wagtailmenus Documentation

Release 2.12

Andy Babic

Nov 17, 2018

Contents

wagtailmenus is an open-source extension for Wagtail CMS to help you define, manage and render menus in a consistent, yet flexible way.

The current version is tested for compatibility with the following:

- Wagtail versions 2.0 to 2.2
- Django versions 1.11 to 2.0
- Python versions 3.4 to 3.7

To find out more about what wagtailmenus does and why, see *Overview and key concepts*

To view the code, open an issue, or submit a pull request, view the wagtailmenus project on github.

Below are some useful links to help you get you started:

- **First steps**
  - *Installing wagtailmenus*
  - *Managing main menus via the CMS*
  - *Managing flat menus via the CMS*
- **Rendering menus**
  - *Template tags reference*
  - *Using your own menu templates*
- **Optional page models**
  - *The MenuPage and MenuPageMixin models*
  - *The AbstractLinkPage model*

CHAPTER 1

Full index

1.1 Overview and key concepts

- *Better control over top-level menu items*
- *Link to pages, custom URLs, or a combination of both*
- *Multi-level menus generated from your existing page tree*
- *Define menus for all your project needs*
- *Suitable for single-site or multi-site projects*
- *Solves the problem of important page links becoming just 'toggles' in multi-level menus*
- *Use the default menu templates for rendering, or easily add your own*

1.1.1 Better control over top-level menu items

When you have a 'main navigation' menu powered purely by the page tree, things can get a little tricky, as they are often designed in a way that is very sensitive to change. Some moderators might not understand that publishing a new page at the top level (or reordering those pages) will dramatically affect the main navigation (and possibly even break the design). And really, *why should they?*

Wagtailmenus solves this problem by allowing you to choose exactly which pages should appear as top-level items. It adds new functionality to Wagtail's CMS, allowing you to simply and effectively create and manage menus, using a familiar interface.

You can also use Wagtail's built-in group permissions system to control which users have permission to make changes to menus.

1.1.2 Link to pages, custom URLs, or a combination of both

The custom URL field won't force you to enter a valid URL, so you can add things like `#request-callback` or `#signup` to link to areas on the active page (perhaps as JS modals).

You can also define additional values to be added to a page's URL, letting you jump to a specific anchor point on a page, or include fixed GET parameters for analytics or to trigger custom functionality.

1.1.3 Multi-level menus generated from your existing page tree

We firmly believe that your page tree is the best place to define the structure, and the 'natural order' of pages within your site. Wagtailmenus only allows you to define the top-level items for each menu, because offering anything more would inevitably lead to site managers redefining parts of the page tree in multiple places, doomed to become outdated as the original tree changes over time.

To generate multi-level menus, wagtailmenus takes the top-level items you define for each menu and automatically combines it with your page tree, efficiently identifying ancestors for each selected pages and outputting them as sub-menus following the same structure and order.

You can prevent any page from appearing menus simply by setting `show_in_menus` to `False`. Pages will also no longer be included in menus if they are unpublished.

1.1.4 Define menus for all your project needs

Have you ever hard-coded a menu into a footer at the start of a project, only for those pages never to come into existence? Or maybe the pages were created, but their URLs changed later on, breaking the hard-coded links? How about 'secondary navigation' menus in headers?

As well as giving you control over your 'main menu', wagtailmenus allows you to manage any number of additional menus via the CMS as 'flat menus', meaning they too can benefit from page links that dynamically update to reflect tree position or status changes.

Don't hard-code another menu again! CMS-managed menus allow you to make those 'emergency changes' and 'last-minute tweaks' without having to touch a single line of code.

Note: Despite the name, 'flat menus' can be configured to render as multi-level menus if you need them to.

1.1.5 Suitable for single-site or multi-site projects

While main menus always have to be defined for each site, for flat menus, you can support multiple sites using any of the following approaches:

- Define a new menu for each site
- Define a menu for your default site and reuse it for the others
- Create new menus for some sites, but use the default site's menu for others

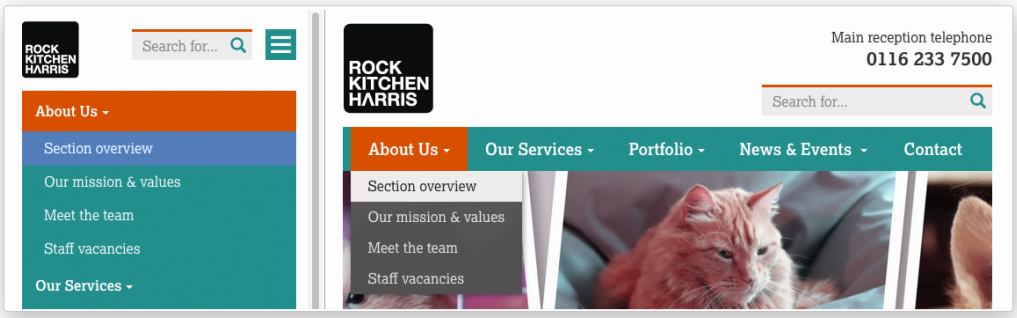
You can even use different approaches for different flat menus in the same project. If you'd like to learn more, take a look at the `fall_back_to_default_site_menus` option in *Supported arguments*

A **copy** feature in also available from the flat menu management interface, allowing you to quickly and easily copy existing menus from one site to another.

In a multi-site project, you can also configure wagtailmenus to use separate sets of templates for each site for rendering (See *Using preferred paths and names for your templates*)

1.1.6 Solves the problem of important page links becoming just 'toggles' in multi-level menus

Extend the `wagtailmenus.models.MenuPage` model instead of the usual `wagtail.wagtailcore.models.Page` model to create your custom page types, and gain a couple of extra fields that will allow you to configure certain pages to appear again alongside their children in multi-level menus. Use the menu tags provided, and that behaviour will remain consistent in all menus throughout your site. To find out more, see: *The MenuPage and MenuPageMixin models*



1.1.7 Use the default menu templates for rendering, or easily add your own

Each menu tag comes with a default template that's designed to be fully accessible and compatible with Bootstrap 3. However, if you don't want to use the default templates, wagtailmenus makes it easy to use your own, using whichever approach works best for you:

- Use settings to change the default templates used for each tag
- Specify templates using `template` and `sub_menu_template` arguments for any of the included menu tags (See *Specifying menu templates using template tag parameters*).
- Put your templates in a preferred location within your project and wagtailmenus will pick them up automatically (See *Using preferred paths and names for your templates*).

1.2 Installing wagtailmenus

1. Install the package using pip:

```
pip install wagtailmenus
```

2. Add `wagtailmenus` and `wagtail.contrib.modeladmin` to the `INSTALLED_APPS` setting in your project settings:

```
INSTALLED_APPS = [
    ...
    'wagtail.contrib.modeladmin',  # Don't repeat if it's there already
```

(continues on next page)

(continued from previous page)

```
    'wagtailmenus',
]
```

3. Add `wagtailmenus.context_processors.wagtailmenus` to the `context_processors` list in your `TEMPLATES` setting. The setting should look something like this:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        '   DIRS': [
            os.path.join(PROJECT_ROOT, 'templates'),
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.contrib.auth.context_processors.auth',
                'django.template.context_processors.debug',
                'django.template.context_processors.i18n',
                'django.template.context_processors.media',
                'django.template.context_processors.request',
                'django.template.context_processors.static',
                'django.template.context_processors.tz',
                'django.contrib.messages.context_processors.messages',
                'wagtail.contrib.settings.context_processors.settings',
                'wagtailmenus.context_processors.wagtailmenus',
            ],
        },
    },
]
```

4. Run migrations to create database tables for wagtailmenus:

```
python manage.py migrate wagtailmenus
```

5. **This step is optional**. If you're adding wagtailmenus to an existing project, and the tree for each site follows a structure similar to the example below, you may find it useful to run the 'autopopulate_main_menus' command to populate main menus for your site(s).

   However, this will only yield useful results if the 'root page' you've set for your site(s) is what you consider to be the 'Home' page, and the pages directly below that are the pages you'd like to link to in your main menu.

   For example, if your page structure looked like the following:

```
Home (Set as 'root page' for the site)
├── About us
├── What we do
├── Careers
|   ├── Vacancy one
|   └── Vacancy two
├── News & events
|   ├── News
|   └── Events
└── Contact us
```

   Running the command from the console:

```
python manage.py autopopulate_main_menus
```

Would create a main menu with the following items:

- About us
- What we do
- Careers
- News & events
- Contact us

If you'd like wagtailmenus to also include a link to the 'home page', you can use the '–add-home-links' option, like so:

```
python manage.py autopopulate_main_menus --add-home-links=True
```

This would create a main menu with the following items:

- Home
- About us
- What we do
- Careers
- News & events
- Contact us

Note: The 'autopopulate_main_menus' command is meant as 'run once' command to help you get started, and will only affect menus that do not already have any menu items defined. Running it more than once won't have any effect, even if you make changes to your page tree before running it again.

1.2.1 Installing `wagtail-condensedinlinepanel`

Although doing so is entirely optional, for an all-round better menu editing experience, we recommend using wagtailmenus together with wagtail-condensedinlinepanel.

`wagtail-condensedinlinepanel` offers a React-powered alternative to Wagtail's built-in `InlinePanel` with some great extra features that make it perfect for managing menu items; including drag-and-drop reordering and the ability to add new items at any position.

If you'd like to give it a try, follow the installation instructions below, and wagtailmenus will automatically use the app's `CollapsedInlinePanel` class.

1. Install the package using pip.

   If your project uses Wagtail `2.0` or later, run:

   ```
   pip install wagtail-condensedinlinepanel==0.5.0
   ```

   Otherwise, run:

   ```
   pip install wagtail-condensedinlinepanel==0.4.2
   ```

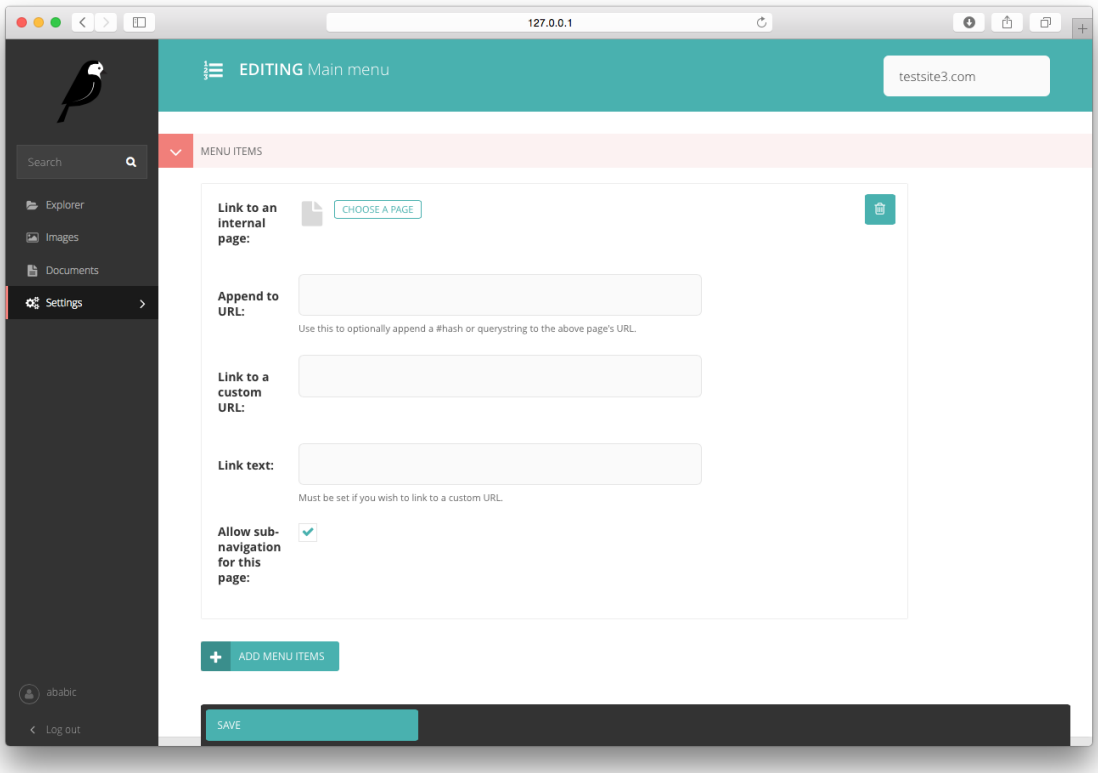
2. Add `condensedinlinepanel` to the `INSTALLED_APPS` setting in your project settings:

```
INSTALLED_APPS = [
    ...
    'condensedinlinepanel',
    ...
]
```

Note: If for some reason you want to use `wagtail-condensedinlinepanel` for other things, but would prefer NOT to use it for editing menus, you can make wagtailmenus revert to using standard `InlinePanel` by adding `WAGTAILMENUS_USE_CONDENSEDINLINEPANEL = False` to your project settings.

1.3 Managing main menus via the CMS

1. Log into the Wagtail CMS for your project (as a superuser).
2. Click on **Settings** in the side menu, then select **Main menu** from the options that appear.
3. You'll be automatically redirected to the an edit page for the current site (or the 'default' site, if the current site cannot be identified). For multi-site projects, a 'site switcher' will appear in the top right, allowing you to edit main menus for each site.



4. Use the **MENU ITEMS** inline panel to define the root-level items. If you wish, you can use the **handle** field to specify an additional value for each item, which you'll be able to access in a custom main menu template.

Note: Even if selected as menu items, pages must be 'live' and have a `show_in_menus` value of `True` in

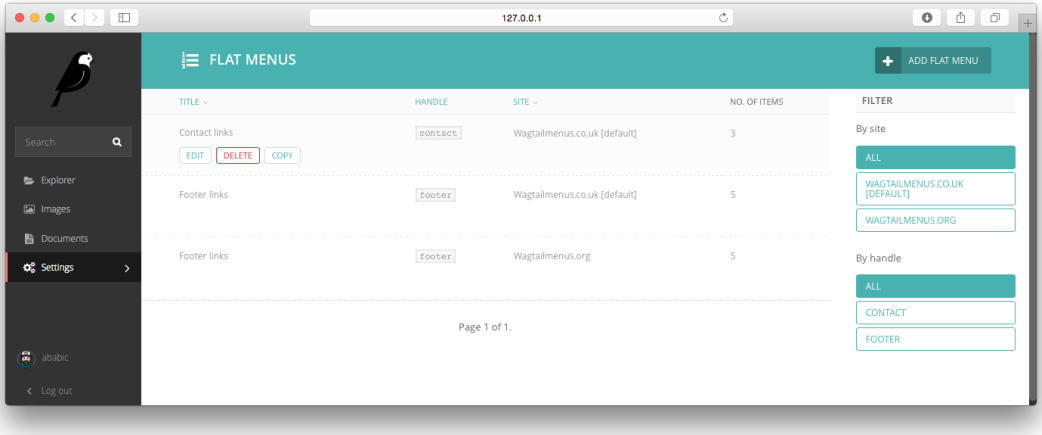
order to appear in menus. If you're expecting to see new page links in a menu, but the pages are not showing up, edit the page and check whether the "Show in menus" checkbox is checked (found under the "Promote" tab by default).

5. At the very bottom of the form, you'll find the **ADVANCED SETTINGS** panel, which is collapsed by default. Click on the arrow icon next to the heading to reveal the **Maximum levels** and **Specific usage** fields, which you can alter to fit the needs of your project. For more information about specific usage see *'Specific' pages and menus*.
6. Click on the **Save** button at the bottom of the page to save your changes.

1.4 Managing flat menus via the CMS

1.4.1 The flat menu list

All of the flat menus created for a project will appear in the menu list page, making it easy to find, update, copy or delete your menus later. As soon as you create menus for more than one site in a multi-site project, the listing page will give you additional information and filters to help manage your menus:

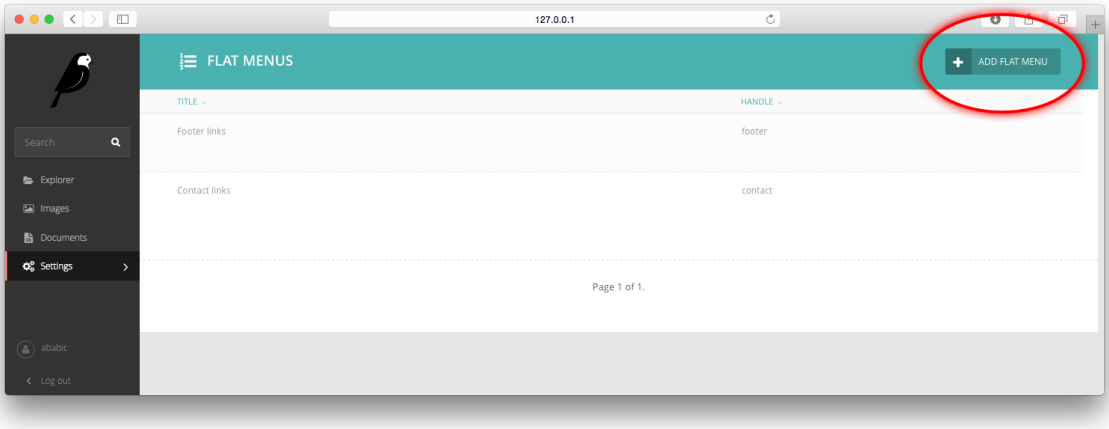


To access the flat menu list, do the following:

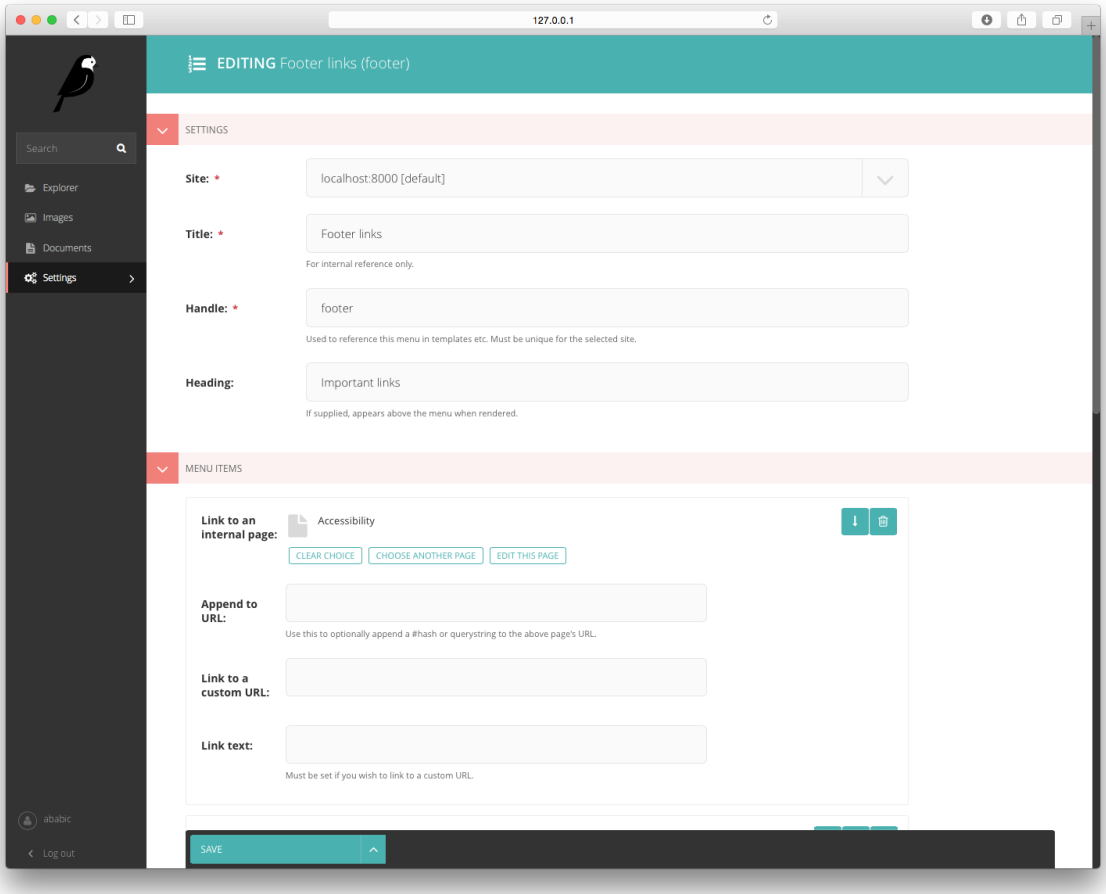
1. Log into the Wagtail CMS for your project (as a superuser).
2. Click on "Settings" in the side menu, then on "Flat menus".

1.4.2 Adding a new flat menu

1. From the listing page above, click the "Add flat menu" button



2. Fill out the form, choosing a unique-for-site "handle", which you'll use to reference the menu when using the `{% flat_menu %}` tag.



Note: If you know in advance what menus you're likely to have in your project, you can define some pre-set choices for the `handle` field using the *WAGTAILMENUS_FLAT_MENUS_HANDLE_CHOICES* setting. When used, the `handle` field will become a select field, saving you from having to enter values manually.

3. Use the "MENU ITEMS" inline panel to define the links you want the menu to have. If you wish, you can use the **handle** field to specify an additional value for each item, which you'll be able to access from within menu templates.

   Note: Even if selected as menu items, pages must be 'live' and have a `show_in_menus` value of `True` in order to appear in menus. If you're expecting to see new page links in a menu, but the pages are not showing up, edit the page and check whether the "Show in menus" checkbox is checked (found under the "Promote" tab by default).

   ---

4. At the very bottom of the form, you'll find the "ADVANCED SETTINGS" panel, which is collapsed by default. Click on the arrow icon next to the heading to reveal the **Maximum levels** and **Specific usage** fields, which you can alter to fit the needs of your project. For more information about usage specific pages in menus, see *'Specific' pages and menus*
5. Click on the **Save** button at the bottom of the page to save your changes.

1.5 Rendering menus

1.5.1 Template tags reference

The `main_menu` tag

The `main_menu` tag allows you to display the `MainMenu` defined for the current site in your Wagtail project, with CSS classes automatically applied to each item to indicate the current page or ancestors of the current page. It also does a few sensible things, like never adding the 'ancestor' class for a homepage link, or outputting children for it.

Example usage

```
{% load menu_tags %}

{% main_menu max_levels=3 use_specific=USE_SPECIFIC_TOP_LEVEL template="menus/custom_
↪main_menu.html" sub_menu_template="menus/custom_sub_menu.html" %}
```

Supported arguments

- *show_multiple_levels*
- *max_levels*
- *use_specific*
- *allow_repeating_parents*
- *apply_active_classes*
- *use_absolute_page_urls*
- *add_sub_menus_inline*
- *template*
- *sub_menu_template*
- *sub_menu_templates*

show_multiple_levels

Required?	Expected value type	Default value
No	`bool`	`True`

Adding `show_multiple_levels=False` to the tag in your template is essentially a more descriptive way of adding `max_levels` to `1`.

---

max_levels

Required?	Expected value type	Default value
No	`int`	`None`

Provide an integer value to override the `max_levels` field value defined on your menu. Controls how many levels should be rendered (when `show_multiple_levels` is `True`).

---

use_specific

Required?	Expected value type	Default value
No	`int` (see *Supported values for fetching specific pages*)	`None`

Provide a value to override the `use_specific` field value defined on your main menu. Allows you to control how wagtailmenus makes use of `PageQuerySet.specific()` and `Page.specific` when rendering the menu. For more information and examples, see: *Using the use_specific template tag argument*.

---

allow_repeating_parents

Required?	Expected value type	Default value
No	`bool`	`True`

Item repetition settings set on each page are respected by default, but you can add `allow_repeating_parents=False` to ignore them, and not repeat any pages in sub-menus when rendering multiple levels.

---

apply_active_classes

Required?	Expected value type	Default value
No	`bool`	`True`

The tag will attempt to add 'active' and 'ancestor' CSS classes to the menu items (where applicable) to indicate the active page and ancestors of that page. To disable this behaviour, add `apply_active_classes=False` to the tag in your template.

You can change the CSS class strings used to indicate 'active' and 'ancestor' statuses by utilising the *WAGTAILMENUS_ACTIVE_CLASS* and *WAGTAILMENUS_ACTIVE_ANCESTOR_CLASS* settings.

---

use_absolute_page_urls

Required?	Expected value type	Default value
No	`bool`	`False`

By default, relative page URLs are used for the `href` attribute on page links when rendering your menu. If you wish to use absolute page URLs instead, add `use_absolute_page_urls=True` to the `main_menu` tag in your template. The preference will also be respected automatically by any subsequent calls to `{% sub_menu %}` during the course of rendering the menu (unless explicitly overridden in custom menu templates).

---

add_sub_menus_inline

New in version 2.12.

Required?	Expected value type	Default value
No	`bool`	`False`

By default, you have to call the `{% sub_menu %}` tag within a menu template to render new branches of a multi-level menu. However, if you add `add_sub_menus_inline=True` to the initial `{% main_menu %}` tag call, then sub menus will be added directly to any menu item where *item.has_children_in_menu* is `True`, allowing you to render them directly, without having to use the template tag.

For example, instead of the following:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {% sub_menu item %}
        {% endif %}
    </li>
{% endfor %}
```

You could do:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {{ item.sub_menu.render_to_template }}
        {% endif %}
    </li>
{% endfor %}
```

---

template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Lets you render the menu to a template of your choosing. If not provided, wagtailmenus will attempt to find a suitable template automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths this tag, see: *Preferred template paths for {% main_menu %}*.

---

sub_menu_template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Lets you specify a template to be used for rendering sub menus. All subsequent calls to `{% sub_menu %}` within the context of the section menu will use this template unless overridden by providing a `template` value to `{% sub_menu %}` in a menu template. If not provided, wagtailmenus will attempt to find a suitable template automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths this tag, see: *Preferred template paths for {% main_menu %}*.

---

sub_menu_templates

Required?	Expected value type	Default value
No	Comma separated template paths (`str`)	`''`

Allows you to specify multiple templates to use for rendering different levels of sub menu. In the following example, `"level_1.html"` would be used to render the first level of the menu, then subsequent calls to `{% sub_menu %}` would use `"level_2.html"` to render any second level menu items, or `"level_3.html"` for and third level menu items.

```
{% main_menu max_levels=3 template="level_1.html" sub_menu_templates="level_2.html,␣
↪level_3.html" %}
```

If not provided, wagtailmenus will attempt to find suitable sub menu templates automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths for this tag, see: *Preferred template paths for {% main_menu %}*.

The `flat_menu` tag

Example usage

```
{% load menu_tags %}

{% flat_menu 'footer' max_levels=1 show_menu_heading=False  use_specific=USE_SPECIFIC_
↪TOP_LEVEL  fall_back_to_default_site_menus=True %}
```

Supported arguments

- *handle*

handle

Required?	Expected value type	Default value
Yes	`str`	`None`

The unique handle for the flat menu you want to render, e.g. `'info'`, `'contact'`, or `'services'`. You don't need to include the `handle` key if supplying as the first argument to the tag (you can just do `{% flat_menu 'menu_handle' %}`).

---

show_menu_heading

Required?	Expected value type	Default value
No	`bool`	`True`

Passed through to the template used for rendering, where it can be used to conditionally display a heading above the menu.

---

show_multiple_levels

Required?	Expected value type	Default value
No	`bool`	`True`

Flat menus are designed for outputting simple, flat lists of links. But, you can alter the `max_levels` field value on your `FlatMenu` objects in the CMS to enable multi-level output for specific menus. If you want to absolutely

never show anything but the `MenuItem` objects defined on the menu, you can override this behaviour by adding `show_multiple_levels=False` to the tag in your template.

---

max_levels

Required?	Expected value type	Default value
No	`int`	`None`

Provide an integer value to override the `max_levels` field value defined on your menu. Controls how many levels should be rendered (when `show_multiple_levels` is `True`).

---

use_specific

Required?	Expected value type	Default value
No	`int` (see *Supported values for fetching specific pages*)	`None`

Provide a value to override the `use_specific` field value defined on your flat menu. Allows you to control how wagtailmenus makes use of `PageQuerySet.specific()` and `Page.specific` when rendering the menu.

For more information and examples, see: *Using the use_specific template tag argument*.

---

apply_active_classes

Required?	Expected value type	Default value
No	`bool`	`False`

Unlike `main_menu` and `section_menu`, `flat_menu` will NOT attempt to add `'active'` and `'ancestor'` classes to the menu items by default, as this is often not useful. You can override this by adding `apply_active_classes=true` to the tag in your template.

You can change the CSS class strings used to indicate 'active' and 'ancestor' statuses by utilising the *WAGTAILMENUS_ACTIVE_CLASS* and *WAGTAILMENUS_ACTIVE_ANCESTOR_CLASS* settings.

---

allow_repeating_parents

Required?	Expected value type	Default value
No	`bool`	`True`

Repetition-related settings on your pages are respected by default, but you can add `allow_repeating_parents=False` to ignore them, and not repeat any pages in sub-menus when rendering. Please note that using this option will only have an effect if `use_specific` has a value of `1` or higher.

---

fall_back_to_default_site_menus

Required?	Expected value type	Default value
No	`bool`	`False`

When using the `flat_menu` tag, wagtailmenus identifies the 'current site', and attempts to find a menu for that site, matching the `handle` provided. By default, if no menu is found for the current site, nothing is rendered. However, if `fall_back_to_default_site_menus=True` is provided, wagtailmenus will search search the 'default' site (In the CMS, this will be the site with the '**Is default site**' checkbox ticked) for a menu with the same handle, and use that instead before giving up.

The default value can be changed to `True` by utilising the *WAGTAILMENUS_FLAT_MENUS_FALL_BACK_TO_DEFAULT_SITE_MENUS* setting.

---

add_sub_menus_inline

New in version 2.12.

Required?	Expected value type	Default value
No	`bool`	`False`

By default, you have to call the `{% sub_menu %}` tag within a menu template to render new branches of a multi-level menu. However, if you add `add_sub_menus_inline=True` to the initial `{% flat_menu %}` tag call, then sub menus will be added directly to any menu item where *item.has_children_in_menu* is `True`, allowing you to render them directly, without having to use the template tag.

For example, instead of the following:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {% sub_menu item %}
        {% endif %}
    </li>
{% endfor %}
```

You could do:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
```

(continues on next page)

(continued from previous page)

```
            {{ item.sub_menu.render_to_template }}
        {% endif %}
    </li>
{% endfor %}
```

template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Lets you render the menu to a template of your choosing. If not provided, wagtailmenus will attempt to find a suitable template automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths this tag, see: *Preferred template paths for {% flat_menu %}*.

use_absolute_page_urls

Required?	Expected value type	Default value
No	`bool`	`False`

By default, relative page URLs are used for the `href` attribute on page links when rendering your menu. If you wish to use absolute page URLs instead, add `use_absolute_page_urls=True` to the `{% flat_menu %}` tag in your template. The preference will also be respected automatically by any subsequent calls to `{% sub_menu %}` during the course of rendering the menu (unless explicitly overridden in custom menu templates).

sub_menu_template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Lets you specify a template to be used for rendering sub menus (if enabled using `show_multiple_levels`). All subsequent calls to `{% sub_menu %}` within the context of the flat menu will use this template unless overridden by providing a `template` value to `{% sub_menu %}` directly in a menu template. If not provided, wagtailmenus will attempt to find a suitable template automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths this tag, see: *Preferred template paths for {% flat_menu %}*.

sub_menu_templates

Required?	Expected value type	Default value
No	Comma separated template paths (`str`)	`''`

Allows you to specify multiple templates to use for rendering different levels of sub menu. In the following example, `"level_1.html"` would be used to render the first level of the menu, then subsequent calls to `{% sub_menu %}` would use `"level_2.html"` to render any second level menu items, or `"level_3.html"` for and third level (or greater) menu items.

```
{% flat_menu 'info' template="level_1.html" sub_menu_templates="level_2.html, level_3.
↪html" %}
```

If not provided, wagtailmenus will attempt to find suitable sub menu templates automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths for this tag, see: *Preferred template paths for {% flat_menu %}*.

The `section_menu` tag

The `section_menu` tag allows you to display a context-aware, page-driven menu in your project's templates, with CSS classes automatically applied to each item to indicate the active page or ancestors of the active page.

Example usage

```
{% load menu_tags %}

{% section_menu max_levels=3 use_specific=USE_SPECIFIC_OFF template="menus/custom_
↪section_menu.html" sub_menu_template="menus/custom_section_sub_menu.html" %}
```

Supported arguments

- *show_section_root*
- *max_levels*
- *use_specific*
- *show_multiple_levels*
- *apply_active_classes*
- *allow_repeating_parents*
- *use_absolute_page_urls*
- *add_sub_menus_inline*
- *template*

- *sub_menu_template*
- *sub_menu_templates*

show_section_root

Required?	Expected value type	Default value
No	`bool`	`True`

Passed through to the template used for rendering, where it can be used to conditionally display the root page of the current section.

---

max_levels

Required?	Expected value type	Default value
No	`int`	`2`

Lets you control how many levels of pages should be rendered (the section root page does not count as a level, just the first set of pages below it). If you only want to display the first level of pages below the section root page (whether pages linked to have children or not), add `max_levels=1` to the tag in your template. You can display additional levels by providing a higher value.

The default value can be changed by utilising the *WAGTAILMENUS_DEFAULT_SECTION_MENU_MAX_LEVELS* setting.

---

use_specific

Required?	Expected value type	Default value
No	`int` (see *Supported values for fetching specific pages*)	`1` (Auto)

Allows you to control how wagtailmenus makes use of `PageQuerySet.specific()` and `Page.specific` when rendering the menu, helping you to find the right balance between functionality and performance.

For more information and examples, see: *Using the use_specific template tag argument*.

The default value can be altered by utilising the *WAGTAILMENUS_DEFAULT_SECTION_MENU_USE_SPECIFIC* setting.

---

show_multiple_levels

Required?	Expected value type	Default value
No	`bool`	`True`

Adding `show_multiple_levels=False` to the tag in your template essentially overrides `max_levels` to `1`. It's just a little more descriptive.

---

apply_active_classes

Required?	Expected value type	Default value
No	`bool`	`True`

The tag will add 'active' and 'ancestor' classes to the menu items where applicable, to indicate the active page and ancestors of that page. To disable this behaviour, add `apply_active_classes=False` to the tag in your template.

You can change the CSS class strings used to indicate 'active' and 'ancestor' statuses by utilising the *WAGTAILMENUS_ACTIVE_CLASS* and *WAGTAILMENUS_ACTIVE_ANCESTOR_CLASS* settings.

---

allow_repeating_parents

Required?	Expected value type	Default value
No	`bool`	`True`

Repetition-related settings on your pages are respected by default, but you can add `allow_repeating_parents=False` to ignore them, and not repeat any pages in sub-menus when rendering. Please note that using this option will only have an effect if `use_specific` has a value of `1` or higher.

---

use_absolute_page_urls

Required?	Expected value type	Default value
No	`bool`	`False`

By default, relative page URLs are used for the `href` attribute on page links when rendering your menu. If you wish to use absolute page URLs instead, add `use_absolute_page_urls=True` to the `{% section_menu %}` tag in your template. The preference will also be respected automatically by any subsequent calls to `{% sub_menu %}` during the course of rendering the menu (unless explicitly overridden in custom menu templates).

---

add_sub_menus_inline

New in version 2.12.

Required?	Expected value type	Default value
No	`bool`	`False`

By default, you have to call the `{% sub_menu %}` tag within a menu template to render new branches of a multi-level menu. However, if you add `add_sub_menus_inline=True` to the initial `{% section_menu %}` tag call, then sub menus will be added directly to any menu item where *item.has_children_in_menu* is `True`, allowing you to render them directly, without having to use the template tag.

For example, instead of the following:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {% sub_menu item %}
        {% endif %}
    </li>
{% endfor %}
```

You could do:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {{ item.sub_menu.render_to_template }}
        {% endif %}
    </li>
{% endfor %}
```

---

template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Lets you render the menu to a template of your choosing. If not provided, wagtailmenus will attempt to find a suitable template automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths this tag, see: *Preferred template paths for {% section_menu %}*.

---

sub_menu_template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Lets you specify a template to be used for rendering sub menus. All subsequent calls to `{% sub_menu %}` within the context of the section menu will use this template unless overridden by providing a `template` value to `{% sub_menu %}` in a menu template. If not provided, wagtailmenus will attempt to find a suitable template automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths this tag, see: *Preferred template paths for {% section_menu %}*.

---

sub_menu_templates

Required?	Expected value type	Default value
No	Comma separated template paths (`str`)	`''`

Allows you to specify multiple templates to use for rendering different levels of sub menu. In the following example, `"level_1.html"` would be used to render the first level of the menu, then subsequent calls to `{% sub_menu %}` would use `"level_2.html"` to render any second level menu items, or `"level_3.html"` for and third level (or greater) menu items.

```
{% section_menu max_levels=3 template="level_1.html" sub_menu_templates="level_2.html,
↪ level_3.html" %}
```

If not provided, wagtailmenus will attempt to find suitable sub menu templates automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths for this tag, see: *Preferred template paths for {% section_menu %}*.

The `children_menu` tag

The `children_menu` tag can be used in page templates to display a menu of children of the current page. You can also use the *parent_page* argument to show children of a different page.

Example usage

```
{% load menu_tags %}

{% children_menu some_other_page max_levels=2 use_specific=USE_SPECIFIC_OFF template=
↪"menus/custom_children_menu.html" sub_menu_template="menus/custom_children_sub_menu.
↪html" %}
```

Supported arguments

parent_page

Required?	Expected value type	Default value
No	A `Page` object	`None`

Allows you to specify a page to output children for. If no alternate page is specified, the tag will automatically use `self` from the context to render children pages for the current/active page.

---

max_levels

Required?	Expected value type	Default value
No	`int`	`1`

Allows you to specify how many levels of pages should be rendered. For example, if you want to display the direct children pages and their children too, add `max_levels=2` to the tag in your template.

The default value can be changed by utilising the *WAGTAILMENUS_DEFAULT_CHILDREN_MENU_MAX_LEVELS* setting.

---

use_specific

Required?	Expected value type	Default value
No	`int` (see *Supported values for fetching specific pages*)	`1` (Auto)

Allows you to specify how wagtailmenus makes use of `PageQuerySet.specific()` and `Page.specific` when rendering the menu.

For more information and examples, see: *Using the use_specific template tag argument*.

The default value can be altered by adding a *WAGTAILMENUS_DEFAULT_CHILDREN_MENU_USE_SPECIFIC* setting to your project's settings.

---

apply_active_classes

Required?	Expected value type	Default value
No	`bool`	`False`

Unlike `main_menu` and *section_menu*', `children_menu` will NOT attempt to add `'active'` and `'ancestor'` classes to the menu items by default, as this is often not useful. You can override this by adding `apply_active_classes=true` to the tag in your template.

You can change the CSS class strings used to indicate 'active' and 'ancestor' statuses by utilising the *WAGTAILMENUS_ACTIVE_CLASS* and *WAGTAILMENUS_ACTIVE_ANCESTOR_CLASS* settings.

---

allow_repeating_parents

Required?	Expected value type	Default value
No	`bool`	`True`

Repetition-related settings on your pages are respected by default, but you can add `allow_repeating_parents=False` to ignore them, and not repeat any pages in sub-menus when rendering. Please note that using this option will only have an effect if `use_specific` has a value of `1` or higher.

---

use_absolute_page_urls

Required?	Expected value type	Default value
No	`bool`	`False`

By default, relative page URLs are used for the `href` attribute on page links when rendering your menu. If you wish to use absolute page URLs instead, add `use_absolute_page_urls=True` to the `{% children_menu %}` tag in your template. The preference will also be respected automatically by any subsequent calls to `{% sub_menu %}` during the course of rendering the menu (unless explicitly overridden in custom menu templates).

---

add_sub_menus_inline

New in version 2.12.

Required?	Expected value type	Default value
No	`bool`	`False`

By default, you have to call the `{% sub_menu %}` tag within a menu template to render new branches of a multi-level menu. However, if you add `add_sub_menus_inline=True` to the initial `{% children_menu %}` tag call, then sub menus will be added directly to any menu item where *item.has_children_in_menu* is `True`, allowing you to render them directly, without having to use the template tag.

For example, instead of the following:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {% sub_menu item %}
        {% endif %}
    </li>
{% endfor %}
```

You could do:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {{ item.sub_menu.render_to_template }}
        {% endif %}
    </li>
{% endfor %}
```

---

template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Lets you render the menu to a template of your choosing. If not provided, wagtailmenus will attempt to find a suitable template automatically (see below for more details).

For more information about overriding templates, see: *Using your own menu templates*

For a list of preferred template paths this tag, see: *Preferred template paths for {% children_menu %}*

---

sub_menu_template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Lets you specify a template to be used for rendering sub menus. All subsequent calls to `{% sub_menu %}` within the context of the section menu will use this template unless overridden by providing a `template` value to `{% sub_menu %}` in a menu template. If not provided, wagtailmenus will attempt to find a suitable template automatically

For more information about overriding templates, see: *Using your own menu templates*

For a list of preferred template paths this tag, see: *Preferred template paths for {% children_menu %}*

---

sub_menu_templates

Required?	Expected value type	Default value
No	Comma separated template paths (`str`)	`''`

Allows you to specify multiple templates to use for rendering different levels of sub menu. In the following example, `"level_1.html"` would be used to render the first level of the menu, then subsequent calls to `{% sub_menu %}` would use `"level_2.html"` to render any second level menu items, or `"level_3.html"` for and third level (or greater) menu items.

```
{% children_menu max_levels=3 template="level_1.html" sub_menu_templates="level_2.
↪html, level_3.html" %}
```

If not provided, wagtailmenus will attempt to find suitable sub menu templates automatically.

For more information about overriding templates, see: *Using your own menu templates*.

For a list of preferred template paths for this tag, see: *Preferred template paths for {% children_menu %}*.

The `sub_menu` tag

The `sub_menu` tag is used within menu templates to render additional levels of pages within a menu. It's designed to pick up on variables added to the context by the other menu tags, and so can behave a little unpredictably if called directly, without those context variables having been set. It requires only one parameter to work, which is `menuitem_or_page`.

Example usage

```
{% load menu_tags %}

{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
```

(continues on next page)

(continued from previous page)

```
        {% if item.has_children_in_menu %}
            {% sub_menu item %}
        {% endif %}
    </li>
{% endfor %}
```

Supported arguments

menuitem_or_page

Required?	Expected value type	Default value
Yes	An item from the `menu_items` list	`None` (inherit from original tag)

When iterating through a list of `menu_items` within a menu template, the current item must be passed to `{% sub_menu %}` so that it knows which page to render a sub-menu for. You don't need to include the `menuitem_or_page` key if supplying the value as the first argument to the tag (you can just do `{% sub_menu item %}`).

---

apply_active_classes

Required?	Expected value type	Default value
No	`bool`	`None` (inherit from original tag)

Allows you to override the value set by the original tag by adding an alternative value to the `{% sub_menu %}` tag in a custom menu template.

---

allow_repeating_parents

Required?	Expected value type	Default value
No	`bool`	`None` (inherit from original tag)

Allows you to override the value set by the original tag by adding an alternative value to the `{% sub_menu %}` tag in a custom menu template.

---

use_specific

Required?	Expected value type	Default value
No	`int` (see *Supported values for fetching specific pages*)	`None`

Allows you to override the value set on the original tag by adding an alternative value to the `{% sub_menu %}` tag in a custom menu template.

For more information and examples, see: *Using the use_specific template tag argument*.

---

use_absolute_page_urls

Required?	Expected value type	Default value
No	`bool`	`None` (inherit from original tag)

Allows you to override the value set on the original tag by explicitly adding `use_absolute_page_urls=True` or `use_absolute_page_urls=False` to a `{% sub_menu %}` tag in a custom menu template.

If `True`, absolute page URLs will be used for the `href` attributes on page links instead of relative URLs.

---

template

Required?	Expected value type	Default value
No	Template path (`str`)	`''`

Allows you to override the template set by the original menu tag (`sub_menu_template` in the context) by passing a fixed template path to the `{% sub_menu %}` tag in a custom menu template.

For more information about overriding templates, see: *Using your own menu templates*

1.5.2 Using your own menu templates

Writing custom menu templates

What context variables are available to use?

The following variables are added to the context by all included template tags, which you can make use of in your templates:

`menu_items` If the template is for rendering the first level of a main or flat menu, then `menu_items` will be a list of `MainMenuItem` or `FlatMenuItem` objects (respectively). In all other cases. it will be a list `Page` objects.

In either case, wagtailmenus adds a number of additional attributes to each item to help keep you menu templates consistent. For more information see: *Attributes added to each item in menu_items*

`current_level` An integer indicating the current level being rendered. This starts at `1` for the initial template tag call, then increments by one for each additional `<ul>` level that is added by calling the `{% sub_menu %}` tag

`max_levels` An integer indicating the maximum number of levels that should be rendered for the current menu, as determined by the original `main_menu`, `section_menu`, `flat_menu` or `children_menu` tag call.

`current_template` The name of the template currently being used for rendering. This is most useful when rendering a `sub_menu` template that calls `sub_menu` recursively, and you wish to use the same template for all recursions.

`sub_menu_template` The name of the template that should be used for rendering any further levels (should be picked up automatically by the `sub_menu` tag).

`original_menu_tag` A string value indicating the name of tag was originally called in order to render the branch currently being rendered. The value will be one of `"main_menu"`, `"flat_menu"`, `"section_menu"`, `"children_menu"` or `"sub_menu"`.

`allow_repeating_parents` A boolean indicating whether `MenuPage` fields should be respected when rendering further menu levels.

`apply_active_classes` A boolean indicating whether `sub_menu` tags should attempt to add 'active' and 'ancestor' classes to menu items when rendering further menu levels.

`use_absolute_page_urls` A boolean indicating whether absolute page URLs should be used for page links when rendering.

Attributes added to each item in `menu_items`

Whether `menu_items` is a list of `Page`, `MainMenuItem` or `FlatMenuItem` objects, the following additional attributes are added to each item to help improve consistency of menu templates:

`href` The URL that the menu item should link to.

`text` The text that should be used for the menu item.

You can change the field or attribute used to populate the `text` attribute by utilising the *WAGTAILMENUS_PAGE_FIELD_FOR_MENU_ITEM_TEXT* setting.

`active_class` A class name to indicate the 'active' state of the menu item. The value will be 'active' if linking to the current page, or 'ancestor' if linking to one of it's ancestors.

You can change the CSS class strings used to indicate 'active' and 'ancestor' statuses by utilising the *WAGTAILMENUS_ACTIVE_CLASS* and *WAGTAILMENUS_ACTIVE_ANCESTOR_CLASS* settings.

has_children_in_menu A boolean indicating whether the menu item has children that should be output as a sub-menu.

Getting wagtailmenus to use your custom menu templates

Using preferred paths and names for your templates

This is the easiest (and recommended) approach for getting wagtailmenus to use your custom menu templates for rendering.

When you do not specify templates to use using the `template`, `sub_menu_template`, or `sub_menu_templates` arguments template tag arguments, wagtailmenus looks in a list of gradually less specific paths until it finds an appropriate template to use. If you're familiar with Django, you'll probably already be familiar with this concept. Essentially, you can easily override the default menu templates by putting your custom templates in a preferred location within your project.

The following sections outline the preferred template paths for each tag, in the order that they are searched for (most specific first).

Preferred template paths for `{% main_menu %}`

Note: Template paths marked with an asterisk (*) will only be searched if you have set the *WAGTAILMENUS_SITE_SPECIFIC_TEMPLATE_DIRS* setting to `True` for your project.

For the first/top level menu items:

1. `"menus/{{ current_site.domain }}/main/level_1.html"` *
2. `"menus/{{ current_site.domain }}/main/menu.html"` *
3. `"menus/{{ current_site.domain }}/main_menu.html"` *
4. `"menus/main/level_1.html"`
5. `"menus/main/menu.html"`
6. `"menus/main_menu.html"`

For any sub-menus:

1. `"menus/{{ current_site.domain }}/level_{{ current_level }}.html"` *
2. `"menus/{{ current_site.domain }}/sub_menu.html"` *
3. `"menus/{{ current_site.domain }}/main_sub_menu.html"` *

4. "menus/{{ current_site.domain }}/sub_menu.html" *
5. "menus/main/level_{{ current_level }}.html"
6. "menus/main/sub_menu.html"
7. "menus/main_sub_menu.html"
8. "menus/sub_menu.html"

Examples

For a multi-level main menu that displays three levels of links, your templates directory might look like this:

```
templates
└── menus
    └── main
        ├── level_1.html  # Used by {% main_menu %} for the 1st level
        ├── level_2.html  # Used by {% sub_menu %} for the 2nd level
        └── level_3.html  # Used by {% sub_menu %} for the 3rd level
```

Preferred template paths for `{% flat_menu %}`

For flat menus, the tag also uses the *handle* field of the specific menu being rendered, so that you can have wagtailmenus use different templates for different menus.

Note: Template paths marked with an asterisk (*) are only searched if you have set the *WAGTAILMENUS_SITE_SPECIFIC_TEMPLATE_DIRS* setting to `True` for your project.

For the first/top level menu items:

1. "menus/{{ current_site.domain }}/flat/{{ menu.handle }}/level_1.html" *
2. "menus/{{ current_site.domain }}/flat/{{ menu.handle }}/menu.html" *
3. "menus/{{ current_site.domain }}/flat/{{ menu.handle }}.html" *
4. "menus/{{ current_site.domain }}/{{ menu.handle }}/level_1.html" *
5. "menus/{{ current_site.domain }}/{{ menu.handle }}/menu.html" *
6. "menus/{{ current_site.domain }}/{{ menu.handle }}.html" *
7. "menus/{{ current_site.domain }}/flat/level_1.html" *
8. "menus/{{ current_site.domain }}/flat/default.html" *
9. "menus/{{ current_site.domain }}/flat/menu.html" *
10. "menus/{{ current_site.domain }}/flat_menu.html" *
11. "menus/flat/{{ menu.handle }}/level_1.html"
12. "menus/flat/{{ menu.handle }}/menu.html"
13. "menus/flat/{{ menu.handle }}.html"
14. "menus/{{ menu.handle }}/level_1.html"
15. "menus/{{ menu.handle }}/menu.html"
16. "menus/{{ menu.handle }}.html"
17. "menus/flat/level_1.html"

18. "menus/flat/default.html"
19. "menus/flat/menu.html"
20. "menus/flat_menu.html"

For any sub-menus:

1. "menus/{{ current_site.domain }}/flat/{{ menu.handle }}/level_{{ current_level }}.html" *
2. "menus/{{ current_site.domain }}/flat/{{ menu.handle }}/sub_menu.html" *
3. "menus/{{ current_site.domain }}/flat/{{ menu.handle }}_sub_menu.html" *
4. "menus/{{ current_site.domain }}/{{ menu.handle }}/level_{{ current_level }}.html" *
5. "menus/{{ current_site.domain }}/{{ menu.handle }}/sub_menu.html" *
6. "menus/{{ current_site.domain }}/{{ menu.handle }}_sub_menu.html" *
7. "menus/{{ current_site.domain }}/flat/level_{{ current_level }}.html" *
8. "menus/{{ current_site.domain }}/flat/sub_menu.html" *
9. "menus/{{ current_site.domain }}/sub_menu.html" *
10. "menus/flat/{{ menu.handle }}/level_{{ current_level }}.html"
11. "menus/flat/{{ menu.handle }}/sub_menu.html"
12. "menus/flat/{{ menu.handle }}_sub_menu.html"
13. "menus/{{ menu.handle }}/level_{{ current_level }}.html"
14. "menus/{{ menu.handle }}/sub_menu.html"
15. "menus/{{ menu.handle }}_sub_menu.html"
16. "menus/flat/level_{{ current_level }}.html"
17. "menus/flat/sub_menu.html"
18. "menus/sub_menu.html"

Examples

For a flat menu with the handle `info`, that is required to show two levels of menu items, your templates directory might look like this:

```
templates
└── menus
    └── info
        ├── level_1.html  # Used by the {% flat_menu %} tag for the 1st level
        └── level_2.html  # Used by the {% sub_menu %} tag for the 2nd level
```

Or, if the `info` menu only ever needed to show one level of menu items, you might prefer to keep things simple, like so:

```
templates
└── menus
    └── info.html
```

If your were happy for most of your flat menus to share the same templates, you might put those common templates in the same folder where they'd automatically get selected for all flat menus:

```
templates
└── menus
    └── flat
        ├── level_1.html  # Used by the {% flat_menu %} tag for the 1st level
        ├── level_2.html  # Used by the {% sub_menu %} tag for the 2nd level
        └── level_3.html  # Used by the {% sub_menu %} tag for the 3rd level
```

Building on the above example, you could then override menu templates for certain menus as required, by putting templates in a preferred location for just those menus. For example:

```
templates
└── menus
    └── flat
        ├── level_1.html
        ├── level_2.html
        ├── level_3.html
        ├── info
        |   |   # This location is preferred when rendering an 'info' menu
        |   └── level_2.html  # Only override the level 2 template
        └── contact
            |   # This location is preferred when rendering a 'contact' menu
            └── level_1.html  # Only override the level 1 template
```

With the above structure, the following templates would be used for rendering the `info` menu if three levels were needed:

1. *menus/flat/level_1.html*
2. *menus/flat/info/level_2.html*
3. *menus/flat/level_3.html*

For rendering a `contact` menu, the following templates would be used:

1. *menus/flat/contact/level_1.html*
2. *menus/flat/level_2.html*
3. *menus/flat/level_3.html*

The above structure would work, but it's not ideal. Imagine if a new front-end developer joined the team, and had no experience with wagtailmenus, or even if you came back to the project after not working with wagtailmenus for a while - It wouldn't be so easy to figure out which templates were being used by each menu. A better approach might be to do something like this:

```
templates
    └── menus
        └── flat
            |   # Still used by default (e.g. for menus with different handles)
            ├── level_1.html
            ├── level_2.html
            ├── level_3.html
            ├── info
            |   |   # This location is preferred when rendering an 'info' menu
            |   ├── level_1.html  # {% extends 'menus/flat/level_1.html' %}
            |   └── level_2.html  # Our custom template from before
            └── contact
                |   # This location is preferred when rendering a 'contact' menu
                ├── level_1.html  # Our custom template from before
                └── level_2.html  # {% extends 'menus/flat/level_2.html' %}
```

That's better, but you might even like to make the `info` and `contact` templates even easier to find, by moving those folders out to the root `menus` folder.

```
templates
    └── menus
        ├── flat
        |   |   # Still used by default (e.g. for menus with different handles)
        |   ├── level_1.html
        |   ├── level_2.html
        |   └── level_3.html
        ├── info
        |   |   # This location is still preferred when rendering an 'info' menu
        |   ├── level_1.html  # {% includes 'menus/flat/level_1.html' %}
        |   └── level_2.html  # Our custom template from before
        └── contact
            |   # This location is still preferred when rendering a 'contact' menu
            ├── level_1.html  # Our custom template from before
            └── level_2.html  # {% includes 'menus/flat/level_2.html' %}
```

The templates in the `info` and `contact` folders will still be preferred over the ones in `flat`, because the folder names are more specific.

Preferred template paths for `{% section_menu %}`

Note: Template paths marked with an asterisk (*) are only searched if you have set the *WAGTAILMENUS_SITE_SPECIFIC_TEMPLATE_DIRS* setting to `True` for your project.

For the first/top level menu items:

1. `"menus/{{ current_site.domain }}/section/level_1.html"` *
2. `"menus/{{ current_site.domain }}/section/menu.html"` *
3. `"menus/{{ current_site.domain }}/section_menu.html"` *
4. `"menus/section/level_1.html"`
5. `"menus/section/menu.html"`
6. `"menus/section_menu.html"`

For any sub-menus:

1. `"menus/{{ current_site.domain }}/section/level_{{ current_level }}.html"` *
2. `"menus/{{ current_site.domain }}/section/sub_menu.html"` *
3. `"menus/{{ current_site.domain }}/section_sub_menu.html"` *
4. `"menus/{{ current_site.domain }}/sub_menu.html"` *
5. `"menus/section/level_{{ current_level }}.html"`
6. `"menus/section/sub_menu.html"`
7. `"menus/section_sub_menu.html"`
8. `"menus/sub_menu.html"`

Examples

If your project needs a multi-level section menu, displaying three levels of links, your templates directory might look something like this:

```
templates
└── menus
    └── section
        ├── level_1.html  # Used by the {% section_menu %} tag for the 1st level
        ├── level_2.html  # Used by the {% sub_menu %} tag for the 2nd level
        └── level_3.html  # Used by the {% sub_menu %} tag for the 3rd level
```

Or, if your section menu only needs to surface the first of level of pages within a section, you might structure things more simply, like so:

```
templates
└── menus
    └── section_menu.html
```

Preferred template paths for `{% children_menu %}`

Note: Template paths marked with an asterisk (*) are only searched if you have set the *WAGTAILMENUS_SITE_SPECIFIC_TEMPLATE_DIRS* setting to `True` for your project.

For the first/top level menu items:

1. `"menus/{{ current_site.domain }}/children/level_1.html"` *
2. `"menus/{{ current_site.domain }}/children/menu.html"` *
3. `"menus/{{ current_site.domain }}/children_menu.html"` *
4. `"menus/children/level_1.html"`
5. `"menus/children/menu.html"`
6. `"menus/children_menu.html"`

For any sub-menus:

1. `"menus/{{ current_site.domain }}/children/level_{{ current_level }}.html"` *
2. `"menus/{{ current_site.domain }}/children/sub_menu.html"` *
3. `"menus/{{ current_site.domain }}/children_sub_menu.html"` *
4. `"menus/{{ current_site.domain }}/sub_menu.html"` *
5. `"menus/children/level_{{ current_level }}.html"`
6. `"menus/children/sub_menu.html"`
7. `"menus/children_sub_menu.html"`
8. `"menus/sub_menu.html"`

Examples

If your project needs multi-level children menus, displaying two levels of links, your templates directory might look something like this:

```
templates
└── menus
    └── children
        ├── level_1.html  # Used by the {% children_menu %} tag for the 1st level
        └── level_2.html  # Used by the {% sub_menu %} tag for the 2nd level
```

Or, if you only need single-level children menus, you might structure things more simply, like so:

```
templates
└── menus
    └── children_menu.html
```

Using a consistent template structure

Even if the various menus in your project tend to share a lot of common templates between them, for the sake of consistency, it might pay you to follow a 'level-specific' pattern of template definition for each menu, even if some of the templates simply use `{% extends %}` or `{% include %}` to include a common template. It'll make it much easier to identify which menu templates are being used by which menus at a later time.

Specifying menu templates using template tag parameters

All template tags included in wagtailmenus support `template`, `sub_menu_template` and `sub_menu_templates` arguments to allow you to explicitly override the templates used during rendering.

For example, if you had created the following templates in your project's root 'templates' directory:

- `"templates/custom_menus/main_menu.html"`
- `"templates/custom_menus/main_menu_sub.html"`
- `"templates/custom_menus/main_menu_sub_level_2.html"`

You could make *The main_menu tag* use those templates for rendering by specifying them in your template, like so:

```
{% main_menu max_levels=3 template="custom_menus/main_menu.html" sub_menu_templates=
↪"custom_menus/main_menu_sub.html, custom_menus/main_menu_sub_level_2.html" %}
```

Or, if you only wanted to use a single template for sub menus, you could specify that template like so:

```
{# A 'sub_menu_templates' value without commas is recognised as a single template #}
{% main_menu max_levels=3 template="custom_menus/main_menu.html" sub_menu_templates=
↪"custom_menus/main_menu_sub.html" %}

{# You can also use the 'sub_menu_template' (no plural) option, which is slightly␣
↪more verbose #}
{% main_menu max_levels=3 template="custom_menus/main_menu.html" sub_menu_template=
↪"custom_menus/main_menu_sub.html" %}
```

Or you could just override one or the other, like so:

```
{# Just override the top-level template #}
{% main_menu max_levels=3 template="custom_menus/main_menu.html" %}

{# Just override the sub menu templates #}
{% main_menu max_levels=3 sub_menu_templates="custom_menus/main_menu_sub.html, custom_
↪menus/main_menu_sub_level_2.html" %}
```

(continues on next page)

(continued from previous page)

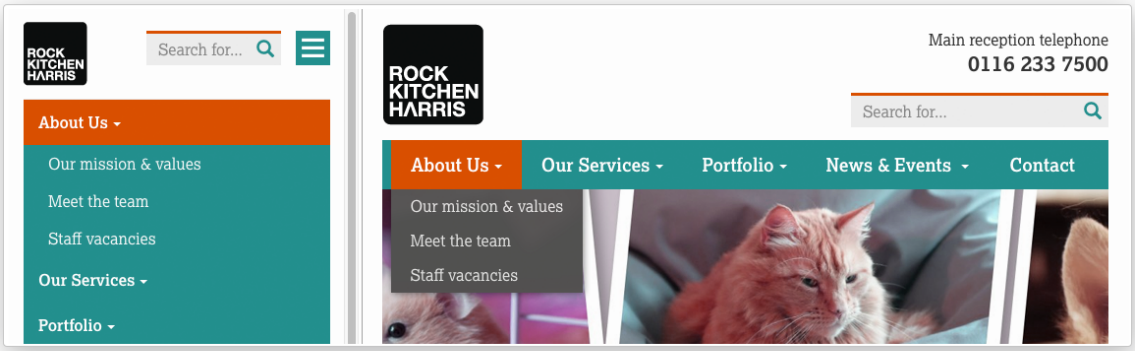
```
{# Just override the sub menu templates with a single template #}
{% main_menu max_levels=3 sub_menu_template="custom_menus/main_menu_sub.html" %}
```

1.6 The `MenuPage` and `MenuPageMixin` models

The `MenuPageMixin` and `MenuPage` models were created specifically to solve the problem of important page links becoming merely toggles in multi-level menus, preventing users from accessing them easily.

1.6.1 A typical scenario

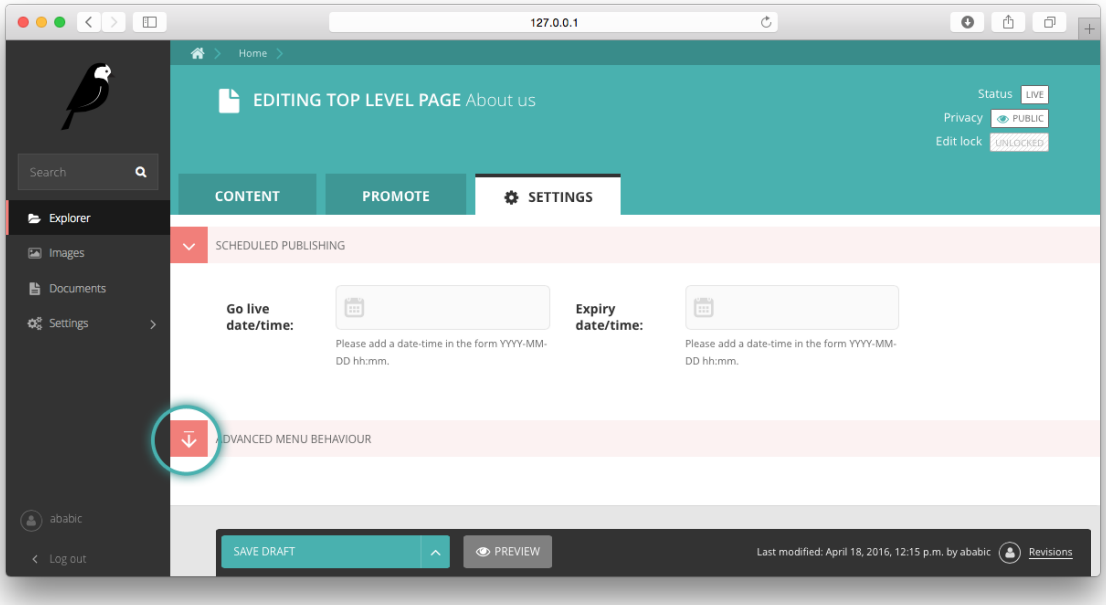
Let's say you have an **About Us** section on your site. The top-level "About Us" page has content on it that is just as important as it's children (e.g. "Meet the team", "Our mission and values", "Staff vacancies"). Because of this, you'd like visitors to be able to access the root page as easily as those pages. But, your site uses some form of collapsible multi-level navigation, and the **About Us** page link has become merely a toggle for hiding and showing its sub-pages, making it difficult to get to directly:



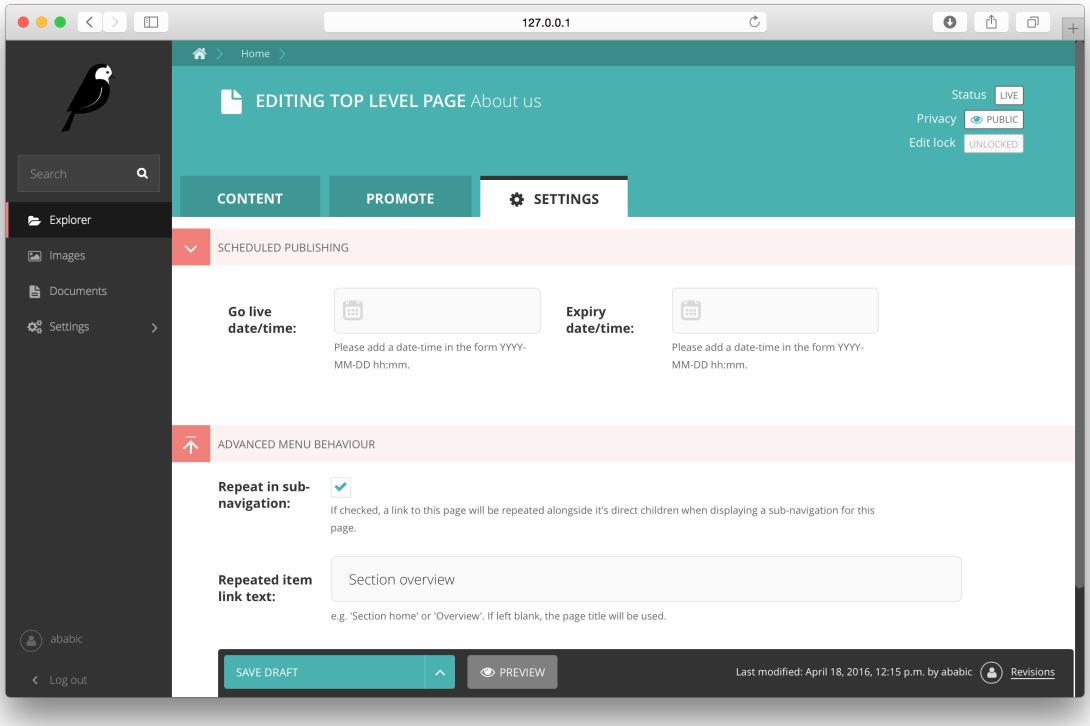
`MenuPage` to the rescue!

If the **About Us** page uses a model that subclasses `MenuPage` or `MenuPageMixin`, you can solve this issue by doing the following:

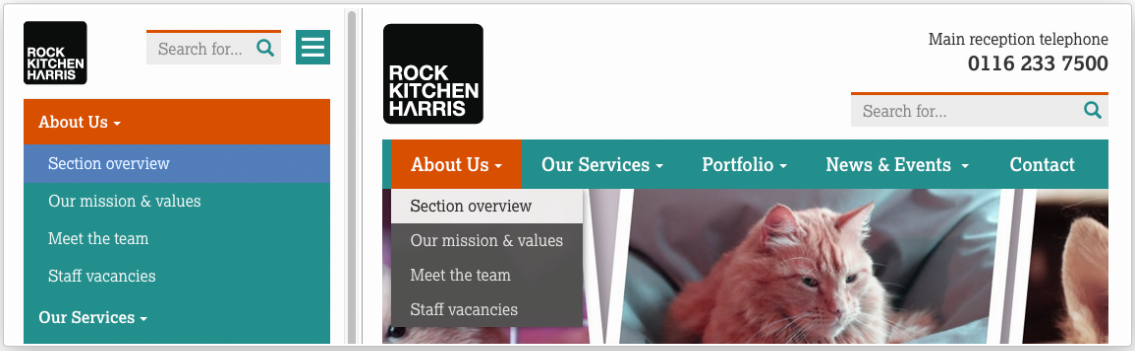
1. Edit the page via the CMS, and click on the "Settings" tab.
2. Uncollapse the "ADVANCED MENU BEHAVIOUR" panel by clicking the downward-pointing arrow next to the panel's label



3. Tick the **Repeat in sub-navigation** checkbox that appears, and publish your changes.



Now, wherever the children of the **About Us** page are output (using one of the above menu tags), an additional link will appear alongside them, allowing the that page to be accessed more easily. In the example above, you'll see **"Section overview"** has been added to the a **Repeated item link text** field. With this set, the link text for the repeated item should read **"Section overview"**, instead of just repeating the page's title, like so:



The menu tags do some extra work to ensure both links are never assigned the 'active' class. When on the 'About Us' page, the tags will treat the repeated item as the 'active' page, and just assign the 'ancestor' class to the original, so that the styling is consistent with other page links rendered at that level.

1.6.2 Implementing `MenuPage` into your project

1. Subclass `wagtailmenus.models.MenuPage` on your model instead of the usual `wagtail.wagtacore.models.Page`, just like in the following example:

```
# appname/models.py

from wagtailmenus.models import MenuPage


class GenericPage(MenuPage):
    """
    This model will gain the fields, methods and 'setting_panels' attribute
    from MenuPage.
    """
    ...
```

   Or, if you're using an abstract 'base' model in you project to improve consistency of common functionality, you could update the base model, like so:

```
# appname/models.py

from wagtailmenus.models import MenuPage


class BaseProjectPage(MenuPage):
    ...


class GenericPage(BaseProjectPage):
    ...


class ContactPage(BaseProjectPage):
    ...
```

2. If you're not overriding the `settings_panels` attribute on any of the models involved, you can skip this

step. But, if you are overriding the `settings_panels` attribute on a custom model to surface other custom fields in that tab, you'll need to include additional panels to surface the new `MenuPage` fields in the page edit interface. Wagtailmenus includes a pre-defined `menupage_panel` to make this easier, which you can use like this:

```
# appname/models.py

from wagtailmenus.models import MenuPage
from wagtailmenus.panels import menupage_panel


class GenericPage(MenuPage):
    """
    This model will gain the fields, methods and `setting_panels` attribute
    from `MenuPage`, but `settings_panels` is being overridden to include
    other fields in the `Settings` tab.
    """

    custom_settings_field_one = BooleanField(default=False)
    custom_settings_field_two = BooleanField(default=True)

    # 'menupage_panel' is a collapsible `MultiFieldPanel` with the important
    # fields already grouped together, making it easy to include in custom
    # panel definitions, like so:
    settings_panels = [
        FieldPanel('custom_settings_field_one'),
        FieldPanel('custom_settings_field_two'),
        menupage_panel
    ]
    ...
```

3. Create migrations for any models you've updated by running:

```
python manage.py makemigrations appname
```

4. Apply the new migrations by running:

```
python manage.py migrate appname
```

1.6.3 Implementing `MenuPageMixin` into your project

Wagtail has a restriction that forbids models from subclassing more than one other class derived from `Page`, and that single page-derived class must be the left-most item when subclassing more than one model class. Most of the time, that doesn't cause any noticeable issues. But, in some cases, it can make it difficult to swap out base model classes used for page models. In these cases, you can use `wagtailmenus.models.MenuPageMixin` instead of `MenuPage`.

Note: `MenuPageMixin` doesn't change make any changes to the panel configuration on your model in order to surface it's new fields in the page editing interface. If you want those fields to appear, you'll have to override `settings_panels` on your model to include `menupage_panel`

---

1. Subclass `wagtailmenus.models.MenuPageMixin` to create your model, including it to the right of any other class that subclasses `Page`:

```
# appname/models.py

from wagtail.wagtailforms.models import AbstractEmailForm
from wagtailmenus.models import MenuPageMixin
from wagtailmenus.panels import menupage_panel


class MyEmailFormPage(AbstractEmailForm, MenuPageMixin):
    """This page will gain the same fields and methods as if it extended
    `wagtailmenus.models.MenuPage`"""

    ...

    # It's not possible for MenuPageMixin to set `settings_panel`, so you
↪must
    # override `settings_panels` yourself, and include `menupage_panel` in
    # order to surface additional field in the 'Settings' tab of the editor
    # interface
    settings_panels = [
        FieldPanel('custom_settings_field_one'),
        FieldPanel('custom_settings_field_two'),
        menupage_panel
    ]
    ...
```

2. Create migrations for any models you've updated by running:

```
python manage.py makemigrations appname
```

3. Apply the new migrations by running:

```
python manage.py migrate appname
```

1.6.4 Using `MenuPage` to manipulating sub-menu items

When a page model subclasses `MenuPage` or `MenuPageMixin`, pages of that type are given special treatment by the menu generation template tags included in wagtailmenus, allowing them to make changes to the sub-menu items that get rendered below them.

The functionality exists to allow `MenuPage` pages to add repeating links to themselves into a sub-menu, but can be extended to meet any custom needs you might have.

For example, if you had a `ContactPage` model, and in main menus, you wanted to add some additional links below each `ContactPage`, you could achieve that by overriding the `modify_submenu_items()` and `has_submenu_items()` methods like so:

```
# appname/models.py

from wagtailmenus.models import MenuPage


class ContactPage(MenuPage):
    ...

    current_page, current_ancestor_ids,
    current_site, allow_repeating_parents, apply_active_classes,
```

(continues on next page)

(continued from previous page)

```
    original_menu_tag, menu_instance, request, use_absolute_page_urls

    def modify_submenu_items(self, menu_items, **kwargs):
        """
        If rendering a 'main_menu', add some additional menu items to the end
        of the list that link to various anchored sections on the same page.

        We're only making use 'original_menu_tag' and 'current_site' in this
        example, but `kwargs` should have all of the following keys:

        * 'current_page'
        * 'current_ancestor_ids'
        * 'current_site'
        * 'allow_repeating_parents'
        * 'apply_active_classes'
        * 'original_menu_tag'
        * 'menu_instance'
        * 'request'
        * 'use_absolute_page_urls'
        """

        # Start by applying default modifications
        menu_items = super(ContactPage, self).modify_submenu_items(menu_items,␣
↪**kwargs)

        if kwargs['original_menu_tag'] == 'main_menu':
            base_url = self.relative_url(kwargs['current_site'])
            """
            Additional menu items can be objects with the necessary attributes,
            or simple dictionaries. `href` is used for the link URL, and `text`
            is the text displayed for each link. Below, I've also used
            `active_class` to add some additional CSS classes to these items,
            so that I can target them with additional CSS
            """
            menu_items.extend((
                {
                    'text': 'Get support',
                    'href': base_url + '#support',
                    'active_class': 'support',
                },
                {
                    'text': 'Speak to someone',
                    'href': base_url + '#call',
                    'active_class': 'call',
                },
                {
                    'text': 'Map & directions',
                    'href': base_url + '#map',
                    'active_class': 'map',
                },
            ))
        return menu_items

    def has_submenu_items(self, **kwargs):
        """
        Because `modify_submenu_items` is being used to add additional menu
        items, we need to indicate in menu templates that `ContactPage` objects
```

(continues on next page)

(continued from previous page)

```
        do have submenu items in main menus, even if they don't have children
        pages.

        We're only making use 'original_menu_tag' in this example, but
        `kwargs` should have all of the following keys:

        * 'current_page'
        * 'allow_repeating_parents'
        * 'original_menu_tag'
        * 'menu_instance'
        * 'request'
        """

        if kwargs['original_menu_tag'] == 'main_menu':
            return True
        # Resort to default behaviour
        return super(ContactPage, self).has_submenu_items(**kwargs)
```

The above changes would result in the following HTML output when rendering a `ContactPage` instance in a main menu:

```
...
<li class=" dropdown">
    <a href="/contact-us/" class="dropdown-toggle" id="ddtoggle_18" data-toggle=
↪"dropdown" aria-haspopup="true" aria-expanded="false">Contact us <span class="caret
↪"></span></a>
    <ul class="dropdown-menu" aria-labelledby="ddtoggle_18">
        <li class="support"><a href="/contact-us/#support">Get support</a></li>
        <li class="call"><a href="/contact-us/#call">Speak to someone</a></li>
        <li class="map"><a href="/contact-us/#map">Map &amp;amp; directions</a></li>
    </ul>
</li>
...
```

You can also modify sub-menu items based on field values for specific instances, rather than doing the same for every page of the same type. Here's another example:

```
# appname/models.py

from django.db import models
from wagtailmenus.models import MenuPage

class SectionRootPage(MenuPage):
    add_submenu_item_for_news = models.BooleanField(default=False)

    def modify_submenu_items(
        self, menu_items, current_page, current_ancestor_ids, current_site,
        allow_repeating_parents, apply_active_classes, original_menu_tag='',
        menu_instance, request, use_absolute_page_urls
    ):
        menu_items = super(SectionRootPage,self).modify_menu_items(
            menu_items, current_page, current_ancestor_ids,
            current_site, allow_repeating_parents, apply_active_classes,
            original_menu_tag, menu_instance, request, use_absolute_page_urls)

        if self.add_submenu_item_for_news:
```

(continues on next page)

(continued from previous page)

```
            menu_items.append({
                'href': '/news/',
                'text': 'Read the news',
                'active_class': 'news-link',
            })
        return menu_items

    def has_submenu_items(
        self, current_page, allow_repeating_parents, original_menu_tag,
        menu_instance, request
    ):

        if self.add_submenu_item_for_news:
            return True
        return super(SectionRootPage, self).has_submenu_items(
            current_page, allow_repeating_parents, original_menu_tag,
            menu_instance, request)
```

Note: If you're overriding `modify_submenu_items()`, please ensure that 'repeated menu items' are still added as the first item in the returned `menu_items` list. If not, active class highlighting might not work as expected.

1.7 The `AbstractLinkPage` model

Because main and flat menus only allow editors to define the top-level items in a menu, the `AbstractLinkPage` model was introduced to give them a way to easily add additional links to menus, by adding additional pages to the page tree.

Just like menu items defined for a menu via the CMS, link pages can link to other pages or custom URLs, and if linking to another page, the link will automatically become hidden if the target page is unpublished, expires, or is set to no longer show in menus. It will also appear again if the target page is published or set to show in menus again.

By default, link pages are not allowed to have children pages, and shouldn't appear in wagtail-generated site maps or search results.

1.7.1 Implementing `AbstractLinkPage` into your project

Like `MenuPage`, `AbstractLinkPage` is an abstract model, so in order to use it in your project, you need to subclass it.

1. Subclass `AbstractLinkPage` to create a new page type model in your project:

```
# appname/models.py

from wagtailmenus.models import AbstractLinkPage


class LinkPage(AbstractLinkPage):
    pass
```

2. Create migrations for any models you've updated by running:

```
python manage.py makemigrations appname
```

3. Apply the new migrations by running:

```
python manage.py migrate appname
```

1.8 Advanced topics

1.8.1 'Specific' pages and menus

For pages, Wagtail makes use of a technique in Django called 'multi-table inheritance'. In simple terms, this means that when you create an instance of a custom page type model, the data is saved in two different database tables:

- All of the standard fields from Wagtail's `Page` model are stored in one table
- Any data for additional fields specific to your custom model are saved in another one

Because of this, in order for Django to return 'specific' page type instance (e.g. an *EventPage*), it needs to fetch and join data from both tables; which has a negative effect on performance.

Menu generation is particularly resource intensive, because a menu needs to know a lot of data about a lot of pages. Add a need for 'specific' page instances to that mix (perhaps you need to access multilingual field values that only exist in the specific database table, or you want to use other custom field values in your menu templates), and that intensity is understandably greater, as the data will likely be spread over many tables (depending on how many custom page types you are using), needing lots of database joins to put everything together.

Because every project has different needs, wagtailmenus gives you some fine grained control over how 'specific' pages should be used in your menus. When defining a `MainMenu` or `FlatMenu` in the CMS, the **Specific page use** field allows you to choose one of the following options, which can also be passed to any of the included template tags using the `use_specific` parameter.

Supported values for fetching specific pages

- **Off** (value: `0`): Use only standard `Page` model data and methods, and make the minimum number of database methods when rendering. If you aren't using wagtailmenus' `MenuPage` model in your project, and don't need to access any custom page model fields or methods in you menu templates, and aren't overriding `get_url_parts()` or other `Page` methods concerned with URL generation, you should use this option for optimal performance.
- **Auto** (value: `1`): Only fetch and use specific pages when needed for `MenuPage` operations (e.g. for 'repeating menu item' behaviour, and manipulation of sub-menu items via `has_submenu_items()` and `modify_submenu_items()` methods).
- **Top level** (value: `2`): Only fetch and return specific page instances for the top-level menu items (pages that were manually added as menu items via the CMS), but only use vanilla `Page` objects for any additional levels.

  Note: Although accepted by all menu tags, using *use_specific=2* will only really effect `main_menu` and `flat_menu` tags. All other tags will behave the same as if you'd supplied a value of **Auto** (`1`).

- **Always** (value: `3`): Fetch and return specific page instances for ALL pages, so that custom page-type data and methods can be accessed in all menu templates. If you have a multilingual site and want to output translated page content in menus, or if you have models that override `get_url_parts()`, `relative_url()` or

other `Page` methods involved in returning URLs, this is the option you should use in order to fetch the data as efficiently as possible.

Using the `use_specific` template tag argument

All of the template tags included in wagtailmenus accept a `use_specific` argument, allowing you to override any default settings, or the settings applied via the CMS to individual `MainMenu` and `FlatMenu` objects. As a value, you can pass in the integer value of any of the above options, for example:

```
...
{% main_menu use_specific=2 %}
...
{% section_menu use_specific=3 %}
...
{% children_menu use_specific=1 %}
```

Or, the following variables should be available in the context for you to use instead:

- `USE_SPECIFIC_OFF` (value: `0`)
- `USE_SPECIFIC_AUTO` (value `1`)
- `USE_SPECIFIC_TOP_LEVEL` (value `2`)
- `USE_SPECIFIC_ALWAYS` (value `3`)

For example:

```
...
{% main_menu use_specific=USE_SPECIFIC_TOP_LEVEL %}
...
{% section_menu use_specific=USE_SPECIFIC_ALWAYS %}
...
{% children_menu use_specific=USE_SPECIFIC_AUTO %}
```

1.8.2 Using hooks to modify menus

On loading, Wagtail will search for any app with the file `wagtail_hooks.py` and execute the contents. This provides a way to register your own functions to execute at certain points in Wagtail's execution, such as when a `Page` object is saved or when the main menu is constructed.

Registering functions with a Wagtail hook is done through the `@hooks.register` decorator:

```
from wagtail.wagtailcore import hooks

@hooks.register('name_of_hook')
def my_hook_function(arg1, arg2...)
    # your code here
```

Alternatively, `hooks.register` can be called as an ordinary function, passing in the name of the hook and a handler function defined elsewhere:

```
hooks.register('name_of_hook', my_hook_function)
```

Wagtailmenus utilises this same 'hooks' mechanism to allow you make modifications to menus at certain points during the rendering process.

Hooks for modifying QuerySets

When a menu instance is gathering the data it needs to render itself, it typically uses one or more QuerySets to fetch `Page` and `MenuItem` data from the database. These hooks allow you to modify those QuerySets before they are evaluated, allowing you to efficiently control menu contents.

If you need to override a lot of menu class behaviour, and you're comfortable with the idea of subclassing the existing classes and models to override the necessary methods, you might want to look at *Using custom menu classes and models*. But, if all you want to do is change the result of a menu's `get_base_page_queryset()` or `get_base_menuitem_queryset()` (say, to limit the links that appear based on the permissions of the currently logged-in user), you may find it quicker & easier to use the following hooks instead.

menus_modify_base_page_queryset

Whenever a menu needs `Page` data, the menu's `get_base_page_queryset()` method is called to get a 'base' queryset, which then has additional `filter()` and `exclude()` statements added to it as required.

By default, `get_base_page_queryset()` applies a few simple filters to prevent certain pages appearing in your menus:

```
Page.objects.filter(live=True, expired=False, show_in_menus=True)
```

However, if you'd like to filter this result down further, you can do so using something like the following:

Note: The below example shows only a subset of the arguments that are passed to methods using this hook. For a full list of the arguments supplied, see the *Argument reference* below.

```
from wagtail.wagtailcore import hooks

@hooks.register('menus_modify_base_page_queryset')
def make_some_changes(
    queryset, request, menu_instance, original_menu_tag, current_site,
    **kwargs
):
    """
    Ensure only pages 'owned' by the currently logged in user are included.
    NOTE: MUST ALWAYS RETURN A QUERYSET
    """
    if not request.user.is_authenticated():
```

(continues on next page)

(continued from previous page)

```
        return queryset.none()
    return queryset.filter(owner=self.request.user)
```

This would ensure that only pages 'owned' by currently logged-in user will appear in menus. And the changes will be applied to ALL types of menu, regardless of what template tag is being called to do the rendering.

Or, if you only wanted to change the queryset for a menu of a specific type, you could modify the code slightly like so:

```
from wagtail.wagtailcore import hooks

@hooks.register('menus_modify_base_page_queryset')
def make_some_changes(
    queryset, request, menu_instance, original_menu_tag, current_site,
    **kwargs
):
    """
    Ensure only pages 'owned' by the currently logged in user are included,
    but only for 'main' or 'flat' menus.
    NOTE: MUST ALWAYS RETURN A QUERYSET
    """
    if menu_type in ('main_menu', 'flat_menu'):
        if not request.user.is_authenticated():
            return queryset.none()
        queryset = queryset.filter(owner=self.request.user)

    return queryset  # always return a queryset
```

menus_modify_base_menuitem_queryset

When rendering a main or flat menu, top-level items are defined in the CMS, so the menu must fetch that data first, before it can work out whatever additional data is required for rendering.

By default, `get_base_menuitem_queryset()` simply returns all of the menu items that were defined in the CMS. Any page data is then fetched separately (using `get_base_page_queryset()`), and the two results are combined to ensure that only links to appropriate pages are included in the menu being rendered.

However, if you'd only like to include a subset of the CMS-defined menu item, or make any further modifications, you can do so using something like the following:

Note: The below example shows only a subset of the arguments that are passed to methods using this hook. For a full list of the arguments supplied, see the *Argument reference* below.

```
from wagtail.wagtailcore import hooks

@hooks.register('menus_modify_base_menuitem_queryset')
def make_some_changes(
    queryset, request, menu_instance, original_menu_tag, current_site,
    **kwargs
):
    """
    If the request is from a specific site, and the current user is
    authenticated, don't show links to some custom URLs.
```

(continues on next page)

(continued from previous page)

```
    NOTE: MUST ALWAYS RETURN A QUERYSET
    """
    if(
        current_site.hostname.startswith('intranet.') and
        request.user.is_authenticated()
    ):
        queryset = queryset.exclude(handle__contains="visiting-only")
    return queryset  # always return a queryset
```

These changes would be applied to all menu types that use menu items to define the top-level (main and flat menus). If you only wanted to change the queryset for a flat menus, or even a specific flat menu, you could modify the code slightly like so:

```
from wagtail.wagtailcore import hooks

@hooks.register('menus_modify_base_menuitem_queryset')
def make_some_changes(
    queryset, request, menu_instance, original_menu_tag, current_site,
    **kwargs
):
    """
    When generating a flat menu with the 'action-links' handle, and the
    request is for a specific site, and the current user is authenticated,
    don't show links to some custom URLs.
    NOTE: MUST ALWAYS RETURN A QUERYSET
    """
    if(
        original_menu_tag == 'flat_menu' and
        menu_instance.handle == 'action-links' and
        current_site.hostname.startswith('intranet.') and
        request.user.is_authenticated()
    ):
        queryset = queryset.exclude(handle__contains="visiting-only")
    return queryset  # always return a queryset
```

Hooks for modifying menu items

While the above tags are concerned with modifying the data used in a menu, the following hooks are called later on in the rendering process, and allow you to modify the list of `MenuItem` or `Page` objects before they are sent to a template to be rendered.

There are two hooks you can use to modify menu items, which are called at different stages of preparation.

menus_modify_raw_menu_items

This hook allows you to modify the list **before** it is 'primed' (a process that sets `href`, `text`, `active_class` and `has_children_in_menu` attributes on each item), and **before** being sent to a parent page's `modify_submenu_items()` method for further modification (see *Using MenuPage to manipulating sub-menu items*).

Note: The below example shows only a subset of the arguments that are passed to methods using this hook. For a full list of the arguments supplied, see the *Argument reference* below.

```
from wagtail.wagtailcore import hooks

@hooks.register('menus_modify_raw_menu_items')
def make_some_changes(
    menu_items, request, parent_page, original_menu_tag, menu_instance,
    current_level, **kwargs
):
    """
    When rendering the first level of a 'section menu', add a copy of the
    first page to the end of the list.

    NOTE: prime_menu_items() will attempt to add 'href', 'text' and other
    attributes to these items before rendering, so ideally, menu items
    should all be `MenuItem` or `Page` instances.
    """
    if original_menu_tag == 'section_menu' and current_level == 1:
        # Try/except in case menu_items is an empty list
        try:
            menu_items.append(menu_items[0])
        except KeyError:
            pass
    return menu_items  # always return a list
```

The modified list of menu items will then continue to be processed as normal, being passed to the menu's 'prime_menu_items()' method for priming, and then on to the parent page's `modify_submenu_items()` for further modification.

menus_modify_primed_menu_items

This hook allows you to modify the list of items **after** they have been 'primed' and the modified by a parent page's `modify_submenu_items()` methods (see *Using MenuPage to manipulating sub-menu items*).

Note: The below example shows only a subset of the arguments that are passed to methods using this hook. For a full list of the arguments supplied, see the *Argument reference* below.

```
from wagtail.wagtailcore import hooks

@hooks.register('menus_modify_primed_menu_items')
def make_some_changes(
    menu_items, request, parent_page, original_menu_tag, menu_instance,
    current_level, **kwargs
):
    """
    When rendering the first level of a 'main menu', add an additional
    link to the RKH website

    NOTE: This result won't undergo any more processing before sending to
    a template for rendering, so you may need to set 'href' and
    'text' attributes / keys so that those values are picked up by menu
    templates.
    """
    if original_menu_tag == 'main_menu' and current_level == 1:
        # Just adding a simple dict here, as these values are all the
        # template needs to render the link
```

(continues on next page)

(continued from previous page)

```
        menu_items.append({
            'href': 'https://rkh.co.uk',
            'text': 'VISIT RKH.CO.UK',
            'active_class': 'external',
        })
    return menu_items  # always return a list
```

Argument reference

In the above examples, `**kwargs` is used in hook method signatures to make them *accepting* of other keyword arguments, without having to declare every single argument that should be passed in. Using this approach helps create leaner, tidier code, and also makes it more 'future-proof', since the methods will automatically accept any new arguments that may be added by wagtailmenus in future releases.

Below is a full list of the additional arguments that are passed to methods using the above hooks:

`request` The `HttpRequest` instance that the menu is currently being rendered for.

`parent_context` The `Context` instance that the menu is being rendered from.

`parent_page` If the menu being rendered is showing 'children' of a specific page, this will be the `Page` instance who's children pages are being displayed. The value might also be `None` if no parent page is involved. For example, if rendering the top level items of a main or flat menu.

`menu_tag` The name of the tag that was called to render the current part of the menu. If rendering the first level of a menu, this will have the same value as `original_menu_tag`. If not, it will have the value *'sub_menu'* (unless you're using custom tags that pass a different 'tag_name' value to the menu class's 'render_from_tag' method)

`original_menu_tag` The name of the tag that was called to initiate rendering of the menu that is currently being rendered. For example, if you're using the `main_menu` tag to render a multi-level main menu, even though `sub_menu` may be called to render subsequent additional levels, 'original_menu_tag' should retain the value `'main_menu'`. Should be one of: `'main_menu'`, `'flat_menu'`, `'section_menu'` or `'children_menu'`. Comparable to the `menu_type` values supplied to other hooks.

`menu_instance` The menu instance that is supplying the data required to generate the current menu. This could be an instance of a model class, like `MainMenu` or `FlatMenu`, or a standard python class like `ChildrenMenu` or `SectionMenu`.

`original_menu_instance` The menu instance that is supplying the data required to generate the current menu. This could be an instance of a model class, like `MainMenu` or `FlatMenu`, or a standard python class like `ChildrenMenu` or `SectionMenu`.

`current_level` An integer value indicating the 'level' or 'depth' that is currently being rendered in the process of rendering a multi-level menu. This will start at *1* for the first/top-level items of a menu, and increment by *1* for each additional level.

`max_levels` An integer value indicating the maximum number of levels that should be rendered for the current menu. This will either have been specified by the developer using the `max_levels` argument of a menu tag, or might have been set in the CMS for a specific `MainMenu` or `FlatMenu` instance.

`current_site` A Wagtail `Site` instance, indicating the site that the current request is for (usually also available as `request.site`)

`current_page` A Wagtail `Page` instance, indicating what wagtailmenus believes to be the page that is currently being viewed / requested by a user. This might be `None` if you're using additional views in your project to provide functionality at URLs that don't map to a `Page` in Wagtail.

current_page_ancestor_ids A list of ids of `Page` instances that are an 'ancestor' of `current_page`.

current_section_root_page If `current_page` has a value, this will be the top-most ancestor of that page, from just below the site's root page. For example, if your page tree looked like the following:

```
Home (Set as 'root page' for the site)
├── About us
├── What we do
├── Careers
|   ├── Vacancy one
|   └── Vacancy two
├── News & events
|   ├── News
|   |   ├── Article one
|   |   └── Article two
|   └── Events
└── Contact us
```

If the current page was 'Vacancy one', the section root page would be 'Careers'. Or, if the current page was 'Article one', the section root page would be 'News & events'.

use_specific An integer value indicating the preferred policy for using `PageQuerySet.specific()` and `Page.specific` in rendering the current menu. For more information see: *'Specific' pages and menus*.

allow_repeating_parents A boolean value indicating the preferred policy for having pages that subclass `MenuPageMixin` add a repeated versions of themselves to it's children pages (when rendering a *sub_menu* for that page). For more information see: *The MenuPage and MenuPageMixin models*.

apply_active_classes A boolean value indicating the preferred policy for setting `active_class` attributes on menu items for the current menu.

use_absolute_page_urls A boolean value indicating the preferred policy for using full/absolute page URLs for menu items representing pages (observed by `prime_menu_items()` when setting the `href` attribute on each menu item). In most cases this will be `False`, as the default behaviour is to use 'relative' URLs for pages.

1.8.3 Using custom menu classes and models

Overriding the models used for main menus

There are a couple of different approaches for overriding the models used for defining / rendering main menus. The best approach for your project depends on which models you need to override.

Replacing the `MainMenuItem` model only

If you're happy with the default `MainMenu` model, but wish customise the menu item model (e.g. to add images, description fields, or extra fields for translated strings), you can use the *WAGTAILMENUS_MAIN_MENU_ITEMS_RELATED_NAME* setting to have main menus use a different model, both within Wagtail's CMS, and for generating the list of `menu_items` used by menu templates.

1. Within your project, define your custom model by subclassing `AbstractMainMenuItem`:

```
# appname/models.py

from django.db import models
from django.utils.translation import ugettext_lazy as _
from modelcluster.fields import ParentalKey
from wagtail.wagtailimages import get_image_model_string
from wagtail.wagtailimages.edit_handlers import ImageChooserPanel
from wagtail.wagtailadmin.edit_handlers import FieldPanel, PageChooserPanel
from wagtailmenus.models import AbstractMainMenuItem


class CustomMainMenuItem(AbstractMainMenuItem):
    """A custom menu item model to be used by ``wagtailmenus.MainMenu``"""

    menu = ParentalKey(
        'wagtailmenus.MainMenu',
        on_delete=models.CASCADE,
        related_name="custom_menu_items", # important for step 3!
    )
    image = models.ForeignKey(
        get_image_model_string(),
        blank=True,
        null=True,
        on_delete=models.SET_NULL,
    )
    hover_description = models.CharField(
        max_length=250,
        blank=True
    )

    # Also override the panels attribute, so that the new fields appear
    # in the admin interface
    panels = (
        PageChooserPanel('link_page'),
        FieldPanel('link_url'),
        FieldPanel('url_append'),
        FieldPanel('link_text'),
        ImageChooserPanel('image'),
        FieldPanel('hover_description'),
        FieldPanel('allow_subnav'),
    )
```

2. Create migrations for the new model by running:

```
python manage.py makemigrations appname
```

3. Apply the new migrations by running:

```
python manage.py migrate appname
```

4. Add a setting to your project to instruct wagtailmenus to use your custom model instead of the default:

```
# Set this to the 'related_name' attribute used on the ParentalKey field
WAGTAILMENUS_MAIN_MENU_ITEMS_RELATED_NAME = "custom_menu_items"
```

5. *That's it!* The custom models will now be used instead of the default ones.

Note: Although you won't be able to see them in the CMS any longer, the default models and any data that was in the original database table will remain intact.

Replacing both the `MainMenu` and `MainMenuItem` models

If you also need to override the `MainMenu` model, that's possible too. But, because the `MainMenuItem` model is tied to `MainMenu`, you'll also need to create custom menu item model (whether you wish to add fields / change their behaviour, or not).

1. Within your project, define your custom models by subclassing the `AbstractMainMenu` and `AbstractMainMenuItem` model classes:

```
# appname/models.py

from django.db import models
from django.utils import translation
from django.utils.translation import ugettext_lazy as _
from django.utils import timezone
from modelcluster.fields import ParentalKey
from wagtail.wagtailadmin.edit_handlers import FieldPanel, MultiFieldPanel,␣
↪PageChooserPanel
from wagtailmenus import app_settings
from wagtailmenus.models import AbstractMainMenu, AbstractMainMenuItem


class LimitedMainMenu(AbstractMainMenu):
    limit_from = models.TimeField()
    limit_to = models.TimeField()

    def get_base_page_queryset(self):
        """
        If the current time is between 'limit_from' and 'limit_to',
        only surface pages that are owned by the logged in user
        """
        if(
            self.request.user and
            self.limit_from < timezone.now() < self.limit_to
        ):

            return self.request.user.owned_pages.filter(
                live=True, expired=False, show_in_menus=True
            )
        return Page.objects.none()

    # Like pages, panels for menus are split into multiple tabs.
```

(continues on next page)

(continued from previous page)

```
    # To update the panels in the 'Content' tab, override 'content_panels'
    # To update the panels in the 'Settings' tab, override 'settings_panels'
    settings_panels = AbstractMainMenu.setting_panels += (
        MultiFieldPanel(
            heading=_('Time limit settings'),
            children=(
                FieldPanel('limit_from'),
                FieldPanel('limit_to'),
            ),
        ),
    )

class CustomMainMenuItem(AbstractMainMenuItem):
    """A minimal custom menu item model to be used by `LimitedMainMenu`.
    No additional fields / method necessary
    """
    menu = ParentalKey(
        LimitedMainMenu, # we can use the model from above
        on_delete=models.CASCADE,
        related_name=app_settings.MAIN_MENU_ITEMS_RELATED_NAME,
    )
```

2. Create migrations for the new models by running:

```
python manage.py makemigrations appname
```

3. Apply the new migrations by running:

```
python manage.py migrate appname
```

4. Add a setting to your project to tell wagtailmenus to use your custom menu model instead of the default one. e.g:

```
# settings.py

WAGTAILMENUS_MAIN_MENU_MODEL = "appname.LimitedMainMenu"
```

5. *That's it!* The custom models will now be used instead of the default ones.

Note: Although you won't be able to see them in the CMS any longer, the default models and any data that was in the original database table will remain intact.

Overriding the models used for flat menus

There are a couple of different approaches for overriding the models used for defining / rendering flat menus. The best approach for your project depends on which models you need to override.

Replacing the `FlatMenuItem` model only

If you're happy with the default `FlatMenu` model, but wish customise the menu item models (e.g. to add images, description fields, or extra fields for translated strings), you can use the *WAGTAILMENUS_FLAT_MENU_ITEMS_RELATED_NAME* setting to have flat menus use a different model, both within Wagtail's CMS, and for generating the list of `menu_items` used by menu templates.

1. Within your project, define your custom model by subclassing `AbstractFlatMenuItem`:

```
# apname/models.py

from django.db import models
from django.utils.translation import ugettext_lazy as _
from modelcluster.fields import ParentalKey
from wagtail.wagtailimages import get_image_model_string
from wagtail.wagtailimages.edit_handlers import ImageChooserPanel
from wagtail.wagtailadmin.edit_handlers import FieldPanel, PageChooserPanel
from wagtailmenus.models import AbstractFlatMenuItem


class CustomFlatMenuItem(AbstractFlatMenuItem):
    """A custom menu item model to be used by ``wagtailmenus.FlatMenu``"""

    menu = ParentalKey(
        'wagtailmenus.FlatMenu',
        on_delete=models.CASCADE,
        related_name="custom_menu_items", # important for step 3!
    )
    image = models.ForeignKey(
        get_image_model_string(),
        blank=True,
        null=True,
        on_delete=models.SET_NULL,
    )
    hover_description = models.CharField(
        max_length=250,
        blank=True
    )

    # Also override the panels attribute, so that the new fields appear
    # in the admin interface
    panels = (
        PageChooserPanel('link_page'),
        FieldPanel('link_url'),
        FieldPanel('url_append'),
        FieldPanel('link_text'),
        ImageChooserPanel('image'),
        FieldPanel('hover_description'),
        FieldPanel('allow_subnav'),
    )
```

2. Create migrations for the new models by running:

```
python manage.py makemigrations appname
```

3. Apply the new migrations by running:

```
python manage.py migrate appname
```

4. Add a setting to your project to tell wagtailmenus to use your custom model instead of the default one. e.g:

```
# settings.py

# Use the 'related_name' attribute you used on your custom model's ParentalKey␣
↪field
```

(continues on next page)

(continued from previous page)

```
WAGTAILMENUS_FLAT_MENU_ITEMS_RELATED_NAME = "custom_menu_items"
```

5. *That's it!* The custom models will now be used instead of the default ones.

Note: Although you won't be able to see them in the CMS any longer, the default models and any data that was in the original database table will remain intact.

Replacing both the `FlatMenu` and `FlatMenuItem` models

If you also need to override the `FlatMenu` model, that's possible too. But, because the `FlatMenuItem` model is tied to `FlatMenu`, you'll also need to create custom menu item model (whether you wish to add fields or their behaviour or not).

1. Within your project, define your custom models by subclassing the `AbstractFlatMenu` and `AbstractFlatMenuItem` model classes:

```
# appname/models.py

from django.db import models
from django.utils import translation
from django.utils.translation import ugettext_lazy as _
from modelcluster.fields import ParentalKey
from wagtail.wagtailadmin.edit_handlers import FieldPanel, MultiFieldPanel,␣
↪PageChooserPanel
from wagtailmenus import app_settings
from wagtailmenus.panels import FlatMenuItemsInlinePanel
from wagtailmenus.models import AbstractFlatMenu, AbstractFlatMenuItem


class TranslatedField(object):
    """
    A class that can be used on models to return a 'field' in the
    desired language, where there a multiple versions of a field to
    cater for multiple languages (in this case, English, German & French)
    """
    def __init__(self, en_field, de_field, fr_field):
        self.en_field = en_field
        self.de_field = de_field
        self.fr_field = fr_field

    def __get__(self, instance, owner):
        active_language = translation.get_language()
        if active_language == 'de':
            return getattr(instance, self.de_field)
        if active_language == 'fr':
            return getattr(instance, self.fr_field)
        return getattr(instance, self.en_field)


class TranslatedFlatMenu(AbstractFlatMenu):
    heading_de = models.CharField(
        verbose_name=_("heading (german)"),
        max_length=255,
```

(continues on next page)

(continued from previous page)

```
        blank=True,
    )
    heading_fr = models.CharField(
        verbose_name=_("heading (french)"),
        max_length=255,
        blank=True,
    )
    translated_heading = TranslatedField('heading', 'heading_de', 'heading_fr')

    # Like pages, panels for menus are split into multiple tabs.
    # To update the panels in the 'Content' tab, override 'content_panels'
    # To update the panels in the 'Settings' tab, override 'settings_panels'
    content_panels = (
        MultiFieldPanel(
            heading=_("Settings"),
            children=(
                FieldPanel("title"),
                FieldPanel("site"),
                FieldPanel("handle"),
            )
        ),
        MultiFieldPanel(
            heading=_("Heading"),
            children=(
                FieldPanel("heading"),
                FieldPanel("heading_de"),
                FieldPanel("heading_fr"),
            ),
            classname='collapsible'
        ),
        FlatMenuItemsInlinePanel(),
    )

class TranslatedFlatMenuItem(AbstractFlatMenuItem):
    """A custom menu item model to be used by ``TranslatedFlatMenu``"""

    menu = ParentalKey(
        TranslatedFlatMenu, # we can use the model from above
        on_delete=models.CASCADE,
        related_name=app_settings.FLAT_MENU_ITEMS_RELATED_NAME,
    )
    link_text_de = models.CharField(
        verbose_name=_("link text (german)"),
        max_length=255,
        blank=True,
    )
    link_text_fr = models.CharField(
        verbose_name=_("link text (french)"),
        max_length=255,
        blank=True,
    )
    translated_link_text = TranslatedField('link_text', 'link_text_de', 'link_
↪text_fr')

    @property
    def menu_text(self):
        """Use `translated_link_text` instead of just `link_text`"""
```

(continues on next page)

(continued from previous page)

```
        return self.translated_link_text or getattr(
            self.link_page,
            app_settings.PAGE_FIELD_FOR_MENU_ITEM_TEXT,
            self.link_page.title
        )

    # Also override the panels attribute, so that the new fields appear
    # in the admin interface
    panels = (
        PageChooserPanel("link_page"),
        FieldPanel("link_url"),
        FieldPanel("url_append"),
        FieldPanel("link_text"),
        FieldPanel("link_text_de"),
        FieldPanel("link_text_fr"),
        FieldPanel("handle"),
        FieldPanel("allow_subnav"),
    )
```

2. Create migrations for the new models by running:

```
python manage.py makemigrations appname
```

3. Apply the new migrations by running:

```
python manage.py migrate appname
```

4. Add a setting to your project to tell wagtailmenus to use your custom menu model instead of the default one. e.g:

```
# settings.py

WAGTAILMENUS_FLAT_MENU_MODEL = "appname.TranslatedFlatMenu"
```

5. That's it! The custom models will now be used instead of the default ones.

Note: Although you won't be able to see them in the CMS any longer, the default models and any data that was in the original database table will remain intact.

Overriding the menu class used by `{% section_menu %}`

Like the `main_menu` and `flat_menu` tags, the `section_menu` tag uses a `Menu` class to fetch all of the data needed to render a menu. Though, because section menus are driven entirely by your existing page tree (and don't need to store any additional data), it's just a plain old Python class and not a Django model.

The class `wagtailmenus.models.menus.SectionMenu` is used by default, but you can use the `WAGTAILMENUS_SECTION_MENU_CLASS` setting in your project to make wagtailmenus use an alternative class (for example, if you want to modify the base queryset that determines which pages should be included when rendering). To implement a custom classes, it's recommended that you subclass the `SectionMenu` and override any methods as required, like in the following example:

```
# mysite/appname/models.py
```

(continues on next page)

(continued from previous page)

```
from django.utils.translation import ugettext_lazy as _
from wagtail.wagtailcore.models import Page
from wagtailmenus.models import SectionMenu


class CustomSectionMenu(SectionMenu):

    def get_base_page_queryset(self):
        # Show draft and expired pages in menu for superusers
        if self.request.user.is_superuser:
            return Page.objects.filter(show_in_menus=True)
        # Resort to default behaviour for everybody else
        return super(CustomSectionMenu, self).get_base_page_queryset()
```

```
# settings.py

WAGTAILMENUS_SECTION_MENU_CLASS = "mysite.appname.models.CustomSectionMenu"
```

Overriding the menu class used by `{% children_menu %}`

Like all of the other tags, the `children_menu` tag uses a `Menu` class to fetch all of the data needed to render a menu. Though, because children menus are driven entirely by your existing page tree (and do not need to store any additional data), it's just a plain old Python class and not a Django model.

The class `wagtailmenus.models.menus.ChildrenMenu` is used by default, but you can use the `WAGTAILMENUS_CHILDREN_MENU_CLASS` setting in your project to make wagtailmenus use an alternative class (for example, if you want to modify which pages are included). For custom classes, it's recommended that you subclass `ChildrenMenu` and override any methods as required e.g:

```
# appname/menus.py

from django.utils.translation import ugettext_lazy as _
from wagtail.wagtailcore.models import Page
from wagtailmenus.models import ChildrenMenu


class CustomChildrenMenu(ChildrenMenu):
    def get_base_page_queryset(self):
    # Show draft and expired pages in menu for superusers
    if self.request.user.is_superuser:
        return Page.objects.filter(show_in_menus=True)
    # Resort to default behaviour for everybody else
    return super(CustomChildrenMenu, self).get_base_page_queryset()
```

```
# settings.py

WAGTAILMENUS_CHILDREN_MENU_CLASS = "mysite.appname.models.CustomChildrenMenu"
```

1.9 Settings reference

You can override some of wagtailmenus' default behaviour by adding one of more of the following to your project's settings.

- *WAGTAILMENUS_SECTION_ROOT_DEPTH*
- *WAGTAILMENUS_CUSTOM_URL_SMART_ACTIVE_CLASSES*

1.9.1 Admin / UI settings

WAGTAILMENUS_ADD_EDITOR_OVERRIDE_STYLES

Default value: `True`

By default, wagtailmenus adds some additional styles to improve the readability of the forms on the menu management pages in the Wagtail admin area. If for some reason you don't want to override any styles, you can disable this behaviour by setting to `False`.

WAGTAILMENUS_FLATMENU_MENU_ICON

Default value: `'list-ol'`

Use this to change the icon used to represent 'Flat menus' in the Wagtail CMS.

WAGTAILMENUS_FLAT_MENUS_HANDLE_CHOICES

Default value: `None`

Can be set to a tuple of choices in the standard Django choices format to change the presentation of the `FlatMenu.handle` field from a text field, to a select field with fixed choices, when adding, editing or copying a flat menus in Wagtail's CMS.

For example, if your project uses an 'info' menu in the header, a 'footer' menu in the footer, and a 'help' menu in the sidebar, you could do the following:

```
# settings.py

WAGTAILMENUS_FLAT_MENUS_HANDLE_CHOICES = (
    ('info', 'Info'),
    ('help', 'Help'),
    ('footer', 'Footer'),
)
```

WAGTAILMENUS_FLAT_MENUS_EDITABLE_IN_WAGTAILADMIN

Default value: `True`

By default, 'Flat menus' are editable in the Wagtail CMS. Setting this to *False* in your project's settings will disable editing 'Flat menus' in the Wagtail CMS.

WAGTAILMENUS_FLAT_MENUS_MODELADMIN_CLASS

Default value: `'wagtailmenus.modeladmin.FlatMenuAdmin'`

If you wish to override the `ModelAdmin` class used to represent **'Flat menus'** in the Wagtail admin area for your project (e.g. to display additional custom fields in the listing view, or change/add new views), you can do so by using this setting to swap out the default class for a custom one. e.g.

```
# settings.py

WAGTAILMENUS_FLAT_MENUS_MODELADMIN_CLASS = "projectname.appname.modulename.ClassName"
```

The value should be an import path string, rather than a direct pointer to the class itself. wagtailmenus will lazily import the class from this path when it is required. If the path is invalid, an `ImproperlyConfigured` exception will be raised.

WAGTAILMENUS_MAINMENU_MENU_ICON

Default value: `'list-ol'`

Use this to change the icon used to represent 'Main menus' in the Wagtail CMS.

WAGTAILMENUS_MAIN_MENUS_EDITABLE_IN_WAGTAILADMIN

Default value: `True`

By default, 'Main menus' are editable in the Wagtail CMS. Setting this to *False* in your project's settings will disable editing 'Main menus' in the Wagtail CMS.

WAGTAILMENUS_MAIN_MENUS_MODELADMIN_CLASS

Default value: `'wagtailmenus.modeladmin.MainMenuAdmin'`

If you wish to override the `ModelAdmin` class used to represent **'Main menus'** in the Wagtail admin area for your project (e.g. to display additional custom fields in the listing view, or change/add new views), you can do so by using this setting to swap out the default class for a custom one. e.g.

```
# settings.py

WAGTAILMENUS_MAIN_MENUS_MODELADMIN_CLASS = "projectname.appname.modulename.ClassName"
```

The value should be an import path string, rather than a direct pointer to the class itself. Wagtailmenus will lazily import the class from this path when it is required. If the path is invalid, and `ImproperlyConfigured` exception will be raised.

1.9.2 Default templates and template finder settings

WAGTAILMENUS_DEFAULT_CHILDREN_MENU_TEMPLATE

Default value: `'menus/children_menu.html'`

The name of the template used for rendering by the `{% children_menu %}` tag when no other template has been specified using the `template` parameter.

WAGTAILMENUS_DEFAULT_FLAT_MENU_TEMPLATE

Default value: `'menus/flat_menu.html'`

The name of the template used for rendering by the `{% flat_menu %}` tag when no other template has been specified using the `template` parameter.

WAGTAILMENUS_DEFAULT_MAIN_MENU_TEMPLATE

Default value: `'menus/main_menu.html'`

The name of the template used for rendering by the `{% main_menu %}` tag when no other template has been specified using the `template` parameter.

WAGTAILMENUS_DEFAULT_SECTION_MENU_TEMPLATE

Default value: `'menus/section_menu.html'`

The name of the template used for rendering by the `{% section_menu %}` tag when no other template has been specified using the `template` parameter.

WAGTAILMENUS_DEFAULT_SUB_MENU_TEMPLATE

Default value: `'menus/sub_menu.html'`

The name of the template used for rendering by the `{% sub_menu %}` tag when no other template has been specified using the `template` parameter or using the `sub_menu_template` parameter on the original menu tag.

WAGTAILMENUS_SITE_SPECIFIC_TEMPLATE_DIRS

Default value: `False`

If you have a multi-site project, and want to be able to use different templates for some or all of those sites, wagtailmenus can be configured to look for additional 'site specific' paths for each template. To enable this feature, you add the following to your project's settings:

```
# settings.py

WAGTAILMENUS_SITE_SPECIFIC_TEMPLATE_DIRS = True
```

With this set, menu tags will attempt to identify the relevant `wagtail.core.models.Site` instance for the current `request`. Wagtailmenus will then look for template names with the `domain` value of that `Site` object in their path.

For more information about where wagtailmenus looks for templates, see: *Using preferred paths and names for your templates*

1.9.3 Default tag behaviour settings

WAGTAILMENUS_FLAT_MENUS_FALL_BACK_TO_DEFAULT_SITE_MENUS

Default value: `False`

The default value used for `fall_back_to_default_site_menus` option of the `{% flat_menu %}` tag when a parameter value isn't provided.

WAGTAILMENUS_GUESS_TREE_POSITION_FROM_PATH

Default value: `True`

When not using wagtail's routing/serving mechanism to serve page objects, wagtailmenus can use the request path to attempt to identify a 'current' page, 'section root' page, allowing `{% section_menu %}` and active item highlighting to work. If this functionality is not required for your project, you can disable it by setting this value to `False`.

WAGTAILMENUS_DEFAULT_ADD_SUB_MENUS_INLINE

New in version 2.12.

Default value: `False`

For all menu types, when preparing menu items for rendering, sub menus are not added to menu items directly by default, because it's more common for developers to use the `{% sub_menu %}` tag in a menu templates to render additional branches of the menu. In which case, the sub menu is created by the tag.

This behaviour can be overridden on an 'individual use' basis by utilising the `add_sub_menus_inline` option available for each template tag. However, users wishing to change the default behaviour (so that sub menus are appended directly to menu items, without having to specify) can do so by providing a value of `True` in their project settings.

WAGTAILMENUS_DEFAULT_CHILDREN_MENU_MAX_LEVELS

Default value: `1`

The maximum number of levels rendered by the `{% children_menu %}` tag when no value has been specified using the `max_levels` parameter.

WAGTAILMENUS_DEFAULT_SECTION_MENU_MAX_LEVELS

Default value: `2`

The maximum number of levels rendered by the `{% section_menu %}` tag when no value has been specified using the `max_levels` parameter.

WAGTAILMENUS_DEFAULT_CHILDREN_MENU_USE_SPECIFIC

Default value: `1` (Auto)

Controls how 'specific' pages objects are fetched and used during rendering of the `{% children_menu %}` tag when no `use_specific` value isn't supplied.

If you'd like to use custom page fields in your children menus (e.g. translated field values or image fields) or if your page models override `get_url_parts()`, `relative_url()` or other `Page` methods involved in URL generation, you'll likely want to update this.

To find out more about what values are supported and the effect they have, see: *'Specific' pages and menus*

WAGTAILMENUS_DEFAULT_SECTION_MENU_USE_SPECIFIC

Default value: `1` (Auto)

Controls how 'specific' pages objects are fetched and used during rendering of the `{% section_menu %}` tag when no alternative value has been specified using the `use_specific` parameter.

If you'd like to use custom page fields in your section menus (e.g. translated field values, images, or other fields / methods) or if your page models override `get_url_parts()`, `relative_url()` or other `Page` methods involved in URL generation, you'll likely want to update this.

To find out more about what values are supported and the effect they have, see: *'Specific' pages and menus*

1.9.4 Menu class and model override settings

WAGTAILMENUS_CHILDREN_MENU_CLASS

Default value: `'wagtailmenus.models.menus.ChildrenMenu'`

Use this to specify a custom menu class to be used by wagtailmenus' `children_menu` tag. The value should be the import path of your custom class as a string, e.g. `'mysite.appname.models.CustomClass'`.

For more details see: *Overriding the menu class used by {% children_menu %}*

WAGTAILMENUS_FLAT_MENU_MODEL

Default value: `'wagtailmenus.FlatMenu'`

Use this to specify a custom model to use for flat menus instead of the default. The model should be a subclass of `wagtailmenus.AbstractFlatMenu`.

For more details see: *Overriding the models used for flat menus*

WAGTAILMENUS_FLAT_MENU_ITEMS_RELATED_NAME

Default value: `'menu_items'`

Use this to specify the 'related name' that should be used to access menu items from flat menu instances. Used to replace the default *FlatMenuItem* model with a custom one.

For more details see: *Overriding the models used for flat menus*

WAGTAILMENUS_MAIN_MENU_MODEL

Default value: `'wagtailmenus.MainMenu'`

Use this to specify an alternative model to use for main menus. The model should be a subclass of `wagtailmenus.AbstractMainMenu`.

For more details see: *Overriding the models used for main menus*

WAGTAILMENUS_MAIN_MENU_ITEMS_RELATED_NAME

Default value: `'menu_items'`

Use this to specify the 'related name' that should be used to access menu items from main menu instances. Used to replace the default `MainMenuItem` model with a custom one.

For more details see: *Overriding the models used for main menus*

WAGTAILMENUS_SECTION_MENU_CLASS

Default value: `'wagtailmenus.models.menus.SectionMenu'`

Use this to specify a custom class to be used by wagtailmenus' `section_menu` tag. The value should be the import path of your custom class as a string, e.g. `'mysite.appname.models.CustomClass'`.

For more details see: *Overriding the menu class used by {% section_menu %}*

1.9.5 Miscellaneous settings

WAGTAILMENUS_ACTIVE_CLASS

Default value: `'active'`

The class added to menu items for the currently active page (when using a menu template with `apply_active_classes=True`)

WAGTAILMENUS_ACTIVE_ANCESTOR_CLASS

Default value: `'ancestor'`

The class added to any menu items for pages that are ancestors of the currently active page (when using a menu template with `apply_active_classes=True`)

WAGTAILMENUS_PAGE_FIELD_FOR_MENU_ITEM_TEXT

Default value: `'title'`

When preparing menu items for rendering, wagtailmenus looks for a field, attribute or property method on each page with this name to set a `text` attribute value, which is used in menu templates as the label for each item. The `title` field is used by default.

Note: wagtailmenus will only be able to access custom page fields or methods if 'specific' pages are being used (See *'Specific' pages and menus*). If no attribute can be found matching the specified name, wagtailmenus will silently fall back to using the page's `title` field value.

WAGTAILMENUS_SECTION_ROOT_DEPTH

Default value: `3`

Use this to specify the 'depth' value of a project's 'section root' pages. For most Wagtail projects, this should be `3` (Root page depth = `1`, Home page depth = `2`), but it may well differ, depending on the needs of the project.

WAGTAILMENUS_CUSTOM_URL_SMART_ACTIVE_CLASSES

Default value: `False`

By default, menu items linking to custom URLs are attributed with the 'active' class only if their `link_url` value matches the path of the current request _exactly_. Setting this to *True* in your project's settings will enable a smarter approach to active class attribution for custom URLs, where only the 'path' part of the `link_url` value is used to

determine what active class should be used. The new approach will also attribute the 'ancestor' class to menu items if the `link_url` looks like an ancestor of the current request URL.

1.10 Contributing to wagtailmenus

Hey! First of all, thanks for considering to help out!

We welcome all support, whether on bug reports, code, reviews, tests, documentation, translations or just feature requests.

1.10.1 Using the issue tracker

The issue tracker is the preferred channel for bug reports, features requests and submitting pull requests. Please don't use the issue tracker for support requests. If you need help with something that isn't a bug, you can join our Wagtailmenus support group and ask your question there.

1.10.2 Submitting translations

Please submit any new or improved translations through Transifex.

1.10.3 Contributing code changes via pull requests

If there are any open issues you think you can help with, please comment on the issue and state your intent to help. Or, if you have an idea for a feature you'd like to work on, raise it as an issue. Once a core contributor has responded and is happy for you to proceed with a solution, you should create your own fork of wagtailmenus, make the changes there. Before committing any changes, we highly recommend that you create a new branch, and keep all related changes within that same branch. When you've finished making your changes, and the tests are passing, you can then submit a pull request for review.

What your pull request should include

In order to be accepted/merged, your pull request will need to meet the following criteria:

1. Documentation updates to cover any new features or changes.
2. If you're not in the list already, add a new line to `CONTRIBUTORS.md` (under the 'Contributors' heading) with your name, company name, and an optional twitter handle / email address.

3. For all new features, please add additional unit tests to `wagtailmenus.tests`, to test what you've written. Although the quality of unit tests is the most important thing (they should be readable, and test the correct thing / combination of things), code coverage is important too, so please ensure as many lines of your code as possible are accessed when the unit tests are run.

1.10.4 Developing locally

If you'd like a runnable Django project to help with development of wagtailmenus, follow these steps to get started (Mac only). The development environment has `django-debug-toolbar` and some other helpful packages installed to help you debug with your code as you develop:

1. In a Terminal window, `cd` to the project's root directory, and run:

```
pip install -e '.[testing,docs]' -U
pip install -r requirements/development.txt
```

2. Create a copy of the development settings:

```
cp wagtailmenus/settings/development.py.example wagtailmenus/settings/development.
↪py
```

3. Create a copy of the development urls:

```
cp wagtailmenus/development/urls.py.example wagtailmenus/development/urls.py
```

4. Create `manage.py` by copying the example provided:

```
cp manage.py.example manage.py
```

5. Run the migrate command to set up the database tables:

```
python manage.py migrate
```

6. To load some test data into the database, run:

```
python manage.py loaddata wagtailmenus/tests/fixtures/test.json
```

7. Create a new superuser that you can use to access the CMS:

```
python manage.py createsuperuser
```

8. Run the project using the standard Django command:

```
python manage.py runserver
```

Your local copies of `settings/development.py` and `manage.py` will be ignored by git when you push any changes, as will anything you add to the `wagtailmenus/development/` directory.

1.10.5 Testing locally

It's important that any new code is tested before submitting. To quickly test code in your active development environment, you should first install all of the requirements by running:

```
pip install -e '.[testing,docs]' -U
```

Then, run the following command to execute tests:

```
python runtests.py
```

Or if you want to measure test coverage, run:

```
coverage --source=wagtailmenus runtests.py
coverage report
```

Testing in a single environment is a quick and easy way to identify obvious issues with your code. However, it's important to test changes in other environments too, before they are submitted. In order to help with this, wagtailmenus is configured to use `tox` for multi-environment tests. They take longer to complete, but running them is as simple as running:

```
tox
```

You might find it easier to set up a Travis CI service integration for your fork in GitHub (look under **Settings > Apps and integrations** in GitHub's web interface for your fork), and have Travis CI run tests whenever you commit changes. The test configuration files already present in the project should work for your fork too, making it a cinch to set up.

1.10.6 Other topics

Release packaging guidelines

Preparing for a new release

Follow the steps outlined below to prep changes in your fork:

1. Merge any changes from `upstream/master` into your fork's `master` branch.

   ```
   git fetch upstream
   git checkout master
   git merge upstream/master
   ```

2. From your fork's `master` branch, create a new branch for preparing the release, e.g.:

   ```
   git checkout -b release-prep/2.X.X
   ```

3. Update `__version__` in `wagtailmenus/__init__.py` to reflect the new release version.
4. Make sure `CHANGELOG.md` is updated with details of any changes since the last release.
5. Make sure the release notes for the new version have been created / updated in `docs/source/releases/` and are referenced in `docs/source/releases/index.rst`. Be sure to remove the '(alpha)' or '(beta)' from the heading in the latest release notes, as well as the 'Wagtailmenus X.X is in the alpha stage of development' just below.
6. If releasing a 'final' version, following an 'alpha' or 'beta' release, ensure the `a` or `b` is removed from the file name for the release, and the reference to it in `docs/source/releases/index.rst`.
7. `cd` into the `docs` directory to check documentation-related stuff:

   ```
   cd docs
   ```

8. Check for and correct any spelling errors raised by sphinx:

```
make spelling
```

9. Check that the docs build okay, and fix any errors raised by sphinx:

```
make html
```

10. Commit changes so far:

```
git commit -am 'Bumped version and updated release notes'
```

11. Update the source translation files by running the following from the project's root directory:

```
# Update source files
django-admin.py makemessages -l en

# Commit the changes
git commit -am 'Update source translation files'
```

12. Push all outstanding changes to GitHub:

```
git push
```

13. Submit your changes as a PR to the main repository via https://github.com/rkhleics/wagtailmenus/compare

Packaging and pushing to PyPi

When satisfied with the PR for preparing the files:

1. From https://github.com/rkhleics/wagtailmenus/pulls, merge the PR into the `master` branch using the "merge commit" option.
2. Locally, `cd` to the project's root directory, checkout the `master` branch, and ensure the local copy is up-to-date:

```
workon wagtailmenus
cd ../path-to-original-repo
git checkout master
git pull
```

3. Ensure dependencies are up-to-date by running:

```
pip install -e '.[deployment]' -U
```

4. Push any updated translation source files to Transifex:

```
tx push -s -l en
```

5. Pull down updated translations from Transifex:

```
tx pull --a
rm -r wagtailmenus/locale/en_GB/
git add *.po
```

6. Convert the `.po` files to `.mo` for each language by running:

```
find . -name \*.po -execdir msgfmt django.po -o django.mo \;
```

7. Commit and push all changes so far:

```
git commit -am 'Pulled updated translations from Transifex and converted to .mo'
git push
```

8. Create a new tag for the new version and push that too. Travis CI should deploy the new version directly to PyPI once it's finished building:

```
git tag -a v2.X
git push --tags
```

9. **Edit the release notes for the release from** https://github.com/rkhleics/wagtailmenus/releases, by copying and pasting the content from `docs/releases/x.x.x.rst`

10. Crack open a beer - you earned it!

1.11 Release notes

1.11.1 Wagtailmenus 2.12 release notes

What's new?

New `add_sub_menus_inline` option for menu tags

By default, you have to call the `{% sub_menu %}` tag within a menu template to render new branches of a multi-level menu. However, if you add `add_sub_menus_inline=True` to the initial `{% main_menu %}`, `{% flat_menu %}`, `{% children_menu %}` or `{% section_menu %}` tag, then sub menu instances will be created added directly to any menu item where `item.has_children_in_menu` is `True`, allowing you to easily render the entire menu structure from within the same template.

For example, instead of the following:

```
{% for item in menu_items %}
    <li class="{{ item.active_class }}">
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {% sub_menu item %}
        {% endif %}
    </li>
{% endfor %}
```

You could do:

```
 {% for item in menu_items %}
    <li class="{{ item.active_class }}">
```

(continues on next page)

(continued from previous page)

```
        <a href="{{ item.href }}">{{ item.text }}</a>
        {% if item.has_children_in_menu %}
            {{ item.sub_menu.render_to_template }}
        {% endif %}
    </li>
{% endfor %}
```

If you'd rather have sub menus be added inline by default (without having to add `add_sub_menus_inline=True` each time you use a template tag), you can change the default behaviour for all template tags by overriding the *WAGTAILMENUS_DEFAULT_ADD_SUB_MENUS_INLINE* setting in your project's Django settings.

Minor changes & bug fixes

- Arabic translations added - Courtesy of @alfuhigi. Many thanks!
- Fixed an issue with the section menu in the release notes section of the docs.
- Changed the signature of `Menu.render_from_tag()` to better indicate common expected/supported arguments for menus.
- Fixed `Menu.get_common_hook_kwargs()`, so the menu instances `use_specific` attribute value is correctly passed to hooks.
- Removed `request`, `max_levels` and `use_specific` as class attributes from `Menu` to prevent unexpected mutation.
- Added a custom `render_from_tag()` method for each individual menu class, with a signature that highlights all relevant options, including those specific to that menu type.
- Renamed `Menu.get_contextual_vals_from_context()` to `_create_contextualvals_obj_from_context`
- Renamed `Menu.get_option_vals_from_options` to `_create_optionvals_obj_from_values()`.

Deprecations

The following items have been deprecated in this release. They will continue to be supported for a standard deprecation period of two minor releases, but will be removed in version 3.0.

`Menu.get_instance_for_rendering()`

In an effort to make method names more reflective of their functionality, this method has been replaced by two methods: `create_from_collected_values()` and `get_from_collected_values()`. The former is implemented on menu classes that are not model based (where instances must be created from scratch each time, for example: `ChildrenMenu`, `SectionMenu`, `SubMenu`), and the latter is implemented on model-based menu classes, where a corresponding object must be retrieved from the database (so, `AbstractMainMenu`, `MainMenu`, `AbstractFlatMenu` and `FlatMenu`).

`render_from_tag()` automatically calls one or the other, depending on whether the class inherits from `django.db.models.Model`.

If you're using custom menu classes in your project, and are currently overriding `get_instance_for_rendering()` for any of those classes, you should update your code to override one of the new methods instead. Both of these new methods accept the same arguments, and return the same values, so the transition should be very easy.

`Menu.get_contextual_vals_from_context()`

In an effort to make method names more reflective of their functionality, and to help dissuade users from overriding functionality that is subject to change, this method has been renamed to `_create_contextualvals_obj_from_context()` (becoming a private method in the process).

`Menu.get_option_vals_from_options()`

In an effort to make method names more reflective of their functionality, and to help dissuade users from overriding functionality that is subject to change, this method has been renamed to `_create_optionvals_obj_from_values()` (becoming a private method in the process).

Upgrade considerations

Following a standard deprecation period a two minor releases, the following functionality has now been removed.

The WAGTAILMENUS_CHILDREN_MENU_CLASS_PATH setting is no longer supported

If you're using this to override the menu class used to render children menus in your project, you'll need to update your Django settings to use the new, shorter setting name: WAGTAILMENUS_CHILDREN_MENU_CLASS.

For example, if you had the following:

```
# settings/base.py

WAGTAILMENUS_CHILDREN_MENU_CLASS_PATH = 'project.menus.CustomChildrenMenu'
```

You would change this to:

```
WAGTAILMENUS_CHILDREN_MENU_CLASS = 'project.menus.CustomChildrenMenu'
```

The WAGTAILMENUS_SECTION_MENU_CLASS_PATH setting is no longer supported

If you're using this to override the menu class used to render section menus in your project, you'll need to update your Django settings to use the new, shorter setting name: WAGTAILMENUS_SECTION_MENU_CLASS.

For example, if you had the following:

```
# settings/base.py

WAGTAILMENUS_SECTION_MENU_CLASS_PATH = 'project.menus.CustomSectionMenu'
```

You would change this to:

```
WAGTAILMENUS_SECTION_MENU_CLASS = 'project.app.module.CustomSectionMenu'
```

The `wagtailmenus.app_settings` module has been removed

If you're importing this in your project from it's previous location, you should update the import statements in your code to use the new module path: `wagtailmenus.conf.settings`

For example, instead of the following:

```
from wagtailmenus import app_settings
```

You should do:

```
from wagtailmenus.conf import settings
```

The `wagtailmenus.constants` module has been removed

If you're importing this in your project from it's previous location, you should update the import statements in your code to use the new module path: `wagtailmenus.conf.constants`

For example, instead of the following:

```
from wagtailmenus import constants
```

You should do:

```
from wagtailmenus.conf import constants
```

1.11.2 Wagtailmenus 2.11.1 release notes

This is a minor release to solve a few non code-related issues:

- Fixed an issue with the section menu in the release notes section of the docs.
- Updated tox configuration to test against Python 3.7 and Wagtail 2.2
- Updated Travis CI configuration to deploy to PyPi automatically when commits are tagged appropriately.
- Pinned `django-cogwheels` dependency to version 0.2 to reduce potential for backwards-incompatibility issues.

1.11.3 Wagtailmenus 2.11 release notes

What's new?

Replace the `ModelAdmin` classes used by wagtailmenus with your own

Previously, the only way to override any of the admin functionality within wagtailmenus was to resort to monkey-patching, which is obviously not ideal.

I wanted to provide a documented, officially supported way to do this, by allowing the default `wagtail.contrib.modelamin.ModelAdmin` classes in wagtailmenus to be swapped out for custom ones with settings.

For more information see the new entries in the settings reference docs:

- *WAGTAILMENUS_FLAT_MENUS_MODELADMIN_CLASS*
- *WAGTAILMENUS_MAIN_MENUS_MODELADMIN_CLASS*

Minor changes & bug fixes

- Added support for Wagtail version 2.1.
- Added new 'Wagtail' trove classifiers in `setup.py` to reflect Wagtail version support.
- Updated `runtests.py` to pass on any unparsed option arguments to Django's test method.
- Updated `runtests.py` to filter out deprecation warnings originating from other apps by default.
- Updated `MenuItem.relative_url()` to accept a `request` parameter (for parity with `wagtail.core.models.Page.relative_url()`), so that it can pass it on to the page method.
- Updated `Menu.prime_menu_items()` to send the current `HttpRequest` to that `MenuItem.relative_url()` and `Page.relative_url()`.
- Moved most 'App settings related' tests to *wagtailmenus.conf.tests*, so that they're all in one place.
- Updated admin views to utilise `wagtail.admin.messages.validation_error()` for reporting field-specific and non-field errors (Wagtail 2.1 has this built-in, but Wagtail 2.0 does not).
- Replaced the custom app settings module with a `django-cogwheels` settings helper and removed a lot of the tests that existed to test its workings.
- Moved remaining app settings tests to `wagtailmenus.conf.tests`.

Deprecations

- If you're using custom `MenuItem` models in your project, and are overriding `relative_url()`, you should update the method signature on your custom model to accept a `request` keyword argument, and it on to `link_page.relative_url()` and `super().relative_url()` (in addition to `site`) if calling either of those. This will be mandatory in wagtailmenus 1.13.

Upgrade considerations

- Dropped support for Wagtail versions 1.10 to 1.13.
- Dropped support for Django versions 1.8 to 1.10.
- Menu.prime_menu_items() now returns a list instead of a generator. This is required to allow the method to raise warnings.
- Wagtailmenus' custom `modeladmin` classes have moved from `wagtailmenus.wagtail_hooks` to a new `wagtailmenus.modeladmin` module. Importing `FlatMenuAdmin` or `MainMenuAdmin` from the *wagtail_hooks* will still continue to work, but any references to `wagtailmenus.wagtail_hooks.FlatMenuButtonHelper` will need to be updated.
- If for any reason you are importing view classes directly from `wagtailmenus.wagtail_hooks`, that will no longer work. You will have to update your code to import view classes from the views module itself (`wagtailmenus.views`).

1.11.4 Wagtailmenus 2.10 release notes

Note: In order to keep wagtailmenus working with the latest versions of Wagtail and Django, with code that is clean and maintainable, this version will be the last feature release to support Wagtail versions earlier than 2.0, and Django versions earlier than 1.11.

Minor changes & bug fixes

- Optimised `MenuWithMenuItems.get_top_level_items()` and `AbstractFlatMenu.get_for_site()` to use fewer database queries.
- Configured `sphinxcontrib.spelling` and used to correct spelling errors in docs.
- Updated testing and documentation dependencies.
- Added Latin American Spanish translations (thanks to José Luis).

Deprecations

The `wagtailmenus.app_settings` module is deprecated

If you're importing this in your project from it's previous location, you should update the import statements in your code to use the new import path.

So, instead of:

```
from wagtailmenus import app_settings
```

Do:

```
from wagtailmenus.conf import settings
```

The `wagtailmenus.constants` module is deprecated

If you're importing this in your project from it's previous location, you should update the import statements in your code to use the new import path.

So, instead of:

```
from wagtailmenus import constants
```

Do:

```
from wagtailmenus.conf import constants
```

The WAGTAILMENUS_CHILDREN_MENU_CLASS_PATH setting is deprecated

If you're using this in your project's settings to override the menu class used for rendering children menus, you'll want to change this:

```
# settings/base.py

WAGTAILMENUS_CHILDREN_MENU_CLASS_PATH = 'project.app.module.Class'
```

To this (without the "_PATH" on the end):

```
WAGTAILMENUS_CHILDREN_MENU_CLASS = 'project.app.module.Class'
```

The WAGTAILMENUS_SECTION_MENU_CLASS_PATH setting is deprecated

If you're using this in your project's settings to override the menu class used for rendering section menus, you'll want to change this:

```
# settings/base.py

WAGTAILMENUS_SECTION_MENU_CLASS_PATH = 'project.app.module.Class'
```

To this (without the "_PATH" on the end):

```
WAGTAILMENUS_SECTION_MENU_CLASS = 'project.app.module.Class'
```

Upgrade considerations

FLAT_MENU_MODEL_CLASS has been removed from app settings

If you're referencing `FLAT_MENU_MODEL_CLASS` directly from wagtailmenus' app settings module, then you may need to make some changes.

If you only need the 'model string' for the model (for example, to use in a `ForeignKey` or `ManyToManyField` field definition), you should use `wagtailmenus.get_flat_menu_model_string()` instead.

If you need the Django model class itself, use `wagtailmenus.get_flat_menu_model()`.

MAIN_MENU_MODEL_CLASS has been removed from app settings

If you're referencing `MAIN_MENU_MODEL_CLASS` directly from wagtailmenus' app settings module, then you may need to make some changes.

If you only need the 'model string' for the model (for example, to use in a `ForeignKey` or `ManyToManyField` field definition), you should use `wagtailmenus.get_main_menu_model_string()` instead.

If you need the Django model class itself, use `wagtailmenus.get_main_menu_model()`.

The CHILDREN_MENU_CLASS app setting no longer returns a class

If you're referencing `CHILDREN_MENU_CLASS` attribute on wagtailmenus' app settings module, then you may need to make some changes.

The attribute still exists, but now only returns the import path of the class as a string, rather than the class itself.

If you need to access the class itself, you can use the app settings module's new `get_object()` method, like so:

```
from wagtailmenus.conf import settings

menu_class = settings.get_object('CHILDREN_MENU_CLASS')
```

The `SECTION_MENU_CLASS` app setting no longer returns a class

If you're referencing `SECTION_MENU_CLASS` attribute on wagtailmenus' app settings module, then you may need to make some changes.

The attribute still exists, but now only returns the import path of the class as a string, rather than the class itself.

If you need to access the class itself, you can use the app settings module's new `get_object()` method, like so:

```
from wagtailmenus.conf import settings

menu_class = settings.get_object('SECTION_MENU_CLASS')
```

1.11.5 Wagtailmenus 2.9 release notes

What's new?

Menu tags will now automatically identify level-specific templates

To make it easier to define templates for menus with three or more levels, each menu class has been updated to look for and use level-specific templates in your template directory. For example, for a main menu with three levels, you could arrange your templates like so:

```
templates
└── menus
    └── main
        ├── level_1.html
        ├── level_2.html
        └── level_3.html
```

With this the arrangement, the `{% main_menu %}` tag would find and use `level_1.html` to render the first level of menu items, then any calls made to `{% sub_menu %}` from within that template would use `level_2.html`, then any further calls to `{% sub_menu %}` from that template would use `level_3.html`.

See the updated documentation for more information and examples:

Using preferred paths and names for your templates

New 'sub_menu_templates' menu tag option allows you to specify templates for multiple levels

To complement the new level-specific template functionality, a new `sub_menu_templates` option has been added to the `main_menu`, `flat_menu`, `section_menu` and `children_menu` tags to allow you to specify multiple templates to use at different levels.

For example, if you had a template (e.g `abc.html`) that you'd like to use for the second level of a section menu, and another (e.g. `xyz.html`) that you'd like to use for the third, you could specify that by doing the following:

```
{% section_menu max_levels=3 sub_menu_templates="path_to/abc.html, path_to/xyz.html"
↪%}
```

See the updated documentation for more information:

Template tags reference

Disable 'Main menu' or 'Flat menu' management options in the Wagtail admin area

Implemented by Michael van de Waeter (@mvdwaeter).

Projects come in all shapes and sizes, and not all call for both types of menu. So, to reduce potential confusion for users in the CMS, you can now hide either (or both) the 'Flat menus' and 'Main menu' items from the 'Settings' menu in the admin area (and disables the underlying CMS functionality for each).

See the settings documentation for more information:

WAGTAILMENUS_MAIN_MENUS_EDITABLE_IN_WAGTAILADMIN

WAGTAILMENUS_FLAT_MENUS_EDITABLE_IN_WAGTAILADMIN

1.11.6 Wagtailmenus 2.8 release notes

What's new?

Smarter 'active class' attribution for menu items that link to custom URLs

By default, menu items linking to custom URLs are currently only attributed with the 'active' class if their `link_url` value matches the path of the current request **exactly**, producing unexpected results when GET parameters or fragments are added to the end of a `link_url` value.

A new and improved approach to active class attribution for custom URL links has now been implemented (by Joshua C. Jackson and Andy Babic), which you can enable by adding `WAGTAILMENUS_CUSTOM_URL_SMART_ACTIVE_CLASSES = True` to your project's settings.

With the new approach, only the 'path' part of the `link_url` value is used for comparison with `request.path`, which produces far more consistent results. The new approach also allows the 'ancestor' class to be attributed to

menu items where the `link_url` looks like an ancestor of the current request URL. For example, if a menu item's `link_url` value is `'/news/'`, and the request path is `'/news/boring-news-article/'`, then the menu item will be attributed with the 'ancestor' class.

The previous behaviour is now deprecated in favour of this new approach, and will be removed in version 2.10, with the new approach becoming the default (and only) option.

Minor changes & bug fixes

- Various documentation spelling/formatting corrections (thanks to Sergey Fedoseev and Pierre Manceaux).

Deprecations

- The current 'active class' attribution behaviour for menu items linking to custom URLs is deprecated (see above for details). Add `WAGTAILMENUS_CUSTOM_URL_SMART_ACTIVE_CLASSES = True` to your project's settings to enable the new and improved behaviour and silence the deprecation warning.

Upgrade considerations

Following the standard deprecation period, the following classes, methods and behaviour have been removed:

- Wagtailmenus now unconditionally uses backend-specific templates for rendering, and the `WAGTAILMENUS_USE_BACKEND_SPECIFIC_TEMPLATES` setting is ignored completely. See the 2.6 release notes for more info: http://wagtailmenus.readthedocs.io/en/stable/releases/2.6.0.html#improved-compatibility-with-alternative-template-backends.
- The `get_template_engine()` method has been removed from `wagtailmenus.models.menus.Menu`.
- The `panels` attribute has been removed from the `AbstractMainMenu` and `AbstractFlatMenu` models, as have the `main_menu_panels` and `flat_menu_panels` panel definition values from `wagtailmenus.panels`.

1.11.7 Wagtailmenus 2.7.1 release notes

This is a maintenance release to fix a bug that was resulting in `content_panels` and `setting_panels` being ignored by Wagtail's editing UI when editing main or flat menus.

There have also been a few minor tweaks to the documentation, including:

- Removing the 'alpha' notice and title notation from the 2.7.0 release notes.
- Adding a badge to README.rst to indicate the documentation build status.
- Adding a missing 'migrate' step to the **Developing locally** instructions in the contribution guidelines.
- Updating the code block in the **.po to .mo** conversion step in the packaging guidelines to the `find` command with `execdir`.

1.11.8 Wagtailmenus 2.7 release notes

What's new?

- Added support for Wagtail 2.0 and Django 2.0
- Dropped support for Wagtail versions 1.8 to 1.9
- Dropped support for Django versions 1.5 to 1.10
- Dropped support for Python 2 and 3.3

Minor changes & bug fixes

Various 'Python 3 only' code optimisations:

- Removed __future__ Unicode handling imports throughout.
- Replaced 'old style' class definitions with 'new style' ones.
- Simplified super() syntax throughout to the cleaner Python 3 implementation.
- Fixed deprecation warnings pertaining to use of `TestCase.assertRaisesRegexp`.
- Fixed an issue that was preventing translated field label text appearing for the `handle` field when using the `FLAT_MENUS_HANDLE_CHOICES` setting (Contributed by @jeromelebleu)

Upgrade considerations

- This version only officially supports Wagtail and Django versions from 1.10 to 2.0. If you're using anything earlier than that, you should consider updating your project. If upgrading Wagtail or Django isn't an option, it might be best to stick with wagtailmenus 2.6 for now (which is a LTS release).
- This version also drops support for anything earlier than Python 3.4.

Following the standard deprecation period, the following classes, methods and behaviour has been removed:

- The `wagtailmenus.models.menus.MenuFromRootPage` class was removed.
- The `__init__()` method of `wagtailmenus.models.menus.ChildrenMenu` no longer accepts a **root_page** keyword argument. The parent page should be passed using the **parent_page** keyword instead.
- The **root_page** attribute has been removed from the `wagtailmenus.models.menus.ChildrenMenu` class. Use the **parent_page** attribute instead.
- The `sub_menu` template tag no longer accepts a **stop_at_this_level** keyword argument.
- The `get_sub_menu_items_for_page()` and `prime_menu_items()` methods have been removed from `wagtailmenus.templatetags.menu_tags`.
- The `get_attrs_from_context()` method has been removed from `wagtailmenus.utils.misc`.
- The `get_template_names()` and `get_sub_menu_template_names()` methods have been removed from `wagtailmenus.utils.template` and the redundant `wagtailmenus.utils.template` module removed.

1.11.9 Wagtailmenus 2.6.0 release notes

Note: Wagtailmenus 2.6 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 (for at least 8 months).

Note: Wagtailmenus 2.6 will be the last LTS release to support Python 2 or Python 3.3.

Note: Wagtailmenus 2.6 will be the last LTS release to support Wagtail versions 1.5 to 1.9.

What's new?

Improved compatibility with alternative template backends

Wagtailmenus has been updated to use backend-specific templates for rendering, making it compatible with template backends other than Django's default backend (such as `jinja2`).

Although the likelihood of this new behaviour introducing breaking changes to projects is minute, it is turned **OFF** by default for now, in order to give developers time to make any necessary changes. However, by version 2.8 the updated behaviour will replace the old behaviour completely, becoming non optional.

To start using wagtailmenus with an alternative backend now (or to test your project's compatibility in advance), you can turn the updated behaviour **ON** by adding the following to your project's settings:

```
WAGTAILMENUS_USE_BACKEND_SPECIFIC_TEMPLATES = True
```

Thank you to Nguyn Hng Quân (@hongquan) for contributing this!

New tabbed interface for menu editing

In an effort to improve the menu editing UI, Wagtail's `TabbedInterface` is now used to split a menu's fields into two tabs for editing: **Content** and **Settings**; with the latter including panels for the `max_levels` and `use_specific` fields (which were previously tucked away at the bottom of the edit page), and the former for everything else.

Two new attributes, `content_panels` and `settings_panels` have also been added to `AbstractMainMenu` and `AbstractFlatMenu` to allow the panels for each tab to be updated independently.

If for any reason you don't wish to use the tabbed interface for editing custom menu models, the `panels` attribute is still supported, and will setting that will result in all fields appearing in a single list (as before). However, the `panels`

attribute currently present on the `AbstractFlatMenu` and `AbstractMainMenu` models is now deprecated and will be removed in the future releases (see below for more info).

Built-in compatibility with `wagtail-condensedinlinepanel`

In an effort to improve the menu editing UI, wagtailmenus now has baked-in compatibility with `wagtail-condensedinlinepanel`. As long as a compatible version (at least *0.3*) of the app is installed, wagtailmenus will automatically use `CondensedInlinePanel` instead of Wagtail's built-in `InlinePanel` for listing menu items, giving menu editors some excellent additional features, including drag-and-drop reordering and the ability to add a new item into any position.

If you have custom Menu models in your project that use the `panels` attribute to customise arrangement of fields in the editing UI, you might need to change the your panel list slightly in order to see the improved menu items list after installing. Where you might currently have something like:

```
class CustomMainMenu(AbstractMainMenu):
    ...

    panels = (
        ...
        InlinePanel('custom_menu_items'),
        ..
    )


class CustomFlatMenu(AbstractFlatMenu):
    ...

    panels = (
        ...
        InlinePanel('custom_menu_items'),
        ..
    )
```

You should import `MainMenuItemsInlinePanel` and `FlatMenuItemsInlinePanel` from `wagtailmenus.panels` and use them instead like so:

```
from wagtailmenus.panels import FlatMenuItemsInlinePanel, MainMenuItemsInlinePanel


class CustomMainMenu(AbstractMainMenu):
    ...

    panels = (
        ...
        MainMenuItemsInlinePanel(),  # no need to pass any arguments!
        ..
    )


class CustomFlatMenu(AbstractFlatMenu):
    ...

    panels = (
        ...
        FlatMenuItemsInlinePanel(),  # no need to pass any arguments!
```

(continues on next page)

(continued from previous page)

```
        ..
    )
```

Minor changes & bug fixes

- Updated tests to test compatibility with Wagtail 1.13.

Deprecations

`Menu.get_template_engine()`

This method is deprecated in favour of using Django's generic 'get_template' and 'select_template' methods, which return backend-specific template instances instead of *django.template.Template* instances.

`AbstractMainMenu.panels` and `AbstractFlatMenu.panels`

If you are referencing `AbstractMainMenu.panels` or `AbstractFlatMenu.panels` anywhere, you should update your code to reference the `content_panels` or `settings_panels` attribute instead, depending on which panels you're trying to make use of.

If you're overriding the `panels` attribute on a custom menu model in order to make additional fields available in the editing UI (or change the default field display order), you might also want to think about updating your code to override the `content_panels` and `settings_panels` attributes instead, which will result in fields being split between two tabs (**Content** and **Settings**). However, this is entirely optional.

Upgrade considerations

- Following the standard deprecation period, any `modify_submenu_items()` methods implemented on custom Page type models must now accept a 'use_absolute_page_urls' keyword argument. See the 2.4 release notes for more info: http://wagtailmenus.readthedocs.io/en/stable/releases/2.4.0.html

1.11.10 Wagtailmenus 2.5.2 release notes

There are no changes in this release

1.11.11 Wagtailmenus 2.5.1 release notes

This is a maintenance release to fix a bug that was resulting in Django creating additional migrations for the app when running the `makemigrations` command after changing the project's default language to something other than `"en"`.

Thanks to @philippbosch from A Color Bright for submitting the fix.

1.11.12 Wagtailmenus 2.5.0 release notes

What's new?

Class-based rendering behaviour for menus

This version of wagtailmenus sees quite a large refactor in an attempt to address a large amount of repetition and inconsistency in template tag code, and to also break the process of rendering menus down into more clearly defined steps that can be overridden individually where needed.

While the existing template tags remain, and are still the intended method for initiating rendering of menus from templates, their responsibilities have diminished somewhat. They are now only really responsible for performing basic validation on the option values passed to them - everything else is handled by the relevant Menu class.

The base `Menu` class defines the default 'rendering' logic and establishes a pattern of behaviour for the other classes to follow. Then, the more specific classes simply override the methods they need to produce the same results as before.

Below is an outline of the new process, once the menu tag has prepared it's option values and is ready to hand things over to the menu class:

1. The menu class's `render_from_tag()` method is called. It takes the current context, as well as any 'option values' passed to / prepared by the template tag.
2. `render_from_tag()` calls the class's `get_contextual_vals_from_context()` method, which analyses the current context and returns a `ContextualVals` instance, which will serve as a convenient (read-only) reference for 'contextual' data throughout the rest of the process.
3. `render_from_tag()` calls the class's `get_option_vals_from_options()` method, which analyses the provided option values and returns an `OptionVals` instance, which will serve as a convenient (read-only) reference for 'option' data throughout the rest of the process. The most common attributes are accessible directly (e.g. `opt_vals.max_levels` and `opt_vals.template_name`), but some menu-specific options, or any additional values passed to the tag, will be stored as a dictionary, available as `opt_vals.extra`.
4. `render_from_tag()` calls the class's `get_instance_for_rendering()` method, which takes the prepared `ContextualVals` and `OptionVals` instances, and uses them to get or create and return a relevant instance to use for rendering.
5. In order for the menu instance to handle the rest of the rendering process, it needs to be able to access the `ContextualVals` and `OptionVals` instances that have already been prepared, so those values are passed to the instance's `prepare_to_render()` method, where references to them are saved on the instance as private attributes; `self._contextual_vals` and *self._option_vals*.
6. With access to everything it needs, the instance's `render_to_template()` method is called. This in turn calls two more instance methods.
7. The `get_context_data()` method creates and returns a dictionary of values that need to be available in the template. This includes obvious things such as a list of `menu_items` for the current level, and other not-so-obvious things, which are intended to be picked up by the `sub_menu` tag (if it's being used in the template to render additional levels). The menu items are provided by the `get_menu_items_for_rendering()` method, which in turn splits responsibility for sourcing, priming, and modifying menu items between three other

methods: `get_raw_menu_items()`, `prime_menu_items()` and `modify_menu_items()`, respectively.

8. The `get_template()` method identifies and returns an appropriate `Template` instance that can be used for rendering.
9. With the data and template gathered, `render_to_template()` then converts the data into a `Context` object and sends it to the template's `render()` method, creating a string representation of the menu, which is sent back for inclusion in the original template.

Hooks added to give developers more options for manipulating menus

While wagtailmenus has long supported the use of custom classes for most things (allowing developers to override methods as they see fit), for a long time, I've felt that it should be easier to override some core/shared behaviour without the technical overhead of having to create and maintain multiple custom models and classes. So, wagtailmenus now supports several 'hooks', which allow you to do just that.

They use the hooks mechanism from Wagtail, so you may already be familiar with the concept. For more information and examples, see the new section of the documentation: *Using hooks to modify menus*.

New 'autopopulate_main_menus' command added

The 'autopopulate_main_menus' command has been introduced to help developers integrate wagtailmenus into an existing project, by removing some of the effort that is often needed to populating main menu for each project from scratch. It's been introduced as an extra (optional) step to the instruction in: *Installing wagtailmenus*.

Utilises the new `add_menu_items_for_pages()` method, mentioned below.

New 'add_menu_items_for_pages()' method added for main & flat menus

For each page in the provided `PageQuerySet` a menu item will be added to the menu, linking to that page. The method has was added to the `MenuWithMenuItems` model class, which is subclassed by `AbstractMainMenu` and `AbstractFlatMenu`, so you should be able to use it on custom menu model objects, as well as objects using the default models.

Overriding 'get_base_page_queryset()' now effects top-level menu items too

Previously, if you overrode get_base_page_queryset() on a custom main menu or flat menu model, the page-tree driven part of the menu (anything below the top-level) would respect that, but top-level menu items linking to pages excluded by get_base_page_queryset() would still be rendered.

Now, 'top_level_items' has been refactored to call get_base_page_queryset() to filter down and return page data for items at the top level too, so developers can always expect changes to get_base_page_queryset() to be reflected throughout entire menus.

'MenuItemManager.for_display()' now returns all items, regardless of the status of linked pages

When sourcing data for a main or flat menu, it doesn't make sense to apply two sets of filters relating to pages status/visibility, so 'for_display' now simply returns ALL menu items defined for a menu, and any unsuitable page links are filtered out in a menu instances 'top_level_items' by calling upon 'get_base_page_queryset'.

Minor changes & bug fixes

- Fixed an issue with `runtests.py` that was causing tox builds in Travis CI to report as successful, even when tests were failing. Contributed by Oliver Bestwalter (@obestwalter).
- The `stop_at_this_level` argument for the `sub_menu` tag has been officially deprecated and the feature removed from documentation. It hasn't worked for a few versions and nobody has mentioned it, so this is the first step to removing it completely.
- Made the logic in 'pages_for_display' easier to override on custom menu classes by breaking it out into a separate 'get_pages_for_display()' method (that isn't decorated with `cached_property`).
- Added support for Wagtail 1.12

Upgrade considerations

The ChildrenMenu's 'root_page' attribute is deprecated in favour of 'parent_page'

In previous versions, the ChildrenMenu and SectionMenu classes both extended the same `MenuFromRootPage` class, which takes `root_page` as an init argument, then stores a reference to that page using an attribute of the same name.

The ChildrenMenu class has now been updated to use `parent_page` as an init argument and attribute name instead, which feels like a much better fit. This same terminology has also been adopted for the SubMenu class too.

If you're subclassing the ChildrenMenu class in your project, please update any code referencing *root_page* to use *parent_page* instead. Support for the old name will be removed in version 2.7.

'MenuWithMenuItems.get_base_menuitem_queryset()' no longer filters the queryset

By default, the queryset returned by 'get_base_menuitem_queryset' on menu instances will now return ALL menu items defined for that menu, regardless of the status / visibility of any linked pages.

Previously, the result was filtered to only include pages with 'live' status, and with a True 'show_in_menus' value.

If you're calling 'get_base_menuitem_queryset' anywhere in your project, and are relying on the original method to return the same value as it did before, you will need to apply the additional filters to the queryset, like so:

```
from django.db.models import Q

...

menu_item_qs = menu.get_base_menuitem_queryset()
menu_item_qs = menu_item_qs.filter(
    Q(link_page__isnull=True) |
    Q(link_page__live=True) &
    Q(link_page__expired=False) &
    Q(link_page__show_in_menus=True)
)
```

'MenuItemManager.for_display()' no longer filters the queryset

If you are subclassing `MenuItemManger` to create managers for your custom menu item models, and are relying on the original 'for_display' method to filter out links based on their linked page's status/visibility, you may wish to revise your code to filter out the pages as before, like so:

```
from django.db.models import Q
from wagtailmenus.managers import MenuItemManager

...

class CustomMenuItemManager(MenuItemManager):

    def for_display(self):
        qs = super(CustomMenuItemManager, self).for_display()
        qs = qs.filter(
            Q(link_page__isnull=True) |
            Q(link_page__live=True) &
            Q(link_page__expired=False) &
            Q(link_page__show_in_menus=True)
        )
        # Now apply any custom filters
        ...
        # Return queryset
        return qs
```

The `sub_menu` tag will raise an error if used in a non-menu template

Despite the docs always having stated that the 'sub_menu' tag is only intended for use in menu templates for other types of menu; Up until now, it has functioned similarly to the 'children_menu' tag if used in a regular Django template. But, if you try to call 'sub_menu' from anything other than a menu template now, a `SubMenuUsageError` error will now be raised.

I highly doubt this will trip anybody up, but sorry if it does. Recent versions of Django seem to swallow deprecation warnings when they occur in the course of rendering a template tag, so even if there were a deprecation period for this, the warnings probably wouldn't have been seen by anyone.

`wagtailmenus.models.menus.MenuFromRootPage` is deprecated

With `ChildrenMenu` being refactored to use 'parent_page' as an attribute instead of 'root_page', and the new `SubMenu` menu class taking a similar approach, the `MenuFromRootPage` name only seems relevant to `SectionMenu`, so it has been deprecated in favour of using a more generically-named `MenuFromPage` class, which is subclassed by all three.

`wagtailmenus.menu_tags.prime_menu_items()` is deprecated

The method has been superseded by new logic added to the `Menu` class.

`wagtailmenus.menu_tags.get_sub_menu_items_for_page()` is deprecated

The method has been superseded by new logic added to the `Menu` class.

`wagtailmenus.utils.misc.get_attrs_from_context()` is deprecated

The method has been superseded by new logic added to the `Menu` class.

`wagtailmenus.utils.template.get_template_names()` is deprecated

The method has been superseded by new logic added to the `Menu` class.

`wagtailmenus.utils.template.get_sub_menu_template_names()` is deprecated

The method has been superseded by new logic added to the `Menu` class.

1.11.13 Wagtailmenus 2.4.3 release notes

This is a maintenance release to fix a bug that was resulting in Django creating additional migrations for the app when running the `makemigrations` command after changing the project's default language to something other than "en".

Thanks to @philippbosch from A Color Bright for submitting the fix.

1.11.14 Wagtailmenus 2.4.2 release notes

There are no changes in this release.

1.11.15 Wagtailmenus 2.4.1 release notes

This is a maintenance release to add a migration that should have been included in the previous release, but wasn't. Thanks to Stuart George (@stuartaccent) for reporting and submitting the fix.

If you experience problems after upgrading from 2.4.0 to 2.4.1 (due to your project creating it's own, conflicting migration), please try running `pip uninstall wagtailmenus` first, then install the latest version.

1.11.16 Wagtailmenus 2.4.0 release notes

What's new?

Check out the new documentation!

It's been a long wait, but I finally got around to making it happen. Wagtailmenus now has easily navigable and searchable documentation, kindly hosted by `readthedocs.org`. Find it at http://wagtailmenus.readthedocs.io/

New `get_text_for_repeated_menu_item()` method on `MenuPageMixin` and `MenuPage` models

The new method is called by `get_repeated_menu_item()` to get a string to use to populate the `text` attribute on repeated menu items.

The method is designed to be overridden in cases where the text value needs to come from different fields. e.g. in multilingual site where different translations of 'repeated_item_text' must be surfaced.

By default, if the `repeated_item_text` field is left blank, the `WAGTAILMENUS_PAGE_FIELD_FOR_MENU_ITEM_TEXT` is respected, instead of just returning `Page.title`.

New `use_absolute_page_urls` parameter added to template tags

The new parameter allows you to render menus that use 'absolute' URLs for pages (including the protocol/domain derived from the relevant `wagtailcore.models.Site` object), instead of the 'relative' URLs used by default.

Other minor changes

- Adjusted Meta classes on menu item models so that common behaviour is defined once in AbastractMenuItem.Meta.
- Refactored the AbstractMenuItem's *menu_text* property method to improve code readability, and better handle instances where neither link_text or link_page are set.

Upgrade considerations

The signature of the `modify_submenu_items()` and `get_repeated_menu_item()` methods on `MenuPage` and `MenuPageMixin` models has been updated to accept a new `use_absolute_page_urls` keyword argument.

If you're overriding either of these methods in your project, you should think about updating the signatures of those methods to accept the new argument and pass it through when calling `super()`, like in the following example:

```
from wagtailmenus.models import MenuPage


class ContactPage(MenuPage):
    ...

    def modify_submenu_items(
        self, menu_items, current_page, current_ancestor_ids,
        current_site, allow_repeating_parents, apply_active_classes,
        original_menu_tag, menu_instance, request, use_absolute_page_urls,
    ):
        # Apply default modifications first of all
        menu_items = super(ContactPage, self).modify_submenu_items(
            menu_items, current_page, current_ancestor_ids, current_site, allow_
↪repeating_parents, apply_active_classes, original_menu_tag,
            menu_instance, request, use_absolute_page_urls
        )
        """
        If rendering a 'main_menu', add some additional menu items to the end
        of the list that link to various anchored sections on the same page
        """
```

(continues on next page)

(continued from previous page)

```
        if original_menu_tag == 'main_menu':
            base_url = self.relative_url(current_site)
            menu_items.extend((
                {
                    'text': 'Get support',
                    'href': base_url + '#support',
                    'active_class': 'support',
                },
                {
                    'text': 'Speak to someone',
                    'href': base_url + '#call',
                    'active_class': 'call',
                },
                {
                    'text': 'Map & directions',
                    'href': base_url + '#map',
                    'active_class': 'map',
                },
            ))
        return menu_items

    def get_repeated_menu_item(
        self, current_page, current_site, apply_active_classes,
        original_menu_tag, request, use_absolute_page_urls,
    ):
        item = super(ContactPage, self).get_repeated_menu_item(
            current_page, current_site, apply_active_classes,
            original_menu_tag, request, use_absolute_page_urls,
        )
        item.text = 'Eat. Sleep. Rave. Repeat!'
        return item
```

If you choose NOT to update your versions of those methods to accept the `use_absolute_page_urls` keyword argument, you will continue to see deprecation warnings until version `2.6.0`, when it will be a requirement, and your existing code will no longer work.

You might want to consider adopting a more future-proof approach to overriding the methods from `MenuPage` and `MenuPageMixin`, so that new keyword arguments added in future will be catered for automatically.

Below shows a version of the above code example, modified to use `**kwargs` in methods:

```
from wagtailmenus.models import MenuPage


class ContactPage(MenuPage):
    ...

    def modify_submenu_items(self, menu_items, **kwargs):
        # Apply default modifications first of all
        menu_items = super(ContactPage, self).modify_submenu_items(menu_items,␣
↪**kwargs)
        """
        If rendering a 'main_menu', add some additional menu items to the end
        of the list that link to various anchored sections on the same page
        """
        if kwargs['original_menu_tag'] == 'main_menu':
            base_url = self.relative_url(kwargs['current_site'])
```

(continues on next page)

(continued from previous page)

```
            menu_items.extend((
                {
                    'text': 'Get support',
                    'href': base_url + '#support',
                    'active_class': 'support',
                },
                {
                    'text': 'Speak to someone',
                    'href': base_url + '#call',
                    'active_class': 'call',
                },
                {
                    'text': 'Map & directions',
                    'href': base_url + '#map',
                    'active_class': 'map',
                },
            ))
        return menu_items

    def get_repeated_menu_item(self, current_page, **kwargs):
        item = super(ContactPage, self).get_repeated_menu_item(current_page, **kwargs)
        item.text = 'Eat. Sleep. Rave. Repeat!'
        return item
```

1.11.17 Wagtailmenus 2.3.2 release notes

This is a maintenance release to fix a bug that was resulting in `{% sub_menu %}` being called recursively (until raising a "maximum recursion depth exceeded" exception) if a 'repeated menu item' was added at anything past the second level. Thanks to @pyMan for raising/investigating.

1.11.18 Wagtailmenus 2.3.1 release notes

This is a maintenance release to fix to address the following:

- Code example formatting fixes, and better use of headings in README.md.
- Added 'on_delete=models.CASCADE' to all relationship fields on models where no 'on_delete' behaviour was previously set (Django 2.0 compatibility).
- Marked a missing string for translation (@einsfr)
- Updated translations for Lithuanian, Portuguese (Brazil), and Russian. Many thanks to @mamorim, @treavis and @einsfr!

1.11.19 Wagtailmenus 2.3.0 release notes

What's new?

Introducing the `AbstractLinkPage` model!

The newly added `AbstractLinkPage` model can be easily sub-classed and used in projects to create 'link pages' that act in a similar fashion to menu items when appearing in menus, but can be placed in any part of the page tree.

Find out more about this feature

Introducing the `MenuPageMixin` model!

Most of the functionality from `MenuPage` model has been abstracted out to a `MenuPageMixin` model, that can more easily be mixed in to existing page type models.

MenuPage methods updated to accept `request` keyword argument

The `modify_submenu_items()`, `has_submenu_items()` and `get_repeated_menu_item()` methods on `MenuPageMixin` / `MenuPage` have been updated to accept a new *request* keyword argument, which is used to pass in the current `HttpRequest` that the menu is being rendered for.

All `Menu` classes are now 'request aware'

A new `set_request()` method on all `Menu` classes is used to set a `request` attribute on the `Menu` instance, immediately after initialisation, allowing you to referencing `self.request` from most methods to access the current `HttpRequest` object

Added `get_base_page_queryset()` method to all `Menu` classes

That can be overridden to change the base `QuerySet` used when identifying pages to be included in a menu when rendering. For example developers could use `self.request.user` to only ever include pages that the current user has some permissions for.

Overridable menu classes for `section_menu` and `children_menu` tags

Added the `WAGTAILMENUS_SECTION_MENU_CLASS_PATH` setting, which can be used to override the `Menu` class used when using the `{% section_menu %}` tag.

Added the `WAGTAILMENUS_CHILDREN_MENU_CLASS_PATH` setting, which can be used to override the `Menu` class used when using the `{% children_menu %}` tag.

Other minor changes

- Added wagtail 1.10 and django 1.11 test environments to tox
- Renamed `test_frontend.py` to `test_menu_rendering.py`
- In situations where `request.site` hasn't been set by wagtail's `SiteMiddleware`, the wagtailmenus context processor will now use the default site to generate menus with.
- Updated `AbstractMenuItem.clean()` to only ever return field-specific validation errors, because Wagtail doesn't render non-field errors for related models added to the editor interface using `InlinePanel`.
- Refactored `runtest.py` to accept a `deprecation` argument that can be used to surface deprecation warnings that arise when running tests.
- Added Russian translations (submitted by Alex @einsfr).

Upgrade considerations

Several methods on the `MenuPage` model have been updated to accept a `request` parameter. If you're upgrading to version `2.3.0` from a previous version, it's not necessary to make any changes immediately in order for wagtailmenus to work, but if you're using the `MenuPage` class in your project, and are overriding any of the following methods:

- `modify_submenu_items()`
- `has_submenu_items()`
- `get_repeated_menu_item()`

Then you should think about updating the signatures of those methods to accept the new argument and pass it through when calling `super()`. See the following code for an example:

```
from wagtailmenus.models import MenuPage


class ContactPage(MenuPage):
    ...

    def modify_submenu_items(
        self, menu_items, current_page, current_ancestor_ids,
        current_site, allow_repeating_parents, apply_active_classes,
        original_menu_tag, menu_instance, request
    ):
        # Apply default modifications first of all
        menu_items = super(ContactPage, self).modify_submenu_items(
            menu_items, current_page, current_ancestor_ids, current_site, allow_
↪repeating_parents, apply_active_classes, original_menu_tag,
            menu_instance, request)
        """
        If rendering a 'main_menu', add some additional menu items to the end
        of the list that link to various anchored sections on the same page
        """
        if original_menu_tag == 'main_menu':
            base_url = self.relative_url(current_site)
            """
            Additional menu items can be objects with the necessary attributes,
            or simple dictionaries. `href` is used for the link URL, and `text`
            is the text displayed for each link. Below, I've also used
            `active_class` to add some additional CSS classes to these items,
```

(continues on next page)

(continued from previous page)

```
                so that I can target them with additional CSS
                """
                menu_items.extend((
                    {
                        'text': 'Get support',
                        'href': base_url + '#support',
                        'active_class': 'support',
                    },
                    {
                        'text': 'Speak to someone',
                        'href': base_url + '#call',
                        'active_class': 'call',
                    },
                    {
                        'text': 'Map & directions',
                        'href': base_url + '#map',
                        'active_class': 'map',
                    },
                ))
            return menu_items

        def has_submenu_items(
            self, current_page, allow_repeating_parents, original_menu_tag,
            menu_instance, request
        ):
            """
            Because `modify_submenu_items` is being used to add additional menu
            items, we need to indicate in menu templates that `ContactPage` objects
            do have submenu items in main menus, even if they don't have children
            pages.
            """
            if original_menu_tag == 'main_menu':
                return True
            return super(ContactPage, self).has_submenu_items(
                current_page, allow_repeating_parents, original_menu_tag,
                menu_instance, request)

If you choose NOT to update your versions of those methods to accept the
``request`` keyword argument, you will continue to see deprecation warnings
until version ``2.5.0``, when it will be a requirement, and your existing code
will no longer work.
```

1.11.20 Wagtailmenus 2.2.3 release notes

This is a maintenance release to fix a bug that was resulting in `{% sub_menu %}` being called recursively (until raising a "maximum recursion depth exceeded" exception) if a 'repeated menu item' was added at anything past the second level. Thanks to @pyMan for raising/investigating.

1.11.21 Wagtailmenus 2.2.2 release notes

This is a maintenance release to improve general code quality and make project management easier. There are no compatibility issues to worry about.

What's changed?

- Update codebase to better handle situations where `request` isn't available in the context when rendering, or `request.site` hasn't been set.
- Got the project set up in Transifex (finally!): https://www.transifex.com/rkhleics/wagtailmenus/
- Updated translatable strings throughout the project to use named variable substitution, and unmarked a few exception messages.
- Added Lithuanian translations (submitted by Matas Dailyda).

1.11.22 Wagtailmenus 2.2.1 release notes

This is a maintenance release to improve code quality and project management.

What's changed?

- Updated Travis CI/tox test settings to test against Wagtail 1.9 & Django 1.10.
- Removed a couple of less useful Travis/tox environment tests to help with test speed.
- Made use of `'extras_require'` in `setup.py` to replace multiple requirements files.
- Optimised the app_settings module so that we can ditch the questionable stuff we're doing with global value manipulation on app load (solution inspired by `django-allauth`).
- Added new semantic version solution to the project (inspired by Wagtail).

1.11.23 Wagtailmenus 2.2.0 release notes

What's new?

- Wagtailmenus now makes use of django's built-in `django.template.loader.select_template()` method to provide a more intuitive way for developers to override templates for specific menus without having to explicitly specify alternative templates via settings or via the `template` and `sub_menu_template` options for each menu tag.

  See the updated documentation for each tag for information about where wagtailmenus looks for templates.
- Added the `WAGTAILMENUS_SITE_SPECIFIC_TEMPLATE_DIRS` setting to allow developers to choose to have wagtailmenus look in additional site-specific locations for templates to render menus.
- wagtailmenus no longer errors if a page's `relative_url()` method raises a `TypeError`.
- Brazilian Portuguese language translations added by @MaxKurama.

Upgrade considerations

N/A

1.11.24 Wagtailmenus 2.1.4 release notes

This is a maintenance release to fix a bug that was resulting in `{% sub_menu %}` being called recursively (until raising a "maximum recursion depth exceeded" exception) if a 'repeated menu item' was added at anything past the second level. Thanks to @pyMan for raising/investigating.

1.11.25 Wagtailmenus 2.1.3 release notes

This is a maintenance release to fix a bug in the `section_menu tag` that arose when attempting to apply the correct active class to `section_root` when the `modify_submenu_items()` method has been overridden to return additional items without an `active_class` attribute (like in the example code in README)

1.11.26 Wagtailmenus 2.1.2 release notes

This is a maintenance release to fix a bug that was preventing reordered menu items from retaining their new order after saving (the `Meta` class on the new abstract models had knocked out the `sort_order` ordering from `wagtail.wagtailcore.models.Orderable`).

1.11.27 Wagtailmenus 2.1.1 release notes

This is a maintenance release to fix a bug introduced in 2.1.0 preventing the app from being installed via pip

1.11.28 Wagtailmenus 2.1.0 release notes

There is a know bug with this release when attempting to install using pip. Please update to v2.1.1 instead.

What's new?

- Added official support for wagtail v1.8
- Added `WAGTAILMENUS_MAIN_MENU_MODEL` and `WAGTAILMENUS_FLAT_MENU_MODEL` settings to allow the default main and flat menu models to be swapped out for custom models.
- Added `WAGTAILMENUS_MAIN_MENU_ITEMS_RELATED_NAME` and `WAGTAILMENUS_FLAT_MENU_ITEMS_RELATED_N` settings to allow the default menu item models to be swapped out for custom models.
- Added the `WAGTAILMENUS_PAGE_FIELD_FOR_MENU_ITEM_TEXT` setting to allow developers to specify a page attribute other than *title* to be used to populate the `text` attribute for menu items linking to pages.
- Added German translations by Pierre (@bloodywing).

Upgrade considerations

N/A

1.11.29 Wagtailmenus 2.0.3 release notes

This is a maintenance release to fix a migration related issue raised by @urlsangel: https://github.com/rkhleics/wagtailmenus/issues/85

1.11.30 Wagtailmenus 2.0.2 release notes

This release is broken and shouldn't be use. Skip straight to v2.0.3 instead.

1.11.31 Wagtailmenus 2.0.1 release notes

This is a maintenance release to fix a bug reported by some users when using the `{% main_menu %}` tag without a `max_levels` value.

1.11.32 Wagtailmenus 2.0.0 release notes

What's new?

New `use_specific` and `max_levels` fields for menu models

MainMenu and *FlatMenu* models now have two new fields:

- `use_specific`: To allow the default `use_specific` setting when rendering that menu to be changed via the Wagtail CMS.
- `max_levels`: To allow the default *max_levels* setting when rendering that menu to be changed via the admin area.

Find the field in the collapsed ADVANCED SETTINGS panel at the bottom of the edit form

More `use_specific` options available

The `use_specific` menu tag argument can now be one of 4 integer values, allowing for more fine-grained control over the use of `Page.specific` and `PageQuerySet.specific()` when rendering menu tags.

Developers not using the `MenuPage` model or overriding any of wagtail's `Page` methods involved in URL generation can now enjoy better performance by choosing not to fetch any specific pages at all during rendering. Simply pass ``use_specific=USE_SPECIFIC_OFF` or `use_specific=0` to the tag, or update the `use_specific` field value on your `MainMenu` or `FlatMenu` objects via the Wagtail admin area.

Basic argument validation added to template tags

The `max_levels`, `use_specific`, `parent_page` and `menuitem_or_page` arguments passed to all template tags are now checked to ensure their values are valid, and if not, raise a `ValueError` with a helpful message to aid debugging.

Upgrade considerations

Dropped features

- Dropped support for the `WAGTAILMENUS_DEFAULT_MAIN_MENU_MAX_LEVELS` and `WAGTAILMENUS_DEFAULT_FLAT_MENU_MAX_LEVELS` settings. Default values are now set using the `max_levels` field on the menu objects themselves.
- Dropped support for the `WAGTAILMENUS_DEFAULT_MAIN_MENU_USE_SPECIFIC` and `WAGTAILMENUS_DEFAULT_FLAT_MENU_USE_SPECFIC` settings. Default values are now set using the `use_specific` field on the menu objects themselves.
- The `has_submenu_items()` method on `MenuPage` no longer accepts the *check_for_children* argument.
- The `modify_submenu_items()` and `has_submenu_items()` methods on the `MenuPage` model now both accept an optional `menu_instance` keyword argument.
- Added the `WAGTAILMENUS_ADD_EDITOR_OVERRIDE_STYLES` setting to allow override styles to be disabled.

Run migrations after updating!

New fields have been added to `MainMenu` and `FlatMenu` models, so you'll need to run the migrations for those. Run the following:

```
django manage.py migrate wagtailmenus
```

Setting `max_levels` and `use_specific` on your existing menus

Edit your existing `MainMenu` and `FlatMenu` objects via the Wagtail CMS.

You should see a new collapsed **ADVANCED SETTINGS** panel at the bottom of each form, where both of these fields live.

Default values for `MainMenu` are `max_levels=2` and `use_specific=1`.

Default values for `FlatMenu` are `max_levels=1` and `use_specific=1`.

Switch to using the new `use_specific` options

If you're passing `use_specific=True` or `use_specific=False` to the any of the menu tags, you'll need to change that to one of the following:

- `use_specific=USE_SPECIFIC_OFF` (or `use_specific=0`)
- `use_specific=USE_SPECIFIC_AUTO` (or `use_specific=1`)
- `use_specific=USE_SPECIFIC_TOP_LEVEL` (or `use_specific=2`)

- `use_specific=USE_SPECIFIC_ALWAYS` (or `use_specific=3`)

See the following section of the README for further info: https://github.com/rkhleics/wagtailmenus/blob/v2.0.0/README.md#using-menupage

Changes to `MenuPage.has_submenu_items()` and `MenuPage.modify_submenu_items()`

If you're extending these methods on your custom page types, you will likely need to make a few changes.

Firstly, the `check_for_children` argument is no longer supplied to `has_submenu_items()`, and is will no longer be accepted as a value.

Secondly, both the `modify_submenu_items()` and `has_submenu_items()` methods both accept an optional `menu_instance` argument, which you'll need to also accept.

See the updated section of the README for corrected code examples: https://github.com/rkhleics/wagtailmenus/blob/v2.0.0/README.md#11-manipulating-sub-menu-items-for-specific-page-types

Adding the `context_processor` to settings

If you're upgrading from wagtailmenus version `1.5.1` or lower, you'll need to update your settings to include a context_processor from wagtailmenus. Your `TEMPLATES` setting should look something like the example below:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(PROJECT_ROOT, 'templates'),
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.contrib.auth.context_processors.auth',
                'django.template.context_processors.debug',
                'django.template.context_processors.i18n',
                'django.template.context_processors.media',
                'django.template.context_processors.request',
                'django.template.context_processors.static',
                'django.template.context_processors.tz',
                'django.contrib.messages.context_processors.messages',
                'wagtail.contrib.settings.context_processors.settings',
                'wagtailmenus.context_processors.wagtailmenus',
            ],
        },
    },
]
```

Release notes for versions preceding `v2.0.0` can be found on GitHub: https://github.com/rkhleics/wagtailmenus/releases?after=v2.0.0