1. 作业1 - 报告

软件03 陈启乾 202012385

1.1. 要求

- 简单描述单元测试部分用例的设计思路、实现思路、测试用例列表及覆盖率并进行分析；
- 简要描述集成测试和端到端测试部分的实现思路；
- 简要描述 Docker 部署部分的实现思路以及体会；

1.2. 代码风格测试

编写 Flake8 配置文件即可。

```
[flake8]
# 忽略且仅忽略 .git，所有 __pycache__ 文件夹，所有 migrations 文件夹
exclude =
    .git,
    __pycache__,
    migrations
```

忽略文件需要使用 `exclude` 命令。

```
# 对 tests/test_e2e.py 忽略E501错误，对 tests/test_api.py 忽略E501错误，对 driver.py 忽略
E501错误，对 app/settings.py 忽略E501错误，对 user/views.py 忽略E722错误，对 app/
settings_prod.py 忽略F401和F403错误
per-file-ignores =
    tests/test_e2e.py: E501,
    tests/test_api.py: E501,
    driver.py: E501,
    app/settings.py: E501,
    user/views.py: E722,
    app/settings_prod.py: F401,F403
```

忽略个别错误需要使用 `per-file-ignores` 命令。

在 lint.sh 文件中：

```
autopep8 --in-place --recursive .
```

使用 `autopep8` 对代码自动格式化。

```
autoflake --in-place --recursive --remove-unused-variables .
```

使用 autoflake 对代码自动格式化。

```
isort .
```

使用 isort 对代码自动格式化。

```
flake8
```

使用 flake8 检查代码风格。

1.3. 单元测试

单元测试包括两部分：基础函数的补全和单元测试的编写。

1.3.1. 基础函数的补全: `register_params_check`

1. 针对必填项，会首先判断是否存在该项目，不存在则直接报错；对于可选项，不存在时则会填充默认值。
2. 其次会判断内容的类型、长度等简单的信息

3. 最后利用正则表达式判断所有其他限制。对于较为复杂的情况例如域名，也会结合 Python 语句先进行分割。

1.3.2. 测试函数

针对每一个分支判断的判断的错误的输入，我们都编写了一个对应的测试函数，函数名就反映了测试的范围。

总体样例：

```
class BasicTestCase(TestCase):

    def test_username_missing(self):···
    def test_username_invalid_too_short(self):···
    def test_username_invalid_layout(self):···
    def test_password_missing(self):···
    def test_password_invalid_too_short(self):···
    def test_password_invalid(self):···
    def test_mobile_missing(self):···
    def test_mobile_wrong_type(self):···
    def test_mobile_invalid(self):···
    def test_nickname_missing(self):···
    def test_nickname_wrong_type(self):···
    def test_url_missing(self):···
    def test_url_wrong_type(self):···
    def test_url_invalid_protocol(self):···
    def test_url_invalid_domain(self):···
    def test_url_invalid_tag(self):···
    def test_magic_number_invalid_type(self):···
    def test_magic_number_missing(self):···
    def test_magic_number_invalid(self):···
    def test_all_valid(self):···
    def test_magic_number_missing(self):···
```

具体实现：

```
def test_username_missing(self):
    content = {
        "password": "Abc12345*",
        "nickname": "test",
        "mobile": "+86.123456789012",
        "url": "https://www.google.com",
        "magic_number": 0
    }
    result = register_params_check(content)
    self.assertEqual(result, ("username", False))
```

例如在这个例子中，我们想要测试没有用户名的情况下程序的输出，因此我们就构造了一个没有用户名的用例。

测试用例见下页：

username	password	nickname	mobile	url	magic_number	备注
	Abc12345*	test	+86.123456789012	https://www.google.com	0	没有用户名
a123	Abc12345*	test	+86.123456789012	https://www.google.com	0	用户名过短
123abc123	Abc12345*	test	+86.123456789012	https://www.google.com	0	用户名错误布局
abc12345		test	+86.123456789012	https://www.google.com	0	密码缺失
abc12345	Abc124*	test	+86.123456789012	https://www.google.com	0	密码太短
abc12345	abc12345	test	+86.123456789012	https://www.google.com	0	密码太简单
abc12345	Abc12345*	test		https://www.google.com	0	手机缺失
abc12345	Abc12345*	test	123456789012	https://www.google.com	0	手机类型错误
abc12345	Abc12345*	test	+86.12345678901	https://www.google.com	0	手机格式错误
abc12345	Abc12345*		+86.123456789012	https://www.google.com	0	昵称缺失
abc12345	Abc12345*	123	+86.123456789012	https://www.google.com	0	昵称错误的类型
abc12345	Abc12345*	test	+86.123456789012		0	URL缺失
abc12345	Abc12345*	test	+86.123456789012	ftp://www.google.com	0	URL格式错误
abc12345	Abc12345*	test	+86.123456789012	https://www.google.123	0	URL domain 错误
abc12345	Abc12345*	test	+86.123456789012	https://www.google.-com	0	URL tag 错误
abc12345	Abc12345*	test	+86.123456789012	https://www.google.com		Magic Number缺失
abc12345	Abc12345*	test	+86.123456789012	https://www.google.com	-1	Magic Number错误
abc12345	Abc12345*	test	+86.123456789012	https://www.google.com	0	所有都符合

1.3.3. 覆盖率

运行：`python manage.py test --filter test_basic`

```
----------------------------------------------------------------------
Ran 20 tests in 0.059s

OK


Coverage Report:

Name                                Stmts   Miss Branch BrPart  Cover
----------------------------------------------------------------------
post\controllers.py                    98     86     18      0    10%
post\urls.py                            3      0      0      0   100%
post\views.py                          98     82     44      0    11%
user\controllers.py                    32     26      6      0    16%
user\urls.py                            3      0      0      0   100%
user\views.py                          62     48     18      0    18%
utils\jwt.py                           47     33     10      0    25%
utils\post_params_check.py              2      1      0      0    50%
utils\register_params_check.py         48      0     38      0   100%
utils\reply_post_params_check.py        2      1      0      0    50%
----------------------------------------------------------------------
TOTAL                                 395    277    134      0    29%
```

1.4. 集成测试

在 `tests/test_api.py` 中完成集成测试。

1.4.1. 整体思路

用请求模拟用户的错误的操作。需要手动记录登录成功后的 `jwt` token 并传作 Header 的 `Authorization` 字段。

```
data = { ... }  # 构造错误数据
response = self.client.patch(
    reverse(user_views.login),
    data=data,
    content_type="application/json"
) # 发送请求
json_data = response.json()
# 判断返回值
self.assertEqual(response.status_code, 401) # 1. 如果失败
self.assertEqual(response.status_code, 200) # 2. 如果成功
self.assertEqual(json_data['message'], "Invalid credentials") # 失败 / 成功信息
```

1.4.2. 登录功能

1. 使用错误的信息进行登录，检查返回值为失败
    - 错误的用户名
    - 错误的密码
2. 使用正确的信息进行登录，检查返回值为成功
3. 进行登出，检查返回值为成功

1.4.3. 注册功能

1. 使用错误信息进行注册，检查返回值为失败

2. 使用正确的的信息进行注册，检查返回值为成功
3. 使用正确的信息进行登录，检查返回值为成功

1.4.4. 登出功能

1. 未登录直接登出，检查返回值为失败

1.4.5. 测试结果

```
(SimpleBBS) PS C:\Users\chenqq\Downloads\code\backend> python manage.py test --filter test_api
>>
test_login (test_api.APITestCase)
使用错误的信息进行登录，检查返回值为失败 ... Unauthorized: /api/v1/login
Unauthorized: /api/v1/login
now: 2023-10-15 22:06:28.680140
expiry: 2023-10-16 22:06:28.680140
ok
test_logout (test_api.APITestCase)
未登录直接登出 ... Unauthorized: /api/v1/logout
ok
test_register (test_api.APITestCase)
Example: 使用错误信息进行注册，检查返回值为失败 ... Bad Request: /api/v1/register
now: 2023-10-15 22:06:29.080895
expiry: 2023-10-16 22:06:29.080895
ok

----------------------------------------------------------------------
Ran 3 tests in 0.827s

OK
```

1.5. 端到端测试

端到端测试集成了 `selenium` 进行自动化操作。

只需将 `DRIVER_PATH` 修改为 `drivers/chromedriver` 即可。

运行 `python manage.py test --filter test_e2e` 即可。

1.6. Docker 部署部分

1.6.1. 主体要求

创建了两个网络：`inner` 和 `front`，用来隔离 Mysql 和 Nginx。

1.6.2. 后端 Django 服务

Django 服务使用 DockerFile 构建，Docker Compose 集成。

Dockerfile 包括如下内容：

1. 从 `python:3.8-buster` 开始构建
2. 升级 `pip`，根据 `requirements.txt` 使用 `pip` 安装依赖
3. 复制当前文件夹内所有的文件到 `/app/` 下
4. 暴露 `8000` 端口
5. 定义环境变量：`ENV DJANGO_SETTINGS_MODULE app.settings_prod` 采用部署设置
6. 进行 migrate：`CMD python manage.py migrate --settings=app.settings_prod`
7. 运行 `Gunicorn` 服务器：`CMD gunicorn -w4 -b 0.0.0.0:8000 --log-level=debug app.wsgi`

Docker Compose 中包括如下内容：

1. 容器名为 `app`
2. 从当前文件夹构建 `build: .`
3. 依赖 mysql 服务：`depends_on: mysql`

4. 使用 `inner` 和 `front` 网络，可以同时被 `Nginx` 访问和访问 `Mysql` 服务

1.6.3. Mysql 服务器

Mysql 服务器在 Docker Compose 中启动。

1. 使用 `mysql:8.1` 镜像
2. 容器名为 `mysql`
3. 配置环境变量如下:
   - `MYSQL_ROOT_PASSWORD=2020012385` 密码为学号
   - `MYSQL_DATABASE=thss` 数据库名为
   - `TZ=Asia/Shanghai` 时区为背景时间
4. 持久化存储，将 `/home/ubuntu/mysql` 目录映射到 `/var/lib/mysql` 下
5. 启动命令：`mysqld --character-set-server=utf8mb4 --collation-server=utf8mb4_unicode_ci`
6. 使用 `inner` 网络，只有后端 Django 服务可以访问到

1.6.4. Nginx 反向代理

Nginx 反向代理在 Docker Compose 中启动

1. 使用 `nginx:latest` 镜像
2. 容器名为 `nginx`
3. 依赖后端 Django 服务: `depends_on: backend`
4. 持久化存储:
   - `./nginx/app.conf:/etc/nginx/conf.d/default.conf` 配置文件
   - `./build/:/opt/build/` 生成的静态文件映射到 `/opt/build`
5. 使用 `front` 网络
6. 向外暴露 8000 端口

关于 Nginx 配置文件:

```
server {
    # 暴露 8000 端口
    listen 8000;
    server_name localhost;

    root /opt/build;

    # 处理静态部分
    location / {
        try_files $uri $uri/ @router;
        index index.html;
    }

    # 后端
    location /api/v1 {
        proxy_pass http://backend:8000/api/v1;
    }
}
```

1.6.5. 体会

Docker 很好地隔绝了各个服务，让我们可以做到解耦之后专注于某个服务本身的构建。