undefined Logo

Organizing Flask Apps with Blueprints

Todd Flask 11 Min Read

XI. Managing Session Data with Flask-Session & Redis
X. Handle User Accounts & Authentication in Flask with Flask-Login
IX. Connect Flask to a Database with Flask-SQLAlchemy
VIII. Compiling and Serving Frontend Assets in Flask
VII. **Organizing Flask Apps with Blueprints**
VI. Demystifying Flask's Application Factory
V. Configuring Your Flask App
IV. The Art of Routing in Flask
III. Handling Forms in Flask with Flask-WTF
II. Rendering Pages in Flask Using Jinja
I. Creating Your First Flask Application

I'm unclear as to when or why it happened, but at some point in the last couple of years, I became emotionally and perhaps romantically involved with Flask (I'm sure you've noticed). I've since accepted my fate as being the *guy-who-writes-flask-tutorials-on-the-*

internet-for-no-particular-reason with few regrets. There aren't many perks that come with that title, but it certainly offers some unique perspective, especially when it comes to widespread misconceptions about what people believe about Flask. Of these misconceptions, two are particularly popular as much as they are false:

1. Flask is intended for building small-scale apps, while Django is better-suited for larger-scale apps (I may address this directly, although the nature of this tutorial should accomplish that on its own).
2. Flask is a *microframework*, which therefore implies we should be writing *"microprojects,"* where all logic is dumped into a single file called **app.py.**

It's no surprise as to why Flask newcomers believe that structuring entire projects into single **app.py** files is normal and acceptable. Almost *every Flask tutorial* is structured this way, presumably out of convenience, but it seems as though an asterisk is missing from these tutorials which should read ***THIS IS NOT A GOOD WAY TO BUILD APPLICATIONS.***

Good software is organized by separation of concerns. It's easy to think of an application as a single entity, yet reality shows us that well-structured apps are *collections* of standalone **modules**, **services**, or **classes**. I'm referring to the Single-responsibility principal: a commonly understood concept where each piece of an application should be limited to handling no more than a single responsibility. It's the same reason why bathrooms and kitchens are separate rooms: one room is for shitting, and one room is for eating. It's a mess to shit where you eat, which is precisely what happens when building applications without structure or encapsulation.

We can organize our Flask apps via a built-in concept called **Blueprints**, which are essentially the Flask equivalent of Python modules. Blueprints are intended to encapsulate feature-sized sections of our application, such as checkout flows, user auth, user profiles, etc. (basically anything suitable to be represented as a JIRA epic). Blueprints keep related logic and assets grouped and separated from one another, which is essential to designing a maintainable project. Blueprints also enable performance benefits associated with code-splitting, similar to Webpack.

Conventional Flask Structure

The best way to get a glance at the benefits associated with using Blueprints is to see the effect they have on an app's structure. First, let's look at what a large app might look like *without* Blueprints.

Meet Billy. Billy is pretty new to Flask, but is confident in his abilities to build large-scale Flask apps after reading parts 1-5 in this series. Armed with only that knowledge, his app looks something like this:

```
/myapp
├── /templates
├── /static
├── app.py
├── routes.py
├── models.py
├── config.py
└── requirements.txt
```

Billy has a lot of what you might expect a lot at first glance. He has his handy **app.py** file (ugh), a folder for templates, a folder for static assets, and so forth. Without knowing what Billy's app does, this all seems reasonable. As it turns out, however, *not knowing what the app does* is our downfall in making such an assessment.

It turns out Billy is building a customer-facing service desk portal for Boeing. After rushing the Boeing 737 to market and murdering hundreds of people with criminal negligence, Boeing hired Billy to build a friendly service desk to field complaints from the families of victims. Service desks are complicated pieces of technology. Consider the major parts of what might go into something like Zendesk:

- A public-facing landing page.
- A logged-in experience for both customers and agents (Boeing employees).
- Dashboards with performance metrics for admins.
- An obscene amount of logic that isn't even worth mentioning.

Billy's app suddenly looks like the type of insult to engineering that only the creators of the Boeing 737 would commission. Is Billy *really* storing routes, templates, and assets for all these things in one place?

App Structure with Blueprints

Flask blueprints enable us to organize our applications by grouping logic into subdirectories. To visualize this, we're going to create an example app with three blueprints: **home**, **profile**, and **products**:

Flask app structure with Blueprints.

```
/flask-blueprint-tutorial
├── /flask_blueprint_tutorial
│   ├── __init__.py
│   ├── assets.py
│   ├── api.py
│   ├── /home
│   │   ├── /templates
│   │   ├── /static
│   │   └── home.py
│   ├── /profile
│   │   ├── /templates
│   │   ├── /static
│   │   └── profile.py
│   ├── /products
│   │   ├── /templates
│   │   ├── /static
│   │   └── product.py
│   ├── /static
│   └── /templates
├── README.md
├── config.py
├── requirements.txt
└── wsgi.py
```

Unlike Billy's app, it's immediately apparent what the above app consists of and how logic is divided. **Product** pages don't share much in common with **profile** pages, so it makes plenty of sense to keep these things separated.

A substantial benefit of blueprints is the ability to separate page templates and static assets into blueprint-specific **/templates** and **/static** folders. Now our landing page won't find itself loading irrelevant CSS, which pertains to product pages, and vice versa.

An important thing to point out is the presence of top-level **/templates** and /**static** folders *in addition* to their blueprint-specific equivalents. While blueprints cannot access the templates or static files of their peers, they *can* utilize common assets to be shared across all blueprints (such as a **layout.html** template, or a general **style.css** file that applies to all parts of our app). We'll dig into how assets and blueprints work together, but let's start by defining our first blueprint.

Protip

If you happen to be a Django person, this may start to sound familiar. That's because we can equate Flask's Blueprints to Django's concept of apps. There are differences and added flexibility, but the concept remains the same.

Defining a Blueprint

Our first blueprint will be our "home" blueprint, which will be where all routes, templates, and logic regarding homepage stuff will live. Inside the **/home** directory, we'll create a file called **home.py** to define our blueprint and its routes:

home/home.py

```
from flask import Blueprint
from flask import current_app as app


# Blueprint Configuration
home_bp = Blueprint(
    'home_bp', __name__,
    template_folder='templates',
    static_folder='static'
)
```

We're configuring our Blueprint as a variable named **home_bp**. The first parameter we pass to `Blueprint()` is the name we want to assign to our Blueprint for Flask's internal routing purposes. It makes sense to keep things consistent by naming our Blueprint `home_bp` to stay consistent with our variable name.

We also pass two optional keyword arguments called `template_folder` and `static_folder`. Defining these arguments tells our blueprint that we'll have blueprint-specific templates and static files. These directories are resolved *in relation to the current file*, thus registering our blueprint's template and static directories as **flask_blueprint_tutorial/home/templates** and **flask_blueprint_tutorial/home/static**, respectively.

Setting `static_folder` mimics the behavior of setting `template_folder`, except for serving static assets.

Creating Routes Inside a Blueprint

Now that we have a Blueprint created, we can start adding routes to this subsection of our app! Here's how that looks:

home/home.py

```
from flask import Blueprint, render_template
from flask import current_app as app
from flask_blueprint_tutorial.api import fetch_products


# Blueprint Configuration
home_bp = Blueprint(
    'home_bp', __name__,
    template_folder='templates',
    static_folder='static'
)


@home_bp.route('/', methods=['GET'])
def home():
    """Homepage."""
    products = fetch_products(app)
    return render_template(
        'index.jinja2',
        title='Flask Blueprint Demo',
        subtitle='Demonstration of Flask blueprints in action.',
        template='home-template',
        products=products
    )
```

The only notable difference here is that we now register our route by using `@home_bp.route('...')` instead of `@app.route('...')`. Creating our `home_bp` Blueprint automatically gives us a decorator function called `@home_bp` with which we can register our routes to.

Registering Our Blueprint

We've created a Blueprint and registered our first route to it, but how do we tell our Flask app that this blueprint and its routes exist?

This is where we break into our top-level entry point, like **__init__.py** in the Flask application factory pattern. Here's the simplest example of registering a Blueprint using this pattern:

__init__.py

```
"""Initialize Flask app."""
from flask import Flask


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

    with app.app_context():
        # Import parts of our application
        from .home import home

        # Register Blueprints
        app.register_blueprint(home.home_bp)
        return app
```

We import **home/home.py** inside our `app_context()` via this line:

```
from .home import home
```

With all the contents of **home.py** imported, we can then use a method on our **app** object called `.register_blueprint()` to tell Flask to recognize this section of our app. We pass in the *variable* name **home_bp** from **home.py**, and that's it! Our Flask app will now respect the routes registered to **home_bp**, which in this case happens to be our homepage.

We'll almost certainly have more than a single Blueprint, since modularizing our app is worthless if we only have one module. Registering multiple Blueprints is simple:

__init__.py

```
"""Initialize Flask app."""
from flask import Flask


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

    with app.app_context():
        # Import parts of our application
        from .profile import profile
        from .home import home
        from .products import products

        # Register Blueprints
        app.register_blueprint(profile.account_bp)
        app.register_blueprint(home.home_bp)
        app.register_blueprint(products.product_bp)

        return app
```

We've now registered the three blueprints we saw in our example: **home**, **profile**, and **products**.

Jinja Templates Routing with Blueprints

A caveat we need to be aware of is how Jinja templates find URLs for routes registered to Blueprints. Let's say we want to create a link to our app's homepage. In an app *without* blueprints, you'd find the route for our homepage using the following:

index.jinja2

```
<a href="{{ url_for('home') }}">Home</a>
```

This looks for a route named **home** registered directly to our app, but in *our* case, we don't have routes registered to our app. In our app, we don't register routes directly to the Flask app – we've registered them to *blueprints* instead.

This is where the `'home_bp'` part of `home_bp = Blueprint('home_bp')` comes into play: the *string* `'home_bp'` we passed into our blueprint is the internal name Flask uses to resolve routes with! Now we can find our homepage by doing the following:

index.jinja2

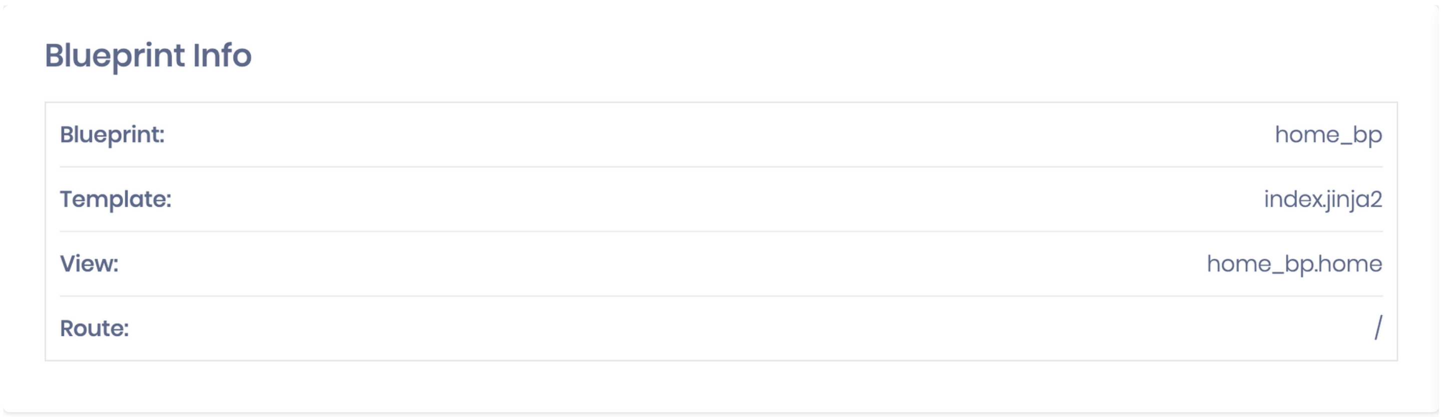
```
<a href="{{ url_for('home_bp.home') }}">Home</a>
```

Jinja Blueprint Extras

Since Jinja is contextually aware of blueprints, we can use a few cool tags to verify where we are in our app. Here are some useful tools to demonstrate how powerful Jinja is when it comes to knowing each page template's context:

- `{{ request.blueprint }}`: Output the blueprint name that the current page template belongs to.
- `{{ self._TemplateReference__context.name }}`: Renders the file name of the current page template.
- `{{ request.endpoint }}` Outputs *both* the name of the current blueprint as well as the name of the route which served the page template.
- `{{ request.path }}`: The URL of the current page template.

Just for fun, I spun up a working demo of our example app here. Here's what the homepage returns for all the above values:



Blueprint Info

Blueprint:	home_bp
Template:	index.jinja2
View:	home_bp.home
Route:	/

Jinja blueprint awareness.

Blueprint-specific Assets

The most difficult aspect of Flask blueprints is the concept of blueprint-specific assets. In my opinion, the best way to get the most out of blueprint-specific assets is with Flask-Assets. Using the Flask-Assets library is a tutorial in itself (clearly seeing as how I've just linked to it), but I'll happily give you a crash course you'll hardly retain for now.

To initialize Flask-Assets (`pip install flask-assets` , by the way), we need to revisit __init__.py:

__init__.py

```python
"""Initialize Flask app."""
from flask import Flask
from flask_assets import Environment  # <-- Import `Environment`


def create_app():
    """Create Flask application."""
    app = Flask(__name__, instance_relative_config=False)
    app.config.from_object('config.Config')
    assets = Environment()  # Create an assets environment
    assets.init_app(app)  # Initialize Flask-Assets

    with app.app_context():
        # Import parts of our application
        from .profile import profile
        from .home import home
        from .products import products
        from .assets import compile_static_assets  # Import logic

        # Register Blueprints
        app.register_blueprint(profile.account_bp)
        app.register_blueprint(home.home_bp)
        app.register_blueprint(products.product_bp)

        # Compile static assets
        compile_static_assets(assets)  # Execute logic

        return app
```

We first create an empty assets "Environment" called **assets** with the line `assets = Environment()`. As with all Flask plugins/add-ons/whatever-they're-called, we then initialize Flask-Assets after our **app** object has been created by calling `.init_app()` on said **app** object.

Within `app_context()`, I then pull in the help of a function called `compile_static_assets()` which I've imported from a file called **assets.py**:

assets.py

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


def compile_static_assets(assets):
    """Create stylesheet bundles."""
    assets.auto_build = True
    assets.debug = False
    common_less_bundle = Bundle(
        'src/less/*.less',
        filters='less,cssmin',
        output='dist/css/style.css',
        extra={'rel': 'stylesheet/less'}
    )
    home_less_bundle = Bundle(
        'home_bp/less/home.less',
        filters='less,cssmin',
        output='dist/css/home.css',
        extra={'rel': 'stylesheet/less'}
    )
    profile_less_bundle = Bundle(
        'profile_bp/less/profile.less',
        filters='less,cssmin',
        output='dist/css/profile.css',
        extra={'rel': 'stylesheet/less'}
    )
    product_less_bundle = Bundle(
        'products_bp/less/products.less',
        filters='less,cssmin',
        output='dist/css/products.css',
        extra={'rel': 'stylesheet/less'}
    )
    assets.register('common_less_bundle', common_less_bundle)
    assets.register('home_less_bundle', home_less_bundle)
    assets.register('profile_less_bundle', profile_less_bundle)
    assets.register('product_less_bundle', product_less_bundle)
    if app.config['FLASK_ENV'] == 'development':
        common_less_bundle.build()
        home_less_bundle.build()
        profile_less_bundle.build()
        product_less_bundle.build()
    return assets
```

Aaand here's where things probably look a little daunting (did I mention there's a separate tutorial for this)? The gist of what's happening is that we're created four stylesheet asset "bundles." That's one asset bundle per blueprint, as well as a bundle for global styles.

The first argument of `Bundle()` is the filepath of static source files to make up our bundle. **common_less_bundle** is given `'src/less/*.less'`, which actually resolves to the following in our app:

Our app's default static folder.

```
flask_blueprint_tutorial/static/src/less/*.less
```

`Bundle()` assumes the `application/static` part of our filepath, as this is the default static folder define with our app. To see where things get more interesting, check out what we pass to **home_less_bundle**:

```
'home_bp/less/home.less'
```

Flask is able to resolve this to the location of our blueprint the same way Jinja does:

The static files of home_bp.

```
flask_blueprint_tutorial/home/static/less/home.less
```

And here's how we'd actually serve the resulting CSS from said bundle in a Jinja template:

Serving asset bundles via Flask-Assets.

```
{% assets "home_less_bundle" %}
  <link href="{{ ASSET_URL }}" rel="stylesheet" type="text/css">
{% endassets %}
```

The `{% assets $}` block is a Jinja tag added by Flask-Assets to help us serve bundles by their registered name! Instead of memorizing where our static assets output to and hardcoding those file paths, Flask-Assets is able to discern that the `{{ ASSET_URL }}` of `home_less_bundle` is `'dist/css/home.css'`, just as we specified in assets.py

Did You Get All That?

This is clearly easier said than done. In recognition of that fact, I went ahead and created a Github repo that contains the working source of the demo we just covered. The only caveat to running the code below is that I used the BestBuy API to source shitty products for the product pages, so you might need to grab a BestBuy API key to get it going (sorry):

hackersandslackers/flask-blueprint-tutorial

:blue_book: :package: Structure your Flask apps in a scalable and intelligent way using Blueprints. - hackersandslackers/flask-…

hackersandslackers • GitHub

Just to prove this isn't some nonsense, the repo is deployed as a working demo here:

Flask Blueprint Demo

Structure your Flask apps in a scalable and intelligent way using Blueprints.

Flask Blueprint Demo

Definitely cruise the demo to see how blueprints work together while paying attention to the "Blueprint Info" module at the bottom of each page. As always, feel free to steal my source code and use it for yourself on whatever project you might have hastily agreed to. Mi code es su code.

Combining the what we've seen here with your knowledge of Flask's application factory and Flask-Assets should be all you need to start building logical, organized Flask apps. Godspeed, and screw Boeing.

Flask Python Software Development

Todd Birchard's avatar

Todd Birchard

118 Posts Website Twitter

Engineer with an ongoing identity crisis. Breaks everything before learning best practices. Completely normal and emotionally stable.

Dekodernl
1 point
4 months ago

In the section 'Defining a Blueprint' the code in 'admin/routes.py' you add 'from flask import current_app as app ' but it's never used. What does it do?

> Todd Birchard
> 0 points
> 3 months ago
>
> Good question! This isn't covered in the tutorial as it's a bit off-topic, but in the Github repository, we use the `app` object passed into each blueprint to register and build each Blueprint's static assets. Each Blueprint has an **assets.py** file we use to generate Stylesheets and Javascript - these modules look at certain config variables in `app` (such as, if the app is in production, don't bother building static assets).
>
> I suppose one could argue this *is* relevant to Blueprints and deserves to be explored in detail in this post as opposed to a separate section on static assets. I'd love to hear your input; I try to be wary not to dump too much nonsense into a single post, but this could very well be something reasonable to explore here.