CODING NOMADS

10) Scaling Up: Configs, Application Factory, and Blueprints Lesson

Part 3: Flask Blueprints & Tying it all Together

27 min to complete · By Brandon Gigous

Contents

- Introduction
- What is a Flask Blueprint
  - How to Create a Flask Blueprint
  - (Re)Defining the Routes
  - (Re)Defining the Error Handlers and Form
  - Registering the Blueprint
- Application Script
- Inheriting Configuration Classes
  - Development Config
  - Testing Config
  - Production Config
- Connecting It All Together
- (Re)Configuring your Application
- Summary: Flask Blueprints & Tying it All Together

This Flask tutorial is part 3 of 3 in our Flask app configuration series. If you haven't already, check out the other parts below:

1. Flask App Configuration Basics
2. Flask Application Factories

3. Flask Blueprints & Tying it All Together

What is a Flask Blueprint

With the application factory you made above, it's unfortunately become more cumbersome to define routes and even error handlers. You have to define them all in the context of the `create_app()` function. Once the function has finished executing, we can no longer add routes because the app is already initialized.

With your single-file app, the application context existed in the global scope, so you could add a route from almost anywhere. But once you put the app creation within a function, the application context disappears after an app is returned. There *is* actually a way to use the `app.route` decorator to define routes directly inside your `create_app()` function. However, it's not very pretty, and we can do better.

Blueprints are a neat solution to managing routes and other handlers. You can think of a blueprint as a "mini-app". Just like a Flask application, you can define routes to implement the behavior of your application, with view functions that get called whenever certain requests are received.

However, a blueprint will do *nothing* until it is **registered** with an actual Flask application. In this respect, blueprints get "activated" as part of the application once they get registered, and only then. Blueprints are like parts of an application that you can register with different environments, or all environments, if you want to.

Think of a blueprint like a mouse or a keyboard you plug into a computer. They don't work if they aren't plugged in, but once they do and "register" with the computer, they work and give you added functionality.

Blueprints can also implement separate bits of functionality to keep your code clean and organized. For example, you can have one blueprint that implements most of the user-facing pages, another that handles authentication and registration of accounts, and yet another that defines the Application Programming Interface (API) for your app. You can define a blueprint in a single module (single file) or a package.

Opting for blueprints defined in packages makes for better flexibility, so that's what you're gonna do. In fact, you'll make a subpackage inside the `app/` package.

How to Create a Flask Blueprint

If you're following along with the course, the blueprint you're going to create will not be doing anything new. It'll actually just replace all the existing functionality you had before: the view function(s), your `NameForm`, and the error handlers. It'll be called `'main'` because it will define the *main* functionality of your app.

The only thing that *is* new is the fact that it's a Blueprint! If you haven't already, make your `main/` package with an `app/main/__init__.py` file.

Note: You can also refer back to the top of this lesson to review the folder structure we're working with.

Then put this in the `__init__.py` file:

```
from flask import Blueprint

main = Blueprint('main', __name__)

from . import views, errors
```

Blueprint instances in Flask are created with the `Blueprint` class.

Here you call the `Blueprint()` constructor with two arguments, the name of the blueprint and the module in which the blueprint is located. In this case the name you've given is `'main'` and the module is the current module which is accessed with the special dunder variable `__name__`. You then assign this new blueprint to a variable `main`.

The last line then imports the view functions and the error handlers, which you'll move to `app/main/views.py` and `app/main/errors.py`, respectively. The views and error handlers will be part of the main blueprint instead of the global app you had before this section. The reason these imports come *after* the blueprint creation is to prevent errors from cropping up due to circular dependencies. You'll see in a sec why this is.

(Re)Defining the Routes

Let's now move on to `app/main/views.py`. This is where your view functions from the original `app.py` will be moved to, and there's a few differences:

```
from flask import session, render_template, redirect, url_for, flash
from . import main
from .forms import NameForm
from .. import db
from ..models import Role, User

@main.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        # ...
        return redirect(url_for('.index'))
    return render_template(
        'index.html',
        form=form,
        name=session.get('name'),
        known=session.get('known', False))
```

In particular, the `index()` function is no longer decorated by `@app.route()`, but by the `main` blueprint with `@main.route()`. Main was imported with `from . import main`.

The code snippet here shows a bunch of other relative imports, because remember, the `main` blueprint is part of the `main/` package, as well as `forms`. To access the `Role` and `User` models as well as the database, that requires a `..`, to go up a level in the package hierarchy, in order to grab those imports since they exist in the parent `app/` package.

See how the `main` blueprint is imported? In your `__init__.py`, if you had imported `views` and `errors` *before* you made your `main` blueprint, this `from . import main` would have caused you an issue due to circular dependencies.

Note: A circular dependency is when two or more modules depend on each other. This can cause issues when importing modules, as Python will

not know which module to import first and will typically throw an error.

When it comes to using blueprints and referencing their view functions, the `url_for()` endpoint argument needs to be the name of the blueprint, followed by a `.`, followed by the view function name. In the case where you want the url for a view function *in the same blueprint*, the blueprint name is optional.

Okay, that might not have made sense, so let's go a bit slower. Say you have a view function outside the `main` blueprint and you want to redirect to the `'/index'` page defined in the `main` blueprint. To do that, you'd need `redirect(url_for('main.index'))`.

Flask works this way because it allows different blueprints to have the same endpoint names. So you could have a "main" blueprint define an `'/about'` page, and a "second" blueprint that has an `'/about'` page, too. Assuming the standard `route` decoration, the endpoint names would then be `main.about` and `second.about`. These two endpoints can coexist peacefully.

If you haven't seen it yet in the code above, what's different is the `'.index'` instead of `'main.index'`, but they are the same thing! Using just `'index'` instead will not work because it's defined in the "main" blueprint. In that way, it's similar to the importing syntax.

(Re)Defining the Error Handlers and Form

There's not too much difference in how your error handlers will be defined, but there's something important to point out. Go ahead and put this in your `error.py` file:

```
from flask import render_template
from . import main


@main.app_errorhandler(403)
def forbidden(e):
    error_title = "Forbidden"
    error_msg = "You shouldn't be here!"
    return render_template("error.html",
                           error_title=error_title,
```

```
                           error_msg=error_msg), 403


@main.app_errorhandler(404)
def page_not_found(e):
    error_title = "Not Found"
    error_msg = "That page doesn't exist"
    return render_template('error.html',
                           error_title=error_title,
                           error_msg=error_msg), 404


@main.app_errorhandler(500)
def internal_server_error(e):
    error_title = "Internal Server Error"
    error_msg = "Sorry, we seem to be experiencing some technical di
    return render_template("error.html",
                           error_title=error_title,
                           error_msg=error_msg), 500
```

Before, when you had your single-file app — where its routes were defined globally — any 404 or 500 errors would get routed to your error handlers with `page_not_found()` or `internal_server_error()`, respectively. Your error handlers were designated as such by the `errorhandler` decorator.

However, if you make a blueprint and define error handlers for it with the `errorhandler` decorator, those only become error handlers for the routes defined in the *same* blueprint, and not the app as a whole. But fear not, for you can make your error handlers handle errors globally by using the `app_errorhandler` decorator instead, as you can see in the example above.

Finally, the `NameForm` you made before can go in `forms.py`. Then once you do that, the main blueprint is complete! Now you'll have to register it with your application.

Registering the Blueprint

If you jumped the gun too quick and tried to run your app, you would have eventually found that no matter where you go, you get those ugly "Not Found" errors wherever you try to navigate to. Neither your routes or even your error handlers will work!

The reason it's not working is because you still need to **register your blueprint**, and you can do so easily with the aptly named `register_blueprint()` method. Bring up your `app/__init__.py` file and add this to your `create_app()` function.

```
def create_app(config_name):
    # ...

    from .main import main as main_blueprint
    app.register_blueprint(main_blueprint)

    return app
```

Bam! Now whenever you create an application with your application factory, you'll get an application that has all your routes, error handlers, and forms. The `app.register_blueprint()` method takes all that's packaged with the `main` package and treats it as part of itself in a way.

Application Script

The last Python file you haven't touched yet is `ragtime.py`. That's where your Flask application will be born. Despite what your parents may have told you about baby webapps, they aren't just flown over by Tim Berners-Lee.

The `ragtime.py` file is named after the whole application, lives in the root of the project, and is a sort of app runner, which brings together the different pieces of your app and it's configuration. It's also where you can define some tasks to manage the application.

Go ahead and bring `ragtime.py` up in your editor:

```
from app import create_app, db
from app.models import User, Role
from flask_migrate import Migrate

app = create_app('default')
migrate = Migrate(app, db)
```

```
@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User, Role=Role)
```

The first thing this does is make the familiar `app`. Did you miss it? You throw in the `'default'` argument because that's the "name" of your configuration. Next is instantiating the `migrate` object. Since your database `db` is created along with the app, it has to get instantiated after the call to `create_app()`. Then last, you put in your `shell_context_processor` for your `flask shell` sessions.

Note: Now that the filename with your Flask instance in it, `ragtime.py`, is different from the previous `hello.py`, you'll have to update the `FLASK_APP` to use the new name. It's also a swell idea to turn on `FLASK_DEBUG` if you'd like.

Inheriting Configuration Classes

If your app is up and running again, congrats! You'll learn about getting it set up for any kind of configuration now.

As you learned previously, your app will probably need all kinds of sets of configuration. There's the configuration sets you may use for development, for testing, and for production when your app is ready for the world.

That means just having one config class just won't do. To get your app's configuration capabilities to its full potential, and taking advantage of object-oriented programming, you'll **inherit** your `Config` class! Here's what that looks like:

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config():
    SECRET_KEY = os.environ.get('SECRET_KEY') or "keep it secret, ke
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    @staticmethod
    def init_app(app):
        pass


class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_DEV_URL') or
        'sqlite:///' + os.path.join(basedir, 'data-dev.sqlite')


class TestingConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_TEST_URL') or
        'sqlite:///{os.path.join(basedir, "data-test.sqlite")}''
```

```
class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        f'sqlite:///{os.path.join(basedir, "data.sqlite")}'

config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}
```

So what happened? The first thing to notice is that `SQLALCHEMY_DATABASE_URI` is not defined in `Config` anymore, but rather inside the new child classes, `DevelopmentConfig`, `TestingConfig`, and `ProductionConfig`. Now to go over all the different classes:

Development Config

First up is the development configuration. You've basically been using a development configuration this whole time because you've been, well, developing and debugging and such. The database URL you had before is the same as it was in `Config`, but now it's in `DevelopmentConfig`. However, now there's `os.environ.get('DATABASE_DEV_URL')`. Whazzat? The `DATABASE_DEV_URL` is an environment variable just like the `FLASK_APP` environment variable. At least, that's what your code will look for if you have it defined or not. If you don't, it will opt for the database URL you were using before. You'll notice the same sort of thing was done for the `SECRET_KEY` in `Config` and the other database URLs in the other classes.

Note: Just like you did for `FLASK_APP`, you can define any other environment variable with the `export` command (Mac and Linux only).

The next config setting in `DevelopmentConfig` is `DEBUG`. This is basically the same as setting the `FLASK_DEBUG` environment variable. However, it's important to know that any setting for `FLASK_DEBUG` will override the `DEBUG` setting you may have in a configuration. For example, if `DEBUG` is `True` and you set `FLASK_DEBUG` to `0`, Flask debugging mode won't be enabled.

Note: Remember, if you want to use VS Code to debug and set breakpoints with a `launch.json`, it's recommended that you *don't* have Flask's debugging mode enabled!

Testing Config

> *Something that is untested is broken.*
>
> *~Unknown*

That comes from Flask's "Testing Flask Applications" page, so it seems like the developers think it's pretty important to test your Flask apps! That's why you'll have a configuration solely for running tests on your app. The `TESTING` configuration flag will disable error catching during request handling, so it's good for getting juicier error reports so you can fix things quicker. If there's no environment variable set for the database, it'll default to no database. The reasoning behind this is that you can theoretically have multiple tests that tests multiple different things, including different *database* setups. Having no database for testing would tell you you didn't set one up when you ran your tests!

Production Config

And now for the `ProductionConfig`, which is used for when you're deploying your app. You'll get to that point soon enough, but this config will be waiting for you once you do. Not much new here, just a different database URL defined.

Connecting It All Together

If you still have your `app.py` file, you can move it's contents to `ragtime.py`, the new "entry point" of your app or just rename it `ragtime.py`. Inside this file should be everything you didn't move, but make sure it looks something like this:

```
from flask_migrate import Migrate
from app import create_app, db
from app.models import Role, User

app = create_app('default')
migrate = Migrate(app, db, render_as_batch=True)
```

```
@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User, Role=Role)
```

As you can see, you now get `app` from the application factory `create_app()` you made earlier. You also need to import the `Role` and `User` models to resolve the other errors your IDE may report.

(Re)Configuring your Application

The last part of `config.py` is the `config` dictionary, and some more names have been added here. The `'default'` name is now `DevelopmentConfig`, and `Config` is no longer used.

Let's go back to `ragtime.py` to make a quick changes in regards to the config. You can change it so that it uses a new `FLASK_CONFIG` environment variable to define the name of the config to use (from the `config` dictionary):

```
app = create_app(os.getenv('FLASK_CONFIG') or 'default')
```

Sweet. There shouldn't be anything else needed, except to set all your environment variables! In fact, you could make a bash script that you can "source" in order to set these environment variables automatically:

```
export FLASK_APP=ragtime.py
export FLASK_DEBUG=1
export FLASK_CONFIG=development
```

You can call it something like `development_settings.sh`, then run it with:

```
. development_settings.sh
```

Every time you're ready to develop.

You can also change your VS Code launch file to include this launch configuration so that you can change your app's configuration and run the app easily:

```
"configurations": [
{
"name": "Python: Ragtime",
"type": "python",
"request": "launch",
"module": "ragtime",
"env": {
"FLASK_APP": "ragtime.py",
"FLASK_ENV": "development",
"FLASK_CONFIG": "development",
"FLASK_DEBUG": "0"
},
}
]
```

Register Here

Summary: Flask Blueprints & Tying it All Together

In this lesson, you:

- Got hands-on with **Flask Blueprints**, enabling you to compartmentalize features and simplify route management.
- **Registered** blueprints with your Flask application to activate them.
- Incrementally built an **application script** (`ragtime.py`) to instantiate and configure a Flask application, forming a clear entry point for the app.

Remember to push your code changes regularly and test each new component to verify it operates as intended before proceeding. Your Flask projects are becoming more sophisticated and closer to what you'd see in real-world, team-centric development environments!

Previous Next → Video: Blueprints