Modular Applications with Blueprints

▶ *Changelog*

Flask uses a concept of *blueprints* for making application components and supporting common patterns within an application or across applications. Blueprints can greatly simplify how large applications work and provide a central means for Flask extensions to register operations on applications. A `Blueprint` object works similarly to a `Flask` application object, but it is not actually an application. Rather it is a *blueprint* of how to construct or extend an application.

Why Blueprints?

Blueprints in Flask are intended for these cases:

- Factor an application into a set of blueprints. This is ideal for larger applications; a project could instantiate an application object, initialize several extensions, and register a collection of blueprints.
- Register a blueprint on an application at a URL prefix and/or subdomain. Parameters in the URL prefix/subdomain become common view arguments (with defaults) across all view functions in the blueprint.
- Register a blueprint multiple times on an application with different URL rules.
- Provide template filters, static files, templates, and other utilities through blueprints. A blueprint does not have to implement applications or view functions.
- Register a blueprint on an application for any of these cases when initializing a Flask extension.

A blueprint in Flask is not a pluggable app because it is not actually an application – it's a set of operations which can be registered on an application, even multiple times. Why not have multiple application objects? You can do that (see Application Dispatching), but your applications will have separate configs and will be managed at the WSGI layer.

Blueprints instead provide separation at the Flask level, share application config, and can change an application object as necessary with being registered. The downside is that you cannot unregister a blueprint once an application was created without having to destroy the whole application object.

The Concept of Blueprints

The basic concept of blueprints is that they record operations to execute when registered on an application. Flask associates view functions with blueprints when dispatching requests and generating URLs from one endpoint to another.

My First Blueprint

This is what a very basic blueprint looks like. In this case we want to implement a blueprint that does simple rendering of static templates:

```
from flask import Blueprint, render_template, abort
from jinja2 import TemplateNotFound

simple_page = Blueprint('simple_page', __name__,
                        template_folder='templates')

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template('pages/%s.html' % page)
    except TemplateNotFound:
        abort(404)
```

When you bind a function with the help of the `@simple_page.route` decorator, the blueprint will record the intention of registering the function `show` on the application when it's later registered. Additionally it will prefix the endpoint of the function with the name of the blueprint which was given to the **`Blueprint`** constructor (in this case also `simple_page`). The blueprint's name does not modify the URL, only the endpoint.

Registering Blueprints

So how do you register that blueprint? Like this:

```
from flask import Flask
from yourapplication.simple_page import simple_page

app = Flask(__name__)
app.register_blueprint(simple_page)
```

If you check the rules registered on the application, you will find these:

```
>>> app.url_map
Map([<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
```

```
 <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/' (HEAD, OPTIONS, GET) -> simple_page.show>])
```

The first one is obviously from the application itself for the static files. The other two are for the *show* function of the `simple_page` blueprint. As you can see, they are also prefixed with the name of the blueprint and separated by a dot (`.`).

Blueprints however can also be mounted at different locations:

```
app.register_blueprint(simple_page, url_prefix='/pages')
```

And sure enough, these are the generated rules:

```
>>> app.url_map
Map([<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/pages/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/pages/' (HEAD, OPTIONS, GET) -> simple_page.show>])
```

On top of that you can register blueprints multiple times though not every blueprint might respond properly to that. In fact it depends on how the blueprint is implemented if it can be mounted more than once.

Blueprint Resources

Blueprints can provide resources as well. Sometimes you might want to introduce a blueprint only for the resources it provides.

Blueprint Resource Folder

Like for regular applications, blueprints are considered to be contained in a folder. While multiple blueprints can originate from the same folder, it does not have to be the case and it's usually not recommended.

The folder is inferred from the second argument to **`Blueprint`** which is usually *__name__*. This argument specifies what logical Python module or package corresponds to the blueprint. If it points to an actual Python package that package (which is a folder on the filesystem) is the resource folder. If it's a module, the package the module is contained in will be the resource folder. You can access the **`Blueprint.root_path`** property to see what the resource folder is:

```
>>> simple_page.root_path
'/Users/username/TestProject/yourapplication'
```

To quickly open sources from this folder you can use the **open_resource()** function:

```
with simple_page.open_resource('static/style.css') as f:
    code = f.read()
```

Static Files

A blueprint can expose a folder with static files by providing the path to the folder on the filesystem with the `static_folder` argument. It is either an absolute path or relative to the blueprint's location:

```
admin = Blueprint('admin', __name__, static_folder='static')
```

By default the rightmost part of the path is where it is exposed on the web. This can be changed with the `static_url_path` argument. Because the folder is called `static` here it will be available at the `url_prefix` of the blueprint + `/static`. If the blueprint has the prefix `/admin`, the static URL will be `/admin/static`.

The endpoint is named `blueprint_name.static`. You can generate URLs to it with **url_for()** like you would with the static folder of the application:

```
url_for('admin.static', filename='style.css')
```

However, if the blueprint does not have a `url_prefix`, it is not possible to access the blueprint's static folder. This is because the URL would be `/static` in this case, and the application's `/static` route takes precedence. Unlike template folders, blueprint static folders are not searched if the file does not exist in the application static folder.

Templates

If you want the blueprint to expose templates you can do that by providing the *template_folder* parameter to the **Blueprint** constructor:

```
admin = Blueprint('admin', __name__, template_folder='templates')
```

For static files, the path can be absolute or relative to the blueprint resource folder.

The template folder is added to the search path of templates but with a lower priority than the actual application's template folder. That way you can easily override templates that a blueprint provides in the actual application. This also means that if you don't want a blueprint template to be accidentally overridden, make sure that no other blueprint or actual

application template has the same relative path. When multiple blueprints provide the same relative template path the first blueprint registered takes precedence over the others.

So if you have a blueprint in the folder `yourapplication/admin` and you want to render the template `'admin/index.html'` and you have provided `templates` as a *template_-folder* you will have to create a file like this: `yourapplication/admin/templates/admin/index.html`. The reason for the extra `admin` folder is to avoid getting our template overridden by a template named `index.html` in the actual application template folder.

To further reiterate this: if you have a blueprint named `admin` and you want to render a template called `index.html` which is specific to this blueprint, the best idea is to lay out your templates like this:

```
yourpackage/
    blueprints/
        admin/
            templates/
                admin/
                    index.html
            __init__.py
```

And then when you want to render the template, use `admin/index.html` as the name to look up the template by. If you encounter problems loading the correct templates enable the `EXPLAIN_TEMPLATE_LOADING` config variable which will instruct Flask to print out the steps it goes through to locate templates on every `render_template` call.

Building URLs

If you want to link from one page to another you can use the **`url_for()`** function just like you normally would do just that you prefix the URL endpoint with the name of the blueprint and a dot (`.`):

```
url_for('admin.index')
```

Additionally if you are in a view function of a blueprint or a rendered template and you want to link to another endpoint of the same blueprint, you can use relative redirects by prefixing the endpoint with a dot only:

```
url_for('.index')
```

This will link to `admin.index` for instance in case the current request was dispatched to any other admin blueprint endpoint.

Error Handlers

Blueprints support the `errorhandler` decorator just like the **`Flask`** application object, so it is easy to make Blueprint-specific custom error pages.

Here is an example for a "404 Page Not Found" exception:

```
@simple_page.errorhandler(404)
def page_not_found(e):
    return render_template('pages/404.html')
```

Most errorhandlers will simply work as expected; however, there is a caveat concerning handlers for 404 and 405 exceptions. These errorhandlers are only invoked from an appropriate `raise` statement or a call to `abort` in another of the blueprint's view functions; they are not invoked by, e.g., an invalid URL access. This is because the blueprint does not "own" a certain URL space, so the application instance has no way of knowing which blueprint error handler it should run if given an invalid URL. If you would like to execute different handling strategies for these errors based on URL prefixes, they may be defined at the application level using the `request` proxy object:

```
@app.errorhandler(404)
@app.errorhandler(405)
def _handle_api_error(ex):
    if request.path.startswith('/api/'):
        return jsonify_error(ex)
    else:
        return ex
```

More information on error handling see Custom Error Pages.