# webassets Documentation

***Release 0.12.1***

**Michael Elsdörfer**

**Dec 10, 2018**

# Contents

CHAPTER 1

# Introduction

webassets is a general, dependency-independent library for managing the assets of your web application. It can merge and compress your CSS and JavaScript files, supporting a wide variety of different filters, and supports working with compilers like CoffeeScript or Sass.

CHAPTER 2

# Framework integration

For some web frameworks, `webassets` provides special integration. If you are using one of the supported frameworks, go to the respective page:

## 2.1 Using `webassets` in standalone mode

You don't need to use one of the frameworks into which `webassets` can integrate. Using the underlying facilities directly is almost as easy.

And depending on what libraries you use, there may still be some things *webassets* can help you with, see *Integration with other libraries*.

### 2.1.1 Quick Start

First, create an environment instance:

```
from webassets import Environment
my_env = Environment(
    directory='../static/media',
    url='/media')
```

As you can see, the environment requires two arguments:

- the path in which your media files are located
- the url prefix under which the media directory is available. This prefix will be used when generating output urls.

Next, you need to define your assets, in the form of so called *bundles*, and register them with the environment. The easiest way to do it is directly in code:

```
from webassets import Bundle
js = Bundle('common/jquery.js', 'site/base.js', 'site/widgets.js',
```

(continues on next page)

(continued from previous page)

```
            filters='jsmin', output='gen/packed.js')
my_env.register('js_all', js)
```

However, if you prefer, you can of course just as well define your assets in an external config file, and read them from there. `webassets` includes a number of *helper classes* for some popular formats like YAML.

#### Using the bundles

Now with your assets properly defined, you want to merge and minify them, and include a link to the compressed result in your web page. How you do this depends a bit on how your site is rendered.

```
>>> my_env['js_all'].urls()
('/media/gen/packed.js?9ae572c',)
```

This will always work. You can call your bundle's `urls()` method, which will automatically merge and compress the source files, and return the url to the final output file. Or, in debug mode, it would return the urls of each source file:

```
>>> my_env.debug = True
>>> my_env['js_all'].urls()
('/media/common/jquery.js',
 '/media/site/base.js',
 '/media/site/widgets.js',)
```

Take these urls, pass them to your templates, or otherwise ensure they'll be used on your website when linking to your Javascript and CSS files.

For some templating languages, `webassets` provides extensions to access your bundles directly within the template. See *Integration with other libraries* for more information.

#### Using the Command Line Interface

See *Command Line Interface*.

### 2.1.2 Further Reading

#### The environment

The environment has two responsibilities: One, it acts as a registry for bundles, meaning you only have to pass around a single object to access all your bundles.

Also, it holds the configuration.

#### Registering bundles

Bundles can be registered with the environment:

```
my_bundle = Bundle(...)
environment.register('my_bundle', my_bundle)
```

A shortcut syntax is also available - you may simply call `register()` with the arguments which you would pass to the `Bundle` constructor:

```
environment.register('my_bundle', 'file1.js', 'file2.js', output='packed.js')
```

The environment allows dictionary-style access to the registered bundles:

```
>>> len(environment)
1

>>> list(environment)
[<Bundle ...>]

>>> environment['my_bundle']
<Bundle ...>
```

### Configuration

The environment supports the following configuration options:

`Environment`**`.directory`**
: The base directory to which all paths will be relative to, unless *`load_path`* are given, in which case this will only serve as the output directory.

  In the url space, it is mapped to `urls`.

`Environment`**`.url`**
: The url prefix used to construct urls for files in *`directory`*.

  To define url spaces for other directories, see *`url_mapping`*.

`Environment`**`.debug`**
: Enable/disable debug mode. Possible values are:

  **`False`** Production mode. Bundles will be merged and filters applied.

  **`True`** Enable debug mode. Bundles will output their individual source files.

  ***"merge"*** Merge the source files, but do not apply filters.

`Environment`**`.auto_build`**
: Controls whether bundles should be automatically built, and rebuilt, when required (if set to `True`), or whether they must be built manually be the user, for example via a management command.

  This is a good setting to have enabled during debugging, and can be very convenient for low-traffic sites in production as well. However, there is a cost in checking whether the source files have changed, so if you care about performance, or if your build process takes very long, then you may want to disable this.

  By default automatic building is enabled.

`Environment`**`.url_expire`**
: If you send your assets to the client using a *far future expires* header (to minimize the 304 responses your server has to send), you need to make sure that assets will be reloaded by the browser when they change.

  If this is set to `True`, then the Bundle URLs generated by webassets will have their version (see `Environment.versions`) appended as a querystring.

  An alternative approach would be to use the `%(version)s` placeholder in the bundle output file.

  The default behavior (indicated by a `None` value) is to add an expiry querystring if the bundle does not use a version placeholder.

Environment.**versions**
Defines what should be used as a Bundle `version`.

A bundle's version is what is appended to URLs when the `url_expire` option is enabled, and the version can be part of a Bundle's output filename by use of the `%(version)s` placeholder.

Valid values are:

**timestamp** The version is determined by looking at the mtime of a bundle's output file.

**hash (default)** The version is a hash over the output file's content.

**False, None** Functionality that requires a version is disabled. This includes the `url_expire` option, the `auto_build` option, and support for the %(version)s placeholder.

Any custom version implementation.

Environment.**manifest**
A manifest persists information about the versions bundles are at.

The Manifest plays a role only if you insert the bundle version in your output filenames, or append the version as a querystring to the url (via the `url_expire` option). It serves two purposes:

- Without a manifest, it may be impossible to determine the version at runtime. In a deployed app, the media files may be stored on a different server entirely, and be inaccessible from the application code. The manifest, if shipped with your application, is what still allows to construct the proper URLs.
- Even if it were possible to determine the version at runtime without a manifest, it may be a costly process, and using a manifest may give you better performance. If you use a hash-based version for example, this hash would need to be recalculated every time a new process is started.

Valid values are:

**"cache" (default)** The cache is used to remember version information. This is useful to avoid recalculating the version hash.

**"file:{path}"** Stores version information in a file at {path}. If not path is given, the manifest will be stored as `.webassets-manifest` in `Environment.directory`.

**"json:{path}"** Same as "file:{path}", but uses JSON to store the information.

**False, None** No manifest is used.

Any custom manifest implementation.

Environment.**cache**
Controls the behavior of the cache. The cache will speed up rebuilding of your bundles, by caching individual filter results. This can be particularly useful while developing, if your bundles would otherwise take a long time to rebuild.

Possible values are:

**False** Do not use the cache.

**True (default)** Cache using default location, a `.webassets-cache` folder inside *directory*.

***custom path*** Use the given directory as the cache directory.

Environment.**load_path**
An list of directories that will be searched for source files.

If this is set, source files will only be looked for in these directories, and *directory* is used as a location for output files only.

To modify this list, you should use `append_path()`, since it makes it easy to add the corresponding url prefix to *url_mapping*.

Environment.**url_mapping**
A dictionary of directory -> url prefix mappings that will be considered when generating urls, in addition to the pair of *directory* and *url*, which is always active.

You should use `append_path()` to add directories to the load path along with their respective url spaces, instead of modifying this setting directly.

### Filter configuration

In addition to the standard options listed above, you can set custom configuration values using `Environment.config`. This is so that you can configure filters through the environment:

```
environment.config['sass_bin'] = '/opt/sass/bin/sass')
```

This allows the *Sass filter* to find the sass binary.

Note: Filters usually allow you to define these values as system environment variables as well. That is, you could also define a `SASS_BIN` environment variable to setup the filter.

### Bundles

A bundle is simply a collection of files that you would like to group together, with some properties attached to tell `webassets` how to do its job. Such properties include the filters which should be applied, or the location where the output file should be stored.

Note that all filenames and paths considered to be relative to the `directory` setting of your *environment*, and generated urls will be relative to the `url` setting.

```
Bundle('common/inheritance.js', 'portal/js/common.js',
       'portal/js/plot.js', 'portal/js/ticker.js',
       filters='jsmin',
       output='gen/packed.js')
```

A bundle takes any number of filenames, as well as the following keyword arguments:

- `filters` - One or multiple filters to apply. If no filters are specified, the source files will merely be merged into the output file. Filters are applied in the order in which they are given.
- `merge` - If `True` (the default), All source files will be merged and the result stored at `output`. Otherwise, each file will be processed separately.
- `output` - Name/path of the output file. A `%(version)s` placeholder is supported here, which will be replaced with the version of the file. See *URL Expiry (cache busting)*. If `merge` is `True`, this argument is required and you can also use these placeholders: - `%(name)s` Just the name of the source file, without path or extension (eg: 'common') - `%(path)s` The path and name of the source file (eg: 'portal/js/common') - `%(ext)s` The extension of source file (eg: 'js')
- `depends` - Additional files that will be watched to determine if the bundle needs to be rebuilt. This is usually necessary if you are using compilers that allow `@import` instructions. Commonly, one would use a glob instruction here for simplicity:

  ```
  Bundle(depends=('**/*.scss'))
  ```

  **Warning:** Currently, using `depends` disables caching for a bundle.

### Nested bundles

Bundles may also contain other bundles:

```
from webassets import Bundle

all_js = Bundle(
    # jQuery
    Bundle('common/libs/jquery/jquery.js',
        'common/libs/jquery/jquery.ajaxQueue.js',
        'common/libs/jquery/jquery.bgiframe.js',),
    # jQuery Tools
    Bundle('common/libs/jqtools/tools.tabs.js',
        'common/libs/jqtools/tools.tabs.history.js',
        'common/libs/jqtools/tools.tabs.slideshow.js'),
    # Our own stuff
    Bundle('common/inheritance.js', 'portal/js/common.js',
        'portal/js/plot.js', 'portal/js/ticker.js'),
    filters='jsmin',
    output='gen/packed.js')
```

Here, the use of nested `Bundle` objects to group the JavaScript files together is purely aesthetical. You could just as well pass all files as a flat list. However, there are some more serious application as well. One of them is the use of *CSS compilers*. Another would be dealing with pre-compressed files:

If you are using a JavaScript library like jQuery, you might find yourself with a file like `jquery.min.js` in your media directory, i.e. it is already minified - no reason to do it again.

While I would recommend always using the raw source files, and letting `webassets` do the compressing, if you do have minified files that you need to merge together with uncompressed ones, you could do it like so:

```
register('js-all',
    'jquery.min.js',
    Bundle('uncompressed.js', filters='jsmin'))
```

Generally speaking, nested bundles allow you to apply different sets of filters to different groups of files, but still everything together into a single output file.

Some things to consider when nesting bundles:

- Duplicate filters are only applied once (the leaf filter is applied).
- If a bundle that is supposed to be processed to a file does not define an output target, it simply serves as a container of its sub-bundles, which in turn will be processed into their respective output files. In this case it must not have any files of its own.
- A bundle with `merge=False` cannot contain nested bundles.

### Building bundles

Once a bundle is defined, the thing you want to do is build it, and then include a link to the final merged and compressed output file in your site.

There are different approaches.

### In Code

For starters, you can simply call the bundle's `urls()` method:

```
>>> env['all_js'].urls()
('/media/gen/packed.js',)
```

Depending on the value of `environment.debug`. it will either return a list of all the bundle's source files, or the merged file pointed to by the bundle's `output` option - all relative to the `environment.url` setting.

`urls()` will always ensure that the files behind the urls it returns actually exist. That is, it will merge and compress the source files in production mode when first called, and update the compressed assets when it detects changes. This behavior can be customized using various *environment configuration values*.

Call `urls()` once per request, and pass the resulting list of urls to your template, and you're good to go.

### In templates

For *some template languages*, webassets includes extensions which allow you to access the bundles you defined. Further, they usually allow you to define bundles on-the-fly, so you can reference your assets directly from within your templates, rather than predefining them in code.

For example, there is a template tag for *Jinja2*, which allows you do something like this:

```
{% assets filters="cssmin,datauri", output="gen/packed.css", "common/jquery.css",
↪"site/base.css", "site/widgets.css" %}
...
```

### Management command

In some cases you might prefer to cause a manual build of your bundles from the command line. See *Command Line Interface* for more information.

### Command Line Interface

While it's often convenient to have webassets automatically rebuild your bundles on access, you sometimes may prefer to build manually, for example for performance reasons in larger deployments.

*webassets* provides a command line interface which is supposed to help you manage your bundles manually. However, due to the generic nature of the webassets core library, it usually needs some help setting up.

You may want to check the *integration page* to see if webassets already provides helpers to expose the command line within your framework. If that is not the case, read on.

### Build a custom command line client

In most cases, you can simply wrap around the `webassets.script.main` function. For example, the command provided by Flask-Assets looks like this:

```
class ManageAssets(flaskext.script.Command):
    def __init__(self, assets_env):
        self.env = assets_env
```

(continues on next page)

(continued from previous page)

```
    def handle(self, app, prog, name, remaining_args):
        from webassets import script
        script.main(remaining_args, env=self.env)
```

In cases where this isn't possible for some reason, or you need more control, you can work directly with the `webassets.script.CommandLineEnvironment` class, which implements all the commands as simple methods.

```
import logging
from webassets.script import CommandLineEnvironment

# Setup a logger
log = logging.getLogger('webassets')
log.addHandler(logging.StreamHandler())
log.setLevel(logging.DEBUG)

cmdenv = CommandLineEnvironment(assets_env, log)
cmdenv.invoke('build')

# This would also work
cmdenv.build()
```

You are responsible for parsing the command line in any way you see fit (using for example the `optparse` or `argparse` libraries, or whatever your framework provides as a command line utility shell), and then invoking the corresponding methods on your instance of `CommandLineEnvironment`.

### Included Commands

The following describes the commands that will be available to you through the *webassets* CLI interface.

#### build

Builds all bundles, regardless of whether they are detected as having changed or not.

#### watch

Start a daemon which monitors your bundle source files, and automatically rebuilds bundles when a change is detected.

This can be useful during development, if building is not instantaneous, and you are losing valuable time waiting for the build to finish while trying to access your site.

#### clean

Will clear out the cache, which after a while can grow quite large.

### Included Filters

The following filters are included in `webassets`, though some may require the installation of an external library, or the availability of external tools.

You can also write *custom filters*.

### Javascript cross-compilers

**class** webassets.filter.babel.**Babel**(***kwargs*)
Processes ES6+ code into ES5 friendly code using Babel.

Requires the babel executable to be available externally. To install it, you might be able to do:

```
$ npm install --global babel-cli
```

You probably also want some presets:

```
$ npm install --global babel-preset-es2015
```

Example python bundle:

```
es2015 = get_filter('babel', presets='es2015')
bundle = Bundle('**/*.js', filters=es2015)
```

Example YAML bundle:

```
es5-bundle:
    output: dist/es5.js
    config:
        BABEL_PRESETS: es2015
    filters: babel
    contents:
        - file1.js
        - file2.js
```

Supported configuration options:

**BABEL_BIN** The path to the babel binary. If not set the filter will try to run `babel` as if it's in the system path.

**BABEL_PRESETS** Passed straight through to `babel --presets` to specify which babel presets to use

**BABEL_EXTRA_ARGS** A list of manual arguments to be specified to the babel command

**BABEL_RUN_IN_DEBUG** May be set to False to make babel not run in debug

### Javascript compressors

#### rjsmin

**class** webassets.filter.rjsmin.**RJSMin**(***kwargs*)
Minifies Javascript by removing whitespace, comments, etc.

Uses the rJSmin library, which is included with webassets. However, if you have the external package installed, it will be used instead. You may want to do this to get access to the faster C-extension.

Supported configuration options:

**RJSMIN_KEEP_BANG_COMMENTS (boolean)** Keep bang-comments (comments starting with an exclamation mark).

### yui_js

Minify Javascript and CSS with YUI Compressor.

YUI Compressor is an external tool written in Java, which needs to be available. One way to get it is to install the yuicompressor package:

```
pip install yuicompressor
```

No configuration is necessary in this case.

You can also get YUI compressor a different way and define a `YUI_COMPRESSOR_PATH` setting that points to the `.jar` file. Otherwise, an environment variable by the same name is tried. The filter will also look for a `JAVA_HOME` environment variable to run the `.jar` file, or will otherwise assume that `java` is on the system path.

**class** webassets.filter.yui.**YUIJS**(***kwargs*)

### closure_js

Minify Javascript with Google Closure Compiler.

Google Closure Compiler is an external tool written in Java, which needs to be available. One way to get it is to install the closure package:

```
pip install closure
```

No configuration is necessary in this case.

You can also define a `CLOSURE_COMPRESSOR_PATH` setting that points to the `.jar` file. Otherwise, an environment variable by the same name is tried. The filter will also look for a `JAVA_HOME` environment variable to run the `.jar` file, or will otherwise assume that `java` is on the system path.

Supported configuration options:

**CLOSURE_COMPRESSOR_OPTIMIZATION** Corresponds to Google Closure's compilation level parameter.

**CLOSURE_EXTRA_ARGS** A list of further options to be passed to the Closure compiler. There are a lot of them.

For options which take values you want to use two items in the list:

```
['--output_wrapper', 'foo: %output%']
```

### uglifyjs

**class** webassets.filter.uglifyjs.**UglifyJS**(***kwargs*)

Minify Javascript using UglifyJS.

The filter requires version 2 of UglifyJS.

UglifyJS is an external tool written for NodeJS; this filter assumes that the `uglifyjs` executable is in the path. Otherwise, you may define a `UGLIFYJS_BIN` setting.

Additional options may be passed to `uglifyjs` using the setting `UGLIFYJS_EXTRA_ARGS`, which expects a list of strings.

#### jsmin

**class** webassets.filter.jsmin.**JSMin**(***kwargs*)
Minifies Javascript by removing whitespace, comments, etc.

This filter uses a Python port of Douglas Crockford's JSMin, which needs to be installed separately.

There are actually multiple implementations available, for example one by Baruch Even. Easiest to install via PyPI is the one by Dave St. Germain:

```
$ pip install jsmin
```

The filter is tested with this `jsmin` package from PyPI, but will work with any module that exposes a `JavascriptMinify` object with a `minify` method.

If you want to avoid installing another dependency, use the *webassets.filter.rjsmin.RJSMin* filter instead.

#### jspacker

**class** webassets.filter.jspacker.**JSPacker**(***kwargs*)
Reduces the size of Javascript using an inline compression algorithm, i.e. the script will be unpacked on the client side by the browser.

Based on Dean Edwards' jspacker 2, as ported by Florian Schulze.

#### slimit

**class** webassets.filter.slimit.**Slimit**(***kwargs*)
Minifies JS.

Requires the `slimit` package (https://github.com/rspivak/slimit), which is a JavaScript minifier written in Python. It compiles JavaScript into more compact code so that it downloads and runs faster.

It offers mangle and mangle_toplevel options through SLIMIT_MANGLE and SLIMIT_MANGLE_TOPLEVEL

### CSS compressors

#### cssmin

**class** webassets.filter.cssmin.**CSSMin**(***kwargs*)
Minifies CSS.

Requires the `cssmin` package (http://github.com/zacharyvoase/cssmin), which is a port of the YUI CSS compression algorithm.

#### cssutils

**class** webassets.filter.cssutils.**CSSUtils**(***kwargs*)
Minifies CSS by removing whitespace, comments etc., using the Python cssutils library.

Note that since this works as a parser on the syntax level, so invalid CSS input could potentially result in data loss.

#### yui_css

**class** webassets.filter.yui.**YUICSS**(***kwargs*)

#### cleancss

**class** webassets.filter.cleancss.**CleanCSS**(***kwargs*)
Minify css using Clean-css.

Clean-css is an external tool written for NodeJS; this filter assumes that the `cleancss` executable is in the path. Otherwise, you may define a `CLEANCSS_BIN` setting.

Additional options may be passed to `cleancss` binary using the setting `CLEANCSS_EXTRA_ARGS`, which expects a list of strings.

#### slimmer_css

**class** webassets.filter.slimmer.**CSSSlimmer**(***kwargs*)
Minifies CSS by removing whitespace, comments etc., using the Python slimmer library.

#### rcssmin

**class** webassets.filter.rcssmin.**RCSSMin**(***kwargs*)
Minifies CSS.

Requires the `rcssmin` package (https://github.com/ndparker/rcssmin). Alike 'cssmin' it is a port of the YUI CSS compression algorithm but aiming for speed instead of maximum compression.

Supported configuration options: RCSSMIN_KEEP_BANG_COMMENTS (boolean)

Keep bang-comments (comments starting with an exclamation mark).

### JS/CSS compilers

#### postcss

**class** webassets.filter.postcss.**PostCSS**(***kwargs*)
Processes CSS code using PostCSS.

Requires the `postcss` executable to be available externally. To install it, you might be able to do:

```
$ npm install --global postcss
```

You should also install the plugins you want to use:

```
$ npm install --global postcss-cssnext
```

You can configure postcss in `postcss.config.js`:

```
module.exports = {
    plugins: [
        require('postcss-cssnext')({
          // optional configuration for cssnext
```

(continues on next page)

(continued from previous page)

```
        })
    ],
};
```

*Supported configuration options*:

**POSTCSS_BIN** Path to the postcss executable used to compile source files. By default, the filter will attempt to run `postcss` via the system path.

**POSTCSS_EXTRA_ARGS** Additional command-line options to be passed to `postcss` using this setting, which expects a list of strings.

### clevercss

**class** webassets.filter.clevercss.**CleverCSS**(***kwargs*)

Converts CleverCSS markup to real CSS.

If you want to combine it with other CSS filters, make sure this one runs first.

### less

**class** webassets.filter.less.**Less**(***kwargs*)

Converts less markup to real CSS.

This depends on the NodeJS implementation of less, installable via npm. To use the old Ruby-based version (implemented in the 1.x Ruby gem), see *Less*.

*Supported configuration options*:

**LESS_BIN (binary)** Path to the less executable used to compile source files. By default, the filter will attempt to run `lessc` via the system path.

**LESS_LINE_NUMBERS (line_numbers)** Outputs filename and line numbers. Can be either 'comments', which will output the debug info within comments, 'mediaquery' that will output the information within a fake media query which is compatible with the SASSPath to the less executable used to compile source files.

**LESS_RUN_IN_DEBUG (run_in_debug)** By default, the filter will compile in debug mode. Since the less compiler is written in Javascript and capable of running in the browser, you can set this to `False` to have your original less source files served (see below).

**LESS_PATHS (paths)** Add include paths for less command line. It should be a list of paths relatives to Environment.directory or absolute paths. Order matters as less will pick the first file found in path order.

**LESS_AS_OUTPUT (boolean)** By default, this works as an "input filter", meaning `less` is called for each source file in the bundle. This is because the path of the source file is required so that @import directives within the Less file can be correctly resolved.

However, it is possible to use this filter as an "output filter", meaning the source files will first be concatenated, and then the Less filter is applied in one go. This can provide a speedup for bigger projects.

---

**Compiling less in the browser**

less is an interesting case because it is written in Javascript and capable of running in the browser. While for performance reason you should prebuild your stylesheets in production, while developing you may be interested in serving the original less files to the client, and have less compile them in the browser.

To do so, you first need to make sure the less filter is not applied when `Environment.debug` is `True`. You can do so via an option:

```
env.config['less_run_in_debug'] = False
```

Second, in order for the less to identify the less source files as needing to be compiled, they have to be referenced with a `rel="stylesheet/less"` attribute. One way to do this is to use the `Bundle.extra` dictionary, which works well with the template tags that webassets provides for some template languages:

```
less_bundle = Bundle(
    '**/*.less',
    filters='less',
    extra={'rel': 'stylesheet/less' if env.debug else 'stylesheet'}
)
```

Then, for example in a Jinja2 template, you would write:

```
{% assets less_bundle %}
    <link rel="{{ EXTRA.rel }}" type="text/css" href="{{ ASSET_URL }}">
{% endassets %}
```

With this, the `<link>` tag will sport the correct `rel` value both in development and in production.

Finally, you need to include the less compiler:

```
if env.debug:
    js_bundle.contents += 'http://lesscss.googlecode.com/files/less-1.3.0.min.js'
```

---

### less_ruby

**class** webassets.filter.less_ruby.**Less**(***kwargs*)

Converts Less markup to real CSS.

This uses the old Ruby implementation available in the 1.x versions of the less gem. All 2.x versions of the gem are wrappers around the newer NodeJS/Javascript implementation, which you are generally encouraged to use, and which is available in webassets via the *Less* filter.

This filter for the Ruby version is being kept around for backwards-compatibility.

*Supported configuration options*:

**LESS_RUBY_PATH (binary)** Path to the less executable used to compile source files. By default, the filter will attempt to run `lessc` via the system path.

### sass

**class** webassets.filter.sass.**Sass**(***kwargs*)

Converts Sass markup to real CSS.

Requires the Sass executable to be available externally. To install it, you might be able to do:

```
$ sudo gem install sass
```

By default, this works as an "input filter", meaning `sass` is called for each source file in the bundle. This is because the path of the source file is required so that @import directives within the Sass file can be correctly resolved.

However, it is possible to use this filter as an "output filter", meaning the source files will first be concatenated, and then the Sass filter is applied in one go. This can provide a speedup for bigger projects.

To use Sass as an output filter:

```
from webassets.filter import get_filter
sass = get_filter('sass', as_output=True)
Bundle(...., filters=(sass,))
```

However, if you want to use the output filter mode and still also use the @import directive in your Sass files, you will need to pass along the `load_paths` argument, which specifies the path to which the imports are relative to (this is implemented by changing the working directory before calling the `sass` executable):

```
sass = get_filter('sass', as_output=True, load_paths='/tmp')
```

With `as_output=True`, the resulting concatenation of the Sass files is piped to Sass via stdin (`cat ... | sass --stdin ...`) and may cause applications to not compile if import statements are given as relative paths.

For example, if a file `foo/bar/baz.scss` imports file `foo/bar/bat.scss` (same directory) and the import is defined as `@import "bat";` then Sass will fail compiling because Sass has naturally no information on where `baz.scss` is located on disk (since the data was passed via stdin) in order for Sass to resolve the location of `bat.scss`:

```
Traceback (most recent call last):
...
webassets.exceptions.FilterError: sass: subprocess had error: stderr=(sass):1:␣
↪File to import not found or unreadable: bat. (Sass::SyntaxError)
        Load paths:
          /path/to/project-foo
         on line 1 of standard input
  Use --trace for backtrace.
, stdout=, returncode=65
```

To overcome this issue, the full path must be provided in the import statement, `@import "foo/bar/bat"`, then webassets will pass the `load_paths` argument (e.g., `/path/to/project-foo`) to Sass via its `-I` flags so Sass can resolve the full path to the file to be imported: `/path/to/project-foo/foo/bar/bat`

Support configuration options:

**SASS_BIN** The path to the Sass binary. If not set, the filter will try to run `sass` as if it's in the system path.

**SASS_STYLE** The style for the output CSS. Can be one of `expanded` (default), `nested`, `compact` or `compressed`.

**SASS_DEBUG_INFO** If set to `True`, will cause Sass to output debug information to be used by the FireSass Firebug plugin. Corresponds to the `--debug-info` command line option of Sass.

Note that for this, Sass uses `@media` rules, which are not removed by a CSS compressor. You will thus want to make sure that this option is disabled in production.

By default, the value of this option will depend on the environment `DEBUG` setting.

**SASS_LINE_COMMENTS** Passes `--line-comments` flag to sass which emit comments in the generated CSS indicating the corresponding source line.

Note that this option is disabled by Sass if `--style compressed` or `--debug-info` options are provided.

Enabled by default. To disable, set empty environment variable `SASS_LINE_COMMENTS=` or pass `line_comments=False` to this filter.

**SASS_AS_OUTPUT** By default, this works as an "input filter", meaning `sass` is called for each source file in the bundle. This is because the path of the source file is required so that @import directives within the Sass file can be correctly resolved.

However, it is possible to use this filter as an "output filter", meaning the source files will first be concatenated, and then the Sass filter is applied in one go. This can provide a speedup for bigger projects.

It will also allow you to share variables between files.

**SASS_SOURCE_MAP** If provided, this will generate source maps in the output depending on the type specified. By default this will use Sass's `auto`. Possible values are `auto`, `file`, `inline`, or `none`.

**SASS_LOAD_PATHS** It should be a list of paths relatives to Environment.directory or absolute paths. Order matters as sass will pick the first file found in path order. These are fed into the -I flag of the sass command and is used to control where sass imports code from.

**SASS_LIBS** It should be a list of paths relatives to Environment.directory or absolute paths. These are fed into the -r flag of the sass command and is used to require ruby libraries before running sass.

#### scss

**class** webassets.filter.sass.**SCSS**(**a*, ***kw*)
Version of the `sass` filter that uses the SCSS syntax.

#### compass

**class** webassets.filter.compass.**Compass**(***kwargs*)
Converts Compass .sass files to CSS.

Requires at least version 0.10.

To compile a standard Compass project, you only need to have to compile your main `screen.sass`, `print.sass` and `ie.sass` files. All the partials that you include will be handled by Compass.

If you want to combine the filter with other CSS filters, make sure this one runs first.

Supported configuration options:

**COMPASS_BIN** The path to the Compass binary. If not set, the filter will try to run `compass` as if it's in the system path.

**COMPASS_PLUGINS** Compass plugins to use. This is equivalent to the `--require` command line option of the Compass. and expects a Python list object of Ruby libraries to load.

**COMPASS_CONFIG** An optional dictionary of Compass configuration options. The values are emitted as strings, and paths are relative to the Environment's `directory` by default; include a `project_path` entry to override this.

The `sourcemap` option has a caveat. A file called _.css.map is created by Compass in the tempdir (where _.scss is the original asset), which is then moved into the output_path directory. Since the tempdir is created one level down from the output path, the relative links in the sourcemap should correctly map. This file, however, will not be versioned, and thus this option should ideally only be used locally for development and not in production with a caching service as the _.css.map file will not be invalidated.

#### pyscss

**class** webassets.filter.pyscss.**PyScss**(***kwargs*)
Converts Scss markup to real CSS.

This uses PyScss, a native Python implementation of the Scss language. The PyScss module needs to be installed. It's API has been changing; currently, version 1.1.5 is known to be supported.

This is an alternative to using the `sass` or `scss` filters, which are based on the original, external tools.

---

**Note:** The Sass syntax is not supported by PyScss. You need to use the `sass` filter based on the original Ruby implementation instead.

---

*Supported configuration options:*

**PYSCSS_DEBUG_INFO (debug_info)** Include debug information in the output for use with FireSass.

If unset, the default value will depend on your `Environment.debug` setting.

**PYSCSS_LOAD_PATHS (load_paths)** Additional load paths that PyScss should use.

> **Warning:** The filter currently does not automatically use `Environment.load_path` for this.

**PYSCSS_STATIC_ROOT (static_root)** The directory PyScss should look in when searching for include files that you have referenced. Will use `Environment.directory` by default.

**PYSCSS_STATIC_URL (static_url)** The url PyScss should use when generating urls to files in `PYSCSS_STATIC_ROOT`. Will use `Environment.url` by default.

**PYSCSS_ASSETS_ROOT (assets_root)** The directory PyScss should look in when searching for things like images that you have referenced. Will use `PYSCSS_STATIC_ROOT` by default.

**PYSCSS_ASSETS_URL (assets_url)** The url PyScss should use when generating urls to files in `PYSCSS_ASSETS_ROOT`. Will use `PYSCSS_STATIC_URL` by default.

**PYSCSS_STYLE (style)** The style of the output CSS. Can be one of `nested` (default), `compact`, `compressed`, or `expanded`.

### libsass

**class** webassets.filter.libsass.**LibSass**(***kwargs*)

Converts Sass markup to real CSS.

Requires the `libsass` package (https://pypi.python.org/pypi/libsass):

```
pip install libsass
```

libsass is binding to C/C++ implementation of a Sass compiler Libsass

*Configuration options:*

**LIBSASS_STYLE (style)** an optional coding style of the compiled result. choose one of: *nested* (default), *expanded*, *compact*, *compressed*

**LIBSASS_INCLUDES (includes)** an optional list of paths to find @imported SASS/CSS source files

**LIBSASS_AS_OUTPUT** use this filter as an "output filter", meaning the source files will first be concatenated, and then the Sass filter is applied.

See libsass documentation for full documentation about these configuration options:

http://hongminhee.org/libsass-python/sass.html#sass.compile

*Example:*

Define a bundle for `style.scss` that contains `@imports` to files in subfolders:

```
Bundle('style.scss', filters='libsass', output='style.css', depends='**/*.scss')
```

### node-sass

**class** webassets.filter.node_sass.**NodeSass**(***kwargs*)

Converts Scss markup to real CSS.

This uses node-sass which is a wrapper around libsass.

This is an alternative to using the `sass` or `scss` filters, which are based on the original, external tools.

*Supported configuration options:*

**NODE_SASS_DEBUG_INFO (debug_info)** Include debug information in the output

If unset, the default value will depend on your `Environment.debug` setting.

**NODE_SASS_LOAD_PATHS (load_paths)** Additional load paths that node-sass should use.

**NODE_SASS_STYLE (style)** The style of the output CSS. Can be one of `nested` (default), `compact`, `compressed`, or `expanded`.

**NODE_SASS_CLI_ARGS (cli_args)** Additional cli arguments

### node-scss

**class** webassets.filter.node_sass.**NodeSCSS**(**a*, ***kw*)

Version of the `node-sass` filter that uses the SCSS syntax.

### stylus

**class** webassets.filter.stylus.**Stylus**(***kwargs*)

Converts Stylus markup to CSS.

Requires the Stylus executable to be available externally. You can install it using the Node Package Manager:

```
$ npm install -g stylus
```

Supported configuration options:

**STYLUS_BIN** The path to the Stylus binary. If not set, assumes `stylus` is in the system path.

**STYLUS_PLUGINS** A Python list of Stylus plugins to use. Each plugin will be included via Stylus's command-line `--use` argument.

**STYLUS_EXTRA_ARGS** A Python list of any additional command-line arguments.

**STYLUS_EXTRA_PATHS** A Python list of any additional import paths.

### coffeescript

**class** webassets.filter.coffeescript.**CoffeeScript**(***kwargs*)
Converts CoffeeScript to real JavaScript.

If you want to combine it with other JavaScript filters, make sure this one runs first.

Supported configuration options:

**COFFEE_NO_BARE** Set to `True` to compile with the top-level function wrapper (suppresses the –bare option to `coffee`, which is used by default).

### typescript

**class** webassets.filter.typescript.**TypeScript**(***kwargs*)
Compile TypeScript to JavaScript.

TypeScript is an external tool written for NodeJS. This filter assumes that the `tsc` executable is in the path. Otherwise, you may define the `TYPESCRIPT_BIN` setting.

To specify TypeScript compiler options, `TYPESCRIPT_CONFIG` may be defined. E.g.: `--removeComments true --target ES6`.

### requirejs

**class** webassets.filter.requirejs.**RequireJSFilter**(***kwargs*)
Optimizes AMD-style modularized JavaScript into a single asset using RequireJS.

This depends on the NodeJS executable `r.js`; install via npm:

```
$ npm install -g requirejs
```

Details on configuring r.js can be found at http://requirejs.org/docs/optimization.html#basics.

*Supported configuration options*:

executable (env: REQUIREJS_BIN)

Path to the RequireJS executable used to compile source files. By default, the filter will attempt to run `r.js` via the system path.

config (env: REQUIREJS_CONFIG)

The RequireJS options file. The path is taken to be relative to the Environment.directory (by default is /static).

baseUrl (env: REQUIREJS_BASEURL)

The `baseUrl` parameter to r.js; this is the directory that AMD modules will be loaded from. The path is taken relative to the Environment.directory (by default is /static). Typically, this is used in conjunction with a `baseUrl` parameter set in the *config* options file, where the baseUrl value in the config file is used for client-side processing, and the value here is for server-side processing.

optimize (env: REQUIREJS_OPTIMIZE)

The `optimize` parameter to r.js; controls whether or not r.js minifies the output. By default, it is enabled, but can be set to `none` to disable minification. The typical scenario to disable minification is if you do some additional processing of the JavaScript (such as removing `console.log()` lines) before minification by the `rjsmin` filter.

extras (env: REQUIREJS_EXTRAS)

Any other command-line parameters to be passed to r.js. The string is expected to be in unix shell-style format, meaning that quotes can be used to escape spaces, etc.

run_in_debug (env: REQUIREJS_RUN_IN_DEBUG)

Boolean which controls if the AMD requirejs is evaluated client-side or server-side in debug mode. If set to a truthy value (e.g. 'yes'), then server-side compilation is done, even in debug mode. The default is false.

---

**Client-side AMD evaluation**

AMD modules can be loaded client-side without any processing done on the server-side. The advantage to this is that debugging is easier because the browser can tell you which source file is responsible for a particular line of code. The disadvantage is that it means that each loaded AMD module is a separate HTTP request. When running client-side, the client needs access to the *config* – for this reason, when running in client-side mode, the webassets environment must be adjusted to include a reference to this configuration. Typically, this is done by adding something similar to the following during webassets initialization:

```
if env.debug and not env.config.get('requirejs_run_in_debug', True):
    env['requirejs'].contents += ('requirejs-browser-config.js',)
```

And the file `requirejs-browser-config.js` will look something like:

```
require.config({baseUrl: '/static/script/'});
```

Set the *run_in_debug* option to control client-side or server-side compilation in debug.

---

### JavaScript templates

#### jst

**class** webassets.filter.jst.**JST**(***kwargs*)

This filter processes generic JavaScript templates. It will generate JavaScript code that runs all files through a template compiler, and makes the templates available as an object.

It was inspired by Jammit.

For example, if you have a file named `license.jst`:

```
<div class="drivers-license">
  <h2>Name: <%= name %></h2>
  <em>Hometown: <%= birthplace %></em>
</div>
```

Then, after applying this filter, you could use the template in JavaScript:

```
JST.license({name : "Moe", birthplace : "Brooklyn"});
```

The name of each template is derived from the filename. If your JST files are spread over different directories, the path up to the common prefix will be included. For example:

```
Bundle('templates/app1/license.jst', 'templates/app2/profile.jst',
       filters='jst')
```

will make the templates available as `app1/license` and `app2/profile`.

---

**Note:** The filter is "generic" in the sense that it does not actually compile the templates, but wraps them in a JavaScript function call, and can thus be used with any template language. webassets also has filters for specific JavaScript template languages like *DustJS* or *Handlebars*, and those filters precompile the templates on the server, which means a performance boost on the client-side.

---

Unless configured otherwise, the filter will use the same micro-templating language that Jammit uses, which is turn is the same one that is available in underscore.js. The JavaScript code necessary to compile such templates will implicitly be included in the filter output.

*Supported configuration options:*

**JST_COMPILER (template_function)** A string that is inserted into the generated JavaScript code in place of the function to be called that should do the compiling. Unless you specify a custom function here, the filter will include the JavaScript code of it's own micro-templating language, which is the one used by underscore.js and Jammit.

If you assign a custom function, it is your responsibility to ensure that it is available in your final JavaScript.

If this option is set to `False`, then the template strings will be output directly, which is to say, `JST.foo` will be a string holding the raw source of the `foo` template.

**JST_NAMESPACE (namespace)** How the templates should be made available in JavaScript. Defaults to `window.JST`, which gives you a global `JST` object.

**JST_BARE (bare)** Whether everything generated by this filter should be wrapped inside an anonymous function. Default to `False`.

---

**Note:** If you enable this option, the namespace must be a property of the `window` object, or you won't be able to access the templates.

---

**JST_DIR_SEPARATOR (separator)** The separator character to use for templates within directories. Defaults to '/'

### handlebars

**class** webassets.filter.handlebars.**Handlebars**(***kwargs*)

Compile Handlebars templates.

This filter assumes that the `handlebars` executable is in the path. Otherwise, you may define a `HANDLEBARS_BIN` setting.

---

**Note:** Use this filter if you want to precompile Handlebars templates. If compiling them in the browser is acceptable, you may use the JST filter, which needs no external dependency.

---

**Warning:** Currently, this filter is not compatible with input filters. Any filters that would run during the input-stage will simply be ignored. Input filters tend to be other compiler-style filters, so this is unlikely to be an issue.

#### dustjs

**class** webassets.filter.dust.**DustJS**(***kwargs*)

DustJS templates compilation filter.

Takes a directory full `.dust` files and creates a single Javascript object that registers to the `dust` global when loaded in the browser:

```
Bundle('js/templates/', filters='dustjs')
```

Note that in the above example, a directory is given as the bundle contents, which is unusual, but required by this filter.

This uses the `dusty` compiler, which is a separate project from the DustJS implementation. To install `dusty` together with LinkedIn's version of `dustjs` (the original does not support NodeJS > 0.4):

```
npm install dusty
rm -rf node_modules/dusty/node_modules/dust
git clone https://github.com/linkedin/dustjs node_modules/dust
```

---

**Note:** To generate the DustJS client-side Javascript, you can then do:

```
cd node_modules/dust
make dust
cp dist/dist-core...js your/static/assets/path
```

---

For compilation, set the `DUSTY_PATH=.../node_modules/dusty/bin/dusty`. Optionally, set `NODE_PATH=.../node`.

### Other

#### cssrewrite

**class** webassets.filter.cssrewrite.**CSSRewrite**(*replace=False*)

Source filter that rewrites relative urls in CSS files.

CSS allows you to specify urls relative to the location of the CSS file. However, you may want to store your compressed assets in a different place than source files, or merge source files from different locations. This would then break these relative CSS references, since the base URL changed.

This filter transparently rewrites CSS `url()` instructions in the source files to make them relative to the location of the output path. It works as a *source filter*, i.e. it is applied individually to each source file before they are merged.

No configuration is necessary.

The filter also supports a manual mode:

```
get_filter('cssrewrite', replace={'old_directory':'/custom/path/'})
```

This will rewrite all urls that point to files within `old_directory` to use `/custom/path` as a prefix instead.

You may plug in your own replace function:

```
get_filter('cssrewrite', replace=lambda url: re.sub(r'^/?images/', '/images/',␣
↪url))
get_filter('cssrewrite', replace=lambda url: '/images/'+url[7:] if url.startswith(
↪'images/') else url)
```

### datauri

**class** webassets.filter.datauri.**CSSDataUri**(***kwargs*)

Will replace CSS url() references to external files with internal data: URIs.

The external file is now included inside your CSS, which minimizes HTTP requests.

---

**Note:** Data Uris have clear disadvantages, so put some thought into if and how you would like to use them. Have a look at some performance measurements.

---

The filter respects a `DATAURI_MAX_SIZE` option, which is the maximum size (in bytes) of external files to include. The default limit is what I think should be a reasonably conservative number, 2048 bytes.

### cssprefixer

**class** webassets.filter.cssprefixer.**CSSPrefixer**(***kwargs*)

Uses CSSPrefixer to add vendor prefixes to CSS files.

### autoprefixer

**class** webassets.filter.autoprefixer.**AutoprefixerFilter**(***kwargs*)

Prefixes vendor-prefixes using *autoprefixer <https://github.com/ai/autoprefixer>*, which uses the *Can I Use? <http://www.caniuse.com>* database to know which prefixes need to be inserted.

This depends on the *autoprefixer <https://github.com/ai/autoprefixer>* command line tool being installed (use `npm install autoprefixer`).

*Supported configuration options*:

**AUTOPREFIXER_BIN** Path to the autoprefixer executable used to compile source files. By default, the filter will attempt to run `autoprefixer` via the system path.

**AUTOPREFIXER_BROWSERS** The browser expressions to use. This corresponds to the `--browsers <value>` flag, see the *–browsers documentation <https://github.com/ai/autoprefixer#browsers>*. By default, this flag won't be passed, and autoprefixer's default will be used.

Example:

```
AUTOPREFIXER_BROWSERS = ['> 1%', 'last 2 versions', 'firefox 24', 'opera 12.1
↪']
```

**AUTOPREFIXER_EXTRA_ARGS** Additional options may be passed to `autoprefixer` using this setting, which expects a list of strings.

#### jinja2

**class** webassets.filter.jinja2.**Jinja2**(***kwargs*)

Process a file through the Jinja2 templating engine.

Requires the `jinja2` package (https://github.com/mitsuhiko/jinja2).

The Jinja2 context can be specified with the *JINJA2_CONTEXT* configuration option or directly with *context={...}*. Example:

```
Bundle('input.css', filters=Jinja2(context={'foo': 'bar'}))
```

Additionally to enable template loading mechanics from your project you can provide *JINJA2_ENV* or *jinja2_env* arg to make use of already created environment.

#### spritemapper

**class** webassets.filter.spritemapper.**Spritemapper**(***kwargs*)

Generate CSS spritemaps using Spritemapper, a Python utility that merges multiple images into one and generates CSS positioning for the corresponding slices. Installation is easy:

```
pip install spritemapper
```

Supported configuration options:

**SPRITEMAPPER_PADDING** A tuple of integers indicating the number of pixels of padding to place between sprites

**SPRITEMAPPER_ANNEAL_STEPS** Affects the number of combinations to be attempted by the box packer algorithm

**Note:** Since the `spritemapper` command-line utility expects source and output files to be on the filesystem, this filter interfaces directly with library internals instead. It has been tested to work with Spritemapper version 1.0.

### Creating custom filters

Creating custom filters can be easy, or very easy.

Before we get to that though, it is first necessary to understand that there are two types of filters: *input filters* and *output filters*. Output filters are applied after the complete content after all a bundle's contents have been merged together. Input filters, on the other hand, are applied to each source file after it is read from the disk. In the case of nested bundles, input filters will be passed down, with the input filters of a parent bundle are applied before the output filter of a child bundle:

```
child_bundle = Bundle('file.css', filters='yui_css')
Bundle(child_bundle, filters='cssrewrite')
```

In this example, because cssrewrite acts as an input filter, what will essentially happen is:

```
yui_css(cssrewrite(file.css))
```

To be even more specific, since a single filter can act as both an input and an output filter, the call chain will actually look something like this:

```
cssrewrite.output(yui_css.output((cssrewrite.input((yui_css.input(file.css)))))
```

The usual reason to use an input filter is that the filter's transformation depends on the source file's filename. For example, the *cssrewrite* filter needs to know the location of the source file relative to the final output file, so it can properly update relative references. Another example are CSS converters like *less*, which work relative to the input filename.

With that in mind...

### The very easy way

In the simplest case, a filter is simply a function that takes two arguments, an input stream and an output stream.

```
def noop(_in, out, **kw):
    out.write(_in.read())
```

That's it! You can use this filter when defining your bundles:

```
bundle = Bundle('input.js', filters=(noop,))
```

If you are using Jinja2, you can also specify the callable inline, provided that it is available in the context:

```
{% assets filters=(noop, 'jsmin') ... %}
```

It even works when using Django templates, although here, you are of course more limited in terms of syntax; if you want to use multiple filters, you need to combine them:

```
{% assets filters=my_filters ... %}
```

Just make sure that the context variable `my_filters` is set to your function.

Note that you currently cannot write input filters in this way. Callables always act as output filters.

### The easy way

This works by subclassing `webassets.filter.Filter`. In doing so, you need to write a bit more code, but you'll be able to enjoy a few perks.

The `noop` filter from the previous example, written as a class, would look something like this:

```
from webassets.filter import Filter

class NoopFilter(Filter):
    name = 'noop'

    def output(self, _in, out, **kwargs):
        out.write(_in.read())

    def input(self, _in, out, **kwargs):
        out.write(_in.read())
```

The `output` and `input` methods should look familiar. They're basically like the callable you are already familiar with, simply pulled inside a class.

Class-based filters have a `name` attribute, which you need to set if you want to register your filter globally.

The `input` method will be called for every source file, the `output` method will be applied once after a bundle's contents have been concatenated.

Among the `kwargs` you currently receive are:

- `source_path` (only for `input()`): The filename behind the `in` stream, though note that other input filters may already have transformed it.
- `output_path`: The final output path that your filters work will ultimatily end up in.

**Note:** Always make your filters accept arbitrary `**kwargs`. The API does allow for additional values to be passed along in the future.

### Registering

The `name` wouldn't make much sense, if it couldn't be used to reference the filter. First, you need to register the class with the system though:

```
from webassets.filter import register_filter
register_filter(NoopFilter)
```

Or if you are using yaml then use the filters key for the environment:

```
directory: .
url: /
debug: True
updater: timestamp
filters:
    - my_custom_package.my_filter
```

After that, you can use the filter like you would any of the built-in ones:

```
{% assets filters='jsmin,noop' ... %}
```

### Options

Class-based filters are used as instances, and as such, you can easily define a `__init__` method that takes arguments. However, you should make all parameters optional, if possible, or your filter will not be usable through a name reference.

There might be another thing to consider. If a filter is specified multiple times, which sometimes can happen unsuspectingly when bundles are nested within each other, it will only be applied a single time. By default, all filters of the same class are considered *the same*. In almost all cases, this will be just fine.

However, in case you want your filter to be applicable multiple times with different options, you can implement the `unique` method and return a hashable object that represents data unique to this instance:

```
class FooFilter(Filter):
    def __init__(self, *args, **kwargs):
        self.args, self.kwargs = args, kwargs
    def unique(self):
        return self.args, self.kwargs
```

This will cause two instances of this filter to be both applied, as long as the arguments given differ. Two instances with the exact same arguments will still be considered equal.

If you want each of your filter's instances to be unique, you can simply do:

```
def unique(self):
    return id(self)
```

### Useful helpers

The `Filter` base class provides some useful features.

#### setup()

It's quite common that filters have dependencies - on other Python libraries, external tools, etc. If you want to provide your filter regardless of whether such dependencies are matched, and fail only if the filter is actually used, implement a `setup()` method on your filter class:

```
class FooFilter(Filter):
    def setup(self):
        import foolib
        self.foolib = foolib

    def apply(self, _in, out):
        self.foolib.convert(...)
```

#### options

Some filters will need to be configured. This can of course be done by passing arguments into `__init__` as explained above, but it restricts you to configuring your filters in code, and can be tedious if necessary every single time the filter is used.

In some cases, it makes more sense to have an option configured globally, like the path to an external binary. A number of the built-in filters do this, allowing you to both specify a config variable in the webassets `Environment` instance, or as an OS environment variable.

```
class FooFilter(Filter):
    options = {
        'binary': 'FOO_BIN'
    }
```

If you define a an `options` attribute on your filter class, these options will automatically be supported both by your filter's __init__, as well as via a configuration or environment variable. In the example above, you may pass `binary` when creating a filter instance manually, or define `FOO_BIN` in `Environment.config`, or as an OS environment variable.

#### get_config()

In cases where the declarative approach of the `options` attribute is not enough, you can implement custom options yourself using the `Filter.get_config()` helper:

```
class FooFilter(Filter):
    def setup(self):
        self.bin = self.get_config('BINARY_PATH')
```

This will check first the configuration, then the environment for `BINARY_PATH`, and raise an exception if nothing is found.

`get_config()` allows you to specify different names for the setting and the environment variable:

```
self.get_config(setting='ASSETS_BINARY_PATH', env='BINARY_PATH')
```

It also supports disabling either of the two, causing only the other to be checked for the given name:

```
self.get_config(setting='ASSETS_BINARY_PATH', env=False)
```

Finally, you can easily make a value optional using the `require` parameter. Instead of raising an exception, `get_config()` then returns `None`. For example:

```
self.java = self.get_config('JAVA_BIN', require=False) or 'java'
```

### Abstract base classes

In some cases, you might want to have a common base class for multiple filters. You can make the base class abstract by setting `name` to `None` explicitly. However, this is currently only relevant for the built-in filters, since your own filters will not be registered automatically in any case.

### More?

You can have a look inside the `webassets.filter` module source code to see a large number of example filters.

Assets can be filtered through one or multiple filters, modifying their contents (think minification, compression).

### CSS compilers

CSS compilers intend to improve upon the default CSS syntax, allow you to write your stylesheets in a syntax more powerful, or more easily readable. Since browsers do not understand this new syntax, the CSS compiler needs to translate its own syntax to original CSS.

`webassets` includes *builtin filters for a number of popular CSS compilers*, which you can use like any other filter. There is one problem though: While developing, you will probably want to disable asset packaging, and instead work with the uncompressed assets (i.e., you would disable the *environment.debug* option). However, you still need to apply the filter for your CSS compiler, since otherwise, the Browser wouldn't understand your stylesheets.

For this reason, such compiler filters run even when in debug mode:

```
less = Bundle('css/base.less', 'css/forms.less',
              filters='less,cssmin', output='screen.css')
```

The above code block behaves exactly like you would want it to: When debugging, the less files are compiled to CSS, but the code is not minified. In production, both filters are applied.

Sometimes, you need to merge together good old CSS code, and you have a compiler that, unlike `less`, cannot process those. Then you can use a child bundle:

```
sass = Bundle('*.sass', filters='sass', output='gen/sass.css')
all_css = Bundle('css/jquery.calendar.css', sass,
                 filters='cssmin', output="gen/all.css")
```

In the above case, the `sass` filter is only applied to the Sass source files, within a nested bundle (which needs it's own output target!). The minification is applied to all CSS content in the outer bundle.

### Loaders

Using these helper classes, you can define your bundles or even your complete environment in some external data source, rather than constructing them in code.

**class** webassets.loaders.**YAMLLoader**(*file_or_filename*)

Will load an environment or a set of bundles from YAML files.

**load_bundles**(*environment=None*)

Load a list of `Bundle` instances defined in the YAML file.

Expects the following format:

```
bundle-name:
    filters: sass,cssutils
    output: cache/default.css
    contents:
        - css/jquery.ui.calendar.css
        - css/jquery.ui.slider.css
another-bundle:
    # ...
```

Bundles may reference each other:

```
js-all:
    contents:
        - jquery.js
        - jquery-ui    # This is a bundle reference
jquery-ui:
    contents: jqueryui/*.js
```

If an `environment` argument is given, it's bundles may be referenced as well. Note that you may pass any compatibly dict-like object.

Finally, you may also use nesting:

```
js-all:
    contents:
        - jquery.js
        # This is a nested bundle
        - contents: "*.coffee"
          filters: coffeescript
```

**load_environment**()

Load an `Environment` instance defined in the YAML file.

Expects the following format:

```
directory: ../static
url: /media
```

(continues on next page)

(continued from previous page)

```
debug: True
updater: timestamp
filters:
    - my_custom_package.my_filter
config:
    compass_bin: /opt/compass
    another_custom_config_value: foo

bundles:
    # ...
```

All values, including `directory` and `url` are optional. The syntax for defining bundles is the same as for *load_bundles()*.

Sample usage:

```
from webassets.loaders import YAMLLoader
loader = YAMLLoader('asset.yml')
env = loader.load_environment()

env['some-bundle'].urls()
```

**class** webassets.loaders.**PythonLoader**(*module_name*)

Basically just a simple helper to import a Python file and retrieve the bundles defined there.

**load_bundles**()

Load `Bundle` objects defined in the Python module.

Collects all bundles in the global namespace.

**load_environment**()

Load an `Environment` defined in the Python module.

Expects as default a global name `environment` to be defined, or overridden by passing a string `module:environment` to the constructor.

### Integration with other libraries

While the *webassets* core is designed to work with any WSGI application, also included are some additional utilities for some popular frameworks and libraries.

### Jinja2

A Jinja2 extension is available as `webassets.ext.jinja2.AssetsExtension`. It will provide a `{% assets %}` tag which allows you to reference your bundles from within a template to render its urls.

It also allows you to create bundles on-the-fly, thus making it possible to define your assets entirely within your templates.

If you are using Jinja2 inside of Django, see this page.

### Setting up the extension

```
from jinja2 import Environment as Jinja2Environment
from webassets import Environment as AssetsEnvironment
from webassets.ext.jinja2 import AssetsExtension

assets_env = AssetsEnvironment('./static/media', '/media')
jinja2_env = Jinja2Environment(extensions=[AssetsExtension])
jinja2_env.assets_environment = assets_env
```

After adding the extension to your Jinja 2 environment, you need to make sure that it knows about your `webassets.Environment` instance. This is done by setting the `assets_environment` attribute.

### Using the tag

To output a bundle that has been registered with the environment, simply pass its name to the tag:

```
{% assets "all_js", "ie_js" %}
    <script type="text/javascript" src="{{ ASSET_URL }}"></script>
{% endassets %}
```

The tag will repeatedly output its content for each `ASSET_URL` of each bundle. In the above case, that might be the output urls of the *all_js* and *ie_js* bundles, or, in debug mode, urls referencing the source files of both bundles.

If you pass something to the tag that isn't a known bundle name, it will be considered a filename. This allows you to define a bundle entirely within your templates:

```
{% assets filters="cssmin,datauri", output="gen/packed.css", "common/jquery.css",
↪"site/base.css", "site/widgets.css" %}
...
```

Of course, this means you can combine the two approaches as well. The following code snippet will merge together the given bundle and the contents of the `jquery.js` file that was explicitly passed:

```
{% assets output="gen/packed.js", "common/jquery.js", "my-bundle" %}
...
```

### Custom resolvers

The resolver is a pluggable object that webassets uses to find the contents of a `Bundle` on the filesystem, as well as to generate the correct urls to these files.

For example, the default resolver searches the *`Environment.load_path`*, or looks within *`Environment.directory`*. The webassets Django integration will use Django's *staticfile finders* to look for files.

For normal usage, you will not need to write your own resolver, or indeed need to know how they work. However, if you want to integrate `webassets` with another framework, or if your application is complex enough that it requires custom file referencing, read on.

### The API as webassets sees it

`webassets` expects to find the resolver via the `Environment.resolver` property, and expects this object to provide the following methods:

Resolver.**resolve_source**(*ctx*, *item*)
Given item from a Bundle's contents, this has to return the final value to use, usually an absolute filesystem path.

---

**Note:** It is also allowed to return urls and bundle instances (or generally anything else the calling Bundle instance may be able to handle). Indeed this is the reason why the name of this method does not imply a return type.

---

The incoming item is usually a relative path, but may also be an absolute path, or a url. These you will commonly want to return unmodified.

This method is also allowed to resolve item to multiple values, in which case a list should be returned. This is commonly used if item includes glob instructions (wildcards).

---

**Note:** Instead of this, subclasses should consider implementing *search_for_source()* instead.

---

Resolver.**resolve_output_to_path**(*ctx*, *target*, *bundle*)
Given target, this has to return the absolute filesystem path to which the output file of bundle should be written.

target may be a relative or absolute path, and is usually taking from the Bundle.output property.

If a version-placeholder is used (%(version)s, it is still unresolved at this point.

Resolver.**resolve_source_to_url**(*ctx*, *filepath*, *item*)
Given the absolute filesystem path in filepath, as well as the original value from Bundle.contents which resolved to this path, this must return the absolute url through which the file is to be referenced.

Depending on the use case, either the filepath or the item argument will be more helpful in generating the url.

This method should raise a ValueError if the url cannot be determined.

Resolver.**resolve_output_to_url**(*ctx*, *target*)
Given target, this has to return the url through which the output file can be referenced.

target may be a relative or absolute path, and is usually taking from the Bundle.output property.

This is different from *resolve_source_to_url()* in that you do not passed along the result of *resolve_output_to_path()*. This is because in many use cases, the filesystem is not available at the point where the output url is needed (the media server may on a different machine).

### Methods to overwrite

However, in practice, you will usually want to override the builtin Resolver, and customize it's behaviour where necessary. The default resolver already splits what is is doing into multiple methods; so that you can either override them, or refer to them in your own implementation, as makes sense.

Instead of the official entrypoints above, you may instead prefer to override the following methods of the default resolver class:

Resolver.**search_for_source**(*ctx*, *item*)
Called by *resolve_source()* after determining that item is a relative filesystem path.

You should always overwrite this method, and let *resolve_source()* deal with absolute paths, urls and other types of items that a bundle may contain.

`Resolver.`**`search_load_path`**(*ctx*, *item*)
This is called by *`search_for_source()`* when a *`Environment.load_path`* is set.

If you want to change how the load path is processed, overwrite this method.

### Helpers to use

The following methods of the default resolver class you may find useful as helpers while implementing your subclass:

`Resolver.`**`consider_single_directory`**(*directory*, *item*)
Searches for `item` within `directory`. Is able to resolve glob instructions.

Subclasses can call this when they have narrowed done the location of a bundle item to a single directory.

`Resolver.`**`glob`**(*basedir*, *expr*)
Evaluates a glob expression. Yields a sorted list of absolute filenames.

`Resolver.`**`query_url_mapping`**(*ctx*, *filepath*)
Searches the environment-wide url mapping (based on the urls assigned to each directory in the load path). Returns the correct url for `filepath`.

Subclasses should be sure that they really want to call this method, instead of simply falling back to `super()`.

### Example: A prefix resolver

The following is a simple resolver implementation that searches for files in a different directory depending on the first directory part.

```
from webassets.env import Resolver

class PrefixResolver(Resolver):

    def __init__(self, prefixmap):
        self.map = prefixmap

    def search_for_source(self, ctx, item):
        parts = item.split('/', 1)
        if len(parts) < 2:
            raise ValueError(
                '"%s" not valid; a static path requires a prefix.' % item)

        prefix, name = parts
        if not prefix in self.map:
            raise ValueError(('Prefix "%s" of static path "%s" is not '
                              'registered') % (prefix, item))

        # For the rest, defer to base class method, which provides
        # support for things like globbing.
        return self.consider_single_directory(self.map[prefix], name)
```

Using it:

```
env = webassets.Environment(path, url)
env.resolver = PrefixResolver({
    'app1': '/var/www/app1/static',
    'app2': '/srv/deploy/media/app2',
```

(continues on next page)

(continued from previous page)

```
})
bundle = Bundle(
    'app2/scripts/jquery.js',
    'app1/*.js',
)
```

### Other implementations

- django-assets Resolver (search for `class DjangoResolver`).
- Flask-Assets Resolver (search for `class FlaskResolver`).
- pyramid_webassets Resolver (search for `class PyramidResolver`).

## FAQ

### Is there a cache-busting feature?

Yes! See *URL Expiry (cache busting)*.

### Relative URLs in my CSS code break if the merged asset is written to a different location than the source files. How do I fix this?

Use the builtin *cssrewrite* filter which will transparently fix `url()` instructions in CSS files on the fly.

### I am using a CSS compiler and I need its filter to apply even in debug mode!

See *CSS compilers* for how this is best done.

### Is Google App Engine supported?

Yes. Due to the way Google App Engine works (static files are stored on separate servers), you need to build your assets locally, possibly using one of the management commands provided for your preferred framework, and then deploy them.

In production mode, you need to disable the `Environment.auto_build` setting.

For URL expiry functionality, you need to use a manifest that holds version information. See *URL Expiry (cache busting)*.

There is a barebone Google App Engine example in the examples/appengine/ folder.

# CHAPTER 3

# Detailed documentation

This documentation also includes some pages which are applicable regardless of the framework used:

## 3.1 URL Expiry (cache busting)

### 3.1.1 For beginners

You are using `webassets` because you care about the performance of your site. For the same reason, you have configured your web server to send out your media files with a so called *far future expires* header: Your web server sets the `Expires` header to some date many years in the future. Your user's browser will never spend any time trying to retrieve an updated version.

---

**Note:** Of course, the user's browser will already use the `Etag` and `Last-Modified/If-Modified-Since` to avoid downloading content it has already cached, and if your web server isn't misconfigured entirely, this will work. The point of *far future expires* is to get rid of **even** those requests which would return only a `304 Not Modified` response.

---

What if you actually deploy an update to your site? Now you need to convince the browser to download new versions of your assets after all, but you have just told it not to bother to check for new versions. You work around this by *modifying the URL with which the asset is included*. There are two distinct ways to so:

1) Append a version identifier as a querystring:

```
http://www.example.org/media/print.css?acefe50
```

2) Add a version identifier to the actual filename:

```
http://www.example.org/media/print.acefe50.css
```

How webassets helps you do this is explained in the sections below.

**Note:** Even if you are not using *far future expires* headers, you might still find `webassets` expiry features useful to navigate around any funny browser caching behaviour that might require a `Shift`-reload.

### 3.1.2 What is the version of a file

To expire an URL, it is modified with a version identifier. What is this identifier? By default, `webassets` will create an MD5-hash of the file contents, and use the first few characters as the file version. `webassets` also allows you to use the *last modified* timestamp of the file. You can configure this via the `versions` option:

```
env = Environment(...)
env.versions = 'hash'          # the default
env.versions = 'hash:32'       # use the full md5 hash
env.versions = 'timestamp'     # use the last modified timestamp
```

It is generally recommended that you use a hash as the version, since it will remain the same as long as the content does not change, regardless of any filesystem metadata, which can change for any number of reasons.

### 3.1.3 Expire using a querystring

`webassets` will automatically add the version as a querystring to the urls it generates, by virtue of the `url_expire` option defaulting to `True`. If you want to be explicit:

```
env = Environment(...)
env.url_expire = True
```

There is nothing else you need to do here. The URLs that are generated might look like this:

```
/media/print.css?acefe50
```

However, while the default, expiring with a querystring is not be the best option:

### 3.1.4 Expire using the filename

Adding the version as a querystring has two problems. First, it may not always be a browser that implements caching through which we need to bust. It is said that certain (possibly older) proxies do ignore the querystring with respect to their caching behavior.

Second, in certain more complex deployment scenarios, where you have multiple frontend and/or multiple backend servers, an upgrade is anything but instantaneous. You need to be able to serve both the old and the new version of your assets at the same time. See for example how this affects you when using Google App Engine.

To expire using the filename, you add a `%(version)s` placeholder to your bundle output target:

```
bundle = Bundle(..., output='screen.%(version)s.css')
```

The URLs that are generated might look like this:

```
/media/screen.acefe50.css
```

**Note:** `webassets` will use this modified filename for the actual output files it writes to disk, as opposed to just modifying the URL it generates. You do not have to configure your web server to do any rewriting.

### 3.1.5 About manifests

**Note:** This is mostly an advanced feature, and you might not have to bother with it at all.

`webassets` supports Environment-wide *manifests*. A manifest remembers the current version of every bundle. What is this good for?

1) Speed. Calculating a hash can be expensive. Even if you are using timestamp-based versions, that still means a stat-request to your disk.

   **Note:** Note that even without a manifest, `webassets` will cache the version in memory. It will only need to be calculated once per process. However, if you have *many* bundles, and a very busy site, a manifest will allow you to both skip calculating the version (e.g. creating a hash), as well as read the versions of all bundles into memory at once.

   **Note:** If you are using automatic building, all of this is mostly not true. In order to determine whether a rebuild is required, `webassets` will need to check the timestamps of all files involved in any case. It goes without saying that using automatic building on a production site is a convenience feature for small sites, and at odds with counting paper clips in the form of filesystem `stat` calls.

2) Making it possible to know the version in the first place.

   Depending on your configuration and deployment, consider that it might not actually be possible for `webassets` to know what the version is.

   If you are using a hash-based version, and your bundle's output target has a placeholder, there is no way to know what the version is, *unless* is has been written to a manifest during the build process.

   The timestamp-based versioning mechanism can actually look at the source files to determine the version. But, in more complex deployments, the source files might not actually be available to read - they might be on a completely different server altogether.

   A manifest allows version information to be persisted.

In practice, by default the version information will be written to the cache. You can explicitly request this behaviour be setting the `manifest` option:

```
env = Environment(...)
env.manifest = 'cache'
```

In a simple setup, where you are separately building on your local machine during development, and building on the web server for production (maybe via the automatic building feature, enabled by default), this is exactly would you want. Don't worry about it.

There is a specific deployment scenario where you want to prebuild your bundles locally, and for either of the two reasons above want to include the version data pre-made when you deploy your app to the web server. In such a case, it is not helpful to have the versions stored in the cache. Instead, `webassets` provides a manifest type that writes all information to a single file:

```
env = Environment(...)
env.manifest = 'file'
env.manifest = 'file:/tmp/manifest.to-be-deployed'  # explicit filename
```

You can then just copy this one file to the web server, and `webassets` will know all about the versions without having to consult the media files.

**Note:** The file is a pickled dict.

## 3.2 Upgrading

When upgrading from an older version, you might encounter some backwards incompatibility. The `webassets` API is not stable yet.

### 3.2.1 In 0.10

- The `Resolver` API has changed. Rather than being bound to an environment via the constructor, the individual methods now receive a ‘‘ctx‘ object, which allows access to the environment's settings.

  See the page on implementing resolvers.

- The `Bundle.build()` and `Bundle.url()` methods no longer accept an environment argument. To work with a Bundle that is not attached to an environment already, use the following syntax instead:

```
with bundle.bind(env):
    bundle.build()
```

- Filters can no longer access a `self.env` attribute. It has been renamed to `self.ctx`, which provides a compatible object.

### 3.2.2 In 0.9

- Python 2.5 is no longer supported.
- The API of the BaseCache.get() method has changed. It no longer receives a `python` keyword argument. This only affects you if you have implemented a custom cache class.

### 3.2.3 In 0.8

- **django-assets is no longer included!** You need to install it's package separately. See the current development version.

**Warning:** When upgrading, you need to take extra care to rid yourself of the old version of webassets before installing the separate `django-assets` package. This is to avoid that Python still finds the old `django_assets` module that used to be included with `webassets`.

In some cases, even `pip uninstall webassets` is not enough, and old `*.pyc` files are kept around. I recommend that you delete your old webassets install manually from the filesystem. To find out where it is stored, open a Python shell and do:

```
>>> import webassets
>>> webassets
<module 'webassets' from '/usr/local/lib/python2.7/dist-packages/webassets/src/
↪webassets/__init__.pyc'>
```

- Some filters now run in debug mode. Specifically, there are two things that deserve mention:
  - `cssrewrite` now runs when `debug="merge"`. This is always what is wanted; it was essentially a bug that this didn't happen before.
  - All kinds of compiler-style filters (Sass, less, Coffeescript, JST templates etc). all now run in debug mode. The presence of such a filter causes bundles to be merged even while `debug=True`.

    In practice, if you've been using custom bundle `debug` values to get such compilers to run, this will continue to work. Though it can now be simplified. Code like this:

    ```
    Bundle(
        Bundle('*.coffee', filters='coffeescript', debug=False)
        filters='jsmin')
    ```

    can be replaced with:

    ```
    Bundle('*.coffee', filters='coffeescript,jsmin')
    ```

    which has the same effect, which is that during debugging, Coffeescript will be compiled, but not minimized. This also allows you to define bundles that use compilers from within the templates tags, because nesting is no longer necessary.

    However, if you need to combine Coffeescript files (or other files needing compiling) with regular CSS or JS files, nesting is still required:

    ```
    Bundle('*.js'
           Bundle('*.coffee', filters='coffeescript'),
           filters='jsmin')
    ```

    If for some reason you do not want these compilers to run, you may still use a manual `debug` value to override the behavior. A case where this is useful is the `less` filter, which can be compiled in the browser:

    ```
    Bundle('*.less', filters='less', debug=True)
    ```

    Here, as long as the environment is in debug mode, the bundle will output the source urls, despite the `less` filter normally forcing a merge.

  As part of this new feature, the handling of nested bundle debug values has changed such that in rare cases you may see a different outcome. In the unlikely case that you are using these a lot, the rule is simple: The debug level can only ever be decreased. Child bundles cannot cannot do "more debugging" than their parent, and if `Environment.debug=False`, all bundle debug values are effectively ignored.

- The internal class names of filters have been renamed. For example, `JSMinFilter` is now simply `JSMin`. This only affects you if you reference these classes directly, rather than using their id (such as `jsmin`), which should be rare.
- Removed the previously deprecated `rebuild` alias for the `build` command.
- Subtly changed how the `auto_build` setting affects the `Bundle.build()` method: It doesn't anymore. Instead, the setting now only works on the level of `Bundle.urls()`. The new behaviour is more consistent, makes more sense, and simplifies the code.

  The main backwards-incompatiblity caused by this is that when `environment.auto_build=False`, and you are calling `bundle.build()` without specifying an explicit `force` argument, it used to be the case that `force=True` was assumed, i.e. the bundle was built without looking at the timestamps to see if a rebuild is necessary. Now, the timestamps will be checked, unless `force=True` is explicitly given.

  In case you don't want to pass `force=True`, you can instead also set the `Environment.updater` property to `False`; without an updater to check timestamps, every `build()` call will act as if `force=True`.

**Note**: This only affects you if you work with the `Bundle.build()` and `Bundle.url()` methods directly. The behavior of the command line interface, or the template tags is not affected.

- The implementation of the `CommandLineEnvironment` has changed, and each command is now a separate class. If you have been subclassing `CommandLineEnvironment` to override individual command methods like `CommandLineEnvironment.build()`, you need to update your code.
- The `JavaMixin` helper class to implement Java-based filters has been removed, and in it's stead there is now a `JavaTool` base class that can be used.
- The code to resolve bundle contents has been refactored. As a result, the behavior of the semi-internal method `Bundle.resolve_contents()` has changed slightly; in addition, the `Environment._normalize_source_path()` method used mainly by extensions like `Flask-Assets` has been removed. Instead, extensions now need to implement a custom `Resolver`. The `Environment.absurl` method has also disappeared, and replacing it can now be done via a custom `Resolver`` class.
- `Environment.directory` now always returns an absolute path; if a relative path is stored, it is based off on the current working directory. This spares *a lot* of calls to `os.abspath` throughout the code. If you need the original value you can always use `environment.config['directory']`.
- If the `JST_COMPILER` option of the `jst` filter is set to `False` (as opposed to the default value, `None`), the templates will now be output as raw strings. Before, `False` behaved like `None` and used the builtin compiler.
- The API of the `concat()` filter method has changed. Instead of a list of hunks, it is now given a list of 2-tuples of `(hunk, info_dict)`.
- The internal `JSTTemplateFilter` base class has changed API. - concat filter - jst handlebar filters have changed, use concat, base class has changed

### 3.2.4 In 0.7

There are some significant backwards incompatible changes in this release.

- The `Environment.updater` property (corresponds to the `ASSETS_UPDATER` setting) can no longer be set to `False` or `"never"` in order to disable the automatic rebuilding. Instead, this now needs to be done using `Environment.auto_build`, or the corresponding `ASSETS_AUTO_BUILD` setting.
- The `Environment.expire` (`ASSETS_EXPIRE`) option as been renamed to `Environment.url_expire` (`ASSETS_URL_EXPIRE`), and the default value is now `True`.
- To disable automatic building, set the new `Environment.auto_build` (`ASSETS_AUTO_BUILD`) option to `False`. Before, this was done via the `Environment.updater`, which is now deprecated.

Other changes:

- If `Environment.auto_build` is disabled, the API of Bundle.build() now assumes a default value of `True` for the `force` argument. This should not cause any problems, since it is the only call signature that really makes sense in this case.
- The former `less` filter, based on the old Ruby version of lessCSS (still available as the 1.x Ruby gems, but no longer developed) has been renamed `less_ruby`, and `less` now uses the new NodeJS/Javascript implementation, which a while ago superseded the Ruby one.
- The `rebuild` command (of the command line mode) has been renamed to `build`.
- The command line interface now requires the external dependency `argparse` on Python versions 2.6 and before. `argparse` is included with Python starting with version 2.7.
- `PythonLoader.load_bundles()` now returns a dict with the bundle names as keys, rather than a list.

- Filters now receive new keyword arguments. The API now officially requires filters to accept arbitrary `**kwargs` for compatibility with future versions. While the documentation has always suggested `**kwargs` be used, not all builtin filters followed this rule. Your custom filters may need updating as well.
- Filter classes now longer get an auto-generated name. If you have a custom filter and have not explicitly given it a name, you need to do this now if you want to register the filter globally.
- `django_assets` no longer tries to load a global `assets.py` module (it will still find bundles defined in application-level `assets.py` files). If you want to define bundles in other modules, you now need to list those explicitly in the ASSETS_MODULES setting.

### 3.2.5 In 0.6

- The `Environment.updater` class no longer support custom callables. Instead, you need to subclass `BaseUpdater`. Nobody is likely to use this feature though.
- The cache is no longer debug-mode only. If you enable `Environment.cache` (`ASSETS_CACHE` in `django-assets`), the cache will be enabled regardless of the `Environment.debug`/`ASSETS_DEBUG` option. If you want the old behavior, you can easily configure it manually.
- The `Bundle.build` method no longer takes the `no_filters` argument. This was always intended for internal use and its existence not advertised, so its removal shouldn't cause too many problems.
- The `Bundle.build` method now returns a list of `FileHunk` objects, rather than a single one. It now works for container bundles (bundles which only have other bundles for children, not files), rather than raising an exception.
- The `rebuild` command now ignores a `debug=False` setting, and forces a build in production mode instead.

### 3.2.6 In 0.4

- Within `django_assets`. the semantics of the `debug` setting have changed again. It once again allows you to specifically enable debug mode for the assets handling, irrespective of Django's own `DEBUG` setting.
- `RegistryError` is now `RegisterError`.
- The `ASSETS_AUTO_CREATE` option no longer exists. Instead, automatic creation of bundle output files is now bound to the `ASSETS_UPDATER` setting. If it is `False`, i.e. automatic updating is disabled, then assets won't be automatically created either.

### 3.2.7 In 0.2

- The filter API has changed. Rather than defining an `apply` method and optionally an `is_source_filter` attribute, those now have been replaced by `input()` and `output()` methods. As a result, a single filter can now act as both an input and an output filter.

### 3.2.8 In 0.1

- The semantics of the `ASSETS_DEBUG` setting have changed. In 0.1, setting this to `True` meant *enable the django-assets debugging mode*. However, `django-assets` now follows the default Django `DEBUG` setting, and `ASSETS_DEBUG` should be understood as meaning *how to behave when in debug mode*. See ASSETS_DEBUG for more information.

- `ASSETS_AUTO_CREATE` now causes an error to be thrown if due it it being disabled a file cannot be created. Previously, it caused the source files to be linked directly (as if debug mode were active).

  This was done due to `Explicit is better than implicit`, and for security considerations; people might trusting their comments to be removed. If it turns out to be necessary, the functionality to fall back to source could be added again in a future version through a separate setting.

- The YUI Javascript filter can no longer be referenced via `yui`. Instead, you need to explicitly specify which filter you want to use, `yui_js` or `yui_css`.

# Python Module Index

## W

# Index

## S



## T



## U



## V



## W



## Y