WhiteNoise Documentation

Release 2.0.6

David Evans

March 04, 2016

Contents

Radically simplified static file serving for Python web apps

> **Warning:** This documentation refers to version 2.0.6 of WhiteNoise, which is no longer officially supported.

With a couple of lines of config WhiteNoise allows your web app to serve its own static files, making it a self-contained unit that can be deployed anywhere without relying on nginx, Amazon S3 or any other external service. (Especially useful on Heroku, OpenShift and other PaaS providers.)

It's designed to work nicely with a CDN for high-traffic sites so you don't have to sacrifice performance to benefit from simplicity.

WhiteNoise works with any WSGI-compatible app but has some special auto-configuration features for Django.

WhiteNoise takes care of best-practices for you, for instance:

- Serving gzipped content (handling Accept-Encoding and Vary headers correctly)
- Setting far-future cache headers on content which won't change

Worried that serving static files with Python is horribly inefficient? Still think you should be using Amazon S3? Have a look at the *Infrequently Asked Questions* below.

CHAPTER 1

QuickStart for Django apps

Edit your `wsgi.py` file and wrap your WSGI application like so:

```
from django.core.wsgi import get_wsgi_application
from whitenoise.django import DjangoWhiteNoise

application = get_wsgi_application()
application = DjangoWhiteNoise(application)
```

That's it, you're ready to go.

Want forever-cachable files and gzip support? Just add this to your `settings.py`:

```
STATICFILES_STORAGE = 'whitenoise.django.GzipManifestStaticFilesStorage'
```

For more details, including on setting up CloudFront and other CDNs see the Using WhiteNoise with Django guide.

CHAPTER 2

QuickStart for other WSGI apps

To enable WhiteNoise you need to wrap your existing WSGI application in a WhiteNoise instance and tell it where to find your static files. For example:

```
from whitenoise import WhiteNoise

from my_project import MyWSGIApp

application = MyWSGIApp()
application = WhiteNoise(application, root='/path/to/static/files')
application.add_files('/path/to/more/static/files', prefix='more-files/')
```

And that's it, you're ready to go. For more details see the full documentation.

CHAPTER 3

Compatibility

WhiteNoise works with any WSGI-compatible application and is tested on Python **2.6**, **2.7**, **3.3**, **3.4**, **3.5** and **PyPy**.

DjangoWhiteNoise is tested with Django versions **1.4 — 1.9**

CHAPTER 4

Endorsements

WhiteNoise is being used in Warehouse, the in-development replacement for the PyPI package repository.

Some of Django and pip's core developers have said nice things about it:

> @jezdez: *[WhiteNoise] is really awesome and should be the standard for Django + Heroku*
>
> @dstufft: *WhiteNoise looks pretty excellent.*
>
> @idangazit *Received a positive brainsmack from @_EvansD's WhiteNoise. Vastly smarter than S3 for static assets. What was I thinking before?*

CHAPTER 5

Issues & Contributing

Raise an issue on the GitHub project or feel free to nudge @_EvansD on Twitter.

CHAPTER 6

Infrequently Asked Questions

6.1 Isn't serving static files from Python horribly inefficient?

The short answer to this is that if you care about performance and efficiency then you should be using WhiteNoise behind a CDN like CloudFront. If you're doing *that* then, because of the caching headers WhiteNoise sends, the vast majority of static requests will be served directly by the CDN without touching your application, so it really doesn't make much difference how efficient WhiteNoise is.

That said, WhiteNoise is pretty efficient. Because it only has to serve a fixed set of files it does all the work of finding files and determing the correct headers upfront on initialization. Requests can then be served with little more than a dictionary lookup to find the appropriate response. Also, when used with gunicorn (and most other WSGI servers) the actual business of pushing the file down the network interface is handled by the kernel's very efficient `sendfile` syscall, not by Python.

6.2 Shouldn't I be pushing my static files to S3 using something like Django-Storages?

No, you shouldn't. The main problem with this approach is that Amazon S3 cannot currently selectively serve gzipped content to your users. Gzipping can make dramatic reductions in the bandwidth required for your CSS and JavaScript. But while all browsers in use today can decode gzipped content, your users may be behind crappy corporate proxies or anti-virus scanners which don't handle gzipped content properly. Amazon S3 forces you to choose whether to serve gzipped content to no-one (wasting bandwidth) or everyone (running the risk of your site breaking for certain users).

The correct behaviour is to examine the `Accept-Encoding` header of the request to see if gzip is supported, and to return an appropriate `Vary` header so that intermediate caches know to do the same thing. This is exactly what WhiteNoise does.

The second problem with a push-based approach to handling static files is that it adds complexity and fragility to your deployment process: extra libraries specific to your storage backend, extra configuration and authentication keys, and extra tasks that must be run at specific points in the deployment in order for everythig to work. With the CDN-as-caching-proxy approach that WhiteNoise takes there are just two bits of configuration: your application needs the URL of the CDN, and the CDN needs the URL of your application. Everything else is just standard HTTP semantics. This makes your deployments simpler, your life easier, and you happier.

CHAPTER 7

License

MIT Licensed

7.1 Using WhiteNoise with Django

Note: To use WhiteNoise with a non-Django application see the generic WSGI documentation.

This guide walks you through setting up a Django project with WhiteNoise. In most cases it shouldn't take more than a couple of lines of configuration.

I mention Heroku in a few place as that was the initial use case which prompted me to create WhiteNoise, but there's nothing Heroku-specific about WhiteNoise and the instructions below should apply whatever your hosting platform.

7.1.1 1. Make sure *staticfiles* is configured correctly

If you're familiar with Django you'll know what to do. If you're just getting started with a new Django project (v1.6 and up) then you'll need add the following to the bottom of your `settings.py` file:

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

As part of deploying your application you'll need to run `./manage.py collectstatic` to put all your static files into `STATIC_ROOT`. (If you're running on Heroku then this is done automatically for you.)

In your templates, make sure you're using the static template tag to refer to your static files. For example:

```
{% load static from staticfiles %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

7.1.2 2. Enable WhiteNoise

Edit your `wsgi.py` file and wrap your WSGI application like so:

```
from django.core.wsgi import get_wsgi_application
from whitenoise.django import DjangoWhiteNoise

application = get_wsgi_application()
application = DjangoWhiteNoise(application)
```

That's it – WhiteNoise will now serve your static files. However, to get the best performance you should proceed to step 3 below and enable gzipping and caching.

7.1.3 3. Add gzip and caching support

WhiteNoise comes with a storage backend which automatically takes care of gzipping your files and creating unique names for each version so they can safely be cached forever. To use it, just add this to your `settings.py`:

```
STATICFILES_STORAGE = 'whitenoise.django.GzipManifestStaticFilesStorage'
```

This uses the new ManifestStaticFilesStorage in Django 1.7, with a backport provided automatically for older versions of Django.

Troubleshooting

If you're having problems with the WhiteNoise storage backend, the chances are they're due to the underlying Django storage engine. This is because WhiteNoise only adds a thin wrapper around Django's storage to add gzip support, and because the gzip code is very simple it generally doesn't cause problems.

To test whether the problems are due to WhiteNoise or not, try swapping the WhiteNoise storage backend for the Django one. If you're running Django 1.7 or above, try:

```
STATICFILES_STORAGE = 'django.contrib.staticfiles.storage.ManifestStaticFilesStorage'
```

Or if you're running Django 1.6 or below, try:

```
STATICFILES_STORAGE = 'django.contrib.staticfiles.storage.CachedStaticFilesStorage'
```

If the problems persist then your issue is with Django itself (try the docs or the mailing list). If the problem only occurs with WhiteNoise then raise a ticket on the issue tracker.

7.1.4 4. Use a Content-Delivery Network *(optional)*

The above steps will get you decent performance on moderate traffic sites, however for higher traffic sites, or sites where performance is a concern you should look at using a CDN.

Because WhiteNoise sends appropriate cache headers with your static content, the CDN will be able to cache your files and serve them without needing to contact your application again.

Below are instruction for setting up WhiteNoise with Amazon CloudFront, a popular choice of CDN. The process for other CDNs should look very similar though.

Instructions for Amazon CloudFront

Go to CloudFront section of the AWS Web Console, and click "Create Distribution". Put your application's domain (without the http prefix) in the "Origin Domain Name" field and leave the rest of the settings as they are.

It might take a few minutes for your distribution to become active. Once it's ready, copy the distribution domain name into your `settings.py` file so it looks something like this:

```
STATIC_HOST = '//d4663kmspf1sqa.cloudfront.net' if not DEBUG else ''
STATIC_URL = STATIC_HOST + '/static/'
```

Or, even better, you can avoid hardcoding your CDN into your settings by doing something like this:

```
STATIC_HOST = os.environ.get('DJANGO_STATIC_HOST', '')
STATIC_URL = STATIC_HOST + '/static/'
```

This way you can configure your CDN just by setting an environment variable. For apps on Heroku, you'd run this command

```
heroku config:set DJANGO_STATIC_HOST=//d4663kmspf1sqa.cloudfront.net
```

Restricting CloudFront to static files

Note: By default your entire site will be accessible via the CloudFront URL. It's possible that this can cause SEO problems if these URLs start showing up in search results. You can restrict CloudFront to only proxy your static files by following the directions below.

---

1. Go to your newly created distribution and click "*Distribution Settings*", then the "*Behaviors*" tab, then "*Create Behavior*". Put `static/*` into the path pattern and click "*Create*" to save.

2. Now select the `Default (*)` behaviour and click "*Edit*". Set "*Restrict Viewer Access*" to "*Yes*" and then click "*Yes, Edit*" to save.

3. Check that the `static/*` pattern is first on the list, and the default one is second. This will ensure that requests for static files are passed through but all others are blocked.

7.1.5 Available Settings

The DjangoWhiteNoise class takes all the same configuration options as the WhiteNoise base class, but rather than accepting keyword arguments to its constructor it uses Django settings. The setting names are just the keyword arguments uppercased with a 'WHITENOISE_' prefix.

WHITENOISE_ROOT

> **Default** `None`
>
> Absolute path to a directory of files which will be served at the root of your application (ignored if not set).
>
> Don't use this for the bulk of your static files because you won't benefit from cache versioning, but it can be convenient for files like `robots.txt` or `favicon.ico` which you want to serve at a specific URL.

WHITENOISE_AUTOREFRESH

> **Default** `settings.DEBUG`
>
> Recheck the filesystem to see if any files have changed before responding. This is designed to be used in development where it can be convenient to pick up changes to static files without restarting the server. For both performance and security reasons, this setting should not be used in production.

WHITENOISE_USE_FINDERS

> **Default** `settings.DEBUG`
>
> Instead of only picking up files collected into `STATIC_ROOT`, find and serve files in their original directories using Django's "finders" API. This is the same behaviour as `runserver` provides by default, and is only useful if you don't want to use the default `runserver` configuration in development.

WHITENOISE_MAX_AGE

> **Default** `60 if not settings.DEBUG else 0`

Time (in seconds) for which browsers and proxies should cache **non-versioned** files.

Versioned files (i.e. files which have been given a unique name like *base.a4ef2389.css* by including a hash of their contents in the name) are detected automatically and set to be cached forever.

The default is chosen to be short enough not to cause problems with stale versions but long enough that, if you're running WhiteNoise behind a CDN, the CDN will still take the majority of the strain during times of heavy load.

WHITENOISE_CHARSET

Default `settings.FILE_CHARSET` (utf-8)

Charset to add as part of the `Content-Type` header for all files whose mimetype allows a charset.

WHITENOISE_ALLOW_ALL_ORIGINS

Default `True`

Toggles whether to send an `Access-Control-Allow-Origin: *` header for all static files.

This allows cross-origin requests for static files which means your static files will continue to work as expected even if they are served via a CDN and therefore on a different domain. Without this your static files will *mostly* work, but you may have problems with fonts loading in Firefox, or accessing images in canvas elements, or other mysterious things.

The W3C explicity state that this behaviour is safe for publicly accessible files.

WHITENOISE_GZIP_EXCLUDE_EXTENSIONS

Default `('jpg', 'jpeg', 'png', 'gif', 'webp','zip', 'gz', 'tgz', 'bz2', 'tbz', 'swf', 'flv', 'woff')`

File extensions to skip when gzipping.

Because the gzip process will only create compressed files where this results in an actual size saving, it would be safe to leave this list empty and attempt to gzip all files. However, for files which we're confident won't benefit from compression, it speeds up the process if we just skip over them.

7.2 Using WhiteNoise with any WSGI application

Note: These instructions apply to any WSGI application. However, for Django applications you would be better off using the DjangoWhiteNoise class which makes integration easier.

To enable WhiteNoise you need to wrap your existing WSGI application in a WhiteNoise instance and tell it where to find your static files. For example:

```
from whitenoise import WhiteNoise

from my_project import MyWSGIApp

application = MyWSGIApp()
application = WhiteNoise(application, root='/path/to/static/files')
application.add_files('/path/to/more/static/files', prefix='more-files/')
```

On initialization, WhiteNoise walks over all the files in the directories that have been added (descending into sub-directories) and builds a list of available static files. Any requests which match a static file get served by WhiteNoise, all others are passed through to the original WSGI application.

See the sections on *gzip handling* and *caching* for further details.

7.2.1 WhiteNoise API

class **WhiteNoise** (*application*, *root=None*, *prefix=None*, ***kwargs*)

Parameters

- **application** (*callable*) – Original WSGI application
- **root** (*str*) – If set, passed to `add_files` method
- **prefix** (*str*) – If set, passed to `add_files` method
- ****kwargs** – Sets *configuration attributes* for this instance

`WhiteNoise.`**add_files** (*root*, *prefix=None*, *followlinks=False*)

Parameters

- **root** (*str*) – Absolute path to a directory of static files to be served
- **prefix** (*str*) – If set, the URL prefix under which the files will be served. Trailing slashes are automatically added.
- **followlinks** (*bool*) – Whether to follow directory symlinks when walking the directory tree to find files. Note that symlinks to files will always work.

7.2.2 Gzip Support

When WhiteNoise builds its list of available files it checks for a corresponding file with a `.gz` suffix (e.g., `scripts/app.js` and `scripts/app.js.gz`). If it finds one, it will assume that this is a gzip-compressed version of the original file and it will serve this in preference to the uncompressed version where clients indicate that they accept gzipped content (see note on Amazon S3 for why this behavour is important).

WhiteNoise comes with a command line utility which will generate gzipped versions of your files for you. Usage is simple:

```
$ python -m whitenoise.gzip --help

usage: gzip.py [-h] [-q] root [extensions [extensions ...]]

Search for all files inside <root> *not* matching <extensions> and produce
gzipped versions with a '.gz' suffix (as long this results in a smaller file)

positional arguments:
  root         Path root from which to search for files
  extensions   File extensions to exclude from gzipping (default: jpg, jpeg,
               png, gif, zip, gz, tgz, bz2, tbz, swf, flv, woff)

optional arguments:
  -h, --help   show this help message and exit
  -q, --quiet  Don't produce log output (default: False)
```

You can either run this during development and commit your compressed files to your repository, or you can run this as part of your build and deploy processs. (Note that DjangoWhiteNoise handles this automatically, if you're using the custom storage backend.)

7.2.3 Caching Headers

By default, WhiteNoise sets a max-age header on all responses it sends. You can configure this by passing a `max_age` keyword argument.

Most modern static asset build systems create uniquely named versions of each file. This results in files which are immutable (i.e., they can never change their contents) and can therefore by cached indefinitely. In order to take advantage of this, WhiteNoise needs to know which files are immutable. This can be done by sub-classing WhiteNoise and overriding the following method:

```
def is_immutable_file(self, static_file, url):
    return False
```

The exact details of how you implement this method will depend on your particular asset build system (see the source for DjangoWhiteNoise for inspiration).

Once you have implemented this, any files which are flagged as immutable will have 'cache forever' headers set.

7.2.4 Using a Content Distribution Network

See the instructions for *using a CDN with Django* . The same principles apply here although obviously the exact method for generating the URLs for your static files will depend on the libraries you're using.

7.2.5 Configuration attributes

These can be set by passing keyword arguments to the constructor, or by sub-classing WhiteNoise and setting the attributes directly.

autorefresh

Default `False`

Recheck the filesystem to see if any files have changed before responding. This is designed to be used in development where it can be convenient to pick up changes to static files without restarting the server. For both performance and security reasons, this setting should not be used in production.

max_age

Default `60`

Time (in seconds) for which browsers and proxies should cache files.

The default is chosen to be short enough not to cause problems with stale versions but long enough that, if you're running WhiteNoise behind a CDN, the CDN will still take the majority of the strain during times of heavy load.

charset

Default `utf-8`

Charset to add as part of the `Content-Type` header for all files whose mimetype allows a charset.

allow_all_origins

Default `True`

Toggles whether to send an `Access-Control-Allow-Origin: *` header for all static files.

This allows cross-origin requests for static files which means your static files will continue to work as expected even if they are served via a CDN and therefore on a different domain. Without this your static files will *mostly* work, but you may have problems with fonts loading in Firefox, or accessing images in canvas elements, or other mysterious things.

The W3C explicity state that this behaviour is safe for publicly accessible files.

7.3 Change Log

7.3.1 v2.0.6

- Rebuild with latest version of *wheel* to get *extras_require* support.

7.3.2 v2.0.5

- Add missing argparse dependency for Python 2.6 (thanks @movermeyer)).

7.3.3 v2.0.4

- Report path on MissingFileError (thanks @ezheidtmann).

7.3.4 v2.0.3

- Add *__version__* attribute.

7.3.5 v2.0.2

- More helpful error message when STATIC_URL is set to the root of a domain (thanks @dominicrodger).

7.3.6 v2.0.1

- Add support for Python 2.6.
- Add a more helpful error message when attempting to import DjangoWhiteNoise before *DJANGO_SETTINGS_MODULE* is defined.

7.3.7 v2.0

- Add an *autorefresh* mode which picks up changes to static files made after application startup (for use in development).
- Add a *use_finders* mode for DjangoWhiteNoise which finds files in their original directories without needing them collected in *STATIC_ROOT* (for use in development). Note, this is only useful if you don't want to use Django's default runserver behaviour.
- Remove the *follow_symlinks* argument from *add_files* and now always follow symlinks.
- Support extra mimetypes which Python doesn't know about by default (including .woff2 format)
- Some internal refactoring. Note, if you subclass WhiteNoise to add custom behaviour you may need to make some small changes to your code.

7.3.8 v1.0.6

- Fix unhelpful exception inside *make_helpful_exception* on Python 3 (thanks @abbottc).

7.3.9 v1.0.5

- Fix error when attempting to gzip empty files (thanks @ryanrhee).

7.3.10 v1.0.4

- Don't attempt to gzip `.woff` files as they're already compressed.
- Base decision to gzip on compression ratio achieved, so we don't incur gzip overhead just to save a few bytes.
- More helpful error message from `collectstatic` if CSS files reference missing assets.

7.3.11 v1.0.3

- Fix bug in Last Modified date handling (thanks to Atsushi Odagiri for spotting).

7.3.12 v1.0.2

- Set the default max_age parameter in base class to be what the docs claimed it was.

7.3.13 v1.0.1

- Fix path-to-URL conversion for Windows.
- Remove cruft from packaging manifest.

7.3.14 v1.0

- First stable release.

A

C

M

W