Apr 21, 2018 / Hsiaoming Yang

Structure of a Flask Project

Flask is very flexible, it has no certain pattern of a project folder structure. Here is my suggestions.

Flask itself is very flexible. It has no certain pattern for a project folder structure, which is very good for experienced developers to organize things in their own favors. However, people new to Flask will get confused, they need some guide on it, and usually they are going to find something works but not good (or even bad).

I didn't know such a problem until someone reported an issue to Authlib. And I can't understand the problem either. Then another person explained it to me with a project structure, I finally got it. I was terrified that lots of posts, guide, boilerplates are backward importing modules from project root `__init__.py`:

```
# project/__init__.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

def create_app():
    app = Flask(__name__)
    db.init_app(app)

# project/auth/models.py
from .. import db

class User(db.Model):
    # define columns
```

The code itself will work, but when your projects grow, sooner or later you will face a cyclic dependencies problem. For instance, another extension requires to init with the `User` model:

```
# project/__init__.py
from flask_sqlalchemy import SQLAlchemy
from another_extension import AnotherExtension
from project.auth.models import User

db = SQLAlchemy()
ext = AnotherExtension(User)
```

Oops, a cyclic dependency occurs. Because `auth.models` is importing `db` from the root, root can not import `User` module. This is a common cyclic problem, not limited to Flask. It is easy to fix, but junior developers may find it very hard. So why not avoid such thing from the very begining? Actually, if you have read the **official** documentation, in application factories you can find this piece of code:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)
    from yourapplication.model import db
    db.init_app(app)
    from yourapplication.views.admin import admin
    from yourapplication.views.frontend import frontend
    app.register_blueprint(admin)
    app.register_blueprint(frontend)
    return app
```

See, we put `db` in `yourapplication.model`.

I always keep this one certain rule when writing modules and packages:

> Don't backward import from root `__init__.py`.

That's why I submitted a ticket to Flask as soon as I found this problem. People need a guide on folder structure. And here I'm going to share my suggestions. But think it yourself, don't treat mine as a golden rule.

Functional Based Structure

There are many ways to setup your project folder structure. One is by its function. For instance:

```
project/
  __init__.py
  models/
    __init__.py
    users.py
    posts.py
    ...
  routes/
    __init__.py
    home.py
    account.py
    dashboard.py
    ...
  templates/
    base.html
    post.html
    ...
  services/
    __init__.py
    google.py
    mail.py
    ...
```

All things are grouped by its function. If it hehaves as a model, put it in models folder; if it behaves as a route, put it in routes folder. Build a `create_app` factory in `project/__init__.py`, and init_app of everything:

```python
# project/__init__.py
from flask import Flask

def create_app()
    from . import models, routes, services
    app = Flask(__name__)
    models.init_app(app)
    routes.init_app(app)
    services.init_app(app)
    return app
```

Here is a trick by me. In official documentation, `db` of Flask-SQLAlchemy is registered in this way:

```
from project.models import db
db.init_app(app)
```

So my trick is define a `init_app` in every folder's `__init__.py`, and unify the init progress as one:

```
# project/models/__init__.py
from .base import db

def init_app(app):
    db.init_app(app)

# project/routes/__init__.py
from .users import user_bp
from .posts import posts_bp
# ...

def init_app(app):
    app.register_blueprint(user_bp)
    app.register_blueprint(posts_bp)

# ...
```

App Based Structure

Another famous folder structure is app based structure, which means things are grouped bp application. For instance:

```
project/
  __init__.py
  db.py
  auth/
    __init__.py
    route.py
    models.py
    templates/
  blog/
    __init__.py
    route.py
    models.py
    templates/
...
```

Each folder is an application. This pattern is used by default in Django. It doesn't mean this pattern is better, you need to choose a folder structure depending on your project. And sometime, you will have to use a mixed pattern.

It is the same as above, we can `init_app` as:

```
# project/__init__.py
from flask import Flask

def create_app()
    from . import db, auth, blog
    app = Flask(__name__)
    db.init_app(app)
    auth.init_app(app)
    blog.init_app(app)
    return app
```

Configuration

Loading configuration would be another issue that many people find difficult, it is also a folder structure problem. I don't know how other people are doing, I'm just sharing my solution.

1. Put a `settings.py` in project folder, treat it as static configuration.
2. Load configration from environment variable.
3. Update configration within `create_app`.

Here is a basic folder structure for configration:

```
conf/
  dev_config.py
  test_config.py
project/
```

```
  __init__.py
  settings.py
app.py
```

Define a `create_app` to load settings and environment variable:

```python
# project/__init__.py
import os
from flask import Flask

def create_app(config=None)
    app = Flask(__name__)
    # load default configuration
    app.config.from_object('project.settings')
    # load environment configuration
    if 'FLASK_CONF' in os.environ:
        app.config.from_envvar('FLASK_CONF')
    # load app sepcified configuration
    if config is not None:
        if isinstance(config, dict):
            app.config.update(config)
        elif config.endswith('.py'):
            app.config.from_pyfile(config)
    return app
```

This `FLASK_CONF` is a python file path which contains configrations. It can be any name you want, e.g. your project is called Expanse, you can name it as `EXPANSE_CONF`.

I use this `FLASK_CONF` to load production configurations.

* * *

Again, Flask is very flexible, there is no certain patterns. You can always find your favors. These are just my suggestions, **do not blind by anyone**.

I don't like to write posts like this. But there are so many wrong guide, I hope this post can get a better SEO, so that bad posts don't mislead people.

ENJOY

#Flask