undefined Logo

Demystifying Flask's Application Factory

Todd Flask 6 Min Read

A skill that continues to fascinate me is how far some developers can get in their careers while sidestepping a basic understanding for what they do. I might be a poster child for this phenomenon myself, as I've somehow managed to fool you (and thousands of others)

into thinking I'm qualified enough to write trustworthy tutorials. Thanks for reading, by the way.

I bring up blissful ignorance because of how upsetting I find most Flask tutorials. I'm mostly referring to how Flask is commonly introduced as a super simple framework where all source code and business logic belongs in a single file called **app.py**. If you've ever come across a Flask tutorial with a file named **app.py**, I'm willing to bet it looked something like this:

The wrong way to build Flask apps.

```
from flask import Flask, render_template

app = Flask(__name__)
app.config['FLASK_DEBUG'] = True
app.config['STATIC_FOLDER'] = '/static'
app.config['TEMPLATES_FOLDER'] = '/templates'


@app.route("/")
def home():
    return render_template("index.html")


@app.route("/about")
def about():
    return render_template("about.html")


@app.route("/contact")
def contact():
    return render_template("contact.html")
```

It's easy to think there's nothing wrong with the example above. It's readable and works fine for an app with three pages... yet, therein lies the rub: nobody aspires to learn a web framework for the purpose of building a three page app. You're going to add more pages... a lot more. You're going to want to pull data from a database. You may even want to handle forms and user logins. If we built an app where all that logic lived in a file called **app.py**, our app would be as much of a *monster* as it would be a *monument to our own ineptitude*. Uploading this type of abomination to a public Github repo is a great way to ensure you never get hired.

The problem is undoubtedly compounded by Flask's official documentation itself. Instead of guiding users towards best practices, Flask's documentation reads like a distracted child attempting to explain everything in the universe. The Flask community puts heavy

emphasis on getting started fast, but fails to mention that this "microframework" can easily scale to build apps as large and feature-heavy as any app built with Django.

Flask's "Application Factory" and "Application Context"

Well-structured web apps separate logic between files and modules, typically with *separation of concerns* in mind. This seems tricky with Flask at first glance because our app depends on an "app object" that we create via `app = Flask(__name__)`. Separating logic between modules means we'd be importing this app object all over the place, eventually leading to problems like circular imports. The **Flask Application Factory** refers to a common "pattern" for solving this dilemma.

The reason why the *Application Factory* is so important has to do with something called Flask's **Application Context**. Our app's "context" is what takes the assortment of Python files and modules which make up our app and brings them together so that they see and work with one another. For Flask to recognize the data models we have in `models.py`, our Blueprints, or anything else, Flask needs to be told that these things exist after the app is "created" with `app = Flask(__name__)`.

Here's Flask's take on Application factories:

> *If you are already using packages and blueprints for your application (Modular Applications with Blueprints) there are a couple of really nice ways to further improve the experience. A common pattern is creating the application object when the blueprint is imported.*

And here's their description of the Application context:

> *The application context keeps track of the application-level data during a request, CLI command, or other activity. Rather than passing the application around to each function, the current_app and g proxies are accessed instead.*

That's not entirely useful if we don't understand what the terminology is referring to. Let's fix that.

Top-Down View of an Application Factory App

Are you the type of person to start an app by first creating an **app.py** file in our base directory? If so, please stop - this simply *isn't* a realistic way to build production-ready applications. When we create an app which follows the Application Factory pattern, our app should look like this:

The **Flask Application Factory** pattern.

```
/app
├── /application
│   ├── __init__.py
│   ├── auth.py
│   ├── forms.py
│   ├── models.py
│   ├── routes.py
│   ├── /static
│   │   ├── /dist
│   │   └── /src
│   └── /templates
├── config.py
├── start.sh
└── wsgi.py
```

Notice there's no **app.py**, **main.py**, or anything of the sort in our base directory. Instead, the entirety of our app lives in the **/application** folder, with the creation of our app happening in **__init__.py**. The init file is where we actually create what's called the Application Factory.

If you're wondering how we deploy an app where the main entry point isn't in the root directory, I'm very proud of you. Yes, our app is being *created* in **application/__init__.py,** so a file called **wsgi.py** simply imports this file to serve as our app gateway. More on that another time.

Initiating Flask in __init__.py

Let's dig into it! We're going to create an example Flask application which utilizes some common patterns, such as connecting to a database utilizing a Redis store. This is is simply for demonstration purposes to show how all globally-accessible Flask plugins are initialized in a standard Flask app.

A properly configured Application Factory should accomplish the following:

- Create a Flask `app` object, which derives configuration values (either from a Python class, a config file, or environment variables).
- Initialize plugins accessible to any part of our app, such as a database (via `flask_sqlalchemy`), Redis (via `flask_redis`) or user authentication (via `Flask-Login`).
- Import the logic which makes up our app (such as routes).
- Register Blueprints.

The below example does all of those things:

application/__init__.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_redis import FlaskRedis


# Globally accessible libraries
db = SQLAlchemy()
r = FlaskRedis()

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

    # Initialize Plugins
    db.init_app(app)
    r.init_app(app)

    with app.app_context():
        # Include our Routes
        from . import routes

        # Register Blueprints
        app.register_blueprint(auth.auth_bp)
        app.register_blueprint(admin.admin_bp)

        return app
```

The order of operations here is critical. Let's break it down.

Step 1: Creating Instances of Plugin Objects

The vast majority of what we do happens in a function called `create_app()`. Before we even get to creating our app, we create global instances of `flask_sqlalchemy` and `flask_redis`. Even though we've set these, nothing has happened until we "initialize" these plugins after our app object is created.

Initializing plugins in __init__.py.

```
...
# Globally accessible libraries
db = SQLAlchemy()
r = FlaskRedis()
...
```

Step 2: App Creation

The first two lines of `create_app()` should be no surprise: we're creating our Flask app object and stating that it should be configured using a class called **Config** in a file named **config.py**:

Starting our Flask app in `create_app()`

```
...

def create_app():
    """Initialize the core application."""
    app = Flask(__name__, instance_relative_config=False)
    app.config.from_object('config.Config')
```

Step 3: Plugin Initialization

After the app object is created, we then "initialize" those plugins we mentioned earlier. Initializing a plugin registers a plugin with our Flask app. Setting plugins as global variables outside of `create_app()` makes them globally accessible to other parts of our application, we can't actually use them until after they're initialized to our app:

Initialize our plugins.

```
...

def create_app():
    ...

    # Initialize Plugins
    db.init_app(app)
    r.init_app(app)
```

Step 4: The Application Context

Now comes the moment of truth: **creating the app context.** What happens in the app context, stays in the app context... but seriously. Any part of our app which is not imported, initialized, or registered within the `with app.app_context():` block effectively *does not exist*. This block is the lifeblood of our Flask app - it's essentially saying "here are all the pieces of my program."

The first thing we do inside the context is import the base parts of our app (any Python files or logic which aren't Blueprints). I've imported **routes.py** via `from . import routes`. This file contains the route for my app's home page, so now visiting the app will actually resolve to something.

Next, we register Blueprints. Blueprints are "registered" by calling `register_blueprint()` on our `app` object and passing the Blueprint module's name, followed by the name of our Blueprint. For example, in `app.register_blueprint(auth.auth_bp)`, I have a file called **auth.py** which contains a Blueprint named **auth_bp**.

We finally wrap things up with `return app`.

Wrapping up `create_app()`.

```
...

def create_app():
    ...

    with app.app_context():
        # Include our Routes
        from . import routes

        # Register Blueprints
        app.register_blueprint(auth.auth_bp)
        app.register_blueprint(admin.admin_bp)

        return app
```

Our App's Entry Point

So our function returns `app`, but *what* are we returning to, exactly? That's where the mysterious **wsgi.py** file from earlier comes in. Almost every **wsgi.py** file looks like this:

wsgi.py

```
from application import create_app

app = create_app()

if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

Ah ha! Now we have a file in our app's root directory that can serve as our entry point. When setting up a production web server to point to your app, you will almost always configure it to point to **wsgi.py**, which in turn imports and starts our entire app.

Flask Python Software Development

Todd Birchard

118 Posts Website Twitter

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

ADD A COMMENT

thon-poetry-package-ager	**Package Python Projects the Proper Way with Poetry** Python, Software Development
omate-ssh-scp-python-niko	**SSH & SCP in Python with Paramiko** Python, Automation
-python-apis-in-two-minutes-really-just-e-of-code-to-change	**Faster Python APIs in Two Minutes** Python, REST APIs