Django _ Follow / like (1)

1. 기초 설정

가상환경을 on 한다. 그리고 배포하는 start 라는 파일을 가지고 시작하자.

start 파일은 movies 그리고 accounts 앱으로 구성되어 있으며

해당 파일은 지금까지 학습한 게시글 그리고 댓글의 CRUD 가 미리 구현이 되어 있는 파일이다.

여기서 학습할 내용은

1. 좋아요 기능 그리고
2. 회원별 프로필 페이지 작성 후
3. 팔로우 언팔로우 기능이다.

1. 좋아요 구현하기

좋아요를 구현하기 앞서 그전에는 댓글 달기를 통해서 1:N 관계를 구현했었다.

즉 하나의 게시글에는 여러개의 댓글이 달릴 수 있는 경우를 구현했다.

`좋아요`를 구현 한다는 것은 `N:M` 즉, 다대다 관계를 구현 한다는 의미다.

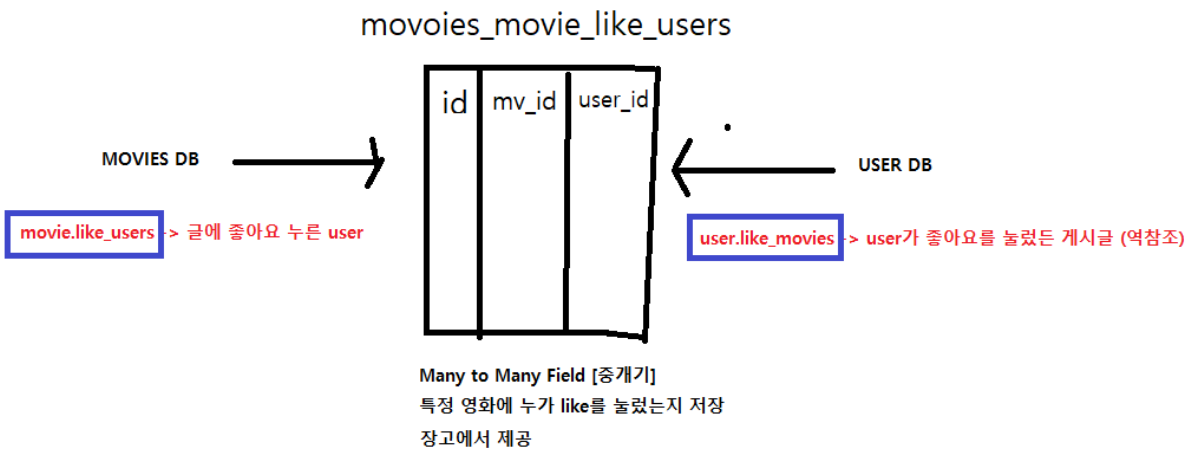
한명의 사람들이 여러 게시글을 좋아요 누를 수도 있고 게시글 하나는 여러명으로 부터 좋아요를 받을 수 있다.

다대다 (M:N, Many to Many) 관계를 구현하는데 장고에서는 ManyToManyField 라는 모델 필드를 제공해준다. 바로 이 ManyToManyField 가 좋아요를 누를 User와 좋아요가 눌릴 Article 사이에

`중개 테이블`역할을 할 것이다.

먼저 models.py 에 `like_users` 를 추가하자

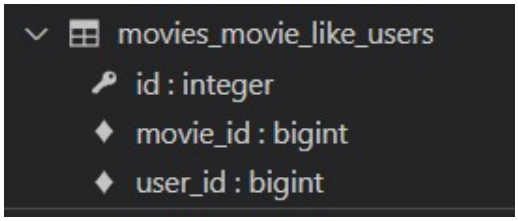
```
# movies/models.py

class Movie(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    like_users =  models.ManyToManyField(settings.AUTH_USER_MODEL, related_name='like_movies')
    title = models.CharField(max_length=20)
    description = models.TextField()
```



like_users 필드를 모델에 추가 함으로써 model 변경이 있으니 migrate를 해주자

```
$ python manage.py makemigrations
$ python manage.py migrate
```

생성된 중개 테이블을 확인 할 수 있다.



여기서 모델관계를 정리 한번 하고 진행하자. 키워드가 나중에 혼란스럽지 않도록 중간 정리를 하는 것이다.

게시글 입장에서 보자.

`movie.user` 라고 표현을 한다면 특정 영화 게시글을 작성한 유저가 될 것이고

`movie.like_users`는 특정 게시글에 좋아요를 누른 유저가 될 것이다.

유저 입장에서는 (*역참조)

`user.movie_set.all`라고 표현을 한다면 특정 유저가 작성한 모든 게시글을 의미하며

`user.like_movies`는 특정 유저가 좋아요한 게시글이 되는 것이다 좋아요 그리고 팔로우 기능을 구현하면서 뒤에서 한번 더 살펴 보자.

자 이제 본격적으로 구현을 해보자.

```
# movies/urls.py

urlpatterns = [
  ...
  path('<int:movie_pk>/likes/', views.likes, name='likes'),
]
```

```
# movies/views.py
@require_POST
def likes(request, movie_pk):
    if request.user.is_authenticated:
        movie = Movie.objects.get(pk=movie_pk)
        if movie.like_users.filter(pk=request.user.pk).exists():
            movie.like_users.remove(request.user)
        else:
            movie.like_users.add(request.user)
        return redirect('movies:detail', movie_pk)
    return redirect('accounts:login')
```

로그인이 되었다면

if 만약에 아티클에 좋아요 한 유저들 중에 해당 유저가 존재 한다면 .exits() 를 통해 True를 반환하여 해당 유저의 라이크를 지우고

else 그게 아니라면 해당 유저의 라이크를 더하라는 의미다.

로그인이 되어있지 않다면 return redirect('accounts:login') 즉, 로그인 페이지로 리다이렉트 한다.

다음, movie 게시글의 detail 페이지에 좋아요 버튼을 달아보자.

```
# movies/detail.html

<div>
    <h5>{{ movie.title }}</h5>
    <p>{{ movie.description }}</p>

    <form action="{% url 'movies:likes' movie.pk %}" method="POST">
      {% csrf_token %}
      {% if request.user in movie.like_users.all %}
        <input type="submit" value="좋아요 취소">
      {%  else %}
        <input type="submit" value="좋아요">
      {%  endif %}
    </form>

    ...

    {% if user == movie.user %}
```

if 만약에 `movie.like_users.all` (해당 영화에 라이크를 누른 모드 유저중에) `in` 방금 라이크를 누른 해당 유저 `request.user` 가 이미 있다면 “좋아요 취소” 가 작동 될 것이고 else 그게 아니면 “좋아요”가 눌릴 것이다.

서버를 켜서 게시글의 디테일 페이지에 들어가서 좋아요를 눌러보자.

"좋아요"를 클릭할 수 있는 기능을 구현 했는데 여러 번 클릭을 해보면 해당 웹페이지에 새로고침, 즉 웹 페이지를 새로 불러오게 된다. 이는 차후에 javascript를 학습한 후 "비동기통신"을 통해 새로고침 없이 좋아요가 눌리는 부드러운 사용자 경험을 제공할 것이다.

2. 팔로우 구현하기

앞에서 "좋아요" 기능을 구현하면서 `N:M` 관계를

장고에서 제공해주는 ManyToManyField 중개기(or 브릿지테이블) 을 통해서

`게시글` 과 <-> `User` 를 연결시켰다.

팔로우 기능은 비슷하지만 조금 다르다. 팔로우 기능 역시 `N:M` 관계라는 점은 위에서 구현한

“좋아요”와 비슷한 상황이지만 브릿지테이블을 통해서 연결 할 대상이 조금 다르다.

“좋아요” 에서의 N:M 관계는 `게시글` 과 <-> `User` 였다면

팔로우 에서의 N:M 관계는 `User` 와 <-> `User` 가 될 것이다.

팔로우 기능을 구현하기 앞서 각 `User`들의 프로필(Profile)을 볼 수 있도록 Profile 페이지를 먼저 구현하도록 하자.

```
# accounts/urls.py

urlpatterns = [
   ...
    path('profile/<username>/', views.profile, name='profile'),
]
```

```
# accounts/views.py

from django.contrib.auth import get_user_model

def profile(request, username):
    User = get_user_model()
    person = User.objects.get(username=username)
    context = {
     'person': person,
    }
    return render(request, 'accounts/profile.html', context)
```

accounts에 profile.html 파일을 하나 생성하자

```
# accounts/templates/accounts/profile.html

{%  extends 'base.html' %}

{%  block content %}
  <h1>{{  person.username }}님의 프로필</h1>
  <hr>
  <h2>{{  person.username }}님의 게시글</h2>
  {% for movie in person.movie_set.all %}
    <div>{{ movie.title }}</div>
  {% endfor %}
  <hr>

  <h2>{{  person.username }}님이 작성한  댓글</h2>
```

```
    {% for comment in person.comment_set.all %}
      <div>{{ comment.content }}</div>
    {% endfor %}
  <hr>

  <h2>{{  person.username }}님이 좋아요한 게시글</h2>
    {% for movie in person.like_movies.all %}
      <div>{{ movie.title }}</div>
    {% endfor %}
  <hr>

  <a href="{% url 'movies:index' %}">back</a>
{% endblock content %}
```

base.html에 내 프로필 을 추가하고

index 페이지에 작성자를 클릭시 작성자의 해당 프로필 페이지로 넘어 갈 수 있도록 수정해 보자.

```
# base.html

<body>
    <nav>
      {% if user.is_authenticated %}
        <h3>Hello, {{ user.username }}</h3>
        <a href="{% url 'accounts:profile' user.username %}">내 프로필</a>
        ...
        <a href="{% url 'accounts:update' %}">회원정보수정</a>
```

```
# movies/index.html

<p>작성자 :  <a href="{% url 'accounts:profile' movie.user.username %}">{{ movie.user }}</a></b>
```

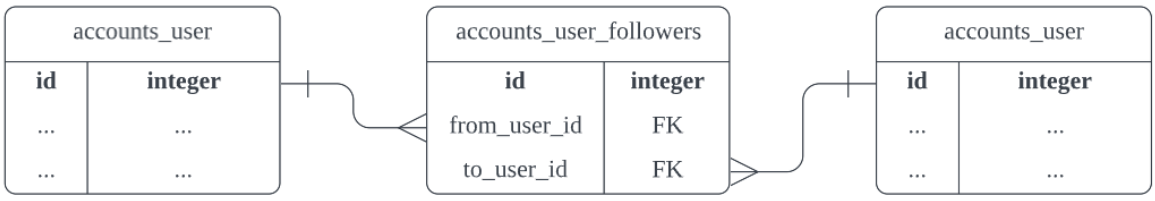
이제 user의 프로필 페이지가 완성 되었다.

내 프로필 또는 작성자를 누르면 해당 유저의 프로필 페이지로 이동 할 것이다.

자 이제 팔로우 기능을 구현 할 준비를 마쳤다.

위에서 언급 한 대로 “팔로우” 에서의 `N:M` 관계는 From User to User 가 될 것이다. 즉 이것 또한 “좋아요”와 마찬가지로 브릿지 테이블로 `ManyToManyField` 로 구성되어야 한다.

테이블을 쪼개보면 다음과 같아지는데,



즉, 양쪽에 `accounts_user` 가 있고, 하나의 브릿지테이블이 차지하고 있는 형태가 된다.

다시 말하자면, 자기 자신에게 ManyToMany 관계를 준 것이라고 할 수 있다.

이에, `accounts_user_followers` 를 포함한 모델을 만들어보면 다음과 같다.

```
# accounts/models.py
```

```
from django.db import models
from django.contrib.auth.models import AbstractUser


class User(AbstractUser):
    followings = models.ManyToManyField('self', symmetrical=False, related_name='followers')
```

`ManyToManyField` 함수를 구현하기 위해 고려해야 할 사항은 다음과 같다.

첫번째로, 유저가 유저에게 ManyToMany 를 한 경우이므로, 첫번째 인자에 자기 자신을 의미하는 `'self'` 를 넣어줘야한다.

두번째, 대칭인지 비대칭인지 고려해서 `symmetrical` 을 지정해야 한다.

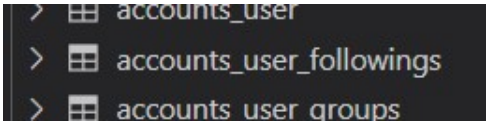
자세히보면, `from_user_id` 에서 `to_user_id` 로, 오로지 한쪽에서만 다른 유저에게로 팔로우를 할 수 있는것으로 확인되는데, 한쪽이 팔로우를 했다고 해서 다른 유저가 똑같이 팔로우함을 의미하진 않는다. 즉, “대칭이 되지 않는다.” 이를 비대칭 재귀 참조라고 하며, 이런 테이블을 만들 땐 `symmetrical=False` 옵션을 지정해주면 된다.

- 그렇다면, `symmetrical=True` 가 되는 관계는 대칭 되는 관계란 뜻인데, 어떤 경우가 있을까? 한쪽이 친구 추가를 하면 다른 쪽도 자동으로 친구가 되어서 양쪽에서 친구로 등록되는 경우 대칭이라고 할 수 있다.

브릿지 테이블의 이름은 `followings` 이고, 이 안에 양쪽의 FK 가 담길 것이다. 그리고 접근은 `followers` 로 한다. 즉, `related_name='followers'` 로 지정 해 주었으므로 해당 테이블에 역으로 접근할 땐 `followers` 라는 이름으로 접근한다.

```
$ python manage.py makemigrations
$ python manage.py migrate
```

SQLITE EXPLORER를 새로고침 후 생성된 중개 테이블을 확인하자.



그리고, `accounts/urls.py` 는 다음과 같이 추가한다.

```
urlpatterns = [
    ...
    path('<int:user_pk>/follow/', views.follow, name='follow'),
]
```

`accounts/views.py` 는 다음과 같이 추가한다.

```
@require_POST
def follow(request, user_pk):
    if request.user.is_authenticated:
        User = get_user_model()
        person = User.objects.get(pk=user_pk)
        if person != request.user:
            if person.followers.filter(pk=request.user.pk).exists():
                person.followers.remove(request.user)
            else:
                person.followers.add(request.user)
        return redirect('accounts:profile', person.username)
    return redirect('accounts:login')
```

좋아요를 구현 했을때와 비슷 한 코드 이다. 자기애가 아무리 넘치더라도 자기 자신이 자기 스스로를 팔로우 하지 않으므로 if 만약에 팔로우의 대상(person) 과 팔로워 (request.user)가 달라야 하고

if 만약에 팔로우를 시전한 유저가 이미 팔로워 라면 remove 하고

else 그게 아니라면 팔로우 add 하겠다는 의미가 되겠다.

다음 user의 프로필에 팔로잉 / 팔로워수 그리고 팔로우 언팔로우 를 클릭 할 버튼을 만들어 보자

```
# accounts/profile.html

{%  block content %}
  <h1>{{ person.username }}님의 프로필</h1>
  <div>
    <div>
      팔로잉 : {{ person.followings.all|length }} /   팔로워 : {{   person.followers.all|length }}
    </div>
    {% if request.user != person %}
      <div>
        <form action="{% url 'accounts:follow' person.pk %}" method="POST">
          {% csrf_token %}
          {% if request.user in person.followers.all %}
            <input type="submit" value="Unfollow">
          {%  else %}
            <input type="submit" value="Follow">
          {%  endif %}
        </form>
```

```
      </div>
    {%  endif %}
  </div>
```

테스트해보자. 적어도 두 명의 유저로 로그인을 바꿔보면서, 팔로잉, 팔로워가 어떻게 동작되는지 확인해보자.

<끝>

3. Fixtures

fixture란?

```
협업시 협업 파트너에게 작업 파일을 넘겨 줄때 내가 작업하며 생성된 DB는 넘겨주지 않는다.

경우에 따라 협업 상대는 DB가 필요 없을 수도 있고 또는 DB의 크기가 커서 git에 DB는 ignore를 할 것이다. 또한 본
 프로젝트를 이어 받는 사람이 DB 관리를 위해 sqlite3를 사용하지 않을 수 도 있다. 즉, Mysql 또는 PostgresSQL
 또는 Oracle 등을 쓸수도 있다!

따라서 파일을 넘길때 모델과 함께 DB를 넘기지 않다 보니 데이터가 일체 없는 상태로 파일을 받는 협업 파트너 입장에서는
 데이터 없는 빈 프로젝트를 가지고는 본 프로젝트를 파악하는데 어려움이 있을 수도 있다. 그래서 상대가 빠르게 프로젝트를
파악하고 간단한 테스트를 할수 있도록 초기 initial data를 제공해 준다.
```

우리도 초기 데이터 json 파일을 넣어 봄으로써 현재 프로젝트가 잘 작동되는지 확인하고 자 한다.

위에서 좋아요 팔로우 진행한 프로젝트에 이어서 생성한 데이터 들을 추출 (`dumpdata`) 을 먼저 할 것이다.

추출 후에는 DB 내용물을 깨끗하게 비운 후 아까 추출한 데이터를 다시 데이터 입력 (`loaddata`)를 실습 해 볼 것이다.

dumpdata

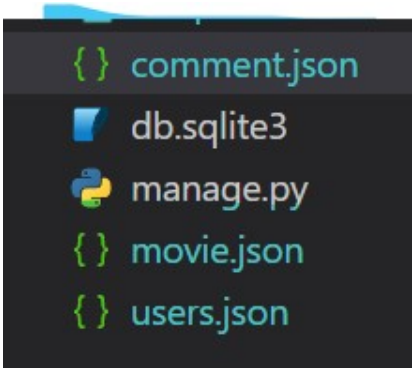
`dumpdata`는 현재 DB에 있는 data를 추출하는 것을 의미한다.

현재 프로젝트의 각각의 앱 (articles , movies) 에서의 data들를 json형식의 파일로 각각 추출할 것이다.

```
$ python -Xutf8  manage.py dumpdata --indent 4 accounts.User > users.json
$ python -Xutf8  manage.py dumpdata --indent 4 movies.Movie > movie.json
$ python -Xutf8  manage.py dumpdata --indent 4 movies.Comment > comment.json
```

라이브 교재와 다르게 -Xutf8 를 적어 주었는데 이는 한글깨짐 UTF-8 인코딩 문제를 사전에 방지하기 위해서 적어 두었다.

[참고] python - Django dumpdata fails on special characters - Stack Overflow



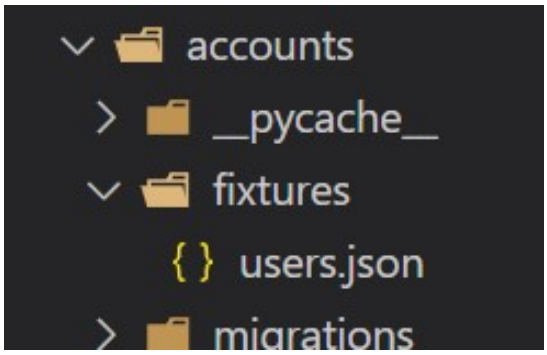
자 이렇게 `dumpdata`를 통해서 json 의 형식으로 data 추출에 성공 했다면 이번에는`loaddata`를 해 보자.

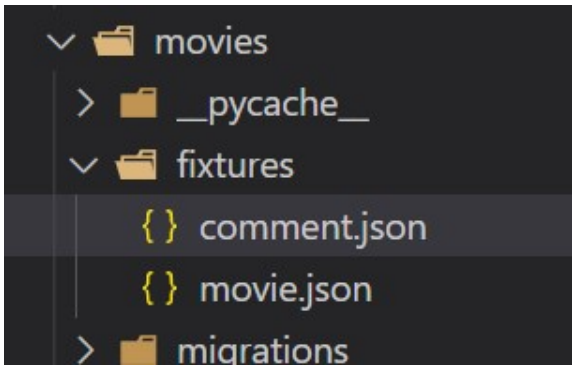
data를 load를 하기 전에 각각의 앱에 fixtures 디렉터리 생성 후 json 파일을 배치 할 것이다.

(디렉터리 이름은 반드시 fixtures 라고 해야 장고가 알아듣는다)

/movies/fixtures/ 폴더를 생성 후 movie.json 파일과 comment.json 파일을 위치 시키고

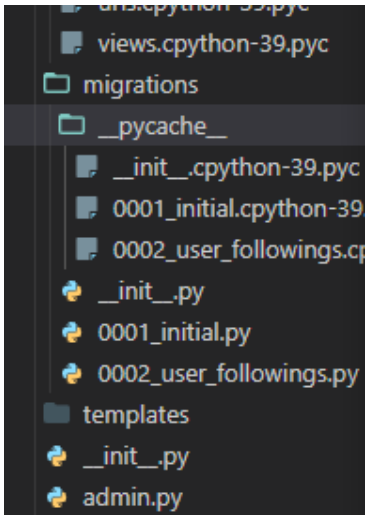
/accounts/fixtures/ 폴더를 생성 후 users.json 파일을 위치 시킬 것이다.





그리고 data들을 load하기 전에 각각의 앱의 migrations 디렉토리에서

init을 제외하고 전부 지운 후 db.sqlite3 DB도 지워버리자. (DB 날려버리기)



migrations 파일에 _init_.py 파일 아래에 있는 파일을 지운다.

pycache 파일은 migration파일에 작성 된 모듈 import시 조금 더 빠르게 하기 위하여 캐싱 해 놓은 것이라 지워도 or 안지워도 괜찮겠습니다.

```
$ python manage.py makemigrations
$ python manage.py migrate
```

db.sqlite3 에서 데이터가 모두 날아간 것을 확인 한 이후, 다음 명령어로 fixtures 에 있는 json형식의 data들을 현재의 프로젝트에 전체 로드 할 것이다.

```
$ python manage.py loaddata movie.json comment.json users.json
$ python manage.py migrate
```

그리고 db.sqlite3 를 열어 데이터 테이블들을 확인해보자. 서버 동작 시켜서 index페이지에 data들이 잘 load 되는지 확인해 보자.

<끝>