
Django Compressor Documentation

Release 1.4a1

Django Compressor authors

Aug 08, 2018

Contents

1	Why another static file combiner for Django?	3
2	Contents	5
2.1	Quickstart	5
2.2	Usage	6
2.3	Common Deployment Scenarios	9
2.4	Settings	10
2.5	Remote storages	16
2.6	Behind the scenes	18
2.7	Jinja2 In-Request Support	18
2.8	Jinja2 Offline Compression Support	19
2.9	django-sekizai Support	21
2.10	Contributing	22
2.11	Changelog	24

Compresses linked and inline JavaScript or CSS into a single cached file.

Why another static file combiner for Django?

Short version: None of them did exactly what I needed.

Long version:

JS/CSS belong in the templates Every static combiner for Django I've seen makes you configure your static files in your `settings.py`. While that works, it doesn't make sense. Static files are for display. And it's not even an option if your settings are in completely different repositories and use different deploy processes from the templates that depend on them.

Flexibility Django Compressor doesn't care if different pages use different combinations of statics. It doesn't care if you use inline scripts or styles. It doesn't get in the way.

Automatic regeneration and cache-foreverable generated output Statics are never stale and browsers can be told to cache the output forever.

Full test suite I has one.

2.1 Quickstart

2.1.1 Installation

- Install Django Compressor with your favorite Python package manager:

```
pip install django_compressor
```

- Add 'compressor' to your INSTALLED_APPS setting:

```
INSTALLED_APPS = (  
    # other apps  
    "compressor",  
)
```

- See the list of *Settings* to modify Django Compressor's default behaviour and make adjustments for your website.
- In case you use Django's *staticfiles* contrib app (or its standalone counterpart *django-staticfiles*) you have to add Django Compressor's file finder to the `STATICFILES_FINDERS` setting, for example with `django.contrib.staticfiles`:

```
STATICFILES_FINDERS = (  
    'django.contrib.staticfiles.finders.FileSystemFinder',  
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',  
    # other finders..  
    'compressor.finders.CompressorFinder',  
)
```

- Define `COMPRESS_ROOT` in settings if you don't have already `STATIC_ROOT` or if you want it in a different folder.

2.1.2 Dependencies

Required

In case you're installing Django Compressor differently (e.g. from the Git repo), make sure to install the following dependencies.

- `django-appconf`

Used internally to handle Django's settings, this is automatically installed when following the above installation instructions.

```
pip install django-appconf
```

Optional

- `BeautifulSoup`

For the `parser` `compressor.parser.BeautifulSoupParser` and `compressor.parser.LxmlParser`:

```
pip install "BeautifulSoup<4.0"
```

- `lxml`

For the `parser` `compressor.parser.LxmlParser`, also requires `libxml2`:

```
STATIC_DEPS=true pip install lxml
```

- `html5lib`

For the `parser` `compressor.parser.Html5LibParser`:

```
pip install html5lib
```

- `Slim It`

For the `Slim It filter` `compressor.filters.jsmin.SlimItFilter`:

```
pip install slimit
```

2.2 Usage

```
{% load compress %}
{% compress <js/css> [<file/inline> [block_name]] %}
<html of inline or linked JS/CSS>
{% endcompress %}
```

2.2.1 Examples

```
{% load compress %}

{% compress css %}
```

(continues on next page)

(continued from previous page)

```
<link rel="stylesheet" href="/static/css/one.css" type="text/css" charset="utf-8">
<style type="text/css">p { border:5px solid green;}</style>
<link rel="stylesheet" href="/static/css/two.css" type="text/css" charset="utf-8">
{% endcompress %}
```

Which would be rendered something like:

```
<link rel="stylesheet" href="/static/CACHE/css/f7c661b7a124.css" type="text/css"
↳ charset="utf-8">
```

or:

```
{% load compress %}

{% compress js %}
<script src="/static/js/one.js" type="text/javascript" charset="utf-8"></script>
<script type="text/javascript" charset="utf-8">obj.value = "value";</script>
{% endcompress %}
```

Which would be rendered something like:

```
<script type="text/javascript" src="/static/CACHE/js/3f33b9146e12.js" charset="utf-8">
↳ </script>
```

Note: Remember that django-compressor will try to *group outputs by media*.

Linked files **must** be accessible via `COMPRESS_URL`.

If the `COMPRESS_ENABLED` setting is `False` (defaults to the opposite of `DEBUG`) the `compress` template tag does nothing and simply returns exactly what it was given.

Note: If you've configured any *precompilers* setting `COMPRESS_ENABLED` to `False` won't affect the processing of those files. Only the *CSS* and *JavaScript filters* will be disabled.

If both `DEBUG` and `COMPRESS_ENABLED` are set to `True`, incompressible files (off-site or non-existent) will throw an exception. If `DEBUG` is `False` these files will be silently stripped.

Warning: For production sites it is **strongly recommended** to use a real cache backend such as `memcached` to speed up the checks of compressed files. Make sure you set your Django cache backend appropriately (also see `COMPRESS_CACHE_BACKEND` and Django's caching documentation).

The `compress` template tag supports a second argument specifying the output mode and defaults to saving the result in a file. Alternatively you can pass `'inline'` to the template tag to return the content directly to the rendered page, e.g.:

```
{% load compress %}

{% compress js inline %}
<script src="/static/js/one.js" type="text/javascript" charset="utf-8"></script>
<script type="text/javascript" charset="utf-8">obj.value = "value";</script>
{% endcompress %}
```

would be rendered something like:

```
<script type="text/javascript" charset="utf-8">
obj = {};
obj.value = "value";
</script>
```

The `compress` template tag also supports a third argument for naming the output of that particular `compress` tag. This is then added to the context so you can access it in the `post_compress signal <signals>`.

2.2.2 Pre-compression

Django Compressor comes with an optional `compress` management command to run the compression outside of the request/response loop – independent from user requests. This allows to pre-compress CSS and JavaScript files and works just like the automatic compression with the `{% compress %}` tag.

To compress the files “offline” and update the offline cache you have to use the `compress` management command, ideally during deployment. Also make sure to enable the `django.conf.settings.COMPRESS_OFFLINE` setting. In case you don’t use the `compress` management command, Django Compressor will automatically fallback to the automatic compression using the template tag.

The command parses all templates that can be found with the template loader (as specified in the `TEMPLATE_LOADERS` setting) and looks for `{% compress %}` blocks. It then will use the context as defined in `django.conf.settings.COMPRESS_OFFLINE_CONTEXT` to render its content. So if you use any variables inside the `{% compress %}` blocks, make sure to list all values you require in `COMPRESS_OFFLINE_CONTEXT`. It’s similar to a template context and should be used if a variable is used in the blocks, e.g.:

```
{% load compress %}
{% compress js %}
<script src="{ path_to_files }}"js/one.js" type="text/javascript" charset="utf-8"></
→script>
{% endcompress %}
```

Since this template requires a variable (`path_to_files`) you need to specify this in your settings before using the `compress` management command:

```
COMPRESS_OFFLINE_CONTEXT = {
    'path_to_files': '/static/js/',
}
```

If not specified, the `COMPRESS_OFFLINE_CONTEXT` will by default contain the commonly used setting to refer to saved files `STATIC_URL`.

The result of running the `compress` management command will be cached in a file called `manifest.json` using the `configured storage` to be able to be transferred from your development computer to the server easily.

2.2.3 Signals

`compressor.signals.post_compress` (*sender, type, mode, context*)

Django Compressor includes a `post_compress` signal that enables you to listen for changes to your compressed CSS/JS. This is useful, for example, if you need the exact filenames for use in an HTML5 manifest file. The signal sends the following arguments:

sender Either `compressor.css.CssCompressor` or `compressor.js.JsCompressor`.

Changed in version 1.2.

The sender is now one of the supported Compressor classes for easier limitation to only one of them, previously it was a string named `'django-compressor'`.

type Either `"js"` or `"css"`.

mode Either `"file"` or `"inline"`.

context The context dictionary used to render the output of the compress template tag.

If `mode` is `"file"` the dictionary named `compressed` in the context will contain a `"url"` key that maps to the relative URL for the compressed asset.

If `type` is `"css"`, the dictionary named `compressed` in the context will additionally contain a `"media"` key with a value of `None` if no media attribute is specified on the link/style tag and equal to that attribute if one is specified.

Additionally, `context['compressed']['name']` will be the third positional argument to the template tag, if provided.

Note: When compressing CSS, the `post_compress` signal will be called once for every different media attribute on the tags within the `{% compress %}` tag in question.

2.2.4 CSS Notes

All relative `url()` bits specified in linked CSS files are automatically converted to absolute URLs while being processed. Any local absolute URLs (those starting with a `'/'`) are left alone.

Stylesheets that are `@import`'d are not compressed into the main file. They are left alone.

If the media attribute is set on `<style>` and `<link>` elements, a separate compressed file is created and linked for each media value you specified. This allows the media attribute to remain on the generated link element, instead of wrapping your CSS with `@media` blocks (which can break your own `@media` queries or `@font-face` declarations). It also allows browsers to avoid downloading CSS for irrelevant media types.

2.2.5 Recommendations

- Use only relative or full domain absolute URLs in your CSS files.
- Avoid `@import`! Simply list all your CSS files in the HTML, they'll be combined anyway.

2.3 Common Deployment Scenarios

This document presents the most typical scenarios in which Django Compressor can be configured, and should help you decide which method you may want to use for your stack.

2.3.1 In-Request Compression

This is the default method of compression. Where-in Django Compressor will go through the steps outlined in *Behind the scenes*. You will find in-request compression beneficial if:

- Using a single server setup, where the application and static files are on the same machine.
- Prefer a simple configuration. By default, there is no configuration required.

2.3.2 Caveats

- If deploying to a multi-server setup and using `COMPRESS_PRECOMPILERS`, each binary is required to be installed on each application server.
- Application servers may not have permissions to write to your static directories. For example, if deploying to a CDN (e.g. Amazon S3)

2.3.3 Offline Compression

This method decouples the compression outside of the request (see *Behind the scenes*) and can prove beneficial in the speed, and in many scenarios, the maintainability of your deployment. You will find offline compression beneficial if:

- Using a multi-server setup. A common scenario for this may be multiple application servers and a single static file server (CDN included). With offline compression, you typically run `manage.py compress` on a single utility server, meaning you only maintain `COMPRESS_PRECOMPILERS` binaries in one location.
- You store compressed files on a CDN.

2.3.4 Caveats

- If your templates have complex logic in how template inheritance is done (e.g. `{% extends context_variable %}`), then this becomes a problem, as offline compression will not have the context, unless you set it in `COMPRESS_OFFLINE_CONTEXT`
- Due to the way the manifest file is used, while deploying across a multi-server setup, your application may use old templates with a new manifest, possibly rendering your pages incoherent. The current suggested solution for this is to change the `COMPRESS_OFFLINE_MANIFEST` path for each new version of your code. This will ensure that the old code uses old compressed output, and the new one appropriately as well.

Every setup is unique, and your scenario may differ slightly. Choose what is the most sane to maintain for your situation.

2.4 Settings

Django Compressor has a number of settings that control its behavior. They've been given sensible defaults.

2.4.1 Base settings

`django.conf.settings.COMPRESS_ENABLED`

Default the opposite of `DEBUG`

Boolean that decides if compression will happen. To test compression when `DEBUG` is `True` `COMPRESS_ENABLED` must also be set to `True`.

When `COMPRESS_ENABLED` is `False` the input will be rendered without any compression except for code with a `mimetype` matching one listed in the `COMPRESS_PRECOMPILERS` setting. These matching files are still passed to the precompiler before rendering.

An example for some javascript and coffeescript.

```
{% load compress %}

{% compress js %}
<script type="text/javascript" src="/static/js/site-base.js" />
<script type="text/coffeescript" charset="utf-8" src="/static/js/awesome.coffee" /
↪>
{% endcompress %}
```

With `COMPRESS_ENABLED` set to `False` this would give you something like this:

```
<script type="text/javascript" src="/static/js/site-base.js"></script>
<script type="text/javascript" src="/static/CACHE/js/8dd1a2872443.js"
↪ charset="utf-8"></script>
```

`django.conf.settings.COMPRESS_URL`

Default `STATIC_URL`

Controls the URL that linked files will be read from and compressed files will be written to.

`django.conf.settings.COMPRESS_ROOT`

Default `STATIC_ROOT`

Controls the absolute file path that linked static will be read from and compressed static will be written to when using the default `COMPRESS_STORAGE` compressor, `storage.CompressorFileStorage`.

`django.conf.settings.COMPRESS_OUTPUT_DIR`

Default `'CACHE'`

Controls the directory inside `COMPRESS_ROOT` that compressed files will be written to.

2.4.2 Backend settings

`django.conf.settings.COMPRESS_CSS_FILTERS`

Default `['compressor.filters.css_default.CssAbsoluteFilter']`

A list of filters that will be applied to CSS.

Possible options are (including their settings):

- `compressor.filters.css_default.CssAbsoluteFilter`

A filter that normalizes the URLs used in `url()` CSS statements.

`django.conf.settings.COMPRESS_CSS_HASHING_METHOD`

The method to use when calculating the hash to append to processed URLs. Either `'mtime'` (default) or `'content'`. Use the latter in case you're using multiple server to serve your static files.

- `compressor.filters.csstidy.CSS tidyFilter`

A filter that passes the CSS content to the `CSSTidy` tool.

`django.conf.settings.COMPRESS_CSSTIDY_BINARY`

The `CSSTidy` binary filesystem path.

`django.conf.settings.COMPRESS_CSSTIDY_ARGUMENTS`

The arguments passed to `CSSTidy`.

- `compressor.filters.datauri.CssDataUriFilter`

A filter for embedding media as `data: URIs` in the CSS.

`django.conf.settings.COMPRESS_DATA_URI_MAX_SIZE`
Only files that are smaller than this in bytes value will be embedded.

- `compressor.filters.yui.YUICSSFilter`

A filter that passes the CSS content to the [YUI compressor](#).

`django.conf.settings.COMPRESS_YUI_BINARY`
The YUI compressor filesystem path. Make sure to also prepend this setting with `java -jar` if you use that kind of distribution.

`django.conf.settings.COMPRESS_YUI_CSS_ARGUMENTS`
The arguments passed to the compressor.

- `compressor.filters.yuglify.YUglifyCSSFilter`

A filter that passes the CSS content to the [yUglify compressor](#).

`django.conf.settings.COMPRESS_YUGLIFY_BINARY`
The yUglify compressor filesystem path.

`django.conf.settings.COMPRESS_YUGLIFY_CSS_ARGUMENTS`
The arguments passed to the compressor. Defaults to `-terminal`.

- `compressor.filters.cssmin.CSSMinFilter`

A filter that uses Zachary Voase's Python port of the YUI CSS compression algorithm [cssmin](#).

- `compressor.filters.template.TemplateFilter`

A filter that renders the CSS content with Django templating system.

`django.conf.settings.COMPRESS_TEMPLATE_FILTER_CONTEXT`
The context to render your css files with.

`django.conf.settings.COMPRESS_JS_FILTERS`

Default `['compressor.filters.jsmin.JSMinFilter']`

A list of filters that will be applied to javascript.

Possible options are:

- `compressor.filters.jsmin.JSMinFilter`

A filter that uses the jsmin implementation [rJSmin](#) to compress JavaScript code.

- `compressor.filters.jsmin.SlimItFilter`

A filter that uses the jsmin implementation [Slim It](#) to compress JavaScript code.

- `compressor.filters.closure.ClosureCompilerFilter`

A filter that uses [Google Closure compiler](#).

`django.conf.settings.COMPRESS_CLOSURE_COMPILER_BINARY`
The Closure compiler filesystem path. Make sure to also prepend this setting with `java -jar` if you use that kind of distribution.

`django.conf.settings.COMPRESS_CLOSURE_COMPILER_ARGUMENTS`
The arguments passed to the compiler.

- `compressor.filters.yui.YUIJSFilter`

A filter that passes the JavaScript code to the [YUI compressor](#).

`django.conf.settings.COMPRESS_YUI_BINARY`

The YUI compressor filesystem path.

`django.conf.settings.COMPRESS_YUI_JS_ARGUMENTS`

The arguments passed to the compressor.

- `compressor.filters.yuglify.YUglifyJSFilter`

A filter that passes the JavaScript code to the [yUglify compressor](#).

`django.conf.settings.COMPRESS_YUGLIFY_BINARY`

The yUglify compressor filesystem path.

`django.conf.settings.COMPRESS_YUGLIFY_JS_ARGUMENTS`

The arguments passed to the compressor.

- `compressor.filters.template.TemplateFilter`

A filter that renders the JavaScript code with Django templating system.

`django.conf.settings.COMPRESS_TEMPLATE_FILTER_CONTEXT`

The context to render your JavaScript code with.

`django.conf.settings.COMPRESS_PRECOMPILERS`

Default ()

An iterable of two-tuples whose first item is the mimetype of the files or hunks you want to compile with the command or filter specified as the second item:

1. **mimetype** The mimetype of the file or inline code that should be compiled.
2. **command_or_filter** The command to call on each of the files. Modern Python string formatting will be provided for the two placeholders `{infile}` and `{outfile}` whose existence in the command string also triggers the actual creation of those temporary files. If not given in the command string, Django Compressor will use `stdin` and `stdout` respectively instead.

Alternatively, you may provide the fully qualified class name of a filter you wish to use as a precompiler.

Example:

```
COMPRESS_PRECOMPILERS = (
    ('text/coffeescript', 'coffee --compile --stdio'),
    ('text/less', 'lessc {infile} {outfile}'),
    ('text/x-sass', 'sass {infile} {outfile}'),
    ('text/x-scss', 'sass --scss {infile} {outfile}'),
    ('text/stylus', 'stylus < {infile} > {outfile}'),
    ('text/foobar', 'path.to.MyPrecompilerFilter'),
)
```

Note: Depending on the implementation, some precompilers might not support outputting to something else than `stdout`, so you'll need to omit the `{outfile}` parameter when working with those. For instance, if you are using the Ruby version of `lessc`, you'll need to set up the precompiler like this:

```
('text/less', 'lessc {infile}'),
```

With that setting (and [CoffeeScript](#) installed), you could add the following code to your templates:

```
{% load compress %}

{% compress js %}
<script type="text/coffeescript" charset="utf-8" src="/static/js/awesome.coffee" /
↪>
<script type="text/coffeescript" charset="utf-8">
# Functions:
square = (x) -> x * x
</script>
{% endcompress %}
```

This would give you something like this:

```
<script type="text/javascript" src="/static/CACHE/js/8dd1a2872443.js"
↪charset="utf-8"></script>
```

The same works for `less`, too:

```
{% load compress %}

{% compress css %}
<link type="text/less" rel="stylesheet" href="/static/css/styles.less" charset=
↪"utf-8">
<style type="text/less">
@color: #4D926F;

#header {
  color: @color;
}
</style>
{% endcompress %}
```

Which would be rendered something like:

```
<link rel="stylesheet" href="/static/CACHE/css/8ccf8d877f18.css" type="text/css"
↪charset="utf-8">
```

`django.conf.settings.COMPRESS_STORAGE`

Default `'compressor.storage.CompressorFileStorage'`

The dotted path to a Django Storage backend to be used to save the compressed files.

Django Compressor ships with one additional storage backend:

- `'compressor.storage.GzipCompressorFileStorage'`

A subclass of the default storage backend, which will additionally create `*.gz` files of each of the compressed files.

`django.conf.settings.COMPRESS_PARSER`

Default `'compressor.parser.AutoSelectParser'`

The backend to use when parsing the JavaScript or Stylesheet files. The `AutoSelectParser` picks the `lxml` based parser when available, and falls back to `HtmlParser` if `lxml` is not available.

`LxmlParser` is the fastest available parser, but `HtmlParser` is not much slower. `AutoSelectParser` adds a slight overhead, but in most cases it won't be necessary to change the default parser.

The other two included parsers are considerably slower and should only be used if absolutely necessary.

Warning: In some cases the `compressor.parser.HtmlParser` parser isn't able to parse invalid HTML in JavaScript or CSS content. As a workaround you should use one of the more forgiving parsers, e.g. the `BeautifulSoupParser`.

The backends included in Django Compressor:

- `compressor.parser.AutoSelectParser`
- `compressor.parser.LxmlParser`
- `compressor.parser.HtmlParser`
- `compressor.parser.BeautifulSoupParser`
- `compressor.parser.Html5LibParser`

See *Dependencies* for more info about the packages you need for each parser.

2.4.3 Caching settings

`django.conf.settings.COMPRESS_CACHE_BACKEND`

Default `CACHES["default"]` or `CACHE_BACKEND`

The backend to use for caching, in case you want to use a different cache backend for Django Compressor.

If you have set the `CACHES` setting (new in Django 1.3), `COMPRESS_CACHE_BACKEND` defaults to "default", which is the alias for the default cache backend. You can set it to a different alias that you have configured in your `CACHES` setting.

If you have not set `CACHES` and are using the old `CACHE_BACKEND` setting, `COMPRESS_CACHE_BACKEND` defaults to the `CACHE_BACKEND` setting.

`django.conf.settings.COMPRESS_REBUILD_TIMEOUT`

Default 2592000 (30 days in seconds)

The period of time after which the compressed files are rebuilt even if no file changes are detected.

`django.conf.settings.COMPRESS_MINT_DELAY`

Default 30 (seconds)

The upper bound on how long any compression should take to run. Prevents dog piling, should be a lot smaller than `COMPRESS_REBUILD_TIMEOUT`.

`django.conf.settings.COMPRESS_MTIME_DELAY`

Default 10

The amount of time (in seconds) to cache the modification timestamp of a file. Should be smaller than `COMPRESS_REBUILD_TIMEOUT` and `COMPRESS_MINT_DELAY`.

`django.conf.settings.COMPRESS_DEBUG_TOGGLE`

Default None

The name of the GET variable that toggles the debug mode and prevents Django Compressor from performing the actual compression. Only useful for debugging.

Warning: Don't use this option in production!

An easy convention is to only set it depending on the `DEBUG` setting:

```
if DEBUG:
    COMPRESS_DEBUG_TOGGLE = 'whatever'
```

Note: This only works for pages that are rendered using the `RequestContext` and the `django.core.context_processors.request` context processor.

`django.conf.settings.COMPRESS_CACHE_KEY_FUNCTION`

Default `'compressor.cache.simple_cachekey'`

The function to use when generating the cache key. The function must take one argument which is the partial key based on the source's hex digest. It must return the full key as a string.

2.4.4 Offline settings

`django.conf.settings.COMPRESS_OFFLINE`

Default `False`

Boolean that decides if compression should also be done outside of the request/response loop – independent from user requests. This allows to pre-compress CSS and JavaScript files and works just like the automatic compression with the `{% compress %}` tag.

`django.conf.settings.COMPRESS_OFFLINE_TIMEOUT`

Default `31536000` (1 year in seconds)

The period of time with which the `compress` management command stores the pre-compressed the contents of `{% compress %}` template tags in the cache.

`django.conf.settings.COMPRESS_OFFLINE_CONTEXT`

Default `{'STATIC_URL': settings.STATIC_URL}`

The context to be used by the `compress` management command when rendering the contents of `{% compress %}` template tags and saving the result in the offline cache.

If available, the `STATIC_URL` setting is also added to the context.

`django.conf.settings.COMPRESS_OFFLINE_MANIFEST`

Default `manifest.json`

The name of the file to be used for saving the names of the files compressed offline.

2.5 Remote storages

In some cases it's useful to use a [CDN](#) for serving static files such as those generated by Django Compressor. Due to the way Django Compressor processes files, it requires the files to be processed (in the `{% compress %}` block) to be available in a local file system cache.

Django Compressor provides hooks to automatically have compressed files pushed to a remote storage backend. Simply set the storage backend that saves the result to a remote service (see [COMPRESS_STORAGE](#)).

2.5.1 django-storages

So assuming your CDN is [Amazon S3](#), you can use the `boto` storage backend from the 3rd party app `django-storages`. Some required settings are:

```
AWS_ACCESS_KEY_ID = 'XXXXXXXXXXXXXXXXXXXXX'
AWS_SECRET_ACCESS_KEY = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
AWS_STORAGE_BUCKET_NAME = 'compressor-test'
```

Next, you need to specify the new CDN base URL and update the URLs to the files in your templates which you want to compress:

```
COMPRESS_URL = "http://compressor-test.s3.amazonaws.com/"
```

Note: For staticfiles just set `STATIC_URL = COMPRESS_URL`

The storage backend to save the compressed files needs to be changed, too:

```
COMPRESS_STORAGE = 'storages.backends.s3boto.S3BotoStorage'
```

2.5.2 Using staticfiles

If you are using Django's `staticfiles` contrib app or the standalone app `django-staticfiles`, you'll need to use a temporary filesystem cache for Django Compressor to know which files to compress. Since `staticfiles` provides a management command to collect static files from various locations which uses a storage backend, this is where both apps can be integrated.

1. Make sure the `COMPRESS_ROOT` and `STATIC_ROOT` settings are equal since both apps need to look at the same directories when to do their job.
2. You need to create a subclass of the remote storage backend you want to use; below is an example of the `boto` S3 storage backend from `django-storages`:

```
from django.core.files.storage import get_storage_class
from storages.backends.s3boto import S3BotoStorage

class CachedS3BotoStorage(S3BotoStorage):
    """
    S3 storage backend that saves the files locally, too.
    """
    def __init__(self, *args, **kwargs):
        super(CachedS3BotoStorage, self).__init__(*args, **kwargs)
        self.local_storage = get_storage_class(
            "compressor.storage.CompressorFileStorage" )()

    def save(self, name, content):
        name = super(CachedS3BotoStorage, self).save(name, content)
        self.local_storage._save(name, content)
        return name
```

3. Set your `COMPRESS_STORAGE` and `STATICFILES_STORAGE` settings to the dotted path of your custom cached storage backend, e.g. `'mysite.storage.CachedS3BotoStorage'`.
4. To have Django correctly render the URLs to your static files, set the `STATIC_URL` setting to the same value as `COMPRESS_URL` (e.g. `"http://compressor-test.s3.amazonaws.com/"`).

2.6 Behind the scenes

This document assumes you already have an up and running instance of Django Compressor, and that you understand how to use it in your templates. The goal is to explain what the main template tag, `{% compress %}`, does behind the scenes, to help you debug performance problems for instance.

2.6.1 Offline cache

If offline cache is activated, the first thing `{% compress %}` tries to do is retrieve the compressed version for its nodelist from the offline manifest cache. It doesn't parse, doesn't check the modified times of the files, doesn't even know which files are concerned actually, since it doesn't look inside the nodelist of the template block enclosed by the `compress` template tag. The offline cache manifest is just a json file, stored on disk inside the directory that holds the compressed files. The format of the manifest is simply a key => value dictionary, with the hash of the nodelist being the key, and the HTML containing the element code for the combined file or piece of code being the value. Generating the offline manifest, using the `compress` management command, also generates the combined files referenced in the manifest.

If offline cache is activated and the nodelist hash can not be found inside the manifest, `{% compress %}` will raise an `OfflineGenerationError`.

If offline cache is de-activated, the following happens:

2.6.2 First step: parsing and file list

A compressor instance is created, which in turns instantiates the HTML parser. The parser is used to determine a file or code hunk list. Each file mtime is checked, first in cache and then on disk/storage, and this is used to determine an unique cache key.

2.6.3 Second step: Checking the “main” cache

Compressor checks if it can get some info about the combined file/hunks corresponding to its instance, using the cache key obtained in the previous step. The cache content here will actually be the HTML containing the final element code, just like in the offline step before.

Everything stops here if the cache entry exists.

2.6.4 Third step: Generating the combined file if needed

The file is generated if necessary. All precompilers are called and all filters are executed, and a hash is determined from the contents. This in turns helps determine the file name, which is only saved if it didn't exist already. Then the HTML output is returned (and also saved in the cache). And that's it!

2.7 Jinja2 In-Request Support

Django Compressor comes with support for [Jinja2](#) via an extension.

2.7.1 Plain Jinja2

In order to use Django Compressor's Jinja2 extension we would need to pass `compressor.contrib.jinja2ext.CompressorExtension` into environment:

```
import jinja2
from compressor.contrib.jinja2ext import CompressorExtension

env = jinja2.Environment(extensions=[CompressorExtension])
```

From now on, you can use same code you'd normally use within Django templates:

```
from django.conf import settings
template = env.from_string('\n'.join([
    '{% compress css %}',
    '<link rel="stylesheet" href="{% STATIC_URL %}css/one.css" type="text/css" charset="utf-8">',
    '{% endcompress %}',
]))
template.render({'STATIC_URL': settings.STATIC_URL})
```

2.7.2 For coffin users

`Coffin` makes it very easy to include additional Jinja2 extensions as it only requires to add extension to `JINJA2_EXTENSIONS` at main settings module:

```
JINJA2_EXTENSIONS = [
    'compressor.contrib.jinja2ext.CompressorExtension',
]
```

And that's it - our extension is loaded and ready to be used.

2.8 Jinja2 Offline Compression Support

You'd need to configure `COMPRESS_JINJA2_GET_ENVIRONMENT` so that Compressor can retrieve the Jinja2 environment for rendering. This can be a lambda or function that returns a Jinja2 environment.

2.8.1 Usage

Run the following compress command along with an `-engine` parameter. The parameter can be either `jinja2` or `django` (default). For example, `./manage.py compress -engine jinja2`.

2.8.2 Using both Django and Jinja2 templates

There may be a chance that the Jinja2 parser is used to parse Django templates if you have a mixture of Django and Jinja2 templates in the same location(s). This should not be a problem since the Jinja2 parser will likely raise a template syntax error, causing Compressor to skip the erroneous template safely. (Vice versa for Django parser).

A typical usage could be :

- `./manage.py compress` for processing Django templates first, skipping Jinja2 templates.

- “./manage.py compress -engine jinja2” for processing Jinja2 templates, skipping Django templates.

However, it is still recommended that you do not mix Django and Jinja2 templates in the same project.

2.8.3 Limitations

- Does not support `{% import %}` and similar blocks within `{% compress %}` blocks.
- Does not support `{{super()}}`.
- All other filters, globals and language constructs such as `{% if %}`, `{% with %}` and `{% for %}` are tested and should run fine.

2.8.4 Jinja2 templates location

IMPORTANT: For Compressor to discover the templates for offline compression, there must be a template loader that implements the `get_template_sources` method, and is in the `TEMPLATE_LOADERS` setting.

If you’re using Jinja2, you’re likely to have a Jinja2 template loader in the `TEMPLATE_LOADERS` setting, otherwise Django won’t know how to load Jinja2 templates. You could use [Jingo](#) or your own custom loader. [Coffin](#) works differently by providing a custom rendering method instead of a custom loader.

Unfortunately, [Jingo](#) does not implement such a method in its loader; [Coffin](#) does not seem to have a template loader in the first place. Read on to understand how to make Compressor work nicely with [Jingo](#) and [Coffin](#).

By default, if you don’t override the `TEMPLATE_LOADERS` setting, it will include the app directories loader that searches for templates under the `templates` directory in each app. If the app directories loader is in use and your Jinja2 templates are in the `<app>/templates` directories, Compressor will be able to find the Jinja2 templates.

However, if you have Jinja2 templates in other location(s), you could include the filesystem loader (`django.template.loaders.filesystem.Loader`) in the `TEMPLATE_LOADERS` setting and specify the custom location in the `TEMPLATE_DIRS` setting.

2.8.5 For Jingo users

You should configure `TEMPLATE_LOADERS` as such:

```
TEMPLATE_LOADERS = (
    'jingo.Loader',
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
)

def COMPRESS_JINJA2_GET_ENVIRONMENT():
    # TODO: ensure the CompressorExtension is installed with Jingo via
    # Jingo's JINJA_CONFIG setting.
    # Additional globals, filters, tests,
    # and extensions used within {%compress%} blocks must be configured
    # with Jingo.
    from jingo import env

    return env
```

This will enable the [Jingo](#) loader to load Jinja2 templates and the other loaders to report the templates location(s).

2.8.6 For Coffin users

You might want to configure `TEMPLATE_LOADERS` as such:

```
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
)

def COMPRESS_JINJA2_GET_ENVIRONMENT():
    # TODO: ensure the CompressorExtension is installed with Coffin
    # as described in the "In-Request Support" section above.
    # Additional globals, filters, tests,
    # and extensions used within {%compress%} blocks must be configured
    # with Coffin.
    from coffin.common import env

    return env
```

Again, if you have the Jinja2 templates in the app template directories, you're done here. Otherwise, specify the location in `TEMPLATE_DIRS`.

2.8.7 Using your custom loader

You should configure `TEMPLATE_LOADERS` as such:

```
TEMPLATE_LOADERS = (
    'your_app.Loader',
    ... other loaders (optional) ...
)
```

You could implement the `get_template_sources` method in your loader or make use of the Django's builtin loaders to report the Jinja2 template location(s).

2.8.8 Python 3 Support

Jingo with Jinja2 are tested and work on Python 2.6, 2.7, and 3.3. Coffin with Jinja2 are tested and work on Python 2.6 and 2.7 only. Jinja2 alone (with custom loader) are tested and work on Python 2.6, 2.7 and 3.3 only.

2.9 django-sekizai Support

Django Compressor comes with support for `_django-sekizai` via an extension. `_django-sekizai` provides the ability to include template code, from within any block, to a parent block. It is primarily used to include js/css from included templates to the master template.

It requires `_django-sekizai` to installed. Refer to the `_django-sekizai_docs` for how to use `render_block`

2.9.1 Usage

```
{% load sekizai_tags %}
{% render_block "<js/css>" postprocessor "compressor.contrib.sekizai.compress" %}
```

2.10 Contributing

Like every open-source project, Django Compressor is always looking for motivated individuals to contribute to it's source code. However, to ensure the highest code quality and keep the repository nice and tidy, everybody has to follow a few rules (nothing major, I promise :))

2.10.1 Community

People interested in developing for the Django Compressor should head over to #django-compressor on the [freenode](#) IRC network for help and to discuss the development.

You may also be interested in following [@jezdez](#) on Twitter.

2.10.2 In a nutshell

Here's what the contribution process looks like, in a bullet-points fashion, and only for the stuff we host on github:

1. Django Compressor is hosted on [github](#), at <https://github.com/django-compressor/django-compressor>
2. The best method to contribute back is to create a github account, then fork the project. You can use this fork as if it was your own project, and should push your changes to it.
3. When you feel your code is good enough for inclusion, "send us a [pull request](#)", by using the nice github web interface.

2.10.3 Contributing Code

Getting the source code

If you're interested in developing a new feature for Compressor, it is recommended that you first discuss it on IRC not to do any work that will not get merged in anyway.

- Code will be reviewed and tested by at least one core developer, preferably by several. Other community members are welcome to give feedback.
- Code *must* be tested. Your pull request should include unit-tests (that cover the piece of code you're submitting, obviously)
- Documentation should reflect your changes if relevant. There is nothing worse than invalid documentation.
- Usually, if unit tests are written, pass, and your change is relevant, then it'll be merged.

Since it's hosted on github, Django Compressor uses [git](#) as a version control system.

The [github help](#) is very well written and will get you started on using git and github in a jiffy. It is an invaluable resource for newbies and old timers alike.

Syntax and conventions

We try to conform to [PEP8](#) as much as possible. A few highlights:

- Indentation should be exactly 4 spaces. Not 2, not 6, not 8. **4**. Also, tabs are evil.
- We try (loosely) to keep the line length at 79 characters. Generally the rule is "it should look good in a terminal-base editor" (eg vim), but we try not be [Godwin's law] about it.

Process

This is how you fix a bug or add a feature:

1. [Fork us on github](#).
2. Checkout your fork.
3. Hack hack hack, test test test, commit commit commit, test again.
4. Push to your fork.
5. Open a pull request.

Tests

Having a wide and comprehensive library of unit-tests and integration tests is of exceeding importance. Contributing tests is widely regarded as a very prestigious contribution (you're making everybody's future work much easier by doing so). Good karma for you. Cookie points. Maybe even a beer if we meet in person :)

Generally tests should be:

- Unitary (as much as possible). I.E. should test as much as possible only one function/method/class. That's the very definition of unit tests.
- Integration tests are interesting too obviously, but require more time to maintain since they have a higher probability of breaking.
- Short running. No hard numbers here, but if your one test doubles the time it takes for everybody to run them, it's probably an indication that you're doing it wrong.

In a similar way to code, pull requests will be reviewed before pulling (obviously), and we encourage discussion via code review (everybody learns something this way) or IRC discussions.

Running the tests

To run the tests simply fork `django_compressor`, make the changes and open a pull request. The [Travis](#) bot will automatically run the tests of your branch/fork (see the [pull request announcement](#) for more info) and add a comment about the test results to the pull requests. Alternatively you can also login at Travis and enable your fork to run there, too. See the [Travis documentation](#) to read about how to do that.

Alternatively, create a virtualenv and activate it, then install the requirements **in the virtualenv**:

```
$ virtualenv compressor_test
$ source compressor_test/bin/activate
(compressor_test) $ make testenv
```

Then run `make test` to run the tests. Please note that this only tests `django_compressor` in the Python version you've created the virtualenv with not all the versions that are required to be supported.

2.10.4 Contributing Documentation

Perhaps considered “boring” by hard-core coders, documentation is sometimes even more important than code! This is what brings fresh blood to a project, and serves as a reference for old timers. On top of this, documentation is the one area where less technical people can help most - you just need to write a semi-decent English. People need to understand you.

Documentation should be:

- We use [Sphinx/restructuredText](#). So obviously this is the format you should use :) File extensions should be `.txt`.
- Written in English. We can discuss how it would bring more people to the project to have a Klingon translation or anything, but that's a problem we will ask ourselves when we already have a good documentation in English.
- Accessible. You should assume the reader to be moderately familiar with Python and Django, but not anything else. Link to documentation of libraries you use, for example, even if they are “obvious” to you. A brief description of what it does is also welcome.

Pulling of documentation is pretty fast and painless. Usually somebody goes over your text and merges it, since there are no “breaks” and that github parses rst files automagically it's really convenient to work with.

Also, contributing to the documentation will earn you great respect from the core developers. You get good karma just like a test contributor, but you get double cookie points. Seriously. You rock.

Note: This very document is based on the contributing docs of the [django CMS](#) project. Many thanks for allowing us to steal it!

2.11 Changelog

2.11.1 v1.4

- Added Python 3 compatibility.
- Added compatibility with Django 1.6.x.
- Fixed compatibility with `html5lib` 1.0.
- Added offline compression for Jinja2 with Jingo and Coffin integration.
- Improved support for template inheritance in offline compression.
- Made offline compression avoid compressing the same block multiple times.
- Added a `testenv` target in the Makefile to make it easier to set up the test environment.
- Allowed `data-uri` filter to handle external/protocol-relative references.
- Made `CssCompressor` class easier to extend.
- Added support for explicitly stating the block being ended.
- Added `rcssmin` and updated `rjsmin`.
- Removed implicit requirement on `BeautifulSoup`.
- Made `GzipCompressorFileStorage` set access and modified times to the same time as the corresponding base file.
- Defaulted to using django's `simplejson`, if present.
- Fixed `CompilerFilter` to always output Unicode strings.
- Fixed windows line endings in offline compression.

2.11.2 v1.3 (03/18/2013)

- *Backward incompatible changes*
 - Dropped support for Python 2.5. Removed any and walk compatibility functions in `compressor.utils`.
 - Removed compatibility with Django 1.2 for default values of some settings:
 - * `COMPRESS_ROOT` no longer uses `MEDIA_ROOT` if `STATIC_ROOT` is not defined. It expects `STATIC_ROOT` to be defined instead.
 - * `COMPRESS_URL` no longer uses `MEDIA_URL` if `STATIC_URL` is not defined. It expects `STATIC_URL` to be defined instead.
 - * `COMPRESS_CACHE_BACKEND` no longer uses `CACHE_BACKEND` and simply defaults to default.
- Added precompiler class support. This enables you to write custom precompilers with Python logic in them instead of just relying on executables.
- Made `CssAbsoluteFilter` smarter: it now handles URLs with hash fragments or querystring correctly. In addition, it now leaves alone fragment-only URLs.
- Removed a `fsync()` call in `CompilerFilter` to improve performance. We already called `self.infile.flush()` so that call was not necessary.
- Added an extension to provide `django-sekizai` support. See [django-sekizai Support](#) for more information.
- Fixed a `DeprecationWarning` regarding the use of `django.utils.hashcompat`
- Updated bundled `rjsmin.py` to fix some JavaScript compression errors.

2.11.3 v1.2

- Added compatibility with Django 1.4 and dropped support for Django 1.2.X.
- Added contributing docs. Be sure to check them out and start contributing!
- Moved CI to Travis: <http://travis-ci.org/django-compressor/django-compressor>
- Introduced a new `compressed` context dictionary that is passed to the templates that are responsible for rendering the compressed snippets.

This is a **backwards-incompatible change** if you've overridden any of the included templates:

- `compressor/css_file.html`
- `compressor/css_inline.html`
- `compressor/js_file.html`
- `compressor/js_inline.html`

The variables passed to those templates have been namespaced in a dictionary, so it's easy to fix your own templates.

For example, the old `compressor/js_file.html`:

```
<script type="text/javascript" src="{{ url }}"></script>
```

The new `compressor/js_file.html`:

```
<script type="text/javascript" src="{{ compressed.url }}"></script>
```

- Removed old templates named `compressor/css.html` and `compressor/js.html` that were originally left for backwards compatibility. If you've overridden them, just rename them to `compressor/css_file.html` or `compressor/js_file.html` and make sure you've accounted for the backwards incompatible change of the template context mentioned above.
- Reverted an unfortunate change to the YUI filter that prepended `'java -jar'` to the binary name, which doesn't always work, e.g. if the YUI compressor is shipped as a script like `/usr/bin/yui-compressor`.
- Changed the sender parameter of the `post_compress()` signal to be either `compressor.css.CssCompressor` or `compressor.js.JsCompressor` for easier customization.
- Correctly handle offline compressing files that are found in `{% if %}` template blocks.
- Renamed the second option for the `COMPRESS_CSS_HASHING_METHOD` setting from `'hash'` to `'content'` to better describe what it does. The old name is also supported, as well as the default being `'mtime'`.
- Fixed `CssAbsoluteFilter`, `src` attributes in includes now get transformed.
- Added a new hook to allow developers to completely bypass offline compression in `CompressorNode` subclasses: `is_offline_compression_enabled`.
- Dropped `versiontools` from required dependencies again.

2.11.4 v1.1.2

- Fixed an installation issue related to `versiontools`.

2.11.5 v1.1.1

- Fixed a stupid `ImportError` bug introduced in 1.1.
- Fixed Jinja2 docs of since `JINJA2_EXTENSIONS` expects a class, not a module.
- Fixed a Windows bug with regard to file resolving with `staticfiles` finders.
- Stopped a potential memory leak when memoizing the rendered output.
- Fixed the integration between `staticfiles` (e.g. in Django `<= 1.3.1`) and `compressor` which prevents the `collectstatic` management command to work.

Warning: Make sure to **remove** the `path` method of your custom *remote storage* class!

2.11.6 v1.1

- Made offline compression completely independent from cache (by writing a `manifest.json` file).
You can now easily run the `compress` management command locally and transfer the `COMPRESS_ROOT` dir to your server.
- Updated installation instructions to properly mention all dependencies, even those internally used.
- Fixed a bug introduced in 1.0 which would prevent the proper deactivation of the compression in production.
- Added a `Jinja2 contrib extension`.

- Made sure the rel attribute of link tags can be mixed case.
- Avoid overwriting context variables needed for compressor to work.
- Stopped the compress management command to require model validation.
- Added missing imports and fixed a few [PEP 8](#) issues.

2.11.7 v1.0.1

- Fixed regression in `compressor.utils.staticfiles` compatibility module.

2.11.8 v1.0

- **BACKWARDS-INCOMPATIBLE** Stopped swallowing exceptions raised by rendering the template tag in production (`DEBUG = False`). This has the potential to breaking lots of apps but on the other hand will help find bugs.
- **BACKWARDS-INCOMPATIBLE** The default function to create the cache key stopped containing the server hostname. Instead the cache key now only has the form `'django_compressor.<KEY>'`.
To revert to the previous way simply set the `COMPRESS_CACHE_KEY_FUNCTION` to `'compressor.cache.socket_cachekey'`.
- **BACKWARDS-INCOMPATIBLE** Renamed ambiguously named `COMPRESS_DATA_URI_MAX_SIZE` setting to `COMPRESS_DATA_URI_MAX_SIZE`. It's the maximum size the `compressor.filters.datauri.DataUriFilter` filter will embed files as data: URIs.
- Added `COMPRESS_CSS_HASHING_METHOD` setting with the options `'mtime'` (default) and `'hash'` for the `CssAbsoluteFilter` filter. The latter uses the content of the file to calculate the cache-busting hash.
- Added support for `{{ block.super }}` to compress management command.
- Dropped Django 1.1.X support.
- Fixed compiler filters on Windows.
- Handle new-style cached template loaders in the compress management command.
- Documented included filters.
- Added [Slim It](#) filter.
- Added new `CallbackOutputFilter` to ease the implementation of Python-based callback filters that only need to pass the content to a callable.
- Make use of [django-appconf](#) for settings handling and [versiontools](#) for versions.
- Uses the current context when rendering the render templates.
- Added `post_compress` signal.

2.11.9 v0.9.2

- Fixed stdin handling of precompiler filter.

2.11.10 v0.9.1

- Fixed encoding related issue.
- Minor cleanups.

2.11.11 v0.9

- Fixed the precompiler support to also use the full file path instead of a temporarily created file.
- Enabled test coverage.
- Refactored caching and other utility code.
- Switched from SHA1 to MD5 for hash generation to lower the computational impact.

2.11.12 v0.8

- Replace naive jsmin.py with rJSmin (<http://opensource.perlig.de/rjsmin/>) and fixed a few problems with JavaScript comments.
- Fixed converting relative URLs in CSS files when running in debug mode.

Note: If you relied on the `split_contents` method of `Compressor` classes, please make sure a fourth item is returned in the iterable that denotes the base name of the file that is compressed.

2.11.13 v0.7.1

- Fixed import error when using the standalone django-staticfiles app.

2.11.14 v0.7

- Created new parser, `HtmlParser`, based on the stdlib `HTMLParser` module.
- Added a new default `AutoSelectParser`, which picks the `LxmlParser` if `lxml` is available and falls back to `HtmlParser`.
- Use `unittest2` for testing goodness.
- Fixed YUI JavaScript filter argument handling.
- Updated bundled jsmin to use version by Dave St.Germain that was refactored for speed.

2.11.15 v0.6.4

- Fixed Closure filter argument handling.

2.11.16 v0.6.3

- Fixed options mangling in CompilerFilter initialization.
- Fixed tox configuration.
- Extended documentation and README.
- In the compress command ignore hidden files when looking for templates.
- Restructured utilities and added staticfiles compat layer.
- Restructured parsers and added a html5lib based parser.

2.11.17 v0.6.2

- Minor bugfixes that caused the compression not working reliably in development mode (e.g. updated files didn't trigger a new compression).

2.11.18 v0.6.1

- Fixed staticfiles support to also use its finder API to find files during development – when the static files haven't been collected in `STATIC_ROOT`.
- Fixed regression with the `COMPRESS` setting, pre-compilation and staticfiles.

2.11.19 v0.6

Major improvements and a lot of bugfixes, some of which are:

- New precompilation support, which allows compilation of files and hunks with easily configurable compilers before calling the actual output filters. See the `COMPRESS_PRECOMPILERS` for more details.
- New staticfiles support. With the introduction of the staticfiles app to Django 1.3, compressor officially supports finding the files to compress using the app's finder API. Have a look at the documentation about *remote storages* in case you want to use those together with compressor.
- New `compress` management command which allows pre-running of what the `compress` template tag does. See the *pre-compression* docs for more information.
- Various performance improvements by better caching and mtime checking.
- Deprecated `COMPRESS_LESSC_BINARY` setting because it's now superseded by the `COMPRESS_PRECOMPILERS` setting. Just make sure to use the correct mimetype when linking to less files or adding inline code and add the following to your settings:

```
COMPRESS_PRECOMPILERS = (  
    ('text/less', 'lessc {infile} {outfile}'),  
)
```

- Added `cssmin` filter (`compressor.filters.CSSMinFilter`) based on Zachary Voase's Python port of the YUI CSS compression algorithm.
- Reimplemented the dog-piling prevention.
- Make sure the `CssAbsoluteFilter` works for relative paths.
- Added inline render mode. See *usage* docs.

- Added `mtime_cache` management command to add and/or remove all mtimes from the cache.
- Moved docs to Read The Docs: <http://django-compressor.readthedocs.org/en/latest/>
- Added optional `compressor.storage.GzipCompressorFileStorage` storage backend that gzips of the saved files automatically for easier deployment.
- Reimplemented a few filters on top of the new `compressor.filters.base.CompilerFilter` to be a bit more DRY.
- Added tox based test configuration, testing on Django 1.1-1.3 and Python 2.5-2.7.

C

- COMPRESS_CACHE_BACKEND (in module django.conf.settings), 15
- COMPRESS_CACHE_KEY_FUNCTION (in module django.conf.settings), 16
- COMPRESS_CLOSURE_COMPILER_ARGUMENTS (in module django.conf.settings), 12
- COMPRESS_CLOSURE_COMPILER_BINARY (in module django.conf.settings), 12
- COMPRESS_CSS_FILTERS (in module django.conf.settings), 11
- COMPRESS_CSS_HASHING_METHOD (in module django.conf.settings), 11
- COMPRESS_CSSTIDY_ARGUMENTS (in module django.conf.settings), 11
- COMPRESS_CSSTIDY_BINARY (in module django.conf.settings), 11
- COMPRESS_DATA_URI_MAX_SIZE (in module django.conf.settings), 11
- COMPRESS_DEBUG_TOGGLE (in module django.conf.settings), 15
- COMPRESS_ENABLED (in module django.conf.settings), 10
- COMPRESS_JS_FILTERS (in module django.conf.settings), 12
- COMPRESS_MINT_DELAY (in module django.conf.settings), 15
- COMPRESS_MTIME_DELAY (in module django.conf.settings), 15
- COMPRESS_OFFLINE (in module django.conf.settings), 16
- COMPRESS_OFFLINE_CONTEXT (in module django.conf.settings), 16
- COMPRESS_OFFLINE_MANIFEST (in module django.conf.settings), 16
- COMPRESS_OFFLINE_TIMEOUT (in module django.conf.settings), 16
- COMPRESS_OUTPUT_DIR (in module django.conf.settings), 11
- COMPRESS_PARSER (in module django.conf.settings), 14
- COMPRESS_PRECOMPILERS (in module django.conf.settings), 13
- COMPRESS_REBUILD_TIMEOUT (in module django.conf.settings), 15
- COMPRESS_ROOT (in module django.conf.settings), 11
- COMPRESS_STORAGE (in module django.conf.settings), 14
- COMPRESS_TEMPLATE_FILTER_CONTEXT (in module django.conf.settings), 12, 13
- COMPRESS_URL (in module django.conf.settings), 11
- COMPRESS_YUGLIFY_BINARY (in module django.conf.settings), 12, 13
- COMPRESS_YUGLIFY_CSS_ARGUMENTS (in module django.conf.settings), 12
- COMPRESS_YUGLIFY_JS_ARGUMENTS (in module django.conf.settings), 13
- COMPRESS_YUI_BINARY (in module django.conf.settings), 12
- COMPRESS_YUI_CSS_ARGUMENTS (in module django.conf.settings), 12
- COMPRESS_YUI_JS_ARGUMENTS (in module django.conf.settings), 13
- compressor.signals.post_compress() (built-in function), 8

P

- Python Enhancement Proposals
PEP 8, 27