biography', blocks.RichTextBlock()),  
    ], icon='user')),  
])
```

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 = ImageBlock(required=False)  
    biography = blocks.RichTextBlock()
```

(continues on next page)

(continued from previous page)

```
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', ImageBlock()),
    ('person', PersonBlock()),
])
```

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](#).
- **`search_index`** – If false (default true), the content of this block will not be indexed for searching.
- **`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 customized by setting a string here with block names contained in braces - for example `label_format = "Profile for {first_name} {surname}"`

`class wagtail.blocks.ListBlock(*args, **kwargs)`

Bases: `Block`

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"))),
])
```

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)),
    ]))),
])
```

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.
- **search_index** – If false (default true), the content of this block will not be indexed for searching.
- **collapsed** – When true, all sub-blocks are initially collapsed.

```
class wagtail.blocks.StreamBlock (*args, **kwargs)
```

Bases: BaseStreamBlock

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

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

```
class CarouselBlock(blocks.StreamBlock):
    image = ImageBlock()
    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}}),
    )
```

StreamBlock accepts the following additional options as either keyword arguments or Meta properties:

Parameters

- **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.
- **min_num** – Minimum number of sub-blocks that the stream must have.
- **max_num** – Maximum number of sub-blocks that the stream may have.
- **search_index** – If false (default true), the content of this block will not be indexed for searching.
- **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 sub-blocks are initially collapsed.
- **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')),
])
```

```
class EventPromotionsBlock(blocks.StreamBlock):
    hashtag = blocks.CharBlock()
    post_date = blocks.DateBlock()

    class Meta:
        form_classname = 'event-promotions'
```

Form widget client-side API

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 that require custom client-side behavior, you will need to provide your own implementation.

This implementation can be driven by `Stimulus` or for deeper integrations you can leverage `telepath`.

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'
```

(continues on next page)

(continued from previous page)

```
# 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']

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 initialization 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 data migration reference**wagtail.blocks.migrations.migrate_operation****MigrateStreamData**

```
class MigrateStreamData (RunPython)
```

Subclass of RunPython for StreamField data migration operations

__init__

```
def __init__(app_name,
              model_name,
              field_name,
              operations_and_block_paths,
              revisions_from=None,
              chunk_size=1024,
              **kwargs)
```

MigrateStreamData constructor

Arguments:

- *app_name str* - Name of the app.
- *model_name str* - Name of the model.
- *field_name str* - Name of the StreamField.
- *operations_and_block_paths List[Tuple[operation, str]]* - List of operations and the block paths to apply them to.
- *revisions_from datetime, optional* - Only revisions created from this date onwards will be updated. Passing None updates all revisions. Defaults to None. Note that live and latest revisions will be updated regardless of what value this takes.
- *chunk_size int, optional* - chunk size for queryset.iterator and bulk_update. Defaults to 1024.
- ***kwargs* - atomic, elidable, hints for superclass RunPython can be given

Example:

Renaming a block named `field1` to `block1`:

```
MigrateStreamData(  
    app_name="blog",  
    model_name="BlogPage",  
    field_name="content",  
    operations_and_block_paths=[  
        (RenameStreamChildrenOperation(old_name="field1", new_name="block1"), ""),  
    ],  
    revisions_from=datetime.datetime(2022, 7, 25)  
)
```

wagtail.blocks.migrations.operations

RenameStreamChildrenOperation

```
class RenameStreamChildrenOperation(BaseBlockOperation)
```

Renames all StreamBlock children of the given type

Notes:

The `block_path_str` when using this operation should point to the parent `StreamBlock` which contains the blocks to be renamed, not the block being renamed.

Attributes:

- `old_name str` - name of the child block type to be renamed
- `new_name str` - new name to rename to

RenameStructChildrenOperation

```
class RenameStructChildrenOperation(BaseBlockOperation)
```

Renames all StructBlock children of the given type

Notes:

The `block_path_str` when using this operation should point to the parent `StructBlock` which contains the blocks to be renamed, not the block being renamed.

Attributes:

- `old_name str` - name of the child block type to be renamed
- `new_name str` - new name to rename to

RemoveStreamChildrenOperation

```
class RemoveStreamChildrenOperation(BaseBlockOperation)
```

Removes all StreamBlock children of the given type

Notes:

The `block_path_str` when using this operation should point to the parent StreamBlock which contains the blocks to be removed, not the block being removed.

Attributes:

- name `str` - name of the child block type to be removed

RemoveStructChildrenOperation

```
class RemoveStructChildrenOperation(BaseBlockOperation)
```

Removes all StructBlock children of the given type

Notes:

The `block_path_str` when using this operation should point to the parent StructBlock which contains the blocks to be removed, not the block being removed.

Attributes:

- name `str` - name of the child block type to be removed

StreamChildrenToListBlockOperation

```
class StreamChildrenToListBlockOperation(BaseBlockOperation)
```

Combines StreamBlock children of the given type into a new ListBlock

Notes:

The `block_path_str` when using this operation should point to the parent StreamBlock which contains the blocks to be combined, not the child block itself.

Attributes:

- `block_name str` - name of the child block type to be combined
- `list_block_name str` - name of the new ListBlock type

StreamChildrenToStreamBlockOperation

```
class StreamChildrenToStreamBlockOperation(BaseBlockOperation)
```

Combines StreamBlock children of the given types into a new StreamBlock

Notes:

The `block_path_str` when using this operation should point to the parent StreamBlock which contains the blocks to be combined, not the child block itself.

Attributes:

- `block_names [str]` - names of the child block types to be combined
- `stream_block_name str` - name of the new StreamBlock type

AlterBlockValueOperation

```
class AlterBlockValueOperation(BaseBlockOperation)
```

Alters the value of each block to the given value

Attributes:

- `new_value: new value to change to`

StreamChildrenToStructBlockOperation

```
class StreamChildrenToStructBlockOperation(BaseBlockOperation)
```

Move each StreamBlock child of the given type inside a new StructBlock

A new StructBlock will be created as a child of the parent StreamBlock for each child block of the given type, and then that child block will be moved from the parent StreamBlocks children inside the new StructBlock as a child of that StructBlock.

Example:

Consider the following StreamField definition:

```
mystream = StreamField([("char1", CharBlock()), ...], ...)
```

Then the stream data would look like the following:

```
[  
  ...,  
  { "type": "char1", "value": "Value1", ... },  
  { "type": "char1", "value": "Value2", ... },  
  ...  
]
```

And if we define the operation like this:

```
StreamChildrenToStructBlockOperation("char1", "struct1")
```

Our altered stream data would look like this:

```
[  
    ...,  
    { "type": "struct1", "value": { "char1": "Value1" } },  
    { "type": "struct1", "value": { "char1": "Value2" } },  
    ...,  
]
```

Notes:

- The `block_path_str` when using this operation should point to the parent `StreamBlock` which contains the blocks to be combined, not the child block itself.
- Block ids are not preserved here since the new blocks are structurally different than the previous blocks.

Attributes:

- `block_names str` - names of the child block types to be combined
- `struct_block_name str` - name of the new `StructBlock` type

wagtail.blocks.migrations.utils**InvalidBlockDefError**

```
class InvalidBlockDefError(Exception)
```

Exception for invalid block definitions

map_block_value

```
def map_block_value(block_value, block_def, block_path, operation, **kwargs)
```

Maps the value of a block.

Arguments:

- `block_value`: The value of the block. This would be a list or dict of children for structural blocks.
- `block_def`: The definition of the block.
- `block_path`: A `". "` separated list of names of the blocks from the current block (not included) to the nested block of which the value will be passed to the operation.
- `operation`: An Operation class instance (extends `BaseBlockOperation`), which has an `apply` method for mapping values.

Returns:

Transformed value

map_struct_block_value

```
def map_struct_block_value(struct_block_value, block_def, block_path,
                           **kwargs)
```

Maps each child block in a `StructBlock` value.

Arguments:

- `stream_block_value`: The value of the `StructBlock`, a dict of child blocks
- `block_def`: The definition of the `StructBlock`
- `block_path`: A `"."` separated list of names of the blocks from the current block (not included) to the nested block of which the value will be passed to the operation.

Returns:

- `mapped_value`: The value of the `StructBlock` after transforming its children.

map_list_block_value

```
def map_list_block_value(list_block_value, block_def, block_path, **kwargs)
```

Maps each child block in a `ListBlock` value.

Arguments:

- `stream_block_value`: The value of the `ListBlock`, a list of child blocks
- `block_def`: The definition of the `ListBlock`
- `block_path`: A `"."` separated list of names of the blocks from the current block (not included) to the nested block of which the value will be passed to the operation.

Returns:

- `mapped_value`: The value of the `ListBlock` after transforming all the children.

apply_changes_to_raw_data

```
def apply_changes_to_raw_data(raw_data, block_path_str, operation, streamfield,
                               **kwargs)
```

Applies changes to raw stream data

Arguments:

- `raw_data`: The current stream data (a list of top level blocks)
- `block_path_str`: A `"."` separated list of names of the blocks from the top level block to the nested block of which the value will be passed to the operation.
- `operation`: A subclass of `operations.BaseBlockOperation`. It will have the `apply` method for applying changes to the matching block values.
- `streamfield`: The `StreamField` for which data is being migrated. This is used to get the definitions of the blocks.

Returns:

altered_raw_data:

Block paths

Operations for StreamField data migrations defined in `wagtail.blocks.migrations` require a “block path” to determine which blocks they should be applied to.

```
block_path = "" | block_name ("." block_name)*
block_name = str
```

A block path is either:

- the empty string, in which case the operation should be applied to the top-level stream; or
- a “.” (period) separated sequence of block names, where block names are the names given to the blocks in the StreamField definition.

Block names are the values associated with the “`type`” keys in the stream data’s dictionary structures. As such, traversing or selecting ListBlock members requires the use of the “`item`” block name.

The value that an operation’s `apply` method receives is the “`value`” member of the dict associated with the terminal block name in the block path.

For examples see [the tutorial](#).

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.

StreamField data migration reference

Details the tools provided in `wagtail.blocks.migrations` for StreamField data migrations.

1.6.3 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()

@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='+'
```

(continues on next page)

(continued from previous page)

```

        )
    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 customize 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'),
    ]

    edit_handler = TabbedInterface([
        ObjectList(first_tab_panels, heading='First tab'),
        ObjectList(second_tab_panels, heading='Second tab'),
    ])

```

Appearance

You can change the label used in the menu by changing the `verbose_name` of your model.

You can add an icon to the menu by passing an `icon` argument to the `register_setting` decorator:

```

@register_setting(icon='placeholder')
class GenericSocialMediaSettings(BaseGenericSetting):
    ...
    class Meta:
        verbose_name = "Social media settings for all sites"

```

(continues on next page)

(continued from previous page)

```
@register_setting(icon='placeholder')
class SiteSpecificSocialMediaSettings(BaseSiteSetting):
    ...
    class Meta:
        verbose_name = "Site-specific social media settings"
```

Wagtail's default icon set can be seen in our [icons overview](#). All icons available in a given project are displayed in 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': [
                ...
            ]
        }
    }
]
```

(continues on next page)

(continued from previous page)

```

        '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 Facebook at {{ generic_social_settings.facebook }},
    or Instagram at @{{ generic_social_settings.instagram }}.
{%- endwith %}

{%- with site_social_settings=settings("app_label.SiteSpecificSocialMediaSettings") %}
    Follow us on Facebook 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 utilize 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"]
```

(continues on next page)

(continued from previous 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 behavior 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('from_email', classname="col6"),
            ]),
            FieldPanel('subject', classname="col12"),
        ], heading="Form settings"),
    ]
```

(continues on next page)

(continued from previous page)

```

        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 customize 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>
  <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 + [

```

(continues on next page)

(continued from previous page)

```
FormSubmissionsPanel(),
FieldPanel('intro'),
# ...
]
```

Index

Form builder customization

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
)
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


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):
        return 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
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
```

(continues on next page)

(continued from previous page)

```

def process_form_submission(self, form):
    return 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):
        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([

```

(continues on next page)

(continued from previous page)

```

        FieldPanel('from_address', classname="col6"),
        FieldPanel('to_address', classname="col6"),
    ],
    FieldPanel('subject'),
], "Email"),
]

def serve(self, request, *args, **kwargs):
    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):
    return 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 %}
    {%
      endif %}

```

(continues on next page)

(continued from previous page)

```
{% 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 is submitted correctly, saves the whole form into a DB.
        """

        session_key_data = 'form_data-%s' % self.pk
        is_last_step = False
        step_number = request.GET.get('p', 1)
```

(continues on next page)

(continued from previous page)

```

paginator = Paginator(self.get_form_fields(), per_page=1)
try:
    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 the 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
    form_class = self.get_form_class_for_step(step)
    form = form_class(page=self, user=request.user)

```

(continues on next page)

(continued from previous page)

```

        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):
    page = ParentalKey('FormPage', on_delete=models.CASCADE, related_name='form_fields'
    ↵')

class FormPage(AbstractEmailForm):

```

(continues on next page)

(continued from previous page)

```
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 to 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)

            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 the 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
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='+',
```

(continues on next page)

(continued from previous page)

```

)
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'),
]

```

Customize form submissions listing in Wagtail Admin

The Admin listing of form submissions can be customized 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 subclass of `wagtail.admin.views.generic.base.BaseListingView` and Django's class based `ListView`.

Example:

```

from wagtail.contrib.forms.models import AbstractEmailForm, AbstractFormField
from wagtail.contrib.forms.views import SubmissionsListView

class CustomSubmissionsListView(SubmissionsListView):
    paginate_by = 50 # show more submissions per page, default is 20
    default_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

```

(continues on next page)

(continued from previous page)

```

class FormField(AbstractFormField):
    page = ParentalKey('FormPage', related_name='form_fields')

class FormPage(AbstractEmailForm):
    """Form Page with customized 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.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')

```

(continues on next page)

(continued from previous page)

```
# 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 the original method)
        title = '{}: {}'.format('Form', self.title)

        content = [title, '', email_content, '']

        # Add a link to the form page
```

(continues on next page)

(continued from previous 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

    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 an 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 the 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 behavior while extending an additional class, you can use the base mixins instead of the abstract models.

```
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):
```

(continues on next page)

(continued from previous page)

```

intro = RichTextField(blank=True)
# ...

class EmailFormPage(EmailFormMixin, FormMixin, BasePage):
    intro = RichTextField(blank=True)
    # ...

```

Custom validation for admin form pages

By default, pages that inherit from `FormMixin` will validate that each field added by an editor has a unique `clean_name`.

If you need to add custom validation, create a subclass of `WagtailAdminFormPageForm` and add your own `clean` definition and set the `base_form_class` on your `Page` model.

Note

Validation only applies when editors use the form builder to add fields in the Wagtail admin, not when the form is submitted by end users.

```

from wagtail.models import Page
from wagtail.contrib.forms.models import FormMixin, WagtailAdminFormPageForm

class CustomWagtailAdminFormPageForm(WagtailAdminFormPageForm):
    def clean(self):
        cleaned_data = super().clean()
        # Insert custom validation here, see `WagtailAdminFormPageForm.clean` for an example
        return cleaned_data

class FormPage(AbstractForm):
    base_form_class = CustomWagtailAdminFormPageForm

```

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 customization 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
    path("", 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](#).

Customizing

URLs

The `Page` class defines a `get_sitemap_urls` method which you can override to customize 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',
    },
}
```

Changed in version 6.2: Previous versions allowed passing a dict for DISTRIBUTION_ID to allow specifying different distribution IDs for different hostnames. This is now deprecated; instead, multiple distribution IDs should be defined as [*multiple backends*](#), with a HOSTNAMES parameter to define the hostnames associated with each one.

boto3 will attempt to discover credentials itself. You can read more about this here: [Boto 3 Docs](#). The user will need a policy similar to:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowWagtailFrontendInvalidation",
            "Effect": "Allow",
            "Action": "cloudfront:CreateInvalidation",
            "Resource": "arn:aws:cloudfront::<account id>:distribution/<distribution_id>"
        }
    ]
}
```

To specify credentials manually, pass them as additional parameters:

```
WAGTAILFRONTENDCACHE = {
    'cloudfront': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudfrontBackend',
        'DISTRIBUTION_ID': 'your-distribution-id',
        'AWS_ACCESS_KEY_ID': os.environ['FRONTEND_CACHE_AWS_ACCESS_KEY_ID'],
        'AWS_SECRET_ACCESS_KEY': os.environ['FRONTEND_CACHE_AWS_SECRET_ACCESS_KEY'],
    }
}
```

(continues on next page)

(continued from previous page)

```
'AWS_SESSION_TOKEN': os.environ['FRONTEND_CACHE_AWS_SESSION_TOKEN']
},
}
```

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-cc</code>	Yes (v10.0 or above)	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>CREDENTIALS</code> setting (more details below).
<code>azure-mgmt-re</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 `WAGTAILFRONTCENDCACHE` 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.

```
WAGTAILFRONTCENDCACHE = {
    'azure_cdn': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.AzureCdnBackend',
        'RESOURCE_GROUP_NAME': 'MY-WAGTAIL-RESOURCE-GROUP',
        'CDN_PROFILE_NAME': 'wagtailio',
        'CDN_ENDPOINT_NAME': 'wagtailio-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

WAGTAILFRONTCENDCACHE = {
    'azure_cdn': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.AzureCdnBackend',
        'RESOURCE_GROUP_NAME': 'MY-WAGTAIL-RESOURCE-GROUP',
        'CDN_PROFILE_NAME': 'wagtailio',
        'CDN_ENDPOINT_NAME': 'wagtailio-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	Essential	Reason
<code>azure-mgmt-fr</code>	Yes (v1.0 or above)	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.

Multiple backends

Multiple backends can be configured by adding multiple entries in `WAGTAILFRONTENDCACHE`.

By default, a backend will attempt to invalidate all invalidation requests. To only invalidate certain hostnames, specify them in `HOSTNAMES`:

```
WAGTAILFRONTENDCACHE = {
    'main-site': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.HTTPBackend',
        'LOCATION': 'http://localhost:8000',
        'HOSTNAMES': ['example.com']
    },
    'cdn': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudflareBackend',
        'BEARER_TOKEN': 'your cloudflare bearer token',
        'ZONEID': 'your cloudflare domain zone id',
        'HOSTNAMES': ['cdn.example.com']
    },
}
```

In the above example, invalidations for `cdn.example.com/foo` will be invalidated by Cloudflare, whilst `example.com/foo` will be invalidated with the `main-site` backend. This allows different configuration to be used for each backend, for example by changing the `ZONEID` for the Cloudflare backend:

```
WAGTAILFRONTENDCACHE = {
    'main-site': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudflareBackend',
        'BEARER_TOKEN': os.environ["CLOUDFLARE_BEARER_TOKEN"],
        'ZONEID': 'example.com zone id',
        'HOSTNAMES': ['example.com']
    },
    'other-site': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudflareBackend',
        'BEARER_TOKEN': os.environ["CLOUDFLARE_BEARER_TOKEN"],
        'ZONEID': 'example.net zone id',
        'HOSTNAMES': ['example.net']
    },
}
```

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).

Much like Django's `ALLOWED_HOSTS`, values in `HOSTNAMES` starting with a `.` can be used as a subdomain wildcard.

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)
```

(continues on next page)

(continued from previous page)

```
@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 `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, re_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):
        """
```

(continues on next page)

(continued from previous page)

```
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={
        '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()

    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 `get_url()` method or the `full_url` attribute on your page:

```
>>> event_page.get_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, re_path

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
        """
```

(continues on next page)

(continued from previous 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.

```
route(request, path_components)
```

This hooks the subpage URLs into Wagtail's routing.

This method overrides the default `Page.route()` method to route requests to the appropriate view method.

It sets `routable_resolver_match` on the request object to make sub-URL routing information available downstream in the same way that Django sets `request.resolver_match`.

```
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" %}
```

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` template-tag library:

```
{% load wagtailcore_tags wagtailsearchpromotions_tags %}

{%
    get_search_promotions search_query as search_promotions
}




    {% for search_promotion in search_promotions %}
        {% if search_promotion.page %}
            - ## {{ search\_promotion.page.title }}



{{ search\_promotion.description }}

        {% else %}
            - ## {{ search\_promotion.external\_link\_text }}



{{ search\_promotion.description }}

        {% endif %}
    {% endfor %}

```

Managing stored search queries

The `searchpromotions` module keeps a log of search queries as well as the number of daily hits through the `Query` and `QueryDailyHits` models.

```
class wagtail.contrib.search_promotions.models.Query

    classmethod get(query_string)
        Retrieves a stored search query or creates a new one if it doesn't exist.

    add_hit(date=None)
        Records another daily hit for a search query by creating a new record or incrementing the number of hits for
        an existing record. Defaults to using the current date but an optional date parameter can be passed in.
```

Example search view

Here's an example Django view for a search page that records a hit for the search query:

```
from django.template.response import TemplateResponse

from wagtail.models import Page
from wagtail.contrib.search_promotions.models import Query

def search(request):
    search_query = request.GET.get("query", None)

    if search_query:
        search_results = Page.objects.live().search(search_query)
        query = Query.get(search_query)

        # Record hit
        query.add_hit()
    else:
        search_results = Page.objects.none()

    return TemplateResponse(
        request,
        "search/search.html",
        {
            "search_query": search_query,
            "search_results": search_results,
        },
    )
```

Management Commands

`searchpromotions_garbage_collect`

```
./manage.py searchpromotions_garbage_collect
```

On high traffic websites, the stored queries and daily hits logs may get large and you may want to clean out old records. 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).

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 synchronization

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](#) (distributed under the MIT license) to provide users with the ability to create and edit HTML tables in Wagtail. Table blocks provides a caption field for accessibility.

Body

The recipe's step-by-step instructions and any other relevant information.

Table block *



Table headers

Display the first row AND first column as headers ▾

Which cells should be displayed as headers?

Table caption

A heading that identifies the overall topic of the table, and is useful for screen reader users.

Expected yield

Buns	Prep time	Cooking time	Instructions
12	30min	1.5h	All quantities as-is
18	40min	2h	1.5x all amounts
24	45min	Depending on oven size	2x all amounts

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',
        'remove_col',
        '-----',
        'undo',
        'redo'
```

(continues on next page)

(continued from previous page)

```
],
'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 behavior 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 behavior for newly created tables. Page editors can override this by checking 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`.
- `mergeCells` - Can be set to `True` or `False`, determined if merging cells is allowed. Remember to add ‘`mergeCells`’ to the ‘`contextMenu`’ option also.

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',
    ],
}

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.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 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()),
    ('rich_text', blocks.RichTextBlock()),
    ('image', ImageChooserBlock()),
    ('country', ChoiceBlock(choices=[
        ('be', 'Belgium'),
        ('fr', 'France'),
        ('de', 'Germany'),
        ('nl', 'Netherlands'),
        ('pl', 'Poland'),
        ('uk', 'United Kingdom'),
    ]))
```

(continues on next page)

(continued from previous page)

```
    ]))  
])
```

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 %}
```

Custom validation

As with other blocks, validation logic on `TypedTableBlock` can be customized by overriding the `clean` method (see [StreamField validation](#)). Raising a `ValidationError` exception from this method will attach the error message to the table as a whole. To attach errors to individual cells, the exception class `wagtail.contrib.typed_table_block.blocks.TypedTableBlockValidation` can be used - in addition to the standard `non_block_errors` argument, this accepts a `cell_errors` argument consisting of a nested dict structure where the outer keys are row indexes and the inner keys are column indexes. For example:

```
from django.core.exceptions import ValidationError  
from wagtail.blocks import IntegerBlock  
from wagtail.contrib.typed_table_block.blocks import TypedTableBlock,   
    TypedTableBlockValidationError  
  
class LuckyTableBlock(TypedTableBlock):  
    number = IntegerBlock()  
  
    def clean(self, value):  
        result = super().clean(value)  
        errors = {}  
        print(result.row_data)  
        for row_num, row in enumerate(result.row_data):  
            row_errors = {}  
            for col_num, cell in enumerate(row['values']):  
                if cell == 13:  
                    row_errors[col_num] = ValidationError("Table cannot contain the  
        ↪number 13")  
                if row_errors:  
                    errors[row_num] = row_errors  
  
        if errors:
```

(continues on next page)

(continued from previous page)

```
raise TypedTableBlockValidationError(cell_errors=errors)

return result
```

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 = [
    # ...
    # 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 our how-to guide: [Manage redirects](#).

Automatic redirect creation

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 [get_route_paths\(\)](#).

Disabling automatic redirect creation

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
<code>src</code>	This is the path to the file you wish to import redirects from.
<code>site</code>	This is the <code>site</code> for the site you wish to save redirects to.
<code>permanent</code>	If the redirects imported should be <code>permanent</code> (True) or not (False). It's True by default.
<code>from</code>	The column index you want to use as redirect from value.
<code>to</code>	The column index you want to use as redirect to value.
<code>dry_run</code>	Lets you run an import without doing any changes.
<code>ask</code>	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

API

You can create an API endpoint to retrieve redirects or find specific redirects by path.

See the [Wagtail API v2 configuration guide](#) documentation on how to configure the Wagtail API.

Add the following code to add the redirects endpoint:

```
from wagtail.contrib.redirects.api import RedirectsAPIViewSet

api_router.register_endpoint('redirects', RedirectsAPIViewSet)
```

With this configuration, redirects will be available at /api/v2/redirects/.

Specific redirects by path can be resolved with /api/v2/redirects/find/?html_path=<path>, which will return either a 200 response with the redirects detail, or a 404 not found response.

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.

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>).

1.6.4 Management commands

start

By default, the `start` command creates a project template, which contains your `models.py`, `templates`, and `settings` files. For example, to create new Wagtail project named `mysite`, use the command like this:

```
wagtail start mysite
```

You can also use the `--template` option with the `start` command to generate a custom template. See [The project template](#) for more information on how the command works with default and custom templates.

publish_scheduled

```
./manage.py publish_scheduled
```

This command publishes, updates, or unpublishes objects 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>] [--pages] [--non-pages]
```

This command deletes old revisions which are not in moderation, live, approved to go live, or the latest revision. If the `days` argument is supplied, only revisions older than the specified number of days will be deleted.

To prevent deleting important revisions when they become stale, you can refer to such revisions in a model using a `ForeignKey` with `on_delete=models.PROTECT`.

If the `pages` argument is supplied, only revisions of page models will be deleted. If the `non-pages` argument is supplied, only revisions of non-page models will be deleted. If both or neither arguments are supplied, revisions of all models will be deleted. If deletion of a revision is not desirable, mark `Revision` with `on_delete=models.PROTECT`.

purge_embeds

```
manage.py purge_embeds
```

This command deletes all the cached embed objects from the database. It is recommended to run this command after changes are made to any embed settings so that subsequent embed usage does not come from the database cache.

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
```

Silencing the command

You can prevent logs to the console by providing `--verbosity 0` as an argument:

```
python manage.py update_index --verbosity 0
```

If this is omitted or provided with any number above 0 it will produce the same logs.

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.

rebuild_references_index

```
./manage.py rebuild_references_index
```

This command populates the table that tracks cross-references between objects, used for the usage reports on images, documents, and snippets. This table is updated automatically saving objects, but it is recommended to run this command periodically to ensure that the data remains consistent.

Silencing the command

You can prevent logs to the console by providing `--verbosity 0` as an argument:

```
python manage.py rebuild_references_index --verbosity 0
```

show_references_index

```
./manage.py show_references_index
```

Displays a summary of the contents of the references index. This shows the number of objects indexed against each model type and can be useful to identify which models are being indexed without rebuilding the index itself.

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 behavior and need to ensure all renditions are created again.

This does not remove unused rendition images, 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.

convert_mariadb_uuids

```
./manage.py convert_mariadb_uuids
```

For sites using MariaDB, this command must be run once when upgrading to Django 5.0 and MariaDB 10.7 from any earlier version of Django or MariaDB. This is necessary because Django 5.0 introduces support for MariaDB's native UUID type, breaking backwards compatibility with CHAR-based UUIDs used in earlier versions of Django and MariaDB. New sites created under Django 5.0+ and MariaDB 10.7+ are unaffected.

1.6.5 Model reference

This document contains reference information for the model classes inside the `wagtail.models` 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](#). `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 object 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 an object that is already an instance of the most specific class, the object will be returned as is, and no database queries or other operations will be triggered.

If the object was originally created using a model that has since been removed from the codebase, an instance of the base class will be returned (without any custom field values or other functionality present on the original class). Usually, deleting these objects is the best course of action, but there is currently no safe way for Wagtail to do that at migration time.

`specific`

Returns this object in its most specific subclassed form with all field values fetched from the database. The result is cached in memory.

`specific_deferred`

Returns this object 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 object 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 object's `content_type` value from the `ContentType` model's cached manager, which will avoid a database query if the content type 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*)**static route_for_request** (*request: HttpRequest, path: str*) → `RouteResult | None`

Find the page route for the given HTTP request object, and URL path. The route result (`page`, `args`, and `kwargs`) will be cached via `request._wagtail_route_for_request`.

static find_for_request (*request: HttpRequest, path: str*) → `Page | None`

Find the page for the given HTTP request object, and URL path. The full page route will be cached via `request._wagtail_route_for_request`.

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`.

```
allowed_http_methods = [<HTTPMethod.DELETE>, <HTTPMethod.GET>,
<HTTPMethod.HEAD>, <HTTPMethod.OPTIONS>, <HTTPMethod.PATCH>,
<HTTPMethod.POST>, <HTTPMethod.PUT>]
```

When customizing this attribute, developers are encouraged to use values from Python’s built-in `http.HTTPMethod` enum in the list, as it is more robust, and makes use of values that already exist in memory. For example:

```
from http import HTTPMethod

class MyPage(Page):
    allowed_http_methods = [HTTPMethod.GET, HTTPMethod.OPTIONS]
```

The `http.HTTPMethod` enum wasn’t added until Python 3.11, so if your project uses an older version of Python, you can use uppercase strings instead. For example:

```
class MyPage(Page):
    allowed_http_methods = ["GET", "OPTIONS"]
```

`check_request_method`(*request*, **args*, ***kwargs*)

Checks the `method` attribute of the request against those supported by the page (as defined by `allowed_http_methods`) and responds accordingly.

If supported, `None` is returned, and the request is processed normally. If not, a warning is logged and an `HttpResponseNotAllowed` is returned, and any further request handling is terminated.

`handle_options_request`(*request*, **args*, ***kwargs*)

Returns an `HttpResponse` with an "Allow" header containing the list of supported HTTP methods for this page. This method is used instead of `serve()` to handle requests when the `OPTIONS` HTTP verb is detected (and `HTTPMethod.OPTIONS` is present in `allowed_http_methods` for this type of page).

`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.

`preview_sizes`

A list of dictionaries, each representing a preview size option for this object. Override this property to customize the preview sizes. Each dictionary in the list should include the following keys:

- `name`: A string representing the internal name of the preview size.
- `icon`: A string specifying the icon's name for the preview size button.
- `device_width`: An integer indicating the device's width in pixels.
- `label`: A string for the aria label on the preview size button.

```
@property
def preview_sizes(self):
    return [
        {
            "name": "mobile",
            "icon": "mobile-icon",
            "device_width": 320,
            "label": "Preview in mobile size"
        },
        # Add more preview size dictionaries as needed.
    ]
```

`default_preview_size`

The default preview size name to use in live preview. Defaults to "mobile", which is the first one defined in `preview_sizes`. If `preview_sizes` is empty, an `IndexError` will be raised.

`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_children ()`

Returns

A queryset of all the node's children

`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.

`get_admin_default_ordering ()`

Returns the default sort order for child pages to be sorted in viewing the admin pages index and not seeing search results.

The following sort orders are available:

- '`content_type`'
- '`-content_type`'
- '`latest_revision_created_at`'
- '`-latest_revision_created_at`'
- '`live`'
- '`-live`'

- 'ord'
- 'title'
- '-title'

For example to make a page sort by title for all the child pages only if there are < 20 pages.

```
class BreadsIndexPage(Page):
    def get_admin_default_ordering(self):
        if Page.objects.child_of(self).count() < 20:
            return 'title'
        return self.admin_default_ordering
```

admin_default_ordering

An attribute version for the method `get_admin_default_ordering()`, defaults to `'-latest_revision_created_at'`.

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()`

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 WAGTAIL_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.

private_page_options

Controls what privacy options are available for the page type.

The following options are available:

- 'password' - Can restrict to use a shared password
- 'groups' - Can restrict to users in specific groups
- 'login' - Can restrict to logged in users

```
class BreadPage(Page):
    ...

    # default
    private_page_options = ['password', 'groups', 'login']

    # disable shared password
    private_page_options = ['groups', 'login']

    # only shared password
    private_page_options = ['password']

    # no privacy options for this page model
    private_page_options = []
```

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 is used as a base for editing Pages of this type in the Wagtail page editor. This attribute can be set on a model to customize the Page editor form. Forms must be a subclass of `WagtailAdminPageForm`. See [Customizing 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 behavior 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`.

copy (`recursive=False, to=None, update_attrs=None, copy_revisions=True, keep_live=True, user=None, process_child_object=None, exclude_fields=None, log_action='wagtail.copy', reset_translation_key=True`)

Copies a given page

Parameters

`log_action` – flag for logging the action. Pass None to skip logging. Can be passed an action string. Defaults to '`wagtail.copy`'.

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, _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)

get_cache_key_components ()

The components of a `Page` which make up the `cache_key`. Any change to a page should be reflected in a change to at least one of these components.

cache_key

A generic cache key to identify a page in its current state. Should the page change, so will the key.

Customizations to the cache key should be made in `get_cache_key_components`.

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`

❶ 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 `WAGTAIL_CONTENT_LANGUAGES` 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 internationalization is enabled, all pages use this primary `Locale` record. This is to satisfy the database constraints and make it easier to switch internationalization 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 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.

    language_name
        Uses data from django.conf.locale to return the language name in English. For example, if the object's language_code were "fr", the return value would be "French".
        Raises KeyError if django.conf.locale has no information for the object's language_code value.

    language_name_local
        Uses data from django.conf.locale to return the language name in the language itself. For example, if the language_code were "fr" (French), the return value would be "français".
        Raises KeyError if django.conf.locale has no information for the object's language_code value.

    language_name_localized
        Uses data from django.conf.locale to return the language name in the currently active language. For example, if language_code were "fr" (French), and the active language were "da" (Danish), the return value would be "Fransk".
        Raises KeyError if django.conf.locale has no information for the object's language_code value.

    is_default
        Returns a boolean indicating whether this object is the default locale.

    is_active
        Returns a boolean indicating whether this object is the currently active locale.

    is_bidi
        Returns a boolean indicating whether the language is bi-directional.

    get_display_name() → str
```

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

```
class wagtail.models.TranslatableMixin
```

locale

(Foreign Key to `wagtail.models.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.

The `translation_key` and `locale` fields have a unique key constraint to prevent the object from being translated into a language more than once.

Note

This is currently enforced via `unique_together` in `TranslatableMixin.Meta`, but may be replaced with a `UniqueConstraint` in `TranslatableMixin.Meta.constraints` in the future.

If your model defines a `Meta class` (either with a new definition or inheriting `TranslatableMixin.Meta` explicitly), be mindful when setting `unique_together` or `constraints`. Ensure that there is either a `unique_together` or a `UniqueConstraint` (not both) on `translation_key` and `locale`. There is a system check for this.

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*, *exclude_fields=None*)

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.

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.

preview_sizes

A list of dictionaries, each representing a preview size option for this object. Override this property to customize the preview sizes. Each dictionary in the list should include the following keys:

- `name`: A string representing the internal name of the preview size.
- `icon`: A string specifying the icon's name for the preview size button.
- `device_width`: An integer indicating the device's width in pixels.
- `label`: A string for the aria label on the preview size button.

```
@property
def preview_sizes(self):
    return [
        {
            "name": "mobile",
            "icon": "phone",
            "device_width": 320,
            "label": "Mobile"
        },
        {
            "name": "small",
            "icon": "phone",
            "device_width": 640,
            "label": "Small"
        },
        {
            "name": "medium",
            "icon": "phone",
            "device_width": 960,
            "label": "Medium"
        },
        {
            "name": "large",
            "icon": "phone",
            "device_width": 1280,
            "label": "Large"
        }
    ]
```

(continues on next page)

(continued from previous page)

```
        "icon": "mobile-icon",
        "device_width": 320,
        "label": "Preview in mobile size"
    },
    # Add more preview size dictionaries as needed.
]
```

default_preview_size

The default preview size name to use in live preview. Defaults to "mobile", which is the first one defined in preview_sizes. If preview_sizes 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.

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 optimization.

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 customize 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, 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.
- `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 `True` to also create a log entry. 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.

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,  
        skip_permission_checks=False)
```

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`

LockableMixin

`LockableMixin` is an abstract model that can be added to any non-page Django model to allow its instances to be locked. Pages already include this mixin, so there is no need to add it. See [Locking snippets](#) for more details.

Database fields

```
class wagtail.models.LockableMixin
```

`locked`

(boolean)

A boolean that is set to `True` if the object is locked.

`locked_at`

(date/time)

The date/time when the object was locked.

`locked_by`

(foreign key to user model)

The user who locked the object.

Methods and properties

```
class wagtail.models.LockableMixin
```

`get_lock()`

Returns a sub-class of `BaseLock` if the instance is locked, otherwise `None`.

`with_content_json` (*content*)

Similar to `RevisionMixin.with_content_json()`, but with the following fields also preserved:

- `locked`

- `locked_at`
- `locked_by`

WorkflowMixin

`WorkflowMixin` is a mixin class that can be added to any non-page Django model to allow its instances to be submitted to workflows. This mixin requires `RevisionMixin` and `DraftStateMixin` to be applied. Pages already include this mixin, so there is no need to add it. See [Enabling workflows for snippets](#) for more details.

Methods and properties

```
class wagtail.models.WorkflowMixin
```

`classmethod get_default_workflow()`

Returns the active workflow assigned to the model.

For non-Page models, workflows are assigned to the model's content type, thus shared across all instances instead of being assigned to individual instances (unless `get_workflow()` is overridden).

This method is used to determine the workflow to use when creating new instances of the model. On Page models, this method is unused as the workflow can be determined from the parent page's workflow.

`has_workflow`

Returns `True` if the object has an active workflow assigned, otherwise `False`.

`get_workflow()`

Returns the active workflow assigned to the object.

`workflow_states`

Returns workflow states that belong to the object.

To allow filtering `WorkflowState` queries by the object, subclasses should define a `GenericRelation` to `WorkflowState` with the desired `related_query_name`. This property can be replaced with the `GenericRelation` or overridden to allow custom logic, which can be useful if the model has inheritance.

`workflow_in_progress`

Returns `True` if a workflow is in progress on the current object, otherwise `False`.

`current_workflow_state`

Returns the in progress or needs changes workflow state on this object, if it exists.

`current_workflow_task_state`

Returns (specific class of) the current task state of the workflow on this object, if it exists.

`current_workflow_task`

Returns (specific class of) the current task in progress on this object, if it exists.

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.

You can use the `purge_revisions` command to delete old revisions that are no longer in use.

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.

`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.

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 wagtail.models.RevisionsManager.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()
```

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.

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,  
       skip_permission_checks=False)
```

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](#))

PageViewRestriction

Database fields

```
class wagtail.models.PageViewRestriction

    page
        (foreign key to Page)
    password
        (text)
    restriction_type
        (text)
        Options: none, password, groups, login
```

Orderable (abstract)

Database fields

```
class wagtail.models.Orderable

    sort_order
        (number)
```

Workflow

Workflows represent sequences of tasks that must be approved for an action to be performed on an object - 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 snippets, and started.
        Inactive workflows cannot.
```

Methods and properties

```
class wagtail.models.Workflow

    start (obj, 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 an object.

Database fields

```
class wagtail.models.WorkflowState

    content_object
        (generic foreign key)

        The object on which the workflow has been started. For page workflows, the object is an instance of the base Page model.

    content_type
        (foreign key to ContentType)

        The content type of the object this workflow state belongs to. For page workflows, 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 workflow state belongs to. For page workflows, this means the content type of the Page model.

    object_id
        (string)

        The primary key of the object this revision belongs to.

    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)

Creates copies of previously approved task states with 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 revisions associated with task states linked to the current workflow state.

Task

Tasks represent stages in a workflow that 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
        Returns this object in its most specific subclassed form with all field values fetched from the database. The result is cached in memory.

    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(obj, user)
        Returns True if a user who would not normally be able to access the editor for the object should be able to if the object is currently on this task. Note that returning False does not remove permissions from users who would otherwise have them.
```

user_can_lock (*obj, user*)

Returns `True` if a user who would not normally be able to lock the object should be able to if the object is currently on this task. Note that returning `False` does not remove permissions from users who would otherwise have them.

user_can_unlock (*obj, user*)

Returns `True` if a user who would not normally be able to unlock the object should be able to if the object is currently on this task. Note that returning `False` does not remove permissions from users who would otherwise have them.

locked_for_user (*obj, user*)

Returns `True` if the object should be locked to a given user's edits. This can be used to prevent editing by non-reviewers.

get_actions (*obj, 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 object. 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 revision.

Database fields

class wagtail.models.TaskState**workflow_state**

(foreign key to [WorkflowState](#))

The workflow state which started this task state.

revision

(foreign key to [Revision](#))

The 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 canceled, rejected, or approved.

finished_by

(foreign key to user model)

The user who completed (canceled, rejected, approved) the task.

comment

(text)

A text comment is 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

Returns this object in its most specific subclassed form with all field values fetched from the database. The result is cached in memory.

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`)

WorkflowContentType

Represents the assignment of a workflow to a Django model.

Database fields

```
class wagtail.models.WorkflowContentType

    workflow
        (foreign key to Workflow)

    content_type
        (foreign key to ContentType)

    A foreign key to the ContentType object that represents the model that is assigned to the workflow.
```

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 LogEntryManager.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
```

timestamp

(date/time)

The date/time when the entry was created.

content_changed

(boolean)

A boolean that can be 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` or `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)
    A parental key to the page being subscribed to.

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.
```

1.6.6 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 customize the view-level behavior of the Wagtail admin and front-end. For customizations that only deal with model-level behavior - 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

@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.

Appearance

Hooks for modifying the display and appearance of basic CMS features and furniture.

get_avatar_url

Specify a custom user avatar to be displayed in the Wagtail admin. The callable passed to this hook should accept a `user` object and a `size` parameter that can be used in any resize or thumbnail processing you might need to do.

```
from datetime import datetime

@hooks.register('get_avatar_url')
def get_profile_avatar(user, size):
    today = datetime.now()
```

(continues on next page)

(continued from previous page)

```
is_christmas_day = today.month == 12 and today.day == 25

if is_christmas_day:
    return '/static/images/santa.png'

return None
```

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.html import format_html

from wagtail.admin.ui.components import Component
from wagtail import hooks

class WelcomePanel(Component):
    order = 50

    def render_html(self, parent_context):
        return format_html(
            """
            <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 subclass 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
```

(continues on next page)

(continued from previous page)

```
class CustomSettingsPanel(BaseSettingsPanel):
    name = 'custom'
    title = "My custom settings"
    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 [Customizing 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.
- `classname` - additional classes applied to the link.
- `order` - an integer that 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 customize its initialization 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
```

(continues on next page)

(continued from previous page)

```
@hooks.register('register_admin_menu_item')
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_admin_viewset

Register a `ViewSet` or `ViewSetGroup` to the admin, which combines a set of views, URL patterns, and menu item into a single unit. The callable fed into this hook should return an instance of `ViewSet` or `ViewSetGroup`.

```
from .views import CalendarViewSet

@hooks.register("register_admin_viewset")
def register_viewset():
    return CalendarViewSet()
```

Alternatively, it can also return a list of `ViewSet` or `ViewSetGroup` instances.

```
from .views import AgendaViewSetGroup, VenueViewSet

@hooks.register("register_admin_viewset")
def register_viewsets():
    return [AgendaViewSetGroup(), VenueViewSet()]
```

`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_help_menu_item`

As `register_admin_menu_item`, but registers menu items into the ‘Help’ sub-menu rather than the top-level menu.

`construct_help_menu`

As `construct_main_menu`, but modifies the ‘Help’ sub-menu rather than the top-level menu.

`register_admin_search_area`

Add an item to the Wagtail admin search “Other Searches”. The behavior 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.
- `classname` - additional CSS classes 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 customize 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}"
    ])
```

register_user_listing_buttons

Add buttons to the user list.

This hook takes two parameters:

- `user`: The user object to generate the button for
- `request_user`: The currently logged-in user

This example will add a simple button to the listing if the currently logged-in user is a superuser:

```
from wagtail.users.widgets import UserListingButton

@hooks.register("register_user_listing_buttons")
def user_listing_external_profile(user, request_user):
```

(continues on next page)

(continued from previous page)

```
if request_user.is_superuser:
    yield UserListingButton(
        "Show profile",
        f"/goes/to/a/url/{user.pk}",
        priority=30,
    )
```

`filter_form_submissions_for_user`

Allows access to form submissions to be customized 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 customizing 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_global_admin_css

Add additional CSS files or snippets to all admin pages.

```
# wagtail_hooks.py
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.

```
# wagtail_hooks.py
from django.templatetags.static import static
from django.utils.html import format_html, format_html_join

from wagtail import hooks

@hooks.register("insert_editor_js")
def editor_js():
    js_files = [
        'js/fireworks.js', # See https://fireworks.js.org for CDN import URLs
        'js/init-fireworks.js',
    ]
    return format_html_join(
        '\n',
        '<script src="{}"></script>',
        ((static(filename),) for filename in js_files)
    )
```

```
// js/init-fireworks.js
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();
});
```

`insert_global_admin_js`

Add additional JavaScript files or code snippets to all admin pages.

```
from django.utils.html import format_html

from wagtail import hooks

@hooks.register('insert_global_admin_js')
def global_admin_js():
    return format_html(
        '<script src="{}"></script>',
        "https://cdnjs.cloudflare.com/ajax/libs/three.js/r74/three.js"
    )
```

`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, user, view_name, 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
- `user` - the logged-in user
- `view_name` - either `index` or `edit`, depending on whether the button is being generated for the page listing or edit view
- `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 customizing 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):
    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 edit_awesome_page(request, parent_page)
```

`after_delete_page`

Do something after a `Page` object is deleted. Uses the same behavior 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 behavior as `before_create_page`, and is run 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):
    """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 behavior 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 behavior 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 behavior 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 behavior 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 behavior 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 behavior 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 customized 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) that 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 that 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 that 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

```
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, 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_wagtail_userbar

Add or remove items from the Wagtail *user bar*. Actions for adding and editing are provided by default. The callable passed into the hook must take the `request` object, a list of menu objects `items`, and an instance of page object `page`. 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. See also the `AccessibilityItem` class for the accessibility checker item in particular.

```
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">Puppies!</a></li>'

@hooks.register('construct_wagtail_userbar')
def add_puppy_link_item(request, items, page):
    return items.append( UserbarPuppyLinkItem() )
```

Admin workflow

Hooks for customizing 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_create_user(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
```

(continues on next page)

(continued from previous page)

```
from .models import AwesomePage
from .admin_views import edit_awesome_page

@hooks.register('before_create_user')
def do_before_create_user(request):
    return HttpResponse("A user creation form", content_type="text/plain")
```

after_delete_user

Do something after a User object is deleted. Uses the same behavior 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 behavior as `before_create_user`.

after_edit_user

Do something with a User object after it has been updated. Uses the same behavior 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 behavior as `before_create_user`.

Choosers

construct_page_chooser_queryset

Called when rendering the page chooser view, to allow the page listing QuerySet to be customized. 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 customized. 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)

    return documents
```

`construct_image_chooser_queryset`

Called when rendering the image chooser view, to allow the image listing QuerySet to be customized. 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 customized. 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, user, next_url=None):
    yield wagtailadmin_widgets.PageListingButton(
        '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
- `user` - the logged-in user
- `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, user, 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
- `user` - the logged-in user
- `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 in 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 testing the user's permission with `page.permissions_for_user`:

```
from wagtail.admin import widgets as wagtailadmin_widgets

@hooks.register('register_page_listing_buttons')
def page_custom_listing_buttons(page, user, next_url=None):
    yield wagtailadmin_widgets.ButtonWithDropdownFromHook(
        'More actions',
        hook_name='my_button_dropdown_hook',
        page=page,
        user=user,
        next_url=next_url,
        priority=50
    )

@hooks.register('my_button_dropdown_hook')
def page_custom_listing_more_buttons(page, user, next_url=None):
    page_perms = page.permissions_for_user(user)
    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 customized by overriding `wagtailadmin/pages/listing/_button_with_dropdown.html`. Make sure to leave the dropdown UI component itself as-is.

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 user object, and a context dictionary, and should modify the list of listing items in-place.

```
@hooks.register('construct_page_listing_buttons')
def remove_page_listing_button_item(buttons, page, user, context=None):
    if page.is_root:
        buttons.pop() # removes the last 'more' dropdown button on the root page ↴listing buttons
```

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

@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>")
```

`on_serve_page`

Called when Wagtail is serving a page, after `before_serve_page` but before the page's `serve()` method is called. Unlike `before_serve_page`, this hook allows you to modify the serving chain rather than just returning an alternative response.

The callable passed to this hook must accept a function as its argument and return a new function that will be used in its place. The passed-in function will be the next callable in the serving chain.

For example, to add custom cache headers to the response:

```
from wagtail import hooks

@hooks.register('on_serve_page')
def add_custom_headers(next_serve_page):
    def wrapper(page, request, args, kwargs):
        response = next_serve_page(page, request, args, kwargs)
        response['Custom-Header'] = 'value'
        return response
    return wrapper
```

Parameters passed to the function:

- `page` - the Page object being served
- `request` - the request object
- `args` - positional arguments that will be passed to the page's `serve` method
- `kwargs` - keyword arguments that will be passed to the page's `serve` method

This hook is particularly useful for:

- Adding/modifying response headers
- Implementing access restrictions
- Modifying the response content
- Adding logging or monitoring

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")
```

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 list 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 list
    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 list 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 list
    total = len(instances)

    if request.method == 'POST':
        for instance in instances:
            # Override the deletion behavior
            instance.delete()

    return HttpResponse(f"{total} snippets have been deleted", content_type="text/plain")
```

`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 receives the snippet's model class as an argument, and 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 customized 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 that 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 that 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(model):
    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, 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, the snippet object and a user, 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):
    buttons.pop() # Removes the 'delete' button
```

Bulk actions

Hooks for registering and customizing 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 subclass 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):
    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')

        def format_message(self, log_entry):
            return _('Hello %(audience)s') % {
                'audience': log_entry.data['audience'],
            }
```

Images

`register_image_operations`

Called on start-up. Register image operations that can be used to create renditions.

See [Custom image filters](#).

1.6.7 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 canceled

- `sender` - `WorkflowState`
- `instance` - The specific `WorkflowState` instance.
- `user` - The user who canceled 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 canceled.

- `sender` - TaskState
- `instance` - The specific TaskState instance.
- `user` - The user who canceled the task
- `kwargs` - Any other arguments passed to `task_cancelled.send()`

copy_for_translation_done

This signal is emitted from CopyForTranslationAction or CopyPageForTranslationAction when a translatable model or page is copied to a new locale (translated).

A translatable model is a model that implements the [TranslatableMixin](#).

- `sender` - CopyForTranslationAction or CopyPageForTranslationAction
- `source_obj` - The source object
- `target_obj` - The copy of the source object in the new locale

1.6.8 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.

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.elasticsearch8',
        '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 `searchpromotions_garbage_collect` command.

Internationalization

Wagtail supports the internationalization of content by maintaining separate trees of pages for each language.

For a guide on how to enable internationalization on your site, see the [configuration guide](#).

WAGTAIL_I18N_ENABLED

(boolean, default `False`)

When set to `True`, Wagtail's internationalization 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 as 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)")),
```

(continues on next page)

(continued from previous page)

```
('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 external providers such as YouTube or X (formerly 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.

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',  
    }  
}
```

Customize the behavior 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 recognized:

- **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. Recognized options are widget-specific, but `DraftailRichTextArea` accepts 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'
```

Customize Wagtail's behavior 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.

If the url is relative, Wagtail will not convert the link if there are more than one `Site` instances. This is to avoid accidentally matching coincidentally named pages on different sites.

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 recognized formats listed in the `DATE_INPUT_FORMATS`, `TIME_INPUT_FORMATS`, or `DATETIME_INPUT_FORMATS` setting respectively.

For example, to use US Imperial style date and time format (AM/PM times) in the Wagtail Admin, you'll need to override the Django format for your site's locale.

```
# settings.py
WAGTAIL_TIME_FORMAT = "%I:%M %p" # 03:00 PM
WAGTAIL_DATE_FORMAT = '%m/%d/%Y' # 01/31/2004
WAGTAIL_DATETIME_FORMAT = '%m/%d/%Y %I:%M %p' # 01/31/2004 03:00 PM

# Django uses formatting based on the system locale.
# Therefore we must specify a locale and then override the date
# formatting for that locale.
FORMAT_MODULE_PATH = ["formats"]
LANGUAGE_CODE = "en-US"
```

Next create the file `formats/en_US/formats.py` in your project:

```
# formats/en_US/formats.py

# Append our custom format to the Django defaults.
TIME_INPUT_FORMATS = [
    "%H:%M:%S", # '14:30:59',
    "%H:%M:%S.%f", # '14:30:59.000200',
    "%H:%M", # '14:30'
    # Custom
    "%I:%M %p",
]

DATETIME_INPUT_FORMATS = [
    "%Y-%m-%d %H:%M:%S", # '2006-10-25 14:30:59',
    "%Y-%m-%d %H:%M:%S.%f", # '2006-10-25 14:30:59.000200',
    "%Y-%m-%d %H:%M", # '2006-10-25 14:30',
    "%m/%d/%Y %H:%M:%S", # '10/25/2006 14:30:59',
    "%m/%d/%Y %H:%M:%S.%f", # '10/25/2006 14:30:59.000200',
    "%m/%d/%Y %H:%M", # '10/25/2006 14:30',
    "%m/%d/%Y %H:%M:%S", # '10/25/06 14:30:59',
    "%m/%d/%Y %H:%M:%S.%f", # '10/25/06 14:30:59.000200',
    "%m/%d/%Y %H:%M", # '10/25/06 14:30'
    # Custom
    "%m/%d/%Y %I:%M %p",
]

# Here you can also customize: DATE_INPUT_FORMATS, DATE_FORMAT,
# DATETIME_FORMAT, TIME_FORMAT, SHORT_DATE_FORMAT.
```

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 behavior is enabled by default.

Changed in version 6.3: This setting is deprecated. Set WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL = 0 to disable automatic preview updates instead.

WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL

```
WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL = 500
```

The interval (in milliseconds) to automatically check for changes made in the page or snippet editor before updating the preview in the preview panel. The default value is 500.

If set to 0, a refresh button will be shown in the panel and the preview is only updated when the button is clicked.

To completely disable previews, set *preview modes* to be empty on your model (preview_modes = []).

WAGTAIL_EDITING_SESSION_PING_INTERVAL

```
WAGTAIL_EDITING_SESSION_PING_INTERVAL = 10000
```

The interval (in milliseconds) to ping the server during an editing session. This is used to indicate that the session is active, as well as to display the list of other sessions that are currently editing the same content. The default value is 10000 (10 seconds). In order to effectively display the sessions list, this value needs to be set to under 1 minute. If set to 0, the interval will be disabled.

WAGTAILADMIN_GLOBAL_EDIT_LOCK

WAGTAILADMIN_GLOBAL_EDIT_LOCK can be set to True to prevent users from editing pages and snippets 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 = 30
```

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.

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

```
# Recommended
WAGTAILIMAGES_RENDERING_STORAGE = 'my_custom_storage'
# Or
WAGTAILIMAGES_RENDERING_STORAGE = 'myapp.backends.MyCustomStorage'
WAGTAILIMAGES_RENDERING_STORAGE = MyCustomStorage()
```

This setting allows image renditions to be stored using an alternative storage configuration. It is recommended to use a storage alias defined in [Django's STORAGES setting](#). Alternatively, this setting also accepts a dotted module path to a Storage subclass, or an instance of such a subclass. The default is None, meaning renditions will use the project's default storage.

Custom storage classes should subclass `django.core.files.storage.Storage`. See the [Django file storage API](#) for more information.

WAGTAILIMAGES_EXTENSIONS

```
WAGTAILIMAGES_EXTENSIONS = ['png', 'jpg']
```

A list of allowed image extensions that will be validated during image uploading. If this isn't supplied, all of AVIF, GIF, JPG, JPEG, PNG, WEBP 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.

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 [Collection 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 several serving methods that 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 can 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`'.

Warning

Allowing direct access to document URLs within `MEDIA_ROOT` may present a security risk if untrusted users are allowed to upload documents - in this case additional configuration will be required at the webserver level to handle these securely. See [User Uploaded Files](#).

`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_BLOCK_EMBEDDED_CONTENT`

```
WAGTAILDOCS_BLOCK_EMBEDDED_CONTENT = True
```

Wagtail serves a restrictive Content-Security policy for documents which ensures embedded content (such as the Javascript in a HTML file) is not executed. This functionality can be disabled by setting this to `False`.

This does not affect Javascript embedded in PDFs, however this is already executed in an isolated environment.

Unless absolutely necessary, it's strongly recommended not to change this setting.

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. This doesn't ensure that the uploaded file is valid, as files can be renamed to have an extension no matter what data they contain.

⚠ Warning

Allowing all file types may present a security risk if untrusted users are allowed to upload documents - in this case additional configuration will be required at the webserver level to handle these securely. See [User Uploaded Files](#).

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; 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.

Changed in version 6.2: This setting has been deprecated in favor of customizing the form classes via `UserViewSet.get_form_class()` and will be removed in a future release. For further information, see [Creating a custom UserViewSet](#).

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.

Changed in version 6.2: This setting has been deprecated in favor of customizing the form classes via `UserViewSet.get_form_class()` and will be removed in a future release. For further information, see [Creating a custom UserViewSet](#).

WAGTAIL_USER_CUSTOM_FIELDS

```
WAGTAIL_USER_CUSTOM_FIELDS = ['country']
```

A list of the extra custom fields to be appended to the default list. The resulting list is passed to `ModelForm`'s `Meta.fields` to generate the form fields.

Changed in version 6.2: This setting has been deprecated in favor of customizing the form classes via `UserViewSet.get_form_class()` and will be removed in a future release. For further information, see [Creating a custom UserViewSet](#).

WAGTAILADMIN_USER_LOGIN_FORM

```
WAGTAILADMIN_USER_LOGIN_FORM = 'users.forms.LoginForm'
```

Allows the default `LoginForm` to be extended with extra fields.

WAGTAILADMIN_LOGIN_URL

```
WAGTAILADMIN_LOGIN_URL = 'http://example.com/login/'
```

This specifies the URL to redirect when a user attempts to access a Wagtail admin page without being logged in. If omitted, Wagtail will fall back to using the standard login view (typically `/admin/login/`).

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.

Any provided query string will merge with the default parameters. For example, using the setting `//www.gravatar.com/avatar?d=robohash` will use the `robohash` override instead of the default `mp` (mystery person). The `s` parameter will be ignored as this is specified depending on location within the admin interface.

See the [Gravatar images URL documentation](#) for more details.

Changed in version 6.4: Added query string merging.

WAGTAIL_USER_TIME_ZONES

Logged-in users can choose their current time zone for the admin interface in the account settings. If there is 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.)

By default, this uses the set of timezones returned by `zoneinfo.available_timezones()`. 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).

WAGTAIL_ENABLE_WHATS_NEW_BANNER

```
WAGTAIL_ENABLE_WHATS_NEW_BANNER = True
```

For new releases, Wagtail may show a notification banner on the dashboard that helps users learn more about the UI changes and new features in the release. Users can dismiss this banner, which will hide it until the next release. If you'd rather not show these banners, you can disable it with this setting.

Frontend authentication

WAGTAIL_PASSWORD_REQUIRED_TEMPLATE

```
WAGTAIL_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.

WAGTAILDOCS_PASSWORD_REQUIRED_TEMPLATE

```
WAGTAILDOCS_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 customized 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.

WAGTAIL_PRIVATE_PAGE_OPTIONS

If you'd rather users not have the ability to use a shared password to make pages private, you can disable it with this setting:

```
WAGTAIL_PRIVATE_PAGE_OPTIONS = {"SHARED_PASSWORD": False}
```

See [Private pages](#) for more details.

WAGTAILDOCS_PRIVATE_COLLECTION_OPTIONS

If you'd rather users not have the ability to use a shared password to make collections (used for documents) private, you can disable it with this setting:

```
WAGTAILDOCS_PRIVATE_COLLECTION_OPTIONS = {"SHARED_PASSWORD": False}
```

See [Private pages](#) for more details.

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 behavior 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.

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 the documentation linked above for the 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, there 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_WORKFLOW_ENABLED`

```
WAGTAIL_WORKFLOW_ENABLED = True
```

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. To use workflows in this mode, set `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. This is the default, `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.

1.6.9 The project template

By default, running the `wagtail start` command (e.g. `wagtail start mysite`) will create a new Django project with the following structure:

```
mysite/
    home/
        migrations/
            __init__.py
            0001_initial.py
```

(continues on next page)

(continued from previous page)

```
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
```

Using custom templates

To use a custom template instead, you can specify the `--template` option when running the `wagtail start` command. This option accepts a directory, file path, or URL of a custom project template (similar to `django-admin startproject --template`).

For example, with a custom template hosted as a GitHub repository, you can use a URL like the following:

```
wagtail start myproject --template=https://github.com/githubuser/wagtail-awesome-
↪template/archive/main.zip
```

See [Templates \(start command\)](#) for a list of custom templates you can use for your projects.

Default project template

The following sections are references for the default project template:

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
```

Writing custom templates

Some examples of custom templates.

- github.com/thibaudcolas/wagtail-tutorial-template
- github.com/torchbox/wagtail-news-template

You might get an error while trying to generate a custom template. This happens because the `--template` option attempts to parse the templates files in your custom template. To avoid this error, wrap the code in each of your template files with the `{% verbatim %}{% endverbatim %}` tag, like this:

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

{% load wagtailcore_tags %}

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

{% block content %}
    <h1>{{ page.title }}</h1>
    <div class="intro">{{ page.intro|richtext }}</div>
    {% for post in page.get_children %}
        <h2><a href="{{ pageurl post }}>{{ post.title }}</a></h2>
        {{ post.specific.intro }}
        {{ post.specific.body }}
    {% endfor %}
    {% endblock %}
    {% endverbatim %}
```

1.6.10 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.django.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

`fullpageurl()`

Generate an absolute URL (`http://example.com/foo/bar/`) for a Page instance:

```
<meta property="og:url" content="{{ fullpageurl(page) }}" />
```

See `fullpageurl` for more information.

`pageurl()`

Generate a URL (`/foo/bar/`) 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 render an `` tag:

```
{{ image(page.header_image, "fill-1024x200", class="header-image") }}
```

Or resize an image and retrieve the resized image object (rendition) for more bespoke use:

```
{% set background=image(page.background_image, "max-1024x1024") %}  
<div class="wrapper" style="background-image: url('{{ background.url }}');"></div>
```

See *How to use images in templates* for more information

`srcset_image()`

Resize an image, and render an `` tag including `srcset` with multiple sizes. Browsers will select the most appropriate image to load based on [responsive image rules](#). The `sizes` attribute is essential unless you store the output of `srcset_image` for later use.

```
{{ srcset_image(page.photo, "width-{400,800}", sizes="(max-width: 600px) 400px, 80vw  
→") }}
```

This outputs:

```

```

Or resize an image and retrieve the renditions for more bespoke use:

```
{% set bg=srcset_image(page.background_image, "max-{512x512,1024x1024}") %}  
<div class="wrapper" style="background-image: image-set(url('{{ bg.renditions[0].url }}  
→) 1x, url('{{ bg.renditions[1].url }}) 2x);"></div>
```

picture()

Resize or convert an image, rendering a `<picture>` tag including multiple source formats with `srcset` for multiple sizes, and a fallback `` tag. Browsers will select the first supported image format, and pick a size based on responsive image rules.

`picture` can render an image in multiple formats:

```
{% picture(page.photo, "format-{avif,webp,jpeg}|width-400") %}
```

This outputs:

```
<picture>
  <source srcset="/media/images/pied-wagtail.width-400.avif" type="image/avif">
  <source srcset="/media/images/pied-wagtail.width-400.webp" type="image/webp">
  
</picture>
```

Or render multiple formats and multiple sizes like `srcset_image` does. The `sizes` attribute is essential when the `picture` tag renders images in multiple sizes:

```
{% picture(page.header_image, "format-{avif,webp,jpeg}|width-{400,800}", sizes="80vw") %}
```

This outputs:

```
<picture>
  <source sizes="80vw" srcset="/media/images/pied-wagtail.width-400.avif 400w, /media/images/pied-wagtail.width-800.avif 800w" type="image/avif">
  <source sizes="80vw" srcset="/media/images/pied-wagtail.width-400.webp 400w, /media/images/pied-wagtail.width-800.webp 800w" type="image/webp">
  
</picture>
```

Or resize an image and retrieve the renditions for more bespoke use:

```
{% set bg=picture(page.background_image, "format-{avif,jpeg}|max-{512x512,1024x1024}") %}
<div class="wrapper" style="background-image: image-set(url({{ bg.formats['avif'][0].url }}) 1x type('image/avif'), url({{ bg.formats['avif'][1].url }}) 2x type('image/avif'), url({{ bg.formats['jpeg'][0].url }}) 1x type('image/jpeg'), url({{ bg.formats['jpeg'][1].url }}) 2x type('image/jpeg'));"></div>
```

| 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 behavior of Jinja's `{% include %}`, so it does not implement the Django version's feature of only passing specified variables into the context.

1.6.11 Panels

Built-in Fields and Choosers

Wagtail's panel mechanism automatically recognizes Django model fields and provides them with an appropriate widget for input. You can use it by defining the field in your Django model as normal and passing the field name into [FieldPanel](#) (or a suitable panel type) when defining your panels.

Here are some built-in panel types that you can use in your panel definitions. These are all subclasses of the base [Panel](#) class, and unless otherwise noted, they accept all of [Panel](#)'s parameters in addition to their own.

FieldPanel

```
class wagtail.admin.panels.FieldPanel(field_name, widget=None, disable_comments=None,
                                       permission=None, read_only=False, **kwargs)
```

This is the panel to use for basic Django model field types. It provides a default icon and heading based on the model field definition, but they can be customized by passing additional arguments to the constructor. For more details, see [Panel customization](#).

`field_name`

This is the name of the class property used in your model definition.

`widget` (optional)

This parameter allows you to specify a [Django form widget](#) to use instead of the default widget for this field type.

`disable_comments` (optional)

This allows you to prevent a field-level comment button from showing for this panel if set to `True`. See [Create and edit comments](#).

`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.

`read_only` (optional)

Allows you to prevent a model field value from being set or updated by editors.

For most field types, the field value will be rendered in the form for editors to see (along with field's label and help text), but no form inputs will be displayed, and the form will ignore attempts to change the value in POST data. For example by injecting a hidden input into the form HTML before submitting.

By default, field values from `StreamField` or `RichTextField` are redacted to prevent rendering of potentially insecure HTML mid-form. You can change this behavior for custom panel types by overriding `Panel.format_value_for_display()`.

`attrs` (optional)

Allows a dictionary containing HTML attributes to be set on the rendered panel. If you assign a value of `True` or `False` to an attribute, it will be rendered as an HTML5 boolean attribute.

Note

A plain string in a panel definition is equivalent to a `FieldPanel` with no arguments.

Use this:

```
content_panels = Page.content_panels + ["title", "body"]
```

Instead of

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

MultifieldPanel

```
class wagtail.admin.panels.MultifieldPanel(children=(), *args, permission=None, **kwargs)
```

This panel condenses several [FieldPanel](#)s or choosers, from a list or tuple, under a single heading string. To save space, you can *collapse the panel by default*.

children

A list or tuple of child panels

permission (optional)

Allows a panel 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 panel will be omitted from the form. Similar to [FieldPanel.permission](#).

attrs (optional)

Allows a dictionary containing HTML attributes to be set on the rendered panel. If you assign a value of `True` or `False` to an attribute, it will be rendered as an HTML5 boolean attribute.

InlinePanel

```
class wagtail.admin.panels.InlinePanel(relation_name, panels=None, label='', min_num=None, max_num=None, **kwargs)
```

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. For a full explanation of the usage of [InlinePanel](#), see [Inline models](#). To save space, you can *collapse the panel by default*.

relation_name

The related_name label given to the cluster’s ParentalKey relation.

panels (optional)

The list of panels that will make up the child object’s form. If not specified here, the *panels* definition on the child model will be used.

label

Text for the add button and heading for child panels. Used as the heading when *heading* is not present.

min_num (optional)

Minimum number of forms a user must submit.

max_num (optional)

Maximum number of forms a user must submit.

attrs (optional)

Allows a dictionary containing HTML attributes to be set on the rendered panel. If you assign a value of `True` or `False` to an attribute, it will be rendered as an HTML5 boolean attribute.

Note

A plain string in a panel definition is equivalent to an [InlinePanel](#) with no arguments.

Use this:

```
content_panels = Page.content_panels + ["gallery_images"]
```

Instead of

```
content_panels = Page.content_panels + [
    InlinePanel('gallery_images'),
]
```

JavaScript DOM events

You may want to execute some JavaScript when `InlinePanel` items are ready, added or removed. The `w-formset:ready`, `w-formset:added` and `w-formset:removed` events allow this.

For example, given a child model that provides a relationship between `Blog` and `Person` on `BlogPage`.

```
class CustomInlinePanel(InlinePanel):
    class BoundPanel(InlinePanel.BoundPanel):
        class Media:
            js = ["js/inline-panel.js"]

class BlogPage(Page):
    # .. fields

    content_panels = Page.content_panels + [
        CustomInlinePanel("blog_person_relationship"),
        # ... other panels
    ]
```

Using JavaScript is as follows.

```
// static/js/inline-panel.js

document.addEventListener('w-formset:ready', function (event) {
    console.info('ready', event);
});

document.addEventListener('w-formset:added', function (event) {
    console.info('added', event);
});

document.addEventListener('w-formset:removed', function (event) {
    console.info('removed', event);
});
```

Events will be dispatched and can trigger custom JavaScript logic such as setting up a custom widget.

MultipleChooserPanel

```
class wagtail.admin.panels.MultipleChooserPanel(relation_name, chooser_field_name=None,
                                                panels=None, label='', min_num=None,
                                                max_num=None, **kwargs)
```

This panel is a variant of `InlinePanel` that can be used when the inline model includes a `ForeignKey` relation to a model that implements Wagtail's chooser interface. Wagtail images, documents, snippets, and pages all implement this interface, and other models may do so by [registering a custom ChooserViewSet](#).

Rather than the “Add” button inserting a new form to be filled in individually, it immediately opens up the chooser interface for that related object, in a mode that allows multiple items to be selected. The user is then returned to the main edit form with the appropriate number of child panels added and pre-filled.

`MultipleChooserPanel` accepts an additional required argument `chooser_field_name`, specifying the name of the `ForeignKey` relation that the chooser is linked to.

For example, given a child model that provides a gallery of images on `BlogPage`:

```
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'),
    ]
```

The `MultipleChooserPanel` definition on `BlogPage` would be:

```
MultipleChooserPanel(
    'gallery_images', label="Gallery images", chooser_field_name="image"
)
```

FieldRowPanel

```
class wagtail.admin.panels.FieldRowPanel(children=(), *args, permission=None, **kwargs)
```

This panel creates a columnar layout in the editing interface, where each of the child Panels appears alongside each other rather than below.

The 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”.

By default, the panel is divided into equal-width columns, but this can be overridden by adding `col*` class names to each of the child Panels of the `FieldRowPanel`. The Wagtail editing interface is laid out using a grid system. Classes `col1-col12` can be applied to each child of a `FieldRowPanel` to define how many columns they span out of the total number of columns. When grid items add up to 12 columns, the class `col3` will ensure that field appears 3 columns wide or a quarter the width. `col4` would cause the field to be 4 columns wide, or a third the width.

children

A list or tuple of child panels to display on the row

permission (optional)

Allows a panel 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 panel will be omitted from the form. Similar to `FieldPanel.permission`.

attrs (optional)

Allows a dictionary containing HTML attributes to be set on the rendered panel. If you assign a value of `True` or `False` to an attribute, it will be rendered as an HTML5 boolean attribute.

HelpPanel

```
class wagtail.admin.panels.HelpPanel (content='', template='wagtailadmin/panels/help_panel.html',  
                                     **kwargs)
```

A panel to display helpful information to the user.

This panel does not support the `help_text` parameter.

content

HTML string that gets displayed in the panel.

template

Path to a template rendering the full panel HTML.

attrs (optional)

Allows a dictionary containing HTML attributes to be set on the rendered panel. If you assign a value of `True` or `False` to an attribute, it will be rendered as an HTML5 boolean attribute.

PageChooserPanel

```
class wagtail.admin.panels.PageChooserPanel (field_name, page_type=None,  
                                         can_choose_root=False, **kwargs)
```

While `FieldPanel` also supports `ForeignKey` to `Page` models, you can explicitly use `PageChooserPanel` to allow Page-specific customizations.

```
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 specialized 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’.

FormSubmissionsPanel

```
class wagtail.contrib.forms.panels.FormSubmissionsPanel(**kwargs)
```

This panel adds a single, read-only section in the edit interface for pages implementing the `wagtail.contrib.forms.models.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

class ContactFormPage(AbstractForm):
    content_panels = [
        FormSubmissionsPanel(),
    ]
```

TitleFieldPanel

```
class wagtail.admin.panels.TitleFieldPanel(*args, apply_if_live=False, classname='title',
                                         placeholder=True, targets=['slug'], **kwargs)
```

Prepares the default widget attributes that are used on Page title fields. Can be used outside of pages to easily enable the slug field sync functionality.

Parameters

- **apply_if_live** – (optional) If `True`, the built in slug sync behaviour will apply irrespective of the published state. The default is `False`, where the slug sync will only apply when the instance is not live (or does not have a `live` property).
- **classname** – (optional) A CSS class name to add to the panel's HTML element. Default is `"title"`.
- **placeholder** – (optional) If a value is provided, it will be used as the field's placeholder, if `False` is provided no placeholder will be shown. If `True`, a placeholder value of `"Title*` will be used or `"Page Title*` if the model is a `Page` model. The default is `True`. If a widget is provided with a placeholder, the widget's value will be used instead.
- **targets** – (optional) This allows you to override the default target of the field named `slug` on the form. Accepts a list of field names, default is `["slug"]`. Note that the slugify/urlify behaviour relies on usage of the `wagtail.admin.widgets.slug` widget on the slug field.

This is the panel to use for Page title fields or main titles on other models. It provides a default classname, placeholder, and widget attributes to enable the automatic sync with the slug field in the form. Many of these defaults can be customized by passing additional arguments to the constructor. All the same `FieldPanel` arguments are supported including a custom widget. For more details, see [Panel customization](#).

Panel customization

By adding extra parameters to your panel/field definitions, you can control much of how your fields will display in the Wagtail page editing interface. Wagtail's page editing interface takes much of its behavior from Django's admin, so you may find many options for customization covered there. (See [Django model field reference](#)).

Icons

Use the `icon` argument to the panel constructor to override the icon to be displayed next to the panel's heading. For a list of available icons, see [Available icons](#).

Heading

Use the `heading` argument to the panel constructor to set the panel's heading. This will be used for the input's label and displayed on the content minimap. If left unset for FieldPanels, it will be set automatically using the form field's label (taken in turn from a model field's `verbose_name`).

CSS classes

Use the `classname` argument to the panel constructor to add CSS classes to the panel. The class will be applied to the HTML `<section>` element of the panel. This can be used to add extra styling to the panel or to control its behavior.

The `title` class can be used to make the input stand out with a bigger font size and weight.

The `collapsed` class 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",
    ),
]
```

Help text

Use the `help_text` argument to the panel constructor to customize the help text to be displayed above the input. If unset for FieldPanels, it will be set automatically using the form field's `help_text` (taken in turn from a model field's `help_text`).

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 customize placeholders for a `Book` snippet model:

```
# 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 = [
        TitleFieldPanel('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 top-level panel definition, a `FieldPanel` will be constructed for each field in your model. If you intend to hide a field on the Wagtail page editor, define the field with `editable=False`. If a field is not present in the panels definition, it will also be hidden.

Permissions

Most panels can accept a `permission` kwarg, allowing the set of panels or specific panels to be restricted to a set permissions. See [Permissions](#) for details about working with permissions in Wagtail.

In this example, ‘notes’ will be visible to all editors, ‘cost’ and ‘details’ will only be visible to those with the `submit` permission, ‘budget approval’ will be visible to super users only. Note that super users will have access to all fields.

```
content_panels = [
    FieldPanel("notes"),
    MultiFieldPanel(
        [
            FieldPanel("cost"),
            FieldPanel("details"),
        ],
        heading="Budget details",
        classname="collapsed",
        permission="submit"
    ),
    FieldPanel("budget_approval", permission="superuser"),
]
```

Additional HTML attributes

Use the `attrs` parameter to add custom attributes to the HTML element of the panel. This allows you to specify additional attributes, such as `data-*` attributes. The `attrs` parameter accepts a dictionary where the keys are the attribute names and these will be rendered in the same way as Django’s `widget attrs` where `True` and `False` will be treated as HTML5 boolean attributes.

For example, you can use the `attrs` parameter to integrate your Stimulus controller into the panel:

```
content_panels = [
    MultiFieldPanel(
        [
            FieldPanel('cover'),
            FieldPanel('book_file'),
            FieldPanel('publisher', attrs={'data-my-controller-target': 'myTarget
→'}),
        ],
        heading="Collection of Book Fields",
        classname="collapsed",
        attrs={'data-controller': 'my-controller'},
    ),
]
```

Panel API

This document describes the reference API for the base `Panel` and the `BoundPanel` classes that are used to render Wagtail's panels. For available panel types and how to use them, see [Built-in Fields and Choosers](#).

`Panel`

```
class wagtail.admin.panels.Panel(heading='', classname='', help_text='', base_form_class=None, icon='', attrs=None)
```

Defines part (or all) of the edit form interface for pages and other models within the Wagtail admin. Each model has an associated top-level panel definition (also known as an edit handler), 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.

The following parameters can be used to customize how the panel is displayed. For more details, see [Panel customization](#).

Parameters

- `heading` – The heading text to display for the panel.
- `classname` – A CSS class name to add to the panel's HTML element.
- `help_text` – Help text to display within the panel.
- `base_form_class` – The base form class to use for the panel. Defaults to the model's `base_form_class`, before falling back to `WagtailAdminModelForm`. This is only relevant for the top-level panel.
- `icon` – The name of the icon to display next to the panel heading.
- `attrs` – A dictionary of HTML attributes to add to the panel's HTML element.

`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.

1.6.12 Viewsets

Viewsets are Wagtail's mechanism for defining a group of related admin views with shared properties, as a single unit.

ViewSet**class wagtail.admin.viewsets.base.ViewSet (name=None, **kwargs)**

Defines a viewset to be registered with the Wagtail admin.

All properties of the viewset can be defined as class-level attributes, or passed as keyword arguments to the constructor (in which case they will override any class-level attributes). Additionally, the `name` property can be passed as the first positional argument to the constructor.

For more information on how to use this class, see [Using ViewSet to group custom admin views](#).

name = None

A name for this viewset, used as the default URL prefix and namespace.

url_prefix

The preferred URL prefix for views within this viewset. When registered through Wagtail's `register_admin_viewset` hook, this will be used as the URL path component following `/admin/`. Other URL registration mechanisms (e.g. editing `urls.py` manually) may disregard this and use a prefix of their own choosing.

Defaults to the viewset's name.

url_namespace

The URL namespace for views within this viewset. Will be used internally as the application namespace for the viewset's URLs, and generally be the instance namespace too.

Defaults to the viewset's name.

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.

icon = ''

The icon to use across the views.

menu_icon

The icon used for the menu item that appears in Wagtail's sidebar.

Defaults to `icon`.

menu_label = ''

The displayed label used for the menu item.

menu_name = ''

The name argument passed to the `MenuItem` constructor, becoming the `name` attribute value for that instance. This can be useful when manipulating the menu items in a custom menu hook, e.g. `construct_main_menu`. If unset, a slugified version of `menu_label` is used.

menu_order = 8999

An integer determining the order of the menu item, 0 being the first place. By default, it will be the last item before Reports, whose order is 9000.

menu_url

The URL to be used for the menu item.

Defaults to the first URL returned by `get_urlpatterns()`.

menu_item_class

A `wagtail.admin.menu.MenuItem` subclass to be registered with a menu hook.

menu_hook

The name of the hook to register the menu item within.

This takes precedence over `add_to_admin_menu` and `add_to_settings_menu`.

add_to_admin_menu = False

Register the menu item within the admin's main menu.

add_to_settings_menu = False

Register the menu item within the admin’s “Settings” menu. This takes precedence if both `add_to_admin_menu` and `add_to_settings_menu` are set to True.

get_menu_item(order=None)

Returns a `wagtail.admin.menu.MenuItem` instance to be registered with the Wagtail admin.

The `order` parameter allows the method to be called from the outside (e.g. a `ViewSetGroup`) to create a sub menu item with the correct order.

ViewSetGroup

class wagtail.admin.viewsets.base.ViewSetGroup

A container for grouping together multiple `ViewSet` instances. Creates a menu item with a submenu for accessing the main URL for each instances.

For more information on how to use this class, see *Combining multiple ViewSets using a ViewSetGroup*.

items = ()

A list or tuple of `ViewSet` classes or instances to be grouped together.

menu_icon = 'folder-open-inverse'

The icon used for the menu item that appears in Wagtail’s sidebar.

menu_label = ''

The displayed label used for the menu item.

menu_name = ''

The name argument passed to the `MenuItem` constructor, becoming the `name` attribute value for that instance. This can be useful when manipulating the menu items in a custom menu hook, e.g. `construct_main_menu`. If unset, a slugified version of `menu_label` is used.

menu_order = 8999

An integer determining the order of the menu item, 0 being the first place. By default, it will be the last item before Reports, whose order is 9000.

menu_item_class

A `wagtail.admin.menu.MenuItem` subclass to be registered with a menu hook.

add_to_admin_menu = True

Register the menu item within the admin’s main menu.

get_menu_item(order=None)

Returns a `wagtail.admin.menu.MenuItem` instance to be registered with the Wagtail admin.

The `order` parameter allows the method to be called from the outside (e.g. a `ViewSetGroup`) to create a sub menu item with the correct order.

ModelViewSet

```
class wagtail.admin.viewsets.model.ModelViewSet(name=None, **kwargs)
```

A viewset to allow listing, creating, editing and deleting model instances.

All attributes and methods from [ViewSet](#) are available.

For more information on how to use this class, see [Generic views](#).

model

Required; the model class that this viewset will work with. The `model_name` will be used as the URL prefix and namespace, unless these are specified explicitly via the `name`, `url_prefix` or `url_namespace` attributes.

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.

get_edit_handler()

Returns the appropriate edit handler for this `ModelViewSet` class. It can be defined either on the model itself or on the `ModelViewSet`, as the `edit_handler` or `panels` properties. If none of these are defined, it will return `None` and the form will be constructed as a Django form using `get_form_class()` (without using [Panels](#)).

get_permissions_to_register()

Returns a queryset of `Permission` objects to be registered with the `register_permissions` hook. By default, it returns all permissions for the model if `inspect_view_enabled` is set to `True`. Otherwise, the "view" permission is excluded.

menu_label

The displayed label used for the menu item.

Defaults to the title-cased version of the model's `verbose_name_plural`.

add_to_reference_index = True

Register the model to the reference index to track its usage. For more details, see [Manage the reference index](#).

ordering = None

The default ordering to use for the index view. Can be a string or a list/tuple in the same format as Django's `ordering`.

list_per_page = 20

The number of items to display per page in the index view. Defaults to 20.

list_display

A list or tuple, where each item is either:

- The name of a field on the model;
- The name of a callable or property on the model that accepts a single parameter for the model instance; or

- An instance of the `wagtail.admin.ui.tables.Column` class.

If the name refers to a database field, the ability to sort the listing by the database column will be offered and the field's verbose name will be used as the column header.

If the name refers to a callable or property, an `admin_order_field` attribute can be defined on it to point to the database column for sorting. A `short_description` attribute can also be defined on the callable or property to be used as the column header.

This list will be passed to the `list_display` attribute of the index view. If left unset, the `list_display` attribute of the index view will be used instead, which by default is defined as `["__str__", wagtail.admin.ui.tables.UpdatedAtColumn()]`.

`list_export`

A list or tuple, where each item is the name of a field, an attribute, or a single-argument callable on the model to be exported.

`list_filter`

A list or tuple, where each item is the name of model fields of type `BooleanField`, `CharField`, `DateField`, `DateTimeField`, `IntegerField` or `ForeignKey`. Alternatively, it can also be a dictionary that maps a field name to a list of lookup expressions. This will be passed as django-filter's `FilterSet.Meta.fields` attribute. See [its documentation](#) for more details. If `filterset_class` is set, this attribute will be ignored.

`filterset_class`

A subclass of `wagtail.admin.filters.WagtailFilterSet`, which is a subclass of `django_filters.FilterSet`. This will be passed to the `filterset_class` attribute of the index view.

`export_headings`

A dictionary of export column heading overrides in the format `{field_name: heading}`.

`export_filename`

The base file name for the exported listing, without extensions. If unset, the model's `db_table` will be used instead.

`search_fields`

The fields to use for the search in the index view. If set to `None` and `search_backend_name` is set to use a Wagtail search backend, the `search_fields` attribute of the model will be used instead.

`search_backend_name`

The name of the Wagtail search backend to use for the search in the index view. If set to a falsy value, the search will fall back to use Django's QuerySet API.

`copy_view_enabled = True`

Whether to enable the copy view. Defaults to `True`.

`inspect_view_enabled = False`

Whether to enable the inspect view. Defaults to `False`.

`inspect_view_fields = []`

The model fields or attributes to display in the inspect view.

If the field has a corresponding `get_FOO_display()` method on the model, the method's return value will be used instead.

If you have `wagtail.images` installed, and the field's value is an instance of `wagtail.images.models.AbstractImage`, a thumbnail of that image will be rendered.

If you have `wagtail.documents` installed, and the field's value is an instance of `wagtail.documents.models.AbstractDocument`, a link to that document will be rendered, along with the document title, file extension and size.

inspect_view_fields_exclude = []

The fields to exclude from the inspect view.

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`.

usage_view_class = <class 'wagtail.admin.views.generic.usage.UsageView'>

The view class to use for the usage view; must be a subclass of `wagtail.admin.views.generic.usage.UsageView`.

history_view_class = <class

'wagtail.admin.views.generic.history.HistoryView'>

The view class to use for the history view; must be a subclass of `wagtail.admin.views.generic.history.HistoryView`.

copy_view_class = <class 'wagtail.admin.views.generic.models.CopyView'>

The view class to use for the copy view; must be a subclass of `wagtail.admin.views.generic.CopyView`.

inspect_view_class = <class

'wagtail.admin.views.generic.models.InspectView'>

The view class to use for the inspect view; must be a subclass of `wagtail.admin.views.generic.InspectView`.

template_prefix = ''

The prefix of template names to look for when rendering the admin views.

index_template_name

A template to be used when rendering `index_view`.

Default: if `template_prefix` is specified, an `index.html` template in the prefix directory and its `{app_label}/{model_name}/` or `{app_label}/` subdirectories will be used. Otherwise, the `index_view_class.template_name` will be used.

index_results_template_name

A template to be used when rendering `index_results_view`.

Default: if `template_prefix` is specified, a `index_results.html` template in the prefix directory and its `{app_label}/{model_name}/` or `{app_label}/` subdirectories will be used. Otherwise, the `index_view_class.results_template_name` will be used.

create_template_name

A template to be used when rendering add_view.

Default: if `template_prefix` is specified, a `create.html` template in the prefix directory and its `{app_label}/{model_name}/` or `{app_label}/` subdirectories will be used. Otherwise, the `add_view_class.template_name` will be used.

edit_template_name

A template to be used when rendering edit_view.

Default: if `template_prefix` is specified, an `edit.html` template in the prefix directory and its `{app_label}/{model_name}/` or `{app_label}/` subdirectories will be used. Otherwise, the `edit_view_class.template_name` will be used.

delete_template_name

A template to be used when rendering delete_view.

Default: if `template_prefix` is specified, a `confirm_delete.html` template in the prefix directory and its `{app_label}/{model_name}/` or `{app_label}/` subdirectories will be used. Otherwise, the `delete_view_class.template_name` will be used.

history_template_name

A template to be used when rendering history_view.

Default: if `template_prefix` is specified, a `history.html` template in the prefix directory and its `{app_label}/{model_name}/` or `{app_label}/` subdirectories will be used. Otherwise, the `history_view_class.template_name` will be used.

inspect_template_name

A template to be used when rendering inspect_view.

Default: if `template_prefix` is specified, an `inspect.html` template in the prefix directory and its `{app_label}/{model_name}/` or `{app_label}/` subdirectories will be used. Otherwise, the `inspect_view_class.template_name` will be used.

ModelViewSetGroup

class wagtail.admin.viewsets.model.ModelViewSetGroup

A container for grouping together multiple `ModelViewSet` instances.

All attributes and methods from `ViewSetGroup` are available.

menu_label

The displayed label used for the menu item.

If unset, defaults to the title-cased version of the model's `app_label` from the first viewset.

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 = <object object>  
Number of results to show per page  
  
preserve_url_parameters = ['multiple']  
A list of URL query parameters that should be passed on unmodified as part of any links or form submissions within the chooser modal workflow.  
  
url_filter_parameters = []  
A list of URL query parameters that, if present in the url, should be applied as filters to the queryset. (These should also be listed in preserve_url_parameters.)  
  
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.  
  
chosen_multiple_view_class = <class  
'wagtail.admin.views.generic.chooser.ChosenMultipleView'>  
The view class used after multiple items have been chosen; must be a subclass of wagtail.admin.views.generic.chooser.ChosenMultipleView.  
  
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

```
get_object_list()
```

Returns a queryset of objects that are available to be chosen. By default, all instances of `model` are returned.

SnippetViewSet

```
class wagtail.snippets.views.snippets.SnippetViewSet(**kwargs)
```

A viewset that instantiates the admin views for snippets.

All attributes and methods from [ModelViewSet](#) are available.

For more information on how to use this class, see [Customizing admin views for snippets](#).

```
model = None
```

The model class to be registered as a snippet with this viewset.

chooser_per_page = 10

The number of items to display in the chooser view. Defaults to 10.

admin_url_namespace = None

The URL namespace to use for the admin views. If left unset, wagtailsnippets_{app_label}_{model_name} is used instead.

Deprecated - the preferred attribute to customise is [url_namespace](#).

base_url_path = None

The base URL path to use for the admin views. If left unset, snippets/{app_label}/{model_name} is used instead.

Deprecated - the preferred attribute to customise is [url_prefix](#).

chooser_admin_url_namespace = None

The URL namespace to use for the chooser admin views. If left unset, wagtailsnippetchoosers_{app_label}_{model_name} is used instead.

chooser_base_url_path = None

The base URL path to use for the chooser admin views. If left unset, snippets/choose/{app_label}/{model_name} is used instead.

index_view_class = <class 'wagtail.snippets.views.snippets.IndexView'>

The view class to use for the index view; must be a subclass of wagtail.snippets.views.snippets.IndexView.

add_view_class = <class 'wagtail.snippets.views.snippets.CreateView'>

The view class to use for the create view; must be a subclass of wagtail.snippets.views.snippets.CreateView.

edit_view_class = <class 'wagtail.snippets.views.snippets.EditView'>

The view class to use for the edit view; must be a subclass of wagtail.snippets.views.snippets.EditView.

delete_view_class = <class 'wagtail.snippets.views.snippets.DeleteView'>

The view class to use for the delete view; must be a subclass of wagtail.snippets.views.snippets.DeleteView.

usage_view_class = <class 'wagtail.snippets.views.snippets.UsageView'>

The view class to use for the usage view; must be a subclass of wagtail.snippets.views.snippets.UsageView.

history_view_class = <class 'wagtail.snippets.views.snippets.HistoryView'>

The view class to use for the history view; must be a subclass of wagtail.snippets.views.snippets.HistoryView.

copy_view_class = <class 'wagtail.snippets.views.snippets.CopyView'>

The view class to use for the copy view; must be a subclass of wagtail.snippet.views.snippets.CopyView.

inspect_view_class = <class 'wagtail.snippets.views.snippets.InspectView'>

The view class to use for the inspect view; must be a subclass of wagtail.snippets.views.snippets.InspectView.

**revisions_view_class = <class
'wagtail.snippets.views.snippets.PreviewRevisionView'>**

The view class to use for previewing revisions; must be a subclass of wagtail.snippets.views.snippets.PreviewRevisionView.

revisions_revert_view_class

The view class to use for reverting to a previous revision.

By default, this class is generated by combining the edit view with `wagtail.admin.views.generic.mixins.RevisionsRevertMixin`. As a result, this class must be a subclass of `wagtail.snippets.views.snippets.EditView` and must handle the reversion correctly.

**revisions_compare_view_class = <class
'wagtail.snippets.views.snippets.RevisionsCompareView'>**

The view class to use for comparing revisions; must be a subclass of `wagtail.snippets.views.snippets.RevisionsCompareView`.

**revisions_unschedule_view_class = <class
'wagtail.snippets.views.snippets.RevisionsUnscheduleView'>**

The view class to use for unscheduling revisions; must be a subclass of `wagtail.snippets.views.snippets.RevisionsUnscheduleView`.

**unpublish_view_class = <class
'wagtail.snippets.views.snippets.UnpublishView'>**

The view class to use for unpublishing a snippet; must be a subclass of `wagtail.snippets.views.snippets.UnpublishView`.

**preview_on_add_view_class = <class
'wagtail.snippets.views.snippets.PreviewOnCreateView'>**

The view class to use for previewing on the create view; must be a subclass of `wagtail.snippets.views.snippets.PreviewOnCreateView`.

**preview_on_edit_view_class = <class
'wagtail.snippets.views.snippets.PreviewOnEditView'>**

The view class to use for previewing on the edit view; must be a subclass of `wagtail.snippets.views.snippets.PreviewOnEditView`.

lock_view_class = <class 'wagtail.snippets.views.snippets.LockView'>

The view class to use for locking a snippet; must be a subclass of `wagtail.snippets.views.snippets.LockView`.

unlock_view_class = <class 'wagtail.snippets.views.snippets.UnlockView'>

The view class to use for unlocking a snippet; must be a subclass of `wagtail.snippets.views.snippets.UnlockView`.

**chooser_viewset_class = <class
'wagtail.snippets.views.chooser.SnippetChooserViewSet'>**

The ViewSet class to use for the chooser views; must be a subclass of `wagtail.snippets.views.chooser.SnippetChooserViewSet`.

get_queryset(*request*)

Returns a QuerySet of all model instances to be shown on the index view. If `None` is returned (the default), the logic in `index_view.get_base_queryset()` will be used instead.

get_edit_handler()

Like `ModelViewSet.get_edit_handler()`, but falls back to extracting panel definitions from the model class if no edit handler is defined.

get_index_template()

Returns a template to be used when rendering `index_view`. If a template is specified by the `index_template_name` attribute, that will be used. Otherwise, a list of preferred template names are returned.

Deprecated - the preferred way to customise this is to define an `index_template_name` property.

get_index_results_template()

Returns a template to be used when rendering `index_results_view`. If a template is specified by the `index_results_template_name` attribute, that will be used. Otherwise, a list of preferred template names are returned.

Deprecated - the preferred way to customise this is to define an `index_results_template_name` property.

get_create_template()

Returns a template to be used when rendering `add_view`. If a template is specified by the `create_template_name` attribute, that will be used. Otherwise, a list of preferred template names are returned.

Deprecated - the preferred way to customise this is to define a `create_template_name` property.

get_edit_template()

Returns a template to be used when rendering `edit_view`. If a template is specified by the `edit_template_name` attribute, that will be used. Otherwise, a list of preferred template names are returned.

Deprecated - the preferred way to customise this is to define an `edit_template_name` property.

get_delete_template()

Returns a template to be used when rendering `delete_view`. If a template is specified by the `delete_template_name` attribute, that will be used. Otherwise, a list of preferred template names are returned.

Deprecated - the preferred way to customise this is to define a `delete_template_name` property.

get_history_template()

Returns a template to be used when rendering `history_view`. If a template is specified by the `history_template_name` attribute, that will be used. Otherwise, a list of preferred template names are returned.

Deprecated - the preferred way to customise this is to define a `history_template_name` property.

get_inspect_template()

Returns a template to be used when rendering `inspect_view`. If a template is specified by the `inspect_template_name` attribute, that will be used. Otherwise, a list of preferred template names are returned.

Deprecated - the preferred way to customise this is to define an `inspect_template_name` property.

get_admin_url_namespace()

Returns the URL namespace for the admin URLs for this model.

Deprecated - the preferred way to customise this is to define a `url_namespace` property.

get_admin_base_path()

Returns the base path for the admin URLs for this model. The returned string must not begin or end with a slash.

Deprecated - the preferred way to customise this is to define a `url_prefix` property.

get_chooser_admin_url_namespace()

Returns the URL namespace for the chooser admin URLs for this model.

get_chooser_admin_base_path()

Returns the base path for the chooser admin URLs for this model. The returned string must not begin or end with a slash.

SnippetViewSetGroup**class wagtail.snippets.views.snippets.SnippetViewSetGroup**

A container for grouping together multiple *SnippetViewSet* instances.

All attributes and methods from *ModelViewSetGroup* are available.

PageListingViewSet**class wagtail.admin.viewsets.pages.PageListingViewSet (name=None, **kwargs)**

A viewset to present a flat listing of all pages of a specific type. All attributes and methods from *ViewSet* are available. For more information on how to use this class, see *Custom page listings*.

model = <class 'wagtail.models.Page'>

Required; the page model class that this viewset will work with.

index_view_class = <class 'wagtail.admin.views.pages.listing.IndexView'>

The view class to use for the index view; must be a subclass of `wagtail.admin.views.pages.listing.IndexView`.

choose_parent_view_class = <class 'wagtail.admin.views.pages.choose_parent.ChooseParentView'>

The view class to use for choosing the parent page when creating a new page of this page type.

columns = [<wagtail.admin.ui.tables.pages.BulkActionsColumn: bulk_actions>, <wagtail.admin.ui.tables.pages.PageTitleColumn: title>, <wagtail.admin.ui.tables.pages.ParentPageColumn: parent>, <wagtail.admin.ui.tables.DateColumn: latest_revision_created_at>, <wagtail.admin.ui.tables.pages.PageStatusColumn: status>]

A list of `wagtail.admin.ui.tables.Column` instances for the columns in the listing.

filterset_class = <class 'wagtail.admin.views.pages.listing.PageFilterSet'>

A subclass of `wagtail.admin.filters.WagtailFilterSet`, which is a subclass of `django_filters.FilterSet`. This will be passed to the `filterset_class` attribute of the index view.

1.7 Deployment & hosting

1.7.1 Deploying Wagtail with Fly.io + Backblaze

This tutorial will use two platforms to deploy your site. You'll host your site on [fly.io](#) and serve your site's images on [Backblaze](#).

You can use [fly.io](#) to host your site and serve your images. However, storing your images on a platform other than the one hosting your site provides better performance, security, and reliability.

Note

In this tutorial, you'll see "yourname" several times. Replace it with a name of your choice.

Setup Backblaze B2 Cloud Storage

To serve your images, set up a Backblaze B2 storage following these steps:

1. Visit the Backblaze [website](#) in your browser.
2. Click **Products** from the top navigation and then select **B2 Cloud Storage** from the dropdown.
3. Sign up to Backblaze B2 Cloud Storage by following these steps:
 - a. Enter your email address and password.
 - b. Select the appropriate region.
 - c. Click **Sign Up Now**.
4. Verify your email by following these steps:
 - a. Go to **Account > My Settings** in your side navigation.
 - b. Click **Verify Email** in the **Security section**.
 - c. Enter your sign-up email address and then click **Send Code**.
 - d. Check your email inbox or spam folder for the verification email.
 - e. Click the verification link or use the verification code.
5. Create a Bucket by going to **B2 Cloud Storage > Bucket** and clicking **Create a Bucket**.
6. Go to **B2 Cloud Storage > Bucket** and then click **Create a Bucket**.
7. Add your Bucket information as follows:

Bucket information	Instruction
Bucket Unique Name	Use a unique Bucket name. For example, <i>yourname-wagtail-portfolio</i>
Files in Bucket are	Select Public
Default Encryption	Select Disable
Object Lock	Select Disable

8. Click **Create a Bucket**.

Link your site to Backblaze B2 Cloud Storage

After setting up your Backblaze B2 Cloud Storage, you must link it to your portfolio site.

Start by creating a `.env.production` file at the root of your project directory. At this stage, your project directory should look like this:

```
mysite/
├── base
├── blog
├── home
├── media
└── mysite
    ├── portfolio
    └── search
    ├── .dockerignore
    ├── .gitignore
    ├── .env.production
    ├── Dockerfile
    └── manage.py
```

(continues on next page)

(continued from previous page)

```
└── mysite/
    └── requirements.txt
```

Now add the following environment variables to your `.env.production` file:

```
AWS_STORAGE_BUCKET_NAME=
AWS_S3_ENDPOINT_URL=https://
AWS_S3_REGION_NAME=
AWS_S3_ACCESS_KEY_ID=
AWS_S3_SECRET_ACCESS_KEY=
DJANGO_ALLOWED_HOSTS=
DJANGO_CSRF_TRUSTED_ORIGINS=https://
DJANGO_SETTINGS_MODULE=mysite.settings.production
```

Fill in your Backblaze B2 bucket information

The next step is to provide values for your environment variables. In your `.env.production` file, use your Backblaze B2 bucket information as values for your environment variables as follows:

Environment variable	Instruction
AWS_STORAGE_BUCKET_NAME	Use your Backblaze B2 bucket name
AWS_S3_ENDPOINT_URL	Use the Backblaze B2 endpoint URL. For example, https://s3.us-east-005.backblazeb2.com
AWS_S3_REGION_NAME	Determine your bucket's region from the endpoint URL. For example, if your endpoint URL is s3.us-east-005.backblazeb2.com , then your bucket's region is <code>us-east-005</code>
AWS_S3_ACCESS_KEY_ID	Leave this empty for now
AWS_S3_SECRET_ACCESS_KEY	Leave this empty for now
DJANGO_ALLOWED_HOSTS	Leave this empty for now
DJANGO_CSRF_TRUSTED_ORIGINS	Use <code>https://</code>
DJANGO_SETTINGS_MODULE	Use <code>mysite.settings.production</code>

In the preceding table, you didn't provide values for your `AWS_S3_ACCESS_KEY_ID`, `AWS_S3_SECRET_ACCESS_KEY`, and `DJANGO_ALLOWED_HOSTS`.

To get values for your `AWS_S3_ACCESS_KEY_ID` and `AWS_S3_SECRET_ACCESS_KEY`, follow these steps:

1. Log in to your Backblaze B2 account.
2. Navigate to **Account > Application Keys**.
3. Click **Add a New Application Key**.
4. Configure the application key settings as follows:

Setting	Instruction
Name of Key	Provide a unique name
Allow access to Buckets	Choose the Backblaze B2 bucket you created earlier
Type of Access	Select Read and Write
Allow List All Bucket Names	Leave this unticked
File name prefix	Leave field empty
Duration (seconds)	Leave field empty

5. Click **Create New Key**.

Now, use your `keyID` as the value of `AWS_S3_ACCESS_KEY_ID` and `applicationKey` for `AWS_S3_SECRET_ACCESS_KEY` in your `.env.production` file:

Environment variable	Instruction
<code>AWS_S3_ACCESS_KEY_ID</code>	Use your keyID
<code>AWS_S3_SECRET_ACCESS_KEY</code>	Use your applicationKey

At this stage, the content of your `.env.production` file looks like this:

```
AWS_STORAGE_BUCKET_NAME=yourname-wagtail-portfolio
AWS_S3_ENDPOINT_URL=https://s3.us-east-005.backblazeb2.com
AWS_S3_REGION_NAME=us-east-005
AWS_S3_ACCESS_KEY_ID=your Backblaze keyID
AWS_S3_SECRET_ACCESS_KEY=your Backblaze applicationKey
DJANGO_ALLOWED_HOSTS=
DJANGO_CSRF_TRUSTED_ORIGINS=https://
DJANGO_SETTINGS_MODULE=mysite.settings.production
```

Note

The Backblaze B2 storage uses AWS and S3 because it works like Amazon Web Services' S3.

Do not commit or share your `.env.production` file. Anyone with the variables can access your site.

If you lost your secret application key, create a new key following the preceding instructions.

For more information on how to set up your Backblaze B2 Cloud Storage, read the [Backblaze B2 Cloud Storage Documentation](#).

Set up Fly.io

Now that you've linked your site to your Backblaze storage, it's time to set up Fly.io to host your site.

To set up your Fly.io account, follow these steps:

1. Visit [Fly.io](#) in your browser.
2. Click **Sign Up**.
3. Sign up using your GitHub account, Google account, or the email option.
4. Check your email inbox for the verification link to verify your email.

Note

If your email verification fails, go to your Fly.io [Dashboard](#) and try again.

5. Go to **Dashboard > Billing** and click **Add credit card** to add your credit card.

Note

Adding your credit card allows you to create a project in Fly.io. Fly.io won't charge you after adding your credit card.

6. Install `flyctl` by navigating to your project directory and then running the following command in your terminal:

On macOS:

```
# If you have the Homebrew package manager installed, run the following command:
brew install flyctl

# If you don't have the Homebrew package manager installed, run the following command:
curl -L https://fly.io/install.sh | sh
```

On Linux:

```
curl -L https://fly.io/install.sh | sh
```

On Windows, navigate to your project directory on **PowerShell**, activate your environment and run the following command:

```
pwsh -Command "iwr https://fly.io/install.ps1 -useb | iex"
```

Note

If you get an error on Windows saying the term `pwsh` is not recognized, install **PowerShell MSI** and then rerun the preceding Windows command.

7. Sign in to your Fly.io by running the following command:

```
fly auth login
```

If you use Microsoft WSL, then run:

```
ln -s /usr/bin/wslview /usr/local/bin/xdg-open
```

Note

If you successfully install `flyctl` but get an error saying “`fly` is not recognized” or “`flyctl: command not found error`”, then you must add `flyctl` to your PATH. For more information, read [Getting `flyctl: command not found error` post install](#).

8. Create your Fly.io project by running `fly launch`. Then press `y` to configure the settings.

9. You will be taken to an admin screen on `fly.io`. Fill out the fields as follows:

Field	Instruction
Choose a region for deployment	Select the region closest to the <code>AWS_S3_REGION_NAME</code> in your <code>env.production</code> file.
CPU & Memory	VM Size - shared-cpu-1x VM Memory - 512 MB
Database	Fly Postgres - choose smallest option

click confirm **Confirm settings**

Note

Not creating the database directly with the application leads to the app and the database not connected. If the app is going to be launched again using `fly launch`, it's recommended to create a new database with the launch of the app through the web UI.

10. Back in your terminal, answer the resulting prompt questions as follows:

Question	Instruction
Overwrite ".../.dockerignore"?	Enter y
Overwrite ".../Dockerfile"?	Enter y

The `fly launch` command creates two new files, `Dockerfile` and `fly.toml`, in your project directory.

If you use a third-party app terminal like the Visual Studio Code terminal, you may get an error creating your Postgres database. To rectify this error, follow these steps:

1. Delete `fly.toml` file from your project directory.
2. Go to your Fly.io account in your browser and click **Dashboard**.
3. Click the created app in your **Apps** list.
4. Click **Settings** in your side navigation.
5. Click **Delete app**.
6. Enter the name your app.
7. Click **Yes delete it**.
8. Repeat steps 3, 4, 5, 6, and 7 for all apps in your **Apps** list.
9. Run the `fly launch` command in your built-in terminal or PowerShell MSI on Windows.

Customize your site to use Fly.io

Now, you must configure your portfolio site for the final deployment.

The `fly launch` command creates two new files, `Dockerfile` and `fly.toml`, in your project directory.

Add the following to your `.gitignore` file to make Git ignore your environment files:

```
.env*
```

Also, add the following to your `.dockerignore` file to make Docker ignore your environment and media files:

```
.env*
media
```

Configure your Fly.io to use 1 worker. This allows your site to work better with Fly.io's low memory allowance. To do this, modify the last line of your `Dockerfile` as follows:

```
CMD ["gunicorn", "--bind", ":8000", "--workers", "1", "mysite.wsgi"]
```

Also, check if your `fly.toml` file has the following:

```
[deploy]
release_command = "python manage.py migrate --noinput"
```

Your `fly.toml` file should look as follows:

```
app = "yourname-wagtail-portfolio"
primary_region = "lhr"
console_command = "/code/manage.py shell"

[build]

# add the deploy command:
[deploy]
release_command = "python manage.py migrate --noinput"

[env]
PORT = "8000"

[http_service]
internal_port = 8000
force_https = true
auto_stop_machines = true
auto_start_machines = true
min_machines_running = 0
processes = ["app"]

[[statics]]
guest_path = "/code/static"
url_prefix = "/static/"
```

Now add your production dependencies by replacing the content of your `requirements.txt` file with the following:

```
Django>=4.2,<4.3
wagtail==5.1.1
gunicorn>=21.2.0,<22.0.0
psycopg[binary]>=3.1.10,<3.2.0
dj-database-url>=2.1.0,<3.0.0
whitenoise>=5.0,<5.1
django-storages[s3]>=1.14.0,<2.0.0
```

The preceding dependencies ensure that the necessary tools and libraries are in place to run your site successfully on the production server. The following are the explanations for the dependencies you may be unaware of:

1. gunicorn is a web server that runs your site in Docker.
2. psycopg is a PostgreSQL adapter that connects your site to a PostgreSQL database.
3. dj-database-url is a package that simplifies your database configurations and connects to your site to a PostgreSQL database.
4. whitenoise is a Django package that serves static files.
5. django-storages is a Django library that handles your file storage and connects to your Backblaze B2 storage.

Replace the content of your `mysite/settings/production.py` file with the following:

```
import os
import random
import string
import dj_database_url

from .base import *
```

(continues on next page)

(continued from previous page)

```
DEBUG = False

DATABASES = {
    "default": dj_database_url.config(
        conn_max_age=600,
        conn_health_checks=True
    )
}

SECRET_KEY = os.environ["SECRET_KEY"]

SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTO", "https")

SECURE_SSL_REDIRECT = True

ALLOWED_HOSTS = os.getenv("DJANGO_ALLOWED_HOSTS", "*").split(", ")

CSRF_TRUSTED_ORIGINS = os.getenv("DJANGO_CSRF_TRUSTED_ORIGINS", "").split(", ")

EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"

MIDDLEWARE.append("whitenoise.middleware.WhiteNoiseMiddleware")
STORAGES["staticfiles"]["BACKEND"] = "whitenoise.storage."
    ↪CompressedManifestStaticFilesStorage"

if "AWS_STORAGE_BUCKET_NAME" in os.environ:
    AWS_STORAGE_BUCKET_NAME = os.getenv("AWS_STORAGE_BUCKET_NAME")
    AWS_S3_REGION_NAME = os.getenv("AWS_S3_REGION_NAME")
    AWS_S3_ENDPOINT_URL = os.getenv("AWS_S3_ENDPOINT_URL")
    AWS_S3_ACCESS_KEY_ID = os.getenv("AWS_S3_ACCESS_KEY_ID")
    AWS_S3_SECRET_ACCESS_KEY = os.getenv("AWS_S3_SECRET_ACCESS_KEY")

    INSTALLED_APPS.append("storages")

    STORAGES["default"]["BACKEND"] = "storages.backends.s3boto3.S3Boto3Storage"

    AWS_S3_OBJECT_PARAMETERS = {
        'CacheControl': 'max-age=86400',
    }

LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
        },
    },
    "loggers": {
        "django": {
            "handlers": ["console"],
            "level": os.getenv("DJANGO_LOG_LEVEL", "INFO"),
        },
    },
}

WAGTAIL_REDIRECTS_FILE_STORAGE = "cache"
```

(continues on next page)

(continued from previous page)

```
try:
    from .local import *
except ImportError:
    pass
```

The explanation of some of the code in your `mysite/settings/production.py` file is as follows:

1. `DEBUG = False` turns off debugging for the production environment. It's important for security and performance.
2. `SECRET_KEY = os.environ["SECRET_KEY"]` retrieves the project's secret key from your environment variable.
3. `SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTO", "https")` ensures that Django can detect a secure HTTPS connection if you deploy your site behind a reverse proxy like Heroku.
4. `SECURE_SSL_REDIRECT = True` enforces HTTPS redirect. This ensures that all connections to the site are secure.
5. `ALLOWED_HOSTS = os.getenv("DJANGO_ALLOWED_HOSTS", "*").split(",")` defines the hostnames that can access your site. It retrieves its values from the `DJANGO_ALLOWED_HOSTS` environment variable. If no specific hosts are defined, it defaults to allowing all hosts.
6. `EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"` configures your site to use the console email backend. You can configure this to use a proper email backend for sending emails.
7. `WAGTAIL_REDIRECTS_FILE_STORAGE = "cache"` configures the file storage for Wagtail's redirects. Here, you set it to use cache.

Now, complete the configuration of your environment variables by modifying your `.env.production` file as follows:

Environment variable	Instruction
<code>DJANGO_ALLOWED_HOSTS</code>	This must match your fly.io project name. For example, <code>yourname-wagtail-portfolio.fly.dev</code>
<code>DJANGO_CSRF_TRUSTED_ORIGINS</code>	This must match your project's domain name. For example, <code>https://yourname-wagtail-portfolio.fly.dev</code>

The content of your `.env.production` file should now look like this:

```
AWS_STORAGE_BUCKET_NAME=yourname-wagtail-portfolio
AWS_S3_ENDPOINT_URL=https://s3.us-east-005.backblazeb2.com
AWS_S3_REGION_NAME=us-east-005
AWS_S3_ACCESS_KEY_ID=your Backblaze keyID
AWS_S3_SECRET_ACCESS_KEY=your Backblaze applicationKey
DJANGO_ALLOWED_HOSTS=yourname-wagtail-portfolio.fly.dev
DJANGO_CSRF_TRUSTED_ORIGINS=https://yourname-wagtail-portfolio.fly.dev
DJANGO_SETTINGS_MODULE=mysite.settings.production
```

Set the secrets for Fly.io to use by running:

```
flyctl secrets import < .env.production
```

On Windows, run the following command in your PowerShell MSI:

```
Get-Content .env.production | flyctl secrets import
```

Finally, deploy your site to Fly.io by running the following command:

```
fly deploy --ha=false
```

Note

Running “`fly deploy`” creates two machines for your app. Using the “`--ha=false`” flag creates one machine for your app.

Congratulations! Your site is now live. However, you must add content to it. Start by creating an admin user for your live site. Run the following command:

```
flyctl ssh console
```

Then run:

```
DJANGO_SUPERUSER_USERNAME=username DJANGO_SUPERUSER_EMAIL=mail@example.com DJANGO_
→SUPERUSER_PASSWORD=password python manage.py createsuperuser --noinput
```

Note

Ensure you replace `username`, `mail@example.com`, and `password` with a username, email address, and password of your choice.

For more information on how to set up your Django project on Fly.io, read [Django on Fly.io](#).

Add content to your live site

All this while, you’ve been adding content to your site in the local environment. Now that your site is live on a server, you must add content to the live site. To add content to your live site, go to `https://yourname-wagtail-portfolio.fly.dev/admin/` in your browser and follow the steps in the following subsections of the tutorial:

- [Add content to your homepage](#)
- [Add your social media links](#)
- [Add footer text](#)
- [Add pages to your site menu](#)
- [Add your contact information](#)
- [Add your resume](#)

Note

If you encounter errors while trying to access your live site in your browser, check your application logs in your Fly.io Dashboard. To check your application logs, click **Dashboard > Apps > yourname-wagtail-portfolio > Monitoring**

1.7.2 Deployment: Under the hood

This doc provides a technical deep-dive into Wagtail hosting concepts. Most likely, you'll want to [choose a hosting provider](#) instead.

Wagtail is built on Django, and so the vast majority of the deployment steps and considerations for deploying Django are also true for Wagtail. We recommend reading Django's “[How to deploy Django](#)” documentation.

Infrastructure Requirements

When designing infrastructure for hosting a Wagtail site, there are a few basic requirements:

WSGI / ASGI server

Django, being a web framework, needs a web server in order to operate. Since most web servers don't natively speak Python, we need an interface to make that communication happen.

Wagtail can be deployed using either [WSGI](#) or [ASGI](#), however Wagtail doesn't natively implement any async views or middleware, so we recommend WSGI.

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 `STORAGES["staticfiles"]` 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 `STORAGES["default"]` setting in conjunction with an add-on package such as `django-storages`.

Security

Any system that allows user-uploaded files is a potential security risk. For example, a user with the ability to upload HTML files could potentially launch a [cross-site scripting attack](#) against a user viewing that file. This may not be a concern if all users with access to the Wagtail admin are fully trusted - for example, a personal site where you are the only editor. With this in mind, Wagtail aims to provide a secure configuration by default, but developers may choose a more permissive setup if they understand the risks, as detailed below.

Images

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 from the `images` subdirectory of `MEDIA_ROOT` (without passing through Django/Wagtail). If [SVG images](#) are enabled, it is possible for a user to upload an SVG file containing scripts that execute when the file is viewed directly; if this is a concern, several approaches for avoiding this are detailed under [Security considerations](#).

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.

Documents

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`. In this case, `WAGTAILDOCS_SERVE_METHOD` defaults to `serve_view`, where Wagtail serves the document through a Django view that enforces privacy checks. This has the following implications:

- **You should block direct access to the `documents` subdirectory of `MEDIA_ROOT` within your web server configuration.** This prevents users from bypassing [collection privacy settings](#) by accessing documents at their direct URL.
- Documents are served as downloads rather than displayed in the browser (unless specified explicitly via `WAGTAILDOCS_INLINE_CONTENT_TYPES`) - this ensures that if the document is a type that can contain scripts (such as HTML or SVG), the browser is prevented from executing them.
- However, since the document is served through the Django application server, this may consume more server resources than serving the document directly from the web server.

The alternative serve methods '`direct`' and '`redirect`' work by serving the documents directly from `MEDIA_ROOT`. This means it is not possible to block direct access to the `documents` subdirectory, and so users may bypass permission checks by accessing the direct URL. Also, in the case that users with access to the Wagtail admin are not fully trusted, you will need to take additional steps to prevent the execution of scripts in documents:

- The `WAGTAILDOCS_EXTENSIONS` setting may be used to restrict uploaded documents to an “allow list” of safe types.
- The web server can be configured to return a `Content-Security-Policy: default-src 'none'` header for files within the `documents` subdirectory, which will prevent the execution of scripts in those files.
- The web server can be configured to return a `Content-Disposition: attachment` header for files within the `documents` subdirectory, which will force the browser to download the file rather than displaying it inline.

If a remote (“cloud”) storage backend is used, the serve method will default to '`redirect`' and the document will be served directly from the cloud storage file url. In this case, users may be able to bypass permission checks, and scripts within documents may be executed (depending on the cloud storage service's configuration). However, the impact of cross-site scripting attacks is reduced, as the document is served from a different domain to the main site.

If these limitations are not acceptable, you may set the `WAGTAILDOCS_SERVE_METHOD` to `serve_view` and ensure that the documents are not publicly accessible using the cloud service's file url.

The steps required to set headers for specific responses will vary, depending on how your Wagtail application is deployed and which storage backend is used. For the '`serve_view` method, a `Content-Security-Policy` header is automatically set for you (unless disabled via `WAGTAILDOCS_BLOCK_EMBEDDED_CONTENT`) to prevent the execution of scripts embedded in documents.

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

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`.

Cache

Wagtail is designed to take advantage of Django's `cache` framework when available to accelerate page loads. The cache is especially useful for the Wagtail admin, which can't take advantage of conventional CDN caching.

Wagtail supports any of Django's cache backend, however we recommend against using one tied to the specific process or environment Django is running (eg `FileBasedCache` or `LocMemCache`).

Deployment tips

Wagtail, and by extension Django, can be deployed in many different ways on many different platforms. There is no "best" way to deploy it, however here are some tips to ensure your site is as stable and maintainable as possible:

Use Django's deployment checklist

Django has a deployment checklist which runs through everything you should have done or should be aware of before deploying a Django application.

Performance optimization

Your production site should be as fast and performant as possible. For tips on how to ensure Wagtail performs as well as possible, take a look at our [performance tips](#).

Deployment examples

Some examples of deployments on a few hosting platforms can be found in [Third-party tutorials](#). This is not a complete list of platforms where Wagtail can run, nor is it necessarily the only way to run Wagtail there.

An example of a production Wagtail site is [guide.wagtail.org](#), which is open-source and runs on Heroku. More information on its hosting environment can be found in [its documentation](#).

If you have successfully installed Wagtail on your platform or infrastructure, please [contribute](#) your notes to this documentation!

Once you've built your Wagtail site, it's time to release it upon the rest of the internet.

Wagtail is built on Django, and so the vast majority of the deployment steps and considerations for deploying Django are also true for Wagtail. We recommend choosing one of the hosting providers listed below.

1.7.3 Choosing a Hosting Provider

Several hosting providers offer varying levels of support for Wagtail. We've organized them into three categories:

- Wagtail-level support (easiest deployment).
- Python-level support (requires some knowledge of WSGI and file storage).
- Infrastructure-level support (requires knowledge of Linux).

1.7.4 Wagtail-Level Support

These hosting providers offer first-class support for Wagtail deployments and installations, designed to make it as easy as possible to run a Wagtail site.

CodeRed Cloud

- Website & pricing: codered.cloud
- Wagtail deployment guide: [CodeRed Wagtail Quickstart](#)
- From the vendor:

CodeRed Cloud is inspired by simplicity and “it just works” philosophy. No special packages or 3rd party services required! Free plans are available, and every plan includes a database, media hosting, daily backups, and more.

Divio

- Website & pricing: divio.com
- Wagtail deployment guide: [Divio Wagtail Setup Guide](#)
- From the vendor:

Divio is a cloud hosting platform designed to simplify the development and deployment of containerized web applications. It integrates smoothly with Wagtail, providing developers with tools to efficiently manage web applications. Divio proactively manages and supports state-of-the-art cloud services, ensuring that your Wagtail applications are scalable, secure, and reliable. The platform's user-friendly interface makes it easy to develop, deploy, manage, and maintain your web applications. With features like automated backups and staging environments, Divio handles the technical infrastructure, allowing you to focus on building and maintaining your Wagtail sites with confidence.

1.7.5 Python-Level Support

These hosting providers offer Python environments as a service. Usually, you will need to configure a WSGI server, file storage for media hosting, and a database.

Fly.io with Backblaze

Read our guide on [deploying to Fly.io](#).

1.7.6 Infrastructure-Level Support

These hosting providers offer the tools needed to run a Linux server, database, file storage, etc. Popular infrastructure providers include: **AWS, Azure, Digital Ocean, Google Cloud, and Linode**.

1.7.7 Others

Some examples of deployments on a few hosting platforms can be found in [Third-party tutorials](#). This is not a complete list of platforms where Wagtail can run, nor is it necessarily the only way to run Wagtail there.

For a technical deep-dive into the many aspects of Wagtail hosting, see [Deployment: Under the hood](#).

Are you a hosting provider who supports Wagtail, and want to add yourself to this list? See if you meet our [requirements for hosting providers](#).

1.8 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.

1.8.1 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".

1.8.2 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.

1.8.3 GitHub discussions

Our [GitHub discussion boards](#) are open for sharing ideas and plans for the Wagtail project.

1.8.4 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](#).

1.8.5 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 @wagtail@fosstodon.org.

1.9 Editor's guide

Wagtail's Editor Guide now has its own website: guide.wagtail.org. This guide 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.

1.10 Contributing

Thank you for your interest in improving Wagtail!

1.10.1 First-time contributors

1. Read this document first.
2. We don't assign tasks. Feel free to pick any issue/task that isn't already being worked on by someone else.
3. Read the [Your first contribution guide](#).

1.10.2 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 that 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 questions 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](#).

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 judgment 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 prioritized 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 endeavor 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 prioritized 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.

1.10.3 Pull requests

If you are just getting started with development and have never contributed to an open-source project, we recommend you read the [Your first contribution guide](#). If you're a confident Python or Django developer, [fork it](#) and read the [developing docs](#) to get stuck in!

We welcome all contributions, whether they solve problems that 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 at 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.

Your first contribution

- [Guide](#)
- [Common questions](#)
- [Helpful links](#)

This page has a step by step guide for how to get started making contributions to Wagtail. It is recommended for developers getting started with open source generally or with only a small amount of experience writing code for shared teams.

This is a long guide - do not worry about following all the steps in one go or doing them perfectly. There are lots of people around in the community to help, but if you can take the time to read and understand yourself you will be a much stronger developer.

Each section has an introduction with an overview and a checklist that can be copied and pasted for you to check through one by one. Get ready to read, there is a lot of reading ahead.

Note

Avoid ‘claiming’ any issues before completing Steps 0-6. This helps you not over-promise what you can contribute and helps the community support you when you are actually ready to contribute. Do not worry about issues ‘running out’ - software development is an endless fractal, there is always more to help with.

Guide

0. Understand your motivations

Before you start contributing to Wagtail, take a moment to think about why you want to do it. If your only goal is to add a “first contribution” to your resume (or if you’re just looking for a quick win) you might be better off doing a boot-camp or an online tutorial.

Contributing to open source projects takes time and effort, but it can also help you become a better developer and learn new skills. However, it’s important to know that it might be harder and slower than following a training course. That said, contributing to open source is worth it if you’re willing to take the time to do things well.

One thing to keep in mind is that “scratching your own itch” can be a great motivator for contributing to open source. If you’re interested in the CMS space or the programming languages used in this project, you’ll be more likely to stick with it over the long term.

1. Understanding what Wagtail is

Before you start contributing to Wagtail, it’s important to understand what it is and how it works. Wagtail is a content management system (CMS) used for building websites. Unlike other CMSs, it requires some development time to build up the models and supporting code to use as a CMS. Additionally, Wagtail is built on top of another framework called Django, which is a Python web framework. This might be confusing at first, but it provides a powerful way to create custom systems for developers to build with.

To get started, we recommend reading the [the Zen of Wagtail](#), which provides a good introduction to the project. You might also want to read the [Django overview](#) to understand what Django provides. To get a sense of how Wagtail fits into the CMS landscape, you can search online for articles that compare WordPress to Wagtail or list the top open source

CMSs. Finally, reading some of the [Wagtail Guide](#) will give you a better understanding of how the CMS works for everyday users.

Note

Below is a checklist. There are many like these you can copy for yourself as you progress through this guide.

- [] Read the Zen of Wagtail
- [] Read the Django Overview
- [] Search online for one or two articles that 'compare Wordpress to Wagtail' or
→ 'top ten open source CMS' and read about the CMS landscape
- [] Read some of the Wagtail Guide

2. Joining the community

Make an account on [Wagtail Slack](#) server, this is the way many of the community interact day to day. Introduce yourself on `#new-contributors` and join some of the other channels, remember to keep your intro short and be nice to others. After this, join [GitHub](#) and set up your profile. It's really helpful to the community if your name can be added to your profiles in both communities and an image. It does not have to be your public name or a real image if you want to keep that private but please avoid it staying as the 'default avatar'.

You may also want to join StackOverflow and follow the [Wagtail tag](#), this way you can upvote great answers to questions you have or maybe consider contributing answers yourself. Before you dive in, take a moment to review the [community guidelines](#) to get a grasp on the expectations for participation.

Checklist

- [] Read the community guidelines
- [] Join GitHub
- [] Add your preferred name and image to your GitHub profile
- [] Join Slack
- [] Add your preferred name, timezone and image to your Slack profile
- [] Introduce yourself in `'#new-contributors'` in Slack
- [] Join the `'#support'` channel in Slack
- [] Optional Join StackOverflow

3. Before contributing code

Firstly, it is important to be able to understand how to **build with Wagtail** before you can understand how to contribute to Wagtail. Take the time to do the full [Wagtail getting started tutorial](#) without focusing yet on how to contribute code but instead on how to use Wagtail to build your own basic demo website. This will require you to have Python and other dependencies installed on your machine and may not be easy the first time, but keep at it and ask questions if you get stuck.

Remember that there are many other ways to contribute, such as answering questions in StackOverflow or `#support`, contributing to one of the [other packages](#) or even the [Wagtail user guide](#). Sometimes, it's best to get started with a non-code contribution to get a feel for Wagtail's code or the CMS interface.

Issue tracking, reading and triage is a critical part of contributing code and it is recommended that you read the [issue tracking guide](#) in full. This will help you understand how to find issues to work on and how to support the team with triaging issues.

Note

Take the time to **read** the issue and links before adding new comments or questions. Remember, it's not time to 'claim' any issues yet either.

Checklist

- [] Do the Wagtail tutorial
- [] Look at the Wagtail organization on GitHub, take note of **any** interesting ↵**projects**
- [] Read through the Issue Tracking section **in** the docs
- [] Give a go at a non-code contribution

4. Setting up your development environment

Many contribution sections gloss over the mammoth task that can be a single line in the documentation similar to "fork the code and get it running locally". This, on its own, can be a daunting task if you are just getting started. This is why it's best to have done the Wagtail tutorial before this step so you have run into and hopefully solved many of the normal developer environment issues.

First, create a clone of Wagtail on your GitHub account (see below for more details).

Note

Do not try to move past this step until you have a working **bakerydemo** code locally and a clone of the Wagtail repo that you can edit. When editing the Wagtail core code (both HTML and JavaScript) you should be able to refresh the site running locally and see the changes.

Read (in full) the *Development guide*. This will walk you through how to get your code running locally so you can contribute. It's strongly recommended to use the Vagrant or Docker setups, especially if you are working on Windows.

Note

When developing, it's recommended that you always read the **latest** version of the docs. Not the **stable** version. This is because it will better reflect what's on the **main** code branch.

Checklist

- [] Install `git` (if not on your machine)
- [] Install a code editor/IDE (we recommend VSCode)
- [] Install the dependencies set out in the development guide
- [] Follow the development guide
- [] Make a change to the `wagtail/admin/templates/wagtailadmin/home.html` template ↵**file** and confirm you can see the changes on the Wagtail dashboard (home) page
- [] Add a `console.log` statement to `client/src/entrypoints/admin/wagtailadmin.js` ↵**and** confirm you can see the logging in the browser

Aside: Understanding Git and GitHub

git is the version control tool, it is something you install on your device and runs usually in the command line (terminal) or via some GUI application.

GitHub & GitLab are two prominent websites that provide a web user interface for repositories using git, Wagtail uses GitHub.

Mozilla has a great guide that helps to explain [Git and GitHub](#).

How to clone a remote repository and what that actually even means:

- On GitHub, you will not be allowed to directly create branches or changes in a repository (project) that you do not have access to.
- However, you can make a copy (clone) of this repository using your own account, this clone will have all the branches and history that the original repository had.
- This is also called ‘fork’ in some cases, as your repository will be a branch of its own that forks the original repository.
- See the [GitHub docs explain forking](#).
- See [Atlassian’s docs on git clone](#) for more details.

5. Finding an issue

Hopefully, at this point, you have a good sense of the purpose of the project and are still keen to contribute.

Once you have the code forked and running locally, you will probably want to start looking for what to contribute.

Finding something to contribute is not always easy, especially if you are new to the project. Once you have a few candidate issues to investigate, be sure to read the entire issue description, all comments and all linked issues or pull requests. You may often find that someone else has started or finished the issue. Sometimes there are clarifications in the comments about how to approach the problem or whether the problem is even something worth solving.

If an issue has a pull request linked and not yet merged read that pull request and the discussion on it. Maybe the previous contributor got stuck or lost momentum, in which case you could pick up where they left off (assuming it’s been enough time). If you have an idea about how to solve a problem, just add a comment with a suggestion, we should all aim to help each other out.

If the issue is labelled `good-first-issue`, that usually means it is smaller and good for first time contributors. There are no problems with finding other issues to contribute to, have a search around and see what you can help with.

Finally, before ‘claiming’ check you can do the following;

Checklist (for a candidate issue)

- [] Confirm that there is not someone actively working on it (no recent PR or [comments](#) in the last ~2 months)
 - [] Ensure you can reproduce the problem/scenario in your local version of Wagtail
 - [] Ensure that you feel confident to write a unit test (if it's a code change) [to validate that the solution ****is**** implemented](#)

6. Contributing a solution

Important: If an issue is not assigned to anyone and doesn't already have a pull request, feel free to work on it, **no need to ask “please assign me this issue”**. We only use GitHub's issue assignment feature to assign certain issues to members of the Wagtail core team.

If you feel ready to contribute a solution, now is a good time to add a comment to the issue describing your intention to do so, to prevent duplicating efforts. Instead of asking “please assign me this issue”, write something similar to the following:

 Note

I have been able to reproduce this problem/scenario. I am planning to work on this, my rough solution is (...explain).

If it's just a documentation request, you may refine this comment to explain where you plan to add the section in the documentation.

Create a fresh branch for your contributions

Before writing any code, take a moment to get your `git` hat on. When you clone the project locally, you will be checked out at the `main` branch. This branch is not suitable for you to make your changes on. It is meant to be the branch that tracks the core development of the project.

Instead, take a moment to create a `new branch`. You can use the command line or install one of the many great git GUI tools. Don't listen to anyone that says you're not doing it right unless you use the command line. Reduce the things you need to learn today and focus on the `git` command line interface later. If you have a Mac, I recommend [Fork](#), otherwise, the [GitHub GUI](#) is good enough.

This new branch name should have some context as to what you are fixing and if possible the issue number being fixed. For example `git checkout -b 'feature/1234-add-unit-tests-for-inline-panel'`. This branch name uses `/` to represent a folder and also has the issue number 1234, finally, it uses lower-kebab-case with a short description of the issue.

 Note

You may find that your editor has some handy Git tooling and will often be able to tell you what branch you are on or whether you have any changes staged. For example, see VS Code's support for Git.

Keep the changes focused

As a developer, it is easy to get distracted, maybe a typo here or white space that does not feel ‘right’ there. Sometimes, even our editor gets distracted and starts adding line breaks at the end of files as we save or it formats code without our consent due to configuration from a different project.

These added changes that are not the primary goal or not strictly required by the project's set-up are noise. This noise makes it harder to review the pull request and also can create confusion for future developers that see these commits and wonder how it relates to the bug that was fixed.

When you go to stage changes, only stage the parts you need or at least review the changes and ‘undo’ them before you save the commits.

If you do find a different problem (maybe a typo in the docs for example) this is what branches are for. Save your commits, create a new branch off `master fix/fix-documentation-typo` and then save that change to that branch. Now you have a small change, one that is easy to merge, which you can prepare a pull request for.

Keep your changes focused on the goal, do not add overhead to the reviewer or to yourself by changing things that do not need it (yet).

Note

It's OK to make changes in a 'messy' way locally, with lots of commits that maybe include things that are not needed. However, be sure to take some time to review your commits and clean up anything that is not required before you do your pull request.

Write unit tests

We are getting close to having a pull request, but the next critical step is unit tests. It's common to find that adding tests for code you wrote will take 5-10x longer than the actual bug fix. Often, if the use case is right, it is better to write the tests first and get them running (but failing) before you fix the problem.

Finding how and where to write the unit tests can be hard in a new project, but hopefully, the project's development docs contain the clues you need to get started. Read through the [dedicated testing section](#) in the development documentation.

If you fix a bug or introduce a new feature, you want to ensure that fix is long-lived and does not break again. You also want to help yourself by thinking through edge cases or potential problems. Testing helps with this. While regressions do happen, they are less likely to happen when code is tested.

Many projects will not even review a pull request without unit tests. Often, fixing a bug is not hard, ensuring the fix is the 'real' fix and that it does not break again is the hard part. Take the time to do the harder thing. It will help you grow as a developer and help your contributions make a longer lasting difference.

Note

A pull request that just adds unit tests to some core functionality that does not yet have tests is a great way to contribute, it helps you learn about the code and makes the project more reliable.

Checklist

- [] After feeling confident about a solution, add a comment to the issue
- [] Create a new branch off 'main' to track your work separate from the main branch
- [] Keep the changes focused towards your goal, asking questions on the issue if ↗ direction is needed
- [] Write unit tests

7. Submitting a Pull Request

A pull request that has the title ‘fixes issue’ is unhelpful at best, and spammy at worst. Take a few moments to think about how to give your change a title. Communicate (in a few words) the problem solved or feature added or bug fixed. Instead of ‘Fixes 10423’, use words and write a title ‘Fixes documentation dark mode refresh issue’. No one in a project knows that issue 10423 is that one about the documentation dark mode refresh issue.

Please try to add a proper title when you create the pull request. This will ensure that any notifications that go out to the team have the a suitable title from the start.

Note

Remember you can make a **draft** pull request in both GitHub and GitLab. This is a way to run the CI steps but in a way that indicates you are not ready for a review yet.

Referencing the issue being fixed within the pull request description is just as important as a good title. A pull request without a description is very difficult to review. Adding a note similar to `fixes #1234` in your description message will let GitHub know that the change is for that issue. Add some context and some steps to reproduce the issue or scenario.

If the change is visual it's strongly recommended to add before and after screenshots. This helps you confirm the change has worked and also helps reviewers understand the change made.

It is often good to write yourself a checklist for any pull request and fill in the gaps. **Remember that the Pull Request template is there for a reason so please use that checklist also.**

Pull Request Checklist

- [] Small description of the solution, one sentence.
- [] Link to issue/s that should be resolved if this pull request gets merged.
- [] Questions or assumptions, maybe you made an assumption we no longer support ↗ IE11 with your CSS change, if it's not in the docs – write the assumption down.
- [] Details – additional details, context or links that help the reviewer ↗ understand the pull request.
- [] Screenshots – added before and after the change has been applied.
- [] Browser and accessibility checks done, or not done. Added to the description.

7a. Review & fix the CI failures

Once you have created your pull request, there will often be a series of [build/check/CI](#) steps that run.

These steps are normally all required to pass before the pull request can be merged. CI is a broad term but usually, the testing and linting will run on the code you have proposed to change. Linting is a tricky one because sometimes the things that are flagged seem trivial, but they are important for code consistency. Re-read the development instructions and see how you can run the linting locally to avoid frustrating back & forth with small linting fixes.

Testing is a bit more complex. Maybe all the tests can be run locally or maybe the CI will run tests on multiple versions of a project or language. Do your best to run all the tests locally, but there may still be issues on the CI when you do. That is OK, and normally you can solve these issues one by one.

The most important thing is to not just ignore CI failures. Read through each error report and try to work out the problem and provide a fix. Ignoring these will likely lead to pull requests that do not get reviewed because they do not get the basics right.

 Note

GitHub will not run the CI automatically for new contributors in some projects. This is an intentional security feature and a core contributor will need to approve your initial CI run.

7b. Push to the same branch with fixes and do not open a new pull request

Finally, after you have fixed the failing linting and tests locally, you will want to push those changes to your remote branch. You do not need to open a new pull request. This creates more noise and confusion. Instead, push your changes up to your branch, and the CI will run automatically on those changes.

You can add a comment if you want to the pull request that you have updated, but often this is not really needed.

Avoid opening multiple pull requests for the same fix. Doing that means all the comments and discussion from the previous pull request will get lost and reviewers will have trouble finding them.

8. Next steps

When you take time to contribute out of your own personal time, or even that from your paid employer, it can be very frustrating when a pull request does not get reviewed. It is best to temper your expectations with this process and remember that many people on the other side of this are also volunteers or have limited time to prioritize.

It is best to celebrate your accomplishment at this point even if your pull request never gets merged. It's good to balance that with an eagerness about getting your amazing fix in place to help people who use the project. Balancing this tension is hard, but the unhelpful thing to do is give up and never contribute or decide that you won't respond to feedback because it came too late.

Remember that it is OK to move on and try something else. Try a different issue or project or area of the code. Don't just sit waiting for a response on the one thing you did before looking at other challenges.

Responding to a review

Almost every Pull Request (PR) (except for the most smallest changes) will have some form of feedback. This will usually come in the form of a review and a request for changes. At this point your PR will be flagged as 'needs work', 'needs tests' or in some cases 'needs design decision'. Take the time to read all the feedback and try to resolve or respond to comments if you have questions.

 Warning

Avoid closing the PR only to create a new one, instead keep it open and push your changes/fixes to the same branch. Unless directed to make the PR smaller, keep the same one open and work through items one by one.

Once you feel that you have answered all the concerns, just add a comment (it does not need to be directed at the reviewer) that this is ready for another review.

Once merged in

Well done! It's time to party! Thank you for taking the time to contribute to Wagtail and making the project better for thousands of users.

Common questions

How can I start contributing?

- Ideally, read this guide in full, otherwise see some quick start tips.
- Start simple - pick something small first. The [good first issue](#) label is a good place to look.
- Read the entire issue, all comments (links) and related issues.
- Someone may have started work (that work may have stalled).
- Check if assigned, we do not usually use that unless assigned to someone within the core team.

If you have done all of that and think you can give it a go just a comment with something like 'I will give this a go', no need to ask for permission.

Do I need to ask for permission to work on an issue?

No. However, check if there is an existing pull request (PR). If there is nothing, you can optionally add a comment mentioning that you're starting work on it.

What should I include in my pull request (PR)

0. The fix or feature you are working on
1. Tests
2. Linted code (we make use of [pre-commit](#). You can run all formatting with `make format`)
3. Updated documentation where relevant (such as when adding a new feature)

What if I fix multiple issues in the same pull request (PR)

It is best to avoid fixing more than one issue in a single pull request, unless you are a core contributor or there is a clear plan that involves fixing multiple things at once. Even then, it is usually a bad idea as it makes it harder for your pull request to be reviewed and it may never get merged as it's too complex. This is especially true for completely unrelated issues such as a documentation fix for translators and a bug fix for StreamField. It is always best to create two branches and then two separate pull requests.

When do I need to write unit tests for a pull request (PR)?

Unless you are updating the documentation or only making visual style changes, your Pull Request should contain tests.

If you are new to writing tests in Django, start by reading the [Django documentation on testing](#). Re-read the [Wagtail documentation notes on testing](#) and have a look at [existing tests](#).

Note that the JavaScript testing is not as robust as the Python testing, if possible at least attempt to add some basic JS tests to new behavior.

Where can I get help with my pull request (PR)?

The `#new-contributors` channel in [Slack](#) is the best place to get started with support for contributing code, especially for help with the process of setting up a dev environment and creating a PR.

There is also the more recently created `#development` channel for advice on understanding and getting around the Wagtail code-base specifically. Finally, if you have a general problem with understanding how to do something in Wagtail itself or with a specific feature, then `#support` can be used.

What if there is already an open pull request (PR)?

Be sure to always read the issue in full and review all links, sometimes there may already be an open pull request for the same issue. To avoid duplicating efforts it would be best to see if that pull request is close to ready and then move on to something else. Alternatively, if it has been a long enough amount of time, you may want to pick up the code and build on it to get it finished or ask if they need help.

Can I just use Gitpod to develop

While Gitpod is great for some small scale Pull Requests, it will not be a suitable tool for complex contributions and it's best to take the time to set up a fully functional development environment so you can manage branches and ongoing commits to one branch.

Here are some links for using Gitpod with the Wagtail packages:

- [Bakerydemo Gitpod instructions](#)
- [Wagtail Gitpod – Wagtail development setup in one click](#)

How can I be assigned an issue to contribute to

We only use GitHub's issue assignment feature for members of the Wagtail core team when tasks are being planned as part of core roadmap features or when being used for a specific internship program. If an issue is not assigned to anyone, feel free to work on it, there is no need to ask to be assigned the issue.

Instead, review the issue, understand it and if you feel you can contribute you can just raise a Pull Request, or add a comment that you are taking a look at this. There are no strict claiming or reserving rules in place, anyone is free to work on any issue, but try to avoid double effort if someone has already got a Pull Request underway.

Helpful links

- Django's contributor guide is a helpful resource for contributors, even those not contributing to Wagtail.
- MDN's open source etiquette is a great guideline for how to be a great contributor.
- Learning Git Branching a solid interactive guide to understand how git branching works.
- Hacktoberfest every October, join in the fun and submit pull requests.
- 21 Pull Requests a December community effort to contribute to open source.
- Windows step by step guide to getting bakerydemo running with local Wagtail

Inspiration for this content

Some great further reading also

- 5 simple ways anyone can contribute to Wagtail
- Ten tasty ingredients for a delicious pull request
- Preparing a Gourmet Pull Request
- Zulip's contributor guide
- Documentation for absolute beginners to software development (discussion)
- New contributor FAQ

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 [Fast Node Manager \(fnm\)](#), 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 `fnm 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](#).

Fork the [the Wagtail codebase](#) and clone the forked copy:

```
git clone https://github.com/username/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, to install the local Wagtail development instance in your virtualenv for your Wagtail site:

```
pip install -e path/to/wagtail"[testing, docs]" -U
```

Here, `path/to/wagtail` is the path to your local Wagtail copy.

Development on Windows

Documentation for development on Windows has some gaps and should be considered a work in progress. We recommend setting up on a local virtual machine using our already available scripts, [docker-wagtail-develop](#) or [vagrant-wagtail-develop](#)

If you are confident with Python and Node development on Windows and wish to proceed here are some helpful tips.

We recommend [Chocolatey](#) for managing packages in Windows. Once Chocolatey is installed you can then install the `make` utility in order to run common build and development commands.

We use LF for our line endings. To effectively collaborate with other developers on different operating systems, use Git's automatic CRLF handling by setting the `core.autocrlf` config to `true`:

```
git config --global core.autocrlf true
```

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 5000 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 the 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 argument `--elasticsearch7` or `--elasticsearch8` (corresponding to the version of Elasticsearch you want to test against):

```
python runtests.py --elasticsearch8
```

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=https://my-elasticsearch-instance:9200 python runtests.py --  
→elasticsearch8
```

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 createschematable
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://127.0.0.1:8000` by default. Use the `TEST_ORIGIN` environment variable to use a different port, or test a remote Wagtail instance: `TEST_ORIGIN=http://127.0.0.1:9000` `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: 17, 18
Mobile Safari	iOS Tablet	Last 2: 17, 18
Chrome	Android	Last 2
Chrome	Desktop	Last 2
MS Edge	Windows	Last 2
Firefox	Desktop	Latest
Firefox ESR	Desktop	Latest: 128
Safari	macOS	Last 3: 16, 17, 18

We aim for Wagtail to work in those environments. Our development standards ensure that the site is usable on other browsers **and will work on future browsers**.

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 specialized training – here are tools we rely on to help identify accessibility issues, to use during development and code reviews:

- [@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:

```
# Starting from the wagtail root directory:

# 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:

```
# Starting from the wagtail root directory:

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:

```
# Starting from the wagtail root directory:

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`:

```
# Starting from the wagtail root directory:  
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.

Using forks for installation

Sometimes it may be necessary to install Wagtail from a fork. For example your site depends on a bug fix that is currently waiting for review, and you cannot afford waiting for a new release.

The Wagtail release process includes steps for static asset building and translations updated which means you cannot update your requirements file to point a particular git commit in the main repository.

To install from your fork, from the root of your Wagtail git checkout (and assuming the tooling for building the static assets has previously been installed using `npm install`), run:

```
python ./setup.py sdist
```

This will create a `.tar.gz` package within `dist/`, which can be installed with `pip`.

For remote deployments, it's usually most convenient to upload this to a public URL somewhere and place that URL in your project's requirements in place of the standard `wagtail` line.

1.10.4 Translations

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

Translation work should be submitted through Transifex, for information on how to get started see [Translations](#).

1.10.5 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. Here are some other ways to contribute if you are getting started or have been using Wagtail for a long time but are unable to contribute code.

- Contribute to one of the other [core Wagtail projects](#) in GitHub.
- Contribute to one of the community-maintained packages on [Wagtail Nest](#).
- Contribute user-facing documentation (including translations) on the [Wagtail guide](#).

Non-code contributions

- Star the [wagtail](#) project on GitHub
- Support others with answers to questions on the [Wagtail StackOverflow topic](#) or in Slack #support.
- Write a review of [Wagtail](#) on G2.
- Provide some thoughtful feedback on the [Wagtail](#) discussions.
- Submit (or write) a tutorial or great package to the This Week in Wagtail newsletter, Awesome Wagtail or *Third-party tutorials*.

1.10.6 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
- Framework :: Wagtail :: 5
- Framework :: Wagtail :: 6

You can also find a curated list of awesome packages, articles, and other cool resources from the Wagtail community at [Awesome Wagtail](#).

1.10.7 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 `{% translate %}` 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`. American English is also the preferred spelling style when writing documentation. Learn more about our documentation writing style in [Writing style guide](#).

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`

Naming conventions

Use `classname` in Python / HTML template tag variables

`classname` is preferred for any API / interface or Django template variables that need to output an HTML class.

Django template tag

Example template tag definition

```
@register.inclusion_tag("wagtailadmin/shared/dialog/dialog_toggle.html")
def dialog_toggle(dialog_id, classname="", text=None):
    return {
        "classname": classname,
        "text": text,
    }
```

Example template

```
{% comment "text/markdown" %}

Variables accepted by this template:

- `classname` - {string?} if present, adds classname to button
- `dialog_id` - {string} unique id to use to reference the modal which will be triggered

{% endcomment %}

<button type="button" class="{{ classname }}" data-a11y-dialog-show="{{ dialog_id }}>
  {{ text }}
</button>
```

Example usage

```
{% dialog_toggle classname='button button-primary' %}
```

Python / Django class driven content

```
class Panel:
    def __init__(self, heading="", classname="", help_text="", base_form_class=None):
        self.heading = heading
        self.classname = classname
```

Details

Convention	Usage
classname	✓ Preferred for any new code.
class	* Only if used as part of a generic attrs-like dict; however avoid due to conflicts with Python class keyword.
class-	✗ Avoid for new code.
names	
class_name	✗ Avoid for new code.
class_names	✗ Avoid for new code.
className	✗ Avoid for new code.
class-	✗ Avoid for new code.
Names	

Python coding guidelines

PEP8

We ask that all Python contributions adhere to the [PEP8](#) style guide. All files should be formatted using the [ruff](#) 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.

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 versions 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`:

```
from django import VERSION as DJANGO_VERSION

if DJANGO_VERSION >= (1, 9):
    # Use new attribute
    related_field = field.rel
else:
    # Use old, deprecated attribute
    related_field = field.related
```

Always compare against the version using greater-or-equals (`>=`), 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 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, and `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 IDs for semantics only, classes for styling, `data-` attributes for JavaScript behavior.
- Order attributes with `id` first, then `class`, then any `data-` or other attributes with Stimulus `data-controller` first.
- 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 colors 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.

Stylesheets

Most of our styles are combined into a single main stylesheet, `core.css`. This is the recommended approach for all new styles, to reduce potential style clashes, and encourage reuse of utilities and component styles between views. Imports within `core.scss` are structured according to ITCSS. There are two major exceptions to the ITCSS structure:

- Legacy vendor CSS in `vendor/` is imported in the order it was loaded in before adding in the main stylesheet, to avoid compatibility issues. If possible, those styles should be converted to components and loaded further down the cascade.
- Legacy layout-specific styles in `layouts/` are imported at the very end of the file, matching how styles were previously loaded across multiple stylesheets. If possible, those styles should be converted to components or utilities and loaded further up the cascade.

When creating new styles, always prefer components, adding a new stylesheet in the `components` folder and importing it in `core.scss`.

Global styles

For all of our styles, we use:

- A very old version of `normalize.css` as a CSS reset.
- `box-sizing: border-box`, with elements always inheriting the `box-sizing` of their parent.
- Global CSS variables for colors, so they can be changed by site implementers.
- Global CSS variables for font family, so they can be changed by site implementers.
- A `--w-direction-factor` CSS variable, set to `1` by default and `-1` for RTL languages, to allow reversing of calculations of physical values (transforms, background positions) and mirroring of icons and visuals with directional elements like arrows.
- The `--w-density-factor` CSS variable, to let users control the information density of the UI. Set to `1` by default, and lower or higher values to reduce or increase the spacing and size of UI elements.

Tailwind usage

We use [Tailwind](#) to manage our design tokens via its theme, and generate CSS utilities. It is configured in `tailwind.config.js`, with a base configuration intended to be reusable in other projects.

Wagtail uses most of Tailwind's core plugins, with an override for them to create [Logical properties and values](#) styles while still using Tailwind's default utility and design token names.

With utility classes, we recommend to:

- Keep their number to a reasonable maximum, creating component styles instead if the utilities are inter-dependent, or if they are frequently reused together.
- Avoid utilities relating to font size, weight, or other typographic considerations. Instead, use the higher-level type scale as defined in `typography.js`.

Sass usage

We keep our Sass usage to a minimum, preferring verbose vanilla CSS over advanced Sass syntax. Here are specific Sass features to completely avoid:

- Placeholders / `@extend`. Leads to unexpected cascading of styles.
- Color manipulation. All of our colors are defined in JavaScript via Tailwind, to generate CSS variable definitions and documentation consistently.

And Sass features to use with caution:

- Sass nesting. Avoid relying on Sass nesting specifically, and overly specific selectors. Most styles can be written with either one or two levels of nesting, 3 for specific UI states, and 4 in the most complex scenarios only.
- Parent selector (`&`) interpolation. Only use interpolation in class names sparingly, so we can more easily search for styles across the project.
- Sass variables. Prefer Tailwind theme variables to reuse our design tokens, or CSS variables when a specific property changes based on state. Sass variables should only be used as shorter aliases for those scenarios, or as local component variables.
- Mixins. Only create new mixins if the styles can't be written as reusable component or utility styles.
- Sass math. With most of our design tokens defined in Tailwind, loaded via PostCSS, we use `calc` functions for math operations rather than Sass.

Forced colors mode

Also known as Windows High Contrast mode, or Contrast Themes. This is a feature of Windows for users to override websites' styles with their own, so text is more readable. We intend to fully support it in all of our styles. Here are recommended practices:

- Add additional borders where the background color would otherwise convey the position of specific elements, particularly for page regions and components layered above the page.
- Overrides with `@media (forced-colors: active)` should only be used when there is no simpler alternative. Write CSS for WHCM support from the get-go rather than with sweeping overrides.
- Never use `forced-color-adjust: none`. It compromises compatibility with a wide range of custom themes, and should only be needed if a component relies on a specific color hue to work (which is an anti-pattern).

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.

Stimulus

Wagtail uses [Stimulus](#) as a lightweight framework to attach interactive behavior to DOM elements via `data-` attributes.

Why Stimulus

Stimulus is a lightweight framework that allows developers to create interactive UI elements in a simple way. It makes it easy to do small-scale reactivity via changes to data attributes and does not require developers to ‘init’ things everywhere, unlike JQuery. It also provides an alternative to using inline script tag usage and window globals which reduces complexity in the codebase.

When to use Stimulus

Stimulus is our [preferred library](#) for simple client-side interactivity. It’s a good fit when:

- The interactivity requires JavaScript. Otherwise, consider using HTML and CSS only.
- Some of the logic is defined via HTML templates, not just JavaScript.
- The interactivity is simple, and doesn’t require usage of more heavyweight libraries like React.

Wagtail’s admin interface also leverages jQuery for similar scenarios. This is considered legacy and will eventually be removed. For new features, carefully consider whether existing jQuery code should be reused, or whether a rebuild with Stimulus is more appropriate.

How to build a Stimulus controller

First think of how to name the controller. Keep it concise, one or two words ideally. Then,

1. Start with the HTML templates, build as much of the UI as you can in HTML alone. Ensure it is accessible and follows the CSS guidelines.
2. Create the controller file in our `client/src/controllers` folder, along with its tests (see [Testing](#)) and Storybook stories.
3. For initialization, consider which [controller lifecycle methods](#) to use, if any (`connect`, `initialize`).
4. If relevant, also consider how to handle the controlled element being removed from the DOM [disconnect lifecycle method](#).
5. Document controller classes and methods with [JSDoc annotations](#).
6. Use [values](#) to provide options and also reactive state, avoiding instance properties if possible. Prefer falsey or empty defaults and avoid too much usage of the Object type when using values.
7. Build the behavior around small, discrete, methods and use [Stimulus actions](#) declared in HTML to drive when they are called.

Helpful tips

- Prefer controllers that do a small amount of ‘work’ that is collected together, instead of lots of large or specific controllers.
- Lean towards dispatching events for key behavior in the UI interaction as this provides a great way for custom code to hook into this without an explicit API, but be sure to document these.
- Multiple controllers can be attached to one DOM element for composing behavior, where practical split out behavior to separate controllers.
- Consider when to document controller usage for non-contributors.
- When writing unit tests, note that DOM updates triggered by data attribute changes are completed async (next microtick) so will require a await Promise or similar to check for the changes in JSDom.
- Avoid hard-coding a controller’s identifier, instead reference it with `this.identifier` if adjusting attributes. This way the controller can be used easily with a changed identifier or extended by other classes in the future.

Multilingual support

This is an area of active improvement for Wagtail, with [ongoing discussions](#).

- Always use the `trimmed` attribute on `blocktranslate` tags to prevent unnecessary whitespace from being added to the translation strings.

Right-to-left language support

We support right-to-left languages, and in particular viewing the Wagtail admin interface in a horizontally mirrored layout. Here are guidelines to guarantee support:

- Write styles with [logical properties and values](#) whenever possible.
- For styles that can only be written with physical properties (translations, background positions), use the `--w-direction-factor` variable equal to 1 or -1 so the value reverses based on the `dir` attribute of the element or page.
- As a last resort, use `[dir='rtl']` style if there is no other way to write styles.

Make sure to also reverse the direction of any position calculation in JavaScript, as there is no support of logical values in DOM APIs (x-axis offsets always from the left).

Icons

We use inline SVG elements for Wagtail’s icons, for performance and so icons can be styled with CSS. View [Icons](#) for information on how icons are set up for Wagtail users.

Adding icons

Icons are SVG files in the Wagtail admin template folder.

When adding or updating an icon,

1. Run it through [SVGO](#) with appropriate compression settings.
2. Manually remove any unnecessary attributes. Set the `viewBox` attribute, and remove `width` and `height` attributes.
3. Manually add its `id` attribute with a prefix of `icon-` and the icon name matching the file name. Keep the icon as named from its source if possible.
4. Add or preserve licensing information as an HTML comment starting with an exclamation mark: `<!--! Icon license -->`. For Font Awesome, we want: `<!--! [icon name] ([icon style]): Font Awesome [version] -->`. For example, `<!--! triangle-exclamation (solid): Font Awesome Pro 6.4.0 -->`.
5. Add the icon to Wagtail's own implementation of the `register_icons` hook, in alphabetical order.
6. Go to the styleguide and copy the Wagtail icons table according to instructions in the template, pasting the result in `wagtail_icons_table.txt`.
7. If the icon requires right-to-left mirroring, add the `class="icon--directional"` attribute.

Documentation guidelines

- [Writing style guide](#)
- [Formatting recommendations](#)
- [Formatting to avoid](#)
- [Code example considerations](#)

Writing style guide

To ensure consistency in tone and language, follow the [Google developer documentation style guide](#) when writing the Wagtail documentation.

Formatting recommendations

Wagtail's documentation uses a mixture of [Markdown](#) (with MyST) and [reStructuredText](#). We encourage writing documentation in Markdown first, and only reaching for more advanced reStructuredText formatting if there is a compelling reason. Docstrings in Python code must be written in reStructuredText, as using Markdown is not yet supported.

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.

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

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 and inline code when relevant.

```
Use **bold** and _italic_ sparingly and inline `code` when relevant.
```

Keep in mind that in reStructuredText, italic is written with *, and inline code must be written with double backticks, like ``code``.

```
Use **bold** and *italic* sparingly and 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, sh.

```
```python
INSTALLED_APPS = [
 ...
 "wagtail",
 ...
]
```
```

```
INSTALLED_APPS = [
    ...
    "wagtail",
    ...
]
```

When using console (terminal) code blocks

Note

\$ or > prompts are not needed, this makes it harder to copy and paste the lines and can be difficult to consistently add in every single code snippet.

Use sh as it has better support for comment and code syntax highlighting in MyST's parser, plus is more compatible with GitHub and VSCode.

```
```sh
some comment
some command
```
```

```
# some comment
some command
```

Use doscon (DOS Console) only if explicitly calling out Windows commands alongside their bash equivalent.

```
```doscon
some comment
some command
```

```

```
# some comment
some command
```

Code blocks that contain triple backticks

You can use three or more backticks to create code blocks. If you need to include triple backticks in a code block, you can use a different number of backticks to wrap the code block. This is useful when you need to demonstrate a Markdown code block, such as in the examples above.

```
```md
```python
print("Hello, world!")
```
```

```

```
```python
print("Hello, world!")
```
```

```

## 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.

Do not let external links hide critical context for the reader. Instead, provide the core information on the page and use links for added context.

```
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 target](register_reports_menu_item).
Do not use [click here](https://www.example.com) as the link's text, use a more descriptive label.
Do not rely on links for critical context, like [why it is important](https://www.example.com).
```

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 target*. Do not use *click here* as the link's text, use a more descriptive label. Do not rely on links for critical context, like *why it is important*.

### Anchor links

Anchor links point to a specific target on a page. They rely on the page having the target created. Each target must have a unique name and should use the `lower_snake_case` format. A target can be added as follows:

```
(my_awesome_section) =

Some awesome section title

...
```

The target 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`)

Rendered output:

### 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 [Cross-references](#).

### Intersphinx links (external docs)

Due to the large amount of links to external documentation (especially Django), we have added the integration via intersphinx references. This is configured via `intersphinx_mapping` in the `docs/conf.py` file. This allows you to link to specific sections of a project's documentation and catch warnings when the target is no longer available.

Markdown example:

```
You can select widgets from [Django's form widgets] (inv:django#ref/forms/widgets).
```

reStructuredText example:

```
You can select widgets from :doc:``Django form widget <django:ref/forms/widgets>``.
```

The format for a Sphinx link in Markdown is `inv:key:domain:type#name`. The `key`, `domain`, and `type` are optional, but are recommended to avoid ambiguity when there are multiple targets with the same name.

If the name contains a space, you need to wrap the whole link in angle brackets `<>`.

```
See Django's docs on [] (<inv:django:std:label#topics/cache:template fragment caching>
→).
```

See Django's docs on [Template fragment caching](#).

## Find the right intersphinx target

The intersphinx target for a specific anchor you want to link to may not be obvious. You can use the `myst-inv` command line tool from MyST-Parser and save the output as a JSON or YAML file to get a visual representation of the available targets.

```
myst-inv https://docs.djangoproject.com/en/stable/_objects/ --format=json > django-
→inv.json
```

Using the output from `myst-inv`, you can follow the tree structure under the `objects` key to build the link target. Some text editors such as VSCode can show you the breadcrumbs to the target as you navigate the file.

Other tools are also available to help you explore Sphinx inventories, such as `sphobjinv` and Sphinx's built-in `sphinx.ext.intersphinx` extension.

```
sphobjinv suggest "https://docs.djangoproject.com/en/stable/_objects/" 'template_
→fragment caching' -su
```

```
python -m sphinx.ext.intersphinx https://docs.djangoproject.com/en/stable/_objects/
```

In some cases, a more specific target may be available in the documentation. However, you may need to inspect the source code of the page to find it.

For example, the above section on Django's docs can also be linked via the type `templatetag` and the name `cache`.

```
See Django's docs on the [](inv:django:std:templatetag#cache) tag.
```

See Django's docs on the `cache` tag.

Note that while the link takes you to the same section, the URL hash and the default text will be different. If you use a custom text, this may not make a difference to the reader. However, they are semantically different.

Use the first approach (with the `label` type) when you are linking in the context of documentation in general, such as a guide on how to do caching. Use the second approach (with the `templatetag` type) when you are linking in the context of writing code, such as the use of the `{% cache %}` template tag. The second approach is generally preferred when writing docstrings.

## Absolute links

Sometimes, there are sections in external docs that do not have a Sphinx target target attached at all. Before linking to such sections, consider linking to the nearest target before that section. If there is one available that is close enough such that your intended section is immediately visible upon clicking the link, use that. Otherwise, you can write it as a full URL. Remember to use the `stable` URL and not a specific version.

A common example of using full URLs over intersphinx links is when linking to sections in Django's release notes.

```
`DeleteView` has been updated to align with [Django 4.0's `DeleteView`_
→implementation] (https://docs.djangoproject.com/en/stable/releases/4.0/#deleteview-
→changes).
```

`DeleteView` has been updated to align with Django 4.0's `DeleteView` implementation.

For external links to websites with no intersphinx mapping, always use the `https://` scheme.

Absolute links are also preferred for one-off links to external docs, even if they have a Sphinx object inventory. Once there are three or more links to the same project, consider adding an intersphinx mapping if possible.

### Code references

When linking to code references, you can use Sphinx's reference roles.

Markdown example:

```
The {class}`~django.db.models.JSONField` class lives in the {mod}`django.db.models.
↳fields.json` module,
but it can be imported from the {mod}`models <django.db.models>` module directly.
For more info, see {ref}`querying-jsonfield`.
```

reStructuredText example:

```
The :class:`~django.db.models.JSONField` class lives in the :mod:`django.db.models.
↳fields.json` module,
but it can be imported from the :mod:`models <django.db.models>` module directly.
For more info, see :ref:`querying-jsonfield`.
```

The `JSONField` class lives in the `django.db.models.fields.json` module, but it can be imported from the `models` module directly. For more info, see [Querying JSONField](#).

Adding `~` before the dotted path will shorten the link text to just the final part (the object name). This can be useful when the full path is already mentioned in the text. You can also set the current scope of the documentation with a `module` or `currentmodule` directive to avoid writing the full path to every object.

```
```{currentmodule} wagtail.admin.viewsets.model  
...  
The {class}`ModelViewSet` class extends the {class}`~wagtail.admin.viewsets.base.  
↳ViewSet` class.
```

The `ModelViewSet` class extends the `ViewSet` class.

A reference role can also define how it renders itself. In the above examples, the `class` and `mod` roles are rendered as an inline code with link, but the `ref` role is rendered as a plain link.

These features make reference roles particularly useful when writing reference-type documentation and docstrings.

Aside from using reference roles, you can also use the link syntax. Unlike reference roles, the link syntax requires the full path to the object and it allows you to customize the link label. This can be useful when you want to avoid the reference role's default rendering, for example to mix inline code and plain text as the link label.

```
For more details on how to query the [`JSONField` model field] (django.db.models.  
↳JSONField),  
see [the section about querying `JSONField`] (inv:django#querying-jsonfield).
```

For more details on how to query the `JSONField` model field, see the section about querying `JSONField`.

Note and warning call-outs

Use notes and warnings sparingly to get the reader's attention when needed. These can be used to provide additional context or to warn about potential issues.

```
```{note}
Notes can provide complementary information.
```

```{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.

```
! [The TableBlock component in StreamField, with row header, column header, caption
→fields - and then the editable table] (/static/images/screen40_table_block.png)
```

Body

The recipe's step-by-step instructions and any other relevant information.

Table block *



Table headers

Display the first row AND first column as headers ▾

Which cells should be displayed as headers?

Table caption

A heading that identifies the overall topic of the table, and is useful for screen reader users.

Expected yield

| Buns | Prep time | Cooking time | Instructions |
|------|-----------|------------------------|----------------------|
| 12 | 30min | 1.5h | All quantities as-is |
| 18 | 40min | 2h | 1.5x all amounts |
| 24 | 45min | Depending on oven size | 2x all amounts |

Docstrings and API reference (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 that we highly recommend using to document Wagtail's APIs.

Modules, classes, and functions can be documented with docstrings. Class and instance attributes can be documented with docstrings (with triple quotes "'''") or doc comments (with hash-colon # :). Docstrings are preferred, as they have better integration with code editors. Docstrings in Python code must be written in reStructuredText syntax.

```

SLUG_REGEX = r'^[-a-zA-Z0-9_]+$'
"""Docstring for module-level variable ``SLUG_REGEX``."""

class Foo:
    """Docstring for class ``Foo``."""

    bar = 1
    """Docstring for class attribute ``Foo.bar``."""

    #: Doc comment for class attribute ``Foo.baz``.
    #: It can have multiple lines, and each line must start with ``#:``.
    #: Note that it is written before the attribute.
    #: While Sphinx supports this, it is not recommended.
    baz = 2

    def __init__(self):
        self.spam = 4
    """Docstring for instance attribute ``spam``."""

```

The autodoc extension provides many directives to document Python code, such as `autoclass`, `autofunction`, `automodule`, along with different options to customize the output. In Markdown files, these directives need to be wrapped in an `eval-rst` directive. As with docstrings, everything inside the `eval-rst` block must be written in reStructuredText syntax.

You can mix automatic and non-automatic documentation. For example, you can use `module` instead of `automodule` and write the module's documentation in the `eval-rst` block, but still use `autoclass` and `autofunction` for classes and functions. Using automatic documentation is recommended, as it reduces the risk of inconsistencies between the code and the documentation, and it provides better integration with code editors.

```

```{eval-rst}
.. module:: wagtail.coreutils

 Wagtail's core utilities.

 .. autofunction:: cautious_slugify
 :no-index:
```

```

Wagtail's core utilities.

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).

For more details on the available directives and options, refer to [Directives](#) and [The Python Domain](#) in Sphinx's documentation.

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

The `toctree` and `contents` directives can be used to render tables of contents.

```
```{toctree}

maxdepth: 2
titlesonly:

getting_started/index
topics/index
```

```{contents}

local:
depth: 1

```
```

Version added, changed, deprecations

Sphinx offers release-metadata directives to present information about new or updated features in a consistent manner.

```
```{versionadded} 2.15
The `WAGTAIL_NEW_SETTING` setting was added.
```

```{versionchanged} 2.15
The `WAGTAIL_OLD_SETTING` setting was deprecated.
```
```

Added in version 2.15: The `WAGTAIL_NEW_SETTING` setting was added.

Changed in version 2.15: The `WAGTAIL_OLD_SETTING` setting was deprecated.

These directives will typically be removed two releases after they are added, so should only be used for short-lived information, such as “The `WAGTAILIMAGES_CACHE_DURATION` setting was added”. Detailed documentation about the

feature should be in the main body of the text, outside of the directive.

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 italicized paragraph underneath the image.

Other reStructuredText syntax and Sphinx directives

We generally want to favor 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.

Markdown in reStructuredText

Conversely, do not use Markdown syntax in places where reStructuredText is required. A common mistake is writing Markdown-style inline `code` (with single backticks) inside Python code docstrings and inside eval-rst directives. This is not supported and will not render correctly.

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.

Code example considerations

When including code examples, particularly JavaScript or embedded HTML, it's important to follow best practices for security, accessibility and approaches that make it easier to understand the example.

These are not hard rules but rather considerations to make when writing example code.

Reference example filename

At the start of a code snippet, it can be helpful to reference an example filename at the top. For example: # wagtail_hooks.py or // js/my-custom.js.

CSP (Content Security Policy) compliance

When adding JavaScript from external sources or custom scripts, ensure CSP compliance to prevent security vulnerabilities like cross-site scripting (XSS).

Avoid mark_safe where possible, and use format_html and use examples that load external files to manage scripts securely instead of inline <script> usage.

Accessibility compliance

Make sure that all examples are accessible and adhere to accessibility standards (for example: WCAG, ATAG).

For interactive components, ensure proper keyboard navigation and screen reader support. When creating dynamic content or effects (such as animations or notifications), provide options for users to pause, stop, or adjust these features as needed.

If needed, call out explicitly that the example is not compliant with accessibility and would need additional considerations before adoption.

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 a 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 that 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, and 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](#)

Translations

Wagtail uses [Transifex](#) to translate the content for the admin interface. Our goal is to ensure that Wagtail can be used by those who speak many different languages. Translation of admin content is a great way to contribute without needing to know how to write code.

Note

For translations and internationalization of content made with Wagtail see [Internationalization](#).

Translation workflow

Wagtail is localized (translated) using Django's [translation system](#) and the translations are provided to and managed by [Transifex](#), a web platform that helps organizations coordinate translation projects.

Translations from Transifex are only integrated into the repository at the time of a new release. When a release is close to being ready there will be a RC (Release Candidate) for the upcoming version and the translations will be exported to Transifex.

During this RC period, usually around two weeks, there will be a chance for all the translators to update and add new translations. We will also notify the `#translators` channel in the Wagtail Slack group at this time.

These new translations are imported into Wagtail for any subsequent RC and the final release. If translations reach a threshold of about 80%, languages are added to the default list of languages users can choose from.

How to help out with translations

- Join the Wagtail community on [Slack](#)
- Search through the channels to join the `#translator` channel and introduce yourself
- Go to [Transifex](#)
- Click on start for free
- Fill in your Username, Email and Password
- Agree to the terms and conditions
- Click on free trial or join an existing organization
- Join [Wagtail](#) and see the list of languages on the dashboard
- Request access to become a member of the language team you want to work with on Slack (mention your Transifex username)
- A view resources button appears when you hover over the ready to use part on the right side of the page
- Click on the button to get access to the resources available
- This takes you to the language section
- This page has a translation panel on the right and a list of strings to be translated on the left
- To translate a project, select it and enter your translation in the translation panel
- Save the translation using the translation button on the panel

Marking strings for translation

In code, strings can be marked for translation with using Django's `gettext` or `gettext_lazy` in Python and `blocktranslate`, `translate`, and `_("")` in templates.

In both Python and templates, make sure to always use a named placeholder. In addition, in Python, only use the printf style formatting. This is to ensure compatibility with Transifex and help translators in their work.

Translations within Python

```
from django.utils.translation import gettext_lazy as _

# Do this: printf style + named placeholders
_("Page %(page_title)s with status %(status)s") % {"page_title": page.title, "status": page.status_string}

# Do not use anonymous placeholders
_("Page %s with status %s") % (page.title, page.status_string)
_("Page {} with status {}").format(page.title, page.status_string)

# Do not use positional placeholders
_("Page {0} with status {1}").format(page.title, page.status_string)

# Do not use new style
_("Page {page_title} with status {status}").format(page_title=page.title, status=page.status_string)

# Do not interpolate within the gettext call
_("Page %(page_title)s with status %(status)s" % {"page_title": page.title, "status": page.status_string})
_("Page {page_title} with status {status}").format(page_title=page.title, status=page.status_string)

# Do not use f-string
_("Page {page.title} with status {page.status_string}")
```

Translations with templates

You can import `i18n` and then translate with the `translate`/`blocktranslate` template tags. You can also translate string literals passed as arguments to tags and filters by using the familiar `_()` syntax.

```
{% extends "wagtailadmin/base.html" %}
{% load i18n %}

<!-- preliminary lines of code --&gt;

<!-- Do this to use the translate tag. --&gt;
{% translate "Any string of your choosing" %}

<!-- Do this to use the blocktranslate tag. --&gt;
{% blocktranslate %}
    A multi-line translatable literal.
{% endblocktranslate %}

<!-- Do these to translate string literals passed to tags and filters. --&gt;</pre>
```

(continues on next page)

(continued from previous page)

```

{%- some_tag _("Any string of your choosing") %}
{%- some_tag arg_of_some_tag=_("Any string of your choosing") %}
{%- some_tag value_of_some_tag|filter=_('Any string of your choosing') value|yesno:_( -->
    "yes,no") %}

<!-- A typical example of when to use translation of string literals is -->
{%- translate "example with literal" as var_name %}
{%- some_tag arg_of_some_tag=var_name %}

<!-- If the variable is only ever used once, you could do this instead -->
{%- some_tag arg_of_some_tag=_("example with literal") %}

```

Note: In Wagtail code, you might see `trans` and `blocktrans` instead of `translate` and `blocktranslate`. This still works fine. `trans` and `blocktrans` were the tags earlier on in Django, but were replaced in Django 3.1.

Additional resources

- Translation
- A screen-share [Wagtail Space US 2020 Lightning Talk](#) that walks through using Transifex step-by-step
- Core development instructions for syncing Wagtail translations with Transifex
- [Django docs](#)

Reporting security issues

⚠ Warning

Ensure you are viewing our [latest security policy](#).

ℹ Note

Please report security issues **only** to 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. 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 acknowledgment 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 is `0xbbed227b4daf93ff9`, and this public key is available from most commonly-used keyservers.

This information can also be found in our [security.txt](#).

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.

Bug Bounties

Wagtail does not have a “Bug Bounty” program. Whilst we appreciate and accept reports from anyone, and will gladly give credit to you and/or your organisation, we aren’t able to “reward” you for reporting the vulnerability.

“Beg Bounties” are ever increasing among security researchers, and it’s not something we condone or support.

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. Publish a [security advisory](#) on Wagtail’s GitHub repository. This describes 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 discussion board, Slack workspace and X feed (@WagtailCMS) that links to the security advisory.

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 minimizes the risk of a user handling CSV files insecurely, as they would have to explicitly choose CSV over the more familiar XLSX format.

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. To easily identify the most relevant changes to users, items are grouped together in the following order:

- Major features (no prefix) - things that will inspire users to upgrade to a new release
- Minor enhancements (no prefix) - other improvements to the developer or end user experience
- Bug fixes (prefixed with “Fix:”) - things that address broken behavior from previous releases
- Documentation (prefixed with “Docs:”) - changes to documentation that do not accompany a specific code change; reorganizations, tutorials, recipes and so on
- Maintenance (prefixed with “Maintenance:”) - cleanup, refactoring and other changes to code or tooling that are not intended to have a visible effect to developers or end users

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 for each major feature, under its own heading. Minor enhancements (“Other features”), bug fixes, documentation and maintenance are listed as bullet points under the

appropriate heading - these can be copied from the changelog, with the prefix (“Fix:”, “Docs:” or “Maintenance:”) removed. Backwards compatibility notes should also be included. 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.md` 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 realize that recently 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
```

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. When B is 0, the release contains backwards-incompatible changes.
- 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.

For more information about how Wagtail issues new releases for security purposes, please see our [security policies](#).

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.”

A feature release will usually stop receiving patch release updates when the next feature release comes out.

Long-term support (LTS) release

Certain feature releases will be designated as long-term support releases. These releases will get security and data loss fixes applied for a guaranteed period of time. Typically, a long-term support release will happen once every four feature releases and receive updates for six feature releases, giving a support period of eighteen months with a six month overlap.

Also, long-term support releases will ensure compatibility with at least one [Django long-term support release](#).

Major release

Certain feature releases (A.0, A+1.0, etc.) will be designated as major releases, marked by incrementing the first part of the version number. These releases will contain significant changes to the user interface or backwards-incompatible changes.

Major releases do not happen on a regular schedule. Typically, they will happen when the previous feature releases have accumulated enough deprecated features that it's time to remove them.

Deprecation policy

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. This is especially true for features documented under the [Extending](#) section of the documentation and their corresponding API reference, which tend to be more actively developed.

We try to strike the balance between:

- keeping the API stable for most users,
- documenting features for advanced developers and third-party package maintainers, and
- allowing for continuous improvement of Wagtail's internals.

A feature release may deprecate certain features from previous releases. If a feature is deprecated in feature release A.x, it will continue to work in all A.x versions (for all versions of x) but raise warnings. Deprecated features will be removed in the A+1.0 release, or A+2.0 for features deprecated in the last A.x feature release to ensure deprecations are done over at least 2 feature releases.

For example:

- Wagtail 5.1 was released. Function `func_a()` that entered deprecation in this version would have a backwards-compatible replica which would raise a `RemovedInWagtail160Warning`.
- Wagtail 5.2 was released. This version still contained the backwards-compatible replica of `func_a()`. Future version numbers are provisional, so the next version could either be 5.3 or 6.0. For function `func_b()` that entered deprecation in version 5.2, it would tentatively raise a `RemovedInWagtail160Warning`.
- Wagtail 6.0 was decided to be the next version after Wagtail 5.2. In this release, `func_a()` was outright removed, and `func_b()` would raise a `RemovedInWagtail170Warning` instead.
- When Wagtail 7.0 is released (after all 6.x versions), `func_b()` will be removed.

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 6.1 and 6.2. At this point in time:

- Features will be added to `main`, to be released as Wagtail 6.2.
- Critical bug fixes will be applied to the `stable/6.1.x` branch, and released as 6.1.1, 6.1.3, etc.
- Security fixes and bug fixes for data loss issues will be applied to `main` and to the `stable/6.1.x` and `stable/5.2.x` (LTS) branches. They will trigger the release of 6.1.3, 5.2.6, etc.
- Documentation fixes will be applied to `main`, and, if easily backported, to the latest stable branch, `stable/6.1.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 the release of Wagtail 6.3 and after the following releases:

- Django 4.2 (LTS)
- Django 5.0
- Wagtail 6.2 - Released before Django 5.1 and supports Django 4.2 and 5.0
- Django 5.1

Wagtail 6.3 will support Django 4.2 (LTS), 5.0, 5.1. Wagtail 6.2 will still support only Django 4.2 (LTS) and 5.0.

In some cases, the latest Wagtail feature release falls in between the beta and final release of a new Django version. In such cases, the Wagtail release may add official support for the new Django version in a patch release. An example of this was Wagtail 5.2, which added support for Django 5.0 in Wagtail 5.2.2.

For a list of supported Django and Python versions for each Wagtail release, see the [Compatible Django / Python versions](#) table.

Release schedule

Wagtail uses a [time-based release schedule](#), with feature releases every three months.

Release cycle

Each release cycle consists of three parts:

Phase one: roadmap update

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.

The development team will announce a roadmap update for the next feature release in the form of a request for comments (RFC) to [Wagtail's RFCs repository](#). Anyone is welcome and encouraged to comment on the RFC. After the RFC is approved by the Wagtail core team, the roadmap update will be available on wagtail.org/roadmap.

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.

Patch releases

After a feature release `A.B`, the previous release will go into security support mode.

The branches for the current feature release `stable/A.B.x` and the last LTS release will receive critical bug, security, and data loss fixes.

The branch for the previous feature release `stable/A.B-1.x` will only include security and data loss fixes.

Bugs fixed on `main` must *also* be fixed on other applicable branches; 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 respective branches.

Acknowledgement

This release process is based on [Django's release process](#).

1.11 Release notes

1.11.1 Upgrading Wagtail

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, 6.2 to 6.3).

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, 6.3 to 6.3.1).

Occasionally, feature releases include significant visual changes to the editor interface or backwards-incompatible changes. In these cases, the first part of the version number will be incremented (for example, 5.2 to 6.0).

For dates of past and upcoming releases and support periods, see [Release schedule](#).

Required reading

If it's your first time doing an upgrade, it is highly recommended to read the guide on [Wagtail's release process](#).

Upgrade process

We recommend upgrading one feature release at a time, even if your project is several versions behind the current one. For example, instead of going from 6.0 directly to 6.3, upgrade to 6.1 and 6.2 first. 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.

With that in mind, follow these steps for each feature release you need to upgrade to.

Resolve deprecation warnings

When Wagtail makes a backwards-incompatible change to a publicly-documented feature in a release, it will continue to work in that release, but a deprecation warning will be raised when that feature is used. These warnings are intended to give you advance notice before the support is completely removed in a future release, so that you can update your code accordingly.

In Python, deprecation warnings are silenced by default. You must turn them on using the `-Wa` Python command line option or the `PYTHONWARNINGS` environment variable. For example, to show warnings while running tests:

```
python -Wa manage.py test
```

If you’re not using the Django test runner, you may need to also ensure that any console output is not captured which would hide deprecation warnings. For example, if you use `pytest`:

```
PYTHONWARNINGS=always pytest tests --capture=no
```

Resolve any deprecation warnings with your current version of Wagtail before continuing the upgrade process.

Third party packages might use deprecated APIs in order to support multiple versions of Wagtail, so deprecation warnings in packages you’ve installed don’t necessarily indicate a problem. If a package doesn’t support the latest version of Wagtail, consider raising an issue or sending a pull request to that package.

Preparing for the upgrade

After resolving the deprecation warnings, you should read the [release notes](#) for the next feature release after your current Wagtail version.

Pay particular attention to the upgrade considerations sections (which describe any backwards-incompatible changes) to get a clear idea of what will be needed for a successful upgrade.

Also read the [Compatible Django / Python versions](#) table below, as they may need upgrading first.

Before continuing with the upgrade, make a backup of your database.

Upgrading

To upgrade:

- Update the `wagtail` line in your project’s `requirements.txt` file (or the equivalent, such as `pyproject.toml`) to specify the latest patch release of the version you wish to install. For example, to upgrade to version 6.3.x, the line should read:

```
wagtail>=6.3,<6.4
```

- 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 behavior 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 `STORAGES["staticfiles"]` setting - this ensures that different versions of files are assigned distinct URLs.

Repeat

Repeat the above steps for each feature release you need to upgrade to.

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
6.4	4.2, 5.0, 5.1	3.9, 3.10, 3.11, 3.12, 3.13
6.3 LTS	4.2, 5.0, 5.1	3.9, 3.10, 3.11, 3.12, 3.13
6.2	4.2, 5.0	3.8, 3.9, 3.10, 3.11, 3.12
6.1	4.2, 5.0	3.8, 3.9, 3.10, 3.11, 3.12
6.0	4.2, 5.0	3.8, 3.9, 3.10, 3.11, 3.12
5.2 LTS	3.2, 4.1, 4.2, 5.0 ¹	3.8, 3.9, 3.10, 3.11, 3.12
5.1	3.2, 4.1, 4.2	3.8, 3.9, 3.10, 3.11
5.0	3.2, 4.1, 4.2	3.7, 3.8, 3.9, 3.10, 3.11
4.2	3.2, 4.0, 4.1	3.7, 3.8, 3.9, 3.10, 3.11
4.1 LTS	3.2, 4.0, 4.1	3.7, 3.8, 3.9, 3.10, 3.11
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

continues on next page

Table 1 – continued from previous page

Wagtail release	Compatible Django versions	Compatible Python versions
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

Acknowledgement

This upgrade guide is based on [How to upgrade Django to a newer version](#).

1.11.2 Wagtail 6.4.1 release notes

Unreleased

- [What's new](#)

What's new

Bug fixes

- Prevent error when filtering by locale and searching with Elasticsearch (Sage Abdullah)
- Support searching `none()` querysets (Matt Westcott)
- Correctly handle UUID primary keys when invoking background tasks (Matt Westcott)
- Fix regression where nested sub-menu items would not be visible (Sage Abdullah)
- Ensure the top of the minimap correctly adjusts when resizing the browser viewport (Thibaud Colas)
- Remove obsolete `SubqueryConstraint` check in search backends, for provisional Django 5.2 compatibility (Sage Abdullah)

¹ Added in a patch release

Documentation

- Fix typo in the headless documentation page (Mahmoud Nasser)

Maintenance

- Remove upper version boundary for django-filter (Dan Braghis)
- Relax upper version boundaries for django-taggit and beautifulsoup4 (Matt Westcott)

1.11.3 Wagtail 6.4 release notes

February 3, 2025

- *What's new*
- *Upgrade considerations - changes affecting all projects*
- *Upgrade considerations - changes affecting Wagtail customizations*
- *Upgrade considerations - changes to undocumented internals*

What's new

Support for background tasks using django-tasks

Wagtail now integrates with the [django-tasks](#) library to allow moving computationally-intensive tasks out of the request-response cycle, and improving the performance of the Wagtail admin interface. These tasks include:

- Updating the *search index*
- Updating or creating entries for the *reference index*
- Calculating *image focal points*
- Purging *frontend caching* URLs
- Deleting image and document files when the model is deleted

In the default configuration, these tasks are executed immediately within the request-response cycle, as they were in previous versions of Wagtail. However, by configuring the `TASKS` setting with an appropriate backend [as per the django-tasks documentation](#), these tasks can be deferred to a background worker process.

This feature was developed by Jake Howard.

Previews for StreamField blocks

You can now set up previews for StreamField blocks. The preview will be shown in the block picker, along with a description for the block. This feature can help users choose the right block when writing content inside a StreamField. To enable this feature, see [Configuring block previews](#).

This feature was developed by Sage Abdullah and Thibaud Colas.

Headless documentation

The new [Headless support](#) documentation page curates important information about Wagtail's headless capabilities, directly within the developer documentation. This is the foundation for a [headless improvements roadmap](#) for Wagtail.

Thank you to Sævar Öfjörð Magnússon and Alex Fulcher for creating this new page at the Wagtail Space NL 2024 sprint.

Alt text improvements

Building upon improvements in past versions, this release comes with further enhancements to alt text management:

- The [content modeling accessibility considerations](#) now reflect the latest capabilities for alt text.
- The new ImageBlock alt text is now populated from the image's default alt text when selecting a new image.
- The ImageBlock alt text field is also populated from the image's default alt text when converting from an ImageChooserBlock.
- Alt text quality is now checked by default, as was intended with the alt-text-quality content check.

Thank you to Matt Westcott, Thibaud Colas, and Cynthia Kiser for their work on these improvements.

Shorthand for FieldPanel and InlinePanel

Plain strings can now be used in panel definitions as a substitute for FieldPanel and InlinePanel, avoiding the need to import these classes from wagtail.admin.panels:

```
class MyPage(Page):  
    body = RichTextField()  
    content_panels = [  
        'body',  
    ]
```

This feature was developed by Matt Westcott.

Drag-and-drop support for StreamField and InlinePanel

The StreamField and InlinePanel interfaces now support drag-and-drop reordering of items within a field. This makes it faster to rearrange content within these fields, particularly when there is a lot of content.

This feature was developed by Thibaud Colas and Sage Abdullah, thanks to a sponsorship by Lyst.

Search terms report

For users of [promoted search results](#), a new search terms report is available in the admin interface. This report shows terms that website users have searched for, and how many times they have been searched. This can help you understand what your users are looking for, and adjust your search promotions accordingly.

This feature was developed by Noah van der Meer and Sage Abdullah.

Performance optimizations

Following from a recent audit, this release comes with performance improvements focused on the user interface.

- Prevent main menu from re-rendering when clicking outside while the menu is closed (Sage Abdullah)
- Skip loading of unused JavaScript to speed up 404 page rendering (Sage Abdullah)
- Remove support for Safari 15 (Thibaud Colas)
- Limit tags autocomplete to 10 items and add delay to avoid performance issues with large number of matching tags (Ayushman Singh)

Other features

- Add the ability to apply basic Page QuerySet optimizations to `specific()` sub-queries using `select_related` & `prefetch_related`, see [Page QuerySet reference](#) (Andy Babic)
- Increase `DATA_UPLOAD_MAX_NUMBER_FIELDS` in project template (Matt Westcott)
- Stop invalid Site hostname records from breaking preview (Matt Westcott)
- Set sensible defaults for InlinePanel heading and label (Matt Westcott)
- Add the ability to restrict what types of requests a Pages supports via `allowed_http_methods` (Andy Babic)
- Only allow selection of valid new parents within the copy Page view (Mauro Soche)
- Add `on_serve_page` hook to modify the serving chain of pages (Krystian Magdziarz, Dawid Bugajewski)
- Add support for `WAGTAIL_GRAVATAR_PROVIDER_URL` URLs with query string parameters (Ayaan Qadri, Guilhem Saurel)
- Add `get_avatar_url` hook to customise user avatars (James Harrington)
- Set content security policy (CSP) headers to block embedded content when serving images and documents (Jake Howard, with thanks to Ali İltizar for the initial report)
- Add `page` as a third parameter to the `construct_wagtail_userbar` hook (claudobahn)
- Enable breadcrumbs in revisions compare view (Sage Abdullah)
- Replace i18n library with JavaScript Intl API for time zone options in Account view (Sage Abdullah)
- Use explicit label for defaulting to server language in account settings (Sage Abdullah)
- Add support for specifying an operator on Fuzzy queries (Tom Usher)
- Make sure typing text at the bottom of the page editor always scrolls enough to keep the text into view (Jatin Bhardwaj)

Bug fixes

- Improve handling of translations for bulk page action confirmation messages (Matt Westcott)
- Ensure custom rich text feature icons are correctly handled when provided as a list of SVG paths (Temidayo Azeez, Joel William, LB (Ben) Johnston)
- Prevent error on lazily loading StreamField blocks after the stream has been modified (Stefan Hammer)
- Fix sub-menus within the main menu cannot be closed on mobile (Bojan Mihelac)
- Fix animation overflow transition when navigating through subpages in the sidebar page explorer (manu)
- Ensure form builder supports custom admin form validation (John-Scott Atlakson, LB (Ben) Johnston)
- Ensure form builder correctly checks for duplicate field names when using a custom related name (John-Scott Atlakson, LB (Ben) Johnston)
- Normalize StreamField.get_default() to prevent creation forms from breaking (Matt Westcott)
- Prevent out-of-order migrations from skipping creation of image/document choose permissions (Matt Westcott)
- Use correct connections on multi-database setups in database search backends (Jake Howard)
- Ensure CloudFront cache invalidation is called with a list, for compatibility with current botocore versions (Jake Howard)
- Show the correct privacy status in the sidebar when creating a new page (Joel William)
- Prevent generic model edit view from unquoting non-integer primary keys multiple times (Matt Westcott)
- Ensure comments are functional when editing Page models with read_only FieldPanels in use (Strapchay)
- Ensure the accessible labels and tooltips reflect the correct private/public status on the live link button within pages after changing the privacy (Ayaan Qadri)
- Fix empty th (table heading) elements that are not compliant with accessibility standards (Jai Vignesh J)
- Ensure MultipleChooserPanel using images or documents work when nested within an InlinePanel when no other choosers are in use within the model (Elhussein Almasri)
- Ensure MultipleChooserPanel works after doing a search in the page chooser modal (Matt Westcott)
- Ensure new ListBlock instances get created with unique IDs in the admin client for accessibility and mini-map element references (Srishti Jaiswal)
- Return never-cache HTTP headers when serving pages and documents with view restrictions (Krystian Magdziarz, Dawid Bugajewski)
- Implement get_block_by_content_path on ImageBlock to prevent errors on commenting (Matt Westcott)
- Add aria-expanded attribute to new column button on TypedTableBlock to reflect menu state (Ayaan Qadri, Scott Cranfill)
- Allow page models to extend base Page panel definitions without importing wagtail.admin (Matt Westcott)
- Fix crash when loading the dashboard with only the “unlock” or “bulk delete” page permissions (Unyime Emmanuel Udooh, Sage Abdullah)
- Improve deprecation warning for WidgetWithScript by raising it with stacklevel=3 (Joren Hammudoglu)
- Correctly place comment buttons next to date / datetime / time fields. (Srishti Jaiswal)
- Add missing FilterField("created_at") to AbstractDocument to fix ordering by created_at after searching in the documents index view (Srishti Jaiswal)

- Add missing heading and breadcrumbs in Account view (Sage Abdullah)
- Reduce confusing spacing below StreamField blocks help text (Rishabh Sharma)
- Prevent redundant calls to `Site.find_for_request()` from `Page.get_url_parts()` (Andy Babic)
- Prevent error on listings when searching and filtering by locale (Matt Westcott, Sage Abdullah)
- Add missing space in panels check warning message (Stéphane Blondon)
- Prevent `StreamChildrenToListBlockOperation` from duplicating data across multiple StreamField instances (Joshua Munn)
- Prevent database error when calling `permission_order.register` on app ready (Daniel Kirkham, Matt Westcott)
- Prevent syntax error on MySQL search when query includes symbols (Matt Westcott)

Documentation

- Move the *model reference page* from reference/pages to the references section as it covers all Wagtail core models (Srishti Jaiswal)
- Move the *panels reference page* from references/pages to the references section as panels are available for any model editing, merge panels API into this page (Srishti Jaiswal)
- Move the tags documentation to standalone *advanced topic for tagging*, instead of being inside the reference/pages section (Srishti Jaiswal)
- Refine the *Adding reports* page so that common (page/non-page) class references are at the top and the full page only example has correct heading nesting (Alessandro Chitarrini)
- Add the `wagtail start` command to the management commands reference page (Damilola Oladele)
- Refine the *The project template* page sections and document common issues encountered when creating custom templates (Damilola Oladele)
- Refine titles, references and URLs to better align with the documentation style guide, including US spelling (Srishti Jaiswal)
- Recommend a larger `DATA_UPLOAD_MAX_NUMBER_FIELDS` when *integrating Wagtail into Django* (Matt Westcott)
- Improve code highlighting and formatting for Python docstrings in core models (Srishti Jaiswal)
- Update all JavaScript inline scripts & some CSS inline style tags to a CSP compliant approach by using external scripts/styles (Aayushman Singh)
- Update usage of `mark_safe` to `format_html` for any script inclusions, to better avoid XSS issues from example code (Aayushman Singh)
- Update documentation writing guidelines to *encourage better considerations* of security, accessibility and good practice when writing code examples (Aayushman Singh, LB (Ben) Johnston)
- Update documentation guidelines on writing links and API reference (Sage Abdullah)
- Replace absolute URLs with intersphinx links where possible to avoid broken links (Sage Abdullah)
- Update upgrading guide and release process to better reflect the current practices (Sage Abdullah)
- Document usage of *Custom validation for admin form pages* when using `AbstractEmailForm` or `AbstractForm` pages (John-Scott Atlakson, LB (Ben) Johnston)
- Rework `BlogTagIndexPage` example for clarity (Clifford Gama)
- Change title of blog index page in tutorial to avoid slug issues (Thibaud Colas)

- Fix `wagtailcache` and `wagtailpagecache` examples to not use quotes for the `fragment_name` (Shiv)
- Update tutorial to reflect the move of the “Add child page” action to a top-level button in the header as a ‘+’ icon (Clifford Gama)
- Fix link to `HTTPMethod` in `Page.handle_options_request()` docs (Sage Abdullah)
- Improve `Theory` with added & more consistent section headings and admonitions to aid in readability (Clifford Gama)
- Fix non-functional link to the community guidelines in the `Your first contribution` page (Ankit Kumar)
- Introduce tags and filters by name in the `Writing templates` docs (Clifford Gama)
- Fix Django HTML syntax formatting issue on the `documents overview` page (LB (Ben) Johnston)
- Separate virtual environment creation and activation steps in tutorial (Ankit Kumar)
- Update tutorial to use plain strings in place of `FieldPanel` / `InlinePanel` where appropriate (Unyime Emmanuel Udoh)
- Update example for customizing “`p-as-heading`” accessibility check without overriding built-in checks (Cynthia Kiser)
- Document `get_template method on StreamField blocks` (Matt Westcott)
- Revert incorrect example of appending a `RichTextBlock` to a `StreamField` (Matt Westcott)

Maintenance

- Close open files when reading within `utils/setup.py` (Ataf Fazledin Ahamed)
- Avoid redundant `ALLOWED_HOSTS` check in `Site.find_for_request` (Jake Howard)
- Update `CloneController` to ensure that added/cleared events are not dispatched as cancelable (LB (Ben) Johnston)
- Remove unused `uuid` UMD module as all code is now using the NPM module (LB (Ben) Johnston)
- Clean up JS comments throughout codebase to be aligned to JSDoc where practical (LB (Ben) Johnston)
- Replace `eslint-disable no-undef` linter directives with global comments (LB (Ben) Johnston)
- Upgrade Node tooling to active LTS version 22 (LB (Ben) Johnston)
- Remove defunct oEmbed providers (Rahul Samant)
- Remove obsolete non-upsert-based code for Postgres search indexing (Jake Howard)
- Remove unused Rangy JS library (LB (Ben) Johnston)
- Update `PreviewController` usage to leverage Stimulus actions instead of calling `preventDefault` manually (Ayaan Qadri)
- Various performance optimizations to page publishing (Jake Howard)
- Remove unnecessary DOM `canvas.toBlob` polyfill (LB (Ben) Johnston)
- Ensure Storybook core files are correctly running through Eslint (LB (Ben) Johnston)
- Add a new Stimulus `ZoneController` (`w-zone`) to support dynamic class name changes & event handling on container elements (Ayaan Qadri)
- Migrate jQuery class toggling & drag/drop event handling within the multiple upload views to the Stimulus `ZoneController` usage (Ayaan Qadri)

- Test project template for warnings when run against Django pre-release versions (Sage Abdullah)
- Refactor redirects create/delete views to use generic views (Sage Abdullah)
- Add JSDoc description, adopt linting recommendations, and add more unit tests for `ModalWorkflow` (LB (Ben) Johnston)
- Add support for a `delay` value in `TagController` to debounce async autocomplete tag fetch requests (Aayushman Singh)
- Add unit tests, Storybook stories & JSDoc items for the `Stimulus DrilldownController` (Srishti Jaiswal)
- Enhance sidebar preview performance by eliminating duplicate asset loads on preview error (Sage Abdullah)
- Remove unused `LinkController` (`w-link`) (Sage Abdullah)
- Refactor settings `EditView` to make better use of generic `EditView` (Sage Abdullah)
- Add a new `Stimulus RulesController` (`w-rules`) to support declarative conditional field enabling from other field values in a form (LB (Ben) Johnston)
- Migrate the conditional enabling of fields in the image URL builder view away from ad-hoc jQuery to use the `RulesController` (`w-rules`) approach (LB (Ben) Johnston)
- Enhance the `Stimulus ZoneController` (`w-zone`) to support inactive class and a mechanism to switch the mode based on data within events (Ayaan Qadri)
- Use the `Stimulus ZoneController` (`w-zone`) to remove ad-hoc jQuery for the privacy switch when toggling visibility of private/public elements (Ayaan Qadri)
- Remove unused `is_active` & `active_menu_items` from `wagtail.admin.menu.MenuItem` (Srishti Jaiswal)
- Only call `openpyxl` at runtime to improve performance for projects that do not use `ReportView`, `SpreadsheetExportMixin` and `wagtail.contrib.redirects` (Sébastien Corbin)
- Adopt the update value `mp` instead of `mm` for ‘mystery person’ as the default Gravatar if no avatar found (Harsh Dange)
- Refactor search promotions views to use generic views and templates (Sage Abdullah, Rohit Sharma)
- Use built-in `venv` instead of `pipenv` in CircleCI (Sage Abdullah)
- Add a new `Stimulus FormsetController` (`w-formset`) to support dynamic formset insertion/deletion behavior (LB (Ben) Johnston)
- Enable breadcrumbs by default on admin views using generic templates (Sage Abdullah)
- Refactor pages `revisions_revert` view to be a subclass of `EditView` (Sage Abdullah)
- Move images and documents `get_usage().count()` call to view code (Sage Abdullah)
- Upgrade sass-loader to resolve Sass deprecation warnings (Ayaan Qadri)

Upgrade considerations - changes affecting all projects

DATA_UPLOAD_MAX_NUMBER_FIELDS update

It's recommended that all projects set the DATA_UPLOAD_MAX_NUMBER_FIELDS setting to 10000 or higher.

This specifies the maximum number of fields allowed in a form submission, and it is recommended to increase this from Django's default of 1000, as particularly complex page models can exceed this limit within Wagtail's page editor:

```
# settings.py
DATA_UPLOAD_MAX_NUMBER_FIELDS = 10_000
```

Form builder - validation of fields with custom related_name

On previous releases, form page models (defined through AbstractEmailForm, AbstractForm, FormMixin or EmailFormMixin) did not validate form fields where the related_name was different to the default value of form_fields. As a result, it was possible to create forms with duplicate field names - in this case, only a single field is displayed and captured in the resulting form.

In this new release, validation is now applied on these fields, existing forms will continue to behave as before and no data will be lost. However, editing them will now raise a validation error and users may need to delete any duplicated fields that they had previously missed.

Background tasks run at end of current transaction

In the default configuration, tasks managed by django-tasks (see above) run during the request-response cycle, as before. However, they are now deferred until the current transaction (if any) is committed. If ATOMIC_REQUESTS is set to True, this will be at the end of the request. This may lead to a change of behaviour on views that expect to see the results of these tasks immediately, such as a view that creates a page and then performs a search query to retrieve it. To restore the previous behaviour, set "ENQUEUE_ON_COMMIT": False in the TASKS setting:

```
# settings.py
TASKS = {
    "default": {
        "BACKEND": "django_tasks.backends.immediate.ImmediateBackend",
        "ENQUEUE_ON_COMMIT": False,
    }
}
```

Django's test framework typically runs tests inside transactions, and so this situation is also likely to arise in tests that perform database updates and then - within the same test - expect these changes to be immediately reflected in search queries, object usage counts, and other processes that are now handled with background tasks. This can be addressed by setting ENQUEUE_ON_COMMIT to False as above in the test settings, or by wrapping the database updates in with self.captureOnCommitCallbacks(execute=True) to ensure that these tasks are completed before the test continues:

```
def test_search(self):
    home_page = Page.objects.get(slug="home")

    with self.captureOnCommitCallbacks(execute=True): # Added
        home_page.add_child(instance=EventPage(title="Christmas party"))

    response = self.client.get("/search/?q=Christmas")
    self.assertContains(response, "Christmas party")
```

Upgrade considerations - changes affecting Wagtail customizations

Added page as a third parameter to the `construct_wagtail_userbar` hook

In previous releases, implementations of the `construct_wagtail_userbar` hook were expected to accept two arguments, whereas now `page` is passed in as a third argument.

Old

```
@hooks.register('construct_wagtail_userbar')
def construct_wagtail_userbar(request, items):
    pass
```

New

```
@hooks.register('construct_wagtail_userbar')
def construct_wagtail_userbar(request, items, page):
    pass
```

The old style will now produce a deprecation warning.

Upgrade considerations - changes to undocumented internals

Changes to `content_panels`, `promote_panels` and `settings_panels` values on base Page model

Previously, the `content_panels`, `promote_panels` and `settings_panels` attributes on the base `Page` model were defined as lists of `Panel` instances. These lists now contain instances of `wagtail.models.PanelPlaceholder` instead, which are resolved to `Panel` instances at runtime. Panel definitions that simply extend these lists (such as `content_panels = Page.content_panels + [...]`) are unaffected; however, any logic that inspects these lists (for example, finding a panel in the list to insert a new one immediately after it) will need to be updated to handle the new object types.

Removal of unused Rangy JS library

The unused JavaScript include `wagtailadmin/js/vendor/rangy-core.js` has been removed from the editor interface, and functions such as `window.rangy.getSelection()` are no longer available. Any code relying on this should now either supply its own copy of the [Rangy library](#), or be migrated to the official `Document.createRange()` browser API.

Deprecation of `window.buildExpandingFormset` global function

The undocumented global function `window.buildExpandingFormset` to attach JavaScript insertion / deletion behavior for Django formsets has been deprecated and will be removed in a future release.

Within the Wagtail admin this only impacts a small set of basic expanding formsets in use across Workflow and Group view editing. `InlinePanel` is not affected.

Previously these expanding formsets required a mix of specific `id` attribute structures and inline scripts to instantiate with callbacks for handling deletion. User code implementing this functionality through `buildExpandingFormset` should be updated - the following data attributes can be used to emulate the same behavior. These are likely to change and should not be considered official documentation.

Element	Attribute(s)
Containing element	data-controller="w-formset"
Element to append new child forms	data-w-formset-target="forms"
Child form element	data-w-formset-target="child"
Deleted form element	data-w-formset-target="deleted" hidden
Template element for blank form	data-w-formset-target="template"
Management field (total forms)	data-w-formset-target="totalFormsInput"
Management field (min forms)	data-w-formset-target="minFormsInput"
Management field (max forms)	data-w-formset-target="maxFormsInput"
Management field (Delete, within child form)	data-w-formset-target="deleteInput"
Add child button	data-action="w-formset#add"
Delete child button (within child form)	data-action="w-formset#delete"

Usage of nested id structures are no longer required but can be left in place for easier debugging.

1.11.4 Wagtail 6.3.4 release notes - IN DEVELOPMENT

Unreleased

- [What's new](#)

What's new

Maintenance

- Remove upper version boundary for django-filter (Dan Braghis)

1.11.5 Wagtail 6.3.3 release notes

February 3, 2025

- [What's new](#)

What's new

Bug fixes

- Correctly place comment buttons next to date / datetime / time fields. (Srishti Jaiswal)
- Reduce confusing spacing below StreamField blocks help text (Rishabh Sharma)
- Make sure alt text quality check is on by default as documented (Thibaud Colas)
- Prevent StreamChildrenToListBlockOperation from duplicating data across multiple StreamField instances (Joshua Munn)

- Prevent database error when calling permission_order.register on app ready (Daniel Kirkham, Matt Westcott)
- Prevent error on lazily loading StreamField blocks after the stream has been modified (Stefan Hammer)
- Prevent syntax error on MySQL search when query includes symbols (Matt Westcott)

Documentation

- Update example for customizing “p-as-heading” accessibility check without overriding built-in checks (Cynthia Kiser)
- Update accessibility considerations on alt text in light of contextual alt text improvements (Cynthia Kiser)
- Revert incorrect example of appending a RichTextBlock to a StreamField (Matt Westcott)

1.11.6 Wagtail 6.3.2 release notes

January 2, 2025

- *What's new*

What's new

Bug fixes

- Ensure CloudFront cache invalidation is called with a list, for compatibility with current botocore versions (Jake Howard)
- Ensure Draftail features wrap when a large amount of features are added (Bart Cieński)
- Implement get_block_by_content_path on ImageBlock to prevent errors on commenting (Matt Westcott)

Documentation

- Update tutorial to reflect the move of the “Add child page” action to a top-level button in the header as a ‘+’ icon (Clifford Gama)

1.11.7 Wagtail 6.3.1 release notes

November 19, 2024

- *What's new*

What's new

Bug fixes

- Restore ability to upload profile picture through account settings (Sage Abdullah)
- Correctly handle `ImageChooserBlock` to `ImageBlock` data conversions where all inputs to `bulk_to_python` are null (Storm Heg, Matt Westcott)
- Improve spacing of page / collection permissions table in Group settings (Sage Abdullah)
- Remove forced capitalization of site name on admin dashboard (Thibaud Colas)

Documentation

- Reword `BlogTagIndexPage` example for clarity and several other tweaks (Clifford Gama)
- Change title of blog index page in tutorial to avoid slug issues (Thibaud Colas)
- Fix `wagtailcache` and `wagtailpagecache` examples to not use quotes for the `fragment_name` (Shiv)
- Lower search result ranking for release notes on readthedocs search (Sage Abdullah)

1.11.8 Wagtail 6.3 (LTS) release notes

November 1, 2024

- *What's new*
- *Upgrade considerations - deprecation of old functionality*
- *Upgrade considerations - changes to undocumented internals*

Wagtail 6.3 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 12 months).

What's new

Python 3.13 support

This release adds formal support for Python 3.13.

Django 5.1 support

This release adds formal support for Django 5.1.

ImageBlock with alt text support

This release introduces a new block type `ImageBlock`, which improves upon `ImageChooserBlock` by allowing editors to specify alt text tailored to the context in which the image is used. This is the new recommended block type for all images that are not purely decorative, and existing instances of `ImageChooserBlock` can be directly replaced with `ImageBlock` (with no data migration or template changes required) to benefit from contextual alt text. This feature was developed by Chiemezuo Akujobi as part of the Google Summer of Code program with mentoring support from Storm Heg, Saptak Sengupta, Thibaud Colas and Matt Westcott.

Incremental dashboard enhancements

The Wagtail dashboard design evolves towards providing more information and navigation features. Mobile support is much improved. Upgrade banners are now dismissible.

This feature was developed by Albina Starykova and Sage Abdullah, based on designs by Ben Enright.

Enhanced contrast admin theme

CMS users can now control the level of contrast of UI elements in the admin interface. This new customization is designed for partially sighted users, complementing existing support for a dark theme and Windows Contrast Themes. The new “More contrast” theming can be enabled in account preferences, or will otherwise be derived from operating system preferences.

This feature was designed thanks to feedback from our blind and partially sighted users, and was developed by Albina Starykova based on design input from Victoria Ottah.

Universal design

This release follows through with “universal listings” user experience and design consistency improvements earlier in 2024, with the following features.

- All create/edit admin forms now use a sticky submit button, for consistency and to speed up edits
- Secondary form actions such as “Delete” are now in the header actions menu, for consistency and to make the actions more easily reachable for keyboard users
- Documents and Images views now use universal listings styles
- Page type usage, workflow usage, and workflow history views also use universal listings styles
- The forms pages listing now supports search and filtering

These features were developed by Sage Abdullah.

HEIC / HEIF image upload support

The `WAGTAILIMAGES_EXTENSIONS` setting now accepts the `.heic` extension, which allows users to upload and use HEIC / HEIF images in Wagtail. These images are automatically converted to JPEG format when rendered. For more details, see [HEIC / HEIF images](#).

This feature was developed by Matt Westcott.

Custom preview sizes support

You can now customize the preview device sizes available in the live preview panel by overriding `preview_sizes`. The default size can also be set by overriding `default_preview_size`.

This feature was developed by Bart Cieliński, alexkiro, and Sage Abdullah.

Other features

- Formalize support for MariaDB (Sage Abdullah, Daniel Black)
- Redirect to the last viewed listing page after deleting form submissions (Matthias Brück)
- Provide `getTextLabel` method on date / time StreamField blocks (Vaughn Dickson)
- Purge frontend cache when modifying redirects (Jake Howard)
- Add search and filters to form pages listing (Sage Abdullah)
- Deprecate the `WAGTAIL_AUTO_UPDATE_PREVIEW` setting, use `WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL = 0` instead (Sage Abdullah)
- Consistently use `capfirst` for title-casing model verbose names (Sébastien Corbin)
- Fire `copy_for_translation_done` signal when copying translatable models as well as pages (Coen van der Kamp)
- Add support for an `image_description` field across all images, to better support accessible image descriptions (Chiemezuo Akujobi)
- Prompt the user about unsaved changes when editing snippets (Sage Abdullah)
- Add support for specifying different preview modes to the “View draft” URL for pages (Robin Varghese)
- Implement new designs for the footer actions dropdown with more contrast and larger text (Sage Abdullah)
- `StaticBlock` now renders nothing by default when no template is specified (Sævar Öfjörð Magnússon)
- Allow setting page privacy rules when a restriction already exists on an ancestor page (Bojan Mihelac)
- Automatically create links when pasting content that contain URLs into a rich text input (Thibaud Colas)
- Add Uyghur language support

Bug fixes

- Prevent page type business rules from blocking reordering of pages (Andy Babic, Sage Abdullah)
- Improve layout of object permissions table (Sage Abdullah)
- Fix typo in aria-label attribute of page explorer navigation link (Sébastien Corbin)
- Reinstate transparency indicator on image chooser widgets (Sébastien Corbin)
- Remove table headers that have no text (Matt Westcott)
- Fix broken link to user search (Shlomo Markowitz)
- Ensure that JS slugify function strips Unicode characters disallowed by Django slug validation (Atif Khan)
- Do not show notices about root / unroutable pages when searching or filtering in the page explorer (Matt Westcott)
- Resolve contrast issue for page deletion warning (Sanjeev Holla S)
- Make sure content metrics falls back to body element only when intended (Sage Abdullah)
- Remove wrongly-added filters from redirects index (Matt Westcott)
- Prevent popular tags filter from generating overly complex queries when not filtering (Matt Westcott)
- Fix content path links in usage view to scroll to the correct element (Sage Abdullah)
- Always show the minimap toggle button (Albina Starykova)
- Ensure invalid submissions are marked as dirty edits on load to trigger UI and browser warnings for unsaved changes, restoring previous behavior from Wagtail 5.2 (Sage Abdullah)
- Update polldaddy oEmbed provider to use the crowdignal URL (Matthew Scouten)
- Remove polleverywhere oEmbed provider as it this application longer supports oEmbed (Matthew Scouten)
- Ensure that dropdown button toggles show with a border in high contrast mode (Ishwari8104, LB (Ben) Johnston)
- Update email notification header to the new logo design (rahulsamant37)
- Change `file_size` field on document model to avoid artificial 2Gb limit (Gabriel Getzie)
- Ensure that `TypedTableBlock` uses the correct API representations of child blocks (Matt Westcott)
- Footer action buttons now include their `media` definitions (Sage Abdullah)
- Improve the text contrast of the bulk actions “Select all” button (Sage Abdullah)
- Fix error on workflow settings view with multiple snippet types assigned to the same workflow on Postgres (Sage Abdullah)
- Prevent history view from breaking if a log entry’s revision is missing (Matt Westcott)
- Prevent long filenames from breaking layout on document chooser listings (Frank Yiu, Shaurya Panchal)
- Fix datetime fields overflowing its parent wrapper in listing filters (Rohit Singh)
- Prevent multiple URLs from being combined into one when pasting links into a rich text input (Thibaud Colas)
- Improve layout of report listing tables (Sage Abdullah)
- Fix regression from creation of `AbstractGroupApprovalTask` to ensure `can_handle` checks for the abstract class correctly (Sumana Sree Angajala)

Documentation

- Upgrade Sphinx to 7.3 (Matt Westcott)
- Upgrade sphinx-wagtail-theme to v6.4.0, with a new search integration and Read the Docs Addons bug fixes (Thibaud Colas)
- Document how to *customize date/time format settings* (Vince Salvino)
- Create a new documentation section for *deployment* and move `fly.io` deployment from the tutorial to this section (Vince Salvino)
- Clarify process for *UserViewSet customization* (Sage Abdullah)
- Correct `WAGTAIL_WORKFLOW_REQUIRE_REAPPROVAL_ON_EDIT` documentation to state that it defaults to `False` (Matt Westcott)
- Add an example of customizing a default accessibility check in the *Authoring accessible content* section (Cynthia Kiser)
- Demonstrate access protection with `TokenAuthentication` in the *Wagtail API v2 Configuration Guide* (Krzysztof Jeziorny)
- Replace X links with Mastodon in the README (Alex Morega)
- Re-enable building offline formats in online documentation (Read the docs) for EPUB/PDF/HTML downloads (Joel William, Sage Abdullah)
- Resolve multiple output errors in the documentation ePub format (Sage Abdullah)
- Update social media examples to use LinkedIn, Reddit, Facebook (Ayaan Qadri)

Maintenance

- Removed support for Python 3.8 (Matt Westcott)
- Drop `pytz` dependency in favour of `zoneinfo.available_timezones` (Sage Abdullah)
- Relax `django-taggit` dependency to allow 6.0 (Matt Westcott)
- Improve page listing performance (Sage Abdullah)
- Phase out usage of `SECRET_KEY` in version and icon hashes (Jake Howard)
- Audit all use of localized and non-localized numbers in templates (Matt Westcott)
- Refactor StreamField `get_prep_value` for closer alignment with `JSONField` (Sage Abdullah)
- Move search implementation logic from generic `IndexView` to `BaseListingView` (Sage Abdullah)
- Upgrade Puppeteer integration tests for reliability (Matt Westcott)
- Restore ability to use `.in_bulk()` on specific querysets under Django 5.2a0 (Sage Abdullah)
- Add generated `test-media` to `.gitignore` (Shlomo Markowitz)
- Improve `debounce` util's return type for better TypeScript usage (Sage Abdullah)
- Ensure the side panel's show event is dispatched after any hide events (Sage Abdullah)
- Migrate preview-panel JavaScript to Stimulus & TypeScript, add full unit testing (Sage Abdullah)
- Move `wagtailConfig` values from inline scripts to the `wagtail_config` template tag (LB (Ben) Johnston, Sage Abdullah)

- Deprecate the `{% locales %}` and `{% js_translation_strings %}` template tags (LB (Ben) Johnston, Sage Abdullah)
- Adopt the modern best practice for `beforeunload` usage in `UnsavedController` to trigger a leave page warning when edits have been made (Shubham Mukati, Sage Abdullah)
- Ensure multi-line comments are cleaned from custom icons in addition to just single line comments (Jake Howard)
- Deprecate `window.wagtailConfig.BULK_ACTION_ITEM_TYPE` usage in JavaScript to reduce reliance on inline scripts (LB (Ben) Johnston)
- Remove `window.fileupload_opts` usage in JavaScript, use data attributes on fields instead to reduce reliance on inline scripts (LB (Ben) Johnston)
- Remove `image_format_name_to_content_type` helper function that duplicates Willow functionality (Matt Westcott)
- Improve code reuse for footer actions markup across generic views (Sage Abdullah)
- Deprecate internal `DeleteMenuItem` API for footer actions (Sage Abdullah)
- Update Pillow dependency to allow 11.x (Storm Heg)

Upgrade considerations - deprecation of old functionality

Removed support for Python 3.8

Python 3.8 is no longer supported as of this release; please upgrade to Python 3.9 or above before upgrading Wagtail.

Deprecation of the `WAGTAIL_AUTO_UPDATE_PREVIEW` setting

The `WAGTAIL_AUTO_UPDATE_PREVIEW` setting has been deprecated and will be removed in a future release.

To disable the automatic preview update feature, set `WAGTAIL_AUTO_UPDATE_PREVIEW_INTERVAL = 0` in your Django settings instead.

Upgrade considerations - changes to undocumented internals

Deprecation of `window.wagtailConfig.BULK_ACTION_ITEM_TYPE`

As part of migrating away from inline scripts, the undocumented use of `window.wagtailConfig.BULK_ACTION_ITEM_TYPE` as a global has been deprecated and will be removed in a future release.

Old

```
{% block extra_js %}
    {{ block.super }}
    <script>
        window.wagtailConfig.BULK_ACTION_ITEM_TYPE = 'SOME_ITEM';
    </script>
{% endblock %}
```

New

Update usage of the `wagtailadmin/bulk_actions/footer.html` template include to declare the `item_type`.

```
{% block bulk_actions %}  
  {% include 'wagtailadmin/bulk_actions/footer.html' ... item_type="SOME_ITEM" %}  
{% endblock %}
```

Note

Custom item types for bulk actions are not officially supported yet and this approach is likely to get further changes in the future.

Deprecation of the `{% locales %}` template tag

The undocumented `locales` template tag will be removed in a future release.

If access to JSON locales within JavaScript is needed, use `window.wagtailConfig.LOCALES` instead.

Deprecation of the `{% js_translation_strings %}` template tag

The undocumented `js_translation_strings` template tag will be removed in a future release.

If access to JSON translation strings within JavaScript is needed, use `window.wagtailConfig.STRINGS` instead.

UpgradeNotificationPanel is no longer removable with `construct_homepage_panels` hook

The upgrade notification panel can still be removed with the `WAGTAIL_ENABLE_UPDATE_CHECK = False` setting.

SiteSummaryPanel is no longer removable with `construct_homepage_panels` hook

The summary items can still be removed with the `construct_homepage_summary_items` hook.

Deprecation of DeleteMenuItem

The undocumented `DeleteMenuItem` API will be removed in a future release.

The delete option is now provided via `EditView.get_header_more_buttons()`, though this is still an internal-only API.

1.11.9 Wagtail 6.2.3 release notes

November 1, 2024

- [What's new](#)

What's new

Bug fixes

- Prevent multiple URLs from being combined into one when pasting links into a rich text input (Thibaud Colas)
- Fix error on workflow settings view with multiple snippet types assigned to the same workflow on Postgres (Sage Abdullah)
- Prevent history view from breaking if a log entry's revision is missing (Matt Westcott)

Documentation

- Upgrade sphinx-wagtail-theme to v6.4.0, with a new search integration and Read the Docs Addons bug fixes (Thibaud Colas)

1.11.10 Wagtail 6.2.2 release notes

September 24, 2024

- *What's new*

What's new

Bug fixes

- Fix various instances of USE_THOUSAND_SEPARATOR formatting numbers where formatting is invalid (Sébastien Corbin, Matt Westcott)
- Fix broken link to user search (Shlomo Markowitz)
- Make sure content metrics falls back to body element only when intended (Sage Abdullah)
- Remove wrongly-added filters from redirects index (Matt Westcott)
- Prevent popular tags filter from generating overly complex queries when not filtering (Matt Westcott)

Documentation

- Clarify process for *UserViewSet customization* (Sage Abdullah)

1.11.11 Wagtail 6.2.1 release notes

August 20, 2024

- [What's new](#)

What's new

Bug fixes

- Handle `child_block` being passed as a kwarg in ListBlock migrations (Matt Westcott)
- Fix broken task type filter in workflow task chooser modal (Sage Abdullah)
- Prevent circular imports between `wagtail.admin.models` and custom user models (Matt Westcott)
- Ensure that concurrent editing check works for users who only have edit access via workflows (Matt Westcott)

1.11.12 Wagtail 6.2 release notes

August 1, 2024

- [What's new](#)
- [Upgrade considerations - changes affecting all projects](#)
- [Upgrade considerations - changes affecting Wagtail customizations](#)
- [Upgrade considerations - changes to undocumented internals](#)

What's new

Content metrics

The page editor's Checks panel now displays two content metrics: word count, and reading time. They are calculated based on the contents of the page preview, with a new mechanism to extract content from the previewed page for processing within the page editor. The Checks panel has also been redesigned to accommodate a wider breadth of types of checks, and interactive checks, in future releases.

This feature was developed by Albina Starykova and sponsored by The Motley Fool.

Concurrent editing notifications

When multiple users concurrently work on the same content, Wagtail now displays notifications to inform them of potential editing conflicts. When a user saves their work, other users are informed and presented with options: they can refresh the page to view the latest changes, or proceed with their own changes, overwriting the other user's work.

Concurrent editing notifications are available for pages, and snippets. Specific messaging about conflicting versions is only available for pages and snippets with [support for saving revisions](#). To configure how often notifications are updated, use `WAGTAIL_EDITING_SESSION_PING_INTERVAL`.

This feature was implemented by Matt Westcott and Sage Abdullah.

Alt text accessibility check

The [built-in accessibility checker](#) now enforces a new alt-text-quality rule, which tests alt text for the presence of known bad patterns such as file extensions and underscores. This rule is enabled by default, but can be disabled if necessary.

This feature was implemented by Albina Starykova, with support from the Wagtail accessibility team.

Universal listings designs for report views

All built-in and custom report views now use the Universal Listings visual design and filtering features introduced in all other listings in the admin interface over past releases. Thank you to Sage Abdullah for implementing this feature and continuing the rollout of the new designs.

Compact StreamField representation for migrations

StreamField definitions within migrations are now represented in a more compact form, where blocks that appear in multiple places within a StreamField structure are only defined once. For complex and deeply-nested StreamFields, this considerably reduces the size of migration files, and the memory consumption when loading them. This feature was developed by Matt Westcott.

Other features

- Optimize and consolidate redirects report view into the index view (Jake Howard, Dan Braghis)
- Support a [`HOSTNAMES` parameter on `WAGTAILFRONTENDCACHE`](#) to define which hostnames a backend should respond to (Jake Howard, sponsored by Oxfam America)
- Refactor redirects edit view to use the generic `EditView` and breadcrumbs (Rohit Sharma)
- Allow custom permission policies on snippets to prevent superusers from creating or editing them (Sage Abdullah)
- Do not link to edit view from listing views if user has no permission to edit (Sage Abdullah)
- Allow access to snippets and other model viewsets to users with “View” permission (Sage Abdullah)
- Skip `ChooseParentView` if only one possible valid parent page is available (Matthias Brück)
- Add `copy_for_translation_done` signal when a page is copied for translation (Arnar Tumi Þorsteinsson)
- Remove reduced opacity for draft page title in listings (Inju Michorius)
- Implement a new design for locale labels in listings (Albina Starykova)

- Add a `deactivate()` method to `ProgressController` (Alex Morega)
- Allow manually specifying credentials for CloudFront frontend cache backend (Jake Howard)
- Automatically register permissions for models registered with a `ModelViewSet` (Sage Abdullah)
- Make `routable_resolver_match` attribute available on `RoutablePageMixin` responses (Andy Chosak)
- Support customizations to `UserViewSet` via the app config (Sage Abdullah)
- Allow changing available privacy options per page model (Shlomo Markowitz)
- Add “soft” client-side validation for `StreamBlock` / `ListBlock` `min_num` / `max_num` (Matt Westcott)
- Log accessibility checker results in the console to help developers with troubleshooting (Thibaud Colas)
- Disable pointer events on checker highlights to simplify DevTools inspections (Thibaud Colas)

Bug fixes

- Make `WAGTAILIMAGES_CHOOSER_PAGE_SIZE` setting functional again (Rohit Sharma)
- Enable `richtext` template tag to convert lazy translation values (Benjamin Bach)
- Ensure permission labels on group permissions page are translated where available (Matt Westcott)
- Preserve whitespace in comment replies (Elhussein Almasri)
- Address layout issues in the title cell of universal listings (Sage Abdullah)
- Support SVG icon id attributes with single quotes in the styleguide (Sage Abdullah)
- Do not show delete button on model edit views if per-instance permissions prevent deletion (Matt Westcott)
- Remove duplicate header in privacy dialog when a privacy setting is set on a parent page or collection (Matthias Brück)
- Allow renditions of `.ico` images (Julie Rymer)
- Fix the rendering of grouped choices when using `ChoiceFilter` in combination with `choices` (Sébastien Corbin)
- Add separators when displaying multiple error messages on a `StructBlock` (Kyle Bayliss)
- Specify `verbose_name` on `TranslatableMixin.locale` so that it is translated when used as a label (Romein van Buren)
- Disallow null characters in API filter values (Jochen Wersdörfer)
- Fix image preview when Willow optimizers are enabled (Alex Tomkins)
- Ensure external-to-internal link conversion works when the `wagtail_serve` view is on a non-root path (Sage Abdullah)
- Add missing `for_instance` method to `PageLogEntryManager` (Matt Westcott)
- Ensure that “User” column on history view is translatable (Romein van Buren)
- Handle StreamField migrations where the field value is null (Joshua Munn)
- Prevent incorrect menu ordering when order value is 0 (Ben Dickinson)
- Fix dynamic image serve view with certain backends (Sébastien Corbin)
- Show not allowed extension in error message (Sahil Jangra)
- Fix focal point chooser when localization enabled (Sébastien Corbin)

- Ensure that system checks for `WAGTAIL_DATE_FORMAT`, `WAGTAIL_DATETIME_FORMAT` and `WAGTAIL_TIME_FORMAT` take `FORMAT_MODULE_PATH` into account (Sébastien Corbin)
- Prevent rich text fields inside choosers from being duplicated when opened repeatedly (Sage Abdullah)

Documentation

- Remove duplicate section on frontend caching proxies from performance page (Jake Howard)
- Document `restriction_type` field on [`PageViewRestriction`](#) (Shlomo Markowitz)
- Document Wagtail's bug bounty policy (Jake Howard)
- Fix incorrect Sphinx-style code references to use MyST style (Byron Peebles)
- Document the fact that `Orderable` is not required for inline panels (Bojan Mihelac)
- Add note about `prefers-reduced-motion` to the [`accessibility documentation`](#) (Roel Koper)
- Update deployment instructions for Fly.io (Jeroen de Vries)
- Add better docs for generating URLs on [`creating admin views`](#) (Shlomo Markowitz)
- Document the `vary_fields` property for [`custom image filters`](#) (Daniel Kirkham)
- Fix documentation build errors (Himanshu Garg, Chris Shenton)
- Fix PDF export (Nathanaël Jourdane)

Maintenance

- Use `DjangoJSONEncoder` instead of custom `LazyStringEncoder` to serialize Draftail config (Sage Abdullah)
- Refactor image chooser pagination to check `WAGTAILIMAGES_CHOOSER_PAGE_SIZE` at runtime (Matt Westcott)
- Exclude the `client/scss` directory in Tailwind content config to speed up CSS compilation (Sage Abdullah)
- Split `contrib.frontend_cache.backends` into dedicated sub-modules (Andy Babic)
- Remove unused `docs/autobuild.sh` script (Sævar Öfjörð Magnússon)
- Replace `urlparse` with `urlsplit` to improve performance (Jake Howard)
- Optimize embed finder lookups (Jake Howard)
- Improve performance of initial admin loading by moving sprite hashing out of module import time (Jake Howard)
- Remove workaround and inline scripts for activating workflow actions (Sage Abdullah)
- Prevent '`BlockWidget`' object has no attribute '`_block_json`' from masking errors during StreamField serialization (Matt Westcott)

Upgrade considerations - changes affecting all projects

Specifying a dict of distribution IDs for CloudFront cache invalidation is deprecated

Previous versions allowed passing a dict for DISTRIBUTION_ID within the WAGTAILFRONTENDCACHE configuration for a CloudFront backend, to allow specifying different distribution IDs for different hostnames. This is now deprecated; instead, multiple distribution IDs should be defined as *multiple backends*, with a HOSTNAMES parameter to define the hostnames associated with each one. For example, a configuration such as:

```
WAGTAILFRONTENDCACHE = {
    'cloudfront': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudfrontBackend',
        'DISTRIBUTION_ID': {
            'www.wagtail.org': 'your-distribution-id',
            'www.madewithwagtail.org': 'other-distribution-id',
        },
    },
}
```

should now be rewritten as:

```
WAGTAILFRONTENDCACHE = {
    'mainsite': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudfrontBackend',
        'DISTRIBUTION_ID': 'your-distribution-id',
        'HOSTNAMES': ['www.wagtail.org'],
    },
    'madewithwagtail': {
        'BACKEND': 'wagtail.contrib.frontend_cache.backends.CloudfrontBackend',
        'DISTRIBUTION_ID': 'other-distribution-id',
        'HOSTNAMES': ['www.madewithwagtail.org'],
    },
}
```

Changes to permissions registration for models with ModelViewSet and SnippetViewSet

Models registered with a ModelViewSet will now automatically have their `Permission` objects registered in the Groups administration area. Previously, you need to use the `register_permissions` hook to register them.

If you have a model registered with a ModelViewSet and you registered the model's permissions using the `register_permissions` hook, you can now safely remove the hook.

If the viewset has `inspect_view_enabled` set to True, all permissions for the model are registered. Otherwise, the “view” permission is excluded from the registration.

To customize which permissions get registered for the model, you can override the `get_permissions_to_register()` method.

This behavior now applies to snippets as well. Previously, the “view” permission for snippets is always registered regardless of `inspect_view_enabled`. If you wish to register the “view” permission, you can enable the inspect view:

```
class FooViewSet(SnippetViewSet):
    ...
    inspect_view_enabled = True
```

Alternatively, if you wish to register the “view” permission without enabling the inspect view (i.e. the previous behavior), you can override `get_permissions_to_register` like the following:

```
from django.contrib.auth.models import Permission
from django.contrib.contenttypes.models import ContentType

class FooViewSet(SnippetViewSet):
    def get_permissions_to_register(self):
        content_type = ContentType.objects.get_for_model(self.model)
        return Permission.objects.filter(content_type=content_type)
```

Deprecation of `WAGTAIL_USER_EDIT_FORM`, `WAGTAIL_USER_CREATION_FORM`, and `WAGTAIL_USER_CUSTOM_FIELDS` settings

This release introduces a customizable `UserViewSet` class, which can be used to customize various aspects of Wagtail's admin views for managing users, including the form classes for creating and editing users. As a result, the `WAGTAIL_USER_EDIT_FORM`, `WAGTAIL_USER_CREATION_FORM`, and `WAGTAIL_USER_CUSTOM_FIELDS` settings have been deprecated in favor of customizing the form classes via `UserViewSet.get_form_class()`.

If you use the aforementioned settings, you can migrate your code by making the following changes.

Before

Given the following custom user model:

```
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)
```

The following custom forms:

```
class CustomUserEditForm(UserEditForm):
    status = forms.ModelChoiceField(queryset=MembershipStatus.objects, required=True,
    ↳ label=_("Status"))

class CustomUserCreationForm(UserCreationForm):
    status = forms.ModelChoiceField(queryset=MembershipStatus.objects, required=True,
    ↳ label=_("Status"))
```

And the following settings:

```
WAGTAIL_USER_EDIT_FORM = "myapp.forms.CustomUserEditForm"
WAGTAIL_USER_CREATION_FORM = "myapp.forms.CustomUserCreationForm"
WAGTAIL_USER_CUSTOM_FIELDS = ["country", "status"]
```

After

Change the custom forms to the following:

```
class CustomUserEditForm(UserEditForm):
    status = forms.ModelChoiceField(queryset=MembershipStatus.objects, required=True, ↴
                                     label=_("Status"))

    # Use ModelForm's automatic form fields generation for the model's `country` ↴
    # field,
    # but use an explicit custom form field for `status`.
    # This replaces the `WAGTAIL_USER_CUSTOM_FIELDS` setting.
    class Meta(UserEditForm.Meta):
        fields = UserEditForm.Meta.fields + {"country", "status"}


class CustomUserCreationForm(UserCreationForm):
    status = forms.ModelChoiceField(queryset=MembershipStatus.objects, required=True, ↴
                                     label=_("Status"))

    # Use ModelForm's automatic form fields generation for the model's `country` ↴
    # field,
    # but use an explicit custom form field for `status`.
    # This replaces the `WAGTAIL_USER_CUSTOM_FIELDS` setting.
    class Meta(UserCreationForm.Meta):
        fields = UserEditForm.Meta.fields + {"country", "status"}
```

Create a custom UserViewSet subclass in e.g. myapp/viewsets.py:

```
# myapp/viewsets.py
from wagtail.users.views.users import UserViewSet as WagtailUserViewSet

from .forms import CustomUserCreationForm, CustomUserEditForm


class UserViewSet(WagtailUserViewSet):
    # This replaces the WAGTAIL_USER_EDIT_FORM and WAGTAIL_USER_CREATION_FORM settings
    def get_form_class(self, for_update=False):
        if for_update:
            return CustomUserEditForm
        return CustomUserCreationForm
```

If you already have a custom GroupViewSet as described in [Customizing group edit/create views](#), you can reuse the custom WagtailUsersAppConfig subclass. Otherwise, create an apps.py file within your project folder (the one containing the top-level settings and urls modules) e.g. myproject/apps.py. Then, create a custom WagtailUsersAppConfig subclass in that file, with a user_viewset attribute pointing to the custom UserViewSet subclass:

```
# myproject/apps.py
from wagtail.users.apps import WagtailUsersAppConfig


class CustomUsersAppConfig(WagtailUsersAppConfig):
    user_viewset = "myapp.viewsets.UserViewSet"
    # If you have customized the GroupViewSet before
    group_viewset = "myapp.viewsets.GroupViewSet"
```

Replace wagtail.users in settings.INSTALLED_APPS with the path to CustomUsersAppConfig:

```
INSTALLED_APPS = [
    ...,
    # Make sure you have two separate entries for the custom user model's app
    # and the custom app config for the wagtail.users app
    "myapp", # an app that contains the custom user model
    "myproject.apps.CustomUsersAppConfig", # a custom app config for the wagtail.
    ↪users app
    # "wagtail.users", # this should be removed in favour of the custom app config
    ...,
]
```

⚠ Warning

You can also place the `WagtailUsersAppConfig` subclass inside the same `apps.py` file of your custom user model's app (instead of in a `myproject/apps.py` file), but you need to be careful. Make sure to use two separate config classes instead of turning your existing `AppConfig` subclass into a `WagtailUsersAppConfig` subclass, as that would cause Django to pick up your custom user model as being part of `wagtail.users`. You may also need to set `default` to `True` in your own app's `AppConfig`, unless you already use a dotted path to the app's `AppConfig` subclass in `INSTALLED_APPS`.

For more details, see [Creating a custom UserViewSet](#).

Upgrade considerations - changes affecting Wagtail customizations

Changes to report views with the new Universal Listings UI

The report views have been reimplemented to use the new Universal Listings UI, which introduces AJAX-based filtering and support for the `wagtail.admin.ui.tables` framework. As a result, a number of changes have been made to the `ReportView` and `PageReportView` classes, as well as their templates.

If you have custom report views as documented in [Adding reports](#), you will need to make the following changes.

Change title to page_title

The `title` attribute on the view class should be renamed to `page_title`:

```
class UnpublishedChangesReportView(PageReportView):
-    title = "Pages with unpublished changes"
+    page_title = "Pages with unpublished changes"
```

Set up the results-only view

Set the `index_url_name` and `index_results_url_name` attributes on the view class:

```
class UnpublishedChangesReportView(PageReportView):
+    index_url_name = "unpublished_changes_report"
+    index_results_url_name = "unpublished_changes_report_results"
```

and register the results-only view:

```
@hooks.register("register_admin_urls")
def register_unpublished_changes_report_url():
    return [
        path("reports/unpublished-changes/", UnpublishedChangesReportView.as_view(), name="unpublished_changes_report"),
        # Add a results-only view to add support for AJAX-based filtering
        path("reports/unpublished-changes/results/", UnpublishedChangesReportView.as_view(results_only=True), name="unpublished_changes_report_results"),
    ]
```

Adjust the templates

If you are only extending the templates to add your own markup for the listing table (and not other parts of the view template), you need to change the `template_name` into `results_template_name` on the view class.

For a page report, the following changes are needed:

- Change `template_name` to `results_template_name`, and optionally rename the template (e.g. `reports/unpublished_changes_report.html` to `reports/unpublished_changes_report_results.html`).
- The template should extend from `wagtailadmin/reports/base_page_report_results.html`.
- The `listing` and `no_results` blocks should be renamed to `results` and `no_results_message`, respectively.

```
class UnpublishedChangesReportView(PageReportView):
-    template_name = "reports/unpublished_changes_report.html"
+    results_template_name = "reports/unpublished_changes_report_results.html"
```

```
{# <project>/templates/reports/unpublished_changes_report_results.html #-}

-{% extends "wagtailadmin/reports/base_page_report.html" %}
+{% extends "wagtailadmin/reports/base_page_report_results.html" %}

-{% block listing %}
+{% block results %}
    {% include "reports/include/_list_unpublished_changes.html" %}
    {% endblock %}
-{% block no_results %}
+{% block no_results_message %}
    <p>No pages with unpublished changes.</p>
    {% endblock %}
```

For a non-page report, the following changes are needed:

- Change `template_name` to `results_template_name`, and optionally rename the template (e.g. `reports/custom_non_page_report.html` to `reports/custom_non_page_report_results.html`).
- The template should extend from `wagtailadmin/reports/base_report_results.html`.
- Existing templates will typically define a `results` block containing both the results listing and the “no results” message; these should now become separate blocks named `results` and `no_results_message`.

Before:

```
class CustomNonPageReportView(ReportView):
    template_name = "reports/custom_non_page_report.html"
```

```
{# <project>/templates/reports/custom_non_page_report.html #}
{&gt; extends "wagtailadmin/reports/base_report.html" %}

{&gt; block results %}
    {&gt; if object_list %}
        <table class="listing">
            <!-- Table markup goes here -->
        </table>
    {&gt; else %}
        <p>No results found.</p>
    {&gt; endif %}
{&gt; endblock %}
```

After:

```
class CustomNonPageReportView(ReportView):
    results_template_name = "reports/custom_non_page_report_results.html"
```

```
{# <project>/templates/reports/custom_non_page_report_results.html #}
{&gt; extends "wagtailadmin/reports/base_report_results.html" %}

{&gt; block results %}
    <table class="listing">
        <!-- Table markup goes here -->
    </table>
{&gt; endblock %}

{&gt; block no_results_message %}
    <p>No results found.</p>
{&gt; endblock %}
```

If you need to completely customize the view's template, you can still override the `template_name` attribute on the view class. Note that both `ReportView` and `PageReportView` now use the `wagtailadmin/reports/base_report.html` template, which now extends the `wagtailadmin/generic/listing.html` template. The `wagtailadmin/reports/base_page_report.html` template is now unused and should be replaced with `wagtailadmin/reports/base_report.html`.

If you override `template_name`, it is still necessary to set `results_template_name` to a template that extends `wagtailadmin/reports/base_report_results.html` (or `wagtailadmin/reports/base_page_report_results.html` for page reports), so the view can correctly update the listing and show the active filters as you apply or remove any filters.

Upgrade considerations - changes to undocumented internals

Deprecation of `window.ActivateWorkflowActionsForDashboard` and `window.ActivateWorkflowActionsForEditView`

The undocumented usage of the JavaScript `window.ActivateWorkflowActionsForDashboard` and `window.ActivateWorkflowActionsForEditView` functions will be removed in a future release.

These functions are only used by Wagtail to initialize event listeners to workflow action buttons via inline scripts and are never intended to be used in custom code. The inline scripts have been removed in favour of initializing the event

listeners directly in the included `workflow-action.js` script. As a result, the functions no longer need to be globally-accessible.

Any custom workflow actions should be done using the [documented approach for customizing the behavior of custom task types](#), such as by overriding `Task.get_actions()`.

1.11.13 Wagtail 6.1.3 release notes

July 11, 2024

- [What's new](#)

What's new

CVE-2024-39317: Regular expression denial-of-service via search query parsing

This release addresses a denial-of-service vulnerability in Wagtail. A bug in Wagtail's `parse_query_string` would result in it taking a long time to process suitably crafted inputs. When used to parse sufficiently long strings of characters without a space, `parse_query_string` would take an unexpectedly large amount of time to process, resulting in a denial of service.

In an initial Wagtail installation, the vulnerability can be exploited by any Wagtail admin user. It cannot be exploited by end users. If your Wagtail site has a custom search implementation which uses `parse_query_string`, it may be exploitable by other users (e.g. unauthenticated users).

Many thanks to Jake Howard for reporting and fixing this issue. For further details, please see [the CVE-2024-39317 security advisory](#).

Bug fixes

- Allow renditions of `.ico` images (Julie Rymer)
- Handle choice groups as dictionaries in active filters (Sébastien Corbin)
- Fix image preview when Willow optimizers are enabled (Alex Tomkins)
- Fix dynamic image serve view with certain backends (Sébastien Corbin)

1.11.14 Wagtail 6.1.2 release notes

May 30, 2024

- [What's new](#)

What's new

CVE-2024-35228: Improper handling of insufficient permissions in `wagtail.contrib.settings`

This release addresses a permission vulnerability in the Wagtail admin interface. Due to an improperly applied permission check in the `wagtail.contrib.settings` module, a user with access to the Wagtail admin and knowledge of the URL of the edit view for a settings model can access and update that setting, even when they have not been granted permission over the model. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to Victor Miti for reporting this issue, and Matt Westcott and Jake Howard for the fix. For further details, please see [the CVE-2024-35228 security advisory](#).

Bug fixes

- Fix client-side handling of select inputs within `ChoiceBlock` (Matt Westcott)
- Support SVG icon id attributes with single quotes in the styleguide (Sage Abdullah)

1.11.15 Wagtail 6.1.1 release notes

May 21, 2024

- [What's new](#)

What's new

Bug fixes

- Fix form action URL in user edit and delete views for custom user models (Sage Abdullah)
- Fix snippet copy view not prefilling form data (Sage Abdullah)
- Address layout issues in the title cell of universal listings (Sage Abdullah)
- Fix incorrect rich text to HTML conversion when multiple link / embed types are present (Andy Chosak, Matt Westcott)
- Restore ability for custom widgets in StreamField blocks to have multiple top-level nodes (Sage Abdullah, Matt Westcott)

1.11.16 Wagtail 6.1 release notes

May 1, 2024

- [What's new](#)
- [Upgrade considerations - changes affecting Wagtail customizations](#)
- [Upgrade considerations - changes to undocumented internals](#)

What's new

Universal listings continued

Continuing work on the Universal Listings project, this release rolls out universal listing styles for the following views:

- Image listings
- Document listings
- Site and locale listings
- Page and snippet history views
- Form builder submissions
- Collections listings
- Groups
- Users
- Workflow and task views
- Search promotions index views
- Redirects index

Universal listing components like header buttons have also been tweaked to improve usability, and the `PageList-ingViewSet` now includes `ChooseParentView` to allow creating pages from custom page listings.

Thank you to everyone who worked on these features: Ben Enright, Sage Abdullah, Rohit Sharma, Storm Heg, Temidayo Azeez, and Abdelrahman Hamada.

Information-dense admin interface

Wagtail now provides a way for CMS users to control the information density of the admin interface, via their user profile preferences. The new setting allows switching between the “default” density and a new “snug” mode, which reduces the spacing and size of UI elements.

It's also possible for site implementers to customize this for the needs of their project – see [Custom UI information density](#).

This feature was developed by Ben Enright and Thibaud Colas.

Custom per-page-type listings

A new viewset class `PageListingViewSet` has been introduced, allowing developers to create custom page listings for specific page types similar to those previously provided by the `Modeladmin` package - see [Custom page listings](#). This feature was developed by Matt Westcott.

Keyboard shortcuts overview

A new dialog is available from the help menu, providing an overview of keyboard shortcuts available in the Wagtail admin. This feature was developed by Karthik Ayangar and Rohit Sharma.

Better guidance for password-protected content

Wagtail now includes extra guidance in its `private pages` and `private collections (documents)` forms, to warn users about the pitfalls of the “shared password” option. For projects with higher security requirements, it’s now possible to disable the shared password option entirely. Thank you to Rohit Sharma, Salvo Polizzi, and Jake Howard for implementing those changes.

Favicon images generation

For sites managing favicons via the CMS, Wagtail now supports `.ico favicon generation`, with `format-ico`:

```
<link rel="icon" href="{% image favicon_image format-ico %}" />
```

This feature was developed by Jake Howard.

CVE-2024-32882: Permission check bypass when editing a model with per-field restrictions through `wagtail.contrib.settings` OR `ModelViewSet`

This release addresses a permission vulnerability in the Wagtail admin interface. If a model has been made available for editing through the `wagtail.contrib.settings` module or `ModelViewSet`, and the permission argument on `FieldPanel` has been used to further restrict access to one or more fields of the model, a user with edit permission over the model but not the specific field can craft an HTTP POST request that bypasses the permission check on the individual field, allowing them to update its value.

The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin, or by a user who has not been granted edit access to the model in question. The editing interfaces for pages and snippets are also unaffected.

Many thanks to Ben Morse and Joshua Munn for reporting this issue, and Jake Howard and Sage Abdullah for the fix. For further details, please see [the CVE-2024-32882 security advisory](#).

Other features

- Add `RelatedObjectsColumn` to the table UI framework (Matt Westcott)
- Reduce memory usage when rebuilding search indexes (Jake Howard)
- Add system checks to ensure that `WAGTAIL_DATE_FORMAT`, `WAGTAIL_DATETIME_FORMAT`, `WAGTAIL_TIME_FORMAT` are *correctly configured* (Rohit Sharma, Coen van der Kamp)
- Allow custom permissions with the same prefix as built-in permissions (Sage Abdullah)
- Allow displaying permissions linked to the Admin model's content type (Sage Abdullah)
- Add support for Draftail's JavaScript to use `chooserUrls` provided by entity options & for the Draftail widget to encode lazy URLs/ translations (Elhussein Almasri)
- Added `AbstractGroupApprovalTask` to simplify *customizing behavior of custom Task models* (John-Scott Atlakson)
- Add ability to bulk toggle permissions in the user group editing view, including shift+click for multiple selections (LB (Ben) Johnston, Kalob Taulien)
- Update the minimum version of `djangorestframework` to 3.15.1 (Sage Abdullah)
- Add support for related fields in generic `IndexView.list_display` (Abdelrahman Hamada)
- Improve page fetching logic and cache route results per request. You can now use `Page.route_for_request()` to find the page route, and `Page.find_for_request()` to find the page given a request object and a URL path, see [Page](#). Results are cached on `request._wagtail_route_for_request` (Gordon Pendleton)
- Optimize rewriting of links / embeds in rich text using bulk database lookups (Andy Chosak)
- Add normalization mechanism to StreamField so that assignments and defaults can be passed in a wider range of data types (Joshua Munn, Matt Westcott)
- Allow specifying a `STORAGES` alias name for `WAGTAILIMAGES_RENDERING_STORAGE` (Alec Baron)
- Update `PASSWORD_REQUIRED_TEMPLATE` setting to `WAGTAIL_PASSWORD_REQUIRED_TEMPLATE` with deprecation of previous naming (Saksham Misra, LB (Ben) Johnston)
- Update `DOCUMENT_PASSWORD_REQUIRED_TEMPLATE` setting to `WAGTAIL_DOCS_PASSWORD_REQUIRED_TEMPLATE` with deprecation of previous naming (Saksham Misra, LB (Ben) Johnston)
- When editing settings (contrib) use the same icon in the editing view that was declared when registering the setting (Vince Salvino, Rohit Sharma)
- Populate django-treebeard cache during page routing to improve performance of `get_parent` (Nigel van Keulen)
- Add additional field types to Elasticsearch mapping (scott-8)

Bug fixes

- Fix typo in `__str__` for MySQL search index (Jake Howard)
- Ensure that unit tests correctly check for migrations in all core Wagtail apps (Matt Westcott)
- Correctly handle `date` objects on `human_readable_date` template tag (Jhonatan Lopes)
- Ensure re-ordering buttons work correctly when using a nested `InlinePanel` (Adrien Hamraoui)
- Consistently remove model's `verbose_name` in group edit view when listing custom permissions (Sage Abdullah, Neeraj Yetheendran, Omkar Jadhav)
- Resolve issue local development of docs when running `make livehtml` (Sage Abdullah)
- Resolve issue with unwanted padding in chooser modal listings (Sage Abdullah)
- Ensure form builder emails that have date or datetime fields correctly localize dates based on the configured `LANGUAGE_CODE` (Mark Niehues)
- Ensure the Stimulus `UnsavedController` checks for nested removal/additions of inputs so that the unsaved warning shows in more valid cases when editing a page (Karthik Ayangar)
- Ensure `get_add_url()` is always used to re-render the add button when the listing is refreshed in viewsets (Sage Abdullah)
- Ensure dropdown content cannot get higher than the viewport and add scrolling within content if needed (Chiemezuo Akujobi)
- Prevent snippets model index view from crashing when a model does not have an `objects` manager (Jhonatan Lopes)
- Fix `get_dummy_request`'s resulting host name when running tests with `ALLOWED_HOSTS = ["*"]` (David Buxton)
- Fix timezone handling in the `timesince_last_update` template tag (Matt Westcott)
- Fix Postgres phrase search to respect the language set in settings (Ihar Marhitych)
- Retain query parameters when switching between locales in the page chooser (Abdelrahman Hamada, Sage Abdullah)
- Add `w-kbd-scope-value` with support for `global` so that specific keyboard shortcuts (e.g. `ctrl+s/cmd+s`) trigger consistently even when focused on fields (Neeraj Yetheendran)
- Improve exception handling when generating image renditions concurrently (Andy Babic)
- Respect `WAGTAIL_ALLOW_UNICODE_SLUGS` setting when auto-generating slugs (LB (Ben) Johnston)
- Use correct URL when redirecting back to page search results after an AJAX search (Sage Abdullah)
- Reinstate missing static files in style guide (Sage Abdullah)
- Provide `convert_mariadb_uuids` management command to assist with upgrading to Django 5.0+ on MariaDB (Matt Westcott)

Documentation

- Add contributing development documentation on how to work with a [fork of Wagtail](#) (Nix Asteri, Dan Braghis)
- Make sure the settings panel is listed in tabbed interface examples (Tibor Leupold)
- Update content and page names to their US spelling instead of UK spelling (Victoria Poromon)
- Update broken and incorrect links throughout the documentation (EK303)
- Fix formatting of `--purge-only` in [`wagtail_update_image_renditions`](#) management command section (Pranith Beeram)
- Update [`template components`](#) documentation to better explain the usage of the Laces library (Tibor Leupold)
- Update Sphinx theme to `6.3.0` with a fix for the missing favicon (Sage Abdullah)
- Document risk of XSS attacks on document upload in [`WAGTAILDOCS_SERVE_METHOD`](#) and example settings (Matt Westcott, with thanks to Georgios Roumeliotis of TwelveSec for the original report)
- Add clarity to how custom [`StreamField validation`](#) works (Tibor Leupold)
- Add additional reference to the [`wagtail_update_image_renditions`](#) management command on the [`using images`](#) page (LB (Ben) Johnston)
- Correct information about line endings in Window development docs (Sage Abdullah)
- Improve code snippets for “[`Create a footer for all pages`](#)” tutorial section (Drikus Roor)
- Update list of third-party tutorials (LB (Ben) Johnston)
- Update [`Integrating into Django`](#) documentation to emphasise creating page models (Matt Westcott)

Maintenance

- Move RichText HTML whitelist parser to use the faster, built in `html.parser` rather than `html5lib` (Jake Howard)
- Remove duplicate ‘path’ in `default_exclude_fields_in_copy` (Ramchandra Shahi Thakuri)
- Update unit tests to always use the faster, built in `html.parser` & remove `html5lib` dependency (Jake Howard)
- Adjust Eslint rules for TypeScript files (Karthik Ayangar)
- Rename the React Button that only renders links (`a` element) to `Link` and remove unused prop & behaviors that was non-compliant for aria role usage (Advik Kabra)
- Set up an `wagtail.models.AbstractWorkflow` model to support future customizations around workflows (Hossein)
- Improve `classnames` template tag to handle nested lists of strings, use template tag for admin `body` element (LB (Ben) Johnston)
- Merge `UploadedDocument` and `UploadedImage` into new `UploadedFile` model for easier shared code usage (Advik Kabra, Karl Hobley)
- Optimize queries in dashboard panels (Sage Abdullah)
- Optimize queries in group create/edit view (Sage Abdullah)
- Move `modal-workflow.js` script usage to base admin template instead of ad-hoc imports (Elhussein Almasri)
- Update all Draftail chooserUrls to be passed in via the Entity options instead of using `window.chooserUrls` globals, removing the need for inline scripts (Elhussein Almasri)

- Enhance `w-init` (`InitController`) to support a `detail` value to be dispatched on events (Chiemezuo Akujobi)
- Remove usage of inline scripts and instead use event dispatching to instantiate standalone Draftail editor instances (Chiemezuo Akujobi)
- Refactor `page_breadcrumbs` tag to use shared `breadcrumbs.html` template (Sage Abdullah)
- Add `keyboard` icon to admin icon set (Rohit Sharma)
- Remove dead code in the minimap when elements are not found (LB (Ben) Johnston)
- Ensure untrusted data sources are logged correctly in the Stimulus `SwapController` (LB (Ben) Johnston)
- Update Wagtail logo in admin sidebar & favicon plus documentation to the latest version (Osaf AliSayed, Albina Starykova, LB (Ben) Johnston)
- Remove usage of inline scripts and instead use a new Stimulus controller (`w-block/BlockController`) to instantiate `StreamField` blocks (Karthik Ayangar)
- Update NPM Babel, TypeScript and Webpack packages (Neeraj Yetheendran)
- Replace ad-hoc JavaScript and vendor Mousetrap usage to a new Stimulus controller (`w-kbd/KeyboardController`) (Neeraj Yetheendran)
- Update `django-filter` to 24.x (Sebastian Muthwill)
- Remove jQuery usage in telepath widget classes (Matt Westcott)
- Remove `xregexp` (IE11 polyfill) along with `window.XRegExp` global util (LB (Ben) Johnston)
- Refactor the Django port of `urlify` to use TypeScript, officially deprecate `window.URLify` global util (LB (Ben) Johnston)

Upgrade considerations - changes affecting Wagtail customizations

Changes to `SubmissionsListView` class

As part of the Universal Listings project, the `SubmissionsListView` for listing form submissions in the `wagtail.contrib.forms` app has been refactored to become a subclass of `wagtail.admin.views.generic.base.BaseListView`. As a result, the class has undergone a number of changes, including the following:

- The filtering mechanism has been reimplemented to use `django-filter`.
- The `ordering` attribute has been renamed to `default_ordering`.
- The template used to render the view has been significantly refactored to use the new universal listings UI.

`register_user_listing_buttons` hook signature changed

The function signature for the `register_user_listing_buttons` hook was updated to accept a `request_user` argument instead of `context`. If you use this hook, make sure to update your function signature to match the new one. The old signature with `context` will continue to work for now, but the `context` only contains the `request` object. Support for the old signature will be removed in a future release.

PASSWORD_REQUIRED_TEMPLATE has changed to WAGTAIL_PASSWORD_REQUIRED_TEMPLATE

The setting `PASSWORD_REQUIRED_TEMPLATE` has been deprecated, it will continue to work until a future release but the new name for this same setting will be `WAGTAIL_PASSWORD_REQUIRED_TEMPLATE` to align with other settings naming conventions.

See [Frontend authentication](#).

`DOCUMENT_PASSWORD_REQUIRED_TEMPLATE` has changed to `WAGTAILDOCS_PASSWORD_REQUIRED_TEMPLATE`

The setting `DOCUMENT_PASSWORD_REQUIRED_TEMPLATE` has been deprecated, it will continue to work until a future release but the new name for this same setting will be `WAGTAILDOCS_PASSWORD_REQUIRED_TEMPLATE` to align with other settings naming conventions.

See [Frontend authentication](#).

Upgrade considerations - changes to undocumented internals

Removal of `html5lib` dependency

Wagtail now uses `html.parser` for its rich text processing, and no longer depends on `html5lib`. If your project relies on `html5lib`, update rich text code to use Wagtail's documented rich text APIs like [rewrite handlers](#) and [format converters](#). Or update project dependencies to include `html5lib`.

Deprecation of `user_listing_buttons` template tag

The undocumented `user_listing_buttons` template tag has been deprecated and will be removed in a future release.

Deprecation of `wagtailusers_groups:users` URL pattern

The undocumented `wagtailusers_groups:users` URL pattern has been deprecated and will be removed in a future release. If you are using `reverse` with this URL pattern name, you should update your code to use the `wagtailusers_users:index` URL pattern name and the group ID as the `group` query parameter. For example:

```
reverse('wagtailusers_groups:users', args=[group.id])
```

should be updated to:

```
reverse('wagtailusers_users:index') + f'?group={group.id}'
```

The corresponding `wagtailusers_groups:users_results` URL pattern has been removed as part of this change.

A redirect from the old URL pattern to the new one has been added to ensure that existing URLs continue to work. This redirect will be removed in a future release.

Deprecation of `window.chooserUrls` within Draftail choosers

The undocumented usage of the JavaScript `window.chooserUrls` within Draftail choosers will be removed in a future release.

The following `chooserUrl` object values will be impacted.

- `anchorLinkChooser`
- `documentChooser`
- `emailLinkChooser`
- `embedsChooser`
- `externalLinkChooser`
- `imageChooser`
- `pageChooser`

Overriding these inner values on the global `window.chooserUrls` object will still override their usage in the Draftail choosers for now but this capability will be removed in a future release.

Example

It's recommended that usage of this global is removed in any customizations or third party packages and instead a custom Draftail Entity be used instead. See example below.

Old

```
# .../wagtail_hooks.py

@hooks.register("insert_editor_js")
def editor_js():
    return format_html(
        """
        <script>
            window.chooserUrls.myCustomChooser = '{0}';
        </script>
        """,
        reverse("myapp_chooser:choose"),
    )
```

New

Remove the `insert_editor_js` hook usage and instead pass the data needed via the Entity's data.

```
# .../wagtail_hooks.py

from django.urls import reverse_lazy

@hooks.register("register_rich_text_features")
def register_my_custom_feature(features):
    # features.register_link_type...
```

(continues on next page)

(continued from previous page)

```

features.register_editor_plugin(
    "draftail",
    "custom-link",
    draftail_features.EntityFeature(
        {
            "type": "CUSTOM_ITEM",
            "icon": "doc-full-inverse",
            "description": gettext_lazy("Item"),
            "chooserUrls": {
                # Important: `reverse_lazy` must be used unless the URL path is
                ↪hard-coded
                "myChooser": reverse_lazy("myapp_chooser:choose")
            },
            js=["..."],
        },
    )
)

```

Overriding existing chooserUrls values

To override existing Entities' chooserUrls values, here is an example of an unsupported monkey patch can achieve a similar goal.

However, it's recommended that a *custom Entity be created* to be registered as a replacement feature for Draftail customizations as per the documentation.

```

# .../wagtail_hooks.py
from django.urls import reverse_lazy

from wagtail import hooks

@hooks.register("register_rich_text_features")
def override_embed_feature_url(features):
    features.plugins_by_editor["draftail"]["embed"].data["chooserUrls"]["embedsChooser"]
    ↪""] = reverse_lazy("my_embeds:chooser")

```

Deprecation of `window.initBlockWidget` to initialize a StreamField block

The undocumented global function `window.initBlockWidget` has now been deprecated and will be removed in a future release.

Any complex customizations that have re-implemented parts of this functionality will need to be modified to adopt the new approach that uses Stimulus and avoids inline scripts.

The usage of this new approach is still unofficial and could change in the future, it's recommended that the documented `BlockWidget` and `StreamField` approaches be used instead.

However, for comparison, here is the old and new approach below.

Old

Assuming we are using Django's `format_html` to prepare the HTML output with `JSON.dumps` strings for the block data values.

The old approach would call the `window.initBlockWidget` global function with an inline script as follows:

```
<div
  id="{id}"
  data-block="{block_json}"
  data-value="{value_json}"
  data-error="{error_json}"
></div>
<script>
  initBlockWidget('{id}');
</script>
```

New

In the new approach, we no longer need to attach an inline script but instead use the Stimulus data attributes to attach the behavior with the `w-block` identifier as follows:

```
<div
  id="{id}"
  data-block
  data-controller="w-block"
  data-w-block-data-value="{block_json}"
  data-w-block-arguments-value="[{value_json}, {error_json}]"
></div>
```

Removal of jQuery from base client-side Widget and BoundWidget classes

The JavaScript base classes `Widget` and `BoundWidget` that provide client-side access to form widgets (see [Form widget client-side API](#)) no longer use jQuery. The `input` property of `BoundWidget` (previously a jQuery collection) is now a native DOM element, and the `element` argument passed to the `BoundWidget` constructor (previously a jQuery collection) is now passed as a native DOM element if the HTML representation consists of a single element, and an iterable of elements (`NodeList` or array) otherwise. User code that extends these classes should be updated accordingly.

`window.URLify` deprecated

The undocumented client-side global util `window.URLify` is now deprecated and will be removed in a future release.

If this was required for slug field behavior, it's recommended that the `SlugInput` widget be used instead. This will automatically convert values entered into a suitable slug in the browser while respecting the global configuration `WAGTAIL_ALLOW_UNICODE_SLUGS`.

```
from wagtail.admin.widgets.slug import SlugInput
# ... other imports

class MyPage(Page):
    promote_panels = [
```

(continues on next page)

(continued from previous page)

```
FieldPanel("slug", widget=SlugInput),  
    # ... other panels  
]
```

If you require this for custom JavaScript functionality, it's recommended you either include your own implementation from the original [Django URLify source](#). Alternatively, the `slugify` or `parameterize` (a Django URLify port) NPM packages might be suitable.

window.XRegExp polyfill removed

The legacy `window.XRegExp` global polyfill util has been removed and will throw an error if called.

Instead, any usage of this should be updated to the well supported [browser native Regex implementation](#).

```
// old  
const newStr = XRegExp.replace(originalStr, XRegExp(' [ab+c] ', 'g'), '')  
  
// new (with RegExp)  
const newStr = originalStr.replace(new RegExp(' [ab+c] ', 'g'), '')  
// OR  
const newStr = originalStr.replace(/ [ab+c] /g, '')
```

1.11.17 Wagtail 6.0.6 release notes

July 11, 2024

- [What's new](#)

What's new

CVE-2024-39317: Regular expression denial-of-service via search query parsing

This release addresses a denial-of-service vulnerability in Wagtail. A bug in Wagtail's `parse_query_string` would result in it taking a long time to process suitably crafted inputs. When used to parse sufficiently long strings of characters without a space, `parse_query_string` would take an unexpectedly large amount of time to process, resulting in a denial of service.

In an initial Wagtail installation, the vulnerability can be exploited by any Wagtail admin user. It cannot be exploited by end users. If your Wagtail site has a custom search implementation which uses `parse_query_string`, it may be exploitable by other users (e.g. unauthenticated users).

Many thanks to Jake Howard for reporting and fixing this issue. For further details, please see [the CVE-2024-39317 security advisory](#).

1.11.18 Wagtail 6.0.5 release notes

May 30, 2024

- *What's new*

What's new

CVE-2024-35228: Improper handling of insufficient permissions in `wagtail.contrib.settings`

This release addresses a permission vulnerability in the Wagtail admin interface. Due to an improperly applied permission check in the `wagtail.contrib.settings` module, a user with access to the Wagtail admin and knowledge of the URL of the edit view for a settings model can access and update that setting, even when they have not been granted permission over the model. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to Victor Miti for reporting this issue, and Matt Westcott and Jake Howard for the fix. For further details, please see [the CVE-2024-35228 security advisory](#).

1.11.19 Wagtail 6.0.4 release notes

May 21, 2024

- *What's new*

What's new

Bug fixes

- Fix snippet copy view not prefilling form data (Sage Abdullah)

1.11.20 Wagtail 6.0.3 release notes

May 1, 2024

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

What's new

CVE-2024-32882: Permission check bypass when editing a model with per-field restrictions through `wagtail.contrib.settings` or `ModelViewSet`

This release addresses a permission vulnerability in the Wagtail admin interface. If a model has been made available for editing through the `wagtail.contrib.settings` module or `ModelViewSet`, and the permission argument on FieldPanel has been used to further restrict access to one or more fields of the model, a user with edit permission over the model but not the specific field can craft an HTTP POST request that bypasses the permission check on the individual field, allowing them to update its value.

The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin, or by a user who has not been granted edit access to the model in question. The editing interfaces for pages and snippets are also unaffected.

Many thanks to Ben Morse and Joshua Munn for reporting this issue, and Jake Howard and Sage Abdullah for the fix. For further details, please see [the CVE-2024-32882 security advisory](#).

Bug fixes

- Respect `WAGTAIL_ALLOW_UNICODE_SLUGS` setting when auto-generating slugs (LB (Ben) Johnston)
- Use correct URL when redirecting back to page search results after an AJAX search (Sage Abdullah)
- Reinstate missing static files in style guide (Sage Abdullah)
- Provide `convert_mariadb_uuids` management command to assist with upgrading to Django 5.0+ on MariaDB (Matt Westcott)
- Fix generic CopyView for models with primary keys that need to be quoted (Sage Abdullah)

Upgrade considerations

Changes to UUID fields on MariaDB when upgrading to Django 5.0

Django 5.0 introduces support for MariaDB's native UUID type on MariaDB 10.7 and above. This breaks backwards compatibility with CHAR-based UUIDs created on earlier versions of Django and MariaDB, and so upgrading a site to Django 5.0+ and MariaDB 10.7+ is liable to result in errors such as `Data too long for column 'translation_key' at row 1` or `Data too long for column 'uuid' at row 1` when creating or editing pages. To fix this, it is necessary to run the `convert_mariadb_uuids` management command (available as of Wagtail 6.0.3) after upgrading:

```
./manage.py convert_mariadb_uuids
```

This will convert all existing UUID fields used by Wagtail to the new format. New sites created under Django 5.0+ and MariaDB 10.7+ are unaffected.

1.11.21 Wagtail 6.0.2 release notes

April 3, 2024

- [What's new](#)

What's new

Bug fixes

- Ensure that modal tabs width are not impacted by side panel opening (LB (Ben) Johnston)
- Resolve issue local development of docs when running `make livehtml` (Sage Abdullah)
- Resolve issue with unwanted padding in chooser modal listings (Sage Abdullah)
- Ensure `get_add_url()` is always used to re-render the add button when the listing is refreshed in viewsets (Sage Abdullah)
- Move `modal-workflow.js` script usage to base admin template instead of ad-hoc imports so that choosers work in `ModelViewSets` (Elhussein Almasri)
- Ensure JavaScript for common widgets such as `InlinePanel` is included by default in `ModelViewSet`'s create and edit views (Sage Abdullah)
- Reinstate styles for customizations of `extra_footer_actions` block in page create/edit templates (LB (Ben) Johnston, Sage Abdullah)
- Prevent crash when loading an empty table block in the editor (Sage Abdullah)

Documentation

- Update Sphinx theme to 6.3.0 with a fix for the missing favicon (Sage Abdullah)

1.11.22 Wagtail 6.0.1 release notes

February 15, 2024

- [What's new](#)

What's new

Bug fixes

- Ensure BooleanRadioSelect uses the same styles as RadioSelect (Thibaud Colas)
- Prevent failure on collectstatic when ManifestStaticFilesStorage is in use (Matt Westcott)
- Prevent error on submitting an empty search in the admin under Elasticsearch (Maikel Martens)

1.11.23 Wagtail 6.0 release notes

February 7, 2024

- *What's new*
- *Upgrade considerations - removal of deprecated features from Wagtail 4.2 - 5.1*
- *Upgrade considerations - changes affecting all projects*
- *Upgrade considerations - deprecation of old functionality*
- *Upgrade considerations - changes affecting Wagtail customizations*
- *Upgrade considerations - changes to undocumented internals*

What's new

Django 5.0 support

This release adds support for Django 5.0. The support has also been backported to Wagtail 5.2 LTS.

New developer tutorial

A new *developer tutorial* series has been added to the documentation. This series builds upon the pre-existing *Your first Wagtail site*, going through the creation and deployment of a portfolio website.

This tutorial series was created by Damilola Oladele as part of the Google Season of Docs program, with support from Meagen Voss, and Thibaud Colas. We also thank Storm Heg, Kalob Taulien, Kátia Nakamura, Mariusz Felisiak, and Rachel Smith for their support and feedback as part of the project.

Universal listings

Following design improvements to the page listing view, Wagtail now provides a unified search and filtering interface for all listings. This will improve navigation capabilities, particularly for sites with a large number of pages or where content tends to use a flat structure.

In this release, the universal listing interface is available for Pages, Snippets, and Forms. For pages, the UI includes the following filters out of the box:

- Page type

- Date updated
- Owner
- Edited by
- Site
- Has child pages
- Locale

This feature was developed by Ben Enright, Matt Westcott, Nick Lee, Thibaud Colas, and Sage Abdullah.

Right-to-left language support

The admin interface now supports right-to-left languages, such as Persian, Arabic, and Hebrew. Though there are still some areas that need improvement, all admin views will now be displayed in the correct direction. Review our [UI guidelines](#) for guidance on supporting right-to-left languages in admin interface customizations.

Thank you to Thibaud Colas, Badr Fourane, and Sage Abdullah for their work on this long-requested improvement.

Accessibility checker in page editor

The [built-in accessibility checker](#) now displays as a side panel within page and snippet editors supporting preview. The new “Checks” side panel only shows accessibility-related issues for pages with the userbar enabled in this release, but will be updated to support [any content checks](#) in the future.

This feature was implemented by Nick Lee, Thibaud Colas, and Sage Abdullah.

Page types usage report

The new Page types report provides a breakdown of the number of pages for each type. It helps answer questions such as:

- Which page types do we have on our CMS?
- How many pages of that page type do we have?
- When was a page of that type last edited? By whom? Which page was that?

This feature was developed by Jhonatan Lopes, as part of a sponsorship by the Mozilla Foundation.

Accessibility improvements

This release comes with a high number of accessibility improvements across the admin interface.

- Improve layout and accessibility of the image URL generator page, reduce reliance on JavaScript (Temidayo Azeez)
- Remove overly verbose image captions in image listings for screen readers (Sage Abdullah)
- Ensure screen readers and dictation tools can more easily navigate bulk actions in images, documents and page listings by streamlining labels and descriptions (Sage Abdullah)
- Add optional caption field to `TypedTableBlock` (Tommaso Amici, Cynthia Kiser)
- Switch the `TableBlock` header controls to a field that requires user input (Bhuvnesh Sharma, Aman Pandey, Cynthia Kiser)
- Add support for `caption` on admin UI Table component (Aman Pandey)

- Replace legacy dropdown component with new Tippy dropdown-button (Thibaud Colas)
- Ensure the sidebar account toggle has no duplicate accessible labels (Nandini Arora)
- Ensure that page listing re-ordering messages and accessible labels can be translated (Aman Pandey, LB (Ben) Johnston)
- Resolve multiple issues with page listing re-ordering using keyboard and screen readers (Aman Pandey)
- When using an empty table header (`th`) for visual spacing, ensure this is ignored by accessibility tooling (V Rohitansh)
- Ensure that TableBlock cells are accessible when using keyboard control only (Elhussein Almasri)

Other features

- Added `search_index option` to `StreamField` blocks to control whether the block is indexed for searching (Vedant Pandey)
- Remember previous location on returning from page add/edit actions (Robert Rollins)
- Update settings file in project settings to address Django 4.2 deprecations (Sage Abdullah)
- Allow `UniqueConstraint` in place of `unique_together` for `TranslatableMixin`'s system check (Temidayo Azeez, Sage Abdullah)
- Make use of `IndexView.get_add_url()` in snippets index view template (Christer Jensen, Sage Abdullah)
- Allow `Page.permissions_for_user()` to be overridden by specific page types (Sébastien Corbin)
- Improve visual alignment of explore icon in Page listings for longer content (Krzysztof Jeziorny)
- Add `extra_actions` blocks to Snippets and generic index templates (Bhuvnesh Sharma)
- Add support for defining panels / `edit_handler` on `ModelViewSet` (Sage Abdullah)
- Use a single instance of `PagePermissionPolicy` in `wagtail.permissions` module (Sage Abdullah)
- Add max tag length validation for multiple uploads (documents/images) (Temidayo Azeez)
- Ensure expanded side panel does not overlap form content for most viewports (Chiemezu Akujobi)
- Add ability to `modify the default ordering` for the page explorer view (Shlomo Markowitz)
- Remove support for Safari 14 (Thibaud Colas)
- Add ability to click to copy the URL in the image URL generator page (Sai Srikanth Dumperi)
- Show edit as a main action in generic history and usage views (Sage Abdullah)
- Make styles for header buttons consistent (Sage Abdullah)
- Improve styles of slim header's search and filters (Sage Abdullah)
- Change page listing's add button to icon-only (Sage Abdullah)
- Add sublabel to breadcrumbs, including history, usage, and inspect views (Sage Abdullah)
- Standardise search form placeholder to 'Search...' (Sage Abdullah)
- Use `SlugInput` on all `SlugFields` by default (LB (Ben) Johnston)
- Show character counts on `RichTextBlock` with `max_length` (Elhussein Almasri)
- Move locale selector in generic `IndexView` to a filter (Sage Abdullah)
- Add ability to `customize a page's copy form` including an auto-incrementing slug example (Neeraj Yetheendran)

- Add `WAGTAILADMIN_LOGIN_URL` setting to allow customizing the login URL (Neeraj Yetheendran)
- Polish dark theme styles and update color tokens (Thibaud Colas, Rohit Sharma)
- Keep database state of pages and snippets updated while in draft state (Stefan Hammer)
- Add `DrilldownController` and `w-drilldown` component to support drilldown menus (Thibaud Colas)
- Add API support for a `redirects (contrib)` endpoint (Rohit Sharma, Jaap Roes, Andreas Donig)
- Add the default ability for all `SnippetViewSet` & `ModelViewSet` to support `being copied`, this can be disabled by `copy_view_enabled = False` (Shlomo Markowitz)
- Support dynamic Wagtail guide links in the admin that are based on the running version of Wagtail (Tidiane Dia)

Bug fixes

- Update system check for overwriting storage backends to recognize the `STORAGES` setting introduced in Django 4.2 (phijma-leukeleu)
- Prevent password change form from raising a validation error when browser autocomplete fills in the “Old password” field (Chiemezuo Akujobi)
- Ensure that the legacy dropdown options, when closed, do not get accidentally clicked by other interactions on wide viewports (CheesyPhoenix, Christer Jensen)
- Add a fallback background for the editing preview iframe for sites without a background (Ian Price)
- Preserve whitespace in rendered comments (Elhussein Almasri)
- Remove search logging from project template so that new projects without the search promotions module will not error (Matt Westcott)
- Ensure text only email notifications for updated comments do not escape HTML characters (Rohit Sharma)
- Use the latest draft when copying an unpublished page for translation (Andrey Nehaychik)
- Make Workflow and Aging Pages reports only available to users with page-related permissions (Rohit Sharma)
- Make searching on specific fields work correctly on Elasticsearch when boost is in use (Matt Westcott)
- Use a visible border and background color to highlight active formatting in the rich text toolbar (Cassidy Pittman)
- Ensure image focal point box can be removed (Gunnar Scherf)
- Ensure that Snippets search results correctly use the `index_results.html` or `index_results_template_name` override on initial load (Stefan Hammer)
- Avoid error when attempting to moderate a page drafted by a now deleted user (Dan Braghis)
- Do not show multiple error messages when editing a Site to use existing hostname and port (Rohit Sharma)
- Avoid error when exporting Aging Pages report where a page has an empty `last_published_by_user` (Chiemezuo Akujobi)
- Ensure Page querysets support using `alias` and `specific` (Tomasz Knapik)
- Ensure workflow dashboard panels work when the page/snippet is missing (Sage Abdullah)
- Ensure `ActionController` explicitly checks for elements that allow select functionality (Nandini Arora)
- Prevent a `ValueError` with `FormSubmissionsPanel` on Django 5.0 when creating a new form page (Matt Westcott)
- Avoid duplicate entries in “Recent edits” panel when copying pages (Matt Westcott)

- Prevent TitleFieldPanel from raising an error when the slug field is missing or read-only (Rohit Sharma)
- Ensure that the close button on the new dialog designs is visible in the non-message variant (Nandini Arora)
- Avoid text overflow issues in comment replies and scroll position issues for long comments (Rohit Sharma)
- Remove ‘Page’ from page types filter on aging pages report (Matt Westcott)
- Prevent page types filter from showing other non-Page models that match by name (Matt Westcott)
- Ensure MultipleChooserPanel modal works correctly when USE_THOUSAND_SEPARATOR is True for pages with ids over 1,000 (Sankalp, Rohit Sharma)
- Ensure the panel anchor button sizes meet accessibility guidelines for minimum dimensions (Nandini Arora)
- Raise a 404 for bulk actions for models which don’t exist instead of throwing a 500 error (Alex Tomkins)
- Raise a SiteSetting.DoesNotExist error when retrieving settings for an unrecognized site (Nick Smith)
- Ensure that defaulted or unique values declared in exclude_fields_in_copy are correctly excluded in new copies, resolving to the default value (Elhussein Almasri)
- Ensure that default_ordering set on IndexView is preserved if ModelViewSet does not specify an explicit ordering (Cynthia Kiser)
- Resolve issue where clicking Publish for a Page that was in workflow in Safari would block publishing and not trigger the workflow confirmation modal (Alex Morega)
- Fix pagination links on model history and usage views (Matt Westcott)
- Fix crash when accessing workflow reports with a deleted snippet (Sage Abdullah)

Documentation

- Document, for contributors, the use of translate string literals passed as arguments to tags and filters using __() within templates (Chiemezuo Akujobi)
- Document all features for the Documents app in one location, see [Documents](#) (Neeraj Yetheendran)
- Add section to [testing docs](#) about creating pages and working with page content (Mariana Bedran Lesche)
- Add more nuance to the database recommendations in [Performance](#) (Jadesola Kareem)
- Add clarity that [MultipleChooserPanel](#) may require a chooser viewset and how the functionality is expected to work (Andy Chosak)
- Clarify where documentation build commands should be run (Nikhil S Kalburgi)
- Add missing import to tutorial BlogPage example (Salvo Polizzi)
- Update contributing guide documentation and GitHub templates to better support new contributors (Thibaud Colas)
- Add more CSS authoring guidelines (Thibaud Colas)
- Update MyST documentation parser library to 2.0.0 (Neeraj Yetheendran)
- Add documentation writing guidelines for intersphinx / external links (LB (Ben) Johnston)
- Add Page model reference get_children documentation (Salvo Polizzi)
- Enforce CI build checks for documentation so that malformed links or missing images will not be allowed (Neeraj Yetheendran)
- Update spelling on customizing admin template and page model section from British to American English (Victoria Poromon)

- Add documentation for how to override the file locations for custom image models *Overriding the upload location* (Osaf AliSayed, Dharmik Gangani)
- Update documentation theme (Sphinx Wagtail Theme) to 6.2.0, fixing the incorrect favicon (LB (Ben) Johnston, Sahil Jangra)
- Refactor promotion banner without jQuery and use sameSite cookies when storing if cleared (LB (Ben) Johnston)
- Use cross-reference for compatible Python versions in tutorial instead of the out of date listing (mirusu400)

Maintenance

Generic class-based views adoption

As part of ongoing refactorings, we have migrated several views to use generic class-based views. This allows for easier extensibility and better code reuse.

- Migrate the contrib styleguide index view to a class-based view (Chiemezuo Akujobi)
- Migrate the contrib settings edit view to a class-based view (Chiemezuo Akujobi, Sage Abdullah)
- Migrate account editing view to a class-based view (Kehinde Bobade)
- Refactor page explorer index template to extend generic index template (Sage Abdullah)
- Refactor snippets index view and template to make better use of generic IndexView (Sage Abdullah)
- Refactor documents listing view to use generic IndexView (Sage Abdullah)
- Refactor images listing view to use generic IndexView (Sage Abdullah)
- Refactor form pages listing view to use generic IndexView (Sage Abdullah)
- Reduce gap between snippets and generic views/templates (Sage Abdullah)

Other maintenance

- Update BeautifulSoup upper bound to 4.12.x (scott-8)
- Migrate initialization of classes (such as `body.ready`) from multiple JavaScript implementations to one Stimulus controller `w-init` (Chiemezuo Akujobi)
- Adopt the usage of translate string literals using `arg=_('...')` in all `wagtailadmin` module templates (Chiemezuo Akujobi)
- Update djhtml to 3.0.6 (Matt Westcott)
- Remove django-pattern-library upper bound in testing dependencies (Sage Abdullah)
- Split up functions in Elasticsearch backend for easier extensibility (Marcel Kornblum, Cameron Lamb, Sam Dudley)
- Relax `draftjs_exporter` dependency to allow using version 5.x (Sylvain Fankhauser)
- Refine styling of listings, account settings panels and the block chooser (Meli Imelda)
- Remove icon font support (Matt Westcott)
- Remove deprecated SVG icons (Matt Westcott)
- Remove icon font styles (Thibaud Colas)
- Upgrade frontend tooling to use Node 20 (LB (Ben) Johnston)

- Upgrade `ruff` and replace `black` with `ruff` format (John-Scott Atlakson)
- Update Willow upper bound to 2.x (Dan Braghis)
- Removed support for Django < 4.2 (Dan Braghis)
- Replace template components implementation with standalone `laces` library (Tibor Leupold)
- Introduce an internal `{ % formatedfield %}` tag to replace direct use of `wagtailadmin/shared/field.html` (Matt Westcott)
- Update Telepath dependency to 0.3.1 (Matt Westcott)
- Allow `ActionController` to have a `noop` method to more easily leverage standalone Stimulus action options (Nandini Arora)
- Upgrade to latest TypeScript and Storybook (Thibaud Colas, Sage Abdullah)
- Turn on `skipLibCheck` for TypeScript (LB (Ben) Johnston)
- Support for the Stimulus `CloneController` to auto clear the added content after a set duration (LB (Ben) Johnston)
- Update Stylelint, our linting configuration, Sass, and related code changes (LB (Ben) Johnston)
- Simplify browserslist and browser support documentation for maintainers (Thibaud Colas)
- Relax django-taggit dependency to allow 5.0 (Sylvain Fankhauser)
- Fix various warnings when building docs (Cynthia Kiser)
- Upgrade sphinxcontrib-spelling to 7.x for Python 3.12 compatibility (Matt Westcott)
- Move logic for `django-filters` filtering into `BaseListView` (Matt Westcott)
- Remove or replace legacy CSS classes: `visuallyhidden`, `visuallyvisible`, `divider-after`, `divider-before`, `inline`, `inline-block`, `block`, `u-hidden`, `clearfix`, `reordering`, `overflow` (Thibaud Colas)
- Prevent future issues with `icon.html` end-of-file newlines (Thibaud Colas)
- Rewrite styles using legacy `c-`, `o-`, `u-`, `t-`, `is-` prefixes (Thibaud Colas)
- Remove invalid CSS styles / Sass selector concatenation (Thibaud Colas)
- Refactor listing views to share more queryset ordering logic (Matt Westcott)
- Remove `initTooltips` in favor of Stimulus controller (LB (Ben) Johnston)
- Enhance the Stimulus `InitController` to allow for custom event dispatching when ready (Aditya, LB (Ben) Johnston)
- Remove inline script usage for comment initialization and adopt an event listener/dispatch approach for better CSP compliance (Aditya, LB (Ben) Johnston)
- Migrate styleguide ad-hoc JavaScript to use styles only to avoid CSP issues (LB (Ben) Johnston)
- Update Jest version - frontend tooling (Nandini Arora)
- Remove non-functional and inaccessible auto-focus on first field in page create forms (LB (Ben) Johnston)
- Migrate the unsaved form checks & confirmation trigger to Stimulus `UnsavedController` (Sai Srikan Dum-peti, LB (Ben) Johnston)
- Migrate page listing menu re-ordering (drag & drop) from jQuery inline scripts to `OrderableController` with a more accessible solution (Aman Pandey, LB (Ben) Johnston)
- Clean up scss variable usage, remove unused variables and mixins, adopt more core token variables (Jai Vignesh J, Nandini Arora, LB (Ben) Johnston)

- Migrate Image URL generator views to class-based views (Rohit Sharma)
- Use Django's `FileResponse` when serving files such as Images or Documents (Jake Howard)
- Deprecated `WidgetWithScript` base widget class (LB (Ben) Johnston)
- Remove support for Django 4.1 and below (Sage Abdullah)

Upgrade considerations - removal of deprecated features from Wagtail 4.2 - 5.1

Features previously deprecated in Wagtail 4.2, 5.0 and 5.1 have been fully removed. For additional details on these changes, see:

- [Wagtail 4.2 release notes](#)
- [Wagtail 5.0 release notes](#)
- [Wagtail 5.1 release notes](#)

The most significant changes are highlighted below.

Removal of ModelAdmin app

The `wagtail.contrib.modeladmin` app has been removed. If you wish to continue using it, it is available as the external package `wagtail-modeladmin`.

Query model moved to `wagtail.contrib.search_promotions`

The `Query` model (used to log search queries performed by users, to identify commonly searched terms) is no longer part of the `wagtail.search` module; it can now be found in the optional `wagtail.contrib.search_promotions` app. When updating code to import the model from the new location, ensure that you have added `wagtail.contrib.search_promotions` to your `INSTALLED_APPS` setting - failing to do this may result in a spurious migration being created within the core `wagtail` app.

Support for Elasticsearch 5 and 6 dropped

The Elasticsearch 5 and 6 backends have been removed. If you are using one of these backends, you will need to upgrade to Elasticsearch 7 or 8 before upgrading to Wagtail 6.0.

StreamField no longer requires `use_json_field=True`

The `use_json_field` argument to `StreamField` is no longer required, and can be removed. `StreamField` now consistently uses `JSONField` for its database representation, and Wagtail 5.0 required older `TextField`-based streams to be migrated. As such, `use_json_field` no longer has any effect.

Other removals

- The `WAGTAILADMIN_GLOBAL_PAGE_EDIT_LOCK` setting is no longer recognized and should be replaced with `WAGTAILADMIN_GLOBAL_EDIT_LOCK`.
- The `wagtail.models.UserPagePermissionsProxy` class and `get_pages_with_direct_explore_permission`, `get_explorable_root_page` and `users_with_page_permission` functions have been removed; equivalent functionality exists in the `wagtail.permission_policies.pages.PagePermissionPolicy` class.
- The `permission_type` field of the `GroupPagePermission` model has been removed; the `permission` field (a foreign key to Django's `Permission` model) should be used instead.
- The legacy moderation system used before the introduction of workflows in Wagtail 2.10 has been removed. Any moderation requests still in the queue from before this time will be lost.
- The Wagtail icon font has been removed; any direct usage of this needs to be converted to SVG icons.
- Various unused icons deprecated in Wagtail 5.0 have been removed.
- The `partial_match` argument on `SearchField` and on `search` methods has been removed. `AutocompleteField` and the `autocomplete` method should be used instead.
- The `insert_editor_css` hook has been removed; the `insert_global_admin_css` hook should be used instead.
- The `wagtail.contrib.frontend_cache` module now supports `azure-mgmt-cdn` version 10 and `azure-mgmt-frontdoor` version 1 as its minimum supported versions.
- The `Task.page_locked_for_user` method has been removed; `Task.locked_for_user` should be used instead.
- The `{% icon %}` template tag no longer accepts `class_name` as an argument; `classname` should be used instead.
- The `wagtail.tests.utils` module has been removed and can now be found at `wagtail.test.utils`.
- The template `wagtailadmin/shared/field_as_li.html` has been removed, and should be replaced with `wagtailadmin/shared/field.html` enclosed in an `` tag.
- The custom client-side events `wagtail:show` and `wagtail:hide` on showing and hiding dialogs have been removed; `w-dialog:show` and `w-dialog:hide` should be used instead.
- The global Javascript definitions `headerSearch`, `initTagField`, `cancelSpinner` and `unicodeSlugsEnabled` have been removed; these should be replaced with Stimulus controllers.

Upgrade considerations - changes affecting all projects

Changes to UUID fields on MariaDB when upgrading to Django 5.0

Django 5.0 introduces support for MariaDB's native UUID type on MariaDB 10.7 and above. This breaks backwards compatibility with CHAR-based UUIDs created on earlier versions of Django and MariaDB, and so upgrading a site to Django 5.0+ and MariaDB 10.7+ is liable to result in errors such as `Data too long for column 'translation_key' at row 1` or `Data too long for column 'uuid' at row 1` when creating or editing pages. To fix this, it is necessary to run the `convert_mariadb_uuids` management command (available as of Wagtail 6.0.3) after upgrading:

```
./manage.py convert_mariadb_uuids
```

This will convert all existing UUID fields used by Wagtail to the new format. New sites created under Django 5.0+ and MariaDB 10.7+ are unaffected.

SnippetViewSet & ModelViewSet copy view enabled by default

The newly introduced copy view will be enabled by default for all ModelViewSet and SnippetViewSet classes.

This can be disabled by setting `copy_view_enabled = False`, for example.

```
class PersonViewSet(SnippetViewSet):
    model = Person
    ...
    copy_view_enabled = False

class PersonViewSet(ModelViewSet):
    model = Person
    ...
    copy_view_enabled = False
```

See [Copy view](#) for additional details about this feature.

Upgrade considerations - deprecation of old functionality

Removed support for Django < 4.2

Django versions before 4.2 are no longer supported as of this release; please upgrade to Django 4.2 or above before upgrading Wagtail.

Upgrade considerations - changes affecting Wagtail customizations

SlugInput widget is now the default for SlugField fields

In Wagtail 5.0 a new `SlugInput` admin widget was added to support slug behavior in Page and Page copy forms. This widget was included by default if the `promote_panels` fields layout was customized, causing confusion.

As of this release, any forms that inherit from `WagtailAdminModelForm` (includes page and snippet model editing) will now use the `SlugInput` by default on all models with `SlugField` fields.

Previously, the widget had to be explicitly added.

```
from wagtail.admin.widgets.slug import SlugInput
# ... other imports

class MyPage(Page):
    promote_panels = [
        FieldPanel("slug", widget=SlugInput),
        # ... other panels
    ]
```

Keeping the widget as above is fine, but will no longer be required. The JavaScript field behavior will be included by default.

```
# ... imports

class MyPage(Page):
    promote_panels = [
        FieldPanel("slug"),
        # ... other panels
    ]
```

If you do not want this for some reason, you will now need to declare a different widget.

```
from django.forms.widgets import TextInput
# ... other imports

class MyPage(Page):
    promote_panels = [
        FieldPanel("slug", widget=TextInput), # use a plain text field
        # ... other panels
    ]
```

Changed handling of database updates of non-live Page objects or subclasses of DraftStateMixin

Before this release, the database record of a Page or any subclass of `DraftStateMixin` either contained the live data (if published), the state of the last published version (if unpublished), or the state of the first revision (if never published). Subsequent draft edits would create new `Revision` records, but the main database record would not be updated. As a result, the database record could lag substantially behind the current state of the object, causing unexpected behavior particularly when unique constraints are in use.

As of this release, the database record of a non-live object will be updated to reflect the draft state of the object. This is unlikely to have a visible effect on existing sites, since the admin backend works with the `Revision` records while the site front-end typically filters out non-live objects. However, any code that relies on the database record being untouched by draft edits (for example, using it to store a specific approved / archived state of the page) may need to be updated.

Upgrade considerations - changes to undocumented internals

`filter_queryset` and `get_filtered_queryset` methods no longer return filters

The undocumented internal methods `filter_queryset(queryset)` on `wagtail.admin.views.generic.IndexView`, and `get_filtered_queryset()` on `wagtail.admin.views.reports.ReportView`, now return just the filtered queryset; previously they returned a tuple of (`filters`, `queryset`). The `FilterSet` instance is always available as the cached property `self.filters`.

Deprecation of the undocumented `window.enableDirtyFormCheck` function

The admin frontend `window.enableDirtyFormCheck` will be removed in a future release and as of this release only supports the basic initialization.

The previous approach was to call a `window` global function as follows.

```
window.enableDirtyFormCheck('.my-form', { alwaysDirty: true, confirmationMessage:
  ↵'You have unsaved changes'});
```

The new approach will be data attribute driven as follows.

```
<form
    method="POST"
    data-controller="w-unsaved"
    data-action="w-unsaved#submit beforeunload@window->w-unsaved#confirm change->w-
    unsaved#check keyup->w-unsaved#check"
    data-w-unsaved-confirm-value="This page has unsaved changes." // equivalent to_
    `confirmationMessage` .
    data-w-unsaved-force-value="true" // equivalent to `alwaysDirty` .
    data-w-unsaved-watch-value="edits comments" // can add 'comments' if comments is_
    enabled, defaults to only 'edits'.
  >
  ... form contents
</form>
```

data-tippy-content attribute support will be removed

The implementation of the JS tooltips have been fully migrated to the Stimulus w-tooltip/TooltipController implementation.

Dynamic support for any `data-tippy-content="..."` usage will be removed this release, for example, within chooser modals or dynamic html response data.

Some minimal backwards compatibility support for `data-tippy-content` will work until a future release, but only in the initial HTML response on a page.

Data attributes

These HTML data attributes were not documented, but if any custom code implemented custom tooltips, these will need to be changed.

Old	New	Notes
	<code>data-controller="w-tooltip"</code>	Required, new addition for any usage
<code>data-tippy-content="<% trans 'History' %>"</code>	<code>data-w-tooltip-content-value="<% trans 'History' %>"</code>	Required
<code>data-tippy-offset="[12, 24]"</code>	<code>data-w-tooltip-offset-value="[12, 24]"</code>	Optional, default is no offset
<code>data-tippy-placement="top"</code>	<code>data-w-tooltip-placement-value="t"</code>	Optional, default is 'bottom'

Deprecated `WidgetWithScript` base widget class

The undocumented `WidgetWithScript` class that used inline scripts to attach JavaScript to widgets will be removed in a future release.

This approach creates security risks and will not be compliant with CSP support. Instead, it's recommended that all similar requirements migrate to use the recommended Stimulus JS integration approach.

A full example of how to build this has been documented on [extending client-side behavior](#), a basic example is below.

Old

```
from django.forms import Media, widgets

class CustomRichTextArea(WidgetWithScript, widgets.Textarea):
    def render_js_init(self, id_, name, value):
        return f"window.customEditorInitScript({json.dumps(id_)});"

    @property
    def media(self):
        return Media(js=["vendor/custom-editor.js"])
```

New

```
from django.forms import Media, widgets

class CustomRichTextArea(widgets.Textarea):
    def build_attrs(self, *args, **kwargs):
        attrs = super().build_attrs(*args, **kwargs)
        attrs['data-controller'] = 'custom-editor'

    @property
    def media(self):
        return Media(js=["vendor/custom-editor.js", "js/custom-editor-controller.js"])
```

```
// myapp/static/js/custom-editor-controller.js

class CustomEditorController extends window.StimulusModule.Controller {
    connect() {
        window.customEditorInitScript(this.element.id);
    }
}

window.wagtail.app.register('custom-editor', CustomEditorController);
```

1.11.24 Wagtail 5.2.8 release notes

February 3, 2025

- *What's new*

What's new

Bug fixes

- Prevent database error when calling permission_order.register on app ready (Daniel Kirkham, Matt Westcott)
- Handle StreamField migrations where the field value is null (Joshua Munn)
- Prevent StreamChildrenToListBlockOperation from duplicating data across multiple StreamField instances (Joshua Munn)
- Prevent error on lazily loading StreamField blocks after the stream has been modified (Stefan Hammer)
- Prevent syntax error on MySQL search when query includes symbols (Matt Westcott)

1.11.25 Wagtail 5.2.7 release notes

November 1, 2024

- *What's new*

What's new

Bug fixes

- Prevent multiple URLs from being combined into one when pasting links into a rich text input (Thibaud Colas)
- Fix error on workflow settings view with multiple snippet types assigned to the same workflow on Postgres (Sage Abdullah)

1.11.26 Wagtail 5.2.6 release notes

July 11, 2024

- *What's new*

What's new

CVE-2024-39317: Regular expression denial-of-service via search query parsing

This release addresses a denial-of-service vulnerability in Wagtail. A bug in Wagtail's `parse_query_string` would result in it taking a long time to process suitably crafted inputs. When used to parse sufficiently long strings of characters without a space, `parse_query_string` would take an unexpectedly large amount of time to process, resulting in a denial of service.

In an initial Wagtail installation, the vulnerability can be exploited by any Wagtail admin user. It cannot be exploited by end users. If your Wagtail site has a custom search implementation which uses `parse_query_string`, it may be exploitable by other users (e.g. unauthenticated users).

Many thanks to Jake Howard for reporting and fixing this issue. For further details, please see [the CVE-2024-39317 security advisory](#).

Bug fixes

- Fix image preview when Willow optimizers are enabled (Alex Tomkins)

Maintenance

- Remove django-pattern-library upper bound in testing dependencies (Sage Abdullah)

1.11.27 Wagtail 5.2.5 release notes

May 1, 2024

- [What's new](#)
- [Upgrade considerations](#)

What's new

Bug fixes

- Respect `WAGTAIL_ALLOW_UNICODE_SLUGS` setting when auto-generating slugs (LB (Ben) Johnston)
- Use correct URL when redirecting back to page search results after an AJAX search (Sage Abdullah)
- Provide `convert_mariadb_uuids` management command to assist with upgrading to Django 5.0+ on MariaDB (Matt Westcott)

Upgrade considerations

Changes to UUID fields on MariaDB when upgrading to Django 5.0

Django 5.0 introduces support for MariaDB's native UUID type on MariaDB 10.7 and above. This breaks backwards compatibility with CHAR-based UUIDs created on earlier versions of Django and MariaDB, and so upgrading a site to Django 5.0+ and MariaDB 10.7+ is liable to result in errors such as Data too long for column 'translation_key' at row 1 or Data too long for column 'uuid' at row 1 when creating or editing pages. To fix this, it is necessary to run the `convert_mariadb_uuids` management command (available as of Wagtail 5.2.5) after upgrading:

```
./manage.py convert_mariadb_uuids
```

This will convert all existing UUID fields used by Wagtail to the new format. New sites created under Django 5.0+ and MariaDB 10.7+ are unaffected.

1.11.28 Wagtail 5.2.4 release notes

April 3, 2024

- *What's new*

What's new

Bug fixes

- Prevent TitleFieldPanel from raising an error when the slug field is missing or read-only (Rohit Sharma)
- Fix pagination links on model history and usage views (Matt Westcott)
- Fix crash when accessing workflow reports with a deleted snippet (Sage Abdullah)
- Prevent error on submitting an empty search in the admin under Elasticsearch (Maikel Martens)

1.11.29 Wagtail 5.2.3 release notes

January 23, 2024

- *What's new*

What's new

Bug fixes

- Prevent a ValueError with `FormSubmissionsPanel` on Django 5.0 when creating a new form page (Matt Westcott)
- Specify `telepath` 0.3.1 as the minimum supported version, for Django 5.0 compatibility (Matt Westcott)

1.11.30 Wagtail 5.2.2 release notes

December 6, 2023

- *What's new*

What's new

Django 5.0 support

This release adds support for Django 5.0.

Bug fixes

- Use a visible border and background color to highlight active formatting in the rich text toolbar (Cassidy Pittman)
- Ensure image focal point box can be removed (Gunnar Scherf)
- Ensure that Snippets search results correctly use the `index_results.html` or `index_results_template_name` override on initial load (Stefan Hammer)
- Avoid error when attempting to moderate a page drafted by a now deleted user (Dan Braghis)
- Ensure workflow dashboard panels work when the page/snippet is missing (Sage Abdullah)
- Prevent custom controls from stacking on top of the comment button in Draftail toolbar (Ben Morse)

1.11.31 Wagtail 5.2.1 release notes

November 16, 2023

- *What's new*

What's new

Bug fixes

- Add a fallback background for the editing preview iframe for sites without a background (Ian Price)
- Remove search logging from project template so that new projects without the search promotions module will not error (Matt Westcott)
- Ensure text only email notifications for updated comments do not escape HTML characters (Rohit Sharma)
- Use logical OR operator to combine search fields for Django ORM in generic IndexView (Varun Kumar)
- Ensure that explorer_results views fill in the correct next_url parameter on action URLs (Matt Westcott)
- Fix crash when accessing the history view for a translatable snippet (Sage Abdullah)
- Prevent upload of SVG images from failing when image feature detection is enabled (Joshua Munn)
- Fix crash when using the locale switcher on the snippets create view (Sage Abdullah)
- Fix performance regression on reports from calling `decorate_paginated_queryset` before pagination / filtering (Alex Tomkins)
- Make searching on specific fields work correctly on Elasticsearch when boost is in use (Matt Westcott)
- Prevent snippet permission post-migrate hook from failing on multiple database configurations (Joe Tsoi)
- Reinstate ability to filter on page type when searching on an empty query (Sage Abdullah)
- Prevent error on locked pages report when a user has locked multiple pages (Matt Westcott)

Documentation

- Fix code example for `{% picture ... as ... %}` template tag (Rezyapkin)

1.11.32 Wagtail 5.2 (LTS) release notes

November 1, 2023

- *What's new*
- *Upgrade considerations - changes affecting all projects*
- *Upgrade considerations - deprecation of old functionality*
- *Upgrade considerations - changes affecting Wagtail customizations*
- *Upgrade considerations - changes to undocumented internals*

Wagtail 5.2 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

Redesigned page listing view

The screenshot shows the Wagtail page listing view for the 'Wagtail Bakery' site. At the top, there is a search bar with the query 'bread'. Below the search bar, a table lists four pages:

Title	Updated	Type	Status
The Greatest Thing Since Sliced Bread > Blog	1 month ago	Blog page	LIVE
Bread and Circuses > Blog	1 month ago	Blog page	LIVE + DRAFT
Black bread > Breads	1 month ago	Bread page	LIVE
Breads	1 month ago	Breads index page	LIVE

At the bottom of the list, there is a navigation arrow pointing right.

The page explorer listing view has been redesigned to allow improved navigation and searching. This feature was developed by Ben Enright, Matt Westcott, Thibaud Colas and Sage Abdullah.

OpenSearch support

OpenSearch is now formally supported as an alternative to Elasticsearch. For configuration details, see [OpenSearch configuration](#). This feature was developed by Matt Westcott.

Responsive & multi-format images with the picture tag

Wagtail has new template tags to reduce the loading time and environmental footprint of images:

- The `picture` tag generates images in multiple formats and-or sizes in one batch, creating an HTML `<picture>` tag.
- The `srcset_image` tag generates images in multiple sizes, creating an `` tag with a `srcset` attribute.

As an example, the `picture` tag allows generating six variants of an image in one go:

```
{% picture page.photo format-{avif,webp,jpeg} width-{400,800} sizes="80vw" %}
```

This outputs:

```
<picture>
  <source sizes="80vw" srcset="/media/images/pied-wagtail.width-400.avif 400w, />
  <source sizes="80vw" srcset="/media/images/pied-wagtail.width-800.avif 800w" type="image/avif">
  <source sizes="80vw" srcset="/media/images/pied-wagtail.width-400.webp 400w, />
  <source sizes="80vw" srcset="/media/images/pied-wagtail.width-800.webp 800w" type="image/webp">
  <img sizes="80vw" srcset="/media/images/pied-wagtail.width-400.jpg 400w, /media/>
  
</picture>
```

We expect those changes to greatly reduce the weight of images for all Wagtail sites. We encourage all site implementers to consider using them to improve the performance of the sites and reduce their carbon footprint. For further details, For more details, see [Multiple formats](#) and [Responsive images](#). Those new template tags are also supported in Jinja templates, see [Jinja2 template support](#) for the Jinja API.

This feature was developed by Paarth Agarwal and Thibaud Colas as part of the Google Summer of Code program and a [partnership with the Green Web Foundation](#) and Green Coding Berlin, with support from Dan Braghis, Thibaud Colas, Sage Abdullah, Arne Tarara (Green Coding Berlin), and Chris Adams (Green Web Foundation). We also thank Aman Pandey for introducing [AVIF support](#) in Wagtail 5.1, Andy Babic for creating [AbstractImage.get_renditions\(\)](#) in the same release; and Storm Heg, Mitchel Cabuloy, Coen van der Kamp, Tom Dyson, and Chris Lawton for their feedback on [RFC 71](#).

Support extending Wagtail client-side with Stimulus

Wagtail now officially supports client-side admin customizations with [Stimulus](#). The developer documentation has a dedicated page about [Extending client-side behavior](#). This covers fundamental topics of client-side extensibility, such as:

- Adding custom JavaScript
- Extending with DOM events and Wagtail's custom DOM events
- Extending with Stimulus
- Extending with React

Thank you to core contributor LB (Ben) Johnston for writing this documentation.

ModelViewSet improvements

Several features from [SnippetViewSet](#) have been implemented in [ModelViewSet](#), allowing you to use them without registering your models as snippets. For more details on using ModelViewSet, refer to [Generic views](#).

- Move SnippetViewSet menu registration mechanism to base ViewSet class (Sage Abdullah)
- Move SnippetViewSet template override mechanism to ModelViewSet (Sage Abdullah)
- Move SnippetViewSet.list_display to ModelViewSet (Sage Abdullah)
- Move list_filter, filterset_class, search_fields, search_backend_name, list_export, export_filename, list_per_page, and ordering from SnippetViewSet to ModelViewSet (Sage Abdullah, Cynthia Kiser)
- Add default header titles to generic IndexView and CreateView (Sage Abdullah)
- Add the ability to use filters and to export listings in generic IndexView (Sage Abdullah)
- Add generic UsageView to ModelViewSet (Sage Abdullah)
- Add generic InspectView to ModelViewSet (Sage Abdullah)
- Extract generic HistoryView from snippets and add it to ModelViewSet (Sage Abdullah)
- Extract generic breadcrumbs functionality from page breadcrumbs (Sage Abdullah)
- Add breadcrumbs support to custom ModelViewSet views (Sage Abdullah)
- Allow ModelViewSet to be used with models that have non-integer primary keys (Sage Abdullah)

- Enable reference index tracking for models registered with `ModelViewSet` (Sage Abdullah)

In addition, the following new features have been added to the generic admin views as part of `ModelViewSet`, which can also be used with `SnippetViewSet`.

- Allow overriding `IndexView.export_headings` via `ModelViewSet` (Christer Jensen, Sage Abdullah)
- Add the ability to define listing buttons on generic `IndexView` (Sage Abdullah)

User interface refinements

Several tweaks have been made to the admin user interface which we hope will make it easier to use.

- Show the full first published at date within a tooltip on the Page status sidebar on the relative date (Rohit Sharma)
- Do not render minimap if there are no panel anchors (Sage Abdullah)
- Use dropdown buttons on listings in dashboard panels (Sage Abdullah)
- Implement breadcrumbs design refinements (Thibaud Colas)
- Add support for Shift + Click behavior in form submissions and simple translations submissions (LB (Ben) Johnston)
- Improve filtering of audit logging based on the user's permissions (Stefan Hammer)

External links in promoted search results

Promoted search result entries can now use an external URL along with custom link text, instead of linking to a page within Wagtail. This makes it easier to manage promoted content across multiple websites. Thank you to TopDevPros, and Brad Busenius from University of Chicago Library.

Other features

- Add support for Python 3.12 (Matt Westcott)
- Add `wagtailcache` and `wagtailpagecache` template tags to ensure previewing Pages or Snippets will not be cached (Jake Howard)
- Always set help text element ID for form fields with help text in `field.html` template (Sage Abdullah)
- When copying a page or creating an alias, copy its view restrictions to the destination (Sandeep Choudhary, Suyash Singh)
- Support pickling of StreamField values (pySilver)
- Remove `wagtail.publish` log action on aliases when they are created from live source pages or the source page is published (Dan Braghis)
- Remove `wagtail.unpublish` log action on aliases when source page is unpublished (Dan Braghis)
- Add compare buttons to workflow dashboard panel (Matt Westcott)
- Support specifying a `get_object_list` method on `ChooserViewSet` (Matt Westcott)
- Add *linked_fields mechanism on chooser widgets* to allow choices to be limited by fields on the calling page (Matt Westcott)
- Add support for merging cells within `TableBlock` with the `mergedCells option` (Gareth Palmer)
- When adding a panel within `InlinePanel`, focus will now shift to that content similar to `StreamField` (Faishal Manzar)

- Add support for placement in `human_readable_date` the tooltip template tag (Rohit Sharma)
- Support passing extra context variables via the `{% component %}` tag (Matt Westcott)
- Allow subclasses of `PagesAPIViewSet` override default `Page model via the mode` attribute (Neeraj Yetheendran, Herbert Poul)
- Add support for subject and body in the Email link chooser form (TopDevPros, Alexandre Joly)
- Add a visual progress bar to the output of the `wagtail_update_image_renditions` management command (Faishal Manzar)
- Increase the read buffer size to improve efficiency and performance when generating file hashes for document or image uploads, use `hashlib.file_digest` if available (Python 3.11+) (Jake Howard)
- API ordering now *supports multiple fields* (Rohit Sharma, Jake Howard)
- Pass block value to `Block.get_template` to allow varying template based on value (Florian Delizy)
- Add `InlinePanel DOM events` for when ready and when items added or removed (Faishal Manzar)
- Support `Filter` instances as input for `AbstractImage.get_renditions()` (Thibaud Colas)
- Improve error messages for image template tags (Thibaud Colas)
- The `purge_revisions management command` now respects revisions that have an `on_delete=PROTECT` foreign key relation and won't delete them (Neeraj P Yetheendran, Meghana Reddy, Sage Abdullah, Storm Heg)

Bug fixes

- Ensure that StreamField's FieldBlocks correctly set the `required` and `aria-describedby` attributes (Storm Heg)
- Avoid an error when the moderation panel (admin dashboard) contains both snippets and private pages (Matt Westcott)
- When deleting collections, ensure the collection name is correctly shown in the success message (LB (Ben) Johnston)
- Filter out comments on Page editing counts that do not correspond to a valid field / block path on the page such as when a field has been removed (Matt Westcott)
- Allow `PublishMenuItem` to more easily support overriding its label via `construct_page_action_menu` (Sébastien Corbin)
- Allow locale selection when creating a page at the root level (Sage Abdullah)
- Ensure the admin login template correctly displays all `non_fields_errors` for any custom form validation (Sébastien Corbin)
- Ensure 'mark as active' label in workflow bulk action set active form can be translated (Rohit Sharma)
- Ensure the panel title for a user's settings correctly reflects the `WAGTAIL_EMAIL_MANAGEMENT_ENABLED` setting by not showing 'email' if disabled (Omkar Jadhav)
- Update Spotify oEmbed provider URL parsing to resolve correctly (Dhrűv)
- Update link colors within help blocks to meet accessible contrast requirements (Rohit Sharma)
- Ensure the search promotions popular search terms picker correctly refers to the correct model (LB (Ben) Johnston)
- Correctly quote non-numeric primary keys on snippet inspect view (Sage Abdullah)
- Prevent crash on snippet inspect view when displaying a null foreign key to an image (Sage Abdullah)

- Ensure that pages in moderation show as “Live + In Moderation” in the page explorer rather than “Live + Draft” (Sage Abdullah)
- Prevent error when updating reference index for objects with a lazy ParentalKey-related object (Chris Shaw)
- Ignore conflicts when inserting reference index entries to prevent race conditions causing uniqueness errors (Chris Shaw)
- Populate the correct return value when creating a new snippet within the snippet chooser (claudobahn)
- Reinstate missing filter by page type on page search (Matt Westcott)
- Ensure very long words can wrap when viewing saved comments (Chiemezuo Akujobi)
- Avoid forgotten password link text conflicting with the supplied aria-label (Thibaud Colas)
- Fix log message to record the correct restriction type when removing a page view restriction (Rohit Sharma, Hazh. M. Adam)
- Avoid potential race condition with new Page subscriptions on the edit view (Alex Tomkins)
- Use the correct action log when creating a redirect (Thibaud Colas)
- Ensure that all password fields consistently allow leading & trailing whitespace (Neeraj P Yetheendran)

Documentation

- Expand documentation on using `ViewSet` and `ModelViewSet` (Sage Abdullah)
- Document `WAGTAILADMIN_BASE_URL` on “[Integrating Wagtail into a Django project](#)” page (Shreshth Srivastava)
- Replace incorrect screenshot for authors listing on tutorial (Shreshth Srivastava)
- Add documentation for building *non-model-based choosers using the queryish library* (Matt Westcott)
- Fix incorrect tag library import on focal points example (Hatim Makki Hoho)
- Add reminder about including your custom Draftail feature in any overridden `WAGTAILADMIN_RICH_TEXT_EDITORS` setting (Charlie Sue)
- Mention the need to install `python3-venv` on Ubuntu (Brian Mugo)
- Document the use of the Google developer documentation style guide in documentation (Damilola Oladele)
- Fix Inconsistent URL Format in Getting Started tutorial (Olumide Micheal)
- Add more extensive documentation for the *permission kwarg support in Panels* (LB (Ben) Johnston)
- Update all `FieldPanel('title')` examples to use the recommended `TitleFieldPanel('title')` panel (Chinedu Ihedioha)

Maintenance

Stimulus adoption

As part of our adoption of Stimulus, in addition to the new documentation, we have migrated several existing components to the framework. Thank you to our core contributor LB who oversees this project, and to all contributors who refactored specific components.

- Migrate form submission listing checkbox toggling to the shared `w-bulk` Stimulus implementation (LB (Ben) Johnston)

- Migrate the editor unsaved messages popup to be driven by Stimulus using the shared `w-message` controller (LB (Ben) Johnston, Hussain Saherwala)
- Migrate all other `data-tippy` HTML attribute usage to the Stimulus `data-*-value` attributes for `w-tooltip` & `w-dropdown` (Subhajit Ghosh, LB (Ben) Johnston)
- Migrate select all on focus/click behavior to Stimulus, used on the image URL generator (Chiemezu Akujobi)
- Add support for a `reset` method to support Stimulus driven dynamic field resets via the `w-action` controller (Chiemezu Akujobi)
- Add support for a `notify` target on the Stimulus dialog for dispatching events internally (Chiemezu Akujobi)
- Migrate publishing schedule dialog field resets to Stimulus (Chiemezu Akujobi)

Other maintenance

- Fix snippet search test to work on non-fallback database backends (Matt Westcott)
- Update ESLint, Prettier & Jest npm packages (LB (Ben) Johnston)
- Add npm scripts for TypeScript checks and formatting SCSS files (LB (Ben) Johnston)
- Run tests in parallel in some of the CI setup (Sage Abdullah)
- Remove unused `WorkflowStatus` view, `urlpattern`, and `workflow-status.js` (Storm Heg)
- Add support for options/attrs in Telepath widgets so that attrs render on the created DOM (Storm Heg)
- Update pre-commit hooks to be in sync with latest changes to ESLint & Prettier for client-side changes (Storm Heg)
- Add `WagtailTestUtils.get_soup()` method for testing HTML content (Storm Heg, Sage Abdullah)
- Allow `ViewSet` subclasses to customize `url_prefix` and `url_namespace` logic (Matt Westcott)
- Simplify `SnippetViewSet` registration code (Sage Abdullah)
- Rename groups `IndexView.results_template_name` to `results.html` (Sage Abdullah)
- Allow viewsets to define a common set of view kwargs (Matt Westcott)
- Do not use jest inside `stubs.js` to prevent Storybook from crashing (LB (Ben) Johnston)
- Refactor snippets templates to reuse the shared `slim_header.html` template (Sage Abdullah)
- Refactor `slim_header.html` template to reduce code duplication (Sage Abdullah)
- Upgrade Willow to v1.6.2 to support MIME type data without reliance on `imghdr` (Jake Howard)
- Replace `imghdr` with Willow's built-in MIME type detection (Jake Howard)
- Replace `@total_ordering` usage with comparison functions implementation (Virag Jain)
- Replace `<script type="text/django-form-template"></script>` template approach with `HTML template` elements in `InlinePanel` and expanding formset (Mansi Gundre, Subhajit Ghosh, LB (Ben) Johnston)
- Refactor side panels code for better reuse in pages and snippets (Sage Abdullah)
- Deprecate legacy URL redirects in `ModelViewSet` and `SnippetViewSet` (Sage Abdullah)
- Simplify code for registering page listing action buttons (Matt Westcott)
- Removed the unused, legacy, Wagtail userbar views set up for an old iframe approach (Sage Abdullah)
- Optimize `lru_cache` usage (Jake Howard)

- Implement `date_since` in `get_most_popular` inside `search_promotions.models.Query` (TopDevPros)
- Refactor generic view subclasses to better reuse the generic templates and breadcrumbs (Sage Abdullah)
- Adopt consistent `classname` (not `classnames`) attributes for all `MenuItem` usage, including deprecation warnings (LB (Ben) Johnston)
- Adopt consistent `classname` (not `classnames`) attribute within the `wagtail.images.formats.Format` instance, including deprecation warnings (LB (Ben) Johnston)
- Deprecate `context` argument of `construct_snippet_listing_buttons` hook (Sage Abdullah)
- Deprecate legacy moderation system (Sage Abdullah)
- Update CI database versions (Jake Howard)
- Add changelog and issue tracker links to the PyPI project page (Panagiotis H.M. Issaris)
- Add better deprecation warnings to the `search.Query & search.QueryDailyHits` model, move final set of templates from the admin search module to the search promotions contrib module (LB (Ben) Johnston)

Upgrade considerations - changes affecting all projects

Changes to UUID fields on MariaDB when upgrading to Django 5.0

Django 5.0 introduces support for MariaDB's native UUID type on MariaDB 10.7 and above. This breaks backwards compatibility with CHAR-based UUIDs created on earlier versions of Django and MariaDB, and so upgrading a site to Django 5.0+ and MariaDB 10.7+ is liable to result in errors such as `Data too long for column 'translation_key' at row 1` or `Data too long for column 'uuid' at row 1` when creating or editing pages. To fix this, it is necessary to run the `convert_mariadb_uuids` management command (available as of Wagtail 5.2.5) after upgrading:

```
./manage.py convert_mariadb_uuids
```

This will convert all existing UUID fields used by Wagtail to the new format. New sites created under Django 5.0+ and MariaDB 10.7+ are unaffected.

Upgrade considerations - deprecation of old functionality

Legacy moderation system is deprecated

The legacy moderation system, which was replaced by the new workflow system in Wagtail 2.10, is now deprecated. Since Wagtail 2.10, submitting a page for moderation will use the new workflow system. However, the legacy moderation system is still in place for approving and rejecting pages that were submitted for moderation before Wagtail 2.10.

To view all pages that are still in the legacy moderation system backlog, you can sign in as a superuser and see if there is a “Pages awaiting moderation” section in the admin dashboard. You can approve or reject the pages from there. You can also do this programmatically by querying for `Revision.objects.filter(submitted_for_moderation=True)` and calling `revision.approve_moderation()` or `revision.reject_moderation()` on each revision.

The legacy moderation system will be removed in a future release. If you still have pages in the moderation queue that were submitted for moderation before Wagtail 2.10, you should approve or reject them before upgrading. See [Wagtail 2.10 release notes](#) for more details.

As a result, the following features are now deprecated:

- `wagtail.models.Revision.submitted_for_moderation`
- `wagtail.models.Revision.submitted_revisions`
- `wagtail.models.Revision.approve_moderation`
- `wagtail.models.Revision.reject_moderation`
- The `submitted_for_moderation` argument in `wagtail.models.RevisionMixin.save_revision()`
- `WAGTAIL_MODERATION_ENABLED`
- `wagtail.admin.userbar.ModeratePageItem`
- `wagtail.admin.userbar.ApproveModerationEditPageItem`
- `wagtail.admin.userbar.RejectModerationEditPageItem`
- `wagtail.admin.views.home.PagesForModerationPanel`
- `wagtail.admin.views.pages.moderation`
- `wagtail.permission_policies.pages.PagePermissionPolicy.revisions_for_moderation`

If you use any of the above features, remove them or replace them with the equivalent features from the new workflow system. The above features will be removed in a future release.

Upgrade considerations - changes affecting Wagtail customizations

Adoption of `classname` convention for `MenuItem` related classes and hooks

Wagtail `MenuItem` and menu hooks have been updated to use the more consistent naming of `classname` (singular) instead of `classnames` (plural), a convention that started in Wagtail 4.2.

The current `classnames` keyword argument naming will be supported, but will trigger a deprecation warning. Support for this variant will be removed in a future release.

The following classes will adopt this new convention.

- `admin.menu.MenuItem`
- `admin.ui.sidebar.ActionMenuItem`
- `admin.ui.sidebar.LinkMenuItem`
- `admin.ui.sidebar.PageExplorerMenuItem`
- `contrib.settings.registry.SettingMenuItem`

The following hooks usage may be impacted if `classnames` were used when generating menu items.

- `register_admin_menu_item`
- `register_settings_menu_item`

Example

```
from django.urls import reverse
from wagtail import hooks
from wagtail.admin.menu import MenuItem

@hooks.register('register_admin_menu_item')
def register_frank_menu_item():
    return MenuItem(
        'Frank',
        reverse('frank'),
        icon_name='folder-inverse',
        order=10000,
        classname="highlight-menu" # not classnames=...
)
```

Edit and delete URLs in ModelViewSet changed to allow non-integer primary keys

To accommodate models with non-integer primary keys, the URL patterns for the edit and delete views in `ModelViewSet` have been changed.

Relative to the viewset's `url_prefix`, the following changes have been made:

- The edit URL pattern has been changed from `<int:pk>/` to `edit/<str:pk>/`
- The delete URL pattern has been changed from `<int:pk>/delete/` to `delete/<str:pk>/`

If you use `reverse()` with `get_url_name()` to generate the URLs for these views, no changes are needed. However, if you have hard-coded these URLs in your code, you will need to update them to match the new patterns.

Redirects for the legacy URLs are in place for backwards compatibility, but will be removed in a future release.

The URLs for snippets underwent similar changes in Wagtail 2.14. The redirects for the legacy URLs in `SnippetViewSet` have now been marked for removal in a future release.

ModelViewSet automatically registers the model to the reference index

Models that are registered with a `ModelViewSet` now have reference index tracking enabled by default. This means that you no longer need to call `ReferenceIndex.register_model()` in your app's `ready()` method for such models. If this is undesired, you can disable it by setting `add_to_reference_index` to `False` on the `ModelViewSet` subclass. For more details, see [Manage the reference index](#).

Groups IndexView.results_template_name renamed from results.html to index_results.html

The `IndexView.results_template_name` attribute in the `GroupViewSet` has been renamed from `wagtailusers/groups/results.html` to `wagtailusers/groups/index_results.html` for consistency with the other viewsets. If you have customized or extended the template, e.g. for [Customizing group edit/create views](#), you will need to rename it to match the new name.

construct_snippet_listing_buttons hook no longer accepts a context argument

The `construct_snippet_listing_buttons` hook no longer accepts a `context` argument. If you have implemented this hook, you will need to remove the `context` argument from your implementation. If you need to access values computed by the view, you'll need to override the `index_view_class` with a custom `IndexView` subclass. The `get_list_buttons` and `get_list_more_buttons` methods in particular may be overridden to customize the buttons on the listing.

Defining a function for this hook that accepts the `context` argument will raise a warning, and the function will receive an empty dictionary (`{}`) as the `context`. Support for defining the `context` argument will be completely removed in a future Wagtail release.

Hooks for page listing and page header buttons no longer accept a page_perms argument

The arguments passed to the hooks `register_page_header_buttons`, `register_page_listing_buttons`, `construct_page_listing_buttons` and `register_page_listing_more_buttons` have changed. For all of these hooks, the `page_perms` argument has been replaced by `user`; in addition, `register_page_header_buttons` is now passed a `view_name` argument, which is either '`edit`' or '`index`', depending on whether the button is being generated for the page listing or edit view. In summary, the changes are:

- `register_page_header_buttons`: Previously `func(page, page_perms, next_url)`, now `func(page, user, next_url, view_name)`.
- `register_page_listing_buttons`: Previously `func(page, page_perms, next_url)`, now `func(page, user, next_url)`.
- `construct_page_listing_buttons`: Previously `func(buttons, page, page_perms, context)`, now `fn(buttons, page, user, context)`.
- `register_page_listing_more_buttons`: Previously `func(page, page_perms, next_url)`, now `func(page, user, next_url)`.

Additionally, the `ButtonWithDropdownFromHook` constructor, and the resulting hook it creates, should now be passed a `user` argument instead of `page_perms`.

Existing code that performs permission checks using `page_perms` can retrieve the same permission tester object using `page.permissions_for_user(user)`.

Hook functions using the old `page_perms` signature will continue to work, but this is deprecated and will raise a warning. Support for the old signature will be removed in a future Wagtail release.

Upgrade considerations - changes to undocumented internals

Breadcrumbs class name has changed

If using custom styling for the breadcrumbs, this class has changed from singular to plural for a more intuitive class.

Old	New
'w-breadcrumb'	'w-breadcrumbs'

Snippets templates refactored to reuse the shared `slim_header.html` template

The templates for the snippets views have been refactored to reuse the shared `slim_header.html` template. If you have customized or extended the templates, e.g. for [Customizing admin views for snippets](#), you will need to update them to match the new structure. As a result, the following templates have been removed:

- `wagtailsnippets/snippets/headers/_base_header.html`
- `wagtailsnippets/snippets/headers/create_header.html`
- `wagtailsnippets/snippets/headers/edit_header.html`
- `wagtailsnippets/snippets/headers/history_header.html`
- `wagtailsnippets/snippets/headers/list_header.html`
- `wagtailsnippets/snippets/headers/usage_header.html`

In most cases, the usage of those templates can be replaced with the `wagtailadmin/shared/headers/sliver_header.html` template. Refer to the snippets views and templates code for more details.

`BaseSidePanels`, `PageSidePanels` and `SnippetSidePanels` classes are removed

The `BaseSidePanels`, `PageSidePanels` and `SnippetSidePanels` classes that were used to combine the side panels (i.e. status, preview and comments side panels) have been removed. Each side panel is now instantiated directly in the view. The `wagtail.admin.ui.components.MediaContainer` class can be used to combine the `Media` objects for the side panels.

The `BasePreviewSidePanel`, `PagePreviewSidePanel` and `SnippetPreviewSidePanel` classes have been replaced with the consolidated `PreviewSidePanel` class.

The `BaseStatusSidePanel` class has been renamed to `StatusSidePanel`.

If you use these classes in your code, you will need to update your code to instantiate the side panels directly in the view.

For example, if you have the following code:

```
from wagtail.admin.ui.side_panels import PageSidePanels

def my_view(request):
    ...

    side_panels = PageSidePanels(
        request,
        page.get_latest_revision_as_object(),
        show_schedule_publishing_toggle=False,
        live_page=page,
        scheduled_page=page.get_scheduled_revision_as_object(),
        in_explorer=False,
        preview_enabled=True,
        comments_enabled=False,
    )

    return render(
        request,
        template_name,
        {"page": page, "side_panels": side_panels, "media": side_panels.media},
    )
```

Update it to the following:

```
from wagtail.admin.ui.components import MediaContainer
from wagtail.admin.ui.side_panels import PageStatusSidePanel, PreviewSidePanel


def my_view(request):
    ...

    side_panels = [
        PageStatusSidePanel(
            page,
            request,
            show_schedule_publishing_toggle=False,
            live_object=page,
            scheduled_object=page.get_scheduled_revision_as_object(),
            locale=page.locale,
            translations=translations,
        ),
        PreviewSidePanel(
            page,
            request,
            preview_url=reverse("wagtailadmin_pages:preview_on_edit", args=[page.id]),
        ),
    ]
    side_panels = MediaContainer(side_panels)

    return render(
        request,
        template_name,
        {"page": page, "side_panels": side_panels, "media": side_panels.media},
    )
```

Breadcrumbs now use different data attributes and events

The undocumented JavaScript implementation for the header breadcrumbs component has been migrated to a Stimulus controller and now uses different data attributes.

This may impact custom header implementations that relied on the previous approach, custom breadcrumbs that did not use breadcrumbs and require the expand/collapse behavior may be impacted.

Events

Old	New
'wagtail:breadcrumbs-expand'	'w-breadcrumbs:opened'
'wagtail:breadcrumbs-collapse'	'w-breadcrumbs:closed'

Data attributes

Old	New
data-breadcrumb data-controller="w-breadcrumbs" data-toggle-bre data-w-breadcrumbs-target="toggle" data-action="w-breadcrumbs#toggle mouseenter->w-breadcrumbs#peek" data-breadcrumb data-w-breadcrumbs-target="content"	

Note that the root DOM element also includes a set of additional data attributes to function as the breadcrumbs:

```
data-controller="w-breadcrumbs"
data-action="keyup.esc@document->w-breadcrumbs#close w-breadcrumbs:open@document->w-
↪breadcrumbs#open w-breadcrumbs:close@document->w-breadcrumbs#close"
data-w-breadcrumbs-close-icon-class="icon-cross"
data-w-breadcrumbs-closed-value="true"
data-w-breadcrumbs-open-icon-class="icon-breadcrumb-expand"
data-w-breadcrumbs-opened-content-class="w-max-w-4xl"
data-w-breadcrumbs-peek-target-value="header"
```

window.updateFooterSaveWarning global util removed

The undocumented global util `window.updateFooterSaveWarning` has been removed, this is part of the footer ‘unsaved’ messages toggling behavior on page forms. This behavior has now moved to a Stimulus controller and leverages DOM events instead. Calling this function will do nothing and in a future release will throw an error.

You can implement roughly the equivalent functionality with this JavaScript function, however, this will not be guaranteed to work in future releases.

```
window.updateFooterSaveWarning = (formDirty, commentsDirty) => {
  if (!formDirty && !commentsDirty) {
    document.dispatchEvent(new CustomEvent('w-unsaved:clear'));
  } else {
    const [type] = [
      formDirty && commentsDirty && 'all',
      commentsDirty && 'comments',
      formDirty && 'edits',
    ].filter(Boolean);
    document.dispatchEvent(new CustomEvent('w-unsaved:add', { detail: { type } }));
  }
};
```

dropdown template tag argument `toggle_tippy_offset` renamed to `toggle_tooltip_offset`

The naming conventions for `tippy` related attributes have been updated to align with the generic `tooltip` naming. If you are using the undocumented dropdown template tag with the `offset` arg, this will need to be updated.

Old	New
{% dropdown toggle_tippy_offset="[0, -2]" %}...{% enddropdown %}	{% dropdown toggle_tooltip_offset="[0, -2]" %}...{% enddropdown %}

escapescript template tag and escape_script functions are deprecated

As of this release, the undocumented `coreutils.escape_script` util and `escapescript` template tag will no longer be supported.

This was used to provide a way for HTML template content in IE11, which is no longer supported, and was non-compliant with CSP support.

The current approach will trigger a deprecation warning and will be removed in a future release.

Old

```
{% load wagtailadmin_tags %}  
<script type="text/django-form-template" id="id_{{ formset.prefix }}-EMPTY_FORM_  
→TEMPLATE">  
    {% escapescript %}  
        <div>Widget template content</div>  
        <script src="/js/my-widget.js"></script>  
    {% endescapescript %}  
</script>
```

New

Use the HTML `template` element to avoid content from being parsed by the browser on load.

```
<template id="id_{{ formset.prefix }}-EMPTY_FORM_TEMPLATE">  
    <div>Widget template content</div>  
    <script src="/js/my-widget.js"></script>  
</template>
```

Adoption of `classname` convention within the Image Format instance

When using `wagtail.images.formats.Format`, the created instance set the argument for classes to the attribute `classnames` (plural), this has now changed to `classname` (singular).

For any custom code that accessed or modified this undocumented attribute, updates will need to be made as follows.

Accessing `selfclassnames` will still work until a future release, simply returning `selfclassname`, but this will raise a deprecation warning.

```
# image_formats.py  
from django.utils.html import format_html  
from wagtail.images.formats import Format, register_image_format  
  
class CustomImageFormat(Format):  
  
    def image_to_html(self, image, alt_text, extra_attributes=None):  
        # contrived example - pull out the class and render on outside element  
        classname = self.classname # not selfclassnames  
        self.classname = "" # not selfclassnames  
        inner_html = super().image_to_html(image, alt_text, extra_attributes)  
        return format_html("<custom-image class='{}'>{}</custom-image>", classname,
```

(continues on next page)

(continued from previous page)

```

↳ inner_html)

custom_format = CustomImageFormat('custom_example', 'Custom example', 'example-image-'
↳ object-fit', 'width=750')

register_image_format(custom_format)

```

Changes to search promotions contrib module

Deprecated `search_garbage_collect` management command has been removed

In 5.0 the documentation advised that the `search_garbage_collect` command used to remove old stored search queries and daily hits has been moved to `searchpromotions_garbage_collect`.

The old command has now been fully removed and if called will throw an error.

Changes to URL names and templates

Some search promotions URLs and templates have now moved from the main admin search module into the search promotions module.

Item	Old	New
URL name	wagtailsearch_admin:queries_cl	wagtailsearchpromotions:chooser
URL name	wagtailsearch_admin:queries_cl	wagtailsearchpromotions:queries_chooserresults
Template	wagtail/search/templates/ wagtailsearch/queries/ chooser/chooser.html	wagtail/contrib/search_promotions/ templates/wagtailsearchpromotions/ queries/chooser/chooser.html
Template	wagtail/search/templates/ wagtailsearch/queries/ chooser/results.html	wagtail/contrib/search_promotions/ templates/wagtailsearchpromotions/ queries/chooser/results.html
Template	wagtail/search/templates/ wagtailsearch/queries/ chooser_field.html	wagtail/contrib/search_promotions/ templates/wagtailsearchpromotions/ queries/chooser_field.html

`Block.get_template` now accepts a `value` argument

The `get_template` method on StreamField blocks now accepts a `value` argument in addition to `context`. Code using the old signature should be updated:

```

# Old
def get_template(self, context=None):
    ...

# New
def get_template(self, value=None, context=None):
    ...

```

1.11.33 Wagtail 5.1.3 release notes

October 19, 2023

- [What's new](#)

What's new

CVE-2023-45809: Disclosure of user names via admin bulk action views

This release addresses an information disclosure vulnerability in the Wagtail admin interface. A user with a limited-permission editor account for the Wagtail admin can make a direct URL request to the admin view that handles bulk actions on user accounts. While authentication rules prevent the user from making any changes, the error message discloses the display names of user accounts, and by modifying URL parameters, the user can retrieve the display name for any user. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to quyenheu for reporting this issue. For further details, please see the [CVE-2023-45809 security advisory](#).

Bug fixes

- Fix SnippetBulkAction not respecting models definition (Sandro Rodrigues)
- Correctly quote non-numeric primary keys on snippet inspect view (Sage Abdullah)
- Prevent crash on snippet inspect view when displaying a null foreign key to an image (Sage Abdullah)
- Populate the correct return value when creating a new snippet within the snippet chooser (claudobahn)
- Reinstate missing filter by page type on page search (Matt Westcott)
- Use the correct action log when creating a redirect (Thibaud Colas)

1.11.34 Wagtail 5.1.2 release notes

September 25, 2023

- [What's new](#)
- [Upgrade considerations](#)

What's new

Bug fixes

- Avoid use of `ignore_conflicts` when creating extra permissions for snippets, for SQL Server compatibility (Sage Abdullah)
- Ensure sequence on `wagtailsearchpromotions_query` table is correctly set after migrating data (Jake Howard)
- Change spreadsheet export headings to match listing view column headings (Christer Jensen, Sage Abdullah)
- Fix numbers, booleans, and `None` from being exported as strings (Christer Jensen)
- Restore fallback on full-word search for snippet choosers and generic index views (Matt Westcott)
- Restore compatibility with pre-7.15 versions of the Elasticsearch Python library, allowing use of OpenSearch (Matt Westcott)
- Fix error when pickling `BaseSiteSetting` instances (Matt Westcott)
- For Python 3.13 support - upgrade Willow to v1.6.2, replace `imghdr` with Willow's built-in MIME type detection (Jake Howard)

Upgrade considerations

Search within chooser interfaces requires `AutocompleteField` for full functionality

In Wagtail 4.2, the search bar within snippet chooser interfaces (and custom choosers created via `ChooserViewSet`) returned results for partial word matches - for example, a search for "wagt" would return results containing "Wagtail" - if this was supported by the search backend in use, and at least one `AutocompleteField` was present in the model's `search_fields` definition. Otherwise, it would fall back to only matching on complete words. In Wagtail 5.0, this fallback behavior was removed, and consequently a model with no `AutocompleteFields` in place would return no results.

As of Wagtail 5.1.2, the fallback behavior has been restored. Nevertheless, it is strongly recommended that you add `AutocompleteField` to your models' `search_fields` definitions, to ensure that users can receive search results continuously as they type. For example:

```
from wagtail.search import index
# ... other imports

@register_snippet
class MySnippet(index.Indexed, models.Model):
    search_fields = [
        index.SearchField("name"),
        index.AutocompleteField("name"),
    ]
```

1.11.35 Wagtail 5.1.1 release notes

August 14, 2023

- *What's new*

What's new

Other features

- Introduce `wagtail.admin.ui.tables.BooleanField` to display boolean values as icons (Sage Abdullah)

Bug fixes

- Show not-None falsy values instead of blank in generic table cell template (Sage Abdullah)
- Fix `read_only` panels for fields with translatable choice labels (Florent Lebreton)

1.11.36 Wagtail 5.1 release notes

August 1, 2023

- *What's new*
- *Upgrade considerations - changes affecting all projects*
- *Upgrade considerations - deprecation of old functionality*
- *Upgrade considerations - changes affecting Wagtail customizations*

What's new

Read-only panels

FieldPanels can now be marked as read-only with the `read_only=True` keyword argument, so that they are displayed in the admin but cannot be edited. This feature was developed by Andy Babic.

Wagtail tutorial improvements

As part of Google Season of Docs 2023, we worked with technical writer Damilola Oladele to make improvements to Wagtail's "Getting started" tutorial. Here are the specific changes made as part of this project:

- Revamp the start of the getting started section, with a separate quick install page
- Move the tutorial's snippets section to come before tags
- Rewrite the getting started tutorial to address identified friction points
- Switch the Getting started tutorial's snippets example to be more understandable

Thank you to Damilola for his work, and to Google for sponsoring this project.

Custom template support for `wagtail start`

The `wagtail start` command now supports an optional `--template` argument that allows you to specify a custom project template to use. This is useful if you want to use a custom template that includes additional features or customizations. For more details, see [the project template reference](#). This feature was developed by Thibaud Colas.

Search query boosting on Elasticsearch 6 and above

The `boost` option on `SearchField`, to increase the ranking of search results that match on the specified field, is now respected by Elasticsearch 6 and above. This was previously only supported up to Elasticsearch 5, due to a change in Elasticsearch's API. This feature was developed by Shohan Dutta Roy.

Elasticsearch 8 support

This release adds support for Elasticsearch 8. This can be set up by installing a version 8.x release of the `elasticsearch` Python package, and setting `wagtail.search.backends.elasticsearch8` as the search backend. Compatibility updates were contributed by Matt Westcott and Wesley van Lee.

Extend Stimulus adoption

As part of tackling Wagtail's technical debt and improving [CSP compatibility](#), we have continued extending our usage of Stimulus, based on the plans laid out in [RFC 78: Adopt Stimulus](#).

- Add support for `attrs` on `FieldPanel` and other panels to aid in custom Stimulus usage (Aman Pandey, Antoni Martyniuk, LB (Ben) Johnston)
- Migrate Tagit initialization to a Stimulus Controller (LB (Ben) Johnston)
- Migrate legacy dropdown implementation to a Stimulus controller (Thibaud Colas)
- Migrate async header search and search with the Task chooser modal to `w-swap`, a Stimulus controller (LB (Ben) Johnston)
- Replace Bootstrap tooltips with a new `w-tooltip` Stimulus controller (LB (Ben) Johnston)
- Migrate dialog instantiation to a new `w-dialog` Stimulus controller (Loveth Omokaro, LB (Ben) Johnston)
- Support dialog template cloning using a new `w-teleport` Stimulus controller (Loveth Omokaro, LB (Ben) Johnston)

AVIF image support

Wagtail now supports [AVIF](#), a modern image format. We encourage all site implementers to consider using it to improve the performance of the sites and reduce their carbon footprint. For further details, see [image file format](#), [output image format](#) and [image quality](#).

This feature was developed by Aman Pandey as part of the Google Summer of Code program and a [partnership with the Green Web Foundation](#) and Green Coding Berlin, with support from Dan Braghis, Thibaud Colas, Sage Abdullah, Arne Tarara (Green Coding Berlin), and Chris Adams (Green Web Foundation).

Permissions consolidation

This release includes several changes to permissions, to make them easier to use and maintain, as well as to improve performance.

- Add initial implementation of `PagePermissionPolicy` (Sage Abdullah)
- Refactor `UserPagePermissionsProxy` and `PagePermissionTester` to use `PagePermissionPolicy` (Sage Abdullah, Tidiane Dia)
- Optimise queries in collection permission policies using cache on the user object (Sage Abdullah)
- Prevent ‘choose’ permission from being ignored when looking up ‘choose’, ‘edit’ and ‘delete’ permissions in combination (Sage Abdullah)
- Take user’s permissions into account for image / document counts on the admin dashboard (Sage Abdullah)
- Deprecate `UserPagePermissionsProxy` (Sage Abdullah)
- Refactor `GroupPagePermission` to use Django’s Permission model (Sage Abdullah)

Snippet enhancements

We have made several improvements to snippets as part of [RFC 85: Snippets parity with ModelAdmin](#), ahead of the deprecation of ModelAdmin contrib app.

- Add the ability to export snippets listing via `SnippetViewSet.list_export` (Sage Abdullah)
- Add Inspect view to snippets (Sage Abdullah)
- Reorganise snippets documentation to cover customizations and optional features (Sage Abdullah)
- Add docs for migrating from ModelAdmin to Snippets (Sage Abdullah)
- Purge revisions of non-page models in `purge_revisions` command (Sage Abdullah)

Other features

- Mark calls to `md5` as not being used for secure purposes, to avoid flagging on FIPS-mode systems (Sean Kelly)
- Return filters from `parse_query_string` as a `QueryDict` to support multiple values (Aman Pandey)
- Explicitly specify `MenuItem.name` for all admin menu and submenu items (Justin Koestinger)
- Add oEmbed provider patterns for YouTube Shorts (e.g. <https://www.youtube.com/shorts/nX84KctJtG0>) and YouTube Live URLs (valnuro, Fabien Le Frapper)
- Add a predictable default ordering of the “Object/Other permissions” in the Group Editing view, allow this *ordering to be customized* (Daniel Kirkham)

- Implement a new design for chooser buttons with better accessibility (Thibaud Colas)
- Add `AbstractImage.get_renditions()` for efficient generation of multiple renditions (Andy Babic)
- Phone numbers entered via a link chooser will now have any spaces stripped out, ensuring a valid `href="tel:.."` attribute (Sahil Jangra)
- Auto-select the StreamField block when only one block type is declared (Sébastien Corbin)
- Add support for more *advanced Draftail customization APIs* (Thibaud Colas)
- Add support for adding `HTML attrs` on FieldPanel, FieldRowPanel, MultiFieldPanel, and others (Aman Pandey, Antoni Martyniuk, LB (Ben) Johnston)
- Change to always cache renditions (Jake Howard)
- Update link/document rich text tooltips for consistency with the inline toolbar (Albina Starykova)
- Increase the contrast between the rich text / StreamField block picker and the page in dark mode (Albina Starykova)
- Change the default WebP quality to 80 to match AVIF (Aman Pandey)
- Adopt optimized Wagtail logo in the admin interface (Albina Starykova)
- Add support for presenting the userbar (Wagtail button) in dark mode (Albina Starykova)

Bug fixes

- Prevent choosers from failing when initial value is an unrecognized ID such as when moving a page from a location where `parent_page_types` would disallow it (Dan Braghis)
- Move comment notifications toggle to the comments side panel (Sage Abdullah)
- Remove comment button on InlinePanel fields (Sage Abdullah)
- Fix missing link to UsageView from EditView for snippets (Christer Jensen)
- Prevent lowercase conversions of IndexView column headers (Virag Jain)
- Ensure that RichText objects with the same values compare as equal (NikilTn)
- Use `gettext_lazy` on generic model views so that language settings are correctly used (Matt Westcott)
- Prevent JS error when reverting the spinner on a submit button after a validation error (LB (Ben) Johnston)
- Prevent crash when comparing page revisions that include MultipleChooserPanel (Matt Westcott)
- Ensure that title and slug continue syncing after entering non-URL-safe characters (LB (Ben) Johnston)
- Ensure that title and slug are synced on keypress, not just on blur (LB (Ben) Johnston)
- Add a more visible active state for side panel toggle buttons (Thibaud Colas)
- Debounce and optimize live preview panel to prevent excessive requests (Sage Abdullah)
- Page listings actions under the “More” dropdown are now accessible for screen reader and keyboard users (Thibaud Colas)
- Bulk actions under the “More” dropdown are now accessible for screen reader and keyboard users (Thibaud Colas)
- Navigation to translations via the locale dropdown is now accessible for screen reader and keyboard users (Thibaud Colas)
- Make it possible for speech recognition users to reveal chooser buttons (Thibaud Colas)
- Use constant-time comparison for image serve URL signatures (Jake Howard)

- Ensure taggit field type-ahead options show correctly in the dark mode theme (Sage Abdullah)
- Fix the lock description message missing the `model_name` variable when locked only by system (Sébastien Corbin)
- Fix empty blocks created in migration operations (Sandil Ranasinghe)
- Ensure that `gettext_lazy` works correctly when using `verbose_name` on a generic Settings models (Sébastien Corbin)
- Remove unnecessary usage of `innerHTML` when modifying DOM content (LB (Ben) Johnston)
- Avoid `ValueError` when extending `PagesAPIViewSet` and setting `meta_fields` to an empty list (Henry Harutyunyan, Alex Morega)
- Improve accessibility for header search, remove autofocus on page load, advise screen readers that content has changed when results update (LB (Ben) Johnston)
- Fix incorrect override of `PagePermissionHelper.user_can_unpublish_obj()` in `ModelAdmin` (Sébastien Corbin)
- Prevent memory exhaustion when updating a large number of image renditions (Jake Howard)
- Add missing Time Zone conversions and date formatting throughout the admin (Stefan Hammer)
- Ensure that audit logs and revisions consistently use UTC and add migration for existing entries (Stefan Hammer)
- Make sure “critical” buttons have enough color contrast in dark mode (Albina Starykova)
- Improve visibility of scheduled publishing errors in status side panel (Sage Abdullah)
- Avoid N+1 queries in users index view (Tidiane Dia)
- Use a theme-agnostic color token for read-only panels support in dark mode (Thibaud Colas)
- Ensure collapsible StreamBlocks expand as necessary to show validation errors (Storm Heg)
- Ensure userbar dialog can sit above other website content (LB (Ben) Johnston)
- Fix preview panel loading issues (Sage Abdullah)
- Fix `search_promotions 0004_copy_queries` migration for long-lived Wagtail instances (Sage Abdullah)
- Guard against `TypeError` in `0088_fix_log_entry_json_timestamps` migration (Sage Abdullah)
- Add migration to replace JSON null values with empty objects in log entries’ data (Sage Abdullah)
- Fix typo in the `page_header_buttons` template tag when accessing the context’s request object (Robert Rollins)

Documentation

- Document how to add non-`ModelAdmin` views to a `ModelAdminGroup` (Onno Timmerman)
- Document how to add `StructBlock` data to a `StreamField` (Ramon Wenger)
- Update ReadTheDocs settings to v2 to resolve `urllib3` issue in linkcheck extension (Thibaud Colas)
- Update documentation for `log_action` parameter on `RevisionMixin.save_revision` (Christer Jensen)
- Update color customization guidance to include theme-agnostic options (Thibaud Colas)
- Mark LTS releases in release note page titles (Thiago C. S. Tioma)
- Revise main Getting started tutorial for clarity (Kevin Chung (kev-odin))

- Update the [deployment documentation](#) page and remove outdated information (Jake Howard)
- Add more items to performance page regarding pre-fetching images and frontend caching (Jake Howard)
- Add docs for managing stored queries in `searchpromotions` (Scott Foster)

Maintenance

- Removed support for Python 3.7 (Dan Braghis)
- Switch to ruff for flake8 / isort code checking (Oliver Parker)
- Deprecate `insert_editor_css` in favour of `insert_global_admin_css` (Ester Beltrami)
- Optimise use of `specific` on Task and TaskState (Matt Westcott)
- Use table UI component for workflow task index view (Matt Westcott)
- Make header search available on generic index view (Matt Westcott)
- Update pagination behavior to reject out-of-range / invalid page numbers (Matt Westcott)
- Remove color tokens which are duplicates / unused (Thibaud Colas)
- Add tests to help with maintenance of theme color tokens (Thibaud Colas)
- Split out a base listing view from generic index view (Matt Westcott)
- Update type hints in `admin/ui/components.py` so that `parent_context` is mutable (Andreas Nüßlein)
- Optimise the Settings context processor to avoid redundantly finding a Site to improve cache ratios (Jake Howard)
- Convert page listing to a class-based view (Matt Westcott)
- Clean up page reports and type usage views to be independent of page listing views (Matt Westcott)
- Refactor “More” dropdowns, locale selector, “Switch locales”, page actions, to use the same dropdown component (Thibaud Colas)
- Convert the `CONTRIBUTORS` file to Markdown (Dan Braghis)
- Move `django-filter` version upper bound to v23 (Yuekui)
- Update Pillow dependency to allow 10.x, only include support for >= 9.1.0 (Yuekui)
- Replace ModelAdmin history header human readable date template tag (LB (Ben) Johnston)
- Update `uuid` to v9 and `Jest` to v29, with `jest-environment-jsdom` and new snapshot format (LB (Ben) Johnston)
- Update test cases producing undesirable console output due to missing mocks, uncaught errors, warnings (LB (Ben) Johnston)
- Remove unused snippets _header_with_history.html template (Thibaud Colas)
- Migrate away from using the "`wagtailadmin/shared/field_as_li.html`" template include (Storm Heg)
- Upgrade documentation theme `sphinx_wagtail_theme` to v6.1.1 which includes multiple styling fixes and always visible code copy buttons (LB (Ben) Johnston)
- Don’t update the reference index while deleting it (Andy Chosak)

Upgrade considerations - changes affecting all projects

Search within chooser interfaces requires `AutocompleteField` for full functionality

In Wagtail 4.2, the search bar within snippet chooser interfaces (and custom choosers created via `ChooserViewSet`) returned results for partial word matches - for example, a search for “wagt” would return results containing “Wagtail” - if this was supported by the search backend in use, and at least one `AutocompleteField` was present in the model’s `search_fields` definition. Otherwise, it would fall back to only matching on complete words. In Wagtail 5.0, this fallback behavior was removed, and consequently a model with no `AutocompleteFields` in place would return no results.

As of Wagtail 5.1.2, the fallback behavior has been restored. Nevertheless, it is strongly recommended that you add `AutocompleteField` to your models’ `search_fields` definitions, to ensure that users can receive search results continuously as they type. For example:

```
from wagtail.search import index
# ... other imports

@register_snippet
class MySnippet(index.Indexed, models.Model):
    search_fields = [
        index.SearchField("name"),
        index.AutocompleteField("name"),
    ]
```

GroupPagePermission now uses Django’s Permission model

The `GroupPagePermission` model that is responsible for assigning page permissions to groups now uses Django’s `Permission` model instead of a custom string. This means that the `permission_type` `CharField` has been deprecated and replaced with a `permission` `ForeignKey` to the `Permission` model.

In addition to this, “edit” permissions now use the term `change` within the code. As a result, `GroupPagePermissions` that were previously recorded with `permission_type="edit"` are now recorded with a `Permission` object that has the `codename="change_page"` and a `content_type` that points to the `Page` model. Any permission checks that are done using `PagePermissionPolicy` should also use `change` instead of `edit`.

If you have any fixtures for the `GroupPagePermission` model, you will need to update them to use the new `Permission` model. For example, if you have a fixture that looks like this:

```
{
    "pk": 11,
    "model": "wagtailcore.grouppagepermission",
    "fields": {
        "group": ["Event moderators"],
        "page": 12,
        "permission_type": "edit"
    }
}
```

Update it to use a natural key for the `permission` field instead of the `permission_type` field:

```
{
    "pk": 11,
    "model": "wagtailcore.grouppagepermission",
    "fields": {
```

(continues on next page)

(continued from previous page)

```

"group": ["Event moderators"],
"page": 12,
"permission": ["change_page", "wagtailcore", "page"]
}
}

```

If you have any code that creates `GroupPagePermission` objects, you will need to update it to use the `Permission` model instead of the `permission_type` string. For example, if you have code that looks like this:

```

from wagtail.models import GroupPagePermission

permission = GroupPagePermission(group=group, page=page, permission_type="edit")
permission.save()

```

Update it to use the `Permission` model instead:

```

from django.contrib.auth.models import Permission
from wagtail.models import GroupPagePermission

permission = GroupPagePermission(
    group=group,
    page=page,
    permission=Permission.objects.get(content_type__app_label="wagtailcore", codename=
    "change_page"),
)
permission.save()

```

During the deprecation period, the `permission_type` field will still be available on the `GroupPagePermission` model and is used to automatically populate empty `permission` field as part of a system check. The `permission_type` field will be removed in Wagtail 6.0.

The default ordering of Group Editing Permissions models has changed

The ordering for “Object permissions” and “Other permissions” now follows a predictable order equivalent to Django’s default `Model` ordering. This will be different to the previous indeterminate ordering.

The default ordering is now `["content_type__app_label", "content_type__model"]`. See [Customizing the group editor permissions ordering](#) for details on how to customize this order.

JSON-timestamps stored in ModelLogEntry and PageLogEntry are now ISO-formatted and UTC

Previously, timestamps stored in the “data”-`JSONField` of `ModelLogEntry` and `PageLogEntry` have used the custom python format `%d %b %Y %H:%M`. Additionally, the “`go_live_at`” timestamp had been stored with the configured local timezone, instead of UTC.

This has now been fixed, all timestamps are now stored as UTC, and because the “data”-`JSONField` now uses Django’s `DjangoJSONEncoder`, those `datetime` objects are now automatically converted to the ISO format. This release contains a new migration `0088_fix_log_entry_json_timestamps` which converts all existing timestamps used by Wagtail to the new format.

If you’ve developed your own subclasses of `ModelLogEntry`, `PageLogEntry` or `BaseLogEntry`, or used those existing models to create custom log entries, and you’ve stored timestamps similarly to Wagtail’s old implementation (using `strftime("%d %b %Y %H:%M")`). You may want to adapt the storage of those timestamps to a consistent format too.

There are probably three places in your code, which have to be changed:

1. Creation: Instead of using `strftime("%d %b %Y %H:%M")`, you can now store the datetime directly in the “data” field. We’ve implemented a new helper `wagtail.utils.timestamps.ensure_utc()`, which ensures the correct timezone (UTC).
2. Display: To display the timestamp in the user’s timezone and format with a `LogFormatter`, we’ve created `utils.parse(wagtail.utils.timestamps.parse_datetime_localized())` and `render(wagtail.utils.timestamps.render_timestamp())` those timestamps. Look at the existing formatters [here](#).
3. Migration: You can use the code of the above migration ([source](#)) as a guideline to migrate your existing timestamps in the database.

Image Renditions are now cached by default

Wagtail will try to use the cache called “renditions”. If no such cache exists, it will fall back to using the default cache. You can [configure the “renditions” cache](#) to use a different cache backend or to provide additional configuration parameters.

Upgrade considerations - deprecation of old functionality

Removed support for Python 3.7

Python 3.7 is no longer supported as of this release; please upgrade to Python 3.8 or above before upgrading Wagtail.

Pillow dependency update

Wagtail no longer supports Pillow versions below 9.1.0.

Elasticsearch 5 and 6 backends are deprecated

The Elasticsearch 5 and 6 search backends are deprecated and will be removed in a future release; please upgrade to Elasticsearch 7 or above.

`insert_editor_css` hook is deprecated

The `insert_editor_css` hook has been deprecated. The `insert_global_admin_css` hook has the same functionality, and all uses of `insert_editor_css` should be changed to `insert_global_admin_css`.

`wagtail.contrib.modeladmin` is deprecated

As part of the [RFC 85: Snippets parity with ModelAdmin implementation](#), the `wagtail.contrib.modeladmin` app is deprecated. To manage non-page models in Wagtail, use `wagtail.snippets` instead.

If you still rely on `ModelAdmin`, use the separate `wagtail-modeladmin` package. The `wagtail.contrib.modeladmin` module will be removed in a future release.

UserPagePermissionsProxy is deprecated

The undocumented `wagtail.models.UserPagePermissionsProxy` class is deprecated.

If you use the `.for_page(page)` method of the class to get a `PagePermissionTester` instance, you can replace it with `page.permissions_for_user(user)`.

If you use the other methods, they can be replaced via the `wagtail.permission_policies.pages.PagePermissionPolicy` class. The following is a list of the `PagePermissionPolicy` equivalent of each method:

```
from wagtail.models import UserPagePermissionsProxy
from wagtail.permission_policies.pages import PagePermissionPolicy

# proxy = UserPagePermissionsProxy(user)
permission_policy = PagePermissionPolicy()

# proxy.revisions_for_moderation()
permission_policy.revisions_for_moderation(user)

# proxy.explorable_pages()
permission_policy.explorable_instances(user)

# proxy.editable_pages()
permission_policy.instances_user_has_permission_for(user, "change")

# proxy.can_edit_pages()
permission_policy.instances_user_has_permission_for(user, "change").exists()

# proxy.publishable_pages()
permission_policy.instances_user_has_permission_for(user, "publish")

# proxy.can_publish_pages()
permission_policy.instances_user_has_permission_for(user, "publish").exists()

# proxy.can_remove_locks()
permission_policy.user_has_permission(user, "unlock")
```

The `UserPagePermissionsProxy` object that is available in page's `ActionMenuItem` context as `user_page_permissions` (which might be used as part of a `register_page_action_menu_item` hook) has been deprecated. In cases where the `page` object is available (e.g. the page edit view), the `PagePermissionTester` object stored as the `user_page_permissions_tester` context variable can still be used.

The `UserPagePermissionsProxy` object that is available in the template context as `user_page_permissions` as a side-effect of the `page_permissions` template tag has also been deprecated.

If you use the `user_page_permissions` context variable or use the `UserPagePermissionsProxy` class directly, make sure to replace it either with the `PagePermissionTester` or the `PagePermissionPolicy` equivalent.

get_pages_with_direct_explore_permission, get_explorable_root_page, and users_with_page_permission are deprecated

The undocumented `get_pages_with_direct_explore_permission` and `get_explorable_root_page` functions in `wagtail.admin.navigation` are deprecated. They can be replaced with `PagePermissionPolicy().instances_with_direct_explore_permission(user)` and `PagePermissionPolicy().explorable_root_instance(user)`, respectively.

The undocumented `users_with_page_permission` function in `wagtail.admin.auth` is also deprecated. It can be replaced with `PagePermissionPolicy().users_with_permission_for_instance(action, page, include_superuser)`.

Shared include `wagtailadmin/shared/last_updated.html` is no longer available

The undocumented shared include `wagtailadmin/shared/last_updated.html` is no longer available as it used the legacy Bootstrap tooltips and was not accessible. If you need to achieve a similar output, an element that shows a simple date with a tooltip for the full date, use the `human_readable_date` template tag instead.

Before

```
{% include "wagtailadmin/shared/last_updated.html" with last_updated=my_model.  
→timestamp %}
```

After

```
{% load wagtailadmin_tags %}  
  
<!-- ... -->  
{% human_readable_date my_model.timestamp %}
```

Shared include `field_as_li.html` will be removed

The documented include `"wagtailadmin/shared/field_as_li.html"` will be removed in a future release, if being used it will need to be replaced with `"wagtailadmin/shared/field.html"` wrapped within `li` tags.

Before

```
{% include "wagtailadmin/shared/field_as_li.html" %}
```

After

```
<li>
  {%- include "wagtailadmin/shared/field.html" %}</li>
```

Upgrade considerations - changes affecting Wagtail customizations

Tag (Tagit) field usage now relies on data attributes

The `AdminTagWidget` widget has now been migrated to a Stimulus controller, if using this widget in Python, no changes are needed to adopt the new approach.

If the widget is being instantiated in JavaScript or HTML with the global util `window.initTagField`, this undocumented util should be replaced with the new `data-*` attributes approach. Additionally, any direct usage of the jQuery widget in JavaScript (e.g. `$('#my-element').tagit()`) should be removed.

The global util will be removed in a future release. It is recommended that the documented `AdminTagWidget` be used. However, if you need to use the JavaScript approach you can do this with the following example.

Old syntax

```
<input id="id_tags" type="text" value="popular, technology" hidden />
<script>
  window.initTagField('id_tags', 'path/to/url', { autocompleteOnly: true });
</script>
```

New syntax

```
<input
  id="id_tags"
  type="text"
  value="popular, technology"
  hidden
  data-controller="w-tag"
  data-w-tag-options-value='{"autocompleteOnly": true}'
  data-w-tag-url-value="/path/to/url"
/>
```

Note: The `data-w-tag-options-value` is a JSON object serialized into string. Django's HTML escaping will handle it automatically when you use the `AdminTagWidget`, but if you are manually writing the attributes, be sure to use quotation marks correctly.

Header searching now relies on data attributes

Previously the header search relied on inline scripts and a `window.headerSearch` global to activate the behavior. This has now changed to a data attributes approach and the `window` global usage will be removed in a future major release.

If you are using the documented Wagtail `viewsets`, Snippets or `ModelAdmin` approaches to building custom admin views, there should be no change required.

If you are using the shared header template include for a custom search integration, here's how to adopt the new approach.

Header include before

```
{% extends "wagtailadmin/base.html" %}  
{% load wagtailadmin_tags %}  
{% block extra_js %}  
    {{ block.super }}  
    <script>  
        window.headerSearch = {  
            url: "{% url 'myapp:search_results' %}",  
            termInput: '#id_q',  
            targetOutput: '#my-results',  
        };  
    </script>  
{% endblock %}  
{% block content %}  
    {% include "wagtailadmin/shared/header.html" with title="my title" search_url=  
    →"myapp:index" %}  
    ... other content  
{% endblock %}
```

Header include after

Note: No need for `extra_js` usage at all.

```
{% extends "wagtailadmin/base.html" %}  
{% load wagtailadmin_tags %}  
{% block content %}  
    {% url 'myapp:search_results' as search_results_url %}  
    {% include "wagtailadmin/shared/header.html" with title="my title" search_url=  
    →"myapp:index" search_results_url=search_results_url search_target="#my-results" %}  
    ... other content  
{% endblock %}
```

Alternatively, if you have customizations that manually declare or override `window.headerSearch`, here's how to adopt the new approach.

Manual usage before

```
<script>
    window.headerSearch = {
        url: '{{ my_async_results_url }}',
        termInput: '#id_q',
        targetOutput: '#some-results',
    };
</script>
<form role="search">
    <input type="text" name="q" id="id_q" />
</form>
<div id="some-results"></div>
```

Manual usage after

```
<form
    role="search"
    data-controller="w-swap"
    data-action="change->w-swap#searchLazy input->w-swap#searchLazy"
    data-w-swap-src-value="{{ my_async_results_url }}"
    data-w-swap-target-value="#some-results"
>
    <input type="text" name="q" id="id_q" data-w-swap-target="input" />
</form>
<div id="some-results"></div>
```

Tooltips now rely on new data attributes

The undocumented Bootstrap jQuery tooltip widget is no longer in use, you will need to update any HTML that is using these attributes to the new syntax.

```
<!-- Old attributes: -->
<span data-wagtail-tooltip="Tooltip content here">Label</span>
<!-- New attributes: -->
<span data-controller="w-tooltip" data-w-tooltip-content-value="Tooltip content here">
    Label
</span>
```

Dialog hide/show custom events name change

The undocumented client-side Custom Event handling for dialog showing & hiding will change in a future release.

Action	Old event	New event
Show	wagtail:show	w-dialog:show
Hide	wagtail:hide	w-dialog:hide

Additionally, two new events will be dispatched when the dialog visibility changes.

Action	Event name
Show	w-dialog:shown
Hide	w-dialog:hidden

Shared template `.../tables/attrs.html` has been renamed to `.../shared/attrs.html`

The undocumented shared template for rendering a dict of `attrs` to HTML, similar to Django form widgets, has been renamed.

	Template location	Usage with <code>include</code>
Old	<code>wagtail/admin/templates/wagtailadmin/tables/attrs.html</code>	<code>{% include "wagtailadmin/tables/attrs.html" with attrs=link_attrs %}</code>
New	<code>wagtail/admin/templates/wagtailadmin/shared/attrs.html</code>	<code>{% include "wagtailadmin/shared/attrs.html" with attrs=link_attrs %}</code>

1.11.37 Wagtail 5.0.5 release notes

October 19, 2023

- *What's new*

What's new

CVE-2023-45809: Disclosure of user names via admin bulk action views

This release addresses an information disclosure vulnerability in the Wagtail admin interface. A user with a limited-permission editor account for the Wagtail admin can make a direct URL request to the admin view that handles bulk actions on user accounts. While authentication rules prevent the user from making any changes, the error message discloses the display names of user accounts, and by modifying URL parameters, the user can retrieve the display name for any user. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to quyenheu for reporting this issue. For further details, please see the [CVE-2023-45809](#) security advisory.

1.11.38 Wagtail 5.0.4 release notes

October 4, 2023

- *What's new*

What's new

Maintenance

- Relax Willow / Pillow dependency to allow use of current Pillow versions with security fixes (Dan Braghis)

1.11.39 Wagtail 5.0.3 release notes

September 25, 2023

- What's new*

What's new

Bug fixes

- Avoid use of `ignore_conflicts` when creating extra permissions for snippets, for SQL Server compatibility (Sage Abdullah)
- Ensure sequence on `wagtailsearchpromotions_query` table is correctly set after migrating data (Jake Howard)
- Update Pillow dependency to 9.1.0 (Daniel Kirkham)

1.11.40 Wagtail 5.0.2 release notes

June 21, 2023

- What's new*
- Upgrade considerations*

What's new

New features

- Added `TitleFieldPanel` to support title / slug field synchronization (LB (Ben) Johnston)

Bug fixes

- Prevent JS error when reverting the spinner on a submit button after a validation error (LB (Ben) Johnston)
- Prevent crash when comparing page revisions that include `MultipleChooserPanel` (Matt Westcott)
- Ensure that title and slug continue syncing after entering non-URL-safe characters (LB (Ben) Johnston)
- Ensure that title and slug are synced on keypress, not just on blur (LB (Ben) Johnston)
- Add a more visible active state for side panel toggle buttons (Thibaud Colas)
- Use custom dark theme colors for revision comparisons (Thibaud Colas)

Upgrade considerations

Use of `TitleFieldPanel` for the page title field

This release introduces a new `TitleFieldPanel` class, which is used by default for the page title field and provides the mechanism for synchronizing the slug field with the title. Before Wagtail 5.0, this happened automatically on any field named ‘title’.

If you have used `FieldPanel("title")` directly in a panel definition (rather than extending `Page`.`content_panels` as standard), and wish to restore the previous behavior of auto-populating the slug, you will need to change this to `TitleFieldPanel("title")`. For example:

```
from wagtail.admin.panels import FieldPanel, MultiFieldPanel

# ...
content_panels = [
    MultiFieldPanel([
        FieldPanel("title"),
        FieldPanel("subtitle"),
    ]),
]
```

should become:

```
from wagtail.admin.panels import FieldPanel, MultiFieldPanel, TitleFieldPanel

# ...
content_panels = [
    MultiFieldPanel([
        TitleFieldPanel("title"),
        FieldPanel("subtitle"),
    ]),
]
```

1.11.41 Wagtail 5.0.1 release notes

May 25, 2023

- *What's new*

What's new

Bug fixes

- Rectify previous fix for TableBlock becoming uneditable after save (Sage Abdullah)
- Ensure that copying page correctly picks up the latest revision (Matt Westcott)
- Ensure comment buttons always respect WAGTAILADMIN_COMMENTS_ENABLED (Thibaud Colas)
- Fix error when deleting a single snippet through the bulk actions interface (Sage Abdullah)
- Pass the correct `for_update` value for `get_form_class` in `SnippetViewSet` edit views (Sage Abdullah)
- Move comment notifications toggle to the comments side panel (Sage Abdullah)
- Remove comment button on `InlinePanel` fields (Sage Abdullah)
- Fix missing link to `UsageView` from `EditView` for snippets (Christer Jensen)
- Prevent lowercase conversions of `IndexView` column headers (Virag Jain)
- Fix various color issues in dark mode (Thibaud Colas)

Documentation

- Update documentation for `log_action` parameter on `RevisionMixin.save_revision` (Christer Jensen)

1.11.42 Wagtail 5.0 release notes

May 2, 2023

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

What's new

Django 4.2 support

This release adds support for Django 4.2.

Object usage information on deleting objects

On deleting a page, image, document or snippet, the confirmation screen now provides a summary of where the object is used, allowing users to see the effect that deletion will have elsewhere on the site. This also prevents objects from being deleted in cases where deletion would be blocked by an `on_delete=PROTECT` constraint. This feature was developed by Sage Abdullah.

SVG image support

The image library can now be configured to allow uploading SVG images. These are handled by the `{% image %}` template tag as normal, with some limitations on image operations - for full details, see [SVG images](#). This feature was developed by Joshua Munn, and sponsored by YouGov.

Custom validation support for StreamField

Support for adding custom validation logic to StreamField blocks has been formalized and simplified. For most purposes, raising a `ValidationError` from the block's `clean` method is now sufficient; more complex behaviors (such as attaching errors to a specific child block) are possible through block-specific subclasses of `ValidationError`. For more details, see [StreamField validation](#). This feature was developed by Matt Westcott.

Customizable SVG icons

Wagtail's icon set is now fully updated, customizable, and extendable. Built-in icons are now based on the latest [FontAwesome](#) visuals, with capabilities to both customize existing icons as well as add new ones. In particular, this includes:

- A new `{% icon %}` icon template tag to reuse icons in custom templates.
- A `register_icons` hook to register new icons and override existing ones.
- Live documentation of all available icons in the styleguide, showcasing the icons available on the current site.
- A list of [all built-in icons](#) within our developer documentation.
- Support for customizing icons for snippets via `SnippetViewSet.icon`.
- Support for custom panel icons, with defaults, displayed for top-level editor panels.
- New icons for StreamField blocks

For more details, see our new [icons documentation](#).

This has been made possible thanks to a multi-year refactoring effort to migrate all icons to SVG. Thank you to all contributors who participated in this effort: Coen van der Kamp, LB (Ben) Johnston, Dan Braghis, Daniel Kirkham, Sage Abdullah, Thibaud Colas, Scott Cranfill, Storm Heg, Steve Steinwand, Jérôme Lebleu, Abayomi Victory.

Accessibility checker improvements

The [built-in accessibility checker](#) has been updated with:

- 5 more Axe rules enabled by default.
- Sorting of checker results according to their position on the page.
- Highlight styles to more easily identify elements with errors.
- Configuration APIs in [AccessibilityItem](#) for simpler customization of the checks performed.

Those improvements were implemented by Albina Starykova as part of an [Outreachy internship](#), with support from mentors Thibaud Colas, Sage Abdullah, and Joshua Munn.

Always-on minimap

Following its introduction in Wagtail 4.1, we have made several improvements to the page editor minimap:

- It now stays opened until dismissed, so users can keep it expanded if desired.
- Its “expanded” state is preserved when navigating between different views of the CMS.
- The minimap and “Collapse all” button now appear next to side panels rather than underneath, so they can be used at any time.
- Clicking any item reveals the minimap, with appropriate text for screen reader users.
- Navigating to a collapsed section of the page will reveal this section.

Thank you to everyone who provided feedback on this new addition to the editor experience. Those changes were implemented by Thibaud Colas.

Dark mode

Wagtail’s admin interface now supports dark mode. The new dark theme can be enabled in account preferences, as well as configuring permanent usage of the light theme, or following system preferences.

We hope this new theme will bring accessibility improvements for users who prefer light text on dark backgrounds, and energy usage efficiency improvements for users of OLED monitors. This feature was developed by Thibaud Colas, with designs from Ben Enright.

Snippets parity with ModelAdmin

Continuing on recent improvements to snippets, we have made the following additions to how snippets can be customized in the admin interface:

- Allow customizing the base URL and URL namespace for snippet views.
- Allow customizing the default ordering and number of items per page for snippet listing views.
- Allow admin templates for snippets to be overridden on a per-model or per-app basis.
- Allow overriding the base queryset to be used in snippet `IndexView`.
- Allow customizing the `search_fields` and search backend via `SnippetViewSet`.
- Allow filters on snippet index views to be customized through the `list_filter` attribute.
- Allow `panels / edit_handler` to be specified via `SnippetViewSet`.

- Support for customizing icons for snippets via `SnippetViewSet.icon`.
- Allow snippets to be registered into arbitrary admin menu items.

For more details, see [Customizing admin views for snippets](#).

Developed by Sage Abdullah, these features were implemented as part of [RFC 85: Snippets parity with ModelAdmin](#). We will start the deprecation process of the ModelAdmin contrib package in the next feature release and publish it as a separate package for users who wish to continue using it. The ModelAdmin package will be removed in Wagtail 6.0.

Other features

- Add `WAGTAILIMAGES_EXTENSIONS` setting to restrict image uploads to specific file types (Aman Pandey, Ananjan-R)
- Update user list column level to `Access level` to be easier to understand (Vallabh Tiwari)
- Migrate `.button-longrunning` behavior to a Stimulus controller with support for custom label element & duration (Loveth Omokaro)
- Implement new simplified userbar designs (Albina Starykova)
- Add usage view for pages (Sage Abdullah)
- Copy page form now updates the slug field dynamically with a slugified value on blur (Loveth Omokaro)
- Ensure selected collection is kept when navigating from documents or images listings to add multiple views & upon upload (Aman Pandey, Bojan Mihelac)
- Keep applied filters when downloading form submissions (Suyash Srivastava)
- Messages added dynamically via JavaScript now have an icon to be consistent with those supplied in the page's HTML (Aman Pandey)
- Switch lock/unlock side panel toggle to a switch, with more appropriate confirmation message status (Sage Abdullah)
- Ensure that changed or cleared selection from choosers will dispatch a `DOM change` event (George Sakkis)
- Add the ability to [`disable model indexing`](#) by setting `search_fields = []` (Daniel Kirkham)
- Enhance `wagtail.search.utils.parse_query_string` to allow inner single quotes for key/value parsing (Aman Pandey)
- Add helpful properties to [`Locale`](#) for more convenient usage within templates, see [Basic example](#) (Andy Babic)
- Re-label “StreamField blocks” option in block picker to “Blocks” (Thibaud Colas)
- Switch styleguide navigation to use panel components and minimap (Thibaud Colas)
- Explicitly specify `MenuItem.name` for Snippets, Reports, and Settings menu items (Sage Abdullah)
- Move the help text of fields and blocks directly below their label for easier reading (Thibaud Colas)
- The select all checkbox in simple translation's submit translation page will now be in sync with other checkbox changes (Hanoon)
- Revise alignment and spacing of form fields and sections (Thibaud Colas)
- Update Wagtail's type scale so StreamField block labels and field labels are the same size (Thibaud Colas)
- Style comments as per page editor design, in side panel (Karl Hobley, Thibaud Colas)
- ReferenceIndex modified to only index Wagtail-related models, and allow other models to be explicitly registered (Daniel Kirkham)

Bug fixes

- Ensure `label_format` on StructBlock gracefully handles missing variables (Aadi jindal)
- Adopt a no-JavaScript and more accessible solution for the ‘Reset to default’ switch to Gravatar when editing user profile (Loveth Omokaro)
- Ensure `Site.get_site_root_paths` works on cache backends that do not preserve Python objects (Jaap Roes)
- Ignore right clicks on side panel resizer (Sage Abdullah)
- Resize in the correct direction for RTL languages with the side panel resizer (Sage Abdullah)
- Fix image uploads on storage backends that require file pointer to be at the start of the file (Matt Westcott)
- Fix “Edit this page” missing from userbar (Satvik Vashisht)
- No longer allow invalid duplicate site hostname creation as hostnames and domain names are a case insensitive (Coen van der Kamp)
- Image and Document multiple upload update forms now correctly use the progress button (longrunning) behavior when clicked (Loveth Omokaro)
- Prevent audit log report from failing on missing models (Andy Chosak)
- Ensure that the privacy collection privacy edit button is styled as a button (Jatin Kumar)
- Fix page/snippet cannot proceed a `GroupApprovalTask` if it’s locked by someone outside of the group (Sage Abdullah)
- Allow manual lock even if `WorkflowLock` is currently applied (Sage Abdullah)
- Add missing log information for `wagtail.schedule.cancel` (Stefan Hammer)
- Fix timezone activation leaking into subsequent requests in `require_admin_access()` (Stefan Hammer)
- Fix dialog component’s message to have rounded corners at the top side (Sam)
- When multiple documents are uploaded and then subsequently updated, ensure that existing success messages are cleared correctly (Aman Pandey)
- Prevent matches from unrelated models from leaking into SQLite FTS searches (Matt Westcott)
- Prevent duplicate addition of StreamField blocks with the new block picker (Deepam Priyadarshi)
- Enable partial search on images and documents index view where available (Mng)
- Adopt a no-JavaScript and more accessible solution for option selection in reporting, using HTML only `radio` input fields (Mehul Aggarwal)
- Ensure that document search results count shows the correct all matches, not the paginate total (Andy Chosak)
- Fix radio and checkbox elements shrinking when using a long label (Sage Abdullah)
- Fix select elements expanding beyond their container when using a long option label (Sage Abdullah)
- Fix timezone handling of `TemplateResponses` for users with a custom timezone (Stefan Hammer, Sage Abdullah)
- Ensure TableBlock initialization correctly runs after load and its width is aligned with the parent panel (Dan Braghis)
- Ensure that the JavaScript media files are loaded by default in Snippet index listings for date fields (Sage Abdullah)
- Fix server-side caching of the icons sprite (Thibaud Colas)
- Avoid showing scrollbars in the block picker unless necessary (Babitha Kumari)

- Always show Add buttons, guide lines, Move up/down, Duplicate, Delete; in StreamField and Inline Panel (Thibaud Colas)
- Make admin JS i18n endpoint accessible to non-authenticated users (Matt Westcott)
- Autosize text area field will now correctly resize when switching between comments toggle states (Suyash Srivastava)
- Fix incorrect API serialization for document `download_url` when `WAGTAILDOCS_SERVE_METHOD` is `direct` (Swojak-A)
- Fix template configuration of snippets index results view (fidoriel, Sage Abdullah)
- Prevent long preview mode names from making the select element overflow the side panel (Sage Abdullah)
- When i18n is not enabled, avoid making a Locale query on every page view (Dan Braghis)
- Fix initialization of commenting widgets within StreamField (Thibaud Colas)
- Fix various regressions in the commenting UI (Thibaud Colas)
- Prevent TableBlock from becoming uneditable after save (Sage Abdullah)
- Correctly show the “new item” badge within menu sections previously dismissed (Sage Abdullah)
- Fix side panel stuck in resize state when pointer is released outside the grip (Sage Abdullah)

Documentation

- Add code block to make it easier to understand contribution docs (Suyash Singh)
- Fix broken formatting for MultiFieldPanel / FieldRowPanel permission kwarg docs (Matt Westcott)
- Add helpful troubleshooting links and refine wording for getting started with development (Loveth Omokaro)
- Ensure search autocomplete overlay on mobile does not overflow the viewport (Ayman Makroo)
- Improve documentation for InlinePanel (Vallabh Tiwari)
- Remove confusing `SettingsPanel` reference in the page editing `TabbedInterface` example as `SettingsPanel` no longer shows anything as of 4.1 (Kenny Wolf, Julian Bigler)
- Add contributor guidelines for building *Stimulus Controllers* (Thibaud Colas, Loveth Omokaro, LB (Ben) Johnston)
- Fix typo in “Extending Draftail” documentation (Hans Kelson)
- Clarify `ClusterableModel` requirements for using relations with `RevisionMixin`-enabled models (Sage Abdullah)
- Add guide to making your first contribution (LB (Ben) Johnston)

Maintenance

- Removed features deprecated in Wagtail 3.0 and 4.0 (Matt Westcott)
- Update djhtml (html formatting) library to v 1.5.2 (Loveth Omokaro)
- Re-enable `strictPropertyInitialization` in `tsconfig` (Thibaud Colas)
- Refactor accessibility checker userbar item (Albina Starykova)
- Removed unused `Page.get_static_site_paths` method (Yosr Karoui)
- Provisional Django 5.0 compatibility fixes (Sage Abdullah)
- Add unit tests for `CollapseAll` and `MinimapItem` components (Albina Starykova)

- Code quality fixes (GLEF1X)
- Refactor image / document / snippet usage views into a shared generic view (Sage Abdullah)
- Rename the Stimulus `AutoFieldController` to the less confusing `SubmitController` (Loveth Omokaro)
- Replace `script` tags with `template` tag for image/document bulk uploads (Rishabh Kumar Bahukhandi)
- Remove unneeded float styles on 404 page (Fabien Le Frapper)
- Convert userbar implementation to TypeScript (Albina Starykova)
- Migrate slug field behavior to a Stimulus controller and create new `SlugInput` widget (Loveth Omokaro)
- Refactor status HTML usage to shared template tag (Aman Pandey, LB (Ben) Johnston, Himanshu Garg)
- Add curlylint and update djhtml, semgrep versions in pre-commit config (Himanshu Garg)
- Use shared header template for `ModelAdmin` and Snippets type index header (Aman Pandey)
- Move models and forms for `wagtailsearch.Query` to `wagtail.contrib.search_promotions` (Karl Hobley)
- Migrate `initErrorDetection` (tabs error counts) to a Stimulus Controller `w-count` (Aman Pandey)
- Migrate `window.addMessage` behavior to a global event listener & Stimulus Controller approach with `w-messages` (Aman Pandey)
- Update Algolia DocSearch to use new application and correct versioning setup (Thibaud Colas)
- Move snippet choosers and model check registration to `SnippetViewSet.on_register()` (Sage Abdullah)
- Remove unused snippets delete-multiple view (Sage Abdullah)
- Improve performance of determining live page URLs across the admin interface using `pageurl template tag` (Satvik Vashisht)
- Migrate `window.initSlugAutoPopulate` behavior to a Stimulus Controller `w-sync` (Loveth Omokaro)
- Rename status classes to `w-status` to align with preferred CSS class naming conventions (Mansi Gundre)
- Include wagtail-factories in `wagtail.test.utils` to avoid cross-dependency issues (Matt Westcott)
- Fix search tests to correctly reflect behavior of search backends other than the fallback backend (Matt Westcott)
- Migrate select all checkbox in simple translation's submit translation page to Stimulus controller `w-bulk`, remove inline script usage (Hanoon)
- Refactor `SnippetViewSet` to extend `ModelViewSet` (Sage Abdullah)
- Migrate `initDismissibles` behavior to a Stimulus controller `w-dismissible` (Loveth Omokaro)
- Replace jQuery autosize v3 with Stimulus `w-autosize` controller using autosize npm package v6 (Suyash Srivastava)
- Update `w-action` controller to support a click method (Suyash Srivastava)
- Migrate the site settings switcher select from jQuery to a refined version of the `w-action` controller usage (Aadi jindal, LB (Ben) Johnston)
- Always use expanded Sass output so CSS processing is identical in development and production builds (Thibaud Colas)
- Refactor admin color palette to semantic, theme-agnostic design tokens (Thibaud Colas)

Upgrade considerations

Removal of deprecated features

The following features deprecated in Wagtail 3.0 have been fully removed. See [Wagtail 3.0 release notes](#) for details on these changes, including how to remove usage of these features:

- The modules `wagtail.core`, `wagtail.tests`, `wagtail.admin.edit_handlers` and `wagtail.contrib.forms.edit_handlers` are removed.
- The field panel classes `StreamFieldPanel`, `RichTextFieldPanel`, `ImageChooserPanel`, `DocumentChooserPanel` and `SnippetChooserPanel` are removed.
- StreamField definitions must include `use_json_field=True` (except migrations created before Wagtail 5.0).
- The `BASE_URL` setting is no longer recognized.
- The `ModelAdmin.get_form_fields_exclude` method is no longer passed a `request` argument.
- The `ModelAdmin.get_edit_handler` method is no longer passed a `request` or `instance` argument.
- The `widget_overrides`, `required_fields`, `required_formsets`, `bind_to`, `render_as_object` and `render_as_field` methods on `Panel` (previously `EditHandler`) are removed.

The following features deprecated in Wagtail 4.0 have been fully removed. See [Wagtail 4.0 release notes](#) for details on these changes, including how to remove usage of these features:

- The `wagtail.contrib.settings.models.BaseSetting` class is removed.
- The `Page.get_latest_revision_as_page` method is removed.
- The `page` and `page_id` properties and `as_page_object` method on `Revision` are removed.
- The JavaScript functions `createPageChooser`, `createSnippetChooser`, `createDocumentChooser` and `createImageChooser` are removed.
- The `wagtail.contrib.modeladmin.menus.SubMenu` class is removed.
- Subclasses of `wagtail.contrib.modeladmin.helpers.AdminURLHelper` are now required to accept a `base_url_path` keyword argument on the constructor.
- The `wagtail.admin.widgets.chooser.AdminChooser` class is removed.
- The `wagtail.snippets.views.snippets.get_snippet_edit_handler` function is removed.

Dropped support for Django 4.0

Django 4.0 reached end of life on 1st April 2023 and is no longer supported by Wagtail. Django 3.2 (LTS) is still supported until April 2024.

Elasticsearch backend no longer performs partial matching on search

The `search` method on pages, images and documents, and on the backend object returned by `wagtail.search.backends.get_search_backend()`, no longer performs partial word matching when the Elasticsearch backend is in use. Previously, a search query such as `Page.objects.search("cat")` would return results containing the word “caterpillar”, while `Page.objects.search("cat", partial_match=False)` would only return results for the exact word “cat”. The `search` method now always performs exact word matches, and the `partial_match` argument has no effect. This change makes the Elasticsearch backend consistent with the database-backed full-text search backends.

To revert to the previous partial word matching behavior, use the `autocomplete` method instead - for example, `Page.objects.autocomplete("cat")`. It may also be necessary to add an `index.AutocompleteField` entry for the relevant fields on the model’s `search_fields` definition, as the old `SearchField("some_field", partial_match=True)` format is no longer supported.

The `partial_match` argument on `search` and `SearchField` is now deprecated, and should be removed from your code; it will be dropped entirely in Wagtail 6.

ReferenceIndex no longer tracks models used outside of Wagtail

When introduced in Wagtail 4.1, the `ReferenceIndex` model recorded references across all of a project’s models by default. The default set of models being indexed has now been changed to only those used within the Wagtail admin, specifically:

- all Page types
- Images
- Documents
- models registered as Snippets
- models registered with ModelAdmin

This change will remove the impact of the indexing on non-Wagtail apps and models.

If you have models that still require reference indexing, and which are not registered as snippets or with ModelAdmin, you will need to explicitly register them within your app’s `AppConfig.ready()` method. See [Reference index](#) for further details.

The use of `wagtail_reference_index_ignore` to prevent indexing of models is unchanged, but in many cases it may no longer be necessary.

It is recommended that the `rebuild_references_index` management command is run after the upgrade to remove any unnecessary records.

Page.get_static_site_paths method removed

The undocumented `Page.get_static_site_paths` method (which returns a generator of URL paths for use by static site generator packages) has been removed. Packages relying on this functionality should provide their own fallback implementation.

wagtailsearch.Query has moved to wagtail.contrib.search_promotions

The `wagtailsearch.Query` model has been moved from the `search` application to the `contrib` application `wagtail.contrib.search_promotions`. All imports will need to be updated and migrations will need to be run via a management command, some imports will still work with a warning until a future version.

To continue using the `Query` model, you must also add the `wagtail.contrib.search_promotions` application to your project's `INSTALLED_APPS` setting.

Migration command

If you have daily hits records in the `wagtailsearch.Query` you can run the management command to move these records to the new location.

```
./manage.py copy_daily_hits_from_wagtailsearch
```

Managing stored search queries

The `search_garbage_collect` command used to remove old stored search queries and daily hits has been moved to `searchpromotions_garbage_collect`.

Import updates

Import	Old import	New import
Query Model	<code>from wagtail.search.models import Query</code>	<code>from wagtail.contrib.search_promotions.models import Query</code>
Query-Form	<code>from wagtail.search.forms import QueryForm</code>	<code>from wagtail.contrib.search_promotions.forms import QueryForm</code>

Changes to header CSS classes in ModelAdmin templates

If there are custom styles in place for the `ModelAdmin`'s header content or more complex template overrides in use, there are a few changes for the following classes to be aware of.

Content	Old classes	New classes
Heading & search (contains h1)	<code>.left.header-left</code>	<code>.left</code>
Action buttons (header_extra)	<code>.right.header-right</code>	<code>.right</code>

Slug field auto-cleaning now relies on data attributes

The slug field JavaScript behavior was previously attached to any field with an ID of `id_slug`, this has now changed to be any field with the appropriate Stimulus data attributes.

If using a custom edit handler or set of panels for page models, the correct widget will now need to be used for these data attributes to be included. This widget will use the `WAGTAIL_ALLOW_UNICODE_SLUGS` Django setting.

```
from wagtail.admin.widgets.slug import SlugInput
# ... other imports

class MyPage(Page):
    promote_panels = [
        FieldPanel("slug", widget=SlugInput),
        # ... other panels
    ]
```

Additionally, the slug behavior can be attached to any field easily by including the following attributes in HTML or via Django's `widget.attrs`.

```
<input
    type="text"
    name="slug"
    data-controller="w-slug"
    data-action="blur->w-slug#slugify"
/>
```

To allow unicode values, add the `data` attribute value;

```
<input
    type="text"
    name="slug"
    data-controller="w-slug"
    data-action="blur->w-slug#slugify"
    data-w-slug-allow-unicode-value="true"
/>
```

Changes to title / slug field synchronization

The mechanism for synchronizing the slug field with the page title has changed, and is no longer hard-coded to activate on fields named 'title'. Notably, this change affects page panel definitions that use `FieldPanel("title")` directly (rather than the convention of extending `Page.content_panels`), as well as non-page models such as snippets.

To assist in upgrading these definitions, Wagtail 5.0.2 provides a new `TitleFieldPanel` class to be used in place of `FieldPanel("title")`. For example:

```
from wagtail.admin.panels import FieldPanel, MultiFieldPanel

# ...
content_panels = [
    MultiFieldPanel([
        FieldPanel("title"),
        FieldPanel("subtitle"),
    ]),
]
```

should become:

```
from wagtail.admin.panels import FieldPanel, MultiFieldPanel, TitleFieldPanel

# ...
content_panels = [
    MultiFieldPanel([
        TitleFieldPanel("title"),
        FieldPanel("subtitle"),
    ]),
]
```

If you have made deeper customizations to this behavior, or are unable to upgrade to Wagtail 5.0.2 or above, please read on as you may need to make some changes to adopt the new approach.

The title field will sync its value with the slug field on Pages if the Page is not published and the slug has not been manually changed. This JavaScript behavior previously attached to any field with an ID of `id_title`; this has now changed to be any field with the appropriate Stimulus data attributes.

There is a new Stimulus controller `w-sync` which allows any field to change one or more other fields when its value changes, the other field in this case will be the slug field (`w-slug`) with the id `id_slug`.

If you need to hook into this behavior, the new approach will now correctly dispatch `change` events on the slug field. Alternatively, you can modify the data attributes on the fields to adjust this behavior.

To adjust the target field (the one to be updated), you can modify `"data-w-sync-target-value"`, the default being `"body:not(.page-is-live) [data-edit-form] #id_slug"` (find the field with id `id_slug` when the page is not live).

To adjust what triggers the initial check (to see if the fields should be in sync), or the trigger the sync, you can use the Stimulus `data-action` attributes.

```
<input
    id="id_title"
    type="text"
    name="title"
    data-controller="w-sync"
    data-action="focus->w-sync#check blur->w-sync#apply change->w-sync#apply"
    data-w-sync-target-value="body:not(.page-is-live) #some_other_slug"
/>
```

Above we have adjusted these attributes to add a ‘change’ event listener to trigger the sync and also adjusted to look for a field with `some_other_slug` instead.

Auto height/size text area widget now relies on data attributes

If you are using the `wagtail.admin.widgets.AdminAutoHeightTextInput` only, this change will have no impact when upgrading. However, if you are relying on the global `autosize` function at `window.autosize` on the client, this will no longer work.

It is recommended that the `AdminAutoHeightTextInput` widget be used instead. You can also adopt the `data-controller` attribute and this will now function as before. Alternatively, you can simply add the required Stimulus data controller attribute as shown below.

Old syntax

```
<textarea id="story" name="story">It was a dark and stormy night...</textarea>
<script>window.autosize($('story'));</script>
```

New syntax

```
<textarea name="story" data-controller="w-autosize">It was a dark and stormy night...
↪</textarea>
```

There are no additional data attributes supported at this time.

Progress button (`button-longrunning`) now relies on data attributes

The `button-longrunning` class usage has been updated to use the newly adopted Stimulus approach, the previous data attributes will be deprecated in a future release.

If using the old approach, ensure any HTML templates are updated to the new approach before the next major release.

Old syntax

```
<button type="submit" class="button action-save button-longrunning" data-clicked-text=
↪"{{ trans 'Creating...' }}"
{{ icon name="spinner" }}
<em>{{ trans 'Create' }}</em>
</button>
```

New syntax

Minimum required attributes are `data-controller` and a `data-action`.

```
<button type="submit" class="button action-save button-longrunning" data-controller=
↪"w-progress" data-action="w-progress#activate" data-w-progress-active-value="{{
↪trans 'Creating...' }}"
{{ icon name="spinner" }}
<em data-w-progress-target="label">{{ trans 'Create' }}</em>
</button>
```

Examples of additional capabilities

Stimulus `targets` and `actions` can be leveraged to revise the behavior via data attributes.

- `<button ... data-w-progress-duration-value="500" ...>` - custom duration can be declared on the element
- `<button ... class="custom-button" data-w-progress-active-class="custom-button--busy" ...>` - custom ‘active’ class to replace the default `button-longrunning-active` (must be a single string without spaces)
- `<button ... ><strong data-w-progress-target="label">{{ trans 'Create' }}</button>` - any element can be the button label (not just `em`)
- `<button ... data-action="w-progress#activate focus->w-progress#activate" ...>` - any event can be used to trigger the in progress behavior
- `<button ... data-action="w-progress#activate:once" ...>` - only trigger the progress behavior once
- `<button ... data-action="readystatechange@document->w-progress#activate:once" data-w-progress-duration-value="5000" disabled ...>` - disabled on load (once JS starts) and becomes enabled after 5s duration

JavaScript `window.addMessages` replaced with event dispatching

The undocumented `window.addMessage` function is no longer available and will throw an error if called, if similar functionality is required use DOM Event dispatching instead as follows.

```
// old
window.addMessage('success', 'Content has updated');

// new
document.dispatchEvent(
    new CustomEvent('w-messages:add', {
        detail: { text: 'Content has updated', type: 'success' },
    }),
);
// new (clearing existing messages before adding a new one)
document.dispatchEvent(
    new CustomEvent('w-messages:add', {
        detail: {
            clear: true,
            text: 'All content has updated',
            type: 'success',
        },
    }),
);
// message types 'success', 'error', 'warning' are supported
```

Note that this event name may change in the future and this functionality is still not officially supported.

Changes to StreamField ValidationError classes

The client-side handling of StreamField validation errors has been updated. The JavaScript classes `StreamBlockValidationError`, `ListBlockValidationError`, `StructBlockValidationError` and `TypedTableBlockValidationError` have been removed, and the corresponding Python classes can no longer be serialized using Telepath. Instead, the `setError` methods on client-side block objects now accept a plain JSON representation of the error, obtained from the `as_json_data` method on the Python class. Custom JavaScript code that works with these objects must be updated accordingly.

Additionally, the Python `StreamBlockValidationError`, `ListBlockValidationError`, `StructBlockValidationError` and `TypedTableBlockValidationError` classes no longer provide a `params` dict with `block_errors` and `non_block_errors` items; these are now available as the attributes `block_errors` and `non_block_errors` on the exception itself (or `cell_errors` and `non_block_errors` in the case of `TypedTableBlockValidationError`).

Snippets delete-multiple view removed

The ability to remove multiple snippet instances from the `DeleteView` and the undocumented `wagtailsnippets_{app_label}_{model_name}:delete-multiple` URL pattern have been removed. The view's functionality has been replaced by the `delete` action of the bulk actions feature introduced in Wagtail 4.0.

The `delete` bulk action view now also calls the `{before,after}_delete_snippet` hooks, in addition to the `{before,after}_bulk_action` hooks.

If you have customized the `IndexView` and/or `DeleteView` views in a `SnippetViewSet` subclass, make sure that the `delete_multiple_url_name` attribute is renamed to `delete_url_name`.

Snippets index views template name changed

The template name for the index view of a snippet model has changed from `wagtailsnippets/snippets/type_index.html` and `wagtailsnippets/snippets/results.html` to `wagtailsnippets/snippets/index.html` and `wagtailsnippets/snippets/index_results.html`. In addition, the model index view that lists the snippet types now looks for the template `wagtailsnippets/snippets/model_index.html` before resorting to the generic index template. If you have customized these templates, make sure to update them accordingly.

status classes are now w-status

Please update any custom styling or usage within the admin when working with status tags to the following new classes.

Old	New
<code>status-tag</code>	<code>w-status</code>
<code>primary</code>	<code>w-status--primary</code>
<code>disabled</code>	<code>w-status--disabled</code>
<code>status-tag--label</code>	<code>w-status--label</code>

Note that a new template tag has been built for usage within the admin that may make it easier to generate status tags.

```
{% load wagtailadmin_tags %}  
{% status "live" url="/test-url/" title=trans_title hidden_label=trans_hidden_label_  
→classname="w-status--primary" attrs='target="_blank" rel="noreferrer"' %}  
{% status status_label classname="w-status--primary" %}
```

Deprecated icon font

The Wagtail icon font has been deprecated and will be removed in a future release, as it is now unused in Wagtail itself. There are no changes to make for any icons usage via dedicated APIs such as `icon` class properties. Any direct icon font usage needs to be converted to SVG icons instead, as documented in our [icons overview](#).

To check whether your project uses the icon font, check for occurrences of:

- Loading of the `wagtail.woff` font file.
- Usage of `font-family: wagtail` in CSS.
- `icon-<name>` CSS classes outside of SVG elements.

Deprecated icons

The following icons are unused in Wagtail itself and will be removed in a future release. If you are using any of these icons, please replace them with an alternative (see our full [list of icons](#)), or re-add the icon to your own project.

Icon name	Alternative
angle-double-left	arrow-left
angle-double-right	arrow-right
arrow-down-big	arrow-down
arrow-up-big	arrow-up
arrows-up-down	order
chain-broken	link
chevron-down	arrow-down (identical)
dots-vertical	dots-horizontal
download-alt	download (identical)
duplicate	copy (identical)
ellipsis-v	dots-horizontal
horizontalrule	minus
repeat	rotate
reset	rotate
tick	check (identical)
undo	rotate
uni52	folder-inverse (identical)
wagtail-inverse	wagtail-icon

`Snippets get_admin_url_namespace() and get_admin_base_path()` moved to `SnippetViewSet`

The undocumented `get_admin_url_namespace()` and `get_admin_base_path()` methods that were set on snippet models at runtime have been moved to the `SnippetViewSet` class. If you use these methods, you could access them via `SnippetModel.snippet_viewset.get_admin_url_namespace()` and `SnippetModel.snippet_viewset.get_admin_base_path()`, respectively.

`Snippets get_usage() and usage_url()` methods removed

The undocumented `get_usage()` and `usage_url()` methods that were set on snippet models at runtime have been removed. Calls to the `get_usage()` method can be replaced with `wagtail.models.ReferenceIndex.get_grouped_references_to(object)`. The `usage_url()` method does not have a direct replacement, but the URL name can be retrieved via `SnippetModel.snippet_viewset.get_url_name("usage")`, which can be used to construct the URL with `reverse()`.

1.11.43 Wagtail 4.2.4 release notes

May 25, 2023

- *What's new*

What's new

Bug fixes

- Rectify previous fix for TableBlock becoming uneditable after save (Sage Abdullah)
- Ensure that copying page correctly picks up the latest revision (Matt Westcott)
- Adjust collection field alignment in multi-upload forms (LB (Ben) Johnston)
- Prevent lowercase conversions of IndexView column headers (Virag Jain)

Documentation

- Update documentation for `log_action` parameter on `RevisionMixin.save_revision` (Christer Jensen)

1.11.44 Wagtail 4.2.3 release notes

May 2, 2023

- *What's new*

What's new

Bug fixes

- Prevent TableBlock from becoming uneditable after save (Sage Abdullah)

1.11.45 Wagtail 4.2.2 release notes

April 3, 2023

- *What's new*

What's new

CVE-2023-28836: Stored XSS attack via ModelAdmin views

This release addresses a stored cross-site scripting (XSS) vulnerability on ModelAdmin views within the Wagtail admin interface. A user with a limited-permission editor account for the Wagtail admin could potentially craft pages and documents 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, and only affects sites with ModelAdmin enabled.

Many thanks to Thibaud Colas for reporting this issue. For further details, please see [the CVE-2023-28836 security advisory](#).

CVE-2023-28837: Denial-of-service via memory exhaustion when uploading large files

This release addresses a memory exhaustion bug in Wagtail's handling of uploaded images and documents. For both images and documents, files are loaded into memory during upload for additional processing. A user with access to upload images or documents through the Wagtail admin interface could upload a file so large that it results in a crash or denial of service.

The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin. It can only be exploited by admin users with permission to upload images or documents.

Many thanks to Jake Howard for reporting this issue. For further details, please see [the CVE-2023-28837 security advisory](#).

Bug fixes

- Fix radio and checkbox elements shrinking when using a long label (Sage Abdullah)
- Fix select elements expanding beyond their container when using a long option label (Sage Abdullah)
- Fix timezone handling of `TemplateResponses` for users with a custom timezone (Stefan Hammer, Sage Abdullah)
- Ensure `TableBlock` initialization correctly runs after load and its width is aligned with the parent panel (Dan Braghis)
- Ensure that the JavaScript media files are loaded by default in Snippet index listings for date fields (Sage Abdullah)
- Fix server-side caching of the icons sprite (Thibaud Colas)
- Avoid showing scrollbars in the block picker unless necessary (Babitha Kumari)
- Always show Add buttons, guide lines, Move up/down, Duplicate, Delete; in StreamField and Inline Panel (Thibaud Colas)
- Ensure datetimepicker widget overlay shows over modals & drop-downs (LB (Ben) Johnston)

Documentation

- Fix module path for `MultipleChooserPanel` in panel reference docs

Maintenance

- Render large image renditions to disk (Jake Howard)

1.11.46 Wagtail 4.2.1 release notes

March 13, 2023

- *What's new*

What's new

Bug fixes

- Support creating `StructValue` copies (Tidiane Dia)
- Fix image uploads on storage backends that require file pointer to be at the start of the file (Matt Westcott)
- Fix “Edit this page” missing from userbar (Satvik Vashisht)
- Prevent audit log report from failing on missing models (Andy Chosak)
- Fix page/snippet cannot proceed a `GroupApprovalTask` if it’s locked by someone outside of the group (Sage Abdullah)
- Add missing log information for `wagtail.schedule.cancel` (Stefan Hammer)
- Fix timezone activation leaking into subsequent requests in `require_admin_access()` (Stefan Hammer)
- Fix dialog component’s message to have rounded corners at the top side (Sam)
- Prevent matches from unrelated models from leaking into SQLite FTS searches (Matt Westcott)
- Prevent duplicate addition of StreamField blocks with the new block picker (Deepam Priyadarshi)
- Update Algolia DocSearch to use new application and correct versioning setup (Thibaud Colas)

Documentation

- Docs: Clarify `ClusterableModel` requirements for using relations with `RevisionMixin`-enabled models (Sage Abdullah)

1.11.47 Wagtail 4.2 release notes

February 6, 2023

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

What's new

StreamField data migration helpers

Wagtail now provides a set of utilities for creating data migrations on StreamField data. For more information, see [*StreamField data migrations*](#). This feature was developed by Sandil Ranasinghe, initially as the `wagtail-streamfield-migration-toolkit` add-on package, as part of the Google Summer of Code 2022 initiative, with support from Jacob Topp-Mugglesstone, Joshua Munn and Karl Hobley.

Locking for snippets

Snippets can now be locked by users to prevent other users from editing, through the use of the `LockableMixin`. For more details, see [*Locking snippets*](#).

This feature was developed by Sage Abdullah.

Workflows for snippets

Snippets can now be assigned to workflows through the use of the `WorkflowMixin`, allowing new changes to be submitted for moderation before they are published. For more details, see [*Enabling workflows for snippets*](#).

This feature was developed by Sage Abdullah.

`fullpageurl` template tag

Wagtail now provides a `fullpageurl` template tag (for both Django templates and Jinja2) to output a page's full URL including the domain. For more details, see [*fullpageurl*](#).

This feature was developed by Jake Howard.

CSP compatibility & Stimulus adoption

Wagtail now uses the Stimulus framework for client-side interactivity (see [RFC 78](#)). Our Outreachy contributor Loveth Omokaro has refactored significant portions of the admin interface:

- The Skip Link component displayed on all pages.
- The dashboard's Upgrade notification message.
- Auto-submitting of listing filters.
- Loading of Wagtail's icon sprite
- Page lock/unlock actions
- Workflow enable actions

Those changes improve the maintainability of the code, and help us move towards compatibility with strict CSP (Content Security Policy) rules. Thank you to Loveth and project mentors LB (Ben Johnston, Thibaud Colas, and Paarth Agarwal.

Accessibility checker integration

The CMS now includes an accessibility checker in the [user bar](#), to assist users in building more accessible websites and follow [ATAG 2.0 guidelines](#). The checker, which is based on the Axe testing engine, is designed for content authors to identify and fix accessibility issues on their own. It scans the loaded page for errors and displays the results, with three rules turned on in this release. It's configurable with the `construct_wagtail_userbar` hook.

This new feature was implemented by Albina Starykova as part of an [Outreachy internship](#), with support from mentors Thibaud Colas, Sage Abdullah, and Joshua Munn.

Rich text improvements

Following feedback from Wagtail users on [rich text UI improvements in Wagtail 4.0](#), we have further refined the behavior of rich text fields to cater for different scenarios:

- Users can now choose between an “inline” floating toolbar, and a fixed toolbar at the top of the editor. Both toolbars display all formatting options.
- The ‘/’ command palette and block picker in rich text fields now contain all formatting options except text styles.
- The ‘/’ command palette and block picker are now always available no matter where the cursor is placed, to support inserting content at any point within text, transforming existing content, and splitting StreamField blocks in the middle of a paragraph when needed.
- The block picker interface now displays two columns so more options are visible without scrolling.

Thank you to all who provided feedback, participants to our usability testing sessions, and to Nick Lee and Thibaud Colas for the implementation.

Multiple chooser panel

A new panel type `MultipleChooserPanel` is available. This is a variant of `InlinePanel` which improves the editor experience when adding large numbers of linked item - rather than creating and populating each sub-form individually, a chooser modal is opened allowing multiple objects to be selected at once.

This feature was developed by Matt Westcott, and sponsored by [YouGov](#).

Other features

- Test assertion `WagtailPageTestCase.assertCanCreate` now supports the kwarg `publish=True` to determine whether to publish the page (Harry Percival, Akua Dokua Asiedu, Matt Westcott)
- Ensure that the `rebuild_references_index` command can run without console output if called with `--verbosity 0` (Omerzahid Ali, Aman Pandey)
- Add full support for secondary buttons with icons in the Wagtail design system - `button bicolor button--icon button-secondary` including the `button-small` variant (Seremba Patrick)
- Add `purge_embeds` management command to delete all the cached embed objects in the database (Aman Pandey)
- Make it possible to resize the page editor’s side panels (Sage Abdullah)
- Add ability to include `form_fields as an APIField` on `FormPage` (Sævar Öfjörð Magnússon, Suyash Singh, LB (Ben) Johnston)
- Ensure that images listings are more consistently aligned when there are fewer images uploaded (Theresa Okoro)

- Add more informative validation error messages for non-unique slugs within the admin interface and for programmatic page creation (Benjamin Bach)
- Always show the page editor title field's border when the field is empty (Thibaud Colas)
- Snippet models extending `DraftStateMixin` now automatically define a “Publish” permission type (Sage Abdullah)
- Users now remain on the edit page after saving a snippet as draft (Sage Abdullah)
- Base project template now populates the meta description tag from the search description field (Aman Pandey)
- Added support for `azure-mgmt-cdn` version ≥ 10 and `azure-mgmt-frontdoor` version ≥ 1 in the frontend cache invalidator (Sylvain Fankhauser)
- Add a system check to warn when a `django-storages` backend is configured to allow overwriting (Rishabh Jain)
- Update admin focus outline color to have higher contrast against white backgrounds (Thibaud Colas)
- Implement latest design for the admin dashboard header (Thibaud Colas, Steven Steinwand)
- Restyle the userbar to follow the visual design of the Wagtail admin (Albina Starykova)
- Adjust the size of panel labels on the “Account” form (Thibaud Colas)
- Delay hiding the contents of the side panels when closing, so the animation is smoother (Thibaud Colas)
- ListBlock now shows item-by-item differences when comparing versions (Tidiane Dia)
- Switch StreamField blocks to use the same picker interface as within rich text fields (Thibaud Colas)

Bug fixes

- Make sure workflow timeline icons are visible in high-contrast mode (Loveth Omokaro)
- Ensure authentication forms (login, password reset) have a visible border in Windows high-contrast mode (Loveth Omokaro)
- Ensure visual consistency between buttons and links as buttons in Windows high-contrast mode (Albina Starykova)
- Ensure `ChooserBlock.extract_references` uses the model class, not the model string (Alex Tomkins)
- Incorrectly formatted link in the documentation for Wagtail community support (Bolarinwa Comfort Ajayi)
- Ensure logo shows correctly on log in page in Windows high-contrast mode (Loveth Omokaro)
- Comments notice background overflows its container (Yekasumah)
- Ensure links within help blocks meet color contrast guidelines for accessibility (Theresa Okoro)
- Ensure the skip link (used for keyboard control) meets color contrast guidelines for accessibility (Dauda Yusuf)
- Ensure tag fields correctly show in both dark and light Windows high-contrast modes (Albina Starykova)
- Ensure new tooltips & tooltip menus have visible borders and tip triangle in Windows high-contrast mode (Juliet Adeboye)
- Ensure there is a visual difference of ‘active/current link’ vs normal links in Windows high-contrast mode (Mohammad Areeb)
- Avoid issues where trailing whitespace could be accidentally removed in translations for new page & snippet headers (Florian Vogt)
- Make sure minimap error indicators follow the minimap scrolling (Thibaud Colas)

- Remove the ability to view or add comments to `InlinePanel` inner fields to avoid lost or incorrectly linked comments (Jacob Topp-Mugglestone)
- Use consistent heading styles on top-level fields in the page editor (Sage Abdullah)
- Allow button labels to wrap onto two lines in dropdown buttons (Coen van der Kamp)
- Remove spurious horizontal resize handle from text areas (Matt Westcott)
- Move `DateField`, `DateTimeField`, `TimeField` comment buttons to be right next to the fields (Theresa Okoro)
- Support text resizing in workflow steps cards (Ivy Jeptoo)
- Ignore images added via fixtures when using `WAGTAILIMAGES_FEATURE_DETECTION_ENABLED` to avoid errors for images that do not exist (Aman Pandey)
- Restore ability to perform `JSONField` query operations against `StreamField` when running against the Django 4.2 development branch (Sage Abdullah)
- Ensure there is correct grammar and pluralization for Tab error counts shown to screen readers (Aman Pandey)
- Pass through expected `cc`, `bcc` and `reply_to` to the Django mail helper from `wagtail.admin.mail.send_mail` (Ben Gosney)
- Allow reviewing or reverting to a Page's initial revision (Andy Chosak)
- Use the correct padding for autocomplete block picker (Umar Farouk Yunusa)
- Ensure that short content pages (such as editing snippets) do not show an inconsistent background (Sage Abdullah)
- Fix horizontal positioning of rich text inline toolbar (Thibaud Colas)
- Ensure that `DecimalBlock` correctly handles `None`, when `required=False`, values (Natarajan Balaji)
- Close the userbar when clicking its toggle (Albina Starykova)
- Add a border around the userbar menu in Windows high-contrast mode so it can be identified (Albina Starykova)
- Make sure browser font resizing applies to the userbar (Albina Starykova)
- Fix check for `delete_url_name` attribute in generic `DeleteView` (Alex Simpson)
- Re-implement design system colors so HSL values exactly match the desired RGB (Albina Starykova)
- Resolve issue where workflow and other notification emails would not include the correct tab URL for account notification management (LB (Ben) Johnston)
- Use consistent spacing above and below page headers (Thibaud Colas)
- Use the correct icon sizes and spacing in slim header (Thibaud Colas)
- Use the correct color for placeholders in rich text fields (Thibaud Colas)
- Prevent obstructing the outline around rich text fields (Thibaud Colas)
- Page editor dropdowns now use indigo backgrounds like elsewhere in the admin interface (Thibaud Colas)
- Allow parsing of multiple key/value pairs from string in `wagtail.search.utils.parse_query_string` (Beniamin Bucur)
- Prevent memory exhaustion when purging a large number of revisions (Jake Howard)
- Add right-to-left (RTL) support for the following form components: Switch, Minimap, live preview (Thibaud Colas)
- Improve right-to-left (RTL) positioning for the following components: Page explorer, Sidebar sub-menu, rich text tooltips, rich text toolbar trigger, editor section headers (Thibaud Colas)
- Center-align `StreamField` and rich text block picker buttons with the dotted guide line (Thibaud Colas)

- Search bar in chooser modals now performs autocomplete searches under PostgreSQL (Matt Westcott)
- Server-side document filenames are preserved when replacing a document file (Suyash Singh, Matt Westcott)
- Do not show bulk actions checkbox in page type usage view (Sage Abdullah)
- Prevent account name from overflowing the sidebar (Aman Pandey)
- Ensure edit form is displayed as unlocked immediately after canceling a workflow (Sage Abdullah)
- Prevent `latest_revision` pointer from being copied over when copying translatable snippets for translation (Sage Abdullah)

Documentation

- Wagtail's documentation (v2.9 to v4.0) has been updated on [Dash user contributions](#) for [Dash](#) or [Zeal](#) offline docs applications (Damilola Oladele, Mary Ayobami, Elizabeth Bassey)
- Wagtail's documentation (v2 to v4.0) has been added to [DevDocs](#) which has offline support and is easily accessible in any browser (Vallabh Tiwari)
- Add custom permissions section to permissions documentation page (Dan Hayden)
- Add documentation for how to get started with [*contributing translations*](#) for the Wagtail admin (Ogunbanjo Oluwadamilare)
- Officially recommend `fpm` over `nvm` in development documentation (LB (Ben) Johnston)
- Mention the importance of passing `request` and `current_site` to `get_url` on the [*performance*](#) documentation page (Jake Howard)
- Add documentation for [*register_user_listing_buttons*](#) hook (LB (Ben Johnston))
- Add development (contributing to Wagtail) documentation notes for [*development on Windows*](#) (Akua Dokua Asiedu)
- Mention Wagtail's usage of Django's default user model by default (Temidayo Azeez)
- Add links to treebeard documentation for relevant methods (Temidayo Azeez)
- Add clarification on where to register entity plugins (Mark McOske)
- Fix logo in README not being visible in high-contrast mode (Benita Anawonah)
- Improve 'first wagtail site' tutorial (Akua Dokua Asiedu)
- Grammatical adjustments of `page models` usage guide (Damilola Oladele)
- Add class inheritance information to StreamField block reference (Temidayo Azeez)
- Document the hook [*register_image_operations*](#) and add an example of a [*custom Image filter*](#) (Coen van der Kamp)
- Fix incorrect example code for StreamField migration of `RichTextField` (Matt Westcott)
- Document the policy needed to create invalidations in CloudFront (Jake Howard)
- Document how to add permission restriction to a report view (Rishabh Jain)
- Add example for how to configure API `renderer_classes` (Aman Pandey)
- Document potential data loss for `BaseLogEntry` migration in 3.0 (Sage Abdullah)
- Add documentation for the reference index mechanism (Daniel Kirkham)

Maintenance

- Switch to using [Willow](#) instead of Pillow for images (Darrel O’Pry)
- Remove unsquashed `testapp` migrations (Matt Westcott)
- Upgrade to Node 18 for frontend build tooling (LB (Ben) Johnston)
- Run Python tests with coverage and upload coverage data to codecov (Sage Abdullah)
- Clean up duplicate JavaScript for the `escapeHtml` function (Jordan Rob)
- Ensure that translation file generation ignores JavaScript unit tests and clean up unit tests for Django gettext utils (LB (Ben Johnston))
- Migrated `initButtonSelects` from `core.js` to own TypeScript file and add unit tests (Loveth Omokaro)
- Migrated `initSkipLink` util to TypeScript and add JSDoc & unit tests (Juliet Adeboye)
- Clean up some unused utility classes and migrate `unlist` to Tailwind utility class `w-list-none` (Loveth Omokaro)
- Clean up linting on legacy code and add shared util `hasOwn` in TypeScript (Loveth Omokaro)
- Remove unnecessary box-sizing: border-box declarations in SCSS (Albina Starykova)
- Migrated `initTooltips` to TypeScript add JSDoc and unit tests (Fatuma Abdullahi)
- Migrated `initTagField` from `core.js` to own TypeScript file and add unit tests (Chisom Okeoma)
- Added unit tests & JSDoc to `initDissmisibles` (Yekasumah)
- Standardise on `classname` for passing HTML class attributes (LB (Ben Johnston))
- Clean up expanding `formset` and `InlinePanel` JavaScript initialization code and adopt a class approach (Matt Westcott)
- Extracted revision and draft state logic from generic views into mixins (Sage Abdullah)
- Extracted generic lock / unlock views from page lock / unlock views (Sage Abdullah)
- Move `identity` JavaScript util into shared utils folder (LB (Ben Johnston))
- Remove unnecessary declaration of function to determine URL query params, instead use `URLSearchParams` (Loveth Omokaro)
- Update `tsconfig` to better support modern TypeScript development and clean up some code quality issues via Eslint (Loveth Omokaro)
- Switch userbar to initialize a Web Component to avoid styling clashes (Albina Starykova)
- Refactor userbar stylesheets to use the same CSS loading as the rest of the admin (Albina Starykova)
- Remove unused search-bar and button-filter styles (Thibaud Colas)
- Use util method to construct dummy requests in tests (Jake Howard)
- Remove unused dev-only react-axe integration (Thibaud Colas)
- Split up `wagtail.admin.panels` into submodules, existing exports have been preserved (Matt Westcott)
- Refactor userbar styles to use the same stylesheet as other components (Thibaud Colas)
- Add deprecation warnings for `wagtail.core` and other imports deprecated in Wagtail 3.0 (Matt Westcott)
- Upgraded Transifex configuration to v3 (Loic Teixeira)
- Replace repeated HTML `avatar` component with a template tag `include { % avatar ... % }` throughout the admin interface (Aman Pandey)

- Refactor accessibility checker userbar item (Albina Starykova)

Upgrade considerations

Wagtail-specific image field (`WagtailImageField`)

The `AbstractImage` and `AbstractRendition` models use a Wagtail-specific `WagtailImageField` which extends Django's `ImageField` to use `Willow` for image file handling. This will generate a new migration if you are using a *custom image model*.

Comments within `InlinePanel` not supported

When the commenting system was introduced, support for `InlinePanel` fields was incorrectly added. This has led to issues where comments can be lost on save, or in most cases will be added to the incorrect item within the `InlinePanel`. The ability to add comments here has now been removed and as such any existing comments that were added will no longer show.

See <https://github.com/wagtail/wagtail/issues/9685> for tracking of adding this back officially in the future.

Adoption of `classname` convention for some template tags & includes

Some undocumented Wagtail admin template tags and includes have been refactored to adopt a more consistent naming of `classname`.

If these are used within packages or customizations they will need to be updated to the new variable naming convention.

Name	New (<code>classname</code>)	Old (various)
icon (see note)	<code>{% icon name='spinner' class-name='...' %}</code>	<code>{% icon name='spinner' class_name='...' %}</code>
dia- log_toc	<code>{% dialog_toggle classname='...' %}</code>	<code>{% dialog_toggle class_name='...' %}</code>
pagi- nate	<code>{% paginate pages classname="..." %}</code>	<code>{% paginate pages classnames="..." %}</code>
tab_nav	<code>{% include 'wagtailadmin/shared/tabs/tab_nav_link.html' with classname="..." %}</code>	<code>{% include 'wagtailadmin/shared/tabs/tab_nav_link.html' with classes="..." %}</code>
side_pa	<code>{% include 'wagtailadmin/shared/side_panels/includes/ side_panel_button.html' with classname="..." %}</code>	<code>{% include 'wagtailadmin/shared/side_panels/includes/ side_panel_button.html' with classes="..." %}</code>

Note that the `icon` template tag will still support `class_name` with a deprecation warning. Support will be dropped in a future release.

InlinePanel JavaScript function is now a class

The (internal, undocumented) `InlinePanel` JavaScript function, used to initialize client-side behavior for inline panels, has been converted to a class. Any user code that calls this function should now replace `InlinePanel(...)` calls with `new InlinePanel(...)`. Additionally, child form controls are now initialized automatically, and so it is no longer necessary to call `initChildControls`, `updateChildCount`, `updateMoveButtonDisabledStates` or `updateAddButtonState`.

Python code that uses the `InlinePanel` panel type is not affected by this change.

WAGTAILADMIN_GLOBAL_PAGE_EDIT_LOCK setting is now WAGTAILADMIN_GLOBAL_EDIT_LOCK

The `WAGTAILADMIN_GLOBAL_PAGE_EDIT_LOCK` setting has been renamed to `WAGTAILADMIN_GLOBAL_EDIT_LOCK`.

Wagtail userbar as a web component

The `wagtailuserbar` template tag now initializes the userbar as a [Web Component](#), with a `wagtail-userbar` custom element using shadow DOM to apply styles without any collisions with the host page.

For any site customizing the position of the userbar, target the styles to `wagtail-userbar::part(userbar)` instead of `.wagtail-userbar`. For example:

```
wagtail-userbar::part(userbar) {  
    bottom: 30px;  
}
```

Configuration of the accessibility checker user bar item

Like other userbar items, the new accessibility checker is configurable with the `construct_wagtail_userbar` hook. For example, to remove the new item, use:

```
from wagtail.admin.userbar import AccessibilityItem  
  
@hooks.register('construct_wagtail_userbar')  
def remove_userbar_accessibility_checks(request, items):  
    items[:] = [item for item in items if not isinstance(item, AccessibilityItem)]
```

Support for legacy versions of azure-mgmt-cdn and azure-mgmt-frontdoor packages will be dropped

If you are using the front-end cache invalidator module (`wagtail.contrib.frontend_cache`) with Azure CDN or Azure Front Door, the following packages need to be updated:

- For Azure CDN: upgrade `azure-mgmt-cdn` to version 10 or above
- For Azure Front Door: upgrade `azure-mgmt-frontdoor` to version 1 or above

Support for older versions will be dropped in a future release.

Changes to Workflow and Task methods

To accommodate workflows support for snippets, the `page` parameter in `Workflow.start()` has been renamed to `obj`.

In addition, some methods on the base `Task` model have been changed. If you have *custom Task types*, make sure to update the methods to reflect the following changes:

- `page_locked_for_user()` is now `locked_for_user()`. Using `page_locked_for_user()` is deprecated and will be removed in a future release.
- The `page` parameter in `user_can_access_editor()`, `locked_for_user()`, `user_can_lock()`, `user_can_unlock()`, `get_actions()`, has been renamed to `obj`.

Changes to WorkflowState and TaskState models

To accommodate workflows support for snippets, the `WorkflowState.page` foreign key has been replaced with a `GenericForeignKey` as `WorkflowState.content_object`. The generic foreign key is defined using a combination of the new `WorkflowState.base_content_type` and `WorkflowState.object_id` fields.

The `TaskState.page_revision` foreign key has been renamed to `TaskState.revision`.

wagtail.admin.forms.search.SearchForm validation logic

The `wagtail.admin.forms.search.SearchForm` class (which is internal and undocumented, but may be in use by applications that extend the Wagtail admin) no longer treats an empty search field as invalid. Any code that checks `form.is_valid` to determine whether or not to apply a `search()` filter to a queryset should now explicitly check that `form.cleaned_data["q"]` is non-empty.

1.11.48 Wagtail 4.1.9 release notes

October 19, 2023

- *What's new*

What's new

CVE-2023-45809: Disclosure of user names via admin bulk action views

This release addresses an information disclosure vulnerability in the Wagtail admin interface. A user with a limited-permission editor account for the Wagtail admin can make a direct URL request to the admin view that handles bulk actions on user accounts. While authentication rules prevent the user from making any changes, the error message discloses the display names of user accounts, and by modifying URL parameters, the user can retrieve the display name for any user. The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin.

Many thanks to quyenheu for reporting this issue. For further details, please see the [CVE-2023-45809 security advisory](#).

1.11.49 Wagtail 4.1.8 release notes

September 28, 2023

- [What's new](#)

What's new

Maintenance

- Additionally update Pillow dependency to allow use of versions with security fixes (Dan Braghis)

1.11.50 Wagtail 4.1.7 release notes

September 27, 2023

- [What's new](#)

What's new

Maintenance

- Relax Willow dependency to allow use of current Pillow versions with security fixes (Dan Braghis)

1.11.51 Wagtail 4.1.6 release notes

May 25, 2023

- [What's new](#)

What's new

Bug fixes

- Rectify previous fix for TableBlock becoming uneditable after save (Sage Abdullah)
- Ensure that copying page correctly picks up the latest revision (Matt Westcott)
- Adjust collection field alignment in multi-upload forms (LB (Ben) Johnston)
- Prevent lowercase conversions of IndexView column headers (Virag Jain)

Documentation

- Update documentation for `log_action` parameter on `RevisionMixin.save_revision` (Christer Jensen)

1.11.52 Wagtail 4.1.5 release notes

May 2, 2023

- *What's new*

What's new

Bug fixes

- Prevent `TableBlock` from becoming uneditable after save (Sage Abdullah)

1.11.53 Wagtail 4.1.4 release notes

April 3, 2023

- *What's new*

What's new

CVE-2023-28836: Stored XSS attack via ModelAdmin views

This release addresses a stored cross-site scripting (XSS) vulnerability on `ModelAdmin` views within the Wagtail admin interface. A user with a limited-permission editor account for the Wagtail admin could potentially craft pages and documents 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, and only affects sites with `ModelAdmin` enabled.

Many thanks to Thibaud Colas for reporting this issue. For further details, please see the [CVE-2023-28836 security advisory](#).

CVE-2023-28837: Denial-of-service via memory exhaustion when uploading large files

This release addresses a memory exhaustion bug in Wagtail's handling of uploaded images and documents. For both images and documents, files are loaded into memory during upload for additional processing. A user with access to upload images or documents through the Wagtail admin interface could upload a file so large that it results in a crash or denial of service.

The vulnerability is not exploitable by an ordinary site visitor without access to the Wagtail admin. It can only be exploited by admin users with permission to upload images or documents.

Many thanks to Jake Howard for reporting this issue. For further details, please see the [CVE-2023-28837 security advisory](#).

Bug fixes

- Fix radio and checkbox elements shrinking when using a long label (Sage Abdullah)
- Fix select elements expanding beyond their container when using a long option label (Sage Abdullah)
- Fix timezone handling of `TemplateResponses` for users with a custom timezone (Stefan Hammer, Sage Abdullah)
- Ensure `TableBlock` initialization correctly runs after load and its width is aligned with the parent panel (Dan Braghis)
- Ensure that the JavaScript media files are loaded by default in Snippet index listings for date fields (Sage Abdullah)
- Fix server-side caching of the icons sprite (Thibaud Colas)
- Always show Add buttons, guide lines, Move up/down, Duplicate, Delete; in StreamField and Inline Panel (Thibaud Colas)
- Ensure datetimepicker widget overlay shows over modals & drop-downs (LB (Ben) Johnston)

Maintenance

- Render large image renditions to disk (Jake Howard)

1.11.54 Wagtail 4.1.3 release notes

March 13, 2023

- *What's new*

What's new

Bug fixes

- Add right-to-left (RTL) support for the following form components: Switch, Minimap, live preview (Thibaud Colas)
- Improve right-to-left (RTL) positioning for the following components: Page explorer, Sidebar sub-menu, rich text tooltips, rich text toolbar trigger, editor section headers (Thibaud Colas)
- Ensure links within help blocks meet color contrast guidelines for accessibility (Theresa Okoro)
- Support creating `StructValue` copies (Tidiane Dia)
- Fix “Edit this page” missing from userbar (Satvik Vashisht)
- Prevent audit log report from failing on missing models (Andy Chosak)
- Add missing log information for `wagtail.schedule.cancel` (Stefan Hammer)
- Fix timezone activation leaking into subsequent requests in `require_admin_access()` (Stefan Hammer)
- Prevent matches from unrelated models from leaking into SQLite FTS searches (Matt Westcott)
- Update Algolia DocSearch to use new application and correct versioning setup (Thibaud Colas)

Documentation

- Docs: Clarify `ClusterableModel` requirements for using relations with `RevisionMixin`-enabled models (Sage Abdullah)

1.11.55 Wagtail 4.1.2 release notes

February 6, 2023

- *What's new*

What's new

Bug fixes

- Make “Cancel scheduled publish” button correctly redirect back to the edit view (Sage Abdullah)
- Prevent crash when reverting revisions on a snippet with `PreviewableMixin` applied (Sage Abdullah)
- Use consistent heading styles on top-level fields in the page editor (Sage Abdullah)
- Allow button labels to wrap onto two lines in dropdown buttons (Coen van der Kamp)
- Move DateField, DateTimeField, TimeField comment buttons to be right next to the fields (Theresa Okoro)
- Support text resizing in workflow steps cards (Ivy Jeptoo)
- Use the correct padding for autocomplete block picker (Umar Farouk Yunusa)
- Fix horizontal positioning of rich text inline toolbar (Thibaud Colas)

- Close the userbar when clicking its toggle (Albina Starykova)
- Do not show bulk actions checkbox in page type usage view (Sage Abdullah)
- Prevent account name from overflowing the sidebar (Aman Pandey)
- Ensure edit form is displayed as unlocked immediately after canceling a workflow (Sage Abdullah)
- Prevent `latest_revision` pointer from being copied over when copying translatable snippets for translation (Sage Abdullah)

Documentation

- Document potential data loss for `BaseLogEntry` migration in 3.0 (Sage Abdullah)
- Add documentation for the reference index mechanism (Daniel Kirkham)

1.11.56 Wagtail 4.1.1 release notes

November 11, 2022

- *What's new*

What's new

Bug fixes

- Fix issue where lock/unlock buttons would not work on the Dashboard (home) page or the page index listing via the status sidebar (Stefan Hammer)
- Fix disabled style on StreamField add button (Matt Westcott)
- Ensure models are fully loaded before registering snippets, to avoid circular import issues (Matt Westcott)
- Prevent fields without a `verbose_name` property from breaking usage report views (Matt Westcott)
- Exclude tags from the reference index (Matt Westcott)
- Fix errors in handling generic foreign keys when populating the reference index (Matt Westcott)
- Prevent error in handling null ParentalKeys when populating the reference index (Matt Westcott)
- Make sure minimap error indicators follow the minimap scrolling (Thibaud Colas)
- Ensure background HTTP request to clear stale preview data correctly respects the `CSRF_HEADER_NAME` setting (Sage Abdullah)
- Prevent error on aging pages report when “Last published by” user has been deleted (Joshua Munn)

1.11.57 Wagtail 4.1 (LTS) release notes

November 1, 2022

- [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

“What’s New” dashboard banner and “Help” menu

To help with onboarding new users, Wagtail now displays a banner on the dashboard, pointing users to our Editor Guide. The sidebar also contains a new “Help” menu item with a prominent indicator to call attention to the new content: a “What’s new” page showcasing new features, and a link to the Editor Guide.

Users can dismiss the new banner and the sidebar items’ indicators by interacting with the corresponding UI element. We store the state in the user’s profile so we only call attention to the content once.

To turn off the new banner, set `WAGTAIL_ENABLE_WHATS_NEW_BANNER` to `False` in your settings. The new menu items can be removed and customized with the following hooks:

- `register_help_menu_item` – to add new items to the “Help” menu
- `construct_help_menu` – to change or remove existing items from the “Help” menu
- `construct_main_menu` – to remove the new “Help” menu altogether

Page editor minimap

When navigating long page editing interfaces, it can be tedious to pinpoint a specific section, or find all validation errors. To help with those tasks, the page editor now has a new “minimap” side panel, with a table of contents of all sections on the form. Clicking a section’s label scrolls straight to this part of the page, and an indicator bar represents which sections are currently visible on the page. When validation errors are present, each section shows a count of the number of errors. This feature was implemented by Thibaud Colas based on a prototype by LB (Ben) Johnston.

New UI for scheduled publishing

Scheduled publishing settings can now be found within the Status side panel of the page editing view. This change aims to improve clarity over who can set schedules, and when they take effect. This feature was developed by Sage Abdullah.

Customizable snippet admin views

The `register_snippet` function now accepts a `SnippetViewSet` class, allowing various aspects of the snippet's admin views to be customized. See [Customizing admin views for snippets](#). This feature was developed by Sage Abdullah.

Scheduled publishing for snippets

Snippet models that inherit from `DraftStateMixin` can now be assigned go-live and expiry dates. This feature was developed by Sage Abdullah.

Object usage reporting

Images, documents and snippets now provide a usage report, listing the places where references to those objects appear. This report is powered by a new `ReferenceIndex` model which records cross-references between objects whenever those objects are saved; this allows it to work more efficiently than the old report available through the `WAGTAIL_USAGE_COUNT_ENABLED` setting, as well as handling references within StreamField and rich text fields.

Note that on first upgrading to Wagtail 4.1, you will need to run the `rebuild_references_index` management command to populate the references table and ensure that reference counts are displayed accurately. By default, references are tracked for all models in the project, including ones not managed through Wagtail - to disable this for specific models, see [Manage the reference index](#).

This feature was developed by Karl Hobley and Matt Westcott.

Documentation improvements

There are multiple improvements to the documentation theme this release, here are some highlights.

- Code snippets now have a quick copy to clipboard button (Mohammad Areeb)
- Improve the dark mode theme adoption, avoid flashing the wrong theme on first load, reduce the need for scrollbars in page TOC, link underline fixes in Safari (LB (Ben) Johnston, Kartik Kankurte)
- Better accessibility support with a skip to content link (LB (Ben) Johnston)

Other features

- Formalised support for Python 3.11 (Matt Westcott)
- 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 behavior 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))
- Use `search` type input in documentation search (LB (Ben) Johnston)
- Render `help_text` when set on `FieldPanel`, `MultiFieldPanel`, `FieldRowPanel`, and other panel APIs where it previously worked without official support (Matt Westcott)
- Consolidate usage of Excel libraries to a single library `openpyxl`, removing usage of `XlsxWriter`, `tablib`, `xlrd` and `xlwt` (Jaap Roes)
- Adopt generic class based views for the create User create view, User edit view, user delete view and Users index listing / search results (Mehrdad Moradizadeh)
- Add `button-secondary bicolor` variants to the pattern library and styleguide (Adinapunyo Banerjee)
- Add better support for non-integer / non-`id` primary keys into Wagtail’s generic views, including for custom Snippets and User models (Mehrdad Moradizadeh)
- Upgrade jQuery UI to version 1.13.2 (LB (Ben) Johnston)
- Update pattern library background & text examples (Albina Starykova)
- Switch StreamField blocks to use a `<section>` element so screen reader users can bypass them more easily (Thibaud Colas)
- Add anchor links to StreamField blocks so users can navigate straight to a given block (Thibaud Colas)
- Support “Ctrl + f” in-page search within collapsed StreamField blocks (Thibaud Colas)
- Remember the last opened side panel in the page editor, activating it on page load (Sage Abdullah)

- Ensure that the `update_index` command can run without console output if called with `--verbosity 0` (Ben Sturmels, Oliver Parker)
- Improve side panels' resizing in page editor and listings (Steven Steinwand)
- Adjust breadcrumb text alignment and size in page listings & page editor (Steven Steinwand)
- Improvements to getting started tutorial aimed at developers who are very new to Python and have no Django experience (Damilola Oladele)
- The `image_url` template tag, when using the `serve` view to redirect rather than serve directly, will now use temporary redirects with a cache header instead of permanent redirects (Jake Howard)
- Add new test assertions to `WagtailPageTestCase` - `assertPageIsRoutable`, `assertPageIsRenderable`, `assertPageIsEditable`, `assertPageIsPreviewable` (Andy Babic)
- Add documentation to the performance section about how to better create image URLs when not used directly on the page (Jake Howard)
- Add ability to provide a required permission to `PanelGroup`, used by `TabbedInterface`, `ObjectList`, `FieldRowPanel` and `MultiFieldPanel` (Oliver Parker)
- Update documentation screenshots of the admin interface to align with changes in this release (Thibaud Colas)

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 color 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)
- Add back rendering of `help_text` for `InlinePanel` (Matt Westcott)
- Ensure `for_user` argument is passed to the form class when previewing pages (Matt Westcott)
- Ensure the capitalization of the `timesince_simple` tag is consistently added in the template based on usage in context (Stefan Hammer)
- Add missing translation usage for the `timesince_last_update` and ensure the translated labels can be easier to work with in Transifex (Stefan Hammer)

- Add additional checks for duplicate form field `clean_name` values in the Form Builder validation and increase performance of checks (Dan Bentley)
- Use correct color for labels of radio and checkbox fields (Steven Steinwand)
- Adjust spacing of fields' error messages and position in tables (Steven Steinwand)
- Update dead or redirected links throughout the documentation (LB (Ben) Johnston)
- Use different icons for workflow timeline component, so the steps can be distinguished with other means than color (Sam Moran)
- Use the correct custom font for the Wagtail userbar (Umar Farouk Yunusa)
- StreamField blocks are now collapsible with the keyboard (Thibaud Colas)
- StreamField block headings now have a label for screen reader users (Thibaud Colas)
- Display the “*” required field indicator for StreamField blocks (Thibaud Colas)
- Resolve inconsistency in action button positions in `InlinePanel` (Thibaud Colas)
- Use `h3` elements with a counter in `InlinePanel` so screen reader users can navigate by heading (Thibaud Colas)
- Ensure that buttons on custom chooser widgets are correctly shown on hover (Thibaud Colas)
- Add missing asterisk to title field placeholder (Seremba Patrick, Stefan Hammer)
- Avoid creating an extra rich text block when inserting a new block at the end of the content (Matt Westcott)
- Removed the extra dot in the Wagtail version shown within the admin settings menu item (Loveth Omokoro)
- Fully remove the obsolete `wagtailsearch_editorspick` table that prevents flushing the database (Matt Westcott)
- Update latest version message on Dashboard to accept dev build version format used on nightly builds (Sam Moran)
- Ensure `ChooserBlock.extract_references` uses the model class, not the model string (Alex Tomkins)
- Regression in field width for authentication pages (log in / password reset) (Chisom Okeoma)
- Ensure the new minimap correctly pluralizes error counts for `aria-labels` (Matt Westcott)

Upgrade considerations

`rebuild_references_index` management command

After upgrading, you will need to run `./manage.py rebuild_references_index` to populate the references table and ensure that usage counts for images, documents and snippets are displayed accurately. By default, references are tracked for all models in the project, including ones not managed through Wagtail - to disable this for specific models, see [Manage the reference index](#).

Recommend WagtailPageTestCase in place of WagtailPageTests

- WagtailPageTestCase is the base testing class and is now recommended over using WagtailPageTests [Testing your Wagtail site](#).
- WagtailPageTests will continue to work and does log in the user on test `setUp` but may be deprecated in the future.

```
# class MyPageTests(WagtailPageTests): # old
class MyPageTests(WagtailPageTestCase): # new
    def setUp(self):
        # WagtailPageTestCase will not log in during setUp - so add if needed
        super().setUp()
        self.login()

    def test_can_create_a_page(self):
        # ...
```

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`

Dropped support for importing .xls Spreadsheet files into Redirects

- .xls legacy Microsoft Excel 97-2003 spreadsheets will no longer be supported for importing into the contrib Redirects listing.
- .xlsx, .csv, .tsv formats are still supported.

1.11.58 Wagtail 4.0.4 release notes

October 18, 2022

- *What's new*

What's new

- Render `help_text` when set on `FieldPanel`, `MultiFieldPanel`, `FieldRowPanel`, and other panel APIs where it previously worked without official support (Matt Westcott)
- Update special-purpose `FieldPanel` deprecation message to add clarity for developers (Matt Westcott)

Bug fixes

- Add back rendering of `help_text` for `InlinePanel` (Matt Westcott)
- Ensure that `AbstractForm` & `AbstractEmailForm` page models correctly pass the form to the preview context (Dan Bentley)
- Use the correct custom font for the Wagtail userbar (Umar Farouk Yunusa)
- Ensure that buttons on custom chooser widgets are correctly shown on hover (Thibaud Colas)

1.11.59 Wagtail 4.0.2 release notes

September 23, 2022

- *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 initializing 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)
- Remove over-eager caching on ModelAdmin permission checks (Matt Westcott, Stefan Hammer)

1.11.60 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-Mugglestone)
- 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)

1.11.61 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. When upgrading, please note that the `django-taggit` library also needs to be updated to 3.0.0 or above.

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 several 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 color themes

In Wagtail 2.12, we introduced theming support for Wagtail's primary brand color. This has now been extended to almost all of Wagtail's color palette. View our [Custom user interface colors](#) documentation for more information, an overview of Wagtail's customizable color palette, and a live demo of the supported customizations. 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 color 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 an 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 `explorerBreadcrumb` 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 colors
- Add support for RTL layouts and better support for small devices
- Add CSS support for more than three summary items if added via customizations

- **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 colors 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 the organization of the settings reference page with logical grouping and better internal linking (Akash Kumar Sen)
- Improve the accessibility of the documentation with higher contrast colors, consistent focus outline, better keyboard only navigation through the sidebar (LB (Ben) Johnston, Vince Salvino)
- Better sidebar scrolling behavior, 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 favor 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 customize 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 optimization 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 22 (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 customizations (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 unrecognized 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 behavior 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 has 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 colors, and links have a darker color 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 Draftail 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 saved (Jacob Topp-Muggleton)
- 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 utilizes 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 `RemovedInWagtail150Warning`.

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 sub-menus 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 initializers replaced with classes

The internal JavaScript functions `createPageChooser`, `createSnippetChooser`, `createDocumentChooser` and `createImageChooser` used for initializing 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`, `wagtaildocs` 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`
- `wagtaildocs:chooser` is now `wagtaildocs_chooser:choose`
- `wagtaildocs:chooser_results` is now `wagtaildocs_chooser:choose_results`
- `wagtaildocs:document_chosen` is now `wagtaildocs_chooser:chosen`
- `wagtaildocs:chooser_upload` is now `wagtaildocs_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 `wagtailsnippets_{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.

Customizing 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 color 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 customizable with `--w-color-primary`, and the teal is customizable as `--w-color-secondary`. See [Custom user interface colors](#) for an overview of all customizable colors. 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 color 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 customizations

As part of the page editor redesign, we have removed support for the `classname="full"` customization to panels. Existing `title` and `collapsed` customizations 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 behavior 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()
```

1.11.62 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)

1.11.63 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 `update_module_paths` 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)

1.11.64 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)

1.11.65 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 customizations*

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 send-file_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 behavior (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 Espinoza)
- 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 reorganized, 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 customizations

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_panel.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 favor 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 customizations 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 initialization 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 behavior 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.

If you have models that are subclasses of `BaseLogEntry` in your project, be careful when generating new migrations for these models. As the field is changed and renamed at the same time, Django's `makemigrations` command will generate `RemoveField` and `AddField` operations instead of `AlterField` and `RenameField`. To avoid data loss, make sure to adjust the migrations accordingly. For example with a model named `MyCustomLogEntry`, change the following operations:

```
operations = [
    migrations.RemoveField(
        model_name='mycustomlogentry',
        name='data_json',
    ),
    migrations.AddField(
        model_name='mycustomlogentry',
        name='data',
        field=models.JSONField(blank=True, default=dict),
    ),
]
```

to the following operations:

```
operations = [
    migrations.AlterField(
        model_name="mycustomlogentry",
        name="data_json",
        field=models.JSONField(blank=True, default=dict),
    ),
    migrations.RenameField(
        model_name="mycustomlogentry",
        old_name="data_json",
        new_name="data",
    ),
]
```

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 customizations 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 serializable 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.

1.11.66 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)

1.11.67 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.

1.11.68 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 customize admin UI colors via CSS (LB (Ben Johnston))

1.11.69 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 disable 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 ≥ 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 color 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 customized 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 customizations 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 customizing arbitrary HTML attributes.
- `MenuItem` can no longer be sub-classed to customize 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.

1.11.70 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)

1.11.71 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.

1.11.72 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) (Thor 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)

1.11.73 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)

1.11.74 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 initialization 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.

1.11.75 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)

1.11.76 Wagtail 2.15 (LTS) 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 organized. 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 color (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” color (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 customize 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 behavior 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 `nl_NL` are moved to the `nl.po` files. `nl_NL` translation files are deleted. Projects that use `LANGUAGE_CODE = 'nl-nl'` will automatically fallback to `nl`. (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 behavior can be customized (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 favor 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 organization, 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 behavior 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 behavior. 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 behavior 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`.

1.11.77 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

Customizing 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 behavior 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'
```

1.11.78 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)

1.11.79 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 customizing group management views. See [*Customizing 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 customizations 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.

1.11.80 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

Customizing 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 behavior 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'
```

1.11.81 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)

1.11.82 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-Mugglestone)
- 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 color customizations (Matt Westcott)
- Avoid validation error when editing rich text content preceding a comment (Jacob Topp-Mufflestone)

1.11.83 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 implementers who wish to retain the existing behavior 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](#).

1.11.84 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-Mugglestone)
- Fix incorrect link in comment notification HTML email (Matt Westcott)

1.11.85 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 customizability 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-Mugglestone)

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 customizations may need to be reimplemented to work with the new client-side rendering model:

- When customizing 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 customizing 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.

1.11.86 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)

1.11.87 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 implementers 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](#).

1.11.88 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)

1.11.89 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)

1.11.90 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)

1.11.91 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)

1.11.92 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 customization. Read on [Custom user interface colors](#). 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 customize 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.specific_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`.

1.11.93 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)

1.11.94 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 implementers who wish to retain the existing behavior 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](#).

1.11.95 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.

1.11.96 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)

1.11.97 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)

1.11.98 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)

1.11.99 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.

1.11.100 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)

1.11.101 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)

1.11.102 Wagtail 2.11 (LTS) 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 `WAGTAIL_DOCS_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 Poulaïn)
- 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'),
    ]

    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.

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.

1.11.103 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)

1.11.104 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)

1.11.105 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 our How-to: [Configure workflows for moderation](#) 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 customization 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 behavior 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 Poulin)
- 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-Muggleton)
- 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 our How-to: [Configure workflows for moderation](#) 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.

1.11.106 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 (see the docs for `help_text`); 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.

1.11.107 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)

1.11.108 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)

1.11.109 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-Muggleton 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 Boaretto)
- 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-Mugglestone)
- 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 `wagtail.core.middleware.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()`.

1.11.110 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.

1.11.111 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.

1.11.112 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 behavior 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 customization 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 Ascari)
- Dropdown initialization on the search page after AJAX call (Eric Sherman)
- Make sure all modal chooser search results correspond to the latest search by canceling 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 behavior 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 behavior, 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 behavior, 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
```

(continues on next page)

(continued from previous page)

```
admin.site.register(Site)
admin.site.register(Page)
```

1.11.113 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 (see the docs for `help_text`); 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)

1.11.114 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.

1.11.115 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.

1.11.116 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)

1.11.117 Wagtail 2.7 (LTS) release notes

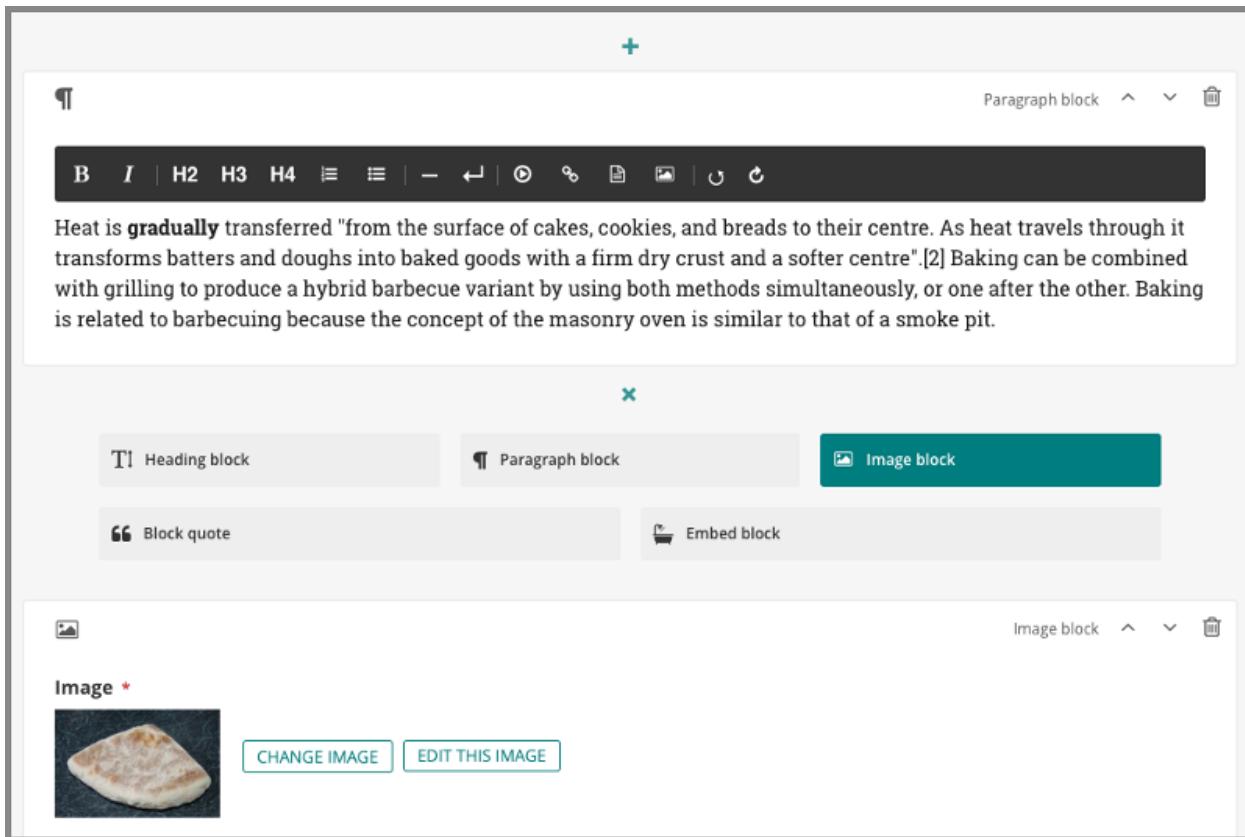
November 6, 2019

- *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-Mugglestone) - thanks to [The Motley Fool](#) for sponsoring this feature
- Added settings to customize 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 color 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 Dodoo)
- 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 behavior 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 behavior by setting `WAGTAILDOCS_SERVE_METHOD` to `'serve_view'`.

Template change for page action menu hooks

When customizing 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 customize the menu item's HTML. As of Wagtail 2.7, the HTML returned from these should *not* include the enclosing `` element.

Any add-on library that uses this feature and needs to preserve backward compatibility with previous Wagtail versions can conditionally reinsert the `` 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('<li>{}</li>', 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>

1.11.118 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)

1.11.119 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)

1.11.120 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)

1.11.121 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 that 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 that should make Wagtail more usable for all users regardless of abilities:

- Increase font-size across the whole admin (Beth Menzies, Katie Locke)
- Improved text color 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 customizations 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 labeled 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 Bansen)
- Move datetime widget initializer 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 customize copy behavior (Karl Hobley)
- Added `Page.with_content_json()`, allowing revision content loading behavior to be customized 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 normalization (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](#).

1.11.122 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)

1.11.123 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)

1.11.124 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 customizing 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 prioritizes more specific rules, to accommodate `<p>` and `
` 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 (frmdstryr)
- 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 canceling 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 favor 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

Several 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 favor 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 behavior.

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.

1.11.125 Wagtail 2.4 release notes

December 19, 2018

- [What's new](#)
- [Upgrade considerations](#)

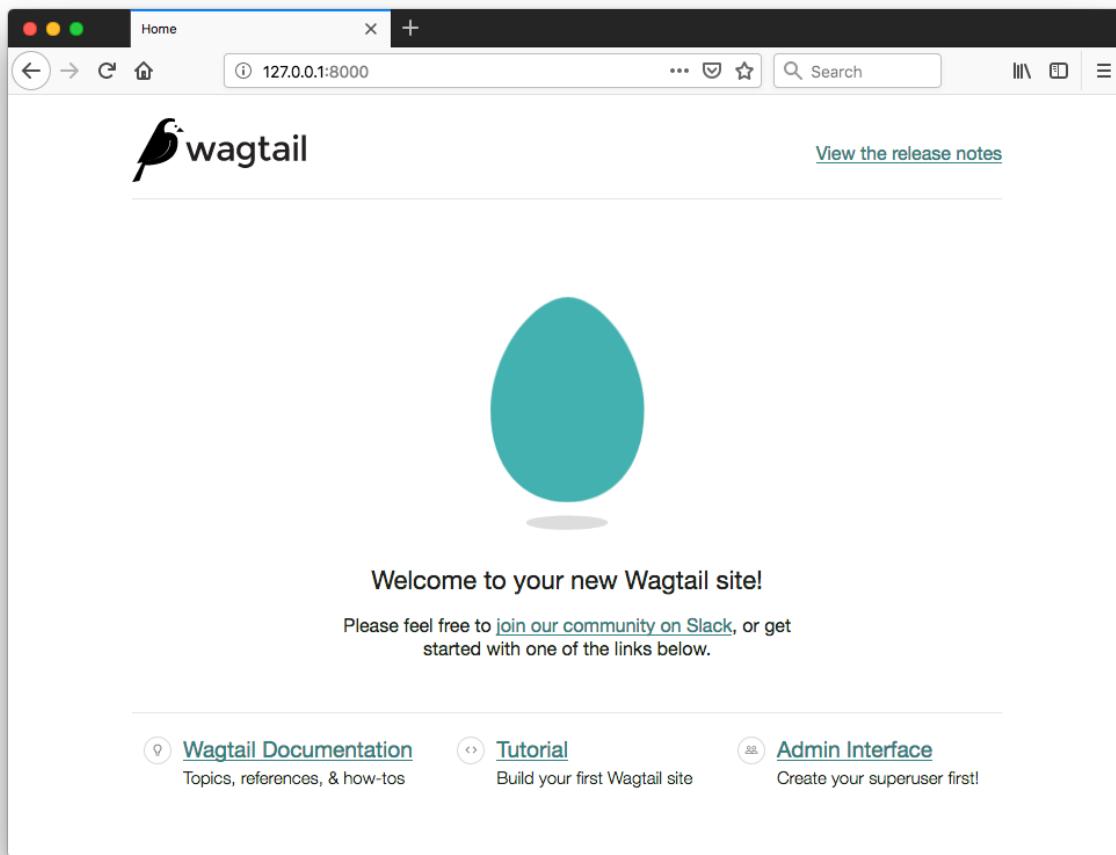
What's new

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 customize 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)) ,
```

1.11.126 Wagtail 2.3 (LTS) 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 color 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 recognized 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 localization in date/time pickers (David Moore, Thibaud Colas)
- Tag input field no longer treats ‘ö’ 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 reorganized

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`.

1.11.127 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)

1.11.128 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)

1.11.129 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 optimization 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.specifc` (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 color 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).

1.11.130 Wagtail 2.1.3 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)

1.11.131 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 BeautifulSoup 4.6.1 as incompatible due to bug in formatting empty elements (Matt Westcott)

1.11.132 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)

1.11.133 Wagtail 2.1 release notes

May 22, 2018

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

What's new

New HelpPanel

A new panel type `HelpPanel` allows you 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 customizing 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 recognized, as per standard Django behavior (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 longer 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 color 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.

1.11.134 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)

1.11.135 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 recognized, as per standard Django behavior (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)

1.11.136 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.

Reorganized modules

The modules that make up Wagtail have been renamed and reorganized, to avoid the repetition in names like `wagtail.wagtailcore.models` (originally an artifact 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 behavior 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 customizations 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))
- Optimised 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 customized. See [Additional methods and properties on StructBlock values](#) (LB (Ben Johnston))
- Added `bgcolor` 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 favor 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 customized (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 recognizes 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 reorganized 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
<code>wagtail.wagtailcore</code>	<code>wagtail.core</code>	
<code>wagtail.wagtailadmin</code>	<code>wagtail.admin</code>	
<code>wagtail.wagtailedocs</code>	<code>wagtail.documents</code>	'documents' no longer abbreviated
<code>wagtail.wagtailembeds</code>	<code>wagtail.embeds</code>	
<code>wagtail.wagtailimages</code>	<code>wagtail.images</code>	
<code>wagtail.wagtailsearch</code>	<code>wagtail.search</code>	
<code>wagtail.wagtailsites</code>	<code>wagtail.sites</code>	
<code>wagtail.wagtailsnippets</code>	<code>wagtail.snippets</code>	
<code>wagtail.wagtailusers</code>	<code>wagtail.users</code>	
<code>wagtail.wagtailforms</code>	<code>wagtail.contrib.forms</code>	Moved into 'contrib'
<code>wagtail.wagtailredirects</code>	<code>wagtail.contrib.redirects</code>	Moved into 'contrib'
<code>wagtail.contrib.wagtailapi</code>	<code>removed</code>	API v1, removed in this release
<code>wagtail.contrib.wagtailfrontendcache</code>	<code>wagtail.contrib.frontend_cache</code>	Underscore added
<code>wagtail.contrib.wagtailroutablepage</code>	<code>wagtail.contrib.routable_page</code>	Underscore added
<code>wagtail.contrib.wagtailsearchpromotions</code>	<code>wagtail.contrib.search_promotions</code>	Underscore added
<code>wagtail.contrib.wagtailsitemaps</code>	<code>wagtail.contrib.sitemaps</code>	
<code>wagtail.contrib.wagtailstyleguide</code>	<code>wagtail.contrib.styleguide</code>	

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 customizations 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 behavior, 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'
    }
}
```

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 customize 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/wagtailemo/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;
}
```

1.11.137 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)

1.11.138 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 BeautifulSoup 4.6.1 as incompatible due to bug in formatting empty elements (Matt Westcott)

1.11.139 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)

1.11.140 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)

1.11.141 Wagtail 1.13 (LTS) 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 Magnússon)
- 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)

1.11.142 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)

1.11.143 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 BeautifulSoup 4.6.1 as incompatible due to bug in formatting empty elements (Matt Westcott)

1.11.144 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)

1.11.145 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)

1.11.146 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)

1.11.147 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)

1.11.148 Wagtail 1.12 (LTS) 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 (dwasyl)
- 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 initialization (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 behavior 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 favor 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 customizing 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 customize 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.

1.11.149 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)

1.11.150 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 Bartnig (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.models.Page.get_url_parts()`, `wagtail.models.Page.get_url()`, `wagtail.models.Page.get_full_url()`, and `wagtail.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).

1.11.151 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)

1.11.152 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 `wagtailedocs` 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 localization 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 `lsafe` 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 `djongo-medusa`

Django-medusa is no longer maintained, and is incompatible with Django 1.8 and above. An alternative module based on the `djongo-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)
```

(continues on next page)

(continued from previous page)

```
# 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 behavior, 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 behavior, you will have to rewrite it to take this into account.

1.11.153 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)

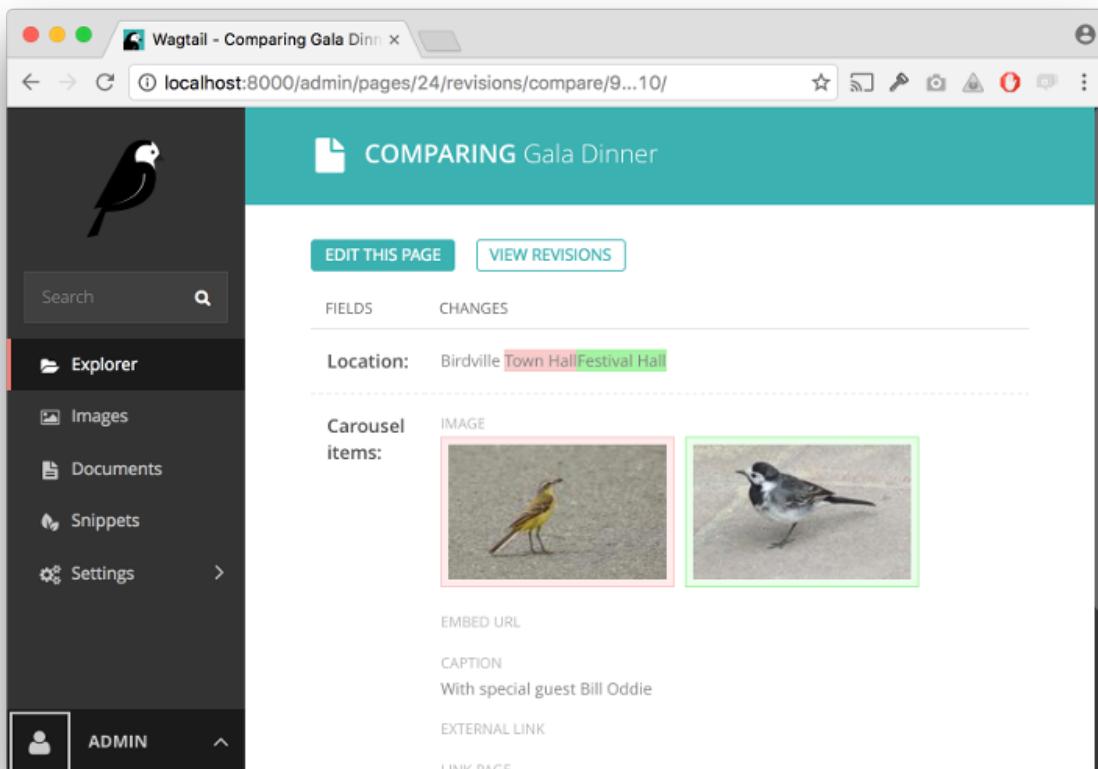
1.11.154 Wagtail 1.9 release notes

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 [Authors](#) 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 customization 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 customized, 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 customized (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 customizations. See [Form builder customization](#) (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.

1.11.155 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)

1.11.156 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)

1.11.157 Wagtail 1.8 (LTS) release notes

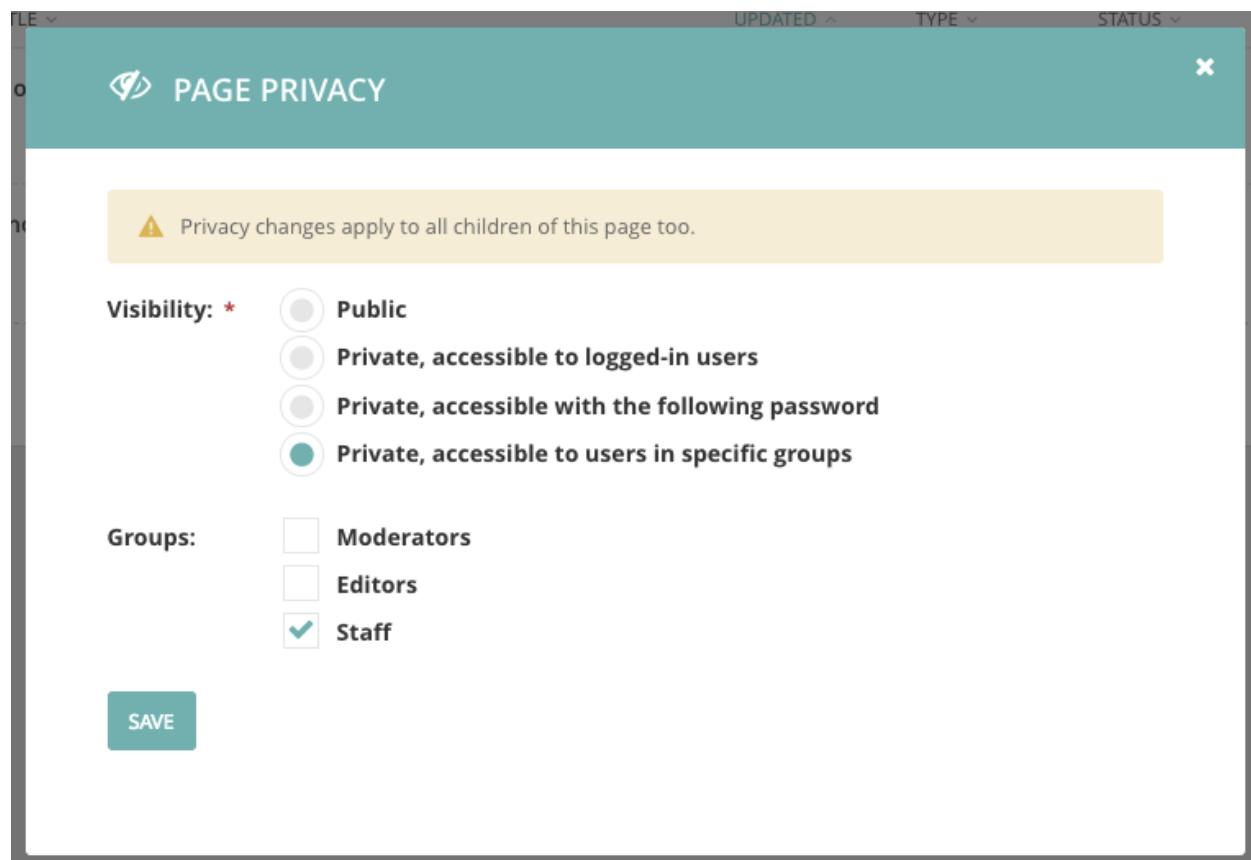
December 15, 2016

- [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; 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 behavior for them.

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. (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` customization blocks to the login page template - see [Customizing 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.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 Marinozzi)
- Extra options for the Elasticsearch constructor should be now defined with the new key `OPTIONS` of the `WAGTAILSEARCH_BACKENDS` setting (PyMan Claudio Marinozzi)

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 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 behavior 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"""
    GroupPagePermission = apps.get_model('wagtailcore', 'GroupPagePermission')

```

(continues on next page)

(continued from previous page)

```
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 behavior is still supported but deprecated.

For example, the following configuration changes the connection class that the Elasticsearch connector uses:

```
from elasticsearch import RequestsHttpConnection
```

(continues on next page)

(continued from previous page)

```
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,
        }
    }
}
```

1.11.158 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 colors 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 optimization 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 localized (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 Lorenzetti)
- 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 before 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 %}
```

1.11.159 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)

1.11.160 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)

1.11.161 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()
```

1.11.162 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 behavior 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 Nateghi.

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 customizing 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 Pastefanos)
- 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 overnormalization
- “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 (Gagaro)
- 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, 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):
        ...
```

1.11.163 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)

1.11.164 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)

1.11.165 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)

1.11.166 Wagtail 1.5 release notes

May 1, 2016

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

What's new

Reorganized page explorer actions

The screenshot shows the Wagtail page explorer interface. At the top, there's a teal header bar with the title "People". Below it, a row of buttons: EDIT, LIVE, + ADD CHILD PAGE, and MORE. To the right of the buttons, the text "1 minute ago" is displayed. The main area lists two items: "James Joyce" and "David Mitchell". For "James Joyce", there are buttons for EDIT, DRAFT, LIVE, ADD CHILD PAGE, and MORE. The "MORE" button is highlighted with a dashed border. A context menu is open over this button, listing the following options: Move, Copy, Delete, Unpublish, and Revisions. The timestamp "2 weeks, 2 days ago" is shown to the right of "James Joyce". For "David Mitchell", there are similar buttons: EDIT, DRAFT, LIVE, ADD CHILD PAGE, and MORE. The timestamp "1." is shown to the right of "David Mitchell".

The action buttons on the page explorer have been reorganized 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 dark sidebar with navigation links: Search, Explorer, Images, Documents, Snippets, Countries (which is highlighted), and Settings. The main area has a teal header with the title 'COUNTRIES' and a 'ADD COUNTRY' button. Below is a table listing three countries: China, India, and Indonesia, with columns for NAME, CONTINENT, and POPULATION. To the right of the table is a 'FILTER' section titled 'By continent' containing buttons for ALL, AFRICA, ANTARCTICA, ASIA (which is selected and highlighted in teal), AUSTRALIA, EUROPE, NORTH AMERICA, and SOUTH AMERICA. The URL in the browser bar is `localhost:8000/admin/demo/country/?continent__exact=Asia&o=-2`.

NAME	CONTINENT	POPULATION
China	Asia	1376570000
India	Asia	1289020000
Indonesia	Asia	258705000

Page 1 of 1.

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.

Body

The recipe's step-by-step instructions and any other relevant information.

 **Table block ***    

Table headers

Display the first row AND first column as headers 

Which cells should be displayed as headers?

Table caption

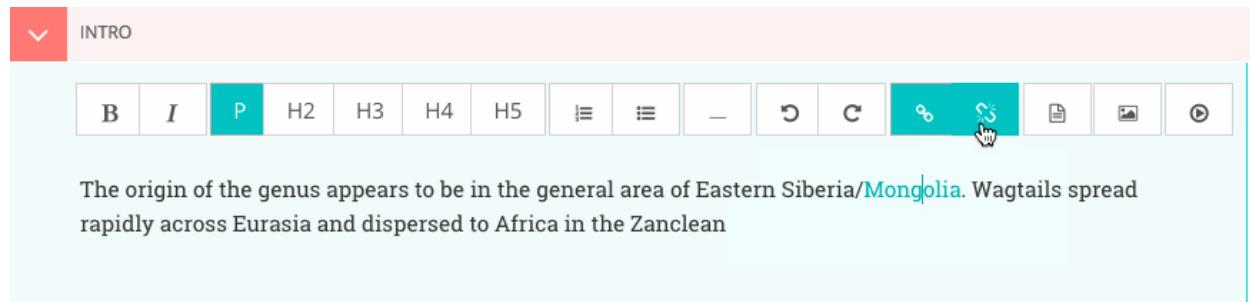
A heading that identifies the overall topic of the table, and is useful for screen reader users.

Expected yield

Buns	Prep time	Cooking time	Instructions
12	30min	1.5h	All quantities as-is
18	40min	2h	1.5x all amounts
24	45min	Depending on oven size	2x all amounts

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 customize 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 + (  
        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 behavior 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 to work on projects created under Wagtail 1.5.

1.11.167 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)

1.11.168 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 canceling 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)

1.11.169 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)

1.11.170 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)

1.11.171 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)

1.11.172 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)

1.11.173 Wagtail 1.4 (LTS) release notes

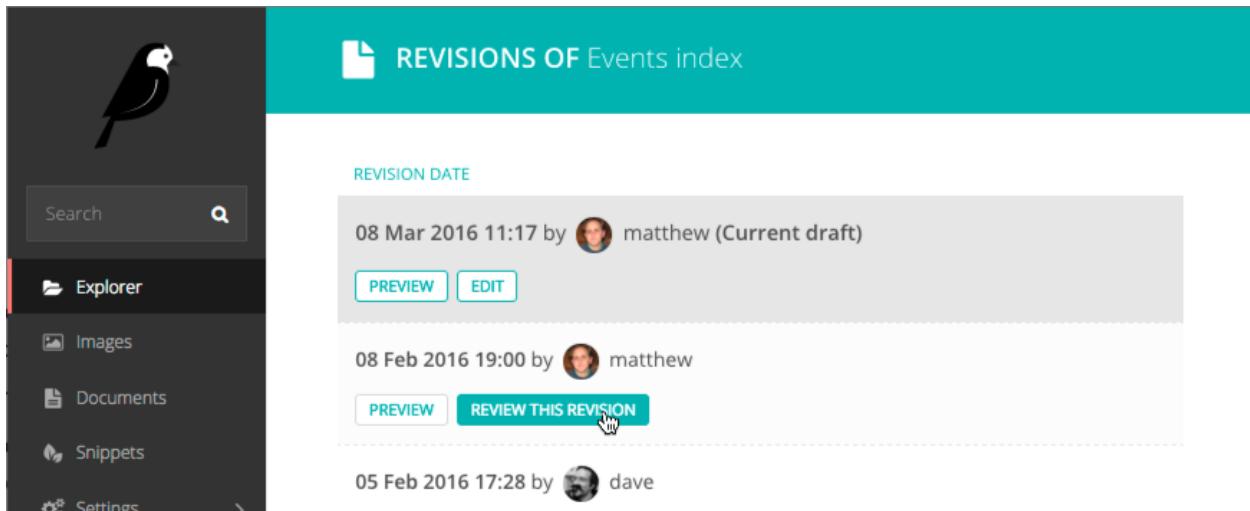
March 16, 2016

- *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

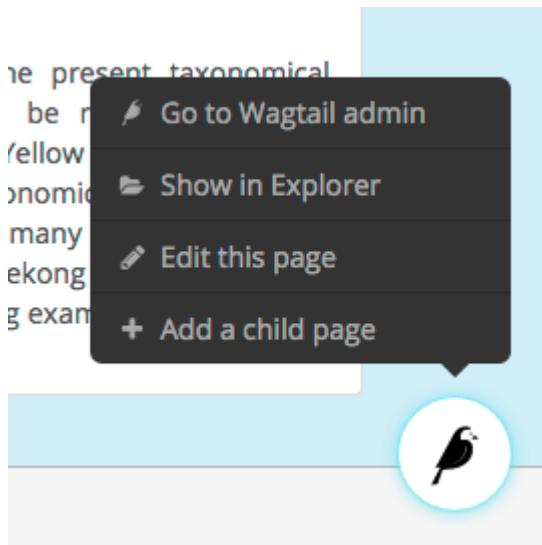


From the page editing interface, editors can now access a list of previous revisions of the page, and preview or roll back to any earlier revision.

Collections for image / document organization

Images and documents can now be organized 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 made 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 [Customizing 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 customizing 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 customizing 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` (alexpiolt11)
- 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.

1.11.174 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](#))

1.11.175 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 author's 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 Voskvtsov)
- 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 customizing 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 behavior 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.)

1.11.176 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 for the page to be valid.

See: [InlinePanel](#)

`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 no redirect 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, customizable 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 standardizes 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 an empty page tree now takes you to the root level, rather than doing nothing
- Added animation and fixed display issues when focusing a rich text field (Alex Gleason)
- Added a system check to warn if Pillow is compiled without JPEG / PNG support
- Page chooser now prevents users from selecting the root node where this would be invalid
- New translations for Dutch (Netherlands), Georgian, Swedish and Turkish (Turkey)

Bug fixes

- Page slugs are no longer auto-updated from the page title if the page is already published
- Deleting a page permission from the groups admin UI does not immediately submit the form
- Wagtail userbar is shown on pages that do not pass a `page` variable to the template (e.g. because they override the `serve` method)
- `request.site` now set correctly on page preview when the page is not in the default site
- Project template no longer raises a deprecation warning (Maximilian Stauss)
- `PageManager.sibling_of(page)` and `PageManager.not_sibling_of(page)` now default to inclusive (i.e. page is considered a sibling of itself), for consistency with other sibling methods
- The “view live” button displayed after publishing a page now correctly reflects any changes made to the page slug (Ryan Pineo)
- API endpoints now accept and ignore the `_query` parameter used by jQuery for cache-busting
- Page slugs are no longer cut off when Unicode characters are expanded into multiple characters (Sævar Öfjörð Magnússon)
- Searching a specific page model while filtering it by either ID or tree position no longer raises an error (Ashia Zawaduk)
- Scrolling an over-long explorer menu no longer causes white background to show through (Alex Gleason)
- Removed jitter when hovering over StreamField blocks (Alex Gleason)
- Non-ASCII email addresses no longer throw errors when generating Gravatar URLs (Denis 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+
- `wagtail.models.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 behavior: 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 behavior: 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) behavior, this behavior 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 favor 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 that 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 behavior is still used for pages in `live + draft` state).

1.11.177 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 `wagtail.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 `wagtail.contrib.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

Several 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 `wagtail.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.

1.11.178 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 specialized 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 `register_rich_text_embed_handler` and `register_rich_text_link_handler` for customizing 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 [Customizing 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 JavaScript behaviour glitches when focusing / unfocusing fields
- Added hook `construct_homepage_summary_items` for customizing the site summary panel on the admin homepage
- Renamed the `construct_wagtail_edit_bird` hook to `construct_wagtail_userbar`
- ‘static’ template tags are now used throughout the admin templates, in place of `STATIC_URL`

Docs

- Support for `django-sendfile` added
- Documents now served with correct mime-type
- Support for `If-Modified-Since` HTTP header

Search

- Search view accepts “page” GET parameter in line with pagination
- Added `AUTO_UPDATE` flag to search backend settings to enable/disable automatically updating the search index on model changes

Routable pages

- Added a new decorator-based syntax for RoutablePage, compatible with Django 1.8

Bug fixes

- The `document_served` signal now correctly passes the Document class as `sender` and the document as `instance`
- Image edit page no longer throws `OSError` when the original image is missing
- Collapsible blocks stay open on any form error
- Document upload modal no longer switches tabs on form errors
- `with_metaclass` is now imported from Django’s bundled copy of the `six` library, to avoid errors on Mac OS X from an outdated system copy of the library being imported

Upgrade considerations

Support for older Django/Python/Elasticsearch versions dropped

This release drops support for Django 1.6, Python 2.6/3.2 and Elasticsearch 0.90.x. Please make sure these are updated before upgrading.

If you are upgrading from Elasticsearch 0.90.x, you may also need to update the `elasticsearch` pip package to a version greater than `1.0` as well.

Wagtail version upgrade notifications are enabled by default

Starting from Wagtail 1.0, the admin dashboard will (for admin users only) perform a check to see if newer releases are available. This also provides the Wagtail team with the hostname of your Wagtail site. If you’d rather not receive update notifications, or if you’d like your site to remain unknown, you can disable it by adding this line to your settings file:

```
WAGTAIL_ENABLE_UPDATE_CHECK = False
```

InlinePanel definitions no longer need to specify the base model

In previous versions of Wagtail, inline child blocks on a page or snippet were defined using a declaration like:

```
InlinePanel(HomePage, 'carousel_items', label="Carousel items")
```

It is no longer necessary to pass the base model as a parameter, so this declaration should be changed to:

```
InlinePanel('carousel_items', label="Carousel items")
```

The old format is now deprecated; all existing `InlinePanel` declarations should be updated to the new format.

Custom image models should now set the `admin_form_fields` attribute

Django 1.8 now requires that all the fields in a `ModelForm` must be defined in its `Meta.fields` attribute.

As Wagtail uses Django's `ModelForm` for creating image model forms, we've added a new attribute called `admin_form_fields` that should be set to a tuple of field names on the image model.

See [Custom image 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 `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 before 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 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 customize the behavior of the page / snippet editing forms. If you have made use of this facility, you will need to update your custom `EditHandlers`, as this mechanism has been refactored (to allow `EditHandler` classes to keep a persistent reference to their corresponding model). If you have only used Wagtail’s built-in panel types (`FieldPanel`, `InlinePanel`, `PageChooserPanel` and so on), you are unaffected by this change.

Previously, functions like `FieldPanel` acted as ‘factory’ functions, where a call such as `FieldPanel('title')` constructed and returned an `EditHandler` subclass tailored to work on a ‘title’ field. These functions now return an object with a `bind_to_model` method instead; the `EditHandler` subclass can be obtained by calling this with the model class as a parameter. As a guide to updating your custom `EditHandler` code, you may wish to refer to the [relevant change](#) to the Wagtail codebase.

chooser_panel templates are obsolete

If you have added your own custom admin views to the Wagtail admin (e.g. through the `register_admin_urls` hook), you may have used one of the following template includes to incorporate a chooser element for pages, documents, images or snippets into your forms:

- `wagtailadmin/edit_handlers/chooser_panel.html`
- `wagtailadmin/edit_handlers/page_chooser_panel.html`
- `wagtaildocs/edit_handlers/document_chooser_panel.html`
- `wagtailyimages/edit_handlers/image_chooser_panel.html`
- `wagtaillsnippets/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 behavior 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 behavior is risky as it may lead to fields being unintentionally exposed to the user, and so Django has deprecated this, for removal in Django 1.8. Accordingly, if you create your own custom subclass of AbstractImage, you must now provide an admin_form_fields property, listing the fields that should appear on the image creation / editing form - for example:

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

class MyImage(AbstractImage):
    photographer = models.CharField(max_length=255)
    has_legal_approval = models.BooleanField()

    admin_form_fields = Image.admin_form_fields + ['photographer']
```

construct_wagtail_edit_bird hook has been renamed

Previously you could customize the Wagtail userbar using the construct_wagtail_edit_bird hook. The hook has been renamed to construct_wagtail_userbar.

The old hook is now deprecated; all existing construct_wagtail_edit_bird declarations should be updated to the new hook.

IMAGE_COMPRESSION_QUALITY setting has been renamed

The IMAGE_COMPRESSION_QUALITY setting, which determines the quality of saved JPEG images as a value from 1 to 100, has been renamed to WAGTAILIMAGES_JPEG_QUALITY. If you have used this setting, please update your settings file accordingly.

1.11.179 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”

1.11.180 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

1.11.181 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

1.11.182 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 behavior. With this option enabled, deleting orphaned pages is always skipped.

1.11.183 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

1.11.184 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

1.11.185 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 serialization 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 color 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 behavior would be to use the default http/https port of 80/443. This behavior 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.)

1.11.186 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

1.11.187 Wagtail 0.8 (LTS) 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 subpages 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')),
```

1.11.188 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 centered on. Previously the focal point could only be set automatically, through image feature detection.

Groups and Sites administration interfaces

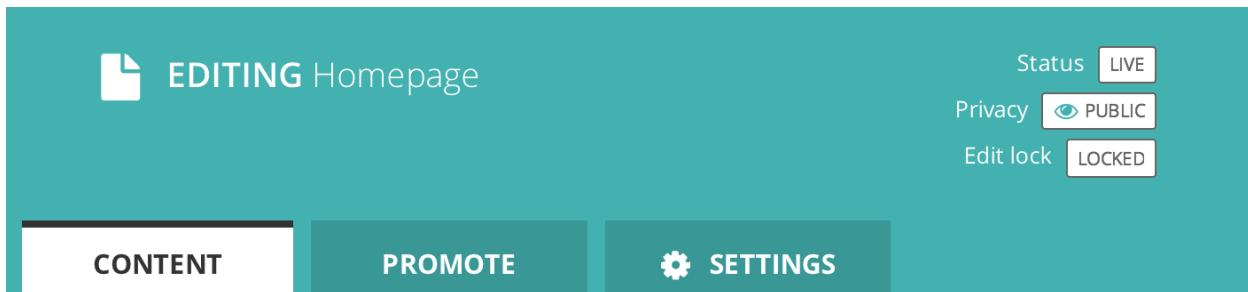
The main navigation menu has been reorganized, placing site configuration options in a ‘Settings’ submenu. This includes two new items, which were previously only available through the Django admin backend: ‘Groups’, for setting up user groups with a specific set of permissions, and ‘Sites’, for managing 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 includes links for Explorer, Images, Documents, and Settings. The main content area has a teal header bar with the title 'EDITING Editors'. A form field labeled 'Name: *' contains the value 'Editors'. Below this is a table titled 'OBJECT PERMISSIONS' with columns for 'NAME', 'ADD', 'CHANGE', and 'DELETE'. The rows show permissions for Document, Image, Group, User, and User profile. The 'Document' row has checked boxes in all three columns. The 'Image' row also has checked boxes in all three columns. The 'Group', 'User', and 'User profile' rows have empty boxes in all three columns. Below this is a section titled 'OTHER PERMISSIONS' with a table showing a single row for 'Can access Wagtail admin' with a checked box.

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 can 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 can generate this automatically - you will need to customize 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 (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 (for Django 1.7)

1.11.189 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 [RoutablePageMixin](#).
- MenuItems can now have bundled JavaScript
- Added the `register_admin_menu_item` hook for registering menu items at startup. See [Hooks](#)
- Added a version indicator into the admin interface (hover over the wagtail to see it)
- Added Russian translation

Bug fixes

- Page URL generation now returns correct URLs for sites that have the main 'serve' view rooted somewhere other than '/'.
- Search results in the page chooser now respect the `page_type` parameter on `PageChooserPanel`.
- Rendition filenames are now prevented from going over 60 chars, even with a large `focal_point_key`.
- Child relations that are defined on a model's superclass (such as the base `Page` model) are now picked up correctly by the page editing form, page copy operations and the `replace_text` management command.
- Tags on images and documents are now committed to the search index immediately on saving.

Upgrade considerations

All features deprecated in 0.4 have been removed

See: *Deprecated features*

Search signal handlers have been moved

If you have an import in your urls.py file like from wagtail.wagtailsearch import register_signal_handlers, this must now be changed to from wagtail.wagtailsearch.signal_handlers import register_signal_handlers

Deprecated features

- The wagtail.wagtailsearch.indexed module has been renamed to wagtail.wagtailsearch.index

1.11.190 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 before 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 the 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.

1.11.191 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

1.11.192 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 customizing 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 customize 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, 556
wagtail.admin.viewsets, 557
wagtail.contrib.forms.panels, 552
wagtail.contrib.frontend_cache.utils,
 432
wagtail.contrib.legacy.richtext, 449
wagtail.contrib.redirects, 447
wagtail.contrib.redirects.models, 449
wagtail.contrib.routable_page, 433
wagtail.contrib.routable_page.models,
 436
wagtail.contrib.search_promotions, 437
wagtail.documents, 161
wagtail.images, 146
wagtail.images.models, 143
wagtail.models, 454
wagtail.query, 370
wagtail.test.utils.form_data, 231

INDEX

A

AccessibilityItem (*class in wagtail.admin.userbar*), 263
active (*wagtail.models.Task attribute*), 480
active (*wagtail.models.Workflow attribute*), 477
active_workflows (*wagtail.models.Task attribute*), 480
add_hit () (*wagtail.contrib.search_promotions.wagtail.contrib.search_promotions.models.QuerySet method*), 439
add_page () (*wagtail.contrib.frontend_cache.utils.PurgeBatch method*), 432
add_pages () (*wagtail.contrib.frontend_cache.utils.PurgeBatch method*), 432
add_redirect () (*wagtail.contrib.redirects.models.Redirect method*), 449
add_to_admin_menu (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
add_to_admin_menu (*wagtail.admin.viewsets.base.ViewSetGroup attribute*), 559
add_to_reference_index (*wagtail.admin.viewsets.model.ModelViewSet attribute*), 560
add_to_settings_menu (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
add_url () (*wagtail.contrib.frontend_cache.utils.PurgeBatch method*), 432
add_urls () (*wagtail.contrib.frontend_cache.utils.PurgeBatch method*), 432
add_view_class (*wagtail.admin.viewsets.model.ModelViewSet attribute*), 562
add_view_class (*wagtail.snippets.views.snippets.SnippetViewSet attribute*), 566
admin_default_ordering (*wagtail.models.Page attribute*), 461
admin_url_namespace (*wagtail.snippets.views.snippets.SnippetViewSet*)

attribute), 566
alias_of (*wagtail.models.Page attribute*), 456
all_pages () (*wagtail.models.Workflow method*), 478
all_tasks_with_status () (*wagtail.models.WorkflowState method*), 479
allowed_http_methods (*wagtail.models.Page attribute*), 458
ancestor_of (*wagtail.query.PageQuerySet method*), 372
approve () (*wagtail.models.TaskState method*), 482
as_object () (*wagtail.models.Revision method*), 476
attrs (*wagtail.admin.panels.FieldPanel attribute*), 547
attrs (*wagtail.admin.panels.FieldRowPanel attribute*), 550
attrs (*wagtail.admin.panels.HelpPanel attribute*), 551
attrs (*wagtail.admin.panels.InlinePanel attribute*), 548
attrs (*wagtail.admin.panels.MultiFieldPanel attribute*), 548
author_name (*wagtail.embeds.models.Embed attribute*), 172
axe_custom_checks (*wagtail.admin.userbar.AccessibilityItem attribute*), 263
axe_custom_rules (*wagtail.admin.userbar.AccessibilityItem attribute*), 263
axe_exclude (*wagtail.admin.userbar.AccessibilityItem attribute*), 263
axe_include (*wagtail.admin.userbar.AccessibilityItem attribute*), 263
axe_messages (*wagtail.admin.userbar.AccessibilityItem attribute*), 263
axe_rules (*wagtail.admin.userbar.AccessibilityItem attribute*), 263
axe_run_only (*wagtail.admin.userbar.AccessibilityItem attribute*), 263
base_block_class (*wagtail.admin.viewsets.chooser.ChooserViewSet*)

B

```

        attribute), 565
base_content_object (wagtail.models.Revision attribute), 476
base_content_type (wagtail.models.Revision attribute), 475
base_content_type (wagtail.models.WorkflowState attribute), 478
base_form_class (wagtail.models.Page attribute), 463
base_url_path (wagtail.snippets.views.snippets.SnippetViewSet attribute), 566
base_widget_class (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564
BaseLogEntry (class in wagtail.models), 484
bind_to_model () (wagtail.admin.panels.Panel method), 556
Block (class in wagtail.blocks), 379
BlockQuoteBlock (class in wagtail.blocks), 384
BooleanBlock (class in wagtail.blocks), 382
BoundPanel (class in wagtail.admin.panels.Panel), 557
built-in function
    log (), 326
    register_form_field_override (), 313

C
cache_key (wagtail.models.Page attribute), 464
cached_content_type (wagtail.models.Page attribute), 457
can_create_at () (wagtail.models.Page method), 462
can_exist_under () (wagtail.models.Page method), 462
can_move_to () (wagtail.models.Page method), 462
cancel () (wagtail.models.TaskState method), 482
cancel () (wagtail.models.WorkflowState method), 479
CharBlock (class in wagtail.blocks), 380
check_request_method () (wagtail.models.Page method), 459
child_of () (wagtail.query.PageQuerySet method), 371
children (wagtail.admin.panels.FieldRowPanel attribute), 550
children (wagtail.admin.panels.MultiFieldPanel attribute), 548
ChoiceBlock (class in wagtail.blocks), 385
choose_another_text (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564
choose_one_text (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564
choose_parent_view_class (wagtail.admin.viewsets.pages.PageListingViewSet attribute), 569
choose_results_view_class (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564
choose_view_class (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564
chooser_admin_url_namespace (wagtail.snippets.views.snippets.SnippetViewSet attribute), 566
chooser_base_url_path (wagtail.snippets.views.snippets.SnippetViewSet attribute), 566
chooser_per_page (wagtail.snippets.views.snippets.SnippetViewSet attribute), 565
chooser_viewset_class (wagtail.snippets.views.snippets.SnippetViewSet attribute), 567
ChooserViewSet (class in wagtail.admin.viewsets.chooser), 563
chosen_multiple_view_class (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564
chosen_view_class (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564
clean_name (wagtail.admin.panels.Panel property), 557
clone () (wagtail.admin.panels.Panel method), 556
clone_kwargs () (wagtail.admin.panels.Panel method), 556
columns (wagtail.admin.viewsets.pages.PageListingViewSet attribute), 569
Comment (class in wagtail.models), 486
comment (wagtail.models.BaseLogEntry attribute), 485
comment (wagtail.models.CommentReply attribute), 487
comment (wagtail.models.TaskState attribute), 482
comment_notifications (wagtail.models.PageSubscription attribute), 487
CommentReply (class in wagtail.models), 487
content (wagtail.admin.panels.HelpPanel attribute), 551
content (wagtail.models.Revision attribute), 475
content_changed (wagtail.models.BaseLogEntry attribute), 485
content_object (wagtail.models.Revision attribute), 475
content_object (wagtail.models.WorkflowState attribute), 478
content_type (wagtail.models.BaseLogEntry attribute), 484
content_type (wagtail.models.Page attribute), 454
content_type (wagtail.models.Revision attribute), 475
content_type (wagtail.models.Task attribute), 480
content_type (wagtail.models.TaskState attribute),

```

482
content_type (*wagtail.models.WorkflowContentType attribute*), 484
content_type (*wagtail.models.WorkflowState attribute*), 478
contentpath (*wagtail.models.Comment attribute*), 486
context_object_name (*wagtail.models.Page attribute*), 458
copy () (*wagtail.models.Page method*), 464
copy () (*wagtail.models.TaskState method*), 483
copy_approved_task_states_to_revision ()
(*wagtail.models.WorkflowState method*), 479
copy_for_translation () (*wagtail.models.Page method*), 460
copy_for_translation ()
(*wagtail.models.TranslatableMixin method*), 468
copy_view_class
(*wagtail.admin.viewsets.model.ModelViewSet attribute*), 562
copy_view_class
(*wagtail.snippets.views.snippets.SnippetViewSet attribute*), 566
copy_view_enabled
(*wagtail.admin.viewsets.model.ModelViewSet attribute*), 561
create_action_clicked_label
(*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 565
create_action_label
(*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 565
create_alias () (*wagtail.models.Page method*), 464
create_rendition ()
(*wagtail.images.models.AbstractImage method*),
143
create renditions ()
(*wagtail.images.models.AbstractImage method*),
144
create_template_name
(*wagtail.admin.viewsets.model.ModelViewSet attribute*), 562
create_view_class
(*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 564
created_at (*wagtail.models.Comment attribute*), 486
created_at (*wagtail.models.CommentReply attribute*),
487
created_at (*wagtail.models.Revision attribute*), 475
created_at (*wagtail.models.WorkflowState attribute*),
478
creation_form_class
(*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 565
creation_tab_label
(*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 565
attribute), 565
current_task_state (*wagtail.models.WorkflowState attribute*), 479
current_workflow_state
(*wagtail.models.WorkflowMixin attribute*), 474
current_workflow_task
(*wagtail.models.WorkflowMixin attribute*), 474
current_workflow_task_state
(*wagtail.models.WorkflowMixin attribute*), 474
custom_field_preprocess, 316
custom_value_preprocess, 316

D

data (*wagtail.models.BaseLogEntry attribute*), 484
DateBlock (*class in wagtail.blocks*), 383
DateTimeBlock (*class in wagtail.blocks*), 383
deactivate () (*wagtail.models.Task method*), 481
deactivate () (*wagtail.models.Workflow method*), 478
DecimalBlock (*class in wagtail.blocks*), 381
default_preview_mode
(*wagtail.models.Page attribute*), 459
default_preview_mode
(*wagtail.models.PreviewableMixin attribute*), 469
default_preview_size
(*wagtail.models.Page attribute*), 459
default_preview_size
(*wagtail.models.PreviewableMixin attribute*), 470
defer_streamfields ()
(*wagtail.query.PageQuerySet method*), 374
delete_template_name
(*wagtail.admin.viewsets.model.ModelViewSet attribute*), 563
delete_view_class
(*wagtail.admin.viewsets.model.ModelViewSet attribute*), 562
delete_view_class
(*wagtail.snippets.views.snippets.SnippetViewSet attribute*), 566
deleted (*wagtail.models.BaseLogEntry attribute*), 485
descendant_of ()
(*wagtail.query.PageQuerySet method*), 371
disable_comments (*wagtail.admin.panels.FieldPanel attribute*), 547
DocumentChooserBlock
(*class in wagtail.documents.blocks*), 386
draft_title (*wagtail.models.Page attribute*), 454
DraftStateMixin (*class in wagtail.models*), 472

E

edit_item_text
(*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 564

```

edit_template_name (wag-
tail.admin.viewsets.model.ModelViewSet attribute), 563
edit_view_class (wag-
tail.admin.viewsets.model.ModelViewSet attribute), 562
edit_view_class (wag-
tail.snippets.views.snippets.SnippetViewSet attribute), 566
EmailBlock (class in wagtail.blocks), 380
EmbedBlock (class in wagtail.embeds.blocks), 387
exact_type() (wagtail.query.PageQuerySet method), 373
exclude_fields_in_copy (wagtail.models.Page attribute), 463
exclude_fields_in_copy (wag-
tail.models.TaskState attribute), 482
exclude_form_fields (wag-
tail.admin.viewsets.chooser.ChooserViewSet attribute), 565
exclude_form_fields (wag-
tail.admin.viewsets.model.ModelViewSet attribute), 560
expand_db_attributes() (LinkHandler method), 343
expand_db_attributes_many() (LinkHandler method), 343
export_filename (wag-
tail.admin.viewsets.model.ModelViewSet attribute), 561
export_headings, 316
export_headings (wag-
tail.admin.viewsets.model.ModelViewSet attribute), 561
F
field_name (wagtail.admin.panels.FieldPanel attribute), 547
FieldBlock (class in wagtail.blocks), 380
FieldPanel (class in wagtail.admin.panels), 547
FieldRowPanel (class in wagtail.admin.panels), 550
filterset_class (wag-
tail.admin.viewsets.model.ModelViewSet attribute), 561
filterset_class (wag-
tail.admin.viewsets.pages.PageListingViewSet attribute), 569
find_existing_rendition() (wag-
tail.images.models.AbstractImage method), 143
find_existing_renditions() (wag-
tail.images.models.AbstractImage method), 144
find_for_request() (wagtail.models.Page static method), 458
find_for_request() (wagtail.models.Site static method), 466
finish() (wagtail.models.WorkflowState method), 479
finished_at (wagtail.models.TaskState attribute), 482
finished_by (wagtail.models.TaskState attribute), 482
first_common_ancestor() (wag-
tail.query.PageQuerySet method), 375
first_published_at (wag-
tail.models.DraftStateMixin attribute), 472
first_published_at (wagtail.models.Page attribute), 455
FloatBlock (class in wagtail.blocks), 381
focus() (built-in function), 393
form (wagtail.admin.panels.Panel.BoundPanel attribute), 557
form_fields (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 565
form_fields (wagtail.admin.viewsets.model.ModelViewSet attribute), 560
FormSubmissionsPanel (class in wag-
tail.contrib.forms.panels), 552
full_url (wagtail.models.Page attribute), 457
G
generate_rendition_file() (wag-
tail.images.models.AbstractImage method), 144
get() (wagtail.contrib.search_promotions.wagtail.contrib.search_promotions class method), 439
get_actions() (wagtail.models.Task method), 481
get_active() (wagtail.models.Locale class method), 467
get_admin_base_path() (wag-
tail.snippets.views.snippets.SnippetViewSet method), 568
get_admin_default_ordering() (wag-
tail.models.Page method), 460
get_admin_display_title() (wag-
tail.models.Page method), 458
get_admin_url_namespace() (wag-
tail.snippets.views.snippets.SnippetViewSet method), 568
get_ancestors() (wagtail.models.Page method), 460
get_axe_context() (wag-
tail.admin.userbar.AccessibilityItem method), 263
get_axe_custom_checks() (wag-
tail.admin.userbar.AccessibilityItem method), 263
get_axe_custom_rules() (wag-
tail.admin.userbar.AccessibilityItem method), 263

```

get_axe_exclude()
`tail.admin.userbar.AccessibilityItem`
 263

get_axe_include()
`tail.admin.userbar.AccessibilityItem`
 263

get_axe_messages()
`tail.admin.userbar.AccessibilityItem`
 263

get_axe_options()
`tail.admin.userbar.AccessibilityItem`
 263

get_axe_rules()
`tail.admin.userbar.AccessibilityItem`
 263

get_axe_run_only()
`tail.admin.userbar.AccessibilityItem`
 263

get_axe_spec()
`tail.admin.userbar.AccessibilityItem`
 264

get_block_class()
`tail.admin.viewsets.chooser.ChooserViewSet`
`method`, 565

get_bound_panel()
`(wagtail.admin.panels.Panel`
`method)`, 556

get_cache_key_components()
`(wag`
`tail.models.Page method)`, 464

get_children()
`(wagtail.models.Page method)`, 460

get_chooser_admin_base_path()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 568

get_chooser_admin_url_namespace()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 568

get_comment()
`(wagtail.models.TaskState`
`method)`, 483

get_context()
`(wagtail.blocks.Block`
`method)`, 379

get_context()
`(wagtail.models.Page`
`method)`, 458

get_converter_rule()
`(FeatureRegistry`
`method)`, 347

get_create_template()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 568

get_default()
`(wagtail.models.Locale`
`class`
`method)`, 467

get_default_workflow()
`(wag`
`tail.models.WorkflowMixin`
`class`
`method)`, 474

get_delete_template()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 568

get_descendants()
`(wagtail.models.Page`
`method)`, 460

get_description()
`(wagtail.blocks.Block`
`method)`, 379

get_description()
`(wagtail.models.Task`
`class`
`method)`, 481

get_display_name()
`(wagtail.models.Locale`
`method)`, 467

get_document_model()
`(in module wag`
`tail.documents)`, 161

get_document_model_string()
`(in module wag`
`tail.documents)`, 161

get_edit_handler()
`(wag`
`tail.admin.viewsets.model.ModelViewSet`
`method)`, 560

get_edit_handler()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 567

get_edit_template()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 568

get_editor_plugin()
`(FeatureRegistry`
`method)`, 347

get_form_class()
`(wagtail.admin.panels.Panel`
`method)`, 556

get_form_class()
`(wag`
`tail.admin.viewsets.model.ModelViewSet`
`method)`, 560

get_form_for_action()
`(wagtail.models.Task`
`method)`, 481

get_form_options()
`(wagtail.admin.panels.Panel`
`method)`, 556

get_full_url()
`(wagtail.models.Page`
`method)`, 457

get_history_template()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 568

get_image_model()
`(in module wagtail.images)`, 146

get_image_model_string()
`(in module wag`
`tail.images)`, 146

get_index_results_template()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 568

get_index_template()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 567

get_inspect_template()
`(wag`
`tail.snippets.views.snippets.SnippetViewSet`
`method)`, 568

get_instance()
`(LinkHandler`
`method)`, 344

get_latest_revision_as_object()
`(wag`
`tail.models.RevisionMixin`
`method)`, 471

get_lock()
`(wagtail.models.LockableMixin`
`method)`, 473

get_many()
`(LinkHandler`
`method)`, 344

get_menu_item()
`(wag`
`tail.admin.viewsets.base.ViewSet`
`method)`,

```

559
get_menu_item() (wagtail.admin.viewsets.base.ViewSetGroup method), 559
get_model() (LinkHandler method), 344
get_next_task() (wagtail.models.WorkflowState method), 479
get_object_list() (wagtail.admin.viewsets.chooser.ChooserViewSet method), 565
get_parent() (wagtail.models.Page method), 460
get_permissions_to_register() (wagtail.admin.viewsets.model.ModelViewSet method), 560
get_preview_context() (wagtail.blocks.Block method), 379
get_preview_context() (wagtail.models.PreviewableMixin method), 470
get_preview_template() (wagtail.blocks.Block method), 379
get_preview_template() (wagtail.models.PreviewableMixin method), 470
get_preview_value() (wagtail.blocks.Block method), 379
get_queryset(), 315
get_queryset() (wagtail.snippets.views.snippets.SnippetViewSet method), 567
get_rendition() (wagtail.images.models.AbstractImage method), 143
get_renditions() (wagtail.images.models.AbstractImage method), 144
get_route_paths() (wagtail.models.Page method), 462
get_siblings() (wagtail.models.Page method), 460
get_site() (wagtail.models.Page method), 458
get_site_root_paths() (wagtail.models.Site static method), 466
get_specific() (wagtail.models.Page method), 456
get_subpage_urls() (wagtail.contrib.routable_page.models.RoutablePageMixin class method), 436
get_task_states_user_can_moderate() (wagtail.models.Task method), 481
get_template() (wagtail.blocks.Block method), 379
get_template() (wagtail.models.Page method), 458
get_template_for_action() (wagtail.models.Task method), 481
get_translation() (wagtail.models.Page method), 460
get_translation() (wagtail.models.TranslatableMixin method), 468
get_translation_model() (wagtail.models.TranslatableMixin class method), 469
get_translation_or_none() (wagtail.models.Page method), 460
get_translation_or_none() (wagtail.models.TranslatableMixin method), 468
get_translations() (wagtail.models.Page method), 460
get_translations() (wagtail.models.TranslatableMixin method), 468
get_upload_to() (wagtail.images.models.AbstractImage method), 147
get_upload_to() (wagtail.images.models.AbstractRendition method), 147
get_url() (wagtail.models.Page method), 457
get_url_name() (wagtail.admin.viewsets.base.ViewSet method), 558
get_url_parts() (wagtail.models.Page method), 458
get_urlpatterns() (wagtail.admin.viewsets.base.ViewSet method), 558
get_verbose_name() (wagtail.models.Task class method), 480
get_workflow() (wagtail.models.WorkflowMixin method), 474
getState() (built-in function), 392
getValue() (built-in function), 392
group (wagtail.models.GroupPagePermission attribute), 476
GroupPagePermission (class in wagtail.models), 476

H
handle_options_request() (wagtail.models.Page method), 459
has_translation() (wagtail.models.Page method), 460
has_translation() (wagtail.models.TranslatableMixin method), 468
has_unpublished_changes (wagtail.models.DraftStateMixin attribute), 472
has_unpublished_changes (wagtail.models.Page attribute), 455
has_workflow (wagtail.models.WorkflowMixin attribute), 474
header_icon, 316
height (wagtail.embeds.models.Embed attribute), 173
HelpPanel (class in wagtail.admin.panels), 551
history_template_name (wagtail.admin.viewsets.model.ModelViewSet attribute), 563

```

history_view_class (wagtail.admin.viewsets.model.ModelViewSet attribute), 562

history_view_class (wagtail.snippets.views.snippets.SnippetViewSet attribute), 566

hostname (wagtail.models.Site attribute), 465

html (wagtail.embeds.models.Embed attribute), 172

|

icon (wagtail.admin.viewsets.base.ViewSet attribute), 558

icon (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 563

id_for_label() (wagtail.admin.panels.Panel.BoundPanel method), 557

identifier (LinkHandler attribute), 343

idForLabel (None attribute), 392

ImageBlock (class in wagtail.images.blocks), 386

ImageChooserBlock (class in wagtail.images.blocks), 387

in_menu() (wagtail.query.PageQuerySet method), 370

in_site() (wagtail.query.PageQuerySet method), 371

index_results_template_name (wagtail.admin.viewsets.model.ModelViewSet attribute), 562

index_results_url_name, 316

index_template_name (wagtail.admin.viewsets.model.ModelViewSet attribute), 562

index_url_name, 316

index_view_class (wagtail.admin.viewsets.model.ModelViewSet attribute), 562

index_view_class (wagtail.admin.viewsets.pages.PageListingViewSet attribute), 569

index_view_class (wagtail.snippets.views.snippets.SnippetViewSet attribute), 566

inline_formset() (in module wagtail.test.utils.form_data), 232

InlinePanel (class in wagtail.admin.panels), 548

inspect_template_name (wagtail.admin.viewsets.model.ModelViewSet attribute), 563

inspect_view_class (wagtail.admin.viewsets.model.ModelViewSet attribute), 562

inspect_view_class (wagtail.snippets.views.snippets.SnippetViewSet attribute), 566

inspect_view_enabled (wagtail.admin.viewsets.model.ModelViewSet attribute), 560

inspect_view_fields (wagtail.admin.viewsets.model.ModelViewSet attribute), 561

inspect_view_fields_exclude (wagtail.admin.viewsets.model.ModelViewSet attribute), 562

instance (wagtail.admin.panels.Panel.BoundPanel attribute), 557

IntegerBlock (class in wagtail.blocks), 381

is_active (wagtail.models.Locale attribute), 467

is_bidi (wagtail.models.Locale attribute), 467

is_creatable (wagtail.models.Page attribute), 462

is_default (wagtail.models.Locale attribute), 467

is_default_site (wagtail.models.Site attribute), 465

is_latest_revision() (wagtail.models.Revision method), 476

is_previewable() (wagtail.models.PreviewableMixin method), 470

is_shown(), 490

is_shown() (wagtail.admin.panels.Panel.BoundPanel method), 557

items (wagtail.admin.viewsets.base.ViewSetGroup attribute), 559

L

label (wagtail.admin.panels.InlinePanel attribute), 548

label (wagtail.models.BaseLogEntry attribute), 484

language_code (wagtail.models.Locale attribute), 467

language_name (wagtail.models.Locale attribute), 467

language_name_local (wagtail.models.Locale attribute), 467

language_name_localized (wagtail.models.Locale attribute), 467

last_published_at (wagtail.models.DraftStateMixin attribute), 472

last_published_at (wagtail.models.Page attribute), 455

last_updated (wagtail.embeds.models.Embed attribute), 173

latest_revision (wagtail.models.RevisionMixin attribute), 470

LinkHandler (built-in class), 343

list_display (wagtail.admin.viewsets.model.ModelViewSet attribute), 560

list_export, 316

list_export (wagtail.admin.viewsets.model.ModelViewSet attribute), 561

list_filter (wagtail.admin.viewsets.model.ModelViewSet attribute), 561

list_per_page (wagtail.admin.viewsets.model.ModelViewSet attribute), 560

ListBlock (*class in wagtail.blocks*), 389
 live (*wagtail.models.DraftStateMixin attribute*), 472
 live (*wagtail.models.Page attribute*), 455
 live () (*wagtail.query.PageQuerySet method*), 370
 live_revision (*wagtail.models.DraftStateMixin attribute*), 472
 Locale (*class in wagtail.models*), 467
 locale (*wagtail.models.Page attribute*), 456
 locale (*wagtail.models.TranslatableMixin attribute*), 468
 localized (*wagtail.models.Page attribute*), 461
 localized (*wagtail.models.TranslatableMixin attribute*), 469
 localized_draft (*wagtail.models.Page attribute*), 461
 lock_view_class (*wagtail.snippets.views.snippets.SnippetViewSet attribute*), 567
 LockableMixin (*class in wagtail.models*), 473
 locked (*wagtail.models.LockableMixin attribute*), 473
 locked (*wagtail.models.Page attribute*), 456
 locked_at (*wagtail.models.LockableMixin attribute*), 473
 locked_at (*wagtail.models.Page attribute*), 456
 locked_by (*wagtail.models.LockableMixin attribute*), 473
 locked_by (*wagtail.models.Page attribute*), 456
 locked_for_user () (*wagtail.models.Task method*), 481
 log ()
 built-in function, 326

M

max_count (*wagtail.models.Page attribute*), 462
 max_count_per_parent (*wagtail.models.Page attribute*), 462
 max_num (*wagtail.admin.panels.InlinePanel attribute*), 548
 max_width (*wagtail.embeds.models.Embed attribute*), 172
 media, 309
 menu_hook (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
 menu_icon (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
 menu_icon (*wagtail.admin.viewsets.base.ViewSetGroup attribute*), 559
 menu_item_class (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
 menu_item_class (*wagtail.admin.viewsets.base.ViewSetGroup attribute*), 559
 menu_label (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
 menu_label (*wagtail.admin.viewsets.base.ViewSetGroup attribute*), 559
 menu_label (*wagtail.admin.viewsets.model.ModelViewSet attribute*), 560
 menu_label (*wagtail.admin.viewsets.model.ModelViewSetGroup attribute*), 563
 menu_name (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
 menu_name (*wagtail.admin.viewsets.base.ViewSetGroup attribute*), 559
 menu_order (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
 menu_order (*wagtail.admin.viewsets.base.ViewSetGroup attribute*), 559
 menu_url (*wagtail.admin.viewsets.base.ViewSet attribute*), 558
 min_num (*wagtail.admin.panels.InlinePanel attribute*), 548
 model (*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 563
 model (*wagtail.admin.viewsets.model.ModelViewSet attribute*), 560
 model (*wagtail.admin.viewsets.pages.PageListingViewSet attribute*), 569
 model (*wagtail.snippets.views.snippets.SnippetViewSet attribute*), 565
 ModelViewSet (*class in wagtail.admin.viewsets.model*), 560
 ModelViewSetGroup (*class in wagtail.admin.viewsets.model*), 563
 module
 wagtail.admin.panels, 556
 wagtail.admin.viewsets, 557
 wagtail.contrib.forms.panels, 552
 wagtail.contrib.frontend_cache.utils, 432
 wagtail.contrib.legacy.richtext, 449
 wagtail.contrib.redirects, 447
 wagtail.contrib.redirects.models, 449
 wagtail.contrib.routable_page, 433
 wagtail.contrib.routable_page.models, 436
 wagtail.contrib.search_promotions, 437
 wagtail.documents, 161
 wagtail.images, 146
 wagtail.images.models, 143
 wagtail.models, 454
 wagtail.query, 370
 wagtail.test.utils.form_data, 231
 MultiFieldPanel (*class in wagtail.admin.panels*), 548
 MultipleChoiceBlock (*class in wagtail.blocks*), 385
 MultipleChooserPanel (*class in wagtail.admin.panels*), 548

`tail.admin.panels), 549`

N

`name (wagtail.admin.viewsets.base.ViewSet attribute), 557`
`name (wagtail.models.Task attribute), 480`
`name (wagtail.models.Workflow attribute), 477`
`nested_form_data () (in module wagtail.test.utils.form_data), 231`
`not_ancestor_of () (wagtail.query.PageQuerySet method), 372`
`not_child_of () (wagtail.query.PageQuerySet method), 372`
`not_descendant_of () (wagtail.query.PageQuerySet method), 371`
`not_exact_type () (wagtail.query.PageQuerySet method), 374`
`not_in_menu () (wagtail.query.PageQuerySet method), 370`
`not_live () (wagtail.query.PageQuerySet method), 370`
`not_page () (wagtail.query.PageQuerySet method), 371`
`not_parent_of () (wagtail.query.PageQuerySet method), 372`
`not_public () (wagtail.query.PageQuerySet method), 373`
`not_sibling_of () (wagtail.query.PageQuerySet method), 372`
`not_type () (wagtail.query.PageQuerySet method), 373`

O

`object_id (wagtail.models.Revision attribute), 475`
`object_id (wagtail.models.WorkflowState attribute), 478`
`object_id () (wagtail.models.BaseLogEntry method), 485`
`object_verbose_name (wagtail.models.BaseLogEntry attribute), 485`
`objects (wagtail.models.Revision attribute), 476`
`on_action () (wagtail.models.Task method), 480`
`on_model_bound () (wagtail.admin.panels.Panel method), 556`
`on_register () (wagtail.admin.viewsets.base.ViewSet method), 558`
`order, 490`
`Orderable (class in wagtail.models), 477`
`ordering (wagtail.admin.viewsets.model.ModelViewSet attribute), 560`
`owner (wagtail.models.Page attribute), 455`

P

`Page (class in wagtail.models), 454`
`page (wagtail.models.Comment attribute), 486`
`page (wagtail.models.GroupPagePermission attribute), 476`
`page (wagtail.models.PageLogEntry attribute), 485`

`page (wagtail.models.PageSubscription attribute), 487`
`page (wagtail.models.PageViewRestriction attribute), 477`
`page (wagtail.models.WorkflowPage attribute), 483`
`page () (wagtail.query.PageQuerySet method), 371`
`page_revisions (wagtail.models.Revision attribute), 476`
`page_title, 315`
`page_title (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564`
`page_type_display_name (wagtail.models.Page attribute), 457`
`PageChooserBlock (class in wagtail.blocks), 386`
`PageChooserPanel (class in wagtail.admin.panels), 551`
`PageListingViewSet (class in wagtail.admin.viewsets.pages), 569`
`PageLogEntry (class in wagtail.models), 485`
`PageQuerySet (class in wagtail.query), 370`
`PageSubscription (class in wagtail.models), 487`
`PageViewRestriction (class in wagtail.models), 477`
`Panel (class in wagtail.admin.panels), 556`
`panel (wagtail.admin.panels.Panel.BoundPanel attribute), 557`
`panels (wagtail.admin.panels.InlinePanel attribute), 548`
`parent_of () (wagtail.query.PageQuerySet method), 372`
`parent_page_types (wagtail.models.Page attribute), 461`
`password (wagtail.models.PageViewRestriction attribute), 477`
`password_required_template (wagtail.models.Page attribute), 462`
`per_page (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564`
`permission (wagtail.admin.panels.FieldPanel attribute), 547`
`permission (wagtail.admin.panels.FieldRowPanel attribute), 550`
`permission (wagtail.admin.panels.MultiFieldPanel attribute), 548`
`port (wagtail.models.Site attribute), 465`
`position (wagtail.models.Comment attribute), 486`
`prefetch_related () (wagtail.query.PageQuerySet method), 376`
`prefix (wagtail.admin.panels.Panel.BoundPanel attribute), 557`
`preserve_url_parameters (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564`
`preview_modes (wagtail.models.Page attribute), 459`
`preview_modes (wagtail.models.PreviewableMixin attribute), 469`
`preview_on_add_view_class (wagtail.snippets.views.snippets.SnippetViewSet`

```

attribute), 567
preview_on_edit_view_class (wag-
tail.snippets.views.snippets.SnippetViewSet
attribute), 567
preview_sizes (wagtail.models.Page attribute), 459
preview_sizes (wagtail.models.PreviewableMixin at-
tribute), 469
PreviewableMixin (class in wagtail.models), 469
private() (wagtail.query.PageQuerySet method), 373
private_page_options (wagtail.models.Page
attribute), 462
provider_name (wagtail.embeds.models.Embed
attribute), 172
public() (wagtail.query.PageQuerySet method), 372
publish() (wagtail.models.DraftStateMixin method),
472
publish() (wagtail.models.Revision method), 476
purge() (wagtail.contrib.frontend_cache.utils.PurgeBatch
method), 432
PurgeBatch (class in wag-
tail.contrib.frontend_cache.utils), 432

R
RawHTMLBlock (class in wagtail.blocks), 384
read_only (wagtail.admin.panels.FieldPanel attribute),
547
Redirect (class in wagtail.contrib.redirects.models), 449
RegexBlock (class in wagtail.blocks), 382
register_converter_rule() (FeatureRegistry
method), 347
register_editor_plugin() (FeatureRegistry
method), 347
register_embed_type() (FeatureRegistry method),
345
register_form_field_override()
built-in function, 313
register_link_type() (FeatureRegistry method),
345
register_widget (wag-
tail.admin.viewsets.chooser.ChooserViewSet
attribute), 565
reject() (wagtail.models.TaskState method), 482
relation_name (wagtail.admin.panels.InlinePanel at-
tribute), 548
relative_url() (wagtail.models.Page method), 457
render() (built-in function), 392
render() (wagtail.contrib.routable_page.models.RoutablePageMixin
method), 436
render_html(), 309
request (wagtail.admin.panels.Panel.BoundPanel
attribute), 557
requested_by (wagtail.models.WorkflowState at-
tribute), 479

resolve_subpage() (wag-
tail.contrib.routable_page.models.RoutablePageMixin
method), 436
resolved_at (wagtail.models.Comment attribute), 486
resolved_by (wagtail.models.Comment attribute), 486
restriction_type (wag-
tail.models.PageViewRestriction attribute),
477
results_template_name, 315
resume() (wagtail.models.WorkflowState method), 479
reverse_subpage() (wag-
tail.contrib.routable_page.models.RoutablePageMixin
method), 437
Revision (class in wagtail.models), 475
revision (wagtail.models.BaseLogEntry attribute), 484
revision (wagtail.models.TaskState attribute), 481
revision_created (wagtail.models.Comment at-
tribute), 486
RevisionMixin (class in wagtail.models), 470
revisions (wagtail.models.RevisionMixin attribute),
471
revisions() (wagtail.models.WorkflowState method),
479
revisions_compare_view_class (wag-
tail.snippets.views.snippets.SnippetViewSet
attribute), 567
revisions_revert_view_class (wag-
tail.snippets.views.snippets.SnippetViewSet
attribute), 567
revisions_unschedule_view_class (wag-
tail.snippets.views.snippets.SnippetViewSet
attribute), 567
revisions_view_class (wag-
tail.snippets.views.snippets.SnippetViewSet
attribute), 566
rich_text() (in module wagtail.test.utils.form_data),
231
RichTextBlock (class in wagtail.blocks), 384
root_page (wagtail.models.Site attribute), 465
root_url (wagtail.models.Site attribute), 466
RoutablePageMixin (class in wag-
tail.contrib.routable_page.models), 436
routablepageurl() (in module wag-
tail.contrib.routable_page.templatetags.wagtailroutablepage_tags),
437
route() (wagtail.contrib.routable_page.models.RoutablePageMixin
method), 436
route() (wagtail.models.Page method), 458
route_for_request() (wagtail.models.Page static
method), 458

S
save() (wagtail.models.Page method), 464

```

save_revision() (*wagtail.models.RevisionMixin method*), 471
search() (*wagtail.query.PageQuerySet method*), 373
search_backend_name (*wagtail.admin.viewsets.model.ModelViewSet attribute*), 561
search_description (*wagtail.models.Page attribute*), 455
search_fields (*wagtail.admin.viewsets.model.ModelViewSet attribute*), 561
search_fields (*wagtail.models.Page attribute*), 461
search_tab_label (*wagtail.admin.viewsets.chooser.ChooserViewSet attribute*), 565
select_related() (*wagtail.query.PageQuerySet method*), 375
seo_title (*wagtail.models.Page attribute*), 455
serve() (*wagtail.models.Page method*), 458
serve_preview() (*wagtail.models.Page method*), 459
serve_preview() (*wagtail.models.PreviewableMixin method*), 470
setState() (*built-in function*), 392
show_in_menus (*wagtail.models.Page attribute*), 455
sibling_of() (*wagtail.query.PageQuerySet method*), 372
Site (*class in wagtail.models*), 465
site_name (*wagtail.models.Site attribute*), 465
slug (*wagtail.models.Page attribute*), 454
SnippetChooserBlock (*class in wagtail.snippets.blocks*), 387
SnippetViewSet (*class in wagtail.snippets.views.snippets*), 565
SnippetViewSetGroup (*class in wagtail.snippets.views.snippets*), 569
sort_order (*wagtail.models.Orderable attribute*), 477
sort_order (*wagtail.models.WorkflowTask attribute*), 483
specific (*wagtail.models.Page attribute*), 457
specific (*wagtail.models.Task attribute*), 480
specific (*wagtail.models.TaskState attribute*), 482
specific() (*wagtail.query.PageQuerySet method*), 374
specific_class (*wagtail.models.Page attribute*), 457
specific_deferred (*wagtail.models.Page attribute*), 457
start() (*wagtail.models.Task method*), 480
start() (*wagtail.models.Workflow method*), 478
started_at (*wagtail.models.TaskState attribute*), 482
StaticBlock (*class in wagtail.blocks*), 388
status (*wagtail.models.TaskState attribute*), 481
status (*wagtail.models.WorkflowState attribute*), 478
STATUS_CHOICES (*wagtail.models.TaskState attribute*), 482
STATUS_CHOICES (*wagtail.models.WorkflowState attribute*), 479
StreamBlock (*class in wagtail.blocks*), 390
streamfield() (*in module wagtail.test.utils.form_data*), 232
StructBlock (*class in wagtail.blocks*), 388
subpage_types (*wagtail.models.Page attribute*), 461
SummaryItem(), 490

T

Task (*class in wagtail.models*), 480
task (*wagtail.models.TaskState attribute*), 481
task (*wagtail.models.WorkflowTask attribute*), 483
task_state_class (*wagtail.models.Task attribute*), 480
task_type_started_at (*wagtail.models.TaskState attribute*), 482
tasks (*wagtail.models.Workflow attribute*), 478
TaskState (*class in wagtail.models*), 481
template (*wagtail.admin.panels.HelpPanel attribute*), 551
template_name, 315
template_prefix (*wagtail.admin.viewsets.model.ModelViewSet attribute*), 562
text (*wagtail.models.Comment attribute*), 486
text (*wagtail.models.CommentReply attribute*), 487
TextBlock (*class in wagtail.blocks*), 380
thumbnail_url (*wagtail.embeds.models.Embed attribute*), 172
TimeBlock (*class in wagtail.blocks*), 383
timestamp (*wagtail.models.BaseLogEntry attribute*), 484
title (*wagtail.embeds.models.Embed attribute*), 172
title (*wagtail.models.Page attribute*), 454
TitleFieldPanel (*class in wagtail.admin.panels*), 552
TranslatableMixin (*class in wagtail.models*), 468
translation_key (*wagtail.models.Page attribute*), 456
translation_key (*wagtail.models.TranslatableMixin attribute*), 468
type (*wagtail.embeds.models.Embed attribute*), 172
type() (*wagtail.query.PageQuerySet method*), 373

U

unlock_view_class (*wagtail.snippets.views.snippets.SnippetViewSet attribute*), 567
unpublish() (*wagtail.models.DraftStateMixin method*), 472
unpublish() (*wagtail.query.PageQuerySet method*), 374
unpublish_view_class (*wagtail.snippets.views.snippets.SnippetViewSet*)

attribute), 567

update() (wagtail.models.WorkflowState method), 479

update_aliases() (wagtail.models.Page method), 464

updated_at (wagtail.models.Comment attribute), 486

updated_at (wagtail.models.CommentReply attribute), 487

url (wagtail.embeds.models.Embed attribute), 172

url_filter_parameters (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564

url_namespace (wagtail.admin.viewsets.base.ViewSet attribute), 558

url_prefix (wagtail.admin.viewsets.base.ViewSet attribute), 557

URLBlock (class in wagtail.blocks), 382

usage_view_class (wagtail.admin.viewsets.model.ModelViewSet attribute), 562

usage_view_class (wagtail.snippets.views.snippets.SnippetViewSet attribute), 566

user (wagtail.models.BaseLogEntry attribute), 484

user (wagtail.models.Comment attribute), 486

user (wagtail.models.CommentReply attribute), 487

user (wagtail.models.PageSubscription attribute), 487

user (wagtail.models.Revision attribute), 475

user_can_access_editor() (wagtail.models.Task method), 480

user_can_lock() (wagtail.models.Task method), 480

user_can_unlock() (wagtail.models.Task method), 481

user_display_name (wagtail.models.BaseLogEntry attribute), 485

V

ViewSet (class in wagtail.admin.viewsets.base), 557

ViewSetGroup (class in wagtail.admin.viewsets.base), 559

W

wagtail.admin.forms.CopyForm (built-in class), 206

wagtail.admin.forms.WagtailAdminModelForm (built-in class), 205

wagtail.admin.forms.WagtailAdminPageForm (built-in class), 205

wagtail.admin.panels module, 556

wagtail.admin.viewsets module, 557

wagtail.contrib.forms.panels module, 552

wagtail.contrib.frontend_cache.utils module, 432

wagtail.contrib.legacy.richtext module, 449

wagtail.contrib.redirects module, 447

wagtail.contrib.redirects.models module, 449

wagtail.contrib.routable_page module, 433

wagtail.contrib.routable_page.models module, 436

wagtail.contrib.search_promotions module, 437

wagtail.contrib.search_promotions.models.Query (class in wagtail.contrib.search_promotions), 439

wagtail.documents module, 161

wagtail.embeds.models.Embed (built-in class), 172

wagtail.fields.StreamField (built-in class), 377

wagtail.images module, 146

wagtail.images.models module, 143

wagtail.models module, 454

wagtail.query module, 370

wagtail.test.utils.form_data module, 231

widget (wagtail.admin.panels.FieldPanel attribute), 547

widget_class (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564

widget_telepath_adapter_class (wagtail.admin.viewsets.chooser.ChooserViewSet attribute), 564

width (wagtail.embeds.models.Embed attribute), 172

with_content_json() (wagtail.models.DraftStateMixin method), 473

with_content_json() (wagtail.models.LockableMixin method), 473

with_content_json() (wagtail.models.Page method), 463

with_content_json() (wagtail.models.RevisionMixin method), 471

Workflow (class in wagtail.models), 477

workflow (wagtail.models.WorkflowContentType attribute), 484

workflow (wagtail.models.WorkflowPage attribute), 483

workflow (wagtail.models.WorkflowState attribute), 478

workflow (wagtail.models.WorkflowTask attribute), 483

workflow_in_progress (wagtail.models.WorkflowMixin attribute), [474](#)
workflow_state (wagtail.models.TaskState attribute), [481](#)
workflow_states (wagtail.models.WorkflowMixin attribute), [474](#)
WorkflowContentType (class in wagtail.models), [484](#)
WorkflowMixin (class in wagtail.models), [474](#)
WorkflowPage (class in wagtail.models), [483](#)
workflows (wagtail.models.Task attribute), [480](#)
WorkflowState (class in wagtail.models), [478](#)
WorkflowTask (class in wagtail.models), [483](#)