1 Why another static file combiner for Django? 3

2 Contents
   2.1 Quickstart ................................................................. 5
   2.2 Usage ................................................................. 5
   2.3 Common Deployment Scenarios .............................................. 6
   2.4 Settings ................................................................. 10
   2.5 Remote Storages ......................................................... 10
   2.6 Behind the Scenes ....................................................... 18
   2.7 Jinja2 Support ............................................................. 19
   2.8 django-sekizai Support .................................................... 20
   2.9 Facebook React Support ................................................ 22
   2.10 Contributing ............................................................. 23
   2.11 Changelog ................................................................. 24

Index 37
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** It 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 you have to add Django Compressor’s file finder to the STATICFILES_FINDERS setting, like this:

```
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 Optional Dependencies

- BeautifulSoup
  For the `parser` `compressor.parser.BeautifulSoupParser` and `compressor.parser.LxmlParser`:
  ```
  pip install beautifulsoup4
  ```

- 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
  ```

- Calmjs
  For the `Calmjs filter` `compressor.filters.jsmin.CalmjsFilter`:
  ```
  pip install calmjs.parse
  ```

- csscompressor
  For the `csscompressor filter` `compressor.filters.cssmin.CSSCompressorFilter`:
  ```
  pip install csscompressor
  ```

- brotli
  For brotli file compressor `compressor.storage.BrotliCompressorFileStorage`:
  ```
  pip install brotli
  ```

2.2 Usage

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

2.2.1 Examples

Basic example:

```
{% compress css %}
<link rel="stylesheet" href="/static/css/one.css" type="text/css" charset="utf-8">
<style type="text/css">p { border:5px solid green;}</style>
``` (continues on next page)
Adding the `inline` parameter will put the content directly to the rendered page instead of a file:

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

Result:

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

Adding the `preload` parameter will generate the preload tag for the compressed resource in the template:

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

Result:

```html
<link rel="preload" href="/static/CACHE/js/d01466eb4fc6.js" as="script" />
```

Specifying a `block_name` will change the output filename. It can also be accessed in the `post_compress signal` in the context parameter.

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

Result:

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

Javascript `async` and `defer` are supported:

```html
{% compress js %}
<script src="/static/js/one.js" async></script>
{% endcompress %}
{% compress js %}
<script src="/static/js/one.js" defer></script>
{% endcompress %}
```

2.2. Usage
Result:

```html
<script async src="/static/CACHE/js/output.ccbb802fbf33.js"></script>
<script defer src="/static/CACHE/js/output.5bd495b0eb05.js"></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 `COMPRESS_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).

2.2.2 Offline Compression

Django Compressor has the ability to run the compression “offline”, i.e. outside of the request/response loop – independent from user requests. If offline compression is enabled, no new files are generated during a request and the `{% compress %}` tag simply inserts links to the files in the offline cache (see Behind the Scenes for details). This results in better performance and enables certain deployment scenarios (see Common Deployment Scenarios).

To use offline compression, enable the `django.conf.settings.COMPRESS_OFFLINE` setting and then run the `compress` management command to compress your assets and update the offline cache.

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.:

```html
{% load compress %}
{% compress js %}
<script type="text/javascript">
  alert("{{ greeting }}");
</script>
{% endcompress %}
```

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

```python
COMPRESS_OFFLINE_CONTEXT = {
    'greeting': 'Hello there!',
}
```
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.
- You want the best possible performance.

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.

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

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

```html
<script type="text/javascript" src="/static/js/site-base.js"></script>
<script type="text/javascript" src="/static/CACHE/js/awesome.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_FILTERS

**Default** {'css': ['compressor.filters.css_default.CssAbsoluteFilter', 'compressor.filters.cssmin.CSSMinFilter'], 'js': ['compressor.filters.jsmin.JSMinFilter']}

A mapping of resource kinds to the list of filters to apply. The key is used to refer to this resource type in templates when using the {% compress [resource_kind] %} template tag. The value is a list of filters to apply in the given order for that resource.

This library currently includes filters for CSS and Javascript.
• CSS Filters

    default ['compressor.filters.css_default.CssAbsoluteFilter',
            'compressor.filters.cssmin.rCSSMinFilter']

A list of filters that will be applied to CSS.

Possible options for CSS filters are (including their settings):

- `compressor.filters.css_default.CssAbsoluteFilter`

  A filter that normalizes the URLs used in `url()` CSS statements. This is necessary since the output css files produced by Django Compressor are in a different location than the source files and relative paths might have become invalid. The filter also appends a hash as query string to the normalized URLs to help with cache busting.

  `django.conf.settings.COMPRESS_CSS_HASHING_METHOD`

  The method to use when calculating the query string. Either `None`, `mtime` (default) or `content`. Use the `None` if you want to completely disable that feature, and the `content` in case you’re using multiple servers to serve your content.

- `compressor.filters.css_default.CssRelativeFilter`

  An alternative to `CssAbsoluteFilter`. It uses a relative instead of an absolute path to prefix URLs. Specifically, the prefix will be `../` * (N + 1) where N is the depth of settings. `COMPRESS_OUTPUT_DIR` folder (i.e. 1 for `CACHE`, or 2```for ```CACHE/data etc). This can be useful if you don’t want to hard-code `COMPRESS_URL` into CSS code.

- `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.CSSCompressorFilter`

  A filter that uses Yury Selivanov’s Python port of the YUI CSS compression algorithm csscompressor.

- `compressor.filters.cssmin.rCSSMinFilter`

  A filter that uses the cssmin implementation rCSSmin to compress CSS (installed by default).
- `compressor.filters.cleancss.CleanCSSFilter`
  A filter that passes the CSS content to the clean-css tool.
- `django.conf.settings.COMPRESS_CLEAN_CSS_BINARY`
  The clean-css binary filesystem path.
- `django.conf.settings.COMPRESS_CLEAN_CSS_ARGUMENTS`
  The arguments passed to clean-css.

- `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.

- Javascript Filters
  
  **Default** ['compressor.filters.jsmin.rJSMinFilter']

  A list of filters that will be applied to javascript.

  Possible options are:
  
  - `compressor.filters.jsmin.rJSMinFilter`
    A filter that uses the jsmin implementation rJSmin to compress JavaScript code (installed by default).
  
  - `compressor.filters.jsmin.CalmjsFilter`
    A filter that uses the jsmin implementation Calmjs 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. mimetyp e 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:

```python
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:

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

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

```template
{% 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:

```html
<script type="text/javascript" src="/static/CACHE/js/output.8dd1a2872443.js" charset="utf-8"></script>
```
The same works for less, too:

```html
{% load compressor %}
{% 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:

```html
<link rel="stylesheet" href="/static/CACHE/css/output.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 some additional storage backends:

- 'compressor.storage.GzipCompressorFileStorage'
  
  A subclass of the default storage backend, which will additionally create *.gz files of each of the compressed files.

- 'compressor.storage.BrotliCompressorFileStorage'
  
  A subclass of the default storage backend, which will additionally create *.br files of each of the compressed files. It is using the maximum level of compression (11) so compression speed will be low.

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

2.4.3 Caching settings

django.conf.settings.COMPRESS_CACHE_BACKEND

Default "default"

The cache to use by Django Compressor. Must be a cache alias specified in your CACHES 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_CACHEABLE_PRECOMPILERS

Default ()

An iterable of precompiler mimetypes as defined in COMPRESS_PRECOMPILERS for which the compiler output can be cached based solely on the contents of the input file. This lets Django Compressor avoid recompiling unchanged files. Caching is appropriate for compilers such as CoffeeScript where files are compiled one-to-one, but not for compilers such as SASS that have an import mechanism for including one file from another. If caching is enabled for such a compiler, Django Compressor will not know to recompile files when a file they import is modified.

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:

```python
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 be done outside of the request/response loop. See Offline Compression for details.

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.

Note: It is also possible to perform offline compression for multiple contexts by providing a list or tuple of dictionaries, or by providing a dotted string pointing to a generator function.

This makes it easier to generate contexts dynamically for situations where a user might be able to select a different theme in their user profile, or be served different stylesheets based on other criteria.

An example of multiple offline contexts by providing a list or tuple:

```python
# project/settings.py:
COMPRESS_OFFLINE_CONTEXT = [
    {'THEME': 'plain', 'STATIC_URL': STATIC_URL},
    {'THEME': 'fancy', 'STATIC_URL': STATIC_URL},
    # ...
]
```

An example of multiple offline contexts generated dynamically:

```python
# project/settings.py:
COMPRESS_OFFLINE_CONTEXT = 'project.module.offline_context'

# project/module.py:
from django.conf import settings
def offline_context():
    # ...
```

(continues on next page)
from project.models import Company
for theme in set(Company.objects.values_list('theme', flat=True)):
    yield {'THEME': theme, 'STATIC_URL': settings.STATIC_URL}

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.

django.conf.settings.COMPRESS_OFFLINE_MANIFEST_STORAGE

Default compressor.storage.OfflineManifestFileStorage

The dotted path to a Django Storage backend to be used to save the offline manifest.

By default, the file configured with COMPRESS_OFFLINE_MANIFEST will be stored into
COMPRESS_OUTPUT_DIR.

An example to output the manifest into the project’s root directory:

# project/settings.py:
COMPRESS_STORAGE = 'project.module.PrivateOfflineManifestFileStorage'

# project/module.py:
from compressor.storage import OfflineManifestFileStorage
from django.conf import settings
class PrivateOfflineManifestFileStorage(OfflineManifestFileStorage):
    def __init__(self, *args, **kwargs):
        super().__init__(settings.BASE_DIR, None, *args, **kwargs)

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 boto3 storage backend from the 3rd party app django-storages. Some required settings are:

AWS_ACCESS_KEY_ID = 'XXXXXXXXXXXXX'
AWS_SECRET_ACCESS_KEY = 'XXXXXXXXXXXXX'
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.s3boto3.S3Boto3Storage'
```

### 2.5.2 Using staticfiles

If you are using Django’s `staticfiles` contrib app, 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 doing their job.

2. You need to create a subclass of the remote storage backend you want to use; below is an example of the boto3 S3 storage backend from `django-storages`:

   ```python
   from django.core.files.storage import get_storage_class
   from storages.backends.s3boto3 import S3Boto3Storage

   class CachedS3Boto3Storage(S3Boto3Storage):
       """S3 storage backend that saves the files locally, too."
       def __init__(self, *args, **kwargs):
           super().__init__(*args, **kwargs)
           self.local_storage = get_storage_class("compressor.storage.CompressorFileStorage")()

       def save(self, name, content):
           self.local_storage._save(name, content)
           super().save(name, self.local_storage._open(name))
           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.CachedS3Boto3Storage`.

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/").

In the end it might look like this:

```
STATIC_ROOT = '/path/to/staticfiles'
COMPRESS_ROOT = STATIC_ROOT
STATICFILES_STORAGE = 'mysite.storage.CachedS3Boto3Storage'
COMPRESS_STORAGE = STATICFILES_STORAGE
STATIC_URL = 'https://compressor-test.s3.amazonaws.com/
COMPRESS_URL = STATIC_URL
```

### 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 compression

If offline compression is activated, the \{ % compress % \} tag will try to 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. The \compress\ management command generates the offline manifest as well as the combined files referenced in the manifest.

If offline compression is enabled and the nodelist hash can not be found inside the manifest, \{ % compress % \} will raise an \OfflineGenerationError\. If offline compression is disabled, 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 Support

Django Compressor comes with support for Jinja2 via an extension.

#### 2.7.1 In-Request Compression

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:

```python
from django.conf import settings
template = env.from_string('
'.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 Offline Compression

**Usage**

First, you will 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.

Then, 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`.

**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).

Templates of both engines can be compressed like this:

- `./manage.py compress --engine django --engine jinja2`

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

**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.

**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.
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.

### Using your custom loader

You should configure `TEMPLATE_LOADERS` as such:

```python
TEMPLATE_LOADERS = ('your_app.Loader',
                   ...
                   ...
)
```

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 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 be installed. Refer to the django-sekizai docs for how to use `render_block`

Please note that the sekizai integration does not work with offline compression. See this issue for details.

#### 2.8.1 Usage

In templates which either extend base templates or are included by other templates, use any of these directives to import Stylesheets and JavaScript files:

```bash
{% load static sekizai_tags %}
{% addtoblock "css" %}
     <link href="https://cdnjs.cloudflare.com/ajax/libs/select2/4.0.5/css/select2.min.css" rel="stylesheet" />
{% endaddtoblock %}
{% addtoblock "css" %}
     <link href="{% static 'app/css/mystyle.css' %}" rel="stylesheet" type="text/css" />
{% endaddtoblock %}
{% addtoblock "js" %}
     <script src="https://cdnjs.cloudflare.com/ajax/libs/select2/4.0.5/js/select2.min.js"></script>
{% endaddtoblock %}
{% addtoblock "js" %}
     <script src="{% static 'js/js/myapp.js' %}" type="text/javascript"></script>
{% endaddtoblock %}
{% addtoblock "js" %}
     <script async="async" defer="defer" src="https://maps.googleapis.com/maps/api/js?key=\{{config.apiKey}\}&callback=initMap"></script>
{% endaddtoblock %}
```

Note that some files are loaded by the browser directly from a CDN and thus can not be compressed by django-compressor. Therefore the Sekizai compressor checks whether a file is compressable, and only if so, concatenates its payload.
In the above example, we render StyleSheets inside the <head> element and JavaScript files just before closing the </body> tag.

Here, we first render some references to CSS and JavaScript files, served from external sources. Afterwards all local files are concatenated and optionally minified. In the last step all references to deferred files are rendered.

### 2.9 Facebook React Support

Assuming you have *npm* available, you can install *babel* via `npm install -g babel` and integrate React with Django Compressor by following the react-tools installation instructions and adding an appropriate `COMPRESS_PRECOMPILERS` setting:

```python
COMPRESS_PRECOMPILERS = {
    ('text/jsx', 'cat {infile} | babel > {outfile}'),
}
```

If the above approach is not suitable for you, compiling React’sjsx files can be done by creating a custom precompressor.

#### 2.9.1 Requirements

- PyReact=0.5.2 for compiling jsx files
- PyExecJS=1.1.0 required by PyReact (automatically installed when using pip)
- A Javascript runtime : options include PyV8, Node.js, PhantomJS among others

The full list of supported javascript engines can be found here: https://github.com/doloopwhile/PyExecJS

#### 2.9.2 Installation

1. Place the following code in a Python file (e.g. `third_party/react_compressor.py`). Also make sure that `third_party/__init__.py` exists so the directory is recognized as a Python package.

```python
from compressor.filters import FilterBase
from react import jsx

class ReactFilter(FilterBase):
```

(continues on next page)
```python
def __init__(self, content, *args, **kwargs):
    self.content = content
    kwargs.pop('filter_type')
    super().__init__(content, *args, **kwargs)

def input(self, **kwargs):
    return jsx.transform_string(self.content)
```

2. In your Django settings, add the following line:

```python
COMPRESS_PRECOMPILERS = {
    ('text/jsx', 'third_party.react_compressor.ReactFilter'),
}
```

Where `third_party.react_compressor.ReactFilter` is the full name of your `ReactFilter` class.

### 2.9.3 Troubleshooting

If you get “file not found” errors, open your Python command line and make sure you are able to import your `ReactFilter` class:

```python
__import__('third_party.react_compressor.ReactFilter')
```

### 2.10 Contributing

Like every open-source project, Django Compressor is always looking for motivated individuals to contribute to its 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 open an issue on GitHub explaining your ideas.

#### 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

24 Chapter 2. Contents
Getting the source code

If you’re interested in developing a new feature for Compressor, it is recommended that you first discuss it in an issue in order not to do any work that wouldn’t get merged 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).

Running the tests

To run the tests simply fork django_compressor, make the changes and open a pull request. The Github actions will automatically run the tests of your branch/fork and add a status about the test results to the pull requests.

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 Unreleased

2.11.2 v4.1 (2022-08-03)

Full Changelog
• Add Django 4.1 compatibility
• New setting COMPRESS_OFFLINE_MANIFEST_STORAGE to customize the offline manifest’s file storage (#1112)

With this change the function compressor.cache.get_offline_manifest_filename() has been removed. You can now use the new file storage compressor.storage.default_offline_manifest_storage to access the location of the manifest.

2.11.3 v4.0 (2022-03-23)

Full Changelog

• Fix intermittent No such file or directory errors by changing strategy to deal with duplicate filenames in CompressorFileStorage
• Deprecate SlimItFilter, stop testing it with Python 3.7 or higher

2.11.4 v3.1 (2021-12-18)

Full Changelog

• Fix error with verbose offline compression when COMPRESS_OFFLINE_CONTEXT is a generator

2.11.5 v3.0 (2021-12-12)

Full Changelog

• Officially support for Python 3.9 and 3.10 as well as Django 3.1, 3.2 and 4.0
• Drop support for Django 1.11, 2.1 and 3.0
• Drop support for Python 2.x and 3.4
• Fix compatibility with Jinja 3.x
• Require django-sekizai 2.0 for django-sekizai extension
• Make template compression in compress command threaded to improve performance
• Correctly handle relative paths in extends tags (#979)
• Enable rCSSMinFilter by default (#888)
• Fix various deprecation warnings
• Add ability to pass a logger to various classes & methods
• Removed deprecated COMPRESS_JS_FILTERS and COMPRESS_CSS_FILTERS settings
• Fix offline compression race condition, which could result in incomplete manifest

2.11.6 v2.4.1 (2021-04-17)

Full Changelog

• Raise proper DeprecationWarning for COMPRESS_JS_FILTERS and COMPRESS_CSS_FILTERS settings
2.11.7 v2.4 (2019-12-31)

Full Changelog

- Add support for Django 3.0 (#950, #967)
- Officially support Python 3.8 (#967)
- Add better support for JS strict mode and validation (#952)
- Add support for rel=preload (#951)
- Add support for Calmjs (#957)

Note: in 2.3, a new setting COMPRESS_FILTERS has been introduced that combines the existing COMPRESS_CSS_FILTERS and COMPRESS_JS_FILTERS. The latter are now deprecated. See the docs on how to use the new setting, the conversion is straightforward.

2.11.8 v2.3 (2019-05-31)

Full Changelog

- Drop support for Django 1.8, 1.9 and 1.10
- Add support for Django 2.1 and 2.2, as well as Python 3.7
- Update all dependencies. This required minor code changes, you might need to update some optional dependencies if you use any
- Allow the mixed use of JS/CSS in Sekizai’s templatetags {% addtoblock "js" %} and {% addtoblock "css" %} (#891)
- Allow the implementation of new types other than css and js. (#900)
- Update jinja2 extension to behave similar to the django tag (#899)
- Fix crash in offline compression when child nodelist is None, again (#605)
- Support STATIC_URL and COMPRESS_URL being string-like objects
- Improve compress command memory usage (#870)
- Ensure generated file always contains a base name (#775)
- Add BrotliCompressorFileStorage (#867)

2.11.9 v2.2 (2017-08-16)

Full Changelog

- Switch from MD5 to SHA256 for hashes generation.
- Add Django 1.11 compatibility
- Various compatibility fixes for Python 3.6 and Django 1.8
- Made OfflineGenerationError easier to debug
- Drop support for Python 3.2
- Add new CssRelativeFilter which works like CssAbsoluteFilter but outputs relative URLs.
- Fix URL CssAbsoluteFilter URL detection
2.11.10 v2.1.1 (2017-02-02)

• Fix to file permissions issue with packaging.

2.11.11 v2.1 (2016-08-09)

Full Changelog

• Add Django 1.10 compatibility
• Add support for inheritance using a variable in offline compression
• Fix recursion error with offline compression when extending templates with the same name
• Fix UnicodeDecodeError when using CompilerFilter and caching
• Fix CssAbsoluteFilter changing double quotes to single quotes, breaking SVG

2.11.12 v2.0 (2016-01-07)

Full Changelog

• Add Django 1.9 compatibility
• Remove official support for Django 1.4 and 1.7
• Add official support for Python 3.5
• Remove official support for Python 2.6
• Remove support for coffin and jingo
• Fix Jinja2 compatibility for Django 1.8+
• Stop bundling vendored versions of rcssmin and rjsmin, make them proper dependencies
• Remove support for CSSTidy
• Remove support for beautifulsoup 3.
• Replace cssmin by csscompressor (cssmin is still available for backwards-compatibility but points to rcssmin)

2.11.13 v1.6 (2015-11-19)

Full Changelog

• Upgrade rcssmin and rjsmin
• Apply CssAbsoluteFilter to precompiled css even when compression is disabled
• Add optional caching to CompilerFilter to avoid re-compiling unchanged files
• Fix various deprecation warnings on Django 1.7 / 1.8
• Fix TemplateFilter
• Fix double-rendering bug with sekizai extension
• Fix debug mode using destination directory instead of staticfiles finders first
• Removed some silent exception catching in compress command
## 2.11.14 v1.5 (2015-03-27)

**Full Changelog**

- Fix compress command and run automated tests for Django 1.8
- Fix Django 1.8 warnings
- Handle TypeError from import_module
- Fix reading UTF-8 files which have BOM
- Fix incompatibility with Windows (shell_quote is not supported)
- Run automated tests on Django 1.7
- Ignore non-existent `{{{ block.super }}}` in offline compression instead of raising AttributeError
- Support for clean-css
- Fix link markup
- Add support for COMPRESS_CSS_HASHING_METHOD = None
- Remove compatibility with old `staticfiles` app
- In compress command, use `get_template()` instead of opening template files manually, fixing compatibility issues with custom template loaders
- Fix FilterBase so that does not override self.type for subclasses if filter_type is not specified at init
- Remove unnecessary filename and existence checks in CssAbsoluteFilter

## 2.11.15 v1.4 (2014-06-20)

- Added Python 3 compatibility.
- Added compatibility with Django 1.6.x and dropped support for Django 1.3.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.
2.11.16 v1.3 (2013-03-18)

• Backward incompatible changes
  – Dropped support for Python 2.5. Removed any and walk compatibility functions in compressor.utils.
  – Removed compatibility with some old django 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.

2.11.17 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 namespace in a dictionary, so it’s easy to fix your own templates.

For example, the old compressor/js_file.html:

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

The new compressor/js_file.html:

```html
"""
- 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.

2.11.18 v1.1.2

- Fixed an installation issue related to `versiontools`.

2.11.19 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!

2.11.20 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.

2.11.21 v1.0.1

• Fixed regression in compressor.utils.staticfiles compatibility module.

2.11.22 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.

2.11.23 v0.9.2

• Fixed stdin handling of precompiler filter.
2.11.24 v0.9.1

- Fixed encoding related issue.
- Minor cleanups.

2.11.25 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.

2.11.26 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.

2.11.27 v0.7.1

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

2.11.28 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.

2.11.29 v0.6.4

- Fixed Closure filter argument handling.
2.11.30 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.
- Restructered parsers and added a html5lib based parser.

2.11.31 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).

2.11.32 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.

2.11.33 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:

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

- Added cssmin filter (compressor.filters.CSSMinFilter) based on Zachary Vouëse’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: https://django-compressor.readthedocs.io/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.
COMPRESS_CACHE_BACKEND (in module django.conf.settings), 16
COMPRESS_CACHE_KEY_FUNCTION (in module django.conf.settings), 17
COMPRESS_CACHEABLE_PRECOMPILERS (in module django.conf.settings), 16
COMPRESS_CLEAN_CSS_ARGUMENTS (in module django.conf.settings), 13
COMPRESS_CLEAN_CSS_BINARY (in module django.conf.settings), 13
COMPRESS_CLOSURE_COMPILER_ARGUMENTS (in module django.conf.settings), 13
COMPRESS_CLOSURE_COMPILER_BINARY (in module django.conf.settings), 13
COMPRESS_CSS_HASHING_METHOD (in module django.conf.settings), 12
COMPRESS_DATA_URI_MAX_SIZE (in module django.conf.settings), 12
COMPRESS_DEBUG_TOGGLE (in module django.conf.settings), 16
COMPRESS_ENABLED (in module django.conf.settings), 11
COMPRESS_FILTERS (in module django.conf.settings), 11
COMPRESS_MINT_DELAY (in module django.conf.settings), 11
COMPRESS_MTIME_DELAY (in module django.conf.settings), 16
COMPRESS_OFFLINE (in module django.conf.settings), 17
COMPRESS_OFFLINE_CONTEXT (in module django.conf.settings), 17
COMPRESS_OFFLINE_MANIFEST (in module django.conf.settings), 18
COMPRESS_OFFLINE_MANIFEST_STORAGE (in module django.conf.settings), 18
COMPRESS_OFFLINE_TIMEOUT (in module django.conf.settings), 17
COMPRESS_OUTPUT_DIR (in module django.conf.settings), 11
COMPRESS_PARSER (in module django.conf.settings), 15
COMPRESS_PRECOMPILERS (in module django.conf.settings), 14
COMPRESS_REBUILD_TIMEOUT (in module django.conf.settings), 16
COMPRESS_ROOT (in module django.conf.settings), 11
COMPRESS_STORAGE (in module django.conf.settings), 15
COMPRESS_TEMPLATE_FILTER_CONTEXT (in module django.conf.settings), 13, 14
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, 13
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), 9

P

Python Enhancement Proposals
PEP 8, 33