

## Prev's Blog

[개발 포스트](#)[잡생각](#)[About](#)

# 높은 품질의 Flask 웹 애플리케이션 설계하기

10월 22일, 2018

#백엔드 #데브옵스

*Flask + MySQL + SQLAlchemy + Docker + PyTest + CircleCI 풀 스택으로 웹  
애플리케이션 만들기*

1. Docker 이미지 정하기
2. Flask 애플리케이션 만들기
3. PyTest를 이용하여 테스트 코드 작성하기
4. MySQL 연결과 SQLAlchemy 사용하기
5. 테스트 코드에서 데이터베이스 초기화 자동화하기
6. CI 서비스 연동하기

## 개요

이제 테스트 코드의 중요성은 굳이 말할 필요가 없는 시기가 왔습니다. 도커의 장점도 마찬가지구요. 이제는 안정성이 매우 중요한 큰 기업들뿐만 아니라 작은 스타트업까지도 코드 품질을 높이기 위한 시도를 적극적으로 합니다. 저도 최근의 프로젝트들은 이런 기술들을 적극 도입하여 높은 품질로 만들려 노력하고 있는데요, 위 기술들을 모두 사용하려는데 한 번에 정리된 포스트가 없어 이번 기회에 작성하게 되었습니다.

## 기술 스택

- 프레임워크: **Flask**
- 데이터베이스: **MySQL**

- ORM: **SQLAlchemy**
- 테스트 프레임워크: **PyTest**
- CI: **CircleCI**
- 패키징: **Docker**

Python으로 웹 서버를 구축한다면 프레임워크 선택지는 아마 Flask 아니면 Django 일 것입니다. 이 포스트에서는 **프레임워크**로 Flask를 사용합니다. **데이터베이스**에도 여러 선택지가 있겠지만 가장 대중적인 MySQL을 선택하기로 했습니다. 나머지 선택지도 모두 가장 대중적인 기술을 사용했습니다. SQLAlchemy, PyTest는 각각 Python 스택에서 가장 많이 사용하는 **ORM**(Object-relational mapping)과 **테스트 프레임워크**입니다. 마지막으로 **CI 서비스**로는 대부분 CircleCI 혹은 TravisCI를 사용하는데, 이번 포스트에서는 CircleCI를 사용하기로 했습니다.

## 1. Docker 이미지 정하기



패키징은 마지막에 하는 것이 옳을 수도 있겠지만, 어떤 도커 이미지를 사용하는냐에 따라서 프로젝트 구조가 달라질 수 있기에 가장 먼저 결정해야 하는 사항이라 생각합니다. Docker에 대한 설명은 인터넷상에 워낙 좋은 글들이 많아서 생략합니다. [이 블로그](#)에도 설명이 잘 되어 있습니다.

Flask 앱을 도커화 하기 위해서는 `Dockerfile`을 만들어야 합니다. 가장 일반적으로 생각할 수 있는 방법은 python 공식 이미지를 베이스로 하여 `CMD` 명령어로 python 파일을 실행하는 것을 떠올릴 수 있습니다.

```
1 FROM python:3.6
2 WORKDIR /app
3
4 COPY requirements.txt /app
5 RUN pip install -r requirements.txt
6
7 COPY . /app
8 EXPOSE 5000
9
```

```
10  CMD python main.py
```

docker-python1.Dockerfile hosted with ❤ by GitHub

[view raw](#)

위는 가장 간단하게 떠올릴 수 있는 Dockerfile 구성이며, 실제로 구글링을 하면 가장 쉽게 발견되는 예제이기도 합니다. 하지만 이 방법은 그리 좋은 방법은 아닙니다.

Docker를 사용하지 않고 일반적으로 python 웹 애플리케이션을 배포(deploy) 하기 위해서는 보통 **Nginx + uWSGI** 조합을 사용합니다. 위처럼 `python` 명령어로 파일을 바로 실행하는 일은 개발용 서버를 열 때 사용하는 방법입니다. Flask 내에서 테스트 또는 디버그를 위해서 제공되는 기능으로 안정성을 보장할 수 없고, 성능 문제 또한 있을 수 있죠. Docker로 배포를 한다 할지라도 uWSGI, 혹은 이에 걸맞은 안정된 인터페이스를 사용해야 합니다.

이를 위해서는 Docker 이미지 내에 **uWSGI**를 포함하여야 합니다. **Nginx**는 상황에 따라 이미지 내에 포함시킬 수도 있고, 외부에 빼서 사용할 수도 있습니다. 위 설정이 귀찮으신 분들을 위해 tiangolo라는 개발자가 만든 이미지를 사용할 수도 있습니다. `tiangolo/uwsgi-nginx-flask` 라는 베이스 이미지를 사용하면 uWSGI와 Nginx 기반 환경에서 Flask 애플리케이션이 실행되도록 이미지를 간단하게 구성할 수 있습니다. (위는 비공식 이미지입니다. 보안이 중요한 상황에서는 직접 구성하는 것이 나을 수 있습니다.)

```
1  FROM tiangolo/uwsgi-nginx-flask:python3.6
2
3  COPY requirements.txt /tmp/
4  RUN pip install -r /tmp/requirements.txt && \
5      rm /tmp/requirements.txt
6
7  COPY ./app /app
```

docker-python2.Dockerfile hosted with ❤ by GitHub

[view raw](#)

`EXPOSE`나 `CMD` 설정은 베이스 이미지에서 모두 설정이 되어 있으므로, 라이브러리 설치 및 Python 코드 복사만 실행해주면 패키징을 위한 설정이 완료됩니다.

## 2. Flask 애플리케이션 만들기



# Flask

web development,  
one drop at a time

그럼 이제 위 이미지를 활용하여 Flask 애플리케이션을 만들어 봅시다. 위 `Dockerfile`에서 `./app` 폴더를 복사하도록 설정하였는데, 우선 이에 맞춰서 전체 소스코드도 `app` 폴더 내에 위치하도록 해야 합니다. 또한 `tiangolo/uwsgi-nginx-flask` 이미지에서 기본으로 entry point로 삼는 파일 이름이 `main.py`인데, 이에 따라 `app` 폴더 하위에 `main.py` 파일을 만들고, 이를 기본 파일로 해주어야 합니다.

당장은 Flask 라이브러리만 필요하기에 `requirements.txt` 파일은 아래처럼 작성합니다. (*`requirements.txt`는 python에서 의존성이 있는 라이브러리를 명시하기 위해 사용하는 파일입니다. `pip install -r requirements.txt` 명령어를 이용하여 라이브러리를 설치할 수 있습니다.*)

```
# requirements.txt
```

```
Flask>=1.0
```

`app/main.py`는 root로 접속했을 때 Hello World를 출력하도록 아래처럼 작성해 봅시다.

```
# app/main.py
```

```
from flask import Flask
```

```
app = Flask(__name__)
```

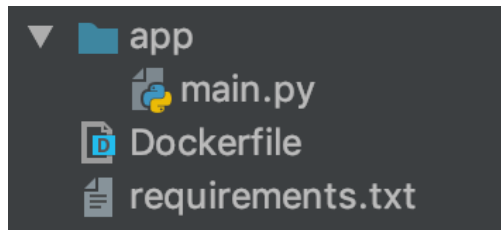
```
@app.route('/', methods=['GET'])
```

```
def index():
```

```
    return 'Hello World!'
```

```
if __name__ == '__main__':
```

```
    app.run(host='0.0.0.0', debug=True, port=8080)\
```



아마 초기 프로젝트 구조는 이렇게 될 것입니다.

코드를 작성했으면 Docker 이미지로 빌드하고, 정상적으로 동작하는지 확인을 해 봅시다. 아래는 `my_flask_app` 이라는 태그로 현재 폴더를 Docker 이미지로 빌드 하는 명령어 입니다.

```
$ docker build -t my_flask_app .
```

```
1. Prev@Prevui-MacBook-Pro-5: ~/Code/python/full-stack-flask-app (zsh)
(venv36) ~/Code/python/full-stack-flask-app ➤ 5e43a62 docker build -t my_flask_app .
Sending build context to Docker daemon 199.7kB
Step 1/4 : FROM tiangolo/uwsgi-nginx-flask:python3.6
--> 839b35a9f270
Step 2/4 : COPY requirements.txt /tmp/
--> 4155b1f2d5d8
Step 3/4 : RUN pip install -r /tmp/requirements.txt && rm /tmp/requirements.txt
--> Running in 2ca9925cd763
Requirement already satisfied: Flask<=1.0 in /usr/local/lib/python3.6/site-packages (from -r /tmp/requirements.txt (line 1)) (1.0.2)
Requirement already satisfied: Werkzeug<=0.14 in /usr/local/lib/python3.6/site-packages (from Flask<=1.0->-r /tmp/requirements.txt (line 1)) (0.14.1)
Requirement already satisfied: itsdangerous<=0.24 in /usr/local/lib/python3.6/site-packages (from Flask<=1.0->-r /tmp/requirements.txt (line 1)) (0.24)
Requirement already satisfied: click<=5.1 in /usr/local/lib/python3.6/site-packages (from Flask<=1.0->-r /tmp/requirements.txt (line 1)) (6.7)
Requirement already satisfied: Jinja2<=2.10 in /usr/local/lib/python3.6/site-packages (from Flask<=1.0->-r /tmp/requirements.txt (line 1)) (2.10)
Requirement already satisfied: MarkupSafe<=0.23 in /usr/local/lib/python3.6/site-packages (from Jinja2<=2.10->Flask<=1.0->-r /tmp/requirements.txt (line 1)) (1.0)
You are using pip version 10.0.1, however version 18.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Removing intermediate container 2ca9925cd763
--> a20d5b9643a1
Step 4/4 : COPY ./app /app
--> 1f239d34d223
Successfully built 1f239d34d223
Successfully tagged my_flask_app:latest
(venv36) ~/Code/python/full-stack-flask-app ➤ 5e43a62
```

다음에는 아래 명령어를 입력하여 컨테이너를 실행해 봅시다. 중요한 점은 `-p` 옵션을 주어 포트를 매핑해주는 것인데, 위 `tiangolo/uwsgi-nginx-flask` 이미지에 서 기본 포트로 80번을 expose 하기에 `원하는포트:80` 로 옵션을 주어야 합니다. 아래 명령어에서는 8080번을 사용하였습니다. `--rm` 옵션은 컨테이너가 종료되었을 때, 자동으로 삭제되도록 하는 옵션입니다. 죽은 컨테이너가 쌓이는 것을 방지하기 위해 일회성으로 사용되는 컨테이너에는 `--rm` 옵션을 붙여 주는 것이 좋습니다.

```
$ docker run -p 8080:80 --rm my_flask_app
```

Docker 컨테이너가 활성화되고 로딩이 완료된 후, `localhost:8080`에 접속하면 Hello World가 출력되는 것을 확인할 수 있습니다. 터미널에서 로그를 확인해보면 Nginx Access Log 또한 정상적으로 출력되는 것을 볼 수 있습니다.

```

clock source: unix
pcre jit disabled
detected number of CPU cores: 2
current working directory: /app
detected binary path: /usr/local/bin/uwsgi
your memory page size is 4096 bytes
detected max file descriptor number: 1048576
lock engine: pthread robust mutexes
thunder lock: disabled (you can enable it with --thunder-lock)
uwsgi socket 0 bound to UNIX address /tmp/uwsgi.sock fd 3
uwsgi running as root, you can use --uid/--gid/--chroot options
*** WARNING: you are running uwsgi as root !!! (use the --uid flag) ***
Python version: 3.6.5 (default, Jun 6 2018, 19:19:24) [GCC 6.3.0 20170516]
*** Python threads support is disabled. You can enable it with --enable-threads ***
Python main interpreter initialized at 0x55dbe2916f50
uwsgi running as root, you can use --uid/--gid/--chroot options
*** WARNING: you are running uwsgi as root !!! (use the --uid flag) ***
your server socket listen backlog is limited to 100 connections
your mercy for graceful operations on workers is 60 seconds
mapped 1239640 bytes (1210 KB) for 16 cores
*** Operational MODE: preforking ***
WSGI app 0 (mountpoint='') ready in 0 seconds on interpreter 0x55dbe2916f50 pid: 11 (default app)
uwsgi running as root, you can use --uid/--gid/--chroot options
*** WARNING: you are running uwsgi as root !!! (use the --uid flag) ***
*** uwsgi is running in multiple interpreter mode ***
spawned uwsgi master process (pid: 11)
spawned uwsgi worker 1 (pid: 14, cores: 1)
spawned uwsgi worker 2 (pid: 15, cores: 1)
running "unix signal:15 gracefully kill them all" (master-start)...
2018-10-20 06:16:08,966 INFO success: nginx entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
2018-10-20 06:16:08,967 INFO success: uwsgi entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
[pid: 15|app: 0|req: 1/1] 172.17.0.1 () {44 vars in 922 bytes} [Sat Oct 20 06:16:13 2018] GET / => generated 12 bytes in 3 msecs (HTTP/1.1 200) 2 head
ers in 79 bytes (1 switches on core 0)
172.17.0.1 -- [20/Oct/2018:06:16:13 +0000] "GET / HTTP/1.1" 200 12 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_0) AppleWebKit/537.36 (KHTML, li
ke Gecko) Chrome/69.0.3497.100 Safari/537.36" "-"

```

### 3. pytest를 이용하여 테스트 코드 작성하기



위 과정을 거쳐 아무 기능도 없지만 그래도 동작하기는 하는 Flask 애플리케이션을 구성했습니다. 이제 여기에 테스트 코드를 작성하여 보다 더 높은 품질로 프로젝트를 유지할 수 있도록 구조를 짜 봅시다.

먼저 `test-requirements.txt` 라는 파일을 루트에 만들어 줍니다. `requirements.txt` 와 굳이 구별까지 안 해도 당장 크게 상관은 없지만, 의존성을 분리해주는 것은 미래를 생각했을 때 분명 도움이 되리라 생각합니다. 우선은 `PyTest` 라는 라이브러리를 사용하여 테스트 코드를 작성할 예정이기에 이 라이브러리만 작성해줍니다.

```
# test-requirements.txt
```

```
pytest>=3.0
```

그다음에 `tests` 폴더를 만들고 테스트를 위한 Python 파일도 작성 해 줍시다. `Flask` 인스턴스에는 `test_client()` 라는 메소드가 존재하는데, 이를 이용하면 쉽게

Flask 애플리케이션을 위한 테스트 코드를 작성할 수 있습니다. 이를 위해 먼저 `main` 파일에서 `app` 인스턴스를 불러옵니다. 이후 `app.test_client()` 를 이용하여 테스트 클라이언트를 초기화합니다.

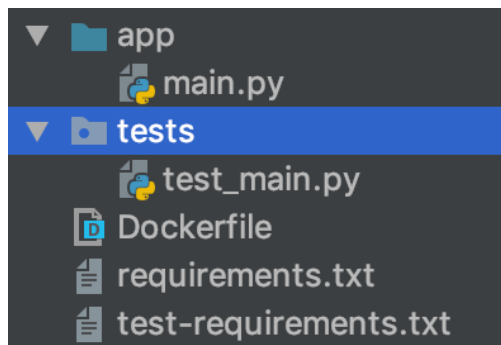
아래 코드에서는 2가지 경우를 테스트하도록 작성하였습니다. 루트(`/`)로 요청을 보내면 200을 반환하는지 확인하고, 이상한 경로로 요청을 보내면 404를 반환하는지 확인하도록 테스트를 작성하였습니다.

```
# tests/test_main.py
from main import app

client = app.test_client()

def test_index():
    response = client.get('/')
    assert response.status_code == 200

def test_404():
    response = client.get('/this_page_would_not_exist')
    assert response.status_code == 404
```



테스트 코드가 포함된 초기 프로젝트 구조는 위처럼 될 것입니다.

그럼 터미널에서 `pytest` 를 실행하여 실제로 테스트를 통과하는지 확인해 봅시다. `PYTHONPATH=app pytest tests` 명령어를 치면 테스트가 수행됩니다

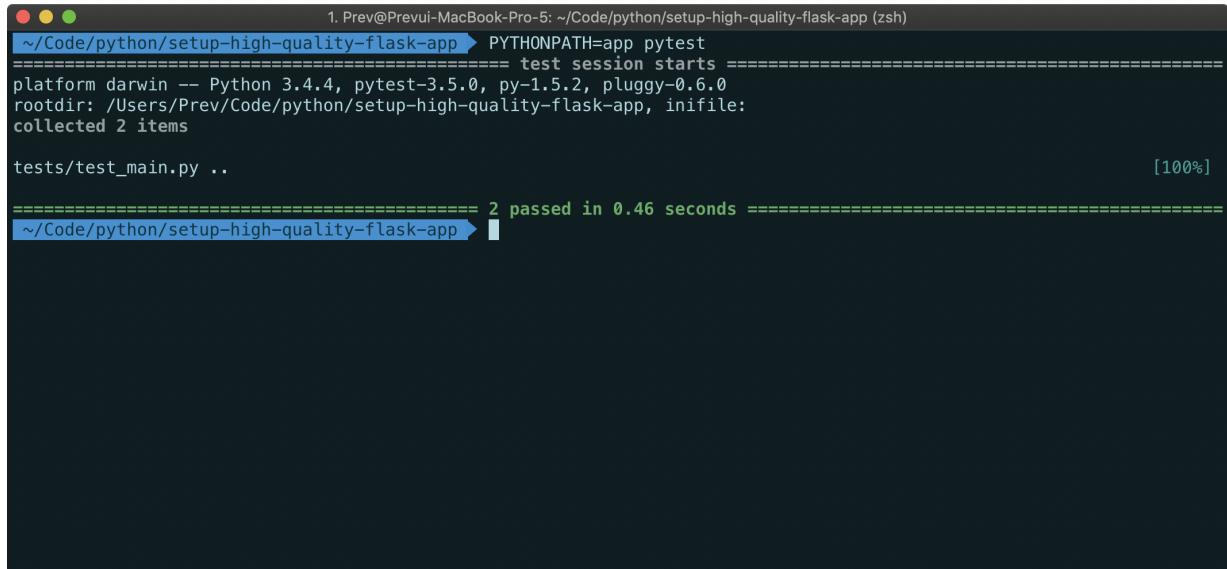
“

`PYTHONPATH` 환경 변수는 어떤 루트에서 테스트가 수행될 것인지를 지정하는 옵션입니다. 테스트 코드 첫 번째 라인에서 `from main import app` 라고 작성하였는데, 이는 `/app` 폴더를 루트로 가정하고 불러온 것이라 할 수 있습니다. `from app.main import app` 처럼 작성한다면 루트가 `/`가 되므로 `PYTHONPATH` 환경 변수 지정 없이 그저 `pytest tests` 명령어만 작성하여도 됩니다.

이 포스트에서 루트를 `/app` 으로 둔 이유는, Docker 컨테이너 내에서 `/app` 폴

더 내부의 파일만 복사해서 사용하기 때문입니다, 이 도커 구조에 맞추기 위해 pytest 명령어를 실행할 때 `PYTHONPATH` 환경 변수를 통해 이 루트 설정을 조작해 준 것이라 볼 수 있습니다.

테스트를 수행하면 다음처럼 2개의 테스트를 통과했다는 결과가 나옵니다. 200이 나와야 하는 상황과 404가 나와야 하는 상황 두 가지를 통과 한 것이죠.



```

1. Prev@Prevui-MacBook-Pro-5: ~/Code/python/setup-high-quality-flask-app (zsh)
~/Code/python/setup-high-quality-flask-app PYTHONPATH=app pytest
===== test session starts =====
platform darwin -- Python 3.4.4, pytest-3.5.0, py-1.5.2, pluggy-0.6.0
rootdir: /Users/Prev/Code/python/setup-high-quality-flask-app, inifile:
collected 2 items

tests/test_main.py .. [100%]

===== 2 passed in 0.46 seconds =====
~/Code/python/setup-high-quality-flask-app

```

## 4. MySQL 연결과 SQLAlchemy 사용하기

# SQLAlchemy

다음으로는 데이터베이스를 위한 설정 작업들을 계속해 봅시다. `pymysql` 등의 라이브러리를 naive로 사용할 수도 있지만, 보다 더 잘 관리될 수 있는 프로젝트로 구성하기 위해 ORM을 사용하고자 하며, Python에서 가장 대중적인 `SQLAlchemy`를 이용하여 개발해보려 합니다.

이를 위해 `requirements.txt`에 SQLAlchemy와 MySQL-Client를 추가해줍니다. Flask와 SQLAlchemy를 연결해주는 Flask-SQLAlchemy 라는 라이브러리도 있는데, 이도 함께 추가해주도록 합니다.

```

# requirements.txt
Flask>=1.0
sqlalchemy>=1.2

```



```
flask-sqlalchemy>=2.3.2
```

```
mysqlclient>=1.3
```

Flask-SQLAlchemy

라이브러리를

설치하면

`app.config['SQLALCHEMY_DATABASE_URI']` 를 정의해줌으로써 데이터베이스 연결을 간단하게 설정할 수 있습니다. `URI` 값은 `MySQL` 를 사용할 경우 `mysql://<username>:<password>@<hostname>/<database>?charset=utf8` 형태입니다. 위 값은 하드코딩해서 넣어선 안됩니다. **환경 변수**로 받아오는 것이 가장 이상적이며, 환경 변수를 처리하는 별도 파일을 하나 작성하도록 합시다. 저는 이 파일을 `config.py` 라고 하였습니다.

```
# app/config.py
```

```
import os
```

```
mysql_config = {
```

```
    'host': os.environ.get('MYSQL_HOST', 'localhost'),
```

```
    'user': os.environ.get('MYSQL_USER', 'root'),
```

```
    'pass': os.environ.get('MYSQL_PASS', ''),
```

```
    'db': os.environ.get('MYSQL_DB', 'my_flask'),
```

```
}
```

```
def alchemy_uri():
```

```
    return 'mysql://%s:%s@%s/%s?charset=utf8' % (
```

```
        mysql_config['user'], mysql_config['pass'], mysql_config['host'], r
```

```
)
```

```
# app/main.py
```

```
from flask import Flask
```

```
from flask_sqlalchemy import SQLAlchemy
```

```
import config
```

```
app = Flask(__name__)
```

```
app.config['SQLALCHEMY_DATABASE_URI'] = config.alchemy_uri()
```

```
db = SQLAlchemy()
```

```
db.init_app(app)
```

```
...
```

위를 통해 기본적인 환경 변수 설정과 함께 데이터베이스 연결을 설정했습니다. 실제 MySQL을 사용하기 위해 **기능 하나를 추가**해 봅시다.

먼저 `post` 라는 테이블을 하나 만들고, `id` 와 `content`, `author_email`, `created_time` 필드를 만듭시다. `id` 는 Primary index로 `auto_increment` 와 함께 사용합니다. `content` 는 `TEXT` 타입, `author_email` 은 `VARCHAR(100)`, `created_time` 은 `DATETIME` 타입으로 설정하였습니다.

Field	Type	Length	Unsigned	Zerofill	Binary	Allow Null	Key	Default	Extra
id	INT	11	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	PRI		auto_increment
content	TEXT		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			None
author_email	VARCHAR	100	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			None
created_time	DATETIME		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			None

그리고 이 테이블에 맞게 `Post` 클래스를 만들고, SQLAlchemy 사용법에 맞게 정의해 주었습니다. `db.Model` 을 상속하면 Object Model을 쉽게 만들 수 있고, DB 타입에 맞게 attribute 정의도 해 줍니다. 먼저 클래스를 만들고 Migration Tool을 이용하여 데이터베이스 상에 자동으로 테이블을 구성할 수도 있습니다.

```
# app/main.py
from flask import Flask, abort, jsonify
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime
import config

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = config.alchemy_uri()
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy()

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    content = db.Column(db.Text, nullable=False)
    author_email = db.Column(db.String(100), nullable=False)
    created_time = db.Column(db.DateTime, nullable=False, default=datetime.utcnow())

db.init_app(app)
```

...

그 뒤 위 모델을 사용하는 새 함수를 하나 만들고, post ID를 바탕으로 해당 포스트의 정보를 반환하는 기능 하나를 추가해 보았습니다.

```
# app/main.py
...

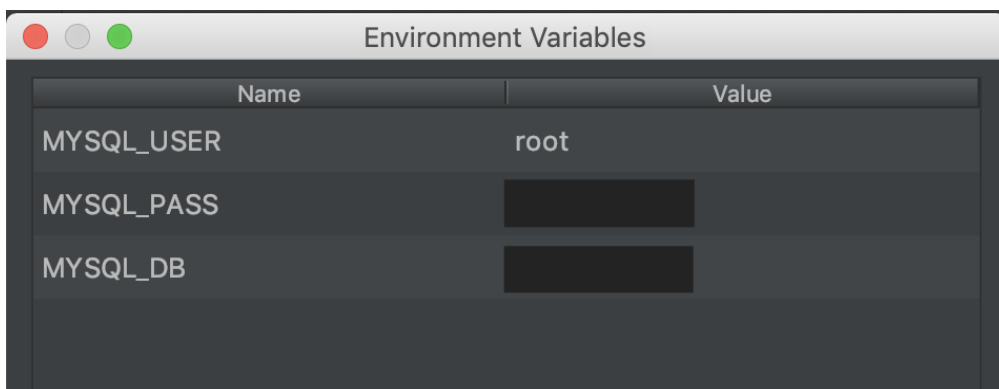
@app.route('/post/<int:id>', methods=['GET'])
def get_post(id):
    post = Post.query.get(id)
    if not post:
        return abort(404)

    return jsonify({
        'content': post.content,
        'author_email': post.author_email,
        'created_time': post.created_time,
    })

...
```

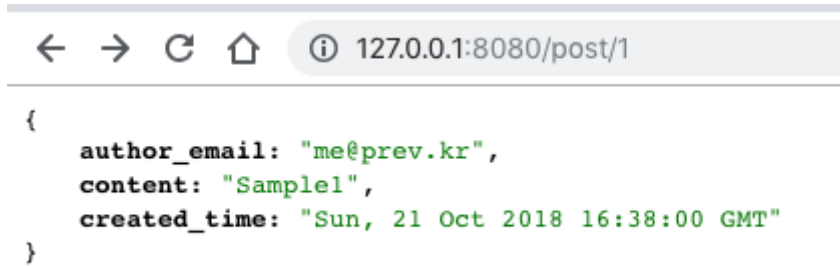
이제 이 기능이 정상적으로 작동하는지 확인해야 합니다. 다만 모든 테스트를 Docker 상에서 진행할 필요는 없습니다. 우선 IDE에서 환경 변수를 데이터베이스에 따라 알맞게 설정하고, 디버그용 서버를 띄워 테스트를 진행해 봅시다. 터미널을 사용한다면 아래처럼 실행할 수도 있습니다.

```
$ MYSQL_PASS=1234 MYSQL_DB=mydb python main.py
```



PyCharm 등의 IDE에서는 기본적으로 환경 변수를 설정하는 기능을 제공합니다.

DB 내에 임의 데이터를 집어넣은 후, `localhost:8080/post/<해당ID>` 로 접속해 보면 데이터가 정상 출력됨을 확인할 수 있습니다.



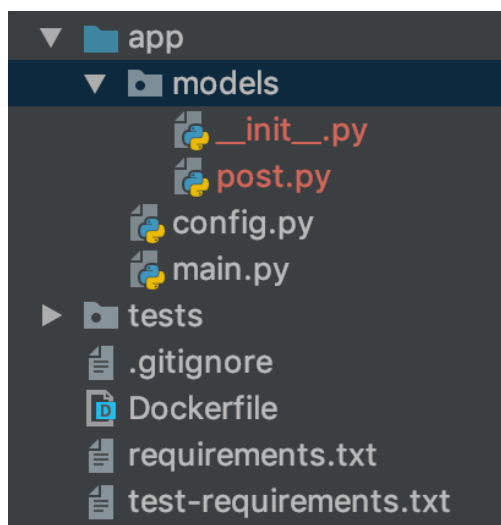
Docker를 통해 실행시킬 때에도 마찬가지로 환경 변수 설정을 해주어야 합니다.

`docker run` 명령어에서 `-e` 옵션을 통해 환경 변수를 설정할 수 있습니다.

```
$ docker run -p 8080:80 \
  -e MYSQL_USER=<username> \
  -e MYSQL_PASS=<password> \
  -e MYSQL_DB=<database> \
  --rm my_flask_app
```

#### 4.1. 모델 파일 분리하기

기능이 정상 동작함은 확인하였지만 `app/main.py` 가 너무 지저분합니다. 최소한 `Post` 같은 모델 파일은 분리하여야 합니다. `models` 폴더를 따로 만들어, 앞으로 모델이 추가될 때마다 해당 폴더 안에 넣도록 하면 보다 나은 관리 환경을 만들 수 있습니다.



```
1 # app/main.py
2 from flask import Flask, abort, jsonify
3 from models import db
4 from models.post import Post
```

```

5  import config
6
7  app = Flask(__name__)
8  app.config['SQLALCHEMY_DATABASE_URI'] = config.alchemy_uri()
9  app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
10
11 db.init_app(app)
12
13 @app.route('/', methods=['GET'])
14 def index():
15     return 'Hello World!'
16
17 @app.route('/post/<int:id>', methods=['GET'])
18 def get_post(id):
19     post = Post.query.get(id)
20     if not post:
21         return abort(404)
22
23     return jsonify({
24         'content': post.content,
25         'author_email': post.author_email,
26         'created_time': post.created_time,
27     })
28
29 if __name__ == '__main__':
30     app.run(host='0.0.0.0', debug=True, port=8080)

```

main.py hosted with ❤ by GitHub

[view raw](#)

```

1  # app/models/__init__.py
2  from flask_sqlalchemy import SQLAlchemy
3  db = SQLAlchemy()

```

models\\_\_init\_\_.py hosted with ❤ by GitHub

[view raw](#)

```

1  # app/models/post.py
2  from models import db
3  from datetime import datetime
4
5  class Post(db.Model):
6      id = db.Column(db.Integer, primary_key=True)
7      content = db.Column(db.Text, nullable=False)
8      author_email = db.Column(db.String(100), nullable=False)
9      created_time = db.Column(db.DateTime, nullable=False, default=datetime.n

```

models\post.py hosted with ❤ by GitHub

[view raw](#)

SQLAlchemy 인스턴스인 `db` 는 다양한 파일에서 참조되기에 `main` 에 두는 것 보다는 별도의 파일로 빼는 것이 오류를 일으키지 않습니다. 저의 경우는 `models/__init__.py` 를 만들어 이 파일에 `db` 인스턴스를 선언하도록 하였습니다.

## 5. 테스트 코드에서 데이터베이스 초기화 자동화하기

이제 앞선 작업에서 추가했던 기능에 대해 테스트 코드를 작성할 차례입니다. 하지만 문제는 테스트를 수행할 때, 기존에 사용되던 데이터베이스에 테스트가 동작하면 안 된다는 점입니다. 테스트가 실패하면서 데이터베이스 내의 중요 정보가 날아가 버린다면 엄청난 문제가 될 수 있습니다. 그렇기에 테스트 환경은 런타임과 완전히 분리되어야 합니다. 데이터베이스도 테스트 전용 DB를 사용하여야 합니다.

이를 해결하기 위한 방법은 여러 가지가 있습니다. 실제로 내부적으로는 가짜 데이터베이스를 사용하지만 결과는 같은 DB mock 라이브러리를 이용하는 방법도 있고, 런타임 환경과 동일한 스택을 사용하지만 런타임과는 다른 DB를 사용하도록 하는 방법도 있죠. 저는 후자를 택하여 진행을 해보려고 합니다.

앞선 설정에서 데이터베이스에 대한 접속 정보는 환경 변수를 이용해서 받아오도록 설정을 해두었습니다. 이 덕분에 테스트를 수행할 때는 환경 변수에 다른 데이터베이스 정보를 넣어주면, 런타임과 다른 환경에서 자연스럽게 테스트가 수행될 것입니다.

```
$ PYTHONPATH=app MYSQL_USER=test_user MYSQL_PASS=<some-other-pass> MYSQL_DE
```

터미널에서 테스트를 수행하고자 하면 위처럼 명령어를 작성하면 되며, IDE를 통해 테스트를 수행할 때에는 앞선 단계에서 `main` 을 실행하기 위해서 환경 변수를 설정했던 것처럼 테스트시에도 알맞은 설정을 해 주면 됩니다.

테스트에 있어서 또 중요한 사항은 모든 테스트가 수행되는 환경은 항상 일치시켜주어야 한다는 것입니다. 지난 테스트 결과에 의해서 환경이 바뀌면 안 되는데, 예를 들어 삭제되어야 할 어떤 데이터가 삭제되지 않았는데 이를 자동으로 처리하는 프로세스가 없다면 다음 테스트에 영향을 끼칠 수도 있기 때문이죠. 그

런칭에 테스트를 수행하기 전에 항상 **초기화** 작업이 자동으로 진행되어야 합니다.

테이블 초기화를 위해 SQLAlchemy에서 몇 가지 함수를 지원하지만, 외래 키 제약이 들어가게 될 경우 초기화에 문제가 많이 발생하였습니다. 테이블 간 릴레이션션을 고려하여 데이터를 초기화해주는 코드는 아래와 같습니다.

```
with app.app_context():
    for table in reversed(db.metadata.sorted_tables):
        try:
            db.engine.execute(table.delete())
        except:
            pass
    db.create_all()
```

위 코드를 통해 테스트 수행 전에 항상 테이블이 비어있는 상태를 보장할 수 있습니다. 다만 특정 상황에서는 테이블 내에 몇 개의 **기본(고정) 데이터**가 필요한 경우가 있을 수 있습니다. 항상 [레코드 생성 - 레코드 읽기] 의 순서로 테스트를 진행할 수는 없으니까요. 이 경우에 기본(고정)으로 들어가 있는 데이터를 별도로 정의하기도 하는데, 이를 **fixture** 라고 부릅니다. 프레임워크 단에서 fixture를 로드해주는 기능을 제공하지 않음으로 우리는 이 프로세스까지 새로 만들어 주어야 합니다.

Fixture는 관리하기 편하도록 CSV 형태로 관리하도록 합시다. 첫 줄에는 컬럼들을 적어주고, 두 번째 줄부터는 데이터를 콤마(,)로 구분하여 작성합니다.

```
# tests/fixtures/posts.csv
id,content,author_email,created_time
1,Hello,me@me.com,2018-10-03 12:30:00
2,Second,me@fb.com,2018-10-04 15:20:00
```

아래는 내장 **CSV** 라이브러리를 사용해서 데이터를 읽고, 이를 DB에 추가해주는 함수입니다. 파라미터 **model** 은 **db.Model** 를 확장하여 정의한 모델 클래스를 받고, **file\_name** 는 csv 파일 이름을 받습니다. **model(\*\*attrs)** 코드를 통해서 새 인스턴스를 생성하게 되고, 이를 **db.session** 에 추가함으로써 **INSERT** 쿼리가 내부적으로 수행될 것입니다.

```
import os
import csv
```

```
def load_model_fixtures(db, model, file_name):
    cur_dir = os.path.dirname(os.path.realpath(__file__))

    with open('%s/fixtures/%s' % (cur_dir, file_name), encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for row in reader:
            attrs = dict(row.items())
            db.session.add(model(**attrs))

    db.session.commit()
```

이들을 모두 포함하면 테스트를 위한 사전 작업 코드가 꽤 길어집니다. `PyTest`에서는 테스트가 수행되기 전의 사전 작업들을 위해 `conftest.py` 라는 파일을 특별하게 정의하고 있습니다. 이 파일을 새로 만들고 테스트 사전 작업과 테스트 함수를 분리하도록 합니다.

```
1  # tests/conftest.py
2  import pytest
3  from main import app
4  from models import db
5  from models.post import Post
6  import os
7  import csv
8
9  def load_model_fixtures(db, model, file_name):
10     """ Load model fixtures in file.
11     :param db: SQLAlchemy db instance
12     :param model: Model class written with SQLAlchemy
13     :param file_name: File name of fixture (CSV)
14     """
15     cur_dir = os.path.dirname(os.path.realpath(__file__))
16
17     with open('%s/fixtures/%s' % (cur_dir, file_name), encoding='utf-8') as f:
18         reader = csv.DictReader(f)
19         for row in reader:
20             attrs = dict(row.items())
21             db.session.add(model(**attrs))
22
23     db.session.commit()
24
25  @pytest.fixture
26  def client():
27     client = app.test_client()
```



```

28
29     with app.app_context():
30         for table in reversed(db.metadata.sorted_tables):
31             try:
32                 db.engine.execute(table.delete())
33             except:
34                 pass
35
36         db.create_all()
37         load_model_fixtures(db, Post, 'posts.csv')
38
39     yield client

```

conftest.py hosted with ❤ by GitHub

[view raw](#)

`@pytest.fixture` 키워드를 이용하면 다른 테스트 코드에서 공통으로 사용할 수 있는 일종의 변수를 정의할 수 있습니다 (데이터베이스 fixture 작업과는 관계가 없습니다). 테스트 함수에서는 파라미터에 해당 함수 이름을 추가해주면 그 변수를 바로 사용할 수 있습니다. 이 기능을 이용하면 추후 로그인된 사용자의 세션을 반환하는 등의 공통 기능을 쉽게 추가할 수 있습니다.

`post` 이외에 다른 테이블에 대해서도 fixture를 사용하고 싶다면 CSV 파일을 추가로 만들어준 다음에, 37번 다음 줄에 `load_model_fixtures` 함수를 더 호출하면 될 것입니다.

```

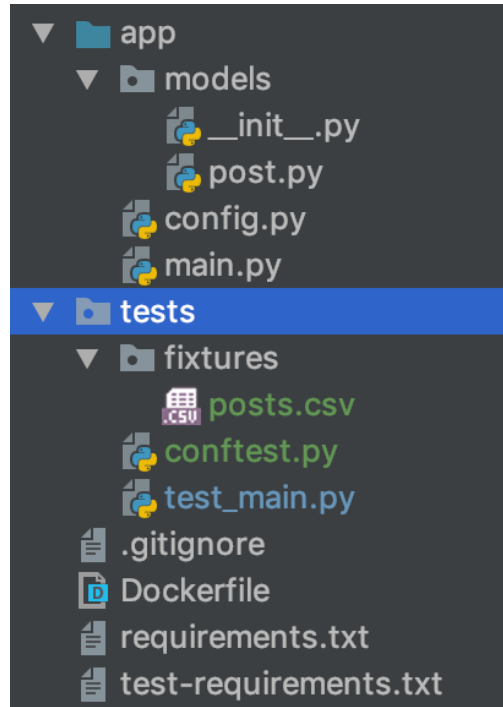
1  def test_index(client):
2      response = client.get('/')
3      assert response.status_code == 200
4
5  def test_404(client):
6      response = client.get('/this_page_would_not_exist')
7      assert response.status_code == 404
8
9  def test_get_post(client):
10     response = client.get('/post/1')
11     assert response.status_code == 200
12
13     response = client.get('/post/2')
14     assert response.status_code == 200
15
16     response = client.get('/post/3')
17     assert response.status_code == 404

```

test\_main.py hosted with ❤ by GitHub

[view raw](#)

기존 테스트 코드에 `get_post()`에 대한 테스트 코드도 추가로 작성해보았습니다. ID=1, ID=2인 포스트는 `fixtures/posts.csv`를 통해서 자동으로 생성되었기에 `200`을 반환하게 됩니다. ID=3인 포스트는 만들어지지 않았으므로 `404`를 반환하게 됩니다. 여기까지 설정을 마치면 프로젝트 구조는 아마 아래처럼 될 것입니다.



이 작업을 통해 테스트는 런타임과 분리된 환경에서 진행되며, 테스트 수행 시마다 매번 데이터가 초기화되고 일부 고정 데이터 값은 파일을 통해 미리 지정해 둘 수 있게 되었습니다.

## 6. CI 서비스 연동하기



CI 서비스를 사용하면 매번 테스트를 직접 돌려보지 않아도, 필요할 때마다 외부 서비스에 의해서 **자동으로 테스트가 작동**하게 됩니다. CI를 이용하면 실수로 테스트를 돌리지 않고 런타임 서비스로 업데이트하는 경우를 사전에 방지할 수 있습니다.

GitHub과 연동되는 CI 서비스는 상당히 많지만, 최근 가장 활발히 사용되고 있는 CI 서비스는 크게 *CircleCI*와 *TravisCI*가 있습니다. **TravisCI**는 public repository에 대해 완전히 무료이며, **CircleCI**는 월 1500빌드까지는 public과 private 모두 무료로 제공합니다. 처음부터 오픈소스로 공개한다면 TravisCI가 적절할 것이고, 내부적으로 사용하여 공개할 수 없는 코드는 CircleCI를 이용하면 됩니다. 월 1500빌드는 소규모 프로젝트라면 충분한 양이라 생각합니다.

CircleCI로 python 프로젝트를 빌드하는 문서는 공식 [도큐먼트](#)에 잘 나와 있습니다. 공식 문서를 참고하여 Python + MySQL 환경을 구성한 `.circleci/config.yml`은 아래와 같습니다.

```

1  version: 2
2  jobs:
3    build:
4      docker:
5        - image: circleci/python:3.6.1
6          environment:
7            MYSQL_HOST: 127.0.0.1
8            MYSQL_USER: root
9            MYSQL_PASS: root
10           MYSQL_DB: test_db
11        - image: mysql:5.7
12          command: mysqld --character-set-server=utf8mb4 --collation-server=utf8mb4_unicode_ci
13          environment:
14            MYSQL_ROOT_PASSWORD: root
15            MYSQL_DATABASE: test_db
16
17      working_directory: ~/app
18      steps:
19        - checkout
20        - run:
21          name: Install MySQL Client
22          command: sudo apt-get update && sudo apt install -y mysql-client
23        - restore_cache:
24          keys:
25            - v1-dependencies-{{ checksum "requirements.txt" }}
26            - v1-dependencies-
27        - run:
28          name: install dependencies
29          command: |
30            python3 -m venv venv
31            . venv/bin/activate
32            pip install -r requirements.txt
33            pip install -r test-requirements.txt
34        - save_cache:

```

```

35     paths:
36         - ./venv
37     key: v1-dependencies-{{ checksum "requirements.txt" }}
38 - run:
39     name: run tests
40     command: |
41         . venv/bin/activate
42         PYTHONPATH=app pytest
43 - store_artifacts:
44     path: test-reports
45     destination: test-reports

```

.circleci.config.yml hosted with ❤ by GitHub

[view raw](#)

CircleCI에 들어가 결과를 확인해보면 테스트가 정상적으로 동작함을 확인할 수 있습니다.

Jobs » Prev » full-stack-flask-app » master » 2 (build)

2.0 Rerun workflow

0 (00:16) Add Containers +

- Restoring cache 00:00
- install dependencies 00:05
- Saving Cache 00:01
- run tests 00:01
 

```

$ #!/bin/bash -eo pipefail
. venv/bin/activate
PYTHONPATH=app pytest

===== test session starts =====
platform linux -- Python 3.6.1, pytest-3.9.1, py-1.7.0, pluggy-0.8.0
rootdir: /home/circleci/app, inifile:
collected 3 items

tests/test_main.py ... [100%]

===== 3 passed in 0.11 seconds =====

```

Exit code: 0
- Uploading artifacts 00:00

한 번 설정이 완료되면 GitHub에 새로운 push/PR이 있을 때마다 자동으로 빌드 (테스트)가 수행됩니다. 그 결과는 commit 탭이나 Pull Request 항목에서 바로 확인할 수 있습니다. [README.md](#)에 배지를 달아 바로 결과를 확인하도록 할 수도 있습니다.

Branch: master ▼

Commits on Oct 21, 2018

Add README.md

Prev committed 2 hours ago ✓

Add mysqlclient on requirements

Prev committed 2 hours ago ✓

Integrate with CircleCI

Prev committed 2 hours ago ✗

테스트에 실패한 커밋은 위처럼 빨간색 X 표시가 함께 띄워집니다.

실제 서비스를 운용할 때에는 항상 테스트를 통과한 경우에만 PR을 수락받도록 하며, `master` 브랜치가 변경 될 때마다 `docker build` 및 `run`을 재 수행하여 자동으로 배포되는 환경 또한 구축할 수 있습니다.



# Jenkins

Jenkins라는 소프트웨어를 이용하면 쉽게 배포 자동화 환경을 구축할 수 있습니다.

이렇게 긴 여정을 거치며 **Docker**, **SQLAlchemy**, **PyTest**, **CircleCI**와 함께 Flask+MySQL 웹 애플리케이션을 구성해 보았습니다. 위와 같은 프로젝트 구조를 바탕으로 꾸준히 테스트 코드를 작성하고 확장성 높은 구조로 개발을 계속하다 보면 높은 품질의 프로젝트가 될 수 있으리라 생각합니다.

위 포스트를 통해 만들어진 최종 샘플 프로젝트는 [github.com/Prev/full-stack-flask-app](https://github.com/Prev/full-stack-flask-app) 에서 확인하실 수 있습니다.

Docker를 이용해서 크롤링 작업을 패키징 한 후, EC2 Spot Instance를 이용하여 병렬로 작업 수행하기

# 백엔드 # 데브옵스

네이버 오픈소스를 활용하여 확장성 있는 서버 아키텍처를 구축하고 성능 개선해보기 - 2편

1월 12일, 2018

한양대학교의 소프트웨어스튜디오2 과목해서 진행했던 프로젝트로, 네이버의 백엔드 관련 오픈소스를 활용하여 확장성 있는 서버 아키텍처를 구축하고 성능 개선해보기

# 백엔드 # 학부

네이버 오픈소스를 활용하여 확장성 있는 서버 아키텍처를 구축하고 성능 개선해보기 - 1편

12월 31일, 2017

한양대학교의 소프트웨어스튜디오2 과목해서 진행했던 프로젝트로, 네이버의 백엔드 관련 오픈소스를 활용하여 확장성 있는 서버 아키텍처를 구축하고 성능 개선해보기

# 백엔드 # 학부

정적 웹 프레임워크(직지) 개발지

7월 14일, 2017

정적 웹 프레임워크인 "직지"의 설계 관점과 개발지

# 백엔드 # 시스템디자인



✉ 이 블로그의 새로운 글 받아보기

## 댓글 1개

댓글 달기...



**정성민**

좋은글 잘보고갑니다.

좋아요 · 답글 달기 · 1년

[Facebook 댓글 플러그인](#)

0 Comments

[prevblog](#)

[Disqus' Privacy Policy](#)

[Login](#) ▾

[Recommend](#)

[Tweet](#)

[Share](#)

[Sort by Best](#) ▾



Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

Be the first to comment.

[Subscribe](#)

[Add Disqus to your site](#) [Add DisqusAdd](#)

[Do Not Sell My Data](#)

## 관련 글 더보기

Docker와 EC2 Spot Instance로 병렬 작업 수행하기

6월 12일, 2019