6. Database(2), Session

회원가입/로그인/로그아웃

사용자 모델 만들기

```
class User(db.Model):
    username = db.Column(db.Unicode(255),
primary_key=True)
    password = db.Column(db.Unicode(255))
    nickname = db.Column(db.Unicode(255), unique=True)
    registered_at = db.Column(db.DateTime,
default=datetime.datetime.now)
    last_logined_at = db.Column(db.DateTime)
```

회원 가입

```
@app.route('register', methods=['GET', 'POST'])
def register():
    if request.method == 'GET':
        return render_template('register.html')
    user = User(username=request.form['username'],
                password=request.form['password'],
                nickname=request.form['nickname'])
    db.session.add(user)
    db.session.commit()
    return redirect(url_for('index'))
```

회원 가입

```
<form action="{{ url_for('register') }}" method="post">
    <input type="text" name="username"><br>
    <input type="password" name="password"><br>
    <input type="password_confirm"
name="password_confirm"><br>
    <input type="text" name="nickname"><br>
    <button type="submit">회원 가입</button>
</form>
```

해시(Hashing)

- 비밀번호는 평문으로-사용자가 입력한 비밀번호 그대로 저장되면 안 된다.
  - 누군가 데이터베이스를 볼 경우, 바로 비밀번호가 드러나기 때문
- 따라서 비밀번호는 다른 방식으로 저장되어야 하는데, 보통 해시 기술을 이용해서 저장된다
- 해시는 주어진 문자열에 특별한 처리를 해서, 알아볼 수 없는 다른 문자열로 바꾸는 기술이다.
  - 해시가 으깬다는 뜻이니, 으깨서 못 알아보게 만든다고 생각하면 된다
- 중요한 건 해시를 한 문자열을 원래 문자열로 되돌릴 수는 없다는 것

해시 예시

- MD5: 예전에 많이 쓰였는데, 요새는 안 쓴다(보안에 취약함)
  - "파이썬" -> "b4bc4dffad3b90de50dd3d86105a8c9f"
- SHA-1: 최근까지 널리 쓰였던 방식. 이것도 뚫린다고...
  - "파이썬" -> "025cca0dfd6bd49f95b26f4e30e632c23b278636"
- 그 외 여러 가지... 일단 우리는 SHA-1을 쓸 예정

해시가 뚫린다?

- 가입할 때 비밀번호는 해시해서 DB에 저장
  - 사용자 A가 입력한 비밀번호를 a, 해시된 비밀번호를 b라고 하자
- 그런데 해커가 DB를 훔쳐가서, b를 알게 되었다.
- 보통은 b를 통해서 a를 알 수 없다. 따라서 해커는 사용자 A로 로그인할 수 없다.
- 하지만 해시한 결과가 b인 다른 문자열 c를 해커가 알고 있으면?(해시 충돌 - Hash collision)
  - 해커는 사용자 A의 아이디와, 비밀번호 c를 이용하여 로그인할 수 있다.
- a가 간단할 경우, 문자열과 해시된 문자열을 나열해 놓은 레인보우 테이블(Rainbow table)에서 찾아낼 수도 있다.
  - 그래서 비밀번호는 긴 게 좋다.

파이썬에서 해시 쓰기

```
>>> import hashlib
>>> a = 'some_string'
>>> hashed = hashlib.sha1(a.encode('utf-8')).hexdigest()
>>> print(hashed)
2fd73fead4611f885b475edfd7069001be7ce6b1
```

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'GET':
        return render_template('register.html')
    hashed_password =
hashlib.sha1(request.form['password'].encode('utf-
8')).hexdigest()
    user = User(username=request.form['username'],
                password=hashed_password,
                nickname=request.form['nickname'])
    db.session.add(user)
    db.session.commit()
    return redirect(url_for('index'))
```

제약 조건(Constraint)

- 모델을 정의할 때 primary_key, unique 등을 썼는데, 이런 것들을 모두 통틀어 제약 조건(Constraint)라고 한다.
- primary_key 제약은, 해당 칼럼을 통해 레코드를 유일하게 정의할 수 있다는 말.
- unique는 다른 레코드가 같은 값을 가질 수 없다는 말
- 둘이 같은 말 아닌가 싶지만, 차이점이 존재: primary_key는 하나만 존재할 수 있다.
  - id 칼럼이 primary_key라면, username을 추가로 primary_key로 지정할 수 없다.

제약조건(Constraint)

- 그런데 만약, 새 유저를 만드는데 username이 이미 테이블에 있으면?
- 이럴 경우, 커밋을 하려고 할 때 데이터베이스는 에러를 내게 된다.
- 그리고 , 이렇게 에러가 나게 되면 이 에러를 해결하기 전까진 db 세션을 이용할 수 없다.
- 그럴 때는 db.session.rollback() 을 통해 롤백을 해줘야 한다.
  - 데이터베이스 세션에 추가/삭제/변경한 정보들이 모두 사라진다.

```
>>> from app import db, User
>>> user1 = User(username='user1', password='some_password', nickname='JC')
>>> db.session.add(user1)
>>> db.session.commit()
>>> user2 = User(username='user2', password='some_password2', nickname='JC')
>>> db.session.add(user2)
>>> db.session.commit()
Traceback (most recent call last):
  File "C:\Python35-32\lib\site-packages\sqlalchemy\engine\base.py", line 1139, in _execute_context
    context)
  File "C:\Python35-32\lib\site-packages\sqlalchemy\engine\default.py", line 450, in do_execute
    cursor.execute(statement, parameters)
sqlite3.IntegrityError: UNIQUE constraint failed: user.nickname
```

(이어서)

```
>>> user3 = User(username='user3',
password='password', nickname='apzzz')
>>> db.session.add(user3)
>>> db.session.commit()
...
sqlalchemy.exc.InvalidRequestError: This
Session's transaction has been rolled back
due to a previous exception during flush.
To begin a new transaction with this
Session, first issue Session.rollback().
```

회원 가입

```
@app.route('register', methods=['GET', 'POST'])
def register():
    if request.method == 'GET':
        return render_template('register.html')
    user = User(username=request.form['username'],
                password=request.form['password'],
                nickname=request.form['nickname'])
    db.session.add(user)
    try:
        db.session.commit()
    except IntegrityError:
        db.session.rollback()
        # TODO: 유저에게 에러 메시지 보여주기
        return redirect(request.referrer)
    return redirect(url_for('index'))
```

로그인

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        return render_template('login.html')
    username = request.form['username']
    password = hashlib.sha1(request.form['password'].encode(
        'utf-8')).hexdigest()
    user = User.query.filter(username=username,
password=password).first()
    if user is None:
        return redirect(url_for('login'))
    # TODO: ???
    return redirect(url_for('index'))
```

세션

- 서버에서 쓸 수 있는, 각 접속자를 위한 저장 공간
- 요청이 끝나고 새 요청이 들어왔을 때도 값이 그대로 남아 있다

```
app.config['SECRET_KEY'] = 'secret key!'
...

def login():
    ...
    if user is None:
        return redirect(url_for('login'))
    # from flask import session
    session['username'] = username
    return redirect(url_for('index'))
```

로그인 여부 확인

```
def index():
    if 'username' in session:
        user =
User.query.filter_by(username=sesion['usern
ame']).first()
    else:
        user = None
    ...
    return render_template('index.html',
user=user, ...)
```

로그인 여부 확인 - 템플릿

```
...
{% if user is not none %}
    <p>{{ user.nickname }}님, 환영합니다!</p>
{% else %}
    <a href="{{ url_for('register') }}">가입
</a> ¦
    <a href="{{ url_for('login') }}">로그인
</a>
{% endif %}
...
```

로그아웃

- 로그인이 세션에 아이디를 등록하는 거라면, 로그아웃은 그 반대 - 세션에 있던 아이디를 삭제하는 것

로그아웃

```
@app.route('/logout')
def logout():
    if 'username' in session:
        del session['username']
    return redirect(url_for('index'))
```

before_request

- 매 함수 앞에 user를 찾는 것은 너무 불편하고, 조금만 바뀌어도 모든 곳을 수정해야 하니 이 또한 불편하다.
- 그래서 Flask에서는, 모든 요청에 대해 공통적으로 처리되어야 할 함수를 등록하는 before_request를 제공한다.
- before_request에 들어간 모든 함수(여러 개일 수 있음)는 모든 요청이 시작되기 전에 처리된다.

before_request

```
@app.before_request
def set_user():
    if 'username' in session:
        # from flask import g
        g.user =
User.query.filter_by(username=session['user
name']).first()
    else:
        g.user = None
```

g

- global variables 의 약자로 생각된다.
- 한 요청에 여러 함수가 불리는 경우, 같은 변수를 공유해야 할 일이 생긴다. 이 때, g에 변수를 넣으면 된다.
  - before_request에서 g.user 에 로그인된 유저를 등록하고, index() 등에서 g.user를 써먹는 식

flash

- Flask에서 제공하는 특수한 기능으로, 1회용 세션이라고 할 수 있다.
- 한 번 쓴 뒤에, 불러오면 사라진다.
- dictionary가 아니라 list
- 잘못된 입력 등이 들어왔을 때, 에러 메시지를 띄울 때 유용

flash 예시

```
def register():
    ...
    try:
        db.session.commit()
    except:
        # from flask import flash
        flash('에러가 발생했습니다!')
        return redirect(url_for('register'))
```

flash 예시

```
<!-- register.html -->
{% for message in get_flashed_messages() %}
<p>{{ message }}</p>
{% endfor %}
```

More Database

관계형 데이터베이스 맛보기

로그인한 유저만 글쓰게 하기

```
@app.route('/write', methods=['GET', 'POST'])
def write():
    if 'username' not in session:
        # from flask import abort, ...
        abort(403)  # 또는 redirect(url_for('index'))
    user =
User.query.filter_by(username=session['username']).firs
t()
    ...
```

Article 모델 변경

```
class Article(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    author_username = db.Column(db.Unicode(255), db.ForeignKey('user.username'))
    ...
    author = db.relationship('User', backref='articles')
```

외래키(Foreign Key)

- primary key, unique 등과 마찬가지로, 제약 조건(Constraint) 중 하나
- 이 제약조건을 가진 칼럼의 내용은, 반드시 어떤 테이블의 primary key나 unique한 칼럼의 내용 중 하나여야 한다.
  - Article 모델에 레코드가 3개고 각 레코드의 아이디가 1, 2, 3일 때, article.id를 ForeignKey로 삼는 칼럼에는 무조건 1, 2, 3만 들어가야 한다. 다른 값이 들어가면 에러.
  - 당연히

relationship

- Article 에 author = relationship('User') 라고 썼는데, 이 말은 Article 모델은 User 테이블과 관계가 있는데, author라는 이름으로 서로 연결된다는 말.
- 어떤 Article 모델 article에 대해, article.author 라고 하면 해당하는 User 자체가 나온다. 물론, article.author = User.query.first() 등으로, article.author에 바로 유저를 넣을 수도 있다.
- backref는 반대쪽에서 참조하는 법이다. 즉, Article에 author = relationship('User', backref='articles') 라고 되어 있는데, 이러면 User에서는 articles 라는 이름으로 관계된 Article 목록에 접근할 수 있다.

```
<!-- read.html -->

...

<h1>{{ article.id }}. {{ article.title }}</h1>

<p>{{ article.author.nickname }}({{ article.published_a
t }})</p>

...
```

```
def write():
    ...
    user =
User.query.filter_by(username=session['username']).firs
t()
    title = request.form['title']
    content = request.form['content']
    article = Article(author=user, title=title,
                      content=content)
    db.session.add(article)
    db.session.commit()

    return redirect(url_for('read',
article_id=article.id))
```

```
def update(article_id):
    ...
    article =
Article.query.filter_by(article_id=article_id).first()
    user =
User.query.filter_by(username=session['username']).firs
t()
    if article.author != user:
        abort(403)
    article.title = request.form['title']
    article.content = request.form['content']
    db.session.commit()


    return redirect(url_for('read',
article_id=article.id))
```

관계의 종류

- one to many
  - 테이블 A의 레코드 하나가 테이블 B의 여러 레코드와 관계가 있는 것. 이 때 테이블 B의 레코드 하나는 테이블 A의 레코드 하나와만 관계가 있을 때
  - 한 명의 유저가 여러 개의 글을 쓸 수 있다
- one to one
  - 테이블 A의 레코드 하나가 테이블 B의 레코드 하나와 관계가 있다
  - one to many의 특수한 형태
- many to many
  - 테이블 A의 레코드 하나가 테이블 B의 여러 레코드와 관계가 있고, 테이블 B의 레코드 하나가 테이블 A의 레코드 여러 개와 관계가 있는 것
  - 한 사람이 좋아요를 여러 글에 누를 수 있고, 한 글의 좋아요는 여러 사람이 누를 수 있다.

Many to many - 추천 만들기

```
like = db.Table(
    db.Column('username', db.Unicode(255), db.ForeignKey('user.username')),
    db.Column('article_id', db.Integer, db.ForeignKey('article.id'))
)


Article(db.Model):
    ...
    liked_user = db.relationship('User', secondary=like, backref='like_article')
```

Many to many - 추천 만들기

```
@app.route('/like/<int:article_id>')
def like(article_id):
    article =
Article.query.filter_by(id=article_id).first()
    if g.user in article.liked_user:
        article.liked_user.remove(g.user)
    else:
        article.liked_user.append(g.user)
    return redirect(url_for('read',
article_id=article_id))
```