Flask 后端项目结构

项目结构

项目的根目录结构

```
├── api
├── docker-compose.yml
├── Dockerfile
├── Makefile
├── requirements.txt
├── setup.py
└── tests
```

自动化部署

`docker-compose.yml` 和 `Dockerfile` 用于容器化部署. 具体来说:

- `Dockerfile` 描述了本项目的后端镜像搭建过程, 例如: 使用什么基础镜像, 安装了哪些依赖, 运行时命令等.
- `docker-compose.yml` 进一步描述了整个项目需要构建的服务, 如 web 后端服务(本 flask 项目)、其它服务(例如 OJ 的评测机, 前端 node.js 项目), 还能包括数据库、缓存等.

`Makefile` 定义了项目常用的指令, 如初始化数据库

```
db-init:
  docker-compose exec web flask db init
```

它会在构建的 docker image 里调用 `flask db init`.

`requirements.txt` 记录了项目依赖的 Python 包.

杂项

`setup.py` 用于打包成模块. 如果你的项目是一个可以被安装的包(例如你需要写一个 python wrapper, 或者发布到 pip), 这个文件定义了包的元数据和安装配置.

`tests` 目录下存放测试用例, 我们通常使用 `pytest` 来编写测试代码和运行测试.

`.gitignore`, `.dockerignore` 用于定义项目的忽略规则, 防止不必要的文件被提交到版本管理/Docker 复制.

项目应当有 `README.md` 文件, 用于描述项目的功能和使用方法. 对于一个开源项目, 如果你的贡献者较多, 可以考虑添加 `CONTRIBUTING.md` 文件, 用于描述如何参与项目的开发(code of conduct).

另外, 根目录还可以存放 `docs` 文档. 项目成熟后可以专门部署一个 doc 页面.

项目核心

```
├── api          # APIs
├── commons      # or utils
├── models       # ORM, data model
├── config.py
├── extensions.py
├── manage.py
├── __init__.py
└── wsgi.py
```

插件项

我们提到 Flask 是一个后端服务器的骨架, 如果你要使用数据库 ORM, 登录认证 JWT, 数据库迁移 migrate[1], 就需要注册相关的插件.

你可以专门在 `extensions.py` 文件里注册插件:

```
from flask_sqlalchemy import SQLAlchemy
from flask_jwt_extended import JWTManager
from flask_marshmallow import Marshmallow
from flask_migrate import Migrate

db = SQLAlchemy()
jwt = JWTManager()
ma = Marshmallow()
migrate = Migrate()
```

后续使用只需要 import 该文件即可:

```
from api.extensions import db
```

通用工具

`commons` 下会存放一些常用的工具, 例如分页函数, 加密解密函数, API 结果类等. 它有时名称是 `utils`.

当这个工具变大的时候, 可以考虑拆分成多个模块. 例如验证模块 `auth` 单独拆分.

数据模型

`models` 存放 ORM 映射的对象.

应用工厂

所谓应用工厂就是将 Flask 应用实例的创建过程封装在一个函数中, 例如 `create_app(config : Config)`.

`wsgi.py` 定义了 Flask 应用的入口. 它会调用 `create_app` 函数来创建应用实例.

- 使用工厂函数能够让我们在不同的环境（如开发环境、测试环境和生产环境）中使用不同的配置.
- `pytest` 的 `fixture` 里可以直接调用 `create_app(test_config)` 来生成一个测试环境下的应用实例.
- 本地运行代码的时候, `create_app(default_config)` 就是开发环境, `flask run` 即可.
- 部署到开发环境里时, `create_app(production_config)` 来生成生产环境下的应用实例.

不同配置

`config.py` 是项目的配置文件.

创建一个配置的基类. 然后在不同的环境中继承该类并修改配置.

```
class Config:
    DEBUG = False
    TESTING = False
    DATABASE_URI = 'sqlite:///:memory:'

class DevelopmentConfig(Config):
    DEBUG = True
```

[1]我们在 Django 里喜欢称其为 middleware, 中间件.

```
class TestingConfig(Config):
    TESTING = True

class ProductionConfig(Config):
    # better use environment variables for production config
    DATABASE_URI = 'mysql://user@localhost/foo'
```

我们再重申一遍: **千万不要把机密的配置信息硬编码进代码中**, 如果你这么做了, 就不要把它纳入版本管理的代码中!

通常我们的惯例是, 配置一个字典来存储配置对象.

```
config = {
    'development' : DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}
```

在创建应用实例时, 利用 `app.config.from_object` 导入配置对象:

```
def create_app(config=config['default']):
    app = Flask(__name__)

    app.config.from_object(config)

    db.init_app(app)
    # ...

    return app
```

注册蓝图

蓝图能在 Flask 应用层面隔离，共享应用配置，并在注册时按需修改应用对象。

下面这个例子使用蓝图来渲染静态模板:

```
from flask import Blueprint, render_template, abort
from jinja2 import TemplateNotFound

simple_page = Blueprint(name='simple_page', import_name=__name__,
                        template_folder='templates')

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template(f'pages/{page}.html')
    except TemplateNotFound:
        abort(404)
```

- 定义了一个 `simple_page` 的蓝图, `import_name` 设为 `__name__` 为了能正确加载模板文件.
- 声明该蓝图下的模板文件夹为 `templates`.

在使用应用工厂的情况下，为了让视图绑定到指定的应用实例下, 我们需要在创建应用的函数 `create_app` 中注册蓝图:

```
app = Flask(__name__)
app.register_blueprint(simple_page)
```

```
>>> app.url_map
Map([<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/' (HEAD, OPTIONS, GET) -> simple_page.show>])
```

我们还可以让蓝图挂载到不同的位置, 即为它添加一段 URL 前缀:

```
app.register_blueprint(simple_page, url_prefix='/pages')
```

```
>>> app.url_map
Map([<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/pages/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/pages/' (HEAD, OPTIONS, GET) -> simple_page.show>])
```

这也可以在声明蓝图时指定. 在写 API 的蓝图时很有用:

```
blueprint = Blueprint("api", __name__, url_prefix="/api/v1")
```

API 下存放核心的 API 蓝图. Flask-RESTful 规范的一种文件结构如下:

```
├── __init__.py
├── resources
│   ├── __init__.py
│   └── user.py
├── schemas
│   ├── __init__.py
│   └── user.py
└── views.py
```

REST 架构把 HTTP 请求方法 + API 端点抽象成获取、操作资源 `resource`. 例如设计 API 端点的时候, 把 `GET /users` 视为获取用户列表的资源.

```
from flask_restful import Resource
from api.api.schemas import UserDetailSchema

class UserList(Resource):
    def get(self):
        schema = UserDetailSchema(many=True)
        query = User.query
        return paginate(query, schema)
```

`views` 里定义了 API 的蓝图, 并创建视图函数, 绑定到 `resources` 里的资源.

```
from flask_restful import Api

api.add_resource(UserResource, "/users/<int:user_id>", endpoint="user_by_id")
api.add_resource(UserList, "/users", endpoint="users")

@blueprint.before_app_first_request
def register_views():
    # used APISpec
    apispec.spec.components.schema("UserDetailSchema", schema=UserDetailSchema)
    apispec.spec.path(view=UserResource, app=current_app)
    apispec.spec.path(view=UserList, app=current_app)
```

处理请求会利用 `schema` 来反序列化 request/序列化 response[2].

例如获取用户详情, 返回前端会有不需要的字段(如密码), 需要定义 `UserDetailSchema`.

[2]你可以把它想象成 Django DRF 框架里的 serializer/deserializer, 或者是 Java Spring 框架里各种的 VO/BO.

```
from api.models import User
from api.extensions import ma, db

class UserSchema(ma.SQLAlchemyAutoSchema):

    id = ma.Int(dump_only=True)
    password = ma.String(load_only=True, required=True)

    class Meta:
        model = User
        sqla_session = db.session
        load_instance = True
        exclude = ("_password",)
```

构建项目模板

类似于 IDE 帮助你创建项目模板. 我们会使用到一个命令行工具 `cookiecutter`:

```
pip install cookiecutter
cookiecutter <template-git-url>
```

Flask 的一些项目模板

- cookiecutter-flask
- cookiecutter-flask-restful: restful api 项目模板