

Database

INDEX

- Many to many relationship
- M:N (Article-User)
 - Like
- M:N (User-User)
 - Follow
- Extra
 - Fixtures
 - Improve query

**Many to many
relationship**

Many to many relationship

RDB에서의 관계 복습

1. 1:1

- One-to-one relationships
- 한 테이블의 레코드 하나가 다른 테이블의 레코드 단 한 개와 관련된 경우

2. N:1

- Many-to-one relationships
- 한 테이블의 0개 이상의 레코드가 다른 테이블의 레코드 한 개와 관련된 경우
- 기준 테이블에 따라(1:N, One-to-many relationships)이라고도 함

3. **M:N**

- Many-to-many relationships
- 한 테이블의 0개 이상의 레코드가 다른 테이블의 0개 이상의 레코드와 관련된 경우
- 양쪽 모두에서 N:1 관계를 가짐

Intro

| 개요

- 병원에 내원하는 환자와 의사의 예약 시스템을 구축하라는 업무를 지시 받음
 - 필요한 데이터 베이스 모델을 고민해보고 모델링 진행하기
 - 모델링을 하는 이유는 현실 세계를 최대한 유사하게 반영하기 위함
- 무엇부터 고민해야 할까?
 - 병원 시스템에서 가장 핵심이 되는 것은? → 의사와 환자
 - 이 둘의 관계를 어떻게 표현할 수 있을까?
- 우리 일상에 가까운 예시를 통해 DB를 모델링하고 그 내부에서 일어나는 데이터의 흐름을 어떻게 제어할 수 있을지 고민해보기

[참고] 데이터 모델링

- 주어진 개념으로부터 논리적인 데이터 모델을 구성하는 작업
- 물리적인 데이터베이스 모델로 만들어 고객의 요구에 따라 특정 정보 시스템의 데이터베이스에 반영하는 작업

| 시작하기 전 용어 정리

- target model
 - 관계 필드를 가지지 않은 모델
- source model
 - 관계 필드를 가진 모델

| N:1의 한계 (1/6)

- 의사와 환자간 예약 시스템을 구현
- 지금까지 배운 N:1 관계를 생각해 한 명의 의사에게 여러 환자가 예약할 수 있다고 모델 관계를 설정

```
# hospitals/models.py

class Doctor(models.Model):
    name = models.TextField()

    def __str__(self):
        return f'{self.pk}번 의사 {self.name}'

class Patient(models.Model):
    doctor = models.ForeignKey(Doctor, on_delete=models.CASCADE)
    name = models.TextField()

    def __str__(self):
        return f'{self.pk}번 환자 {self.name}'
```

| N:1의 한계 (2/6)

- Migration 진행 및 shell_plus 실행

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

N:1의 한계 (3/6)

- 각각 2명의 의사와 환자를 생성하고 환자는 서로 다른 의사에게 예약을 했다고 가정

```
doctor1 = Doctor.objects.create(name='alice')
doctor2 = Doctor.objects.create(name='bella')
patient1 = Patient.objects.create(name='carol', doctor=doctor1)
patient2 = Patient.objects.create(name='dane', doctor=doctor2)
```

```
doctor1
<Doctor: 1번 의사 alice>
```

```
doctor2
<Doctor: 2번 의사 bella>
```

```
patient1
<Patient: 1번 환자 carol>
```

```
patient2
<Patient: 2번 환자 dane>
```

hospitals_doctor

id	name
1	alice
2	bella

hospitals_patient

id	name	doctor_id
1	carol	1
2	dane	2

| N:1의 한계 (4/6)

- 1번 환자(carol)가 두 의사 모두에게 방문하려고 함

```
patient3 = Patient.objects.create(name='carol', doctor=doctor2)
```

hospitals_doctor

id	name
1	alice
2	bella

hospitals_patient

id	name	doctor_id
1	carol	1
2	dane	2
3	carol	2

| N:1의 한계 (5/6)

- 동시에 예약 할 수는 없을까?

```
patient4 = Patient.objects.create(name='carol', doctor=doctor1, doctor2)
File "<ipython-input-9-6edaf3ffb4e6>", line 1
    patient4 = Patient.objects.create(name='carol', doctor=doctor1, doctor2)
                                   ^
SyntaxError: positional argument follows keyword argument
```

hospitals_doctor

id	name
1	alice
2	bella

hospitals_patient

id	name	doctor_id
1	carol	1
2	dane	2
3	carol	2
4	carol	1, 2

| N:1의 한계 (6/6)

- 동일한 환자지만 다른 의사에게 예약하기 위해서는 객체를 하나 더 만들어서 예약을 진행해야 함
 - 새로운 환자 객체를 생성할 수 밖에 없음
- 외래 키 컬럼에 '1, 2' 형태로 참조하는 것은 Integer 타입이 아니기 때문에 불가능
- 그렇다면 “예약 테이블을 따로 만들자”

중개 모델 (1/6)

- 환자 모델의 외래 키를 삭제하고 별도의 예약 모델을 새로 작성
- 예약 모델은 의사와 환자에 각각 N:1 관계를 가짐

```
# hospitals/models.py

# 외래키 삭제
class Patient(models.Model):
    name = models.TextField()

    def __str__(self):
        return f'{self.pk}번 환자 {self.name}'

# 중개모델 작성
class Reservation(models.Model):
    doctor = models.ForeignKey(Doctor, on_delete=models.CASCADE)
    patient = models.ForeignKey(Patient, on_delete=models.CASCADE)

    def __str__(self):
        return f'{self.doctor_id}번 의사의 {self.patient_id}번 환자'
```

hospitals_reservation		
id	doctor_id	patient_id
.	.	.

| 중개 모델 (2/6)

- 데이터베이스 초기화 후 Migration 진행 및 shell_plus 실행

1. migration 파일 삭제
2. 데이터베이스 파일 삭제

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


| 중개 모델 (3/6)

- 의사와 환자 생성 후 예약 만들기

```
doctor1 = Doctor.objects.create(name='alice')
patient1 = Patient.objects.create(name='carol')

Reservation.objects.create(doctor=doctor1, patient=patient1)
```

hospitals_doctor

id	name
1	alice

hospitals_patient

id	name
1	carol

hospitals_reservation

id	doctor_id	patient_id
1	1	1

| 중개 모델 (4/6)

- 예약 정보 조회

```
# 의사 -> 예약 정보 찾기  
doctor1.reservation_set.all()  
<QuerySet [<Reservation: 1번 의사의 1번 환자>]>
```

```
# 환자 -> 예약 정보 찾기  
patient1.reservation_set.all()  
<QuerySet [<Reservation: 1번 의사의 1번 환자>]>
```

| 중개 모델 (5/6)

- 1번 의사에게 새로운 환자 예약이 생성 된다면

```
patient2 = Patient.objects.create(name='dane')  
Reservation.objects.create(doctor=doctor1, patient=patient2)
```

hospitals_doctor

id	name
1	alice

hospitals_patient

id	name
1	carol
2	dane

hospitals_reservation

id	doctor_id	patient_id
1	1	1
2	1	2

| 중개 모델 (6/6)

- 1번 의사의 예약 정보 조회

```
# 의사 -> 환자 목록  
doctor1.reservation_set.all()  
<QuerySet [<Reservation: 1번 의사의 1번 환자>, <Reservation: 1번 의사의 2번 환자>]>
```

Django ManyToManyField (1/9)

- 환자 모델에 Django ManyToManyField 작성

```
# hospitals/models.py

class Patient(models.Model):
    # ManyToManyField 작성
    doctors = models.ManyToManyField(Doctor)
    name = models.TextField()

    def __str__(self):
        return f'{self.pk}번 환자 {self.name}'

# Reservation Class 주석 처리
```

Django ManyToManyField (2/9)

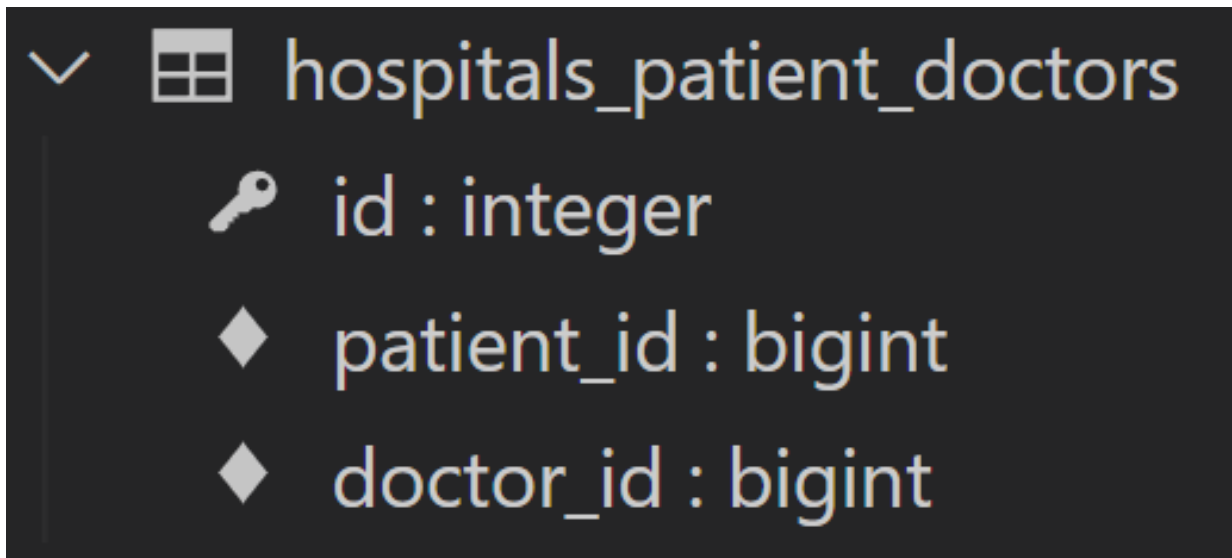
- 데이터베이스 초기화 후 Migration 진행 및 shell_plus 실행

1. migration 파일 삭제
2. 데이터베이스 파일 삭제




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

Django ManyToManyField (3/9)

- 생성된 중개 테이블 hospitals_patient_doctors 확인



A screenshot of a database table structure for the table `hospitals_patient_doctors`. The table is shown with a checkmark icon and a grid icon. The fields are listed below the table name:

Field	Type
 <code>id</code>	<code>integer</code>
 <code>patient_id</code>	<code>bigint</code>
 <code>doctor_id</code>	<code>bigint</code>

| Django ManyToManyField (4/9)

- 의사 1명과 환자 2명 생성

```
doctor1 = Doctor.objects.create(name='alice')  
patient1 = Patient.objects.create(name='carol')  
patient2 = Patient.objects.create(name='dane')
```


Django ManyToManyField (5/9)

- 예약 생성 (환자가 의사에게 예약)

```
# patient1이 doctor1에게 예약
patient1.doctors.add(doctor1)

# patient1 - 자신이 예약한 의사목록 확인
patient1.doctors.all()
<QuerySet [<Doctor: 1번 의사 alice>]>

# doctor1 - 자신의 예약된 환자목록 확인
doctor1.patient_set.all()
<QuerySet [<Patient: 1번 환자 carol>]>
```

Django ManyToManyField (6/9)

- 예약 생성 (의사가 환자를 예약)

```
# doctor1이 patient2을 예약
doctor1.patient_set.add(patient2)

# doctor1 - 자신의 예약 환자목록 확인
doctor1.patient_set.all()
<QuerySet [<Patient: 1번 환자 carol>, <Patient: 2번 환자 dane>]>

# patient1, 2 - 자신이 예약한 의사목록 확인
patient1.doctors.all()
<QuerySet [<Doctor: 1번 의사 alice>]>

patient2.doctors.all()
<QuerySet [<Doctor: 1번 의사 alice>]>
```

Django ManyToManyField (7/9)

- 예약 현황 확인

id	patient_id	doctor_id
1	1	1
2	2	1

Django ManyToManyField (8/9)

- 예약 취소하기 (삭제)
- 기존에는 해당하는 Reservation을 찾아서 지워야 했다면, 이제는 .remove() 사용

```
# doctor1이 patient1 진료 예약 취소  
  
doctor1.patient_set.remove(patient1)  
  
doctor1.patient_set.all()  
<QuerySet [<Patient: 2번 환자 harry>]>  
  
patient1.doctors.all()  
<QuerySet []>
```

```
# patient2가 doctor1 진료 예약 취소  
  
patient2.doctors.remove(doctor1)  
  
patient2.doctors.all()  
<QuerySet []>  
  
doctor1.patient_set.all()  
<QuerySet []>
```

| Django ManyToManyField (9/9)

- Django는 ManyToManyField를 통해 중개 테이블을 자동으로 생성함

| 'related_name' argument (1/3)

- target model이 source model을 참조할 때 사용할 manager name
- ForeignKey()의 related_name과 동일

```
class Patient(models.Model):  
    # ManyToManyField - related_name 작성  
    doctors = models.ManyToManyField(Doctor, related_name='patients')  
    name = models.TextField()  
  
    def __str__(self):  
        return f'{self.pk}번 환자 {self.name}'
```

| 'related_name' argument (2/3)

- Migration 진행 및 shell_plus 실행

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

| 'related_name' argument (3/3)

- related_name 설정 값 확인하기

```
# 1번 의사 조회하기
doctor1 = Doctor.objects.get(pk=1)

# 에러 발생 (related_name 을 설정하면 기존 _set manager는 사용할 수 없음)
doctor1.patient_set.all()
AttributeError: 'Doctor' object has no attribute 'patient_set'

# 변경 후
doctor1.patients.all()
<QuerySet []>
```


| 'through' argument (1/7)

- 그렇다면 중개 모델을 직접 작성하는 경우는 없을까?
 - 중개 테이블을 수동으로 지정하려는 경우 **through** 옵션을 사용하여
사용하려는 중개 테이블을 나타내는 Django 모델을 지정할 수 있음
- 가장 일반적인 용도는 **중개테이블에 추가 데이터를 사용해 다대다 관계와
연결하려는 경우**

| 'through' argument (2/7)

- through 설정 및 Reservation Class 수정
 - 이제는 예약 정보에 증상과 예약일이라는 추가 데이터가 생김

```
class Patient(models.Model):
    doctors = models.ManyToManyField(Doctor, through='Reservation')
    name = models.TextField()

    def __str__(self):
        return f'{self.pk}번 환자 {self.name}'

class Reservation(models.Model):
    doctor = models.ForeignKey(Doctor, on_delete=models.CASCADE)
    patient = models.ForeignKey(Patient, on_delete=models.CASCADE)
    symptom = models.TextField()
    reserved_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f'{self.doctor.pk}번 의사의 {self.patient.pk}번 환자'
```

| 'through' argument (3/7)

- 데이터베이스 초기화 후 Migration 진행 및 shell_plus 실행

1. migration 파일 삭제
2. 데이터베이스 파일 삭제

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

| 'through' argument (4/7)

- 의사 1명과 환자 2명 생성

```
doctor1 = Doctor.objects.create(name='alice')  
patient1 = Patient.objects.create(name='carol')  
patient2 = Patient.objects.create(name='dane')
```

| 'through' argument (5/7)

- 예약 생성 1

```
# 1. Reservation class를 통한 예약 생성
```

```
reservation1 = Reservation(doctor=doctor1, patient=patient1, symptom='headache')  
reservation1.save()
```

```
doctor1.patient_set.all()  
<QuerySet [<Patient: 1번 환자 carol>]>
```

```
patient1.doctors.all()  
<QuerySet [<Doctor: 1번 의사 alice>]>
```

| 'through' argument (6/7)

- 예약 생성 2

```
# 2. Patient 객체를 통한 예약 생성
```

```
patient2.doctors.add(doctor1, through_defaults={'symptom': 'flu'})
```

```
doctor1.patient_set.all()
```

```
<QuerySet [<Patient: 1번 환자 carol>, <Patient: 2번 환자 dane>]>
```

```
patient2.doctors.all()
```

```
<QuerySet [<Doctor: 1번 의사 alice>]>
```

❖ `through_defaults` 값에 딕셔너리 타입으로 입력

| 'through' argument (7/7)

- 예약 삭제

```
doctor1.patient_set.remove(patient1)  
patient2.doctors.remove(doctor1)
```

정리

- M:N 관계로 맺어진 두 테이블에는 변화가 없음
- Django의 ManyToManyField은 중개 테이블을 자동으로 생성함
- Django의 ManyToManyField는 M:N 관계를 맺는 두 모델 어디에 위치해도 상관 없음
 - 대신 필드 작성 위치에 따라 참조와 역참조 방향을 주의할 것
- N:1은 완전한 종속의 관계였지만 M:N은 의사에게 진찰받는 환자, 환자를 진찰하는 의사의 두 가지 형태로 모두 표현이 가능한 것

이어서 ..

삼성 청년 SW 아카데미

ManyToManyField

ManyToManyField 란

- `ManyToManyField(to, **options)`
- 다대다 (M:N, many-to-many) 관계 설정 시 사용하는 모델 필드
- 하나의 필수 위치인자(M:N 관계로 설정할 모델 클래스)가 필요
- 모델 필드의 `RelatedManager`를 사용하여 관련 개체를 추가, 제거 또는 만들 수 있음
 - `add()`, `remove()`, `create()`, `clear()` ...

| 데이터베이스에서의 표현

- Django는 다대다 관계를 나타내는 중개 테이블을 만듦
- 테이블 이름은 ManyToManyField 이름과 이를 포함하는 모델의 테이블 이름을 조합하여 생성됨
- ‘db_table’ arguments을 사용하여 중개 테이블의 이름을 변경할 수도 있음

| ManyToManyField's Arguments (1/5)

1. `related_name`
2. `through`
3. `symmetrical`

| ManyToManyField's Arguments (2/5)

1. related_name

- target model이 source model을 참조할 때 사용할 manager name
- ForeignKey의 related_name과 동일

| ManyToManyField's Arguments (3/5)

2. through

- 중개 테이블을 직접 작성하는 경우, through 옵션을 사용하여 중개 테이블을 나타내는 Django 모델을 지정
- 일반적으로 중개 테이블에 추가 데이터를 사용하는 다대다 관계와 연결하려는 경우(extra data with a many-to-many relationship)에 사용됨

ManyToManyField's Arguments (4/5)

3. symmetrical

- 기본 값 : True
- ManyToManyField가 동일한 모델(on self)을 가리키는 정의에서만 사용

예시

```
class Person(models.Model):  
    friends = models.ManyToManyField('self')  
    # friends = models.ManyToManyField('self', symmetrical=False)
```


ManyToManyField's Arguments (5/5)

3. symmetrical

- True일 경우
 - `_set` 매니저를 추가 하지 않음
 - source 모델의 인스턴스가 target 모델의 인스턴스를 참조하면 자동으로 target 모델 인스턴스도 source 모델 인스턴스를 자동으로 참조하도록 함(대칭)
 - 즉, 내가 당신의 친구라면 당신도 내 친구가 됨
- 대칭을 원하지 않는 경우 False로 설정
 - Follow 기능 구현에서 다시 확인할 예정

Related Manager

- N:1 혹은 M:N 관계에서 사용 가능한 문맥(context)
- Django는 모델 간 N:1 혹은 M:N 관계가 설정되면 역참조시에 사용할 수 있는 manager를 생성
 - 우리가 이전에 모델 생성 시 objects 라는 매니저를 통해 queryset api를 사용했던 것처럼 related manager를 통해 queryset api를 사용할 수 있게 됨
- 같은 이름의 메서드여도 각 관계(N:1, M:N)에 따라 다르게 사용 및 동작됨
 - N:1에서는 target 모델 객체만 사용 가능
 - M:N 관계에서는 관련된 두 객체에서 모두 사용 가능
- 메서드 종류
 - add(), remove(), create(), clear(), set() 등

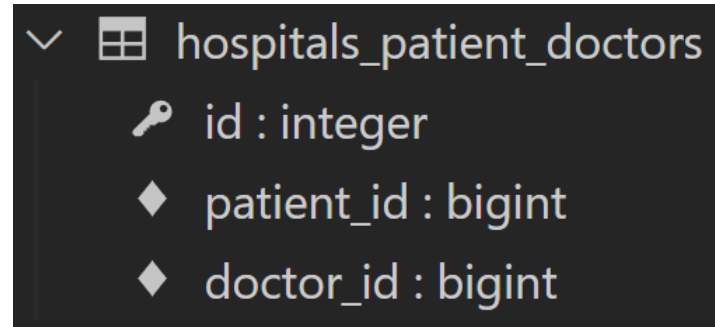
methods ❖ many-to-many relationships 일 때의 동작만 작성되었음

- **add()**
 - “지정된 객체를 관련 객체 집합에 추가”
 - 이미 존재하는 관계에 사용하면 관계가 복제되지 않음
 - 모델 인스턴스, 필드 값(PK)을 인자로 허용
- **remove()**
 - “관련 객체 집합에서 지정된 모델 개체를 제거”
 - 내부적으로 `QuerySet.delete()`를 사용하여 관계가 삭제됨
 - 모델 인스턴스, 필드 값(PK)을 인자로 허용

중개 테이블 필드 생성 규칙

1. 소스(source model) 및 대상(target model) 모델이 다른 경우

- id
- <containing_model>_id
- <other_model>_id



A screenshot of a Django ORM model definition for a ManyToManyField. The field is named 'hospitals_patient_doctors'. It has three fields: 'id' of type 'integer' (marked as a primary key with a key icon), 'patient_id' of type 'bigint' (marked with a diamond icon), and 'doctor_id' of type 'bigint' (marked with a diamond icon).

▼	hospitals_patient_doctors
🔑	id : integer
◆	patient_id : bigint
◆	doctor_id : bigint

2. ManyToManyField가 동일한 모델을 가리키는 경우

- id
- from_<model>_id
- to_<model>_id

이어서 ..

삼성 청년 SW 아카데미

M:N (Article-User)

M:N (Article-User)

| 개요

- Article과 User의 M:N 관계 설정을 통한 좋아요 기능 구현하기

LIKE

| 모델 관계 설정 (1/6)

- ManyToManyField 작성

```
# articles/models.py

class Article(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    like_users = models.ManyToManyField(settings.AUTH_USER_MODEL)
    title = models.CharField(max_length=10)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

| 모델 관계 설정 (2/6)

- Migration 진행 후 에러 확인

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

ERRORS:

```
articles.Article.like_users: (fields.E304) Reverse accessor for 'Article.like_users' clashes with reverse accessor for 'Article.user'.
```

```
    HINT: Add or change a related_name argument to the definition for 'Article.like_users' or 'Article.user'.
```

```
articles.Article.user: (fields.E304) Reverse accessor for 'Article.user' clashes with reverse accessor for 'Article.like_users'.
```

```
    HINT: Add or change a related_name argument to the definition for 'Article.user' or 'Article.like_users'.
```

| 모델 관계 설정 (3/6)

- like_users 필드 생성 시 자동으로 역참조에는 .article_set 매니저가 생성됨
- 그러나 이전 N:1(Article-User) 관계에서 이미 해당 매니저를 사용 중
 - user.article_set.all() → 해당 유저가 작성한 모든 게시글 조회
 - user가 작성한 글들(user.article_set)과
user가 좋아요를 누른 글(user.article_set)을 구분할 수 없게 됨
- user와 관계된 ForeignKey 혹은 ManyToManyField 중 하나에 related_name을 작성해야 함

| 모델 관계 설정 (4/6)

- ManyToManyField에 related_name 작성 후 Migration


```
# articles/models.py

class Article(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    like_users = models.ManyToManyField(settings.AUTH_USER_MODEL, related_name='like_articles')
    title = models.CharField(max_length=10)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

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

| 모델 관계 설정 (5/6)

- 생성된 중개 테이블 확인

```
✓  articles_article_like_users  
  🔑 id : integer  
  ♦ article_id : bigint  
  ♦ user_id : bigint
```

| 모델 관계 설정 (6/6)

- User - Article간 사용 가능한 related manager 정리
 - **article.user**
 - 게시글을 작성한 유저 - N:1
 - **user.article_set**
 - 유저가 작성한 게시글(역참조) - N:1
 - **article.like_users**
 - 게시글을 좋아요한 유저 - M:N
 - **user.like_articles**
 - 유저가 좋아요한 게시글(역참조) - M:N

| LIKE 구현 (1/5)

- url 및 view 함수 작성

```
# articles/urls.py

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

```
# articles/views.py

def likes(request, article_pk):
    article = Article.objects.get(pk=article_pk)

    if
article.like_users.filter(pk=request.user.pk).exists():
    # if request.user in article.like_users.all():
        article.like_users.remove(request.user)
    else:
        article.like_users.add(request.user)
    return redirect('articles:index')
```

`.exists()`

- QuerySet에 결과가 포함되어 있으면 True를 반환하고 그렇지 않으면 False를 반환
- 특히 큰 QuerySet에 있는 특정 개체의 존재와 관련된 검색에 유용

LIKE 구현 (2/5)

- index 템플릿에서 각 게시글에 좋아요 버튼 출력하기

```
<!-- articles/index.html -->

{% extends 'base.html' %}

{% block content %}

...
{% for article in articles %}

...
<div>
  <form action="{% url 'articles:likes' article.pk %}" method="POST">
    {% csrf_token %}
    {% if request.user in article.like_users.all %}
      <input type="submit" value="좋아요 취소">
    {% else %}
      <input type="submit" value="좋아요">
    {% endif %}
  </form>
</div>
<a href="{% url 'articles:detail' article.pk %}">DETAIL</a>
<hr>
{% endfor %}
{% endblock content %}
```

LIKE 구현 (3/5)

- 좋아요 버튼 출력 확인

Hello, test1

[정보수정](#)

Articles

[CREATE](#)

작성자 : test1

글 번호: 1

글 제목: 제목

글 내용: 내용

[DETAIL](#)

LIKE 구현 (4/5)

- 좋아요 버튼 클릭 후 좋아요 테이블 확인

Hello, test1

[정보수정](#)

[Logout](#)

[회원탈퇴](#)

Articles

[CREATE](#)

작성자 : test1

글 번호: 1

글 제목: 제목

글 내용: 내용

[좋아요 취소](#)

[DETAIL](#)

id	article_id	user_id
1	1	1

LIKE 구현 (5/5)

- 데코레이터 및 is_authenticated 추가

```
# articles/views.py

@require_POST
def likes(request, article_pk):
    if request.user.is_authenticated:
        article = Article.objects.get(pk=article_pk)

        if article.like_users.filter(pk=request.user.pk).exists():
            # if request.user in article.like_users.all():
            article.like_users.remove(request.user)
        else:
            article.like_users.add(request.user)
        return redirect('articles:index')
    return redirect('accounts:login')
```

이어서 ..

삼성 청년 SW 아카데미

M:N (User-User)

M:N (User-User)

| 개요

- User 자기 자신과의 M:N 관계 설정을 통한 팔로우 기능 구현하기

Profile

| 개요

- 자연스러운 follow 흐름을 위한 프로필 페이지를 먼저 작성

Profile 구현 (1/4)

- url 및 view 함수 작성

```
# 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)
```

Profile 구현 (2/4)

- profile 템플릿 작성

```
<!-- accounts/profile.html -->
{% extends 'base.html' %}

{% block content %}
  <h1>{{ person.username }}님의 프로필</h1>

  <hr>

  <h2>{{ person.username }}'s 게시글</h2>
  {% for article in person.article_set.all %}
    <div>{{ article.title }}</div>
  {% endfor %}

  <hr>
  ...
```

```
...
<h2>{{ person.username }}'s 댓글</h2>
  {% for comment in person.comment_set.all %}
    <div>{{ comment.content }}</div>
  {% endfor %}

  <hr>

  <h2>{{ person.username }}'s 좋아요한 게시글</h2>
  {% for article in person.like_articles.all %}
    <div>{{ article.title }}</div>
  {% endfor %}

  <hr>

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

Profile 구현 (3/4)

- Profile 템플릿으로 이동할 수 있는 하이퍼 링크 작성

```
<!-- base.html -->

<body>
  <div class="container">
    {% if request.user.is_authenticated %}
      <h3>Hello, {{ user }}</h3>
      <a href="{% url 'accounts:profile' user.username %}">내 프로필</a>
    ...
```

```
<!-- articles/index.html -->

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

Profile 구현 (4/4)

- Profile 템플릿으로 이동할 수 있는 하이퍼 링크 출력 확인

Hello, test1

[내 프로필 정보수정](#)

Logout

회원탈퇴

Articles

[CREATE](#)

작성자 : [test1](#)

글 번호: 1

글 제목: 제목

글 내용: 내용

좋아요 취소

[DETAIL](#)

Follow

| 모델 관계 설정 (1/2)

- ManyToManyField 작성 및 Migration 진행

```
# accounts/models.py

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

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

| 모델 관계 설정 (2/2)

- 생성된 중개 테이블 확인

accounts_user_followings
id : integer
from_user_id : integer
to_user_id : integer

| Follow 구현 (1/4)

- url 및 view 함수 작성

```
# accounts/urls.py

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

```
# accounts/views.py

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

Follow 구현 (2/4)

- 프로필 유저의 팔로잉, 팔로워 수 & 팔로우, 언팔로우 버튼 작성

```
<!-- accounts/profile.html -->

{% extends 'base.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>

```

Follow 구현 (3/4)

- 팔로우 버튼 클릭 후 팔로우 버튼 변화 및 테이블 확인

Hello, test2

[내 프로필 정보수정](#)

Logout

회원탈퇴

test1님의 프로필

팔로잉 : 0 / 팔로워 : 1

Unfollow

test1's 게시글

제목

test1's 댓글

test1's 좋아요한 게시글

제목

[back](#)

id	from_user_id	to_user_id
1	2	1

| Follow 구현 (4/4)

- 데코레이터 및 is_authenticated 추가

```
# accounts/views.py

@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():
                # if request.user in person.followers.all():
                person.followers.remove(request.user)
            else:
                person.followers.add(request.user)
        return redirect('accounts:profile', person.username)
    return redirect('accounts:login')
```

이어서 ..

삼성 청년 SW 아카데미

EXTRA

Fixtures

| 개요

- Fixtures를 사용해 모델에 초기 데이터 제공하는 방법

초기 데이터의 필요성

- 협업하는 A, B 유저가 있다고 생각해보자.
 1. A가 먼저 프로젝트를 작업 후 github에 push 한다.
 - gitignore 설정으로 인해 DB는 업로드하지 않기 때문에 A가 개발하면서 사용한 데이터는 올라가지 않는다.
 2. B가 github에서 A push한 프로젝트를 pull (혹은 clone) 한다.
 - 마찬가지로 프로젝트는 받았지만 A가 생성하고 조작한 데이터는 없는 빈 프로젝트를 받게 된다.
- 이처럼 Django 프로젝트의 앱을 처음 설정할 때 동일하게 준비 된 데이터로 데이터베이스를 미리 채우는 것이 필요한 순간이 있다.
- Django에서는 fixtures를 사용해 앱에 초기 데이터(initial data)를 제공할 수 있다.
- 즉, migrations와 fixtures를 사용하여 data와 구조를 공유하게 된다.

Providing data with fixtures

Providing data with fixtures

| 사전준비

- M:N 까지 모두 작성된 Django 프로젝트에서
 유저, 게시글, 댓글, 좋아요 등 각 데이터를 최소 2개 이상 생성해두기

Providing data with fixtures

fixtures

- Django가 데이터베이스로 가져오는 방법을 알고 있는 데이터 모음
 - 가져오는 방법을 알고 있다?
 - Django가 직접 만들기 때문에 데이터베이스 구조에 맞추어 작성 되어있음
- <https://docs.djangoproject.com/en/3.2/howto/initial-data/#providing-data-with-fixtures>

Providing data with fixtures

| fixtures 생성 및 로드

- 생성 (데이터 추출)
 - dumpdata
- 로드 (데이터 입력)
 - loaddata

Providing data with fixtures

| dumpdata (1/3)

- 응용 프로그램과 관련된 데이터베이스의 모든 데이터를 표준 출력으로 출력함
- 여러 모델을 하나의 json 파일로 만들 수 있음

작성 예시

```
$ python manage.py dumpdata [app_name[.ModelName] [app_name[.ModelName] ...]] > {filename}.json
```

Providing data with fixtures

| dumpdata (2/3)

- articles app의 article 모델에 대한 data를 json 형식으로 저장하기

```
$ python manage.py dumpdata --indent 4 articles.article > articles.json
```

- manage.py와 동일한 위치에 data가 담긴 articles.json 파일이 생성됨
- dumpdata의 출력 결과물은 loaddata 의 입력으로 사용됨
 - ❖ fixtures 파일은 직접 만드는 것이 아니라 dumpdata를 사용하여 생성하는 것이다.

Providing data with fixtures

| dumpdata (3/3)

- 추가로 나머지 모델에 대한 데이터를 dump 한다.

```
$ python manage.py dumpdata --indent 4 articles.user > users.json  
$ python manage.py dumpdata --indent 4 articles.comment > comments.json
```


Providing data with fixtures

[참고] 모든 모델을 한번에 dump 하기

```
# 3개의 모델을 하나의 json 파일로
```

```
$ python manage.py dumpdata --indent 4 articles.article articles.comment accounts.user > data.json
```

```
# 모든 모델을 하나의 json 파일로
```

```
$ python manage.py dumpdata --indent 4 > data.json
```

Providing data with fixtures

| loaddata (1/3)

- fixtures의 내용을 검색하여 데이터베이스로 로드

```
# 작성 예시  
$ python manage.py loaddata data.json
```

- fixtures 기본 경로
 - app_name/fixtures/
 - Django는 설치된 모든 app의 디렉토리에서 fixtures 폴더 이후의 경로로 fixtures 파일을 찾음

Providing data with fixtures

| loaddata (2/3)

- fixtures의 내용을 검색하여 데이터베이스로 로드

```
# 해당 위치로 fixture 파일 이동
```

```
articles/  
  fixtures/  
    articles.json  
    users.json  
    comments.json
```

- db.sqlite3 파일 삭제 후 migrate 작업을 진행

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

Providing data with fixtures

| loaddata (3/3)

- fixtures load 하기

```
$ python manage.py loaddata articles.json users.json comments.json
```

- load 후 데이터가 잘 입력되었는지 확인하기

Providing data with fixtures

[참고] loaddata를 하는 순서

- loaddata를 한번에 실행하지 않고 하나씩 실행한다면 모델 관계에 따라 순서가 중요할 수 있음
 - comment는 article에 대한 key 및 user에 대한 key가 필요
 - article은 user에 대한 key가 필요
- 즉, 현재 모델 관계에서는 user → article → comment 순으로 data를 넣어야 오류가 발생하지 않음

```
$ python manage.py loaddata users.json  
$ python manage.py loaddata articles.json  
$ python manage.py loaddata comments.json
```

Providing data with fixtures

[참고] loaddata 시 encoding codec 관련 에러가 발생하는 경우

- 2가지 방법 중 택 1

1. dumpdata 시 추가 옵션 작성

```
$ python -Xutf8 manage.py dumpdata [생략]
```

2. 메모장 활용

1. 메모장으로 json 파일 열기
2. "다른 이름으로 저장" 클릭
3. 인코딩을 UTF8로 선택 후 저장

Providing data with fixtures

| fixtures 정리

- fixtures 파일은 직접 만드는 것이 아니라 dumpdata를 사용하여 생성하는 것 !

이어서 ..

삼성 청년 SW 아카데미

Improve query

| 개요

- Query를 개선하는 방법
 - annotate
 - select_related
 - prefetch_related

| select_related

- 1:1 또는 N:1 참조 관계에서 사용
- SQL 에서 INNER JOIN 절을 활용
 - SQL의 INNER JOIN을 사용하여 참조하는 테이블의 일부를 가져오고, SELECT FROM을 통해 관련된 필드들을 가져옴

| prefetch_related

- M:N 또는 N:1 역참조 관계에서 사용
- SQL이 아닌 Python 을 통한 JOIN이 진행됨

| 설부른 최적화를 하지 말자

"작은 효율성(small efficiency)에 대해서는, 말하자면 97% 정도에 대해서는, 잊어버려라. 설부른 최적화 (premature optimization)는 모든 악의 근원이다."

- 도널드 커누스(Donald E. Knuth)

이어서 ..

삼성 청년 SW 아카데미

마무리

| 마무리 INDEX

- Many-to-many relationship
 - ManyToManyField()
- M:N (Article-User)
 - Like
- M:N (User-User)
 - Profile
 - Follow

다음 방송에서 만나요!

삼성 청년 SW 아카데미