WhiteNoise Documentation

Release 4.0b2

David Evans

Jul 19, 2017

Contents

Radically simplified static file serving for Python web apps

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 compressed content (gzip and Brotli formats, 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 `settings.py` file and add WhiteNoise to the `MIDDLEWARE_CLASSES` list, above all other middleware apart from Django's SecurityMiddleware:

```
MIDDLEWARE = [
  # 'django.middleware.security.SecurityMiddleware',
  'whitenoise.middleware.WhiteNoiseMiddleware',
  # ...
]
```

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

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

```
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

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

Using WhiteNoise with Flask

WhiteNoise was not specifically written with Flask in mind, but as Flask uses the standard WSGI protocol it is easy to integrate with WhiteNoise (see the *Using WhiteNoise with Flask* guide).

CHAPTER 4

Compatibility

WhiteNoise works with any WSGI-compatible application and is tested on Python **2.7**, **3.3** – **3.6** and **PyPy**, on both Linux and Windows.

Django WhiteNoiseMiddlware is tested with Django versions **1.8** — **1.11**

CHAPTER 5

Endorsements

WhiteNoise owes its initial popularity to the nice things that some of Django and pip's core developers said 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?*

It's now being used by thousands of projects, including some high-profile sites such as mozilla.org.

CHAPTER 6

Issues & Contributing

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

CHAPTER 7

Infrequently Asked Questions

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 determining 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.

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 compressed content to your users. Compression (using either the venerable gzip or the more modern brotli algorithms) can make dramatic reductions in the bandwidth required for your CSS and JavaScript. But in order to do this correctly the server needs to examine the `Accept-Encoding` header of the request to determine which compression formats are supported, and return an appropriate `Vary` header so that intermediate caches know to do the same. This is exactly what WhiteNoise does, but Amazon S3 currently provides no means of doing this.

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 everything 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.

What's the point in WhiteNoise when I can do the same thing in a few lines of Apache/nginx config?

There are two answers here. One is that WhiteNoise is designed to work in situations were Apache, nginx and the like aren't easily available. But more importantly, it's easy to underestimate what's involved in serving static files correctly. Does your few lines of nginx config distinguish between files which might change and files which will never change and set the cache headers appropriately? Did you add the right CORS headers so that your fonts load correctly when served via a CDN? Did you turn on the special nginx setting which allows it to send gzipped content in response to an `HTTP/1.0` request, which for some reason CloudFront still uses? Did you install the extension which allows you to serve pre-compressed brotli-encoded content to modern browsers?

None of this is rocket science, but it's fiddly and annoying and WhiteNoise takes care of all it for you.

CHAPTER 8

License

MIT Licensed

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.

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 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 Django 1.9 and older, 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!" />
```

In Django 1.10 and later, you can use `{% load static %}` instead.

2. Enable WhiteNoise

Edit your `settings.py` file and add WhiteNoise to the `MIDDLEWARE` list. The WhiteNoise middleware should be placed directly after the Django SecurityMiddleware and before all other middleware:

```
MIDDLEWARE = [
  'django.middleware.security.SecurityMiddleware',
  'whitenoise.middleware.WhiteNoiseMiddleware',
  # ...
]
```

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 compression and caching.

3. Add compression and caching support

WhiteNoise comes with a storage backend which automatically takes care of compressing 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.storage.CompressedManifestStaticFilesStorage'
```

If you need to compress files outside of the static files storage system you can use the supplied *command line utility*

Note: If you are having problems after switching to the WhiteNoise storage backend please see the *troubleshooting guide*.

Brotli compression

As well as the common gzip compression format, WhiteNoise supports the newer, more efficient brotli format. This helps reduce bandwidth and increase loading speed. To enable brotli compression you will need the brotlipy Python package installed, usually by running `pip install brotlipy` and updating your `requirements.txt` file.

Brotli is supported by Firefox, Chrome and no doubt other browsers too. WhiteNoise will only serve brotli data to browsers which request it so there are no compatibility issues with enabling brotli support.

Also note that browsers will only request brotli data over an HTTPS connection.

4. Use a Content-Delivery Network

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 = 'https://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=https://d4663kmspf1sqa.cloudfront.net
```

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 *these directions*.

---

5. Using WhiteNoise in development

In development Django's `runserver` automatically takes over static file handling. In most cases this is fine, however this means that some of the improvements that WhiteNoise makes to static file handling won't be available in development and it opens up the possibility for differences in behaviour between development and production environments. For this reason it's a good idea to use WhiteNoise in development as well.

You can disable Django's static file handling and allow WhiteNoise to take over simply by passing the `--nostatic` option to the `runserver` command, but you need to remember to add this option every time you call `runserver`. An easier way is to edit your `settings.py` file and add `whitenoise.runserver_nostatic` to the top of your `INSTALLED_APPS` list:

```
INSTALLED_APPS = [
    'whitenoise.runserver_nostatic',
    'django.contrib.staticfiles',
    # ...
]
```

Note: In older versions of WhiteNoise (below v4.0) it was not possible to use `runserver_nostatic` with Channels as Channels provides its own implementation of runserver. Newer versions of WhiteNoise do not have this problem and will work with Channels or any other third-party app that provides its own implementation of runserver.

---

6. Index Files

When the *`WHITENOISE_INDEX_FILE`* option is enabled:

- Visiting `/example/` will serve the file at `/example/index.html`
- Visiting `/example` will redirect (302) to `/example/`
- Visitng `/example/index.html` will redirect (302) to `/example/`

If you want to something other than `index.html` as the index file, then you can also set this option to an alternative filename.

Available Settings

The WhiteNoiseMiddlware 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_INDEX_FILE

Default `False`

If `True` enable *index file serving*. If set to a non-empty string, enable index files and use that string as the index file name.

WHITENOISE_MIMETYPES

Default `None`

A dictionary mapping file extensions (lowercase) to the mimetype for that extension. For example:

```
{'.foo': 'application/x-foo'}
```

Note that WhiteNoise ships with its own default set of mimetypes and does not use the system-supplied ones (e.g. `/etc/mime.types`). This ensures that it behaves consistently regardless of the environment in which it's run. View the defaults in the media_types.py file.

In addition to file extensions, mimetypes can be specified by supplying the entire filename, for example:

```
{'some-special-file': 'application/x-custom-type'}
```

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 explicitly state that this behaviour is safe for publicly accessible files.

WHITENOISE_SKIP_COMPRESS_EXTENSIONS

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

File extensions to skip when compressing.

Because the compression 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 compress 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.

WHITENOISE_ADD_HEADERS_FUNCTION

Default `None`

Reference to a function which is passed the headers object for each static file, allowing it to modify them.

For example:

```
def force_download_pdfs(headers, path, url):
    if path.endswith('.pdf'):
        headers['Content-Disposition'] = 'attachment'

WHITENOISE_ADD_HEADERS_FUNCTION = force_download_pdfs
```

The function is passed:

headers A wsgiref.headers instance (which you can treat just as a dict) containing the headers for the current file

path The absolute path to the local file

url The host-relative URL of the file e.g. `/static/styles/app.css`

The function should not return anything; changes should be made by modifying the headers dictionary directly.

WHITENOISE_IMMUTABLE_FILE_TEST

Default See immutable_file_test in source

Reference to a function which is passed the path and URL for each static file and should return whether that file is immutable, i.e. guaranteed not to change, and so can be safely cached forever. The default is designed to

work with Django's ManifestStaticFilesStorage backend, and any derivatives of that, so you should only need to change this if you are using a different system for versioning your static files.

Example:

```
def immutable_file_test(path, url):
    # Match filename with 12 hex digits before the extension
    # e.g. app.db8f2edc0c8a.js
    return re.match(r'^.+\.[0-9a-f]{12}\..+$', url)

WHITENOISE_IMMUTABLE_FILE_TEST = immutable_file_test
```

The function is passed:

path The absolute path to the local file

url The host-relative URL of the file e.g. `/static/styles/app.css`

WHITENOISE_STATIC_PREFIX

Default Path component of `settings.STATIC_URL`

The URL prefix under which static files will be served.

Usually this can be determined automatically by using the path component of `STATIC_URL`. So if `STATIC_URL` is `https://example.com/static/` then `WHITENOISE_STATIC_PREFIX` will be `/static/`. However there are cases where it's useful to set these independently, for instance if the application is not running at the root of the domain or if your CDN is doing path rewriting.

WHITENOISE_KEEP_ONLY_HASHED_FILES

Default `False`

Stores only files with hashed names in `STATIC_ROOT`.

By default, Django's hashed static files system creates two copies of each file in `STATIC_ROOT`: one using the original name, e.g. `app.js`, and one using the hashed name, e.g. `app.db8f2edc0c8a.js`. If WhiteNoise's compression backend is being used this will create another two copies of each of these files (using Gzip and Brotli compression) resulting in six output files for each input file.

In some deployment scenarios it can be important to reduce the size of the build artifact as much as possible. This setting removes the "un-hashed" version of the file (which should be not be referenced in any case) which should reduce the space required for static files by half.

Note, this setting is only effective if the WhiteNoise storage backend is being used.

Additional Notes

Django Compressor

For performance and security reasons WhiteNoise does not check for new files after startup (unless using Django *DEBUG* mode). As such, all static files must be generated in advance. If you're using Django Compressor, this can be performed using its offline compression feature.

Serving Media Files

WhiteNoise is not suitable for serving user-uploaded "media" files. For one thing, as described above, it only checks for static files at startup and so files added after the app starts won't be seen. More importantly though, serving user-uploaded files from the same domain as your main application is a security risk (this blog post from Google security describes the problem well). And in addition to that, using local disk to store and serve your user media makes it harder to scale your application across multiple machines.

For all these reasons, it's much better to store files on a separate dedicated storage service and serve them to users from there. The django-storages library provides many options e.g. Amazon S3, Azure Storage, and Rackspace CloudFiles.

Troubleshooting the WhiteNoise Storage backend

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 compression support, and because the compression code is very simple it generally doesn't cause problems.

The most common issue is that there are CSS files which reference other files (usually images or fonts) which don't exist at that specified path. When Django attempts to rewrite these references it looks for the corresponding file and throws an error if it can't find it.

To test whether the problems are due to WhiteNoise or not, try swapping the WhiteNoise storage backend for the Django one:

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

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.

Restricting CloudFront to static files

The instructions for setting up CloudFront given above will result in the entire site being 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 these directions:

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.

Using other storage backends

WhiteNoise will only work with storage backends that stores their files on the local filesystem in `STATIC_ROOT`. It will not work with backends that store files remotely, for instance on Amazon S3.

WhiteNoise makes my tests run slow!

WhiteNoise is designed to do as much work as possible upfront when the application starts so that it can serve files as efficiently as possible while the application is running. This makes sense for long-running production processes, but you might find that the added startup time is a problem during test runs when application instances are frequently being created and destroyed.

The simplest way to fix this is to make sure that during testing the `WHITENOISE_AUTOREFRESH` setting is set to `True`. (By default it is `True` when `DEBUG` is enabled and `False` otherwise.) This stops WhiteNoise from scanning your static files on start up but other than that its behaviour should be exactly the same.

It is also worth making sure you don't have unnecessary files in your `STATIC_ROOT` directory. In particular, be careful not to include a `node_modules` directory which can contain a very large number of files and significantly slow down your application startup. If you need to include specific files from `node_modules` then you can create symlinks from within your static directory to just the files you need.

Using WhiteNoise with Webpack / Browserify / $LATEST_JS_THING

A simple technique for integrating any frontend build system with Django is to use a directory layout like this:

```
./static_src
       ↓
  $ ./node_modules/.bin/webpack
       ↓
./static_build
       ↓
  $ ./manage.py collectstatic
       ↓
./static_root
```

Here `static_src` contains all the source files (JS, CSS, etc) for your project. Your build tool (which can be Webpack, Browserify or whatever you choose) then processes these files and writes the output into `static_build`.

The path to the `static_build` directory is added to `settings.py`:

```
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static_build')
]
```

This means that Django can find the processed files, but doesn't need to know anything about the tool which produced them.

The final `manage.py collectstatic` step writes "hash-versioned" and compressed copies of the static files into `static_root` ready for production.

Note, both the `static_build` and `static_root` directories should be excluded from version control (e.g. through `.git-ignore`) and only the `static_src` directory should be checked in.

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 *WhiteNoiseMiddleware* 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 *compression* and *caching* for further details.

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*)

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.

Compression Support

When WhiteNoise builds its list of available files it checks for corresponding files with a `.gz` and a `.br` suffix (e.g., `scripts/app.js`, `scripts/app.js.gz` and `scripts/app.js.br`). If it finds them, it will assume that they are (respectively) gzip and brotli compressed versions of the original file and it will serve them in preference to the uncompressed version where clients indicate that they that compression format (see note on Amazon S3 for why this behaviour is important). WhiteNoise comes with a command line utility which will generate compressed versions of your files for you. Note that in order for brotli compression to work the brotlipy Python package must be installed.

Usage is simple:

```
$ python -m whitenoise.compress --help
usage: compress.py [-h] [-q] [--no-gzip] [--no-brotli]
                   root [extensions [extensions ...]]

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

```
positional arguments:
  root         Path root from which to search for files
  extensions   File extensions to exclude from compression (default: jpg,
               jpeg, png, gif, webp, zip, gz, tgz, bz2, tbz, swf, flv, woff,
               woff2)

optional arguments:
  -h, --help   show this help message and exit
  -q, --quiet  Don't produce log output
  --no-gzip    Don't produce gzip '.gz' files
  --no-brotli  Don't produce brotli '.br' files
```

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 processes. (Note that this is handled automatically in Django if you're using the custom storage backend.)

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.

WhiteNoise sets both `Last-Modified` and `ETag` headers for all files and will return Not Modified responses where appropriate. The ETag header uses the same format as nginx which is based on the size and last-modified time of the file. If you want to use a different scheme for generating ETags you can set them via you own function by using the *`add_headers_function`* option.

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 using the *`immutable_file_test`* option which accepts a reference to a function.

The exact details of how you implement this method will depend on your particular asset build system but see the *`option documentation`* for a simple example.

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

Index Files

When the *`index_file`* option is enabled:

- Visiting `/example/` will serve the file at `/example/index.html`
- Visiting `/example` will redirect (302) to `/example/`
- Visitng `/example/index.html` will redirect (302) to `/example/`

If you want to something other than `index.html` as the index file, then you can also set this option to an alternative filename.

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.

Redirecting to HTTPS

WhiteNoise does not handle redirection itself, but works well alongside wsgi-sslify, which performs HTTP to HTTPS redirection as well as optionally setting an HSTS header. Simply wrap the WhiteNoise WSGI application with `sslify()` - see the wsgi-sslify documentation for more details.

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.

`index_file`

Default `False`

If `True` enable *index file serving*. If set to a non-empty string, enable index files and use that string as the index file name.

`mimetypes`

Default `None`

A dictionary mapping file extensions (lowercase) to the mimetype for that extension. For example:

```
{'.foo': 'application/x-foo'}
```

Note that WhiteNoise ships with its own default set of mimetypes and does not use the system-supplied ones (e.g. `/etc/mime.types`). This ensures that it behaves consistently regardless of the environment in which it's run. View the defaults in the media_types.py file.

In addition to file extensions, mimetypes can be specified by supplying the entire filename, for example:

```
{'some-special-file': 'application/x-custom-type'}
```

`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 explicitly state that this behaviour is safe for publicly accessible files.

`add_headers_function`

Default `None`

Reference to a function which is passed the headers object for each static file, allowing it to modify them.

For example:

```
def force_download_pdfs(headers, path, url):
    if path.endswith('.pdf'):
        headers['Content-Disposition'] = 'attachment'

application = WhiteNoise(application,
                         add_headers_function=force_download_pdfs)
```

The function is passed:

headers A wsgiref.headers instance (which you can treat just as a dict) containing the headers for the current file

path The absolute path to the local file

url The host-relative URL of the file e.g. `/static/styles/app.css`

The function should not return anything; changes should be made by modifying the headers dictionary directly.

`immutable_file_test`

Default `return False`

Reference to a function which is passed the path and URL for each static file and should return whether that file is immutable, i.e. guaranteed not to change, and so can be safely cached forever.

Example:

```
def immutable_file_test(path, url):
    # Match filename with 12 hex digits before the extension
    # e.g. app.db8f2edc0c8a.js
    return re.match(r'^.+\.[0-9a-f]{12}\..+$', url)
```

The function is passed:

path The absolute path to the local file

url The host-relative URL of the file e.g. `/static/styles/app.css`

Using WhiteNoise with Flask

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

1. Make sure where your *static* is located

If you're familiar with Flask you'll know what to do. If you're just getting started with a new Flask project then the default is the `static` folder in the root path of the application.

Check the `static_folder` argument in Flask Application Object documentation for further information.

2. Enable WhiteNoise

In the file where you create your app you instantiate Flask Application Object (the `flask.Flask()` object). All you have to do is to wrap it with `WhiteNoise()` object.

If you use Flask quick start approach it will look something like that:

```
from flask import Flask
from whitenoise import WhiteNoise

app = WhiteNoise(Flask(__name__), root='static/')
```

If you opt for the pattern of creating your app with a function, then it would look like that:

```
from flask import Flask
from sqlalchemy import create_engine
from whitenoise import WhiteNoise

from myapp import config
from myapp.views import frontend

def create_app(database_uri, debug=False):
    app = Flask(__name__)
    app.debug = debug

    # set up your database
    app.engine = create_engine(database_uri)

    # register your blueprints
    app.register_blueprint(frontend)

    # add whitenoise
    app.wsgi_app = WhiteNoise(app.wsgi_app, root='static/')

    # other setup tasks

    return app
```

That's it – WhiteNoise will now serve your static files.

3. Custom *static* folder

If it turns out that your are not using the Flask default for *static* folder, fear not. You can instantiate WhiteNoise and add your *static* folders later:

```
from flask import Flask
from whitenoise import WhiteNoise

app = WhiteNoise(Flask(__name__))
```

```
my_static_folders = (
    'static/folder/one/',
    'static/folder/two/',
    'static/folder/three/'
)
for static in my_static_folders:
    app.add_files(static)
```

And check `WhiteNoise.add_file` documentation for further customization.

Change Log

v4.0b2

Note:

Breaking changes The latest version of WhiteNoise removes some options which were deprecated in the previous major release:

- The WSGI integration option for Django (which involved editing `wsgi.py`) has been removed. Instead, you should add WhiteNoise to your middleware list in `settings.py` and remove any reference to WhiteNoise from `wsgi.py`. See the *documentation* for more details. (The *pure WSGI* integration is still available for non-Django apps.)
- The `whitenoise.django.GzipManifestStaticFilesStorage` alias has now been removed. Instead you should use the correct import path: `whitenoise.storage.CompressedManifestStaticFilesStorage`.

If you are not using either of these integration options you should have no issues upgrading to the latest version.

Index file support

WhiteNoise now supports serving *index files* for directories (e.g. serving `/example/index.html` at `/example/`). It also creates redirects so that visiting the index file directly, or visiting the URL without a trailing slash will redirect to the correct URL.

Range header support ("byte serving")

WhiteNoise now respects the HTTP Range header which allows a client to request only part of a file. The main use for this is in serving video files to iOS devices as Safari refuses to play videos unless the server supports the Range header.

ETag support

WhiteNoise now adds ETag headers to files using the same algorithm used by nginx. This gives slightly better caching behviour than relying purely on Last Modified dates (although not as good as creating immutable files using something like `ManifestStaticFilesStorage`, which is still the best option if you can use it).

If you need to generate your own ETags headers for any reason you can define a custom *`add_headers_function`*.

Customisable immutable files test

WhiteNoise ships with code which detects when you are using Django's ManifestStaticFilesStorage backend and sends optimal caching headers for files which are guaranteed not to change. If you are using a different system for generating cacheable files then you might need to supply your own function for detecting such files. Previously this required subclassing WhiteNoise, but now you can use the *WHITENOISE_IMMUTABLE_FILE_TEST* setting.

Fix runserver_nostatic to work with Channels

The old implementation of *runserver_nostatic* (which disables Django's default static file handling in development) did not work with Channels, which needs its own runserver implementation. The runserver_nostatic command has now been rewritten so that it should work with Channels and with any other app which provides its own runserver.

Reduced storage requirements for static files

The new *WHITENOISE_KEEP_ONLY_HASHED_FILES* setting reduces the number of files in STATIC_ROOT by half by storing files only under their hashed names (e.g. `app.db8f2edc0c8a.js`), rather than also keeping a copy with the original name (e.g. `app.js`).

Improved start up performance

When in production mode (i.e. when *autorefresh* is disabled), WhiteNoise scans all static files when the application starts in order to be able to serve them as efficiently and securely as possible. For most applications this makes no noticeable difference to start up time, however for applications with very large numbers of static files this process can take some time. In WhiteNoise 4.0 the file scanning code has been rewritten to do the minimum possible amount of filesystem access which should make the start up process considerably faster.

Windows Testing

WhiteNoise has always aimed to support Windows as well as *NIX platforms but we are now able to run the test suite against Windows as part of the CI process which should ensure that we can maintain Windows compatibility in future.

Modification times for compressed files

The compressed storage backend (which generates Gzip and Brotli compressed files) now ensures that compressed files have the same modification time as the originals. This only makes a difference if you are using the compression backend with something other than WhiteNoise to actually serve the files, which very few users do.

---

v3.3.0

- Support the new immutable Cache-Control header. This gives better caching behaviour for immutable resources than simply setting a large max age.

v3.2.3

- Gracefully handle invalid byte sequences in URLs.
- Gracefully handle filenames which are too long for the filesystem.
- Send correct Content-Type for Adobe's `crossdomain.xml` files.

v3.2.2

- Convert any config values supplied as byte strings to text to avoid runtime encoding errors when encountering non-ASCII filenames.

v3.2.1

- Handle non-ASCII URLs correctly when using the `wsgi.py` integration.
- Fix exception triggered when a static files "finder" returned a directory rather than a file.

v3.2

- Add support for the new-style middleware classes introduced in Django 1.10. The same WhiteNoiseMiddleware class can now be used in either the old `MIDDLEWARE_CLASSES` list or the new `MIDDLEWARE` list.
- Fixed a bug where incorrect Content-Type headers were being sent on 304 Not Modified responses (thanks @oppianmatt).
- Return Vary and Cache-Control headers on 304 responses, as specified by the RFC.

v3.1

- Add new *`WHITENOISE_STATIC_PREFIX`* setting to give flexibility in supporting non-standard deployment configurations e.g. serving the application somewhere other than the domain root.
- Fix bytes/unicode bug when running with Django 1.10 on Python 2.7

v3.0

Note: The latest version of WhiteNoise contains some small **breaking changes**. Most users will be able to upgrade without any problems, but some less-used APIs have been modified:

- The setting `WHITENOISE_GZIP_EXCLUDE_EXTENSIONS` has been renamed to `WHITENOISE_SKIP_COMPRESS_EXTENSIONS`.
- The CLI *compression utility* has moved from `python -m whitenoise.gzip` to `python -m whitenoise.compress`.
- The now redundant `gzipstatic` management command has been removed.
- WhiteNoise no longer uses the system mimetypes files, so if you are serving particularly obscure filetypes you may need to add their mimetypes explicitly using the new *`mimetypes`* setting.
- Older versions of Django (1.4-1.7) and Python (2.6) are no longer supported. If you need support for these platforms you can continue to use WhiteNoise 2.x.

- The `whitenoise.django.GzipManifestStaticFilesStorage` storage backend has been moved to `whitenoise.storage.CompressedManifestStaticFilesStorage`. The old import path **will continue to work** for now, but users are encouraged to update their code to use the new path.

Simpler, cleaner Django middleware integration

WhiteNoise can now integrate with Django by adding a single line to `MIDDLEWARE_CLASSES` without any need to edit `wsgi.py`. This also means that WhiteNoise plays nicely with other middleware classes such as *SecurityMiddleware*, and that it is fully compatible with the new Channels system. See the *updated documentation* for details.

Brotli compression support

Brotli is the modern, more efficient alternative to gzip for HTTP compression. To benefit from smaller files and faster page loads, just install the brotlipy library, update your `requirements.txt` and WhiteNoise will take care of the rest. See the *documentation* for details.

Simpler customisation

It's now possible to add custom headers to WhiteNoise without needing to create a subclass, using the new *`add_headers_function`* setting.

Use WhiteNoise in development with Django

There's now an option to force Django to use WhiteNoise in development, rather than its own static file handling. This results in more consistent behaviour between development and production environments and fewer opportunities for bugs and surprises. See the *documentation* for details.

Improved mimetype handling

WhiteNoise now ships with its own mimetype definitions (based on those shipped with nginx) instead of relying on the system ones, which can vary between environments. There is a new *`mimetypes`* configuration option which makes it easy to add additional type definitions if needed.

Thanks

A big thank-you to Ed Morley and Tim Graham for their contributions to this release.

---

v2.0.6

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

v2.0.5

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

v2.0.4

- Report path on MissingFileError (thanks @ezheidtmann).

v2.0.3

- Add *__version__* attribute.

v2.0.2

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

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.

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.

v1.0.6

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

v1.0.5

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

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.

v1.0.3

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

v1.0.2

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

v1.0.1

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

v1.0

- First stable release.

Index