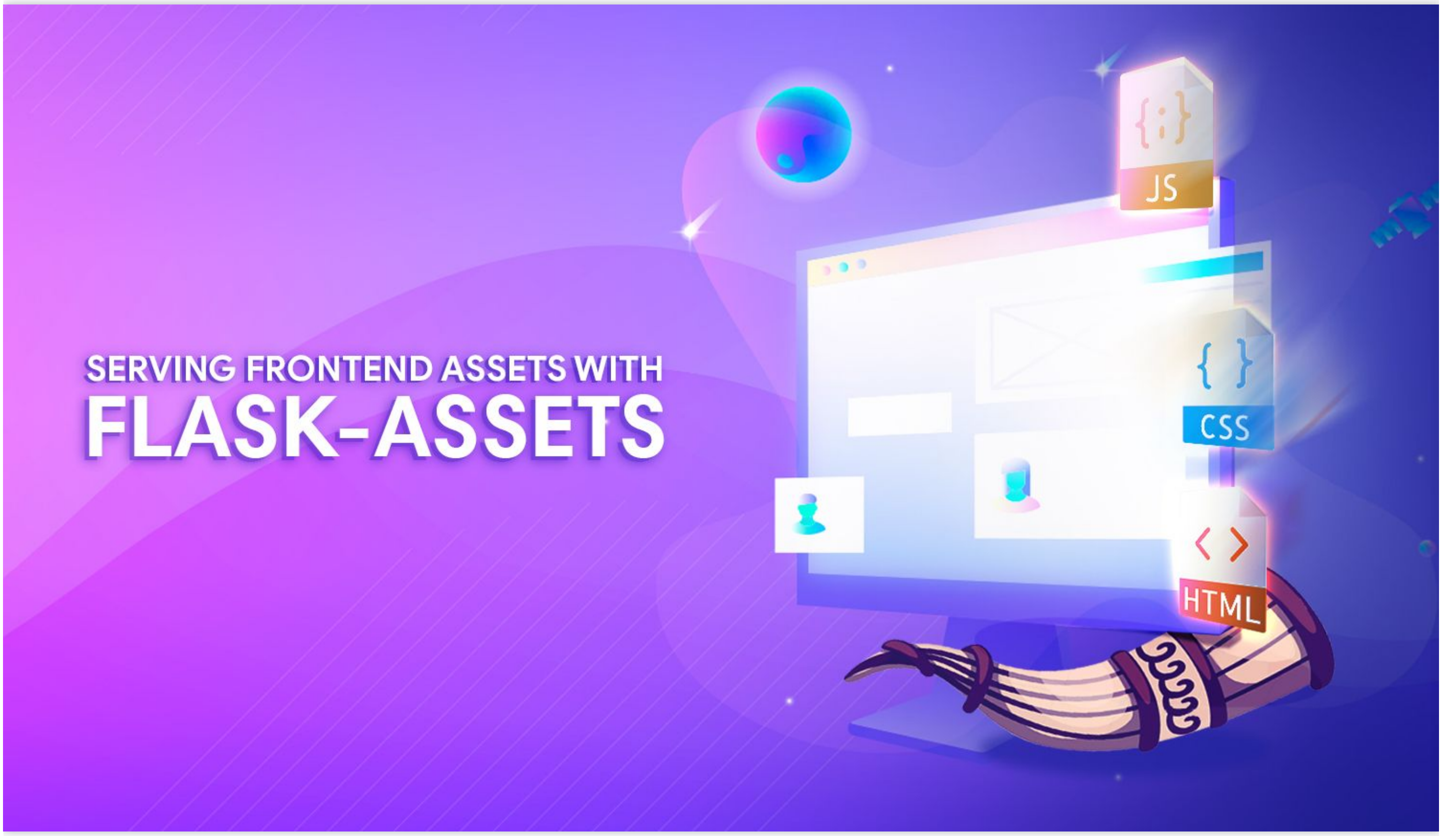
undefined Logo

Compiling and Serving Frontend Assets in Flask

Todd Flask 8 Min Read



You're probably not a frontend developer, and I'm guessing you have no desire to become one. I'm fairly confident of this because you're currently engaged in a Flask tutorial, as opposed to complaining about the number of frontend frameworks you've been forced to learn in the last decade (sorry frontend fam, you know I love you 😄).

Every backend dev needs to come to terms with a stark reality: meaningful apps can't be built without eventually digging into writing some styles or frontend JavaScript. Since the early days of Django, Python developers have opted to coin frontend code as static assets: a backhanded term that seemingly downplays the importance of JavaScript, CSS, or any presentation layer assets which make up a web application. It had been decreed that only two types of code exist in the universe: Python, and not-Python. Anything that isn't a **.py** file is considered to be a "static" asset, tucked into a single directory after presumably receiving a handoff from a frontend developer. This implied mindset is both somewhat absurd yet also unsurprising.

It's time to face our fears of dealing with frontend code in Flask the **right** way: by writing styles in CSS preprocessors, minifying files, and serving them in "bundles" akin to the webpack equivalent. If you're the type of person who'd prefer to link to a CDN-hosted version of jQuery 1.11 or include a hideously generic Bootstrap library to style your app for you, please stop reading and go away. For those of us with higher aspirations than a full-time job maintaining 30-year old legacy code for Citibank, we'll be learning about Flask-Assets.

How it Works

Flask applications utilizing the Flask-Assets library will generally have two directories related to "static" assets. One directory is reserved for raw uncompressed code, while the other is for production-ready, compressed assets generated from files stored in the former (this production-ready directory is commonly referred to as a **/dist** or **/build** folder). Depending on how we configure our app, Flask-Assets will compile static assets into bundles upon startup. This is accomplished with the help of a few auxiliary Python libraries dedicated to building and compressing files: lesscpy and pyscss compile LESS/SCSS to CSS respectively, while cssmin and jsmin compress CSS/JavaScript.

Depending on our choice of stack, we'll need to install the appropriate Python packages. I plan on writing styles with LESS and adding some frontend JavaScript to my app:

Install dependencies

```
$ pip install flask-assets lesscpy cssmin jsmin
```

Flask-Assets Configuration

As with most Flask plugins, we can configure Flask-assets by setting variables in our top-level Flask config. Whoever created this library made the interesting choice of *not documenting any of this*, so I'm going to save you some time:

config.py

```
LESS_BIN = '/usr/local/bin/lessc'
ASSETS_DEBUG = False
ASSETS_AUTO_BUILD = True
```

- LESS_BIN: LESS requires the npm library called LESS (known to the system as *lessc*) to be installed on our system in order to compile LESS into CSS. Run `which lessc` in your terminal to find this path. The SASS equivalent to this is SASS_BIN.
- ASSETS_DEBUG: We're bound to screw up a style or JS function every now and then, which sometimes doesn't become obvious until we build our bundles and try to use our app. If we set this variable to `True`, Flask-Assets *won't* bundle our static files while we're running Flask in debug mode. This is useful for troubleshooting styles or JavaScript gone wrong.
- ASSETS_AUTO_BUILD: A flag to tell Flask to build our bundles of assets when Flask starts up automatically. We generally want to set this to be True, unless you'd prefer only to build assets with some external force (AKA: you're annoying).

Creating Asset Bundles

Before diving into code, let's establish our app's structure. Nothing out of the ordinary here:

Our project structure.

```
/app
├── /static
│   ├── /img
│   ├── /dist
│   │   ├── /css
│   │   └── /js
│   └── /src
│       ├── /js
│       └── /less
├─ /templates
├─ config.py
└─ app.py
```

The key takeaway is that we're going to set up Flask-Assets to look for raw files to compile in **static/src/less** and **static/src/js**. When the contents of these folders are compiled, we'll output the results to **static/dist/less** and **static/dist/js** where they'll be ready to serve.

Let's get this baby fired up in **app.py**:

app.py

```
from flask import Flask
from flask_assets import Environment


app = Flask(__name__, instance_relative_config=False)

assets = Environment(app)
```

We initialize Flask-Assets by creating an **Environment** instance, and initializing it against Flask's **app** object. There's an alternative way of initializing the library via the Flask application factory, but we'll get there in a moment.

Our first asset "bundle" is going to come from LESS files found in our **static/src/less** directory. We're going to compile these files into CSS, minify the resulting CSS, and output it for production:

app.py

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


app = Flask(__name__, instance_relative_config=False)
assets = Environment(app)

style_bundle = Bundle(
    'src/less/*.less',
    filters='less,cssmin',
    output='dist/css/style.min.css',
    extra={'rel': 'stylesheet/css'}
)

assets.register('main_styles', style_bundle)  # Register style bundle
style_bundle.build()  # Build LESS styles
```

The **Bundle** object is where most of our magic happens. Bundle can accept any number of positional arguments, where each argument is the path of a file to include in our bundle.

Our example creates a bundle from `'src/less/*.less'`, which means "all **.less** files in **src/less**." We could instead pass individual files (like `'src/less/main.less', 'src/less/variables.less'`, etc) if we want to cherry-pick specific stylesheets. In this scenario, it's important to note that any imports within these LESS files (such as `@import './other_styles.less'`) would automatically be included in our bundle as well.

The next step in creating our **Bundle** is passing some necessary keyword arguments:

- **Filters**: This is where we tell Flask-Assets which Python libraries to use to handle compiling our assets. Passing `'less,cssmin'` tells our Bundle that we're going to build a bundle from LESS files using **lesscpy**, which we should then compress using **cssmin**.
- **Output**: The directory where our production-ready bundle will be saved and served from.
- **Extra**: Appends additional HTML attributes to the bundle output in our Jinja template (this usually isn't necessary).

After creating a bundle of styles called **style_bundle**, we "register" the bundle with our environment with `assets.register('main_styles', style_bundle)` (`'main_styles'` is the arbitrary name we assign to our bundle, and `style_bundle` is the variable itself).

We finally build our bundle with `style_bundle.build()`. This outputs a single CSS file to **dist/css/style.css**.

Creating a JavaScript Bundle

The process for creating a bundle for frontend JavaScript is almost identical to to the bundle we created for styles. The main difference is our usage of the **jsmin** filter to minimize the JS files.

Here's an example of creating bundles for both in unison:

```
...

style_bundle = Bundle(
    'src/less/*.less',
    filters='less,cssmin',
    output='dist/css/style.min.css',
    extra={'rel': 'stylesheet/css'}
)
js_bundle = Bundle(
    'src/js/main.js',
    filters='jsmin',
    output='dist/js/main.min.js'
)
assets.register('main_styles', style_bundle)
assets.register('main_js', js_bundle)
style_bundle.build()
js_bundle.build()
```

Serving Bundles via Page Templates

We can serve our newly created assets in our page templates as we would any other static asset:

index.jinja2

```
{% extends "layout.jinja2" %}

<link
   href="{{ url_for('static', filename='dist/css/style.min.css') }}"
   rel="stylesheet"
   type="text/css"
/>
<script src="{{ url_for('static', filename='dist/js/main.min.js') }}"></script>
```

Launching your Flask app should now result in styles and JavaScript being built to **src/dist**. If all went well, your page templates will pick these up and your app will be looking snazzy with some sexy, performance-optimized frontend assets.

Protip

There's apparently an alternative way to create bundles entirely in templates with zero Python code. I personally haven't tried this out, but the simplicity of the syntax is pretty f'in cool:

```
{% assets filters="jsmin", output="src/js/main.js", "src/js/jquery.js" %} {% endassets %}
```

Blueprint-specific Bundles

One of the most powerful features of Flask-assets is the ease with which we can create Blueprint-specific asset bundles. This allows us to "code-split" relevant frontend assets to be exclusive to Blueprints, primarily for performance purposes. We're going to build on the example project we created last time in our Flask Blueprints tutorial to create blueprint-specific bundles.

Here's a quick overview of a project with two registered blueprints (/main and /admin) which we'll create asset bundles for:

```
flask-assets-tutorial
├── /flask_assets_tutorial
│   ├── __init__.py
│   ├── assets.py
│   ├── /admin
│   │   ├── admin_routes.py
│   │   ├── /static
│   │   └── /templates
│   ├── /main
│   │   ├── main_routes.py
│   │   ├── /static
│   │   └── /templates
│   ├── /static
│   └── /templates
├── config.py
└── wsgi.py
```

Each of our blueprints has its own **static** folder, which is intended to contain blueprint-specific assets. Our app still has a **static** folder of its own, containing assets to be shared across all blueprints. Before getting into the details of how that works, let's look at how we instantiate Flask-assets in the Flask application factory:

__init__.py

```python
"""Initialize app."""
from flask import Flask
from flask_assets import Environment
from .assets import compile_assets

assets = Environment()


def create_app():
    """Construct the core application."""
    app = Flask(__name__, instance_relative_config=False)
    app.config.from_object('config.Config')

    # Initialize plugins
    assets.init_app(app)

    with app.app_context():
        # Import parts of our application
        from .admin import admin_routes
        from .main import main_routes
        app.register_blueprint(admin_routes.admin_bp)
        app.register_blueprint(main_routes.main_bp)
        compile_assets(assets)

        return app
```

Let's check out what's happening in **assets.py**:

assets.py

```
"""Compile static assets."""
from flask import current_app as app
from flask_assets import Bundle


def compile_static_assets(assets):
    """Configure and build asset bundles."""
    main_style_bundle = Bundle(
        'src/less/*.less',
        'main_bp/homepage.less',
        filters='less,cssmin',
        output='dist/css/landing.css',
        extra={'rel': 'stylesheet/css'}
    )  # Main Stylesheets Bundle
    main_js_bundle = Bundle(
        'src/js/main.js',
        filters='jsmin',
        output='dist/js/main.min.js'
    )  # Main JavaScript Bundle
    admin_style_bundle = Bundle(
        'src/less/*.less',
        'admin_bp/admin.less',
        filters='less,cssmin',
        output='dist/css/account.css',
        extra={'rel': 'stylesheet/css'}
    )  # Admin Stylesheets Bundle
    assets.register('main_styles', main_style_bundle)
    assets.register('main_js', main_js_bundle)
    assets.register('admin_styles', admin_style_bundle)
    if app.config['FLASK_ENV'] == 'development':
        main_style_bundle.build()
        main_js_bundle.build()
        admin_style_bundle.build()
```

The magic comes in to play in `Bundle()`'s positional arguments. In the case of **main_style_bundle**, we pass global styles with `src/less/*.less` and *blueprint specific* styles with `main_bp/homepage.less`. But wait, **main_bp** isn't a directory! In fact, it's the registered name of our main blueprint that we create in **main_routes.py**:

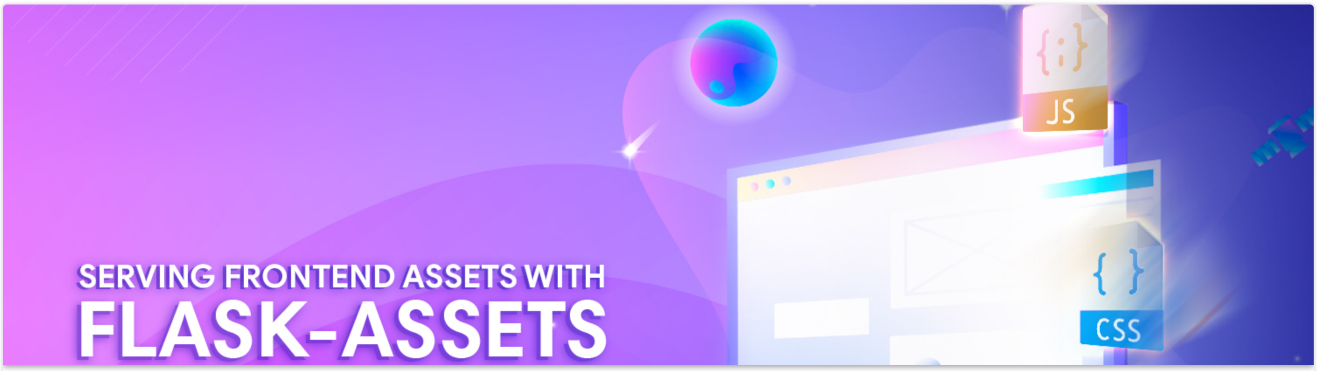
main_routes.py

```
"""Routes for main pages."""
from flask import Blueprint


main_bp = Blueprint(
    'main_bp',
    __name__,
    template_folder='templates',
    static_folder='static'
)
```

When setting `static_folder='static'` on our blueprint, we're telling Flask to check for assets in *two* places: first in flask_assets_tutorial/static, and then flask_assets_tutorial/main/static second. `Bundle()` recognizes that `main_bp` is a blueprint name as opposed to a directory and thus resolves the asset correctly.

Setting up blueprint-specific assets isn't a walk in the park the first time around. The official documentation for Flask-assets is simply god-awful. As a result, there are plenty of pitfalls and quirks that can ruin somebody's day... especially mine. If you find yourself getting stuck, I've set up a working project on Github here:



hackersandslackers/flask-assets-tutorial

Working example of a Flask application utilizing blueprint-specific assets. - hackersandslackers/flask-assets-tutorial

hackersandslackers • GitHub

Flask Python Software Development Frontend

Todd Birchard's avatar