CODING NOMADS

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

Part 2: Flask Application Factories

13 min to complete · By Brandon Gigous

Contents

This Flask tutorial is part 2 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

Flask Application Factories

Flask Application factories are, in simple terms, software or code that produce different objects depending on certain input. In practice, this allows you to quickly create different Flask app configurations by using the factory method pattern:

> *In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor. Wikipedia*

Wow that's a mouthful. The gist of it is that you'll be able to create different configurations for your app by calling a method that will create an app with the configuration you specify.

Why do we need Application Factories?

Why would we want to create more than one Flask app? You have a working Flask app already! You can just keep making the same one over and over, right?

Yes, you may want to eventually deliver to the public a web app that pretty much matches what you've been developing. But, hold up! Let's break down that last sentence. Let's consider these "two" apps:

- The one that would be available to a user on the Internet
- The one that you are developing

These apps are to be the same... but different! They'll need a different database, for instance. You don't want your public app to rely on your laptop database server! They'll also need different security settings, different logging settings, and so on. The point is, the app you develop on your local machine will not be in the same *environment* as the app that will be deployed to the public on a server in another state or even another country.

Much of the confusion can stem from the work "app". Often we think of the app as the code we write — as the whole project, essentially. However, within the context of a Flask project, the "app" is the `Flask` *instance* you create in your code. It's the thing that you run with `flask run`. It's *created* anew every time.

So, your app needs to have different *Configurations*. The most fundamental different configurations that almost all apps have are **development** and **production** environments.

The app you develop on your *local* machine will not be in the same environment as the same app that will be deployed to the public on a server in another state or even another country. The database configuration will not be the same; the secret key will not and certainly *shouldn't* be the same; there will even be things you haven't even *imagined* yet that will need to be configured for production.

The worst part about all this configuration stuff? *You have to keep it straight*. The best part? You can *A U T O M A T E* it! And that's where **application factories** come in. Here's how it will work:

> *You: I want an app for development purposes! Application Factory: Okay sure! Here's a development flavored app (an app configured for development).*
>
> *You: Ready to deploy this app to the world. Give me an app for production! Application Factory: Okay sure! Here's a production flavored app (an app configured for production).*

The `create_app()` Function

To get your super cool application factory spitting out applications like a pro, you'll have to move much of your original `app.py` script to the new `__init__.py` file in your `app/` folder. So go do that if you haven't already.

Then, you're ready to start your `create_app()` function! Your `__init__.py` would looking something like this:

```
from flask import Flask
from flask_bootstrap import Bootstrap
from flask_sqlalchemy import SQLAlchemy
import os

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

db = SQLAlchemy()
bootstrap = Bootstrap()
```

```
"""
Database Models...
LoginManager User Loader...
"""

def create_app(config_name='default'):
    app = Flask(__name__)

    app.config['SECRET_KEY'] = "keep it secret, keep it safe"
    app.config['SQLALCHEMY_DATABASE_URI'] =\
        'sqlite:///' + os.path.join(basedir, 'data-dev.sqlite')
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

    bootstrap.init_app(app)
    db.init_app(app)

    @app.route('/'):
    def index():
        # Code for index route...

    """
    Your other routes and error handlers here...
    """

    return app
```

This `create_app()` function contains code that is not much different from what you've been doing already. First, it creates a `Flask` instance. Then, it sets a few configuration options. Finally, it initializes the extensions and defines your routes.

Let's consider what is different from your single-file Flask app.

Routes Inside A Function

The routes may look weird inside the `create_app()` function, but functions within functions is totally okay in Python. In the next section, you'll get clean them up, so don't worry too much about them now. They are still routes that function in the same way as before.

Extension Initialization

The two extensions that are defined don't get initialized right away. Did you notice there's no application instance to hand to them at first? That's okay, though, because they can be initialized later with their own `init_app()` method once an application is created. You'll still want them in the global scope, though, so you can import them from other modules if needed.

Application Returned By A Function

Now for the most significant difference of all: now your app is *returned* by a function.

The `create_app()` function is actually something that the `flask run` command looks for and calls if it can't find a `Flask` instance in the file, or module specified by a `FLASK_APP` environment variable.

This new `create_app()` function is a factory function, otherwise known as your **application factory**!

If you or the Flask framework call the function, out pops an app. But, surely there's a question on your mind. What's that `config_name` doing as a parameter? Well, for now, just passing in a default value, but that will soon change.

The idea is, you can give the `create_app()` function the name of a configuration you'd like to use, and it'll give you a brand spankin' new Flask application instance with configuration settings suited for the kind of app you specify. Right now, you only have one *set* of configuration settings, the one your app has been setting up to this point.

Learn by Doing

Exercise! Create An Application Factory

Before moving onto the next lesson, let's make sure you understand how application factories actually work with a quick exercise. The app in this Repl.It exercise has *something* for an application factory. Looks like it needs a little help to become the application factory it was meant to be!

app_factory @gigabot

Show files Open on Replit

README.md

Background

Your friend Julia learned about application factories in Flask. However, it looks like either she misunderstood how they work, or her learning material did not teach her how to do it right... Time for you to step in.

Her app currently doesn't seem to work. In trying to fix it herself, she might have made it a bit worse. Oops. Since you've learned the basics of application factories, it's your job to help her!

Instructions

Modifying *only* the `__init__.py` file in the `my_app_pkg` application package, get it into a working state. The file contains an attempt at creating an application factory. Use what you've learned to get the application factory working properly.

You will notice the app shows an error message. In making the necessary changes, you will probably see that a few other errors show up. Do your best to give it what it wants.

Helpful Notes

To the left is a button called "Files" where you can see all the files for this simple project. For this exercise, *don't* move any of the files around. You should be able to complete it without doing so.

Note: Please don't move, rename, delete, or change the `.replit` file! Otherwise you may not be able to complete the assignment.

Info: To complete this exercise, you'll need to make an account on Replit. Afterwards, sign into your account, click "open in replit" here in the lesson, and finally click "fork repl."

Revisit Your Application Factories

Once you complete the exercise and have a better grasp on how application factories work, try to get your own app to work in the same way. The code above should help. Once you get it working, you're ready to continue.

Summary: Flask Application Factories

In this lesson, you explored Flask **Application Factories**, which allow you to create Flask app instances with specific configurations based on the environment (development, testing, production).

Cool, now you have an application factory or two pumping out apps. How does it feel? Coming up, you'll see how to flavor your apps with different configurations using Flask Blueprint, and more! Forward!