Wagtail App Pages Documentation

Release 0.2.16

Marco Westerhof

Sep 09, 2020

Contents:

CHAPTER 1

Wagtail App Pages

This module provides full MVC support for wagtail pages. Using it, it's possible to extend routing for a page by using url configs and views. It addresses the same problem solved by wagtail's own RoutablePageMixin, without breaking clean MVC principles.

- Free software: MIT license
- Documentation: https://wagtail-app-pages.readthedocs.io.

1.1 Features

- Add URL endpoints to wagtail pages by simply providing a url config
- Use regular django views instead of routing methods in the page model
- Enrich (class based) views and request objects, so views always have access to the parent page
- Adds a *reverse()* method to pages, so we can do reverse lookups with respect to the page itself
- Provides a template tag to reverse urls within the same page (automatically detecting parent page if available)
- Full url conf support, including django 2.0's new *path()* urls

CHAPTER 2

Installation

2.1 Stable release

To install Wagtail App Pages, run this command in your terminal:

```
$ pip install wagtail_app_pages
```

This is the preferred method to install Wagtail App Pages, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Wagtail App Pages can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/mwesterhof/wagtail_app_pages
```

Or download the tarball:

```
$ curl -OL https://github.com/mwesterhof/wagtail_app_pages/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use Wagtail App Pages in your project, simply add it to your INSTALLED_APPS:

```
INSTALLED_APPS = [
    # ...
    'wagtail_app_pages',
    # ...
]
```

use the provided context processor:

```
TEMPLATES = [
    {
        # ...
        'OPTIONS': {
            'context_processors': [
                # ...
                'wagtail_app_pages.context_processors.parent_page',
                # ...
            ],
        },
    },
]
```

and use the provided mixin in your page model(s):

```
# myapppage/models.py
from wagtail.core.models import Page
from wagtail_app_pages.models import AppPageMixin


class MyAppPage(AppPageMixin, Page):
    url_config = 'myapppage.urls'
```

When using this mixin, the url_config attribute becomes a requirement. It should provide a dot-separated path to a valid django url config module:

```
# myapppage/urls.py

from django.urls import path

from .views import ViewA, ViewB


urlpatterns = [
    path('A/', ViewA.as_view(), name='view_a'),
    path('B/', ViewB.as_view(), name='view_b'),
]
```

CHAPTER 4

Examples

For a more in-depth example, we'll create a wagtail page implementing a simple blog app. This example is a little contrived, but it will demonstrate some of the more advanced functionality that can be implemented using app pages.

```
# file: blog/models.py
from django.db import models
from wagtail.core.models import Page
from wagtail_app_pages.models import AppPageMixin


class Article(models.Model):
    slug = models.SlugField()
    title = models.CharField(max_length=200)


class BlogPage(AppPageMixin, Page):
    url_config = 'blog.urls'
```

To implement the blog functionality as an extension of the blog page, it references the following url config:

```
# file: blog/urls.py
from django.urls import path

from .views import ArticleCreate, ArticleDetail, ArticleList


urlpatterns = [
    path('articles/', ArticleList.as_view(), name='article_list'),
    path('articles/<slug:slug>/', ArticleDetail.as_view(), name='article_detail'),
    path('article/new/', ArticleCreate.as_view(), name='article_create'),
]
```

This url config is used to add additional endpoints to the page routing. It essentially turns the blog page into a blog *app*.

```
# file: blog/views.py
from django.views.generic import CreateView, DetailView, ListView

from .models import Article


class ArticleDetail(DetailView):
    model = Article


class ArticleList(ListView):
    model = Article


class ArticleCreate(CreateView):
    model = Article

    def get_success_url(self):
        # on successful create, redirect to the article_list url *for the same parent page*
        # if we have more than one BlogPage, each will have its own ArticleCreate view,
        # and this redirect will always happen with respect to the parent BlogPage
        return self.parent_page.reverse('article_list')
```

These views mostly work the way they always do. The *ArticleCreate* view highlights one exception. Any class-based view served through an app-page will have access to a *parent_page* attribute. This will be a reference to the app-page that served it (the BlogPage, in this case).

The previous example also shows how to perform a reverse lookup within the scope of the page. App-pages will have a *reverse* method that does just that.

Performing a reverse lookup from a template within the app is also easy; a template tag is provided for this. It will find the parent page from the context, so there's no need to supply it. This means that if there's more than one BlogPage within a project, these lookups will always return urls that stay within that app's url structure.

The following example shows how to perform these lookups from the template:

```
{% load app_pages_tags %}
<ul>
    {% for article in article_list %}
    <li>
        <a href="{% app_page_url 'article_detail' slug=article.slug %}">{{ article }}</a>
    </li>
    {% endfor %}
</ul>
```

CHAPTER 5

Caveats

There are things to be considered when using this library.

5.1 Root URL or page template?

A page that uses the AppPageMixin, can route to different views using a supplied url config. In the case of the root URL of the page (in other words, the actual page url), 2 different scenarios are possible:

1. if there's a root URL in the pages' url config, it will be used as expected
2. if no root URL is specified for the page, it will be served as any other wagtail page

This means that it's up to the developer who uses this library to decide which is more appropriate for his case.

5.2 Lack of determinism

When using the regular django MTV (Model, Template, View) architecture, the act of calling *reverse()* or using the *{% url %}* template tag is relatively straight forward. We always target a view in a way that should yield (no more than) one url. This will always work, no matter the context.

Since this library introduces a different way to integrate urls and views, it won't always be quite that simple. Consider the following:

```
# file blog/models.py
from django.db import models
from wagtail.core.models import Page
from wagtail_app_pages.models import AppPageMixin


class Article(models.Model):
    slug = models.SlugField()
    title = models.CharField(max_length=200)
```

(continues on next page)

(continued from previous page)

```
class BlogPage(AppPageMixin, Page):
    url_config = 'blog.urls'
```

```
# file blog/urls.py
from django.urls import path

from blog.views import ArticleDetail, ArticleList


urlpatterns = [
    path('articles/', ArticleList.as_view(), name='article_list'),
    path('articles/<slug:slug>/', ArticleDetail.as_view(), name='article_detail'),
]
```

```
# file blog/views.py
from django.views.generic import DetailView, ListView

from .models import Article


class ArticleList(ListView):
    model = Article


class ArticleDetail(DetailView):
    model = Article
```

```
<!-- file blog/article_list.html -->
{% load app_page_tags %}
<ul>
    {% for article in article_list %}
    <li>
        <a href="{% app_page_url "article_detail" slug=article.slug ">{{ article.
↪title }}</a>
    </li>
    {% endfor %}
</ul>
```

This example will attach 2 views to a page, an *article list view* and an *article detail view*. The list view will render links to the detail view for the respective rendered article. The url lookup will happen within the same app page, and will work as desired. Any view served through wagtail_app_pages will have access to the *parent_page* object, available in the context.

If the view is served from outside of the app, the *{% app_page_url %}* won't be able to resolve the url lookup. This is a direct consequence of extending pages with urls, although there are different ways to deal with it.

- *Can we assume that only one blogpage ever exists?* In that case, a very simple custom templatetag could resolve it:

```
# file templatetags/blog_tags.py
@register.simple_tag
def blog_page_url(name, *args, **kwargs):
    blog_page = BlogPage.objects.live().first()
    return blog_page.reverse(name, *args, **kwargs)
```

- *Multiple blogpages exist, but one per site* In this case, one could create a site setting to specify the main blog page for that site. Any code that has access to the request object, could use this to perform a *request.site.blog_page.reverse* Even if there are multiple versions of the blog page for the site, this could be used to find the "default one".
- *Multiple blogpages exist, and we want to use a specific one* If we want all url lookups from a certain page to always use a specific blogpage instance, we could simply link these together in the content. For instance, we could write a ProductsPage to serve all the views of a product catalogue. We could have 2 versions of this products page, one *main* one, that shows all the products, and is set to show up in the menu. Aside from this, we'd have a *promotion* version, configured to only show products with a promotional state. We'll want the main version to be used by default, and the promotional one when linking from the blogpage. This could simply be achieved by adding a parentalkey to the blogpage model, and explicitly linking the pages together.

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/mwesterhof/wagtail_app_pages/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

Wagtail App Pages could always use more documentation, whether as part of the official Wagtail App Pages docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mwesterhof/wagtail_app_pages/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *wagtail_app_pages* for local development.

1. Fork the *wagtail_app_pages* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/wagtail_app_pages.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv wagtail_app_pages
$ cd wagtail_app_pages/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 wagtail_app_pages tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

6.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_wagtail_app_pages
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

CHAPTER 7

Credits

7.1 Development Lead

- Marco Westerhof <westerhof.marco@gmail.com>

7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

History

8.1 0.2.16 (2020-09-08)

- Wagtail 2.10 support

8.2 0.2.15 (2020-06-17)

- Wagtail 2.9 support

8.3 0.2.14 (2020-02-03)

- Wagtail 2.8 support

8.4 0.2.13 (2019-11-13)

- Wagtail 2.7 support

8.5 0.2.12 (2019-10-01)

- Re-integrating multi-site fix

8.6 0.2.11 (2019-09-11)

- Correctly update wagtail 2.6 support in setup.py

8.7 0.2.10 (2019-09-02)

- Wagtail 2.6 support

8.8 0.2.9 (2019-08-12)

- fix issue regarding url resolving in multi-site scenarios

8.9 0.2.8 (2019-06-03)

- Wagtail 2.5 support

8.10 0.2.7 (2019-01-21)

- Wagtail 2.4 support

8.11 0.2.6 (2018-11-06)

- update requirements

8.12 0.2.5 (2018-08-06)

- ensure that parent page will also be available as “page” in the context

8.13 0.2.4 (2018-07-26

- Support for app page revisions

8.14 0.2.3 (2018-05-29)

- Django 1.11 LTS support (issue #1)

8.15 0.2.2 (2018-04-19)

- add context processor to provide parent_page to context

8.16 0.2.1 (2018-04-10)

- fix templatetags missing in dist

8.17 0.2.0 (2018-03-30)

- change to beta

8.18 0.1.0 (2018-03-15)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search