Flask-Assets Documentation

Release 0.12

Michael Elsdörfer

Apr 26, 2017

Contents

Flask-Assets helps you to integrate webassets into your Flask application.

CHAPTER 1

Installation

Install the extension with one of the following commands:

```
$ easy_install Flask-Assets
```

or alternatively if you have pip installed:

```
$ pip install Flask-Assets
```

CHAPTER 2

Usage

You initialize the app by creating an *Environment* instance, and registering your assets with it in the form of so called *bundles*.

```
from flask import Flask
from flask_assets import Environment, Bundle

app = Flask(__name__)
assets = Environment(app)

js = Bundle('jquery.js', 'base.js', 'widgets.js',
            filters='jsmin', output='gen/packed.js')
assets.register('js_all', js)
```

A bundle consists of any number of source files (it may also contain other nested bundles), an output target, and a list of filters to apply.

All paths are relative to your app's **static** directory, or the static directory of a *Flask blueprint*.

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.

Like is common for a Flask extension, a Flask-Asssets instance may be used with multiple applications by initializing through `init_app` calls, rather than passing a fixed application object:

```
app = Flask(__name__)
assets = flask_assets.Environment()
assets.init_app(app)
```

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:

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

That's it, really. **Flask-Assets** will automatically merge and compress your bundle's source files the first time the template is rendered, and will automatically update the compressed file everytime a source file changes. If you set `ASSETS_DEBUG` in your app configuration to `True`, then each source file will be outputted individually instead.

Flask blueprints

If you are using Flask blueprints, you can refer to a blueprint's static files via a prefix, in the same way as Flask allows you to reference a blueprint's templates:

```
js = Bundle('app_level.js', 'blueprint/blueprint_level.js')
```

In the example above, the bundle would reference two files, `{APP_ROOT}/static/app_level.js`, and `{BLUEPRINT_ROOT}/static/blueprint_level.js`.

If you have used the `webassets` library standalone before, you may be familiar with the requirement to set the `directory` and `url` configuration values. You will note that this is not required here, as Flask's static folder support is used instead. However, note that you *can* set a custom root directory or url if you prefer, for some reason. However, in this case the blueprint support of Flask-Assets is disabled, that is, referencing static files in different blueprints using a prefix, as described above, is no longer possible. All paths will be considered relative to the directory and url you specified.

Pre 0.7 modules are also supported; they work exactly the same way.

Templates only

If you prefer, you can also do without defining your bundles in code, and simply define everything inside your template:

```
{% assets filters="jsmin", output="gen/packed.js",
          "common/jquery.js", "site/base.js", "site/widgets.js" %}
    <script type="text/javascript" src="{{ ASSET_URL }}"></script>
{% endassets %}
```

CHAPTER 3

Configuration

webassets supports a couple of configuration options. Those can be set both through the *Environment* instance, as well as the Flask configuration. The following two statements are equivalent:

```
assets_env.debug = True
app.config['ASSETS_DEBUG'] = True
```

For a list of available settings, see the full webassets documentation.

Babel Configuration

If you use Babel for internationalization, then you will need to add the extension to your babel configuration file as `webassets.ext.jinja2.AssetsExtension`

Otherwise, babel will not extract strings from any templates that include an `assets` tag.

Here is an example `babel.cfg`:

```
[python: **.py]
[jinja2: **.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_,webassets.ext.jinja2.AssetsExtension
```

Flask-S3 Configuration

Flask-S3 allows you to upload and serve your static files from an Amazon S3 bucket. It accomplishes this by overwriting the Flask `url_for` function. In order for Flask-Assets to use this overwritten `url_for` function, you need to let it know that you are using Flask-S3. Just set

```
app.config['FLASK_ASSETS_USE_S3']=True
```

Flask-CDN Configuration

Flask-CDN allows you to upload and serve your static files from a CDN (like Amazon Cloudfront), without having to modify your templates. It accomplishes this by overwriting the Flask `url_for` function. In order for Flask-Assets to use this overwritten `url_for` function, you need to let it know that you are using Flask-CDN. Just set

```
app.config['FLASK_ASSETS_USE_CDN']=True
```

CHAPTER 4

Command Line Interface

New in version 0.12.

Flask 0.11+ comes with build-in integration of CLI using click library. The `assets` command is automatically installed through *setuptools* using `flask.commands` entry point group in **setup.py**.

```
entry_points={
    'flask.commands': [
        'assets = flask_assets:assets',
    ],
},
```

After installing Flask 0.11+ you should see following line in the output when executing `flask` command in your shell:

```
$ flask --help
...
Commands:
  assets   Web assets commands.
...
```

Legacy support

If you have Flask-Script installed, then a command will be available as `flask_assets.ManageAssets`:

```
from flask_assets import ManageAssets
manager = Manager(app)
manager.add_command("assets", ManageAssets(assets_env))
```

You can explicitly pass the `assets_env` when adding the command as above. Alternatively, `ManageAssets` will import the `current_app` from Flask and use the `jinja_env`.

The command allows you to do things like rebuilding bundles from the command line. See the list of available subcommands.

Using in Google App Engine

You can use flask-assets in Google App Engine by manually building assets. The GAE runtime cannot create files, which is necessary for normal flask-assets functionality, but you can deploy pre-built assets. You can use a file change listener to rebuild assets on the fly in development.

For a fairly minimal example which includes auto-reloading in development, see flask-assets-gae-example.

Also see the relevant webassets documentation.

CHAPTER 5

API

Integration of the `webassets` library with Flask.

class `flask_assets.`**`Environment`**(*app=None*)
This object is used to hold a collection of bundles and configuration.

If it initialized with an instance of Flask application then webassets Jinja2 extension is automatically registered.

`config_storage_class`
alias of *`FlaskConfigStorage`*

`directory`
The base directory to which all paths will be relative to.

`from_module`(*path*)
Register bundles from a Python module

`from_yaml`(*path*)
Register bundles from a YAML configuration file

`resolver_class`
alias of *`FlaskResolver`*

`url`
The base url to which all static urls will be relative to.

class `flask_assets.`**`Bundle`**(**contents*, ***options*)
A bundle is the unit webassets uses to organize groups of media files, which filters to apply and where to store them.

Bundles can be nested arbitrarily.

A note on the connection between a bundle and an "environment" instance: The bundle requires a environment that it belongs to. Without an environment, it lacks information about how to behave, and cannot know where relative paths are actually based. However, I don't want to make the `Bundle.__init__` syntax more complicated than it already is by requiring an Environment object to be passed. This would be a particular nuisance when nested bundles are used. Further, nested bundles are never explicitly connected to an Environment, and what's more, the same child bundle can be used in multiple parent bundles.

This is the reason why basically every method of the Bundle class takes an `env` parameter - so a parent bundle can provide the environment for child bundles that do not know it.

build(*force=None*, *output=None*, *disable_cache=None*)
Build this bundle, meaning create the file given by the `output` attribute, applying the configured filters etc.

If the bundle is a container bundle, then multiple files will be built.

Unless `force` is given, the configured `updater` will be used to check whether a build is even necessary.

If `output` is a file object, the result will be written to it rather than to the filesystem.

The return value is a list of `FileHunk` objects, one for each bundle that was built.

depends
Allows you to define an additional set of files (glob syntax is supported), which are considered when determining whether a rebuild is required.

extra
A custom user dict of extra values attached to this bundle. Those will be available in template tags, and can be used to attach things like a CSS 'media' value.

get_version(*ctx=None*, *refresh=False*)
Return the current version of the Bundle.

If the version is not cached in memory, it will first look in the manifest, then ask the versioner.

`refresh` causes a value in memory to be ignored, and the version to be looked up anew.

id()
This is used to determine when a bundle definition has changed so that a rebuild is required.

The hash therefore should be built upon data that actually affect the final build result.

is_container
Return true if this is a container bundle, that is, a bundle that acts only as a container for a number of sub-bundles.

It must not contain any files of its own, and must have an empty `output` attribute.

iterbuild(*ctx*)
Iterate over the bundles which actually need to be built.

This will often only entail `self`, though for container bundles (and container bundle hierarchies), a list of all the non-container leafs will be yielded.

Essentially, what this does is "skip" bundles which do not need to be built on their own (container bundles), and gives the caller the child bundles instead.

The return values are 3-tuples of (bundle, filter_list, new_ctx), with the second item being a list of filters that the parent "container bundles" this method is processing are passing down to the children.

resolve_contents(*ctx=None*, *force=False*)
Return an actual list of source files.

What the user specifies as the bundle contents cannot be processed directly. There may be glob patterns of course. We may need to search the load path. It's common for third party extensions to provide support for referencing assets spread across multiple directories.

This passes everything through `Environment.resolver`, through which this process can be customized.

At this point, we also validate source paths to complain about missing files early.

The return value is a list of 2-tuples `(original_item, abspath)`. In the case of urls and nested bundles both tuple values are the same.

Set `force` to ignore any cache, and always re-resolve glob patterns.

resolve_output(*ctx=None*, *version=None*)
Return the full, absolute output path.

If a %(version)s placeholder is used, it is replaced.

urls(**args*, ***kwargs*)
Return a list of urls for this bundle.

Depending on the environment and given options, this may be a single url (likely the case in production mode), or many urls (when we source the original media files in DEBUG mode).

Insofar necessary, this will automatically create or update the files behind these urls.

class flask_assets.**FlaskConfigStorage**(**a*, ***kw*)
Uses the config object of a Flask app as the backend: either the app instance bound to the extension directly, or the current Flask app on the stack.

Also provides per-application defaults for some values.

Note that if no app is available, this config object is basically unusable - this is by design; this could also let the user set defaults by writing to a container not related to any app, which would be used as a fallback if a current app does not include a key. However, at least for now, I specifically made the choice to keep things simple and not allow global across-app defaults.

setdefault(*key*, *value*)
We may not always be connected to an app, but we still need to provide a way to the base environment to set it's defaults.

class flask_assets.**FlaskResolver**
Adds support for Flask blueprints.

This resolver is designed to use the Flask staticfile system to locate files, by looking at directory prefixes (`foo/bar.png` looks in the static folder of the `foo` blueprint. `url_for` is used to generate urls to these files.

This default behaviour changes when you start setting certain standard *webassets* path and url configuration values:

If a *Environment.directory* is set, output files will always be written there, while source files still use the Flask system.

If a `Environment.load_path` is set, it is used to look up source files, replacing the Flask system. Blueprint prefixes are no longer resolved.

convert_item_to_flask_url(*ctx*, *item*, *filepath=None*)
Given a relative reference like *foo/bar.css*, returns the Flask static url. By doing so it takes into account blueprints, i.e. in the aforementioned example, `foo` may reference a blueprint.

If an absolute path is given via `filepath`, it will be used instead. This is needed because `item` may be a glob instruction that was resolved to multiple files.

If app.config("FLASK_ASSETS_USE_S3") exists and is True then we import the url_for function from flask_s3, otherwise we import url_for from flask directly.

If app.config("FLASK_ASSETS_USE_CDN") exists and is True then we import the url_for function from flask.

split_prefix(*ctx*, *item*)
See if `item` has blueprint prefix, return (directory, rel_path).

class flask_assets.**Jinja2Filter**(*context=None*)
Will compile all source files as Jinja2 templates using the standard Flask contexts.

class flask_assets.**ManageAssets**(*assets_env=None*, *impl=<class 'flask_assets.FlaskArgparseInterface'>*, *log=None*)
Manage assets.

run(*args*)
Runs the management script. If `self.env` is not defined, it will import it from `current_app`.

CHAPTER 6

Webassets documentation

For further information, have a look at the complete webassets documentation, and in particular, the following topics:

- Configuration
- All about bundles
- Builtin filters
- Custom filters
- CSS compilers
- FAQ

Python Module Index

f

Index