Django StaticFilesPlus Documentation

Release 0.1dev

David Evans

November 05, 2013

Contents

A tiny library that adds asset pre-processor support to Django's *contrib.staticfiles*.

The bundled *contrib.staticfiles* does most of what I want from an asset manager, but not quite all. Rather than replace it entirely with another asset manager, I wrote this little hack to add the missing piece. Now assets that need pre-processing (like LESS stylesheets, or Sprocketized JS files) get compiled dynamically in development and then written out as static files for serving in production.

Features:

- Seamleess integration with *contrib.staticfiles*: Continue to use the `static` template tag and `collectstatic` command as normal.
- Ships with support for LESS stylesheets and Sprockets-like JavaScript concatenation and minification.
- Very simple API for defining new pre-processors.
- Not a single line of my code needs to run in production: once you've run `collectstatic`, its work is done.

CHAPTER 1

Contents

1.1 Installation and Configuration

```
$ pip install django-staticfilesplus
```

In `settings.py` replace the default `STATICFILES_FINDERS` definition with this:

```
STATICFILES_FINDERS = (
    'staticfilesplus.finders.FileSystemFinder',
    'staticfilesplus.finders.AppDirectoriesFinder',
)
```

And enable the default processors:

```
STATICFILESPLUS_PROCESSORS = (
    'staticfilesplus.processors.less.LESSProcessor',
    'staticfilesplus.processors.js.JavaScriptProcessor',
)
```

Assuming that `django.contrib.staticfiles` is in your `INSTALLED_APPS` (which it is by default) you're ready to go.

1.1.1 Settings

STATICFILESPLUS_PROCESSORS

Default `()`

A list of active processors. See the processor documentation for details on how these work.

STATICFILESPLUS_TMP_DIR

Default `os.path.join(STATIC_ROOT, 'staticfilesplus_tmp)`

A directory in which to write temporary working files. If it doesn't exist it will be created.

1.2 Processors

Processors are responsible for processing static files: rewriting the content and optionally changing the name.

StaticfilesPlus ships with a couple of default processors:

1.2.1 JavaScript Processor

This adds support for Sprockets-like dependancy management for JavaScript files.

Dependencies between files are specified by specially formatted comments (known as directives) at the top of the files. The processor compiles all these dependencies together into a single file.

Activating

Ensure that the processor is in the list of enabled processors in `settings.py`:

```
STATICFILESPLUS_PROCESSORS = (
    ...
    'staticfilesplus.processors.js.JavaScriptProcessor',
    ...
)
```

Directives

The directive processor scans for comment lines beginning with = in comment blocks at the top of the file.

```
//= require jquery
//= require lib/myplugin.js
```

The first word immediately following = specifies the directive name. Any words following the directive name are treated as arguments. Arguments may be placed in single or double quotes if they contain spaces, similar to commands in the Unix shell.

Note: Non-directive comment lines will be preserved in the final asset, but directive comments are stripped after processing. The processor will not look for directives in comment blocks that occur after the first line of code.

The directive processor understands comment blocks in three formats:

```
/* Multi-line comment blocks (CSS, SCSS, JavaScript)
 *= require foo
 */

// Single-line comment blocks (SCSS, JavaScript)
//= require foo

# Single-line comment blocks (CoffeeScript)
#= require foo
```

Directives are comments of the form:

```
/*
*= <directive> <path>
*/

// ... OR ...

//= <directive> <path>
```

Path arguments are parsed like shell arguments so they can be unquoted if they contain no special characters (like spaces) or surrounded with single or double quotes.

To maintain compatibilty with Sprockets you can omit the `.js` extension from paths, but I prefer to be explicit and include the extension.

```
/*
*= <directive> <filename>
*/

// ... OR ...

//= require <filename>
```

Currently, we only support two of the standard Sprockets directives:

require *<filename>* Includes the content of the specified file, if it hasn't already been included. Note: processing is recursive so that directives in required files are themselves processed.

stub *<filename>* Marks the specified file (and all its dependencies) as not for inclusion, even if they are required by other directives. This is useful when you have multiple scripts on a page which may share dependencies and you want to ensure that the common dependencies only get included once. (There's no need to manually work out what the common dependencies are, just stub the entire file.)

```
/*
*
*= require some-library
*= require you-can-explicily-specify-extension.js
*/

//= require "quoting works just like in shell"
//= require ./paths/starting/with-a-dot/are-relative.js
```

Hidden files

The JavaScript processor ignores all files and directories which **start with an underscore**.

These files can still be *require*d by other JavaScript files, but they will not be individually included in the list of compiled files. This allows you to prevent library files from being compiled as stand-alone files.

I tend to put library files in a directory called `_lib` for this reason.

Processing with Django template engine

Files with the extension `.djtmpl.js` will be first processed by Django's templating engine. You should use this feature sparingly (it's quite a nasty hack) but it can help to avoid repeating configuration values (particularly your URL config) in both Python and JavaScript.

In the example below, `config.djtmp.js` pulls in a couple of values from Django's configuration and then `application.js` *requires* it and uses those values.

```
/* application.js */

//= require config.djtmpl.js
$.ajax(URLS.my_endpoint);
console.log(SETTINGS.title);
```

```
/* config.djtmpl.js */

var URLS = {
   my_endpoint: "{% url  'my_endpoint '%}"
};

var SETTINGS = {
   title: "{{ settings.SOME_TITLE }}"
};
```

1.2.2 LESS Processor

This adds support for LESS, the extended CSS dialect.

Usage

Note: You'll need to install the LESS compiler. See the section on *Server-side Usage* in the LESS documentation

---

Ensure that the processor is in the list of enabled processors in `settings.py`:

```
STATICFILESPLUS_PROCESSORS = (
    ...
    'staticfilesplus.processors.less.LESSProcessor',
    ...
)
```

In your templates, link to your LESS files using their post-processed `.css` extension, not their original `.less` extension. For example:

```
/* myapp/static/styles.less */
p {
  color: green;
}

<!-- myapp/templates/base.html -->
...
<link rel="stylesheet" href="{% static 'styles.css' %}" type="text/css"/>
...
```

Hidden files

The LESS processor ignores all files and directories which **start with an underscore**.

LESS itself can still *@import* these files but they will not be individually included in the list of compiled files. This allows you to prevent library files from being compiled as stand-alone files, which will often break anyway if these files rely on variables that have not been defined.

For example, in this case:

```
/* myapp/static/styles.less */
@import '_lib/base.less';

h1 {
```

```
  background-color: blue;
}

/* myapp/static/_lib/base.less */
p {
  color: green;
}
...
```

The final output would contain a `styles.css` file (with the imported content of `base.less`) but no stand-alone `_lib/base.css` file.

Settings

STATICFILESPLUS_LESS_COMPRESS

Default the opposite of `DEBUG`

Passes the `--compress` flag to the LESS compiler to produce minified output suitable for production.

1.2.3 Writing your own processor

The processor API is very simple. For example, here is how you might write a processor to handle CoffeeScript files:

```
from subprocess import check_call
from staticfilesplus.processors import BaseProcessor

class CoffeeScriptProcessor(BaseProcessor):
    original_suffix = '.coffee'
    processed_suffix = '.js'

    def process_file(self, input_path, output_path):
        with open(input_path, 'rb') as infile:
            with open(output_path, 'wb') as outfile:
                check_call(['coffee', '--stdio'], stdin=infile, stdout=outfile)
```

Processor API

A valid processor is class that implements the following set of methods.

First, we have a pair of methods which are given filenames and determine which files the processor will operate on:

get_original_name(*name*)
Takes a processed filename and returns the original filename (which may be the same if the processor doesn't alter filenames) or `None` if the processor doesn't handle this type of file.

For example, the `LESSProcessor` matches any string with a `.css` extension and returns it with a `.less` extension. The `JavaScriptProcessor` matches any string with a `.js` extension and returns it unchanged.

get_processed_name(*name*)
The reverse of the above method. Takes the original filename and returns the file name after processing (which can be the same) or `None` if the processor doesn't handle this type of file.

Then we have the method which does the actual processing of the file:

process_file(input_path, output_path):
Takes the file given by `input_path`, processes it and writes it to `output_path`.

Finally, we have:

is_ignored_file(name):
Takes a filename (before processing) and returns a Boolean. If it returns `True` the file will be ignored by `contrib.staticfiles` as if it did not exist.

To activate your processor, add its dotted path to `STATICFILESPLUS_PROCESSORS` in `settings.py`

BaseProcessor

A convenient base class is provided in `staticfilesplus.processors.BaseProcessor`

```
class BaseProcessor(object):
    original_suffix = None
    processed_suffix = None

    def get_original_name(self, name):
        if name.endswith(self.processed_suffix):
            return name[:-len(self.processed_suffix)] + self.original_suffix

    def get_processed_name(self, name):
        if name.endswith(self.original_suffix):
            return name[:-len(self.original_suffix)] + self.processed_suffix

    def is_ignored_file(self, path):
        return False

    def process_file(self, input_path, output_path):
        raise NotImplementedError()
```