

# Flask로 만드는 블로그

공동공부 (2명)

커버 페이지

## 토픽 목록

Hello Flask

개발환경 구축

템플릿

부트 스트랩

데이터베이스

## 생산자



a476548

토픽 5 / 봤어요 0

## 데이터베이스

2018-08-10 08:54:48

이전 시간에 우리는 부트스트랩을 이용해서 어플리케이션의 디자인을 조금 세련되게 변경했습니다.

이번에는 사용자와 게시물에 대한 정보를 저장하기 위한 데이터베이스를 만들어보도록 하겠습니다.

Flask는 데이터베이스 기능을 내장하고 있지 않습니다. 따라서 우리는 적합한 데이터베이스를 골라서 사용해야 합니다. 각 데이터베이스마다 사용되는 API도 다르고, SQL 문법도 다릅니다. Python의 객체에 데이터 모델을 정의하고 이를 데이터베이스와 매핑해주는 것을 ORM(Object Relation Model)이라고 합니다. 덕분에 코드는 특정 데이터베이스에 종속되지 않고, 기본 객체만으로 데이터를 기술할 수 있기 때문에 조금 더 OOP 스타일의 코드를 작성할 수 있습니다.

Python에서 ORM으로 많이 쓰이는 것 중 SQLAlchemy가 있는데, 이를 Flask에서 플러그인처럼 사용하기 쉽게 만들어진 Flask-SQLAlchemy가 있습니다.

Flask-SQLAlchemy를 사용하려면 패키지를 설치해야 합니다.

터미널

```
1 (venv) PS flask_blog> pip install flask_sqlalchemy
```

Flask-SQLAlchemy를 사용하기 위해서는 'flask\_sqlalchemy' 패키지를 임포트 해야 합니다.

'app.py'파일에 임포트를 추가합니다.

파일: /app.py

```
1 from datetime import datetime
2
3 from flask import Flask, render_template
4 from flask_sqlalchemy import SQLAlchemy
5
6 # ...
```

그 다음 'app'객체에 몇 가지 설정을 추가해야 합니다. 먼저 어플리케이션의 시크릿 키를 추가 합니다. 다음으로는

'SQLAlchemy'에서 사용할 데이터베이스의 위치를 알려줘야 합니다. 마지막으로

'SQLALCHEMY\_TRACK\_MODIFICATIONS'의 경우에는 추가적인 메모리를 필요로 하므로 꺼두는 것을 추천합니다.

설정을 완료했으면 'SQLAlchemy' 객체를 하나 만듭니다.

파일: /app.py

```
1 # ...
2
3 app = Flask(__name__)
4
5 app.config['SECRET_KEY'] = 'this is secret'
6 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'
7 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
8
9 db = SQLAlchemy(app)
10
11 # ...
```



## 사용자 모델 추가


본 프로젝트에서 사용자는 사용자 계정 이름을 갖고, 이메일과 암호가 있으며 프로필 사진을 저장할 수 있는 공간이 필요합니다.

먼저 사용자 데이터 모델을 나타내는 객체를 하나 선언 합니다. 그리고 SQLAlchemy의 기능을 사용하기 위해 `db.Model`을 상속 받습니다. 기본적으로 데이터베이스 테이블 이름은 자동으로 정의되지만 `\_\_table\_name\_\_`을 이용해 명시적으로 정할 수 있습니다.

파일: /app.py

```
1 # ...
2
3 class User(db.Model):
4     __table_name__ = 'user'
```

사용자 데이터 모델을 나타내는 객체를 선언했는데, 이제 여기에 모델이 갖고 있어야 하는 필드와 관련된 제약사항들을 적어줘야 합니다.

| user   |                            |
|--|----------------------------|
|  PK | id: integer                |
| <hr/>  |                            |
|  | username: string(100)      |
|  | email: string(120)         |
|  | password: string(100)      |
|  | profile_image: string(100) |

`id` 필드는 대부분의 모델에서 기본 키로 사용합니다.

`username`, `email`, `password`, `profile\_image` 필드는 문자열로 정의를 하고, 최대 길이를 명시하여 공간을 절약할 수 있도록 합니다. 또한 `username`과 `email` 필드는 서로 중복되지 않아야 하고, 비어있지 않아야 합니다. `password` 필드의 경우에는 중복되는 것은 괜찮지만 비어있지 않아야 합니다. 그리고 보안을 위해 평문으로 저장하는 것이 아니라 암호화를 해서 저장을 해야 합니다. `profile\_image` 필드는 이미지 데이터를 DB에 직접 저장하는 것이 아니라 파일 시스템에 저장한 다음 그 파일 이름만 저장할 예정입니다. 그리고 프로필 이미지는 모든 사람이 처음부터 넣는 것은 아니기 때문에 기본 이미지 파일을 가리킬 필요가 있습니다. 기본 이미지 파일은 `default.png`로 설정하도록 하겠습니다.

테이블의 컬럼을 만들기 위해서는 `db.Column()`을 이용합니다. 컬럼의 이름은 기본적으로 변수 이름을 사용합니다. `db.Column()`은 데이터 타입에 대한 정보와 제약 조건들을 넣어줄 수 있습니다. 위의 조건들을 바탕으로 작성된 객체는 다음과 같습니다.

파일: /app.py

```
1 # ...
2
3 class User(db.Model):
4     __table_name__ = 'user'
5
6     id = db.Column(db.Integer, primary_key=True)
7     username = db.Column(db.String(100), unique=True, nullable=False)
8     email = db.Column(db.String(120), unique=True, nullable=False)
9     password = db.Column(db.String(100), nullable=False)
10    profile_image = db.Column(db.String(100), default='default.png')
11
12    def __repr__(self):
13        return f"<User('{self.id}', '{self.username}', '{self.email}')>"
```

데이터 모델 객체의 경우에도 일반적인 Python 객체처럼 `\_\_repr\_\_`과 같은 메소드를 사용할 수 있습니다.

이제 비밀번호를 평문이 아닌 암호화된 해시로 저장해보겠습니다.

`werkzeug.security`에 있는 `generate\_password\_hash`와 `check\_password\_hash`를 이용해 비밀번호를 할 수 있습니다.



`generate\_password\_hash` 함수는 문자열을 암호화된 해시로 바꿔주는 역할을 합니다. `check\_password\_hash` 함수는 암호화된 해시와 문자열을 비교해서 이 문자열이 동일한 해시를 갖는 경우 참을 반환합니다.

다음은 터미널 상에서 `generate\_password\_hash` 함수와 `check\_password\_hash`를 사용해본 결과입니다. 먼저 `password`라는 비밀번호를 `generate\_password\_hash` 함수로 암호화 하고, `check\_password\_hash` 함수를 이용해 맞는 비밀번호화 틀린 비밀번호를 넣었을 경우 결과 값을 비교했습니다.

Python shell

```
1 >>> from werkzeug.security import generate_password_hash, check_password_hash
2 >>> hash = generate_password_hash('password')
3 >>> print(hash)
4 pbkdf2:sha256:50000$97whKFrP$22cd755cfd64564252a44588b7457942850288339932!
5 >>> check_password_hash(hash, 'password')
6 True
7 >>> check_password_hash(hash, 'wrong password')
8 False
9 >>>
```

맞는 경우에는 `True`를 틀린 경우에는 `False`를 반환하는 것을 알 수 있는데요.

이제 코드 상에서 비밀번호를 암호화, 복호화 하는 코드를 작성하겠습니다. 먼저 `app.py` 파일에 `werkzeug.security`에서 `generate\_password\_hash`와 `check\_password\_hash`를 임포트 합니다.

파일: /app.py

```
1 from datetime import datetime
2
3 from flask import Flask, render_template
4 from flask_sqlalchemy import SQLAlchemy
5 from werkzeug.security import generate_password_hash, check_password_hash
6
7 # ...
```

사용자를 추가해야 할 필요가 있을 때마다 개발자가 비밀번호를 암호화해서 저장하는 로직을 넣어줄 수도 있지만 이 경우에는 사람이 하는 일이다 보니 실수하여 빼먹을 가능성이 높습니다. 따라서 암호화하는 과정은 객체를 생성할 때 해주는 것이 좋은데, `\_\_init\_\_`을 하는 과정에서 암호화를 할 수 있도록 객체를 수정하겠습니다.

파일: /app.py

```
1 # ...
2
3 class User(db.Model):
4     __table_name__ = 'user'
5
6     id = db.Column(db.Integer, primary_key=True)
7     username = db.Column(db.String(100), unique=True, nullable=False)
8     email = db.Column(db.String(120), unique=True, nullable=False)
9     password = db.Column(db.String(100), nullable=False)
10    profile_image = db.Column(db.String(100), default='default.png')
11
12    def __init__(self, username, email, password, **kwargs):
13        self.username = username
14        self.email = email
15
16        self.set_password(password)
17
18    def __repr__(self):
19        return f"<User('{self.id}', '{self.username}', '{self.email}'>"
20
21    def set_password(self, password):
```



```

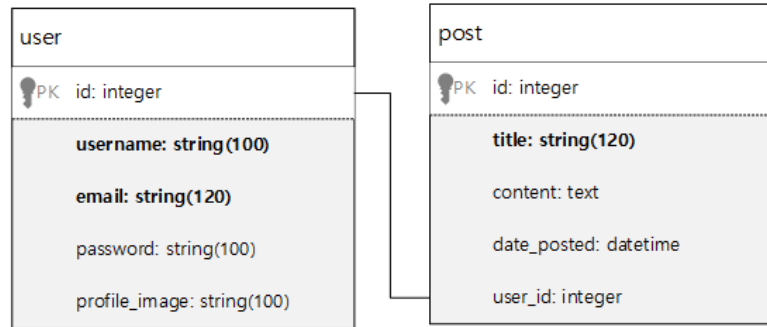
22     self.password = generate_password_hash(password)
23
24     def check_password(self, password):
25         return check_password_hash(self.password, password)

```

## 게시물 모델 추가

이제 게시물을 저장하는 객체에 대해 생각해보겠습니다. 게시물은 제목과 내용으로 이루어 질 것이고, 추가적으로 언제 게시되었는지, 누가 게시했는지에 대한 정보가 필요합니다.

이 중 게시자에 대한 정보는 사용자 데이터 모델에서 참조할 수 있습니다. 데이터 베이스에서 관계라고 하는데, 지금 같은 경우는 하나의 유저가 여러 게시물을 갖을 수 있는 관계가 형성됩니다.



이를 코드로 나타내면 다음과 같습니다.

파일: /app.py

```

1  # ...
2
3  class User(db.Model):
4      __table_name__ = 'user'
5
6      id = db.Column(db.Integer, primary_key=True)
7      username = db.Column(db.String(100), unique=True, nullable=False)
8      email = db.Column(db.String(120), unique=True, nullable=False)
9      password = db.Column(db.String(100), nullable=False)
10     profile_image = db.Column(db.String(100), default='default.png')
11
12     posts = db.relationship('Post', backref='author', lazy=True)
13
14     def __init__(self, username, email, password, **kwargs):
15         self.username = username
16         self.email = email
17
18         self.set_password(password)
19
20     def __repr__(self):
21         return f"<User('{self.id}', '{self.username}', '{self.email}')>"
22
23     def set_password(self, password):
24         self.password = generate_password_hash(password)
25
26     def check_password(self, password):
27         return check_password_hash(self.password, password)
28
29     class Post(db.Model):
30         __table_name__ = 'post'
31

```



```

32     id = db.Column(db.Integer, primary_key=True)
33     title = db.Column(db.String(120), unique=True, nullable=False)
34     content = db.Column(db.Text)
35     date_posted = db.Column(db.DateTime, default=datetime.utcnow())
36
37     user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
38
39     def __repr__(self):
40         return f"<Post('{self.id}', '{self.title}')>"

```

위 코드를 보면 기본적으로 게시물에 대한 데이터 모델 객체를 선언합니다. `title`은 게시물 제목을, `content`는 게시물 내용을, `date\_posted`는 게시일을 나타냅니다. 게시일의 경우에는 기본값을 `datetime.utcnow()`를 사용함으로써 명시적으로 게시일을 나타내지 않은 경우 현재 시간을 게시일로 하도록 하였습니다. 그 다음 게시자에 대한 내용을 나타내기 위해 `user` 테이블의 id를 외래키로 하는 `user\_id`라는 컬럼을 만들었습니다. 여기서 중요한 점은 `db.ForeignKey`는 **테이블 이름**을 인자로 받습니다. SQLAlchemy에서 테이블 이름은 기본적으로 소문자를 사용하고 여러 단어의 조합인 경우에는 스네이크 케이스를 사용합니다.

`User` 객체를 보면 `posts` 컬럼이 추가되어 있는 것을 확인할 수 있습니다. `posts` 컬럼은 `db.relationship`를 사용하는데 이는 실제 데이터베이스에 나타나는 필드는 아닙니다. 이 가상 필드는 데이터베이스를 좀 더 높은 추상화 수준에서 바라볼 수 있게 도와주는 역할을 하는데요. 예를 들어 사용자를 `user`이라는 변수에 저장했다고 한다면, 이 사용자가 작성했던 모든 게시물에 대한 정보는 `user.posts`를 이용해 접근할 수 있습니다. `db.relationship`의 첫 번째 인자는 `db.ForeignKey`와는 다르게 **객체 이름**을 받습니다.

그리고 `backref`는 `Posts` 객체에 삽입되는 가상 필드 이름입니다. 즉, 게시물을 `post`라는 변수에 저장했다고 한다면 이 게시물을 작성한 게시자를 `post.author`을 이용해 접근할 수 있음을 의미합니다. 이를 통해 데이터베이스의 데이터를 Python 코드 상에서 접근할 때, 고수준의 추상화된 레벨에서 사용할 수 있습니다.

## 데이터베이스 초기화 및 데이터 추가

데이터베이스에 어떤 내용을 채워 넣을지 구조적인 부분을 코드상으로 나타내었습니다. 하지만 실제 데이터베이스에 테이블을 만들고 데이터를 넣어준 것은 아니라 데이터베이스에서 데이터를 접근하려 하면 에러를 나타낼 것입니다. 따라서 데이터베이스를 초기화해 줄 필요가 있습니다.

먼저 프로젝트 디렉터리에서 python 터미널을 열어줍니다.

터미널

```

1 (venv) PS flask_blog> python
2 Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>>

```

Python 터미널 상에서 우리의 Flask 어플리케이션과 데이터베이스, 사용자 모델, 게시물 모델을 불러옵니다. 그 다음 `db.create\_all()`을 이용해 데이터베이스를 초기화할 수 있습니다.

```

1 >>> from app import app, db, User, Post
2 >>> db.create_all()

```

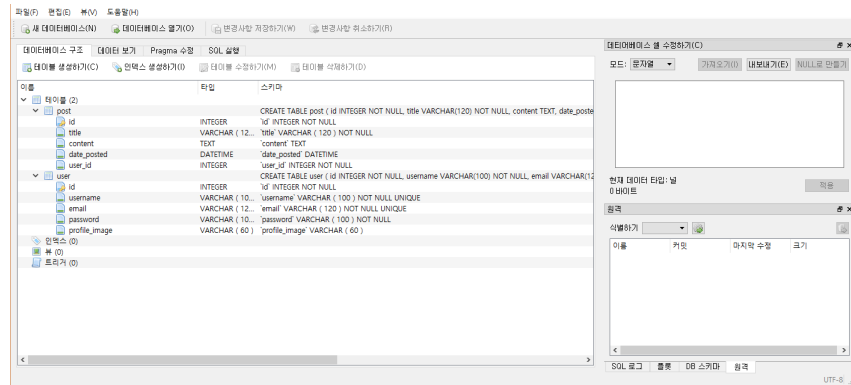
프로젝트 디렉터리를 보면 `site.db`라는 파일이 하나 생겼습니다. 그리고 DB를 열어보면 우리가 만든 사용자 모델과 게시물 모델에 대한 테이블이 만들어져 있는 것을 확인할 수 있습니다.

```

1 flask_blog/
2 └─ templates/
3   └─ about.html
4   └─ index.html
5   └─ layout.html
6 └─ static/
7   └─ layout.css
8 └─ venv/
9 └─ app.py
10 └─ site.db

```





데이터베이스를 초기화했으면 테스트를 위한 데이터를 한 번 작성해보도록 하겠습니다. 터미널을 이용하여 사용자를 하나 추가하겠습니다.

```
1 >>> user = User(username='user', email='user@blog.com', password='password')
2 >>> db.session.add(user)
3 >>> db.session.commit()
```

`user` 변수에 `User` 데이터 모델에 대한 인스턴스를 만들어 저장한 다음 이를 데이터베이스에 저장하기 위해 `db.session`을 이용합니다. 데이터베이스를 수정하는 작업이 끝나면 `db.session.commit()`을 이용해 변경 내용을 저장할 수도 있고, `db.session.rollback()`을 이용해 변경 사항을 취소할 수도 있습니다. 여기서 `db.session.add()`를 통해 데이터베이스에 객체를 추가했다더라도 실제 데이터가 저장되는 시점은 `db.session.commit()`을 수행한 다음입니다.

이제 게시물을 2개 추가해보도록 하겠습니다.

```
1 >>> post1 = Post(title='첫 번째 게시물', content='첫 번째 게시물 내용', author_id=1)
2 >>> post2 = Post(title='두 번째 게시물', content='두 번째 게시물 내용', author_id=1)
3 >>> db.session.add(post1)
4 >>> db.session.add(post2)
5 >>> db.session.commit()
```

게시물을 추가할 때 자세히 보면 `user\_id`를 넣어주지 않는 것을 확인할 수 있습니다. 우리가 `Post` 데이터 모델을 만들 때, 게시자에 대한 정보를 넣어주기 위해 외래 키로 `user\_id` 필드를 삽입했지만 `User` 데이터 모델에 관계에서 `backref`를 이용해 `author`에 대한 정보도 넣어줬습니다. 따라서 `User` 모델에서 만들어진 가상의 `author` 필드를 사용해 `user\_id` 필드에 직접 데이터를 넣어주지 않아도 게시자에 대한 정보를 쉽게 추적할 수 있습니다.

데이터베이스에 저장한 데이터를 가져오기 위해서는 각 모델에 `query`를 이용합니다. 예를 들어 전체 사용자를 가져오고 싶으면 `User.query.all()`을 입력하면 됩니다.

```
1 >>> User.query.all()
2 [<User('user', 'user@blog.com')>]
3 >>> Post.query.all()
4 [<Post('1', '첫 번째 게시물')>, <Post('2', '두 번째 게시물')>]
```

## 더미 데이터 삭제

게시물을 더미 데이터로 처리 했던 것을 db에 있는 데이터로 바꿔주도록 하겠습니다. `app.py`에 작성되었던 더미 데이터를 삭제하고, `db.query` 명령을 이용해 데이터베이스에 저장되어 있는 모든 게시물 데이터를 불러와 `render\_template` 함수에 전달합니다.

```
1 from datetime import datetime
2
3 from flask import Flask, render_template
4 from flask_sqlalchemy import SQLAlchemy
5
6 app = Flask(__name__)
7
8 app.config['SECRET_KEY'] = 'this is secret'
9 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'
10 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

오픈튜토리얼스의  
후원회원을 모집합니다

```

11
12 db = SQLAlchemy(app)
13
14 @app.route('/')
15 @app.route('/index')
16 def index():
17     posts = Post.query.all()
18
19     return render_template('index.html', posts=posts)
20
21 # ...

```

그리고 'index.html' 파일을 수정해 'holder.js'를 통해 이미지를 보여줄 공간만 마련했던 것을 실제 데이터와 연결 하도록 하겠습니다.

```

1 {% for post in posts %}
2 <article>
3   <hr>
4   <div class="row">
5     <div class="col">
6       <a class="h1" href="#">{{ post.title }}</a>
7       <p class="text-justify">{{ post.content }}</p>
8     </div>
9     <div class="col-md-2 d-flex flex-column">
10      {{ post.author.username }}</p>
12    </div>
13  </div>
14  <small class="text-muted">{{ post.date_posted.strftime('%Y-%m-%d') }}</
15 </article>
16 {% endfor %}

```

'<article>' 태그가 있는 부분에 '<img>' 태그를 보면 'src="#"'로 비어있는 것을 확인할 수 있습니다. 이를 'static/profile\_imgs' 하위에 저장 되어 있는 프로필 사진들로 연결합니다.



## 기본 프로필 이미지 저장

'User' 모델에 기본 프로필 이미지로 'default.png'를 적어주었습니다. 이 사진 파일을 저장하기 위해서 'static' 디렉터리 하위에 'profile\_imgs' 디렉터리를 만들고 프로필 이미지 파일들을 저장하겠습니다. 폴더를 나누는 이유는 'static' 폴더에는 프로필 사진 뿐 아니라 다양한 자바스크립트 파일, css 스타일 시트 파일 등이 저장될텐데 여러 파일들이 혼란되면 정리하는데 불편하기 때문입니다.

기본 사진 파일은 github 저장소에서 다운받아 저장해주세요.

사진 파일은 [링크](#)에서 다운로드 받았으며 CC0 라이선스임을 밝힙니다.

오픈튜토리얼스의  
후원회원을 모집합니다

```

1 flask_blog/
2   └─ templates/
3     └─ about.html
4     └─ index.html
5     └─ layout.html
6   └─ static/
7     └─ profile_imgs/
8       └─ default.png
9     └─ layout.css
10  └─ venv/
11  └─ app.py
12  └─ site.db

```

현재까지 작성된 전체 코드는 [Flask blog tutorial Chapter4](#)에서 확인할 수 있습니다.

Python 인터랙티브 셸에서 작성했던 코드는 `scripts.py`라는 파일에 넣어두었습니다.

봤어요 (0명)

이전

다음

댓글을 작성하려면 로그인하셔야 합니다.



**다주** 9개월 전

감사합니다 !!



**hate\_db** 9개월 전

good



**로다** 1년 전

좋은 글 감사합니다^^  
정말 원하던 내용의 글입니다



**김플라스크** 1년 전

flask 내용에 대해 잘 없는데 좋은 글 잘 봤어요!

모바일 버전

