Structuring a Large Production Flask Application

Arash Soheili
Jan 11 · 5 min read ★

Flask Python

In the world of Python web frameworks, Flask and Django are considered the two major options. Flask is a micro-framework and is generally easier to get started building apps. Django is a batteries included framework and comes pre-loaded with almost everything you need to get started. Having used both I'm a big fan of Flask and it's what we've been using at our work for the last 6 years to build our web applications. I won't get into the details of Flask vs Django here but you can learn more about them and other Python frameworks here.

Although Flask is a fantastic framework, one of the disadvantages is it doesn't provide project structure or design patterns. That's great when you want to get started quickly but eventually in a larger application that becomes an issue. When building a large production application that should be maintained by current and future developers, it is of paramount importance to have built-in structure and to follow a clear design pattern. Here I will discuss my learnings in designing a Flask application that hopefully

can help others. As a disclaimer, our Flask app is mainly api based although we do serve a few routes like login via Flask template.

Here is an outline of the Flask application structure:

```
~/YourApplication
    |-- requirements.txt
    |-- config.py
    |-- wsgi.py                # wsgi server
    |__ .env                   # Virtual Environment
    |__ /app                   # Our Application package
         |-- __init__.py       # Application factory method
         |-- errors.py         # Custom error classes
         |-- db.py             # Database connection
         |-- utils.py          # Utility helper functions
         |-- services.py       # Business logic
         |-- daos.py           # Database access object
         |-- models.py         # Database models
         |-- security.py       # Security access checks
         |__ /routes           # Routes package (Blueprints)
             |-- __init__.py
             |-- users.py
         |__ /templates
         |__ /static
    |__ ..
    |__ .
```

In the root directory, I have placed the virtual environment directory as a hidden `.env` folder. I prefer to keep the virtual env within the application folder rather than in a common directory. This keeps a common `.env` naming convention for all projects and I don't need to find where I have the virtual env folder. I also put a global ignore of `.env` in my global git configuration as well to keep it out of version control.

Application Factory: The easiest way to get started with Flask is to create an instance of the Flask class and setup configuration on the instance. The app instances can then be used for routing and registering middleware as is shown in the official docs:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

But this pattern quickly breaks down in larger Flask applications as there will be several environments like staging, dev, and testing. You will need to have different setup and configuration on the app instances which becomes very hard to manage in this method.

Instead, a better way is to use an application factory which is a function that returns an instance of the app. Having this function allows you to pass in different parameters to set up your app. This decouples your app configuration and setup from the creation of the app itself. Now in your dev or testing environment you can pass in different configuration settings.

app/__init__.py

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    return app
```

When running in production you would then use a different file to import the factory function and create your app instance and pass to gunicorn or your choice of wsgi web server. The application factory method also works well with Blueprint which we will cover next.

wsgi.py

```
from app import create_app
config = "config.Production"

application = create_app(config) # call from wsgi server
```

Blueprints: Flask routing is setup by using the application instance as decorator on your route functions. This forces you to declare all the routes in the same file as the app instance. This is not maintainable in a larger flask app with many routes.

The way to break up routes into separate files is to use Flask blueprints. If your Flask app is api based then you can match files names to resources. Therefore, you would have a *user.py* file in a routes directory that contains all routes related to the user and

this can be extended to other resources. Blueprints are a great way to organize your routes and a must use for large applications.

Services/DAO Singleton Design Pattern: A pattern I have found that works very well in a Flask app is the services/dao (database access object) singleton pattern. The summary of this pattern is outlined below:

- Business logic is placed in services methods.
- Database access (ORM) is in dao methods which use models.
- Routes are kept light and either use service or dao methods.

To demonstrate this pattern I will use an example for getting all users.

Declare your user model (this is based on sqlalchemy ORM).

app/models.py

```
class User:
    id = Column(Integer, primary_key=True)
    username = Column(String(80), unique=True, nullable=False)
    email = Column(String(120), unique=True, nullable=False)
```

You would then declare the UserDao class with methods for interacting with the user model in the database. We create a singleton object for the dao instance that can be used in the whole app.

app/daos.py

```
from app.models import User
from app.db import session

class UserDAO:
    def __init__(self, model):
        self.model = model

    def get_all(self):
        return session.query(self.model).all()

    def get_by_username(self, username -> str):
        return (
            session.query(self.model)
            .filter_by(get_by_username=username)
```

```
            .first()
        )

    def get_by_email(self, email -> str):
        return (
            session.query(self.model)
            .filter_by(email=email)
            .first()
        )

user_dao = UserDAO(User)
```

Finally define the get all user route in your blueprint route.

app/routes/users.py

```
from flask import Blueprint, jsonfiy
from app.daos import user_dao

user_bp = Blueprint('user_bp', __name__)

@user_bp.route("/api/users", methods=["GET])
def get_users():
    return jsonify(user_dao.get_all())
```

In any case where you need business logic for the user then it would be created in services as shown. Let's say for example you want certain logic when updating the user model. Like in the dao module a singleton object would be declared and used in the entire app

app/services.py

```
from app.daos import user_dao

class UserService:
    def update_user(self, user_id, data -> dict):
        user = user_dao.get(user_id)
        # Update user with data
        # check user for business logic
        # example: lowercase all emails
        return user

user_service = UserService()
```

You could then use this service in your blueprint route as shown here.

app/routes/users.py

```
from flask import Blueprint, jsonfiy, request
from app.services import user_service

user_bp = Blueprint('user_bp', __name__)

@user_bp.route("/api/users", methods=["GET])
def get_users():
    return jsonify(user_dao.get_all())

@user_bp.route("/api/users/<int:user_id>", methods=["POST"])
def update_user(user_id):
    data = request.json
    user = user_service.update_user(user_id, data)
    return jsonify(user)
```

In cases where any of your models.py, services.py or daos.py files get too large, you can convert them to a package with smaller modules. How you define the smaller modules will depend on the type of application and your own use case.

We have used this design pattern in production for the last 6 years on our production app and it has served us well. I plan to expand in more details about other aspects of running a production Flask application like configuration management, third party api integration, background tasks, docker setup and more. Be sure to follow if you want to get all the articles.

Python Flask Application Design Patterns Design Thinking