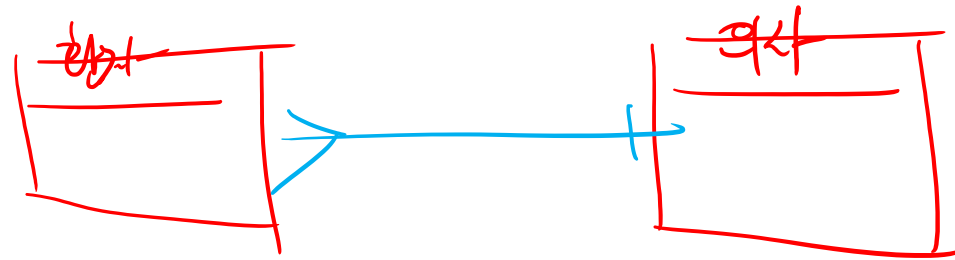


RDB에서의 관계 복습

- 1:1
 - One-to-one relationships
 - 한 테이블의 레코드 하나가 다른 테이블의 레코드 단 한 개와 관련된 경우
- 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
- M:N
 - Many-to-many relationships
 - 한 테이블의 0개 이상의 레코드가 다른 테이블의 0개 이상의 레코드와 관련된 경우
 - 양쪽 모두에서 N:1 관계를 가짐

개요

- 병원 예약 시스템 구축을 위한 데이터 베이스 모델링을 진행한다면?



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
 - migration 파일 삭제
 - 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
```

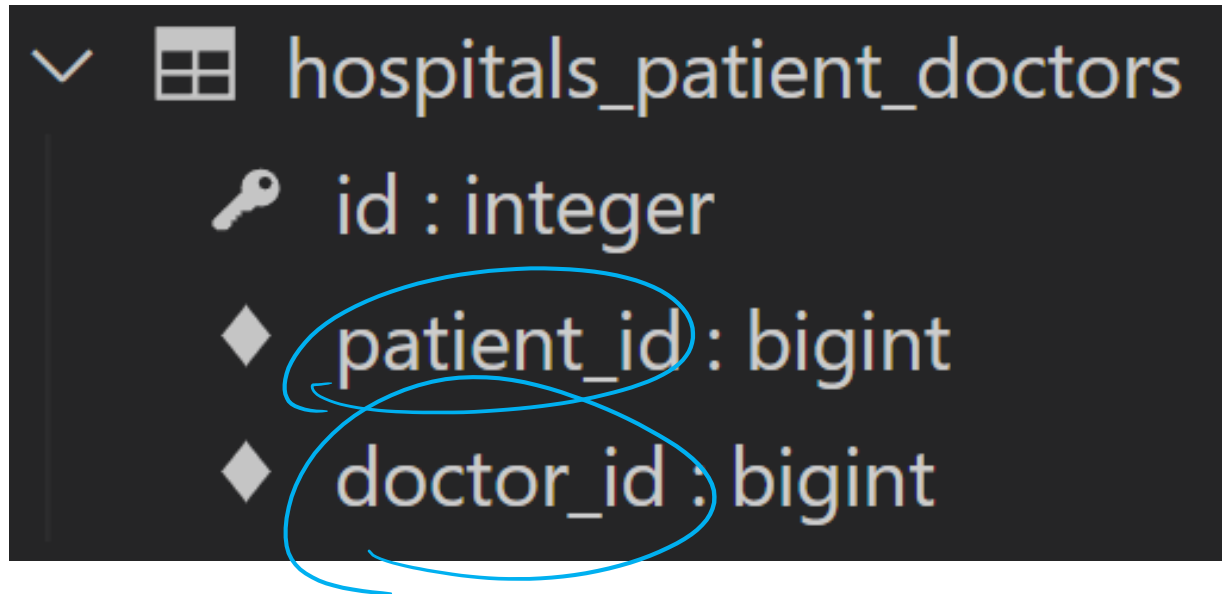

Django ManyToManyField (2/9)

- 데이터베이스 초기화 후 Migration 진행 및 shell_plus 실행
 - migration 파일 삭제
 - 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



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>]>
```

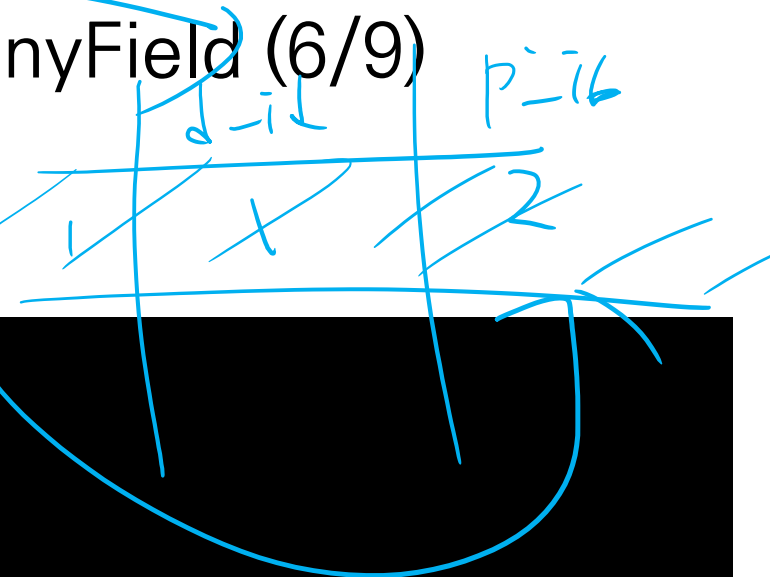
```
# patient1, 2 - 자신이 예약한 의사목록 확인
```

```
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 model | 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 (1/2)

- 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 (2/2)

- 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'
```

정리

- M:N 관계로 맺어진 두 테이블에는 변화가 없음
- Django의 ManyToManyField은 중개 테이블을 자동으로 생성함
- Django의 ManyToManyField는 M:N 관계를 가진 모델 어디에 위치해도 상관 없음
 - 대신 필드 작성 위치에 따라 참조와 역참조 방향을 주의할 것



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)

- `related_name`
- `through`
- `symmetrical`

ManyToManyField's Arguments (2/5)

- `related_name`
 - target model이 source model을 참조할 때 사용할 manager name
 - ForeignKey의 `related_name`과 동일

ManyToManyField's Arguments (3/5)

- 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)

- 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)

- 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

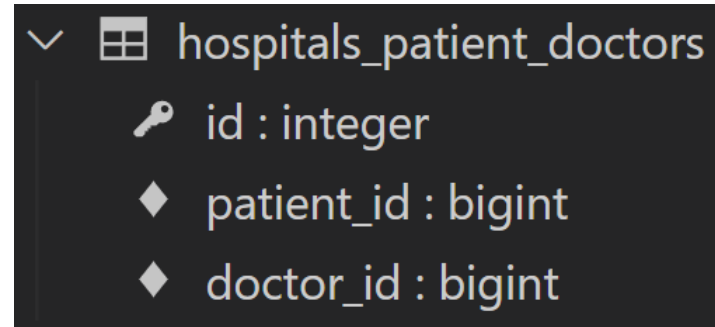
methods

- `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

- 소스(source model) 및 대상(target model) 모델이 다른 경우

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



```
▼ hospitals_patient_doctors
  🔑 id : integer
  ♦ patient_id : bigint
  ♦ doctor_id : bigint
```

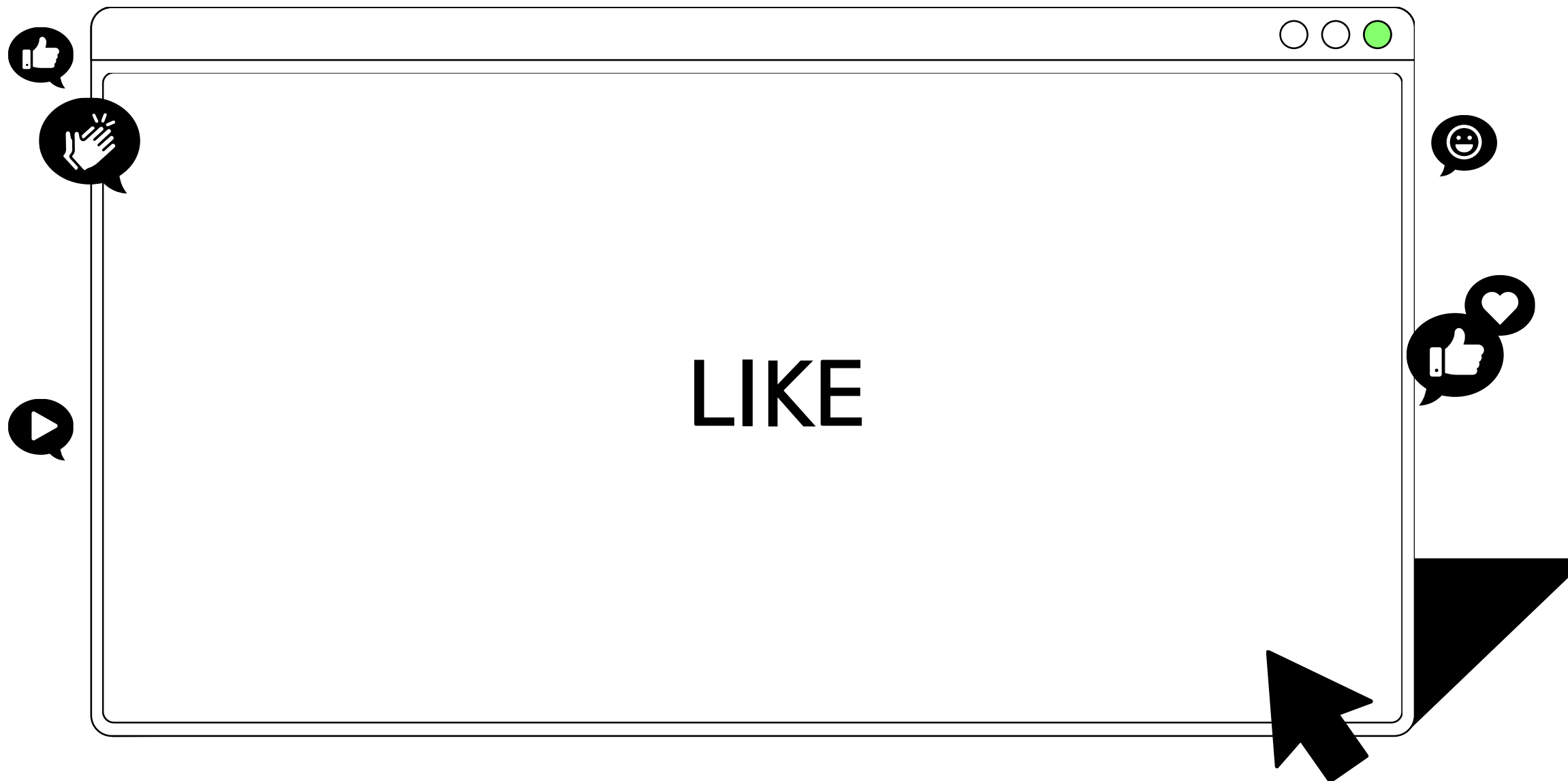
- ManyToManyField가 동일한 모델을 가리키는 경우

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



The diagram shows a white browser window with a thin black border. In the top right corner of the window are three small circles: two white and one green. The text "M:N (Article-User)" is centered in the window. Surrounding the window are several black circular icons: a thumbs-up, a clapping hand, a play button, a speech bubble with a smiley face, a thumbs-up with a heart, and a large black arrow pointing towards the bottom right corner of the window.

M:N (Article-User)



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 게시물 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 (1/4)

- 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 (2/4)

- 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 (3/4)

- 좋아요 버튼 출력 확인

Hello, test1

[정보수정](#)

Logout

회원탈퇴

Articles

[CREATE](#)

작성자 : test1

글 번호: 1

글 제목: 제목

글 내용: 내용

좋아요

[DETAIL](#)

LIKE 구현 (4/4)

- 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