

---

# **Pipeline Documentation**

*Release 1.6.9*

**Timothée Peignier**

February 17, 2017



<b>1</b>	<b>Table Of Contents</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Configuration . . . . .	4
1.3	Usage . . . . .	9
1.4	Compressors . . . . .	11
1.5	Compilers . . . . .	15
1.6	Javascript Templates . . . . .	18
1.7	Storages . . . . .	20
1.8	Signals . . . . .	22
1.9	Sites using Pipeline . . . . .	23
<b>2</b>	<b>Indices and tables</b>	<b>25</b>



Pipeline is an asset packaging library for Django, providing both CSS and JavaScript concatenation and compression, built-in JavaScript template support, and optional data-URI image and font embedding.

You can report bugs and discuss features on the [issues page](#).

You can discuss features or ask questions on the IRC channel on freenode : #django-pipeline



---

## Table Of Contents

---

### Installation

1. Either check out Pipeline from [GitHub](#) or to pull a release off [PyPI](#)

```
pip install django-pipeline
```

2. Add 'pipeline' to your INSTALLED\_APPS

```
INSTALLED_APPS = (  
    'pipeline',  
)
```

3. Use a pipeline storage for STATICFILES\_STORAGE

```
STATICFILES_STORAGE = 'pipeline.storage.PipelineCachedStorage'
```

4. Add the PipelineFinder to STATICFILES\_FINDERS

```
STATICFILES_FINDERS = (  
    'django.contrib.staticfiles.finders.FileSystemFinder',  
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',  
    'pipeline.finders.PipelineFinder',  
)
```

---

**Note:** You need to use Django>=1.7 to be able to use this version of pipeline.

---

### Upgrading from 1.3

To upgrade from pipeline 1.3, you will need to follow these steps:

1. Update templates to use the new syntax

```
{# pipeline<1.4 #}  
{% load compressed %}  
{% compressed_js 'group' %}  
{% compressed_css 'group' %}
```

```
{# pipeline>=1.4 #}  
{% load pipeline %}
```

```
{% javascript 'group' %}
{% stylesheet 'group' %}
```

### 2. Add the PipelineFinder to STATICFILES\_FINDERS

```
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
    'pipeline.finders.PipelineFinder',
)
```

## Upgrading from 1.5

To upgrade from pipeline 1.5, you will need update all your PIPELINE\_\* settings and move them under the new PIPELINE setting. See *Configuration*.

## Recommendations

Pipeline's default CSS and JS compressor is Yuglify. Yuglify wraps UglifyJS and cssmin, applying the default YUI configurations to them. It can be downloaded from: <https://github.com/yui/yuglify/>.

If you do not install yuglify, make sure to disable the compressor in your settings.

## Configuration

Configuration and list of available settings for Pipeline. Pipeline settings are namespaced in a PIPELINE dictionary in your project settings, e.g.:

```
PIPELINE = {
    'PIPELINE_ENABLED': True,
    'JAVASCRIPT': {
        'stats': {
            'source_filenames': (
                'js/jquery.js',
                'js/d3.js',
                'js/collections/*.js',
                'js/application.js',
            ),
            'output_filename': 'js/stats.js',
        }
    }
}
```

## Specifying files

You specify groups of files to be compressed in your settings. You can use glob syntax to select multiples files.

The basic syntax for specifying CSS/JavaScript groups files is

```
PIPELINE = {
    'STYLESHEETS': {
        'colors': {
            'source_filenames': (
```



```

        'css/core.css',
        'css/colors/*.css',
        'css/layers.css'
    ),
    'output_filename': 'css/colors.css',
    'extra_context': {
        'media': 'screen,projection',
    },
},
},
},
'JAVASCRIPT': {
    'stats': {
        'source_filenames': (
            'js/jquery.js',
            'js/d3.js',
            'js/collections/*.js',
            'js/application.js',
        ),
        'output_filename': 'js/stats.js',
    }
}
}
}

```

## Group options

**source\_filenames**

### Required

Is a tuple with the source files to be compressed. The files are concatenated in the order specified in the tuple.

**output\_filename**

### Required

Is the filename of the (to be) compressed file.

**variant**

### Optional

Is the variant you want to apply to your CSS. This allow you to embed images and fonts in CSS with data-URI. Allowed values are : None and datauri.

Defaults to None.

**template\_name**

### Optional

Name of the template used to render <script> for js package or <link> for css package.

Defaults to None.

`extra_context`

### Optional

Is a dictionary of values to add to the template context, when generating the HTML for the HTML-tags with the `templatetags`.

For CSS, if you do not specify `extra_context/media`, the default media in the `<link>` output will be `media="all"`.

For JS, the default templates support the `async` and `defer` tag attributes which are controlled via `extra_context`:

```
'extra_context': {
    'async': True,
},
```

`manifest`

### Optional

Indicate if you want this group to appear in your cache manifest.

Defaults to `True`.

### Other settings

`PIPELINE_ENABLED`

True if assets should be compressed, False if not.

Defaults to `not settings.DEBUG`.

`PIPELINE_COLLECTOR_ENABLED`

True if assets should be collected in develop, False if not.

Defaults to `True`

---

**Note:** this only work when `PIPELINE_ENABLED` is False.

---

`SHOW_ERRORS_INLINE`

True if errors compiling CSS/JavaScript files should be shown inline at the top of the browser window, or False if they should trigger exceptions (the older behavior).

This only applies when compiling through the `{% stylesheet %}` or `{% javascript %}` template tags. It won't impact `collectstatic`.

Defaults to `settings.DEBUG`.

#### CSS\_COMPRESSOR

Compressor class to be applied to CSS files.

If empty or None, CSS files won't be compressed.

Defaults to `'pipeline.compressors.yuglify.YuglifyCompressor'`.

#### JS\_COMPRESSOR

Compressor class to be applied to JavaScript files.

If empty or None, JavaScript files won't be compressed.

Defaults to `'pipeline.compressors.yuglify.YuglifyCompressor'`

---

**Note:** Please note that in order to use Yuglify compressor, you need to install Yuglify (see [Installation](#) for more details).

---

#### TEMPLATE\_NAMESPACE

Object name where all of your compiled templates will be added, from within your browser. To access them with your own JavaScript namespace, change it to the object of your choice.

Defaults to `"window.JST"`

#### TEMPLATE\_EXT

The extension for which Pipeline will consider the file as a Javascript template. To use a different extension, like `.mustache`, set this settings to `.mustache`.

Defaults to `".jst"`

#### TEMPLATE\_FUNC

JavaScript function that compiles your JavaScript templates. Pipeline doesn't bundle a javascript template library, but the default setting is to use the `underscore` template function.

Defaults to `"_.template"`

#### TEMPLATE\_SEPARATOR

Character chain used by Pipeline as replacement for directory separator.

Defaults to `"_"`

#### MIMETYPES

Tuple that match file extension with their corresponding mimetypes.

Defaults to

```
(
  (b'text/coffeescript', '.coffee'),
  (b'text/less', '.less'),
  (b'text/javascript', '.js'),
  (b'text/x-sass', '.sass'),
  (b'text/x-scss', '.scss')
)
```

**Warning:** If you support Internet Explorer version 8 and below, you should declare javascript files as text/javascript.

## Embedding fonts and images

You can embed fonts and images directly in your compiled css, using Data-URI in modern browsers.

To do so, setup variant group options to the method you wish to use :

```
'STYLESHEETS' = {
  'master': {
    'source_filenames': (
      'css/core.css',
      'css/button/*.css',
    ),
    'output_filename': 'css/master.css',
    'variant': 'datauri',
  },
}
```

Images and fonts are embedded following these rules :

- If asset is under **32 kilobytes** to avoid rendering delay or not rendering at all in Internet Explorer 8.
- If asset path contains a directory named “**embed**”.

## Overriding embedding settings

You can override these rules using the following settings:

### **EMBED\_MAX\_IMAGE\_SIZE**

Setting that controls the maximum image size (in bytes) to embed in CSS using Data-URIs. Internet Explorer 8 has issues with assets over 32 kilobytes.

Defaults to 32700

### **EMBED\_PATH**

Setting the directory that an asset needs to be in so that it is embedded

Defaults to r' [/?embed/'

## Rewriting CSS urls

If the source CSS contains relative URLs (i.e. relative to current file), those URLs will be converted to full relative path.

## Wrapped javascript output

All javascript output is wrapped in an anonymous function :

```
(function(){
  //JS output...
})();
```

This safety wrapper, make it difficult to pollute the global namespace by accident and improve performance.

You can override this behavior by setting `DISABLE_WRAPPER` to `True`. If you want to use your own wrapper, change the `JS_WRAPPER` setting. For example:

```
JS_WRAPPER = "(function(){stuff();%s})();"
```

## Usage

Describes how to use Pipeline when it is installed and configured.

## Templatetags

Pipeline includes two template tags: `stylesheet` and `javascript`, in a template library called `pipeline`.

They are used to output the `<link>` and `<script>`-tags for the specified CSS/JavaScript-groups (as specified in the settings). The first argument must be the name of the CSS/JavaScript group.

When `settings.DEBUG` is set to `True` the use of these template tags will result in a separate tag for each resource in a given group (i.e., the combined, compressed files will not be used), in order to make local debugging easy. When `settings.DEBUG` is set to `False` the opposite is true. You can override the default behavior by setting `settings.PIPELINE['PIPELINE_ENABLED']` manually. When set to `True` or `False` this enables or disables, respectively, the usage of the combined, compressed file for each resource group. This can be useful, if you encounter errors in your compressed code that don't occur in your uncompressed code and you want to debug them locally.

If you need to change the output of the HTML-tags generated from the templatetags, this can be done by overriding the templates `pipeline/css.html` and `pipeline/js.html`.

## Example

If you have specified the CSS-groups “colors” and “stats” and a JavaScript-group with the name “scripts”, you would use the following code to output them all

```
{% load pipeline %}
{% stylesheet 'colors' %}
{% stylesheet 'stats' %}
{% javascript 'scripts' %}
```

### Collect static

Pipeline integrates with staticfiles, you just need to setup `STATICFILES_STORAGE` to

```
STATICFILES_STORAGE = 'pipeline.storage.PipelineStorage'
```

Then when you run `collectstatic` command, your CSS and your javascripts will be compressed in the same time

```
$ python manage.py collectstatic
```

### Cache-busting

Pipeline 1.2+ no longer provides its own cache-busting URL support (using e.g. the `PIPELINE_VERSIONING` setting) but uses Django's built-in staticfiles support for this. To set up cache-busting in conjunction with `collectstatic` as above, use

```
STATICFILES_STORAGE = 'pipeline.storage.PipelineCachedStorage'
```

This will handle cache-busting just as staticfiles's built-in `CachedStaticFilesStorage` does.

### Middleware

To enable HTML compression add `pipeline.middleware.MinifyHTMLMiddleware`, to your `MIDDLEWARE_CLASSES` settings.

Ensure that it comes after any middleware which modifies your HTML, like `GZipMiddleware`

```
MIDDLEWARE_CLASSES = (  
    'django.middleware.gzip.GZipMiddleware',  
    'pipeline.middleware.MinifyHTMLMiddleware',  
)
```

### Cache manifest

Pipeline provide a way to add your javascripts and stylesheets files to a cache-manifest via [Manifesto](#).

To do so, you just need to add manifesto app to your `INSTALLED_APPS`.

### Jinja

Pipeline also includes Jinja2 support and is used almost identically to the Django Template tags implementation. You just need to pass `pipeline.jinja2.PipelineExtension` to your Jinja2 environment:

```
{  
    'BACKEND': 'django.template.backends.jinja2.Jinja2',  
    'DIRS': [],  
    'APP_DIRS': True,  
    'OPTIONS': {  
        'environment': 'myproject.jinja2.environment',  
        'extensions': ['pipeline.jinja2.PipelineExtension']  
    }  
}
```

## Templates

Unlike the Django template tag implementation the Jinja2 implementation uses different templates, so if you wish to override them please override `pipeline/css.jinja` and `pipeline/js.jinja`.

## Compressors

### Yuglify compressor

The Yuglify compressor uses `yuglify` for compressing javascript and stylesheets.

To use it for your stylesheets add this to your `PIPELINE['CSS_COMPRESSOR']`

```
PIPELINE['CSS_COMPRESSOR'] = 'pipeline.compressors.yuglify.YuglifyCompressor'
```

To use it for your javascripts add this to your `PIPELINE['JS_COMPRESSOR']`

```
PIPELINE['JS_COMPRESSOR'] = 'pipeline.compressors.yuglify.YuglifyCompressor'
```

#### `YUGLIFY_BINARY`

Command line to execute for the Yuglify program. You will most likely change this to the location of `yuglify` on your system.

Defaults to `'/usr/bin/env yuglify'`.

#### `YUGLIFY_CSS_ARGUMENTS`

Additional arguments to use when compressing CSS.

Defaults to `'--terminal'`.

#### `YUGLIFY_JS_ARGUMENTS`

Additional arguments to use when compressing JavaScript.

Defaults to `'--terminal'`.

### YUI Compressor compressor

The YUI compressor uses `yui-compressor` for compressing javascript and stylesheets.

To use it for your stylesheets add this to your `PIPELINE['CSS_COMPRESSOR']`

```
PIPELINE['CSS_COMPRESSOR'] = 'pipeline.compressors.yui.YUICompressor'
```

To use it for your javascripts add this to your `PIPELINE['JS_COMPRESSOR']`

```
PIPELINE['JS_COMPRESSOR'] = 'pipeline.compressors.yui.YUICompressor'
```

### YUI\_BINARY

Command line to execute for the YUI program. You will most likely change this to the location of yui-compressor on your system.

Defaults to `' /usr/bin/env yuicompressor'`.

**Warning:** Don't point to `yuicompressor.jar` directly, we expect to find a executable script.

### YUI\_CSS\_ARGUMENTS

Additional arguments to use when compressing CSS.

Defaults to `''`.

### YUI\_JS\_ARGUMENTS

Additional arguments to use when compressing JavaScript.

Defaults to `''`.

## Closure Compiler compressor

The Closure compressor uses [Google Closure Compiler](#) to compress javascripts.

To use it add this to your PIPELINE [ ' JS\_COMPRESSOR' ]

```
PIPELINE['JS_COMPRESSOR'] = 'pipeline.compressors.closure.ClosureCompressor'
```

### CLOSURE\_BINARY

Command line to execute for the Closure Compiler program. You will most likely change this to the location of closure on your system.

Default to `' /usr/bin/env closure'`

**Warning:** Don't point to `compiler.jar` directly, we expect to find a executable script.

### CLOSURE\_ARGUMENTS

Additional arguments to use when closure is called.

Default to `''`

## UglifyJS compressor

The UglifyJS compressor uses [UglifyJS](#) to compress javascripts.

To use it add this to your PIPELINE [ ' JS\_COMPRESSOR' ]



```
PIPELINE['JS_COMPRESSOR'] = 'pipeline.compressors.uglifyjs.UglifyJSCompressor'
```

### UGLIFYJS\_BINARY

Command line to execute for the UglifyJS program. You will most likely change this to the location of uglifyjs on your system.

Defaults to `'/usr/bin/env uglifyjs'`.

### UGLIFYJS\_ARGUMENTS

Additional arguments to use when uglifyjs is called.

Default to `''`

## JSMin compressor

The jsmin compressor uses Douglas Crockford jsmin tool to compress javascripts.

To use it add this to your `PIPELINE['JS_COMPRESSOR']`

```
PIPELINE['JS_COMPRESSOR'] = 'pipeline.compressors.jsmin.JSMinCompressor'
```

Install the jsmin library with your favorite Python package manager

```
pip install jsmin
```

## SlimIt compressor

The slimit compressor uses SlimIt to compress javascripts.

To use it add this to your `PIPELINE['JS_COMPRESSOR']`

```
PIPELINE['JS_COMPRESSOR'] = 'pipeline.compressors.slimit.SlimItCompressor'
```

Install the slimit library with your favorite Python package manager

```
pip install slimit
```

## CSSTidy compressor

The CSSTidy compressor uses CSSTidy to compress stylesheets.

To use it for your stylesheets add this to your `PIPELINE['CSS_COMPRESSOR']`

```
PIPELINE['CSS_COMPRESSOR'] = 'pipeline.compressors.csstidy.CSS_TidyCompressor'
```

### CSSTIDY\_BINARY

Command line to execute for csstidy program. You will most likely change this to the location of csstidy on your system.

Defaults to `'/usr/bin/env csstidy'`

### CSSTIDY\_ARGUMENTS

Additional arguments to use when csstidy is called.

Default to `'--template=highest'`

### CSSMin compressor

The cssmin compressor uses the `cssmin` command to compress stylesheets. To use it, add this to your `PIPELINE['CSS_COMPRESSOR']`

```
PIPELINE['CSS_COMPRESSOR'] = 'pipeline.compressors.cssmin.CSSMinCompressor'
```

### CSSMIN\_BINARY

Command line to execute for cssmin program. You will most likely change this to the location of cssmin on your system.

Defaults to  `'/usr/bin/env cssmin'`

### CSSMIN\_ARGUMENTS

Additional arguments to use when cssmin is called.

Default to `''`

### No-Op Compressors

The No-Op compressor don't perform any operation, when used, only concatenation occurs. This is useful for debugging faulty concatenation due to poorly written javascript and other errors.

To use it, add this to your settings

```
PIPELINE['CSS_COMPRESSOR'] = 'pipeline.compressors.NoopCompressor'  
PIPELINE['JS_COMPRESSOR'] = 'pipeline.compressors.NoopCompressor'
```

### Write your own compressor class

You can write your own compressor class, for example if you want to implement other types of compressors.

To do so, you just have to create a class that inherits from `pipeline.compressors.CompressorBase` and implements `compress_css` and/or a `compress_js` when needed.

Finally, add it to `PIPELINE['CSS_COMPRESSOR']` or `PIPELINE['JS_COMPRESSOR']` settings (see [Configuration](#) for more information).

### Example

A custom compressor for an imaginary compressor called jam

```

from pipeline.compressors import CompressorBase

class JamCompressor(CompressorBase):
    def compress_js(self, js):
        return jam.compress(js)

    def compress_css(self, css):
        return jam.compress(css)

```

Add it to your settings

```

PIPELINE['CSS_COMPRESSOR'] = 'jam.compressors.JamCompressor'
PIPELINE['JS_COMPRESSOR'] = 'jam.compressors.JamCompressor'

```

## Compilers

### Coffee Script compiler

The Coffee Script compiler uses [Coffee Script](#) to compile your javascript.

To use it add this to your PIPELINE['COMPILERS']

```

PIPELINE['COMPILERS'] = (
    'pipeline.compilers.coffee.CoffeeScriptCompiler',
)

```

#### COFFEE\_SCRIPT\_BINARY

Command line to execute for coffee program. You will most likely change this to the location of coffee on your system.

Defaults to '/usr/bin/env coffee'.

#### COFFEE\_SCRIPT\_ARGUMENTS

Additional arguments to use when coffee is called.

Defaults to ''.

### Live Script compiler

The LiveScript compiler uses [LiveScript](#) to compile your javascript.

To use it add this to your PIPELINE['COMPILERS']

```

PIPELINE['COMPILERS'] = (
    'pipeline.compilers.livescript.LiveScriptCompiler',
)

```

### **LIVE\_SCRIPT\_BINARY**

Command line to execute for LiveScript program. You will most likely change this to the location of lsc on your system.

Defaults to `'/usr/bin/env lsc'`.

### **LIVE\_SCRIPT\_ARGUMENTS**

Additional arguments to use when lsc is called.

Defaults to `' '`.

## **LESS compiler**

The LESS compiler uses **LESS** to compile your stylesheets.

To use it add this to your PIPELINE [ 'COMPILERS' ]

```
PIPELINE['COMPILERS'] = (  
  'pipeline.compilers.less.LessCompiler',  
)
```

### **LESS\_BINARY**

Command line to execute for lessc program. You will most likely change this to the location of lessc on your system.

Defaults to `'/usr/bin/env lessc'`.

### **LESS\_ARGUMENTS**

Additional arguments to use when lessc is called.

Defaults to `' '`.

## **SASS compiler**

The SASS compiler uses **SASS** to compile your stylesheets.

To use it add this to your PIPELINE [ 'COMPILERS' ]

```
PIPELINE['COMPILERS'] = (  
  'pipeline.compilers.sass.SASSCompiler',  
)
```

### **SASS\_BINARY**

Command line to execute for sass program. You will most likely change this to the location of sass on your system.

Defaults to `'/usr/bin/env sass'`.

### **SASS\_ARGUMENTS**

Additional arguments to use when sass is called.

Defaults to ' '.

## **Stylus compiler**

The Stylus compiler uses [Stylus](#) to compile your stylesheets.

To use it add this to your PIPELINE [ 'COMPILERS' ]

```
PIPELINE['COMPILERS'] = (  
  'pipeline.compilers.stylus.StylusCompiler',  
)
```

### **STYLUS\_BINARY**

Command line to execute for stylus program. You will most likely change this to the location of stylus on your system.

Defaults to '/usr/bin/env stylus'.

### **STYLUS\_ARGUMENTS**

Additional arguments to use when stylus is called.

Defaults to ' '.

## **ES6 compiler**

The ES6 compiler uses [Babel](#) to convert ES6+ code into vanilla ES5.

Note that for files to be transpiled properly they must have the file extension **.es6**

To use it add this to your PIPELINE [ 'COMPILERS' ]

```
PIPELINE['COMPILERS'] = (  
  'pipeline.compilers.es6.ES6Compiler',  
)
```

### **BABEL\_BINARY**

Command line to execute for babel program. You will most likely change this to the location of babel on your system.

Defaults to '/usr/bin/env babel'.

### **BABEL\_ARGUMENTS**

Additional arguments to use when babel is called.

Defaults to ' '.

## Write your own compiler class

You can write your own compiler class, for example if you want to implement other types of compilers.

To do so, you just have to create a class that inherits from `pipeline.compilers.CompilerBase` and implements `match_file` and `compile_file` when needed.

Finally, specify it in the tuple of compilers `PIPELINE['COMPILERS']` in the settings.

### Example

A custom compiler for an imaginary compiler called jam

```
from pipeline.compilers import CompilerBase

class JamCompiler(CompilerBase):
    output_extension = 'js'

    def match_file(self, filename):
        return filename.endswith('.jam')

    def compile_file(self, infile, outfile, outdated=False, force=False):
        if not outdated and not force:
            return # No need to recompiled file
        return jam.compile(infile, outfile)
```

## 3rd Party Compilers

Here is an (in)complete list of 3rd party compilers that integrate with django-pipeline

### Compass (requires RubyGem)

**Creator** [Mila Labs](#)

**Description** Compass compiler for django-pipeline using the original Ruby gem.

**Link** <https://github.com/mila-labs/django-pipeline-compass-rubygem>

### Compass (standalone)

**Creator** [Vitaly Babiy](#)

**Description** django-pipeline-compass is a compiler for [django-pipeline](#). Making it really easy to use scss and compass with out requiring the compass gem.

**Link** <https://github.com/vbabiy/django-pipeline-compass>

## Javascript Templates

Pipeline allows you to use javascript templates along with your javascript views. To use your javascript templates, just add them to your `JAVASCRIPT` group

```
PIPELINE['JAVASCRIPT'] = {
  'application': {
    'source_filenames': (
      'js/application.js',
      'js/templates/**/*.jst',
    ),
    'output_filename': 'js/application.js'
  }
}
```

For example, if you have the following template `js/templates/photo/detail.jst`

```
<div class="photo">
  
  <div class="caption">
    <%= caption %>
  </div>
</div>
```

It will be available from your javascript code via `window.JST`

```
JST.photo_detail({ src:"images/baby-panda.jpg", caption:"A baby panda is born" });
```

## Configuration

### Template function

By default, Pipeline uses a variant of [Micro Templating](#) to compile the templates, but you can choose your preferred JavaScript templating engine by changing `PIPELINE['TEMPLATE_FUNC']`

```
PIPELINE['TEMPLATE_FUNC'] = 'template'
```

### Template namespace

Your templates are made available in a top-level object, by default `window.JST`, but you can choose your own via `PIPELINE['TEMPLATE_NAMESPACE']`

```
PIPELINE['TEMPLATE_NAMESPACE'] = 'window.Template'
```

### Template extension

Templates are detected by their extension, by default `.jst`, but you can use your own extension via `PIPELINE['TEMPLATE_EXT']`

```
PIPELINE['TEMPLATE_EXT'] = '.mustache'
```

### Template separator

Templates identifier are built using a replacement for directory separator, by default `_`, but you specify your own separator via `PIPELINE['TEMPLATE_SEPARATOR']`

```
PIPELINE['TEMPLATE_SEPARATOR'] = '/'
```

## Using it with your favorite template library

### Mustache

To use it with [Mustache](#) you will need some extra javascript

```
Mustache.template = function(templateString) {
  return function() {
    if (arguments.length < 1) {
      return templateString;
    } else {
      return Mustache.to_html(templateString, arguments[0], arguments[1]);
    }
  };
};
```

And use these settings

```
PIPELINE['TEMPLATE_EXT'] = '.mustache'
PIPELINE['TEMPLATE_FUNC'] = 'Mustache.template'
```

### Handlebars

To use it with [Handlebars](#), use the following settings

```
PIPELINE['TEMPLATE_EXT'] = '.handlebars'
PIPELINE['TEMPLATE_FUNC'] = 'Handlebars.compile'
PIPELINE['TEMPLATE_NAMESPACE'] = 'Handlebars.templates'
```

### Ember.js + Handlebars

To use it with [Ember.js](#), use the following settings

```
PIPELINE['TEMPLATE_EXT'] = '.handlebars'
PIPELINE['TEMPLATE_FUNC'] = 'Ember.Handlebars.compile'
PIPELINE['TEMPLATE_NAMESPACE'] = 'window.Ember.TEMPLATES'
PIPELINE['TEMPLATE_SEPARATOR'] = '/'
```

### Prototype

To use it with [Prototype](#), just setup your PIPELINE['TEMPLATE\_FUNC']

```
PIPELINE['TEMPLATE_FUNC'] = 'new Template'
```

## Storages

### Using with staticfiles

Pipeline is providing a storage for [staticfiles](#) app, to use it configure STATICFILES\_STORAGE like so

```
STATICFILES_STORAGE = 'pipeline.storage.PipelineStorage'
```



And if you want versioning use

```
STATICFILES_STORAGE = 'pipeline.storage.PipelineCachedStorage'
```

There is also non-packing storage available, that allows you to run `collectstatic` command without packaging your assets. Useful for production when you don't want to run compressor or compilers

```
STATICFILES_STORAGE = 'pipeline.storage.NonPackagingPipelineStorage'
```

Also available if you want versioning

```
STATICFILES_STORAGE = 'pipeline.storage.NonPackagingPipelineCachedStorage'
```

If you use staticfiles with `DEBUG = False` (i.e. for integration tests with Selenium) you should install the finder that allows staticfiles to locate your outputted assets :

```
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
    'pipeline.finders.PipelineFinder',
)
```

If you use `PipelineCachedStorage` you may also like the `CachedFileFinder`, which allows you to use integration tests with cached file URLs.

If you want to exclude Pipelunable content from your collected static files, you can also use Pipeline's `FileSystemFinder` and `AppDirectoriesFinder`. These finders will also exclude *unwanted* content like READMEs, tests and examples, which are particularly useful if you're collecting content from a tool like Bower.

```
STATICFILES_FINDERS = (
    'pipeline.finders.FileSystemFinder',
    'pipeline.finders.AppDirectoriesFinder',
    'pipeline.finders.CachedFileFinder',
    'pipeline.finders.PipelineFinder',
)
```

## GZIP compression

Pipeline can also creates a gzipped version of your collected static files, so that you can avoid compressing them on the fly.

```
STATICFILES_STORAGE = 'your.app.GZIPCachedStorage'
```

The storage need to inherit from `GZIPMixin`:

```
from django.contrib.staticfiles.storage import CachedStaticFilesStorage

from pipeline.storage import GZIPMixin

class GZIPCachedStorage(GZIPMixin, CachedStaticFilesStorage):
    pass
```

## Using with other storages

You can also use your own custom storage, for example, if you want to use S3 for your assets :

```
STATICFILES_STORAGE = 'your.app.S3PipelineManifestStorage'
```

Your storage only needs to inherit from `PipelineMixin` and `ManifestFilesMixin` or `CachedFilesMixin`.

In Django 1.7+ you should use `ManifestFilesMixin` unless you don't have access to the local filesystem in which case you should use `CachedFilesMixin`.

```
from django.contrib.staticfiles.storage import CachedFilesMixin, ManifestFilesMixin
from pipeline.storage import PipelineMixin
from storages.backends.s3boto import S3BotoStorage

class S3PipelineManifestStorage(PipelineMixin, ManifestFilesMixin, S3BotoStorage):
    pass

class S3PipelineCachedStorage(PipelineMixin, CachedFilesMixin, S3BotoStorage):
    pass
```

## Using Pipeline with Bower

`Bower` is a *package manager for the web* that allows you to easily include frontend components with named versions. Integrating Bower with Pipeline is straightforward.

Add your Bower directory to your `STATICFILES_DIRS` :

```
STATICFILES_DIRS = (
    os.path.join(os.path.dirname(__file__), '..', 'bower_components'),
)
```

Then process the relevant content through Pipeline :

```
PIPELINE['JAVASCRIPT'] = {
    'components': {
        'source_filenames': (
            'jquery/jquery.js',
            # you can choose to be specific to reduce your payload
            'jquery-ui/ui/*.js',
        ),
        'output_filename': 'js/components.js',
    },
}
```

`pipeline.finders.FileSystemFinder` will help you by excluding much of the extra content that Bower includes with its components, such as READMEs, tests and examples, while still including images, fonts, CSS fragments etc.

## Signals

List of all signals sent by pipeline.

### css\_compressed

`pipeline.signals.css_compressed`

Whenever a css package is compressed, this signal is sent after the compression.

Arguments sent with this signal :

**sender** The `Packager` class that compressed the group.

**package** The package actually compressed.

## js\_compressed

### pipeline.signals.js\_compressed

Whenever a js package is compressed, this signal is sent after the compression.

Arguments sent with this signal :

**sender** The `Packager` class that compressed the group.

**package** The package actually compressed.

## Sites using Pipeline

The following sites are a partial list of people using Pipeline.

Are you using pipeline and not being in this list? Drop us a line.

### 20 Minutes

For their internal tools: <http://www.20minutes.fr>

### Borsala

Borsala is the social investment plaform. You can follow stock markets that are traded in Turkey: <http://borsala.com>

### Croisé dans le Métro

For their main and mobile website:

- <http://www.croisedanslemetro.com>
- <http://m.croisedanslemetro.com>

### The Molly Project

Molly is a framework for the rapid development of information and service portals targeted at mobile internet devices: <https://github.com/mollyproject/mollyproject>

It powers the University of Oxford's mobile portal: <http://m.ox.ac.uk/>

## Mozilla

- mozilla.org (<https://github.com/mozilla/bedrock>)
- Mozilla Developer Network (<https://github.com/mozilla/kuma>)

## Novapost

For PeopleDoc suite products: <http://www.people-doc.com/>

## Sophicware

Sophicware offers web hosting and DevOps as a service: <http://sophicware.com>

## Ulule

For their main and forum website:

- <http://www.ulule.com>
- <http://vox.ulule.com>

---

## Indices and tables

---

- search