CODING NOMADS

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

Part 1: Flask App Configuration & Project Structure

13 min to complete · By Brandon Gigous

Contents

This Flask tutorial is part 1 of 3 in our Flask app configuration series. In this lesson, you'll learn how to organize your Flask project into a more modular structure.

After finishing this lesson, make sure you continue on to the rest of the series! These lessons combined teach you how to use these features to make your project more organized and easier to maintain:

1. Flask App Configuration Basics
2. Flask Application Factories
3. Flask Blueprints & Tying it All Together

Flask App Configuration Features

Having one single Python file for your web application can be really nice and straightforward. It's very convenient and goes to show how flexible and powerful Flask can be. But as the project grows, it may become harder to navigate, extend and maintain.

Flask has your back though! One of the things people love about Flask is that it lets you organize your project in a way that makes sense for you, your project and your team. It has a few features that can help you organize your project better. The main features for high level project organization are:

- Configurations
- Blueprints
- Application Factories

Why go Modular?

Modularizing your project can make things easier to:

- **Maintain**: You can find things more easily, and you can make changes without worrying about breaking something else.
- **Scale**: As your project grows, you can add new features and functionality without worrying so much about how it will affect the rest of the project.
- **Collaborate**: You can work on different parts of the project without stepping on each other's toes.
- **Test**: You can isolate different parts of the application for precise testing.
- **Understand**: You can see how different parts of the project fit together.
- **Debug**: You can find and fix bugs more easily because it's easier to understand.
- **Document**: You can write nicely scoped documentation for the different modules without having to write about everything all the time.

Keeping organized is generally a good thing, but you also need to be wary of over-engineering. If you're working on a small project, you might not need to worry about these things. But in this course we'll want to over-engineer a bit to get a feel for things you'd probably have to deal with when working with a team.

To see these features in action, we'll want to start off with a project structure that lends itself to these features. So, let's start by examining the structure of the example project.

If you're following along with this course and building out the example project step-by-step (as you should be!), now would be a good time to commit your changes thus far, so you can easily revert back to that point if you need to.

Flask File and Folder Structure

Let's create a project structure that will make it easier to see how configurations, application factories and blueprints work. Here's a basic structure for a typical Flask project:

```
flask-webdev/
├───app/
│    ├── main/
│    │    ├── __init__.py
│    │    ├── errors.py
│    │    ├── forms.py
│    │    └── views.py
│    ├── static/
│    ├── templates/
│    │    ├── base.html
│    │    ├── error.html
│    │    ├── index.html
│    │    └── user.html
│    ├── __init__.py
│    └── models.py
├── env/
├── migrations/
├── config.py
├── ragtime.py
└── requirements.txt
```

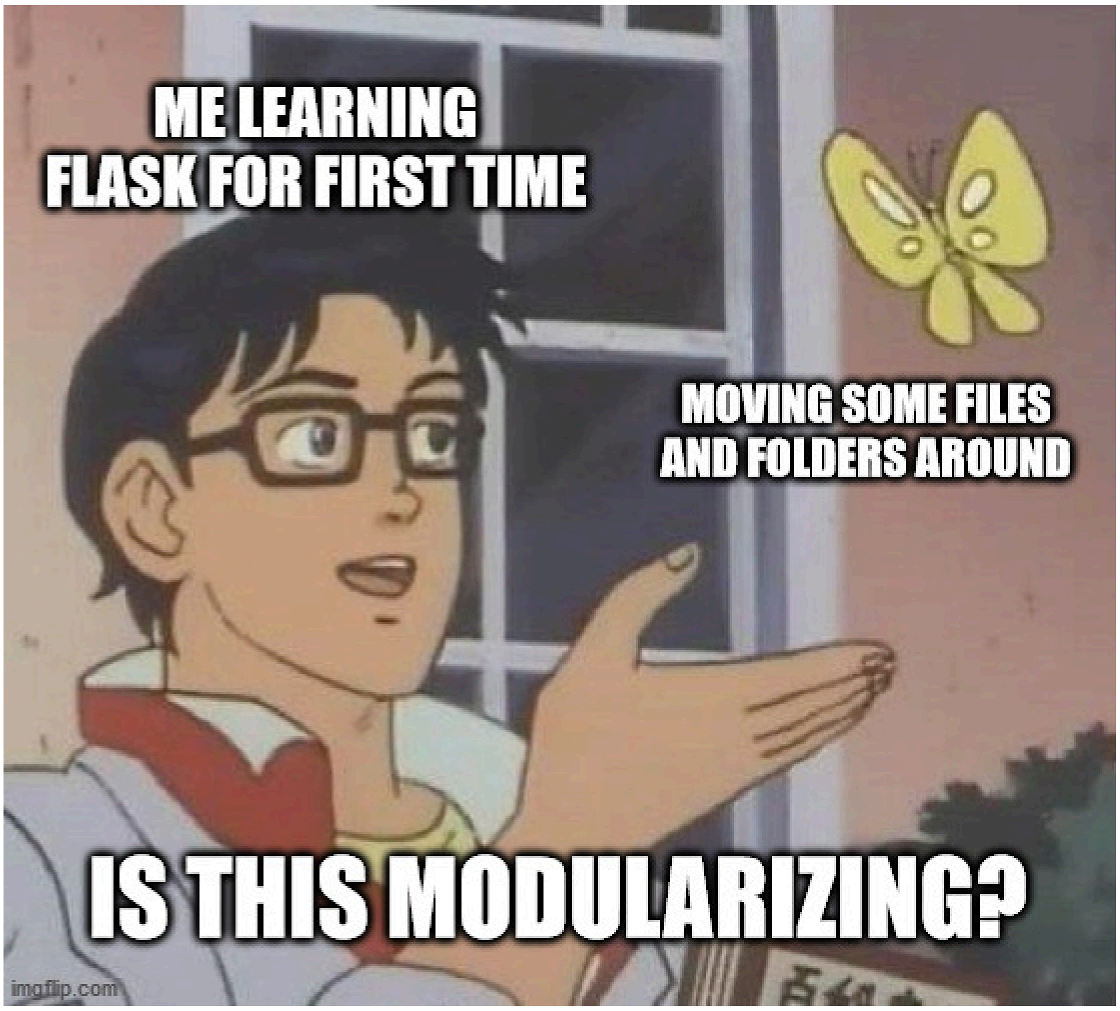
Make sure your project looks like this before moving on. If you didn't have the file before, just create a blank file for now. You'll fill it in later.

Let's review the main parts of this structure:

- `app/` : A package for your application! The package is made by creating a new folder and an `__init__.py` file inside. This is where the to-be-created Flask application will live.
- `migrations/` : Just like before, this contains the database migration scripts.
- `env/` : Your virtual environment.
- `config.py` : This is where your configuration settings will live, and contain some configuration classes. They were foretold about a bit in the last lesson.
- `ragtime.py` : This is where you'll make an instance of your Flask application. It'll also contain a few helper functions that define tasks to manage the application
- `requirements.txt` : Think of this file as a courtesy to your fellow programmer peers, or even yourself. If you or someone else wants to *regenerate* your own virtual environment, they can do so with this. It lists the package dependencies of your project.

The `__init__.py` files are there to tell Python that the directories are *packages*. They can be empty, but they can also execute initialization code for the package. For now, you don't need to worry about them, just know that they serve a purpose, even if they are blank.

Great! While organizing the files in this way is a good start, we can take things much further!

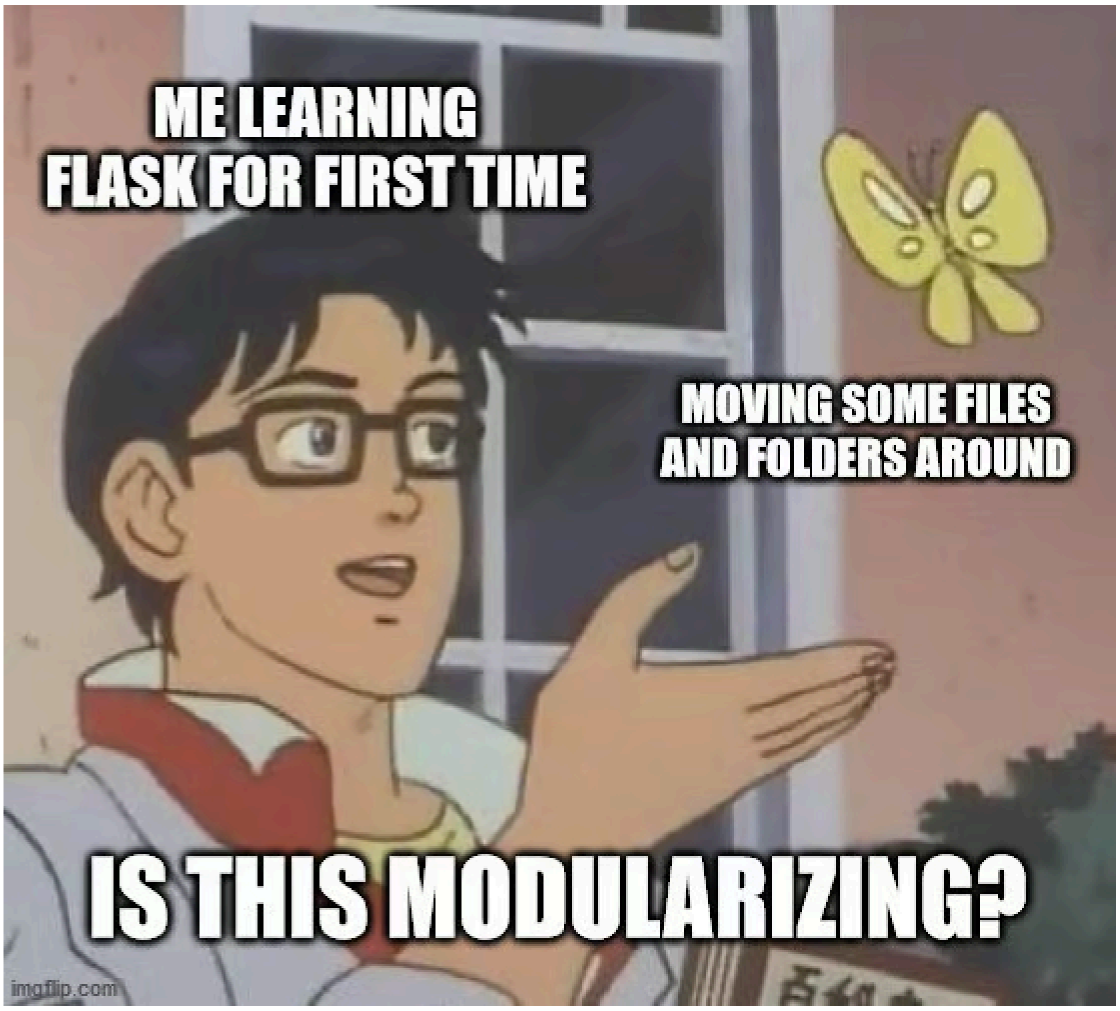

ME LEARNING FLASK FOR FIRST TIME
MOVING SOME FILES AND FOLDERS AROUND
IS THIS MODULARIZING?
imgflip.com

Creating a Flask `Config` Class

Once you get your project organized with the new structure, it's time to define the Flask app's configuration. The configuration is where you can manage different settings for your app. This can include things like:

- Database settings
- Security settings
- Debugging settings
- Logging settings
- And more!

These types of settings are one of the first things to get modularized in any project.


ME LEARNING FLASK FOR FIRST TIME
MOVING SOME FILES AND FOLDERS AROUND
IS THIS MODULARIZING?
imgflip.com

Flask allows you to load a configuration as a class, which means you can create a class and define constant variables to pass your settings to the app. Put this in your `config.py` file:

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

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

    @staticmethod
    def init_app(app):
        pass

config = {'default': Config}
```

You'll notice that it's not much different than the `app.config` settings you defined in your single-file Flask app. The difference is that you are using a separate class instead of defining them directly.

Note the static `init_app()` method. You'll see in the next section that `init_app()` is called for some Flask extensions, and you can treat your configuration class as if it were an extension itself. The `Config` class isn't an extension, though. The `init_app()` method is just a Flask convention for initialization, which it expects to be there for things like extensions or configuration classes. Extensions and configuration classes can use the `init_app()` method to do any setup that requires the app to be fully configured.

Reminder: A static method is a method that doesn't require an instance of the class to be called. It's a method that belongs to the class, not the instance.

A neat thing about having your configuration as a class is that you can *extend* it now. So, if you wanted a configuration that's *slightly* different than one you have, you can just use the original as a base class and change what you need. This will come into play when you want to have different configurations for development, testing, and production. This is also where application factories come in, but you'll get to that in a bit.

The last thing here is a dictionary called `config`. While there's only one configuration class so far to make things easy, you'll add more classes to the `config.py` file later. This dictionary is just a way of "naming" these classes so they can be referenced more easily. So, the `'default'` configuration refers to the new `Config` class.

If that doesn't quite make sense in terms of why a config class may have it's own initialization, or why there's a dictionary, not to worry. You'll learn more about this shortly.

Summary: Flask App Configuration & Project Structure

In this lesson, you:

- Recognized the limitations of a single-file Flask application and why **modularization** is useful as projects grow.
- Reviewed the typical Flask file and folder structure.

- Discovered how to define **configuration classes** with in a separate `config.py` file.

Coming up, you'll see how you'll use this class to get your app configured using application factories!

Previous Next → Video: Flask Project Structuring for Optimization and...