

## DB 05

# Many to many relationships 01

# • 라이브 INDEX

- Many to many relationships
  - N:1의 한계
  - 중개 모델
  - ManyToManyField
  - 'through' argument
- ManyToManyField
- 좋아요 기능 구현
  - 모델 관계 설정
  - 기능 구현

# Many to many relationships

---

# 개요

# Many to many relationships

## N:M or M:N

---

한 테이블의 0개 이상의 레코드가  
다른 테이블의 0개 이상의 레코드와 관련된 경우  
❖ 양쪽 모두에서 N:1 관계를 가짐

## M:N 관계의 역할과 필요성 이해하기

---

- ‘병원 진료 시스템 모델 관계’를 만들며 M:N 관계의 역할과 필요성 이해하기
  - 환자와 의사 2개의 모델을 사용하여 모델 구조 구상하기
- 제공된 ‘99-mtm-practice’ 프로젝트를 기반으로 진행

# N:1의 한계

## 의사와 환자 간 모델 관계 설정

---

한 명의 의사에게 여러 환자가 예약할 수 있도록 설계

```
# 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}'
```



## 의사와 환자 데이터 생성

2명의 의사와 환자를 생성하고 환자는 서로 다른 의사에게 예약

```
doctor1 = Doctor.objects.create(name='allie')
doctor2 = Doctor.objects.create(name='barbie')
patient1 = Patient.objects.create(name='carol', doctor=doctor1)
patient2 = Patient.objects.create(name='duke', doctor=doctor2)
```

```
doctor1
<Doctor: 1번 의사 allie>
```

```
doctor2
<Doctor: 2번 의사 barbie>
```

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

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

**hospitals\_doctor**

id	name
1	allie
2	barbie

**hospitals\_patient**

id	name	doctor_id
1	carol	1
2	duke	2

## N:1의 한계 상황 (1/3)

1번 환자(carol)가 두 의사 모두에게 진료를 받고자 한다면  
환자 테이블에 1번 환자 데이터가 중복으로 입력될 수 밖에 없음

**hospitals\_doctor**

id	name
1	allie
2	barbie

**hospitals\_patient**

id	name	doctor_id
1	carol	1
2	duke	2
3	carol	2

## N:1의 한계 상황 (2/3)

동시에 예약을 남길 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	allie
2	barbie

**hospitals\_patient**

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

## N:1의 한계 상황 (3/3)

---

- 동일한 환자지만 다른 의사에게도 진료 받기 위해 예약하기 위해서는 객체를 하나 더 만들어 진행해야 함
  - 외래 키 컬럼에 '1, 2' 형태로 저장하는 것은 DB 타입 문제로 불가능
- “예약 테이블을 따로 만들자”

# 중개 모델

## 1. 예약 모델 생성

---

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

```
# hospitals/models.py

# 외래 키 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
.	.	.

## 2. 예약 데이터 생성

---

- 데이터베이스 초기화 후 Migration 진행 및 shell\_plus 실행
- 의사와 환자 생성 후 예약 만들기

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

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

**hospitals\_doctor**

id	name
1	allie

**hospitals\_patient**

id	name
1	carol

**hospitals\_reservation**

id	doctor_id	patient_id
1	1	1

## 3. 예약 정보 조회

---

- 의사와 환자가 예약 모델을 통해 각각 본인의 진료 내역 확인

```
# 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>]>
```



## 4. 추가 예약 생성

---

- 1번 의사에게 새로운 환자 예약 생성

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

**hospitals\_doctor**

id	name
1	allie

**hospitals\_patient**

id	name
1	carol
2	duke

**hospitals\_reservation**

id	doctor_id	patient_id
1	1	1
2	1	2

## 5. 예약 정보 조회

---

- 1번 의사의 예약 정보 조회

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

**Django에서는  
'ManyToManyField'로  
중개모델을 자동으로 생성**

# ManyToManyField

# ManyToManyField()

---

M:N 관계 설정 모델 필드

# Django ManyToManyField (1/7)

---

- 환자 모델에 ManyToManyField 작성
  - 의사 모델에 작성해도 상관 없으며 참조/역참조 관계만 잘 기억할 것

```
# hospitals/models.py

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

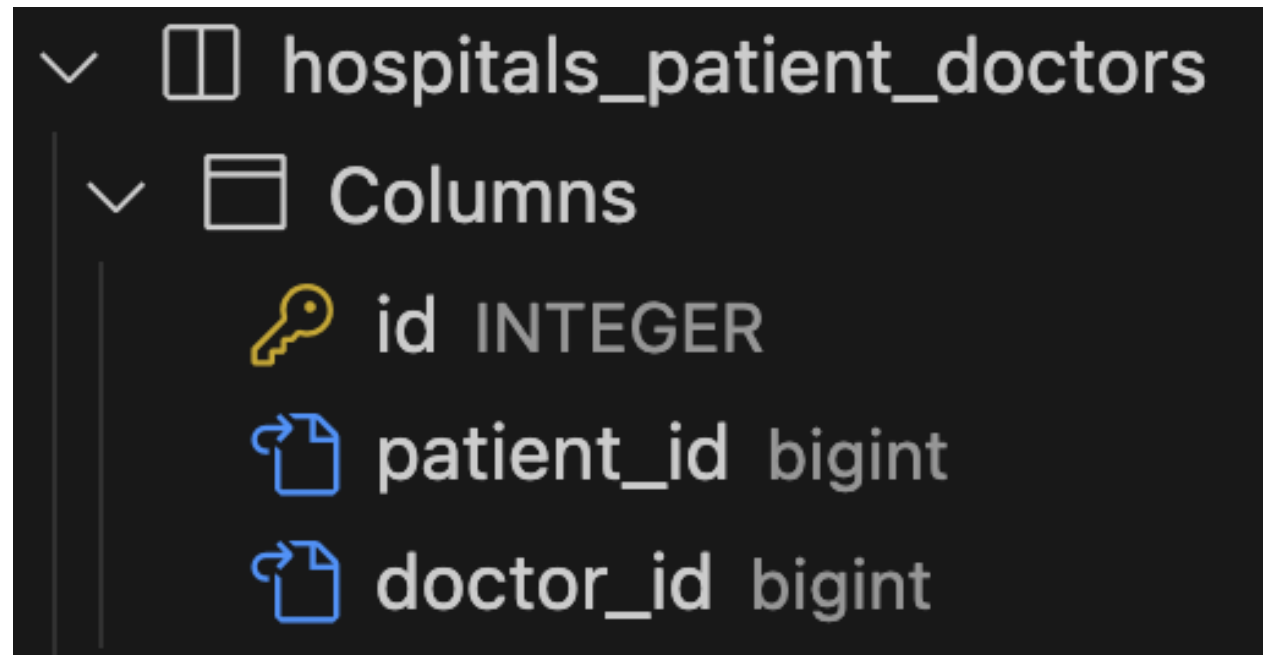
    def __str__(self):
        return f'{self.pk}번 환자 {self.name}'

# Reservation Class 주식 처리
```

## Django ManyToManyField (2/7)

---

- 데이터베이스 초기화 후 Migration 진행 및 shell\_plus 실행
- 생성된 중개 테이블 hospitals\_patient\_doctors 확인



## Django ManyToManyField (3/7)

---

- 의사 1명과 환자 2명 생성

```
doctor1 = Doctor.objects.create(name='allie')
patient1 = Patient.objects.create(name='carol')
patient2 = Patient.objects.create(name='duke')
```



## Django ManyToManyField (4/7)

---

- 예약 생성 (환자가 예약)

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

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

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

## Django ManyToManyField (5/7)

---

- 예약 생성 (의사가 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 duke>]>

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

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

## Django ManyToManyField (6/7)

---

- 중개 테이블에서 예약 현황 확인

id	patient_id	doctor_id
1	1	1
2	2	1

## Django ManyToManyField (7/7)

---

- 예약 취소하기 (삭제)
- 이전에는 Reservation을 찾아서 지워야 했다면, 이제는 `.remove()` 로 삭제 가능

```
# doctor1이 patient1 진료 예약 취소

doctor1.patient_set.remove(patient1)

doctor1.patient_set.all()
<QuerySet [<Patient: 2번 환자 duke>]>

patient1.doctors.all()
<QuerySet []>
```

```
# patient2가 doctor1 진료 예약 취소

patient2.doctors.remove(doctor1)

patient2.doctors.all()
<QuerySet []>

doctor1.patient_set.all()
<QuerySet []>
```

**만약 예약 정보에  
병의 증상, 예약일 등  
추가 정보가 포함되어야 한다면?**

**‘through’ argument**

# ‘through’ argument

---

중개 테이블에 ‘추가 데이터’를 사용해  
M:N 관계를 형성하려는 경우에 사용

## 'through' argument (1/6)

---

- Reservation Class 재작성 및 through 설정
  - 이제는 예약 정보에 “증상”과 “예약일”이라는 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 (2/6)

---

- 데이터베이스 초기화 후 Migration 진행 및 shell\_plus 실행
- 의사 1명과 환자 2명 생성

```
doctor1 = Doctor.objects.create(name='allie')  
patient1 = Patient.objects.create(name='carol')  
patient2 = Patient.objects.create(name='duke')
```

## ‘through’ argument (3/6)

---

- 예약 생성 방법 - 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 allie>]>
```

## ‘through’ argument (4/6)

---

- 예약 생성 방법 - 2
  - Patient 또는 Doctor의 인스턴스를 통한 예약 생성 (**through\_defaults**)

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

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

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

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

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

## 'through' argument (5/6)

---

- 생성된 예약 확인

* id INTEGER	* symptom TEXT	* reserved_at datetime	* doctor_id bigint	* patient_id bigint
Filter	Filter	Filter	Filter	Filter
1	headache		1	1
2	flu		1	2

## 'through' argument (6/6)

---

- 생성과 마찬가지로 의사와 환자 모두 각각 예약 삭제 가능

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

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

## M:N 관계 주요 사항

---

- M:N 관계로 맺어진 두 테이블에는 물리적인 변화가 없음
- `ManyToManyField`는 중개 테이블을 자동으로 생성
- `ManyToManyField`는 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 종속적인 관계가 아니며  
‘의사에게 진찰받는 환자 & 환자를 진찰하는 의사’ 이렇게  
2가지 형태 모두 표현 가능

# 이어서..

삼성 청년 SW 아카데미

# ManyToManyField

---



# ManyToManyField(to, \*\*options)

---

M:N 관계 설정 시 사용하는 모델 필드

## ManyToManyField 특징

---

- 양방향 관계
  - 어느 모델에서든 관련 객체에 접근할 수 있음
- 중복 방지
  - 동일한 관계는 한 번만 저장됨

# ManyToManyField의 대표 인자 3가지

---

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

# 1. 'related\_name' arguments

---

- 역참조시 사용하는 manager name을 변경

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

```
# 변경 전
```

```
doctor.patient_set.all()
```

```
# 변경 후 (변경 후 이전 manager name은 사용 불가)
```

```
doctor.patients.all()
```

## 2. 'symmetrical' arguments (1/2)

---

- 관계 설정 시 대칭 유무 설정
- ManyToManyField가 동일한 모델을 가리키는 정의에서만 사용
- 기본 값 : True

# 예시

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

## 2. 'symmetrical' arguments (2/2)

---

- True일 경우

- source 모델의 인스턴스가 target 모델의 인스턴스를 참조하면 자동으로 target 모델 인스턴스도 source 모델 인스턴스를 자동으로 참조하도록 함(대칭)
- 즉, 내가 당신의 친구라면 자동으로 당신도 내 친구가 됨

- False일 경우

- True와 반대 (대칭되지 않음)

- source 모델
  - 관계를 시작하는 모델
- target 모델
  - 관계의 대상이 되는 모델

### 3. 'through' arguments

---

- 사용하고자 하는 중개모델을 지정
- 일반적으로 “추가 데이터를 M:N 관계와 연결하려는 경우”에 활용

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

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)
```

# M:N에서의 대표 조작 methods

---

- `add()`
  - 관계 추가
  - “지정된 객체를 관련 객체 집합에 추가”
- `remove()`
  - 관계 제거
  - "관련 객체 집합에서 지정된 모델 객체를 제거"



# 이어서..

삼성 청년 SW 아카데미

# 좋아요 기능 구현

---

# 모델 관계 설정

# Many to many relationships

---

한 테이블의 0개 이상의 레코드가  
다른 테이블의 0개 이상의 레코드와 관련된 경우  
❖ 양쪽 모두에서 N:1 관계를 가짐

# Article(M) - User(N)

---

0개 이상의 게시글은 0명 이상의 회원과 관련

# Article(M) - User(N)

---

0개 이상의 게시글은 0명 이상의 회원과 관련

- 게시글은 회원으로부터 0개 이상의 좋아요를 받을 수 있고,  
회원은 0개 이상의 게시글에 좋아요를 누를 수 있음

## 모델 관계 설정 (1/4)

---

Article 클래스에 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 (2/4)

---

Migration 진행 후 에러 발생

ERRORS:

```
articles.Article.like_users: (fields.E304) Reverse accessor 'User.article_set' for 'articles.Article.like_users' clashes with reverse accessor for 'articles.Article.user'.  
    HINT: Add or change a related_name argument to the definition for 'articles.Article.like_users' or 'articles.Article.user'.  
articles.Article.user: (fields.E304) Reverse accessor 'User.article_set' for 'articles.Article.user' clashes with reverse accessor for 'articles.Article.like_users'.  
    HINT: Add or change a related_name argument to the definition for 'articles.Article.user' or 'articles.Article.like_users'.
```



## 역참조 매니저 충돌 (1/2)

---

- N:1
  - “유저가 작성한 게시글”
  - `user.article_set.all()`
- M:N
  - “유저가 좋아요 한 게시글”
  - `user.article_set.all()`

## 역참조 매니저 충돌 (2/2)

---

- like\_users 필드 생성 시 자동으로 역참조 매니저 `.article_set`가 생성됨
- 그러나 이전 N:1(Article-User) 관계에서 이미 같은 이름의 매니저를 사용 중
  - `user.article_set.all()` → 해당 유저가 작성한 모든 게시물 조회
- ‘user가 작성한 글 (`user.article_set`)’과  
‘user가 좋아요를 누른 글(`user.article_set`)’을 구분할 수 없게 됨
- user와 관계된 ForeignKey 혹은 ManyToManyField 둘 중 하나에 `related_name` 작성 필요

## 모델 관계 설정 (3/4)

---

related\_name 작성 후 Migration 재진행

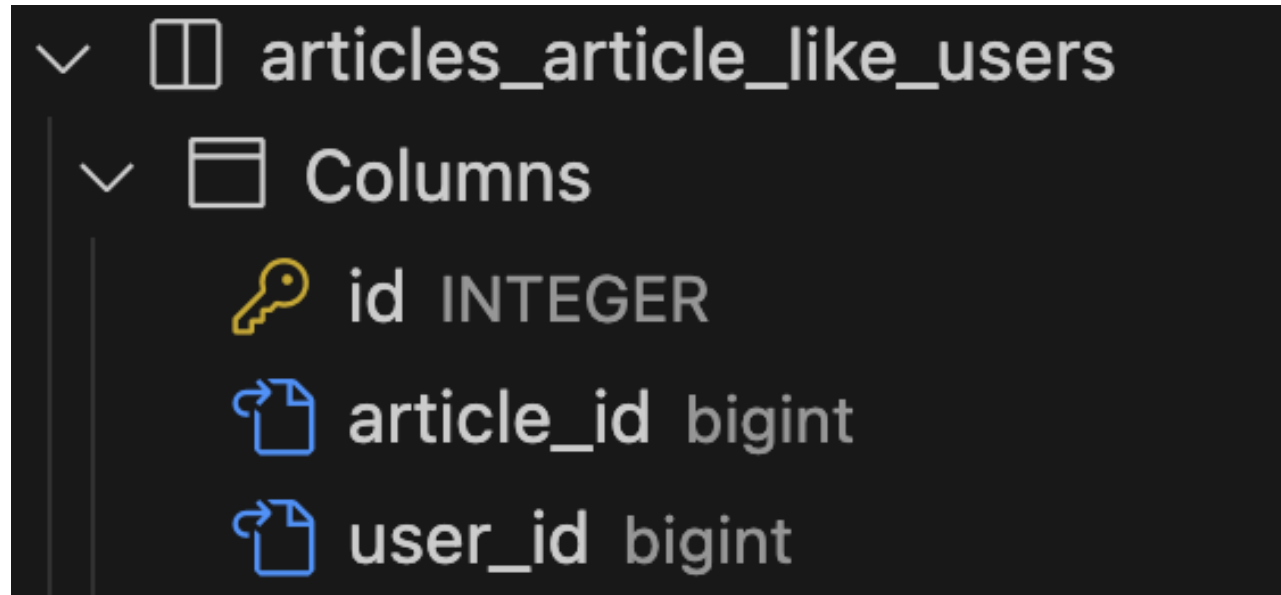
```
# articles/models.py

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)
```

## 모델 관계 설정 (4/4)

---

생성된 중개 테이블 확인



The screenshot shows a database schema viewer with a dark background. It displays a table named 'articles\_article\_like\_users'. Under the 'Columns' section, three columns are listed: 'id' (INTEGER, primary key), 'article\_id' (bigint, foreign key), and 'user\_id' (bigint, foreign key). Each foreign key is indicated by a blue document icon with an arrow.

✓	articles_article_like_users
✓	Columns
✓	id INTEGER
✓	article_id bigint
✓	user_id bigint

# User - Article간 사용 가능한 전체 related manager

---

- `article.user`
  - 게시글을 작성한 유저 - N:1
- `user.article_set`
  - 유저가 작성한 게시글(역참조) - N:1
- `article.like_users`
  - 게시글을 좋아요 한 유저 - M:N
- `user.like_articles`
  - 유저가 좋아요 한 게시글(역참조) - M:N

# 기능 구현

## 기능 구현 (1/5)

---

url 작성

```
# articles/urls.py

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

## 기능 구현 (2/5)

---

view 함수 작성

```
# articles/views.py

@login_required
def likes(request, article_pk):
    article = Article.objects.get(pk=article_pk)
    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')
```



## 기능 구현 (3/5)

index 템플릿에서 각 게시 글에 좋아요 버튼 출력

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

{% for article in articles %}
...
<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>
<hr>
{% endfor %}
```

## 기능 구현 (4/5)

좋아요 버튼 출력 확인

### Articles

Hello, admin

[NEW](#)

[Logout](#)

[회원탈퇴](#)

[회원정보 수정](#)

---

작성자 : admin

글 번호: 1

[글 제목: title](#)

글 내용: content

[좋아요](#)

---

작성자 : admin

글 번호: 2



[글 제목: 제목](#)

글 내용: 내용

[좋아요](#)

## 기능 구현 (5/5)

좋아요 버튼 클릭 후 테이블 확인

* id INTEGER	*  article_id bigint	*  user_id bigint
Filter	Filter	Filter
1	1	1

다음 시간에  
만나요!

삼성 청년 SW 아카데미

# DB 06

## Many to many relationships 02

# • 오프라인 INDEX

- 팔로우 기능 구현
  - 프로필 페이지
  - 모델 관계 설정
  - 기능 구현
- Fixtures
  - dumpdata
  - loaddata

# • 오프라인 INDEX

- Improve query
  - 사전 준비
  - annotate
  - select\_related
  - prefetch\_related
  - select\_related & prefetch\_related
- 참고
  - 'exists' method
  - 한꺼번에 dump 하기
  - loaddata 인코딩 에러

# 팔로우 기능 구현

---



# 프로필 페이지

## 프로필 페이지

---

- 각 회원의 개인 프로필 페이지에 팔로우 기능을 구현하기 위해  
프로필 페이지를 먼저 구현하기

## 프로필 구현 (1/5)

---

url 작성

```
# accounts/urls.py

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

## 프로필 구현 (2/5)

---

view 함수 작성

```
# 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)
```

## 프로필 구현 (3/5)

---

profile 템플릿 작성

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

<h1>{{ person.username }}님의 프로필</h1>

<hr>

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

<hr>
...
```

```
...

<h2>{{ person.username }} 가 작성한 댓글</h2>
{% for comment in person.comment_set.all %}
  <div>{{ comment.content }}</div>
{% endfor %}

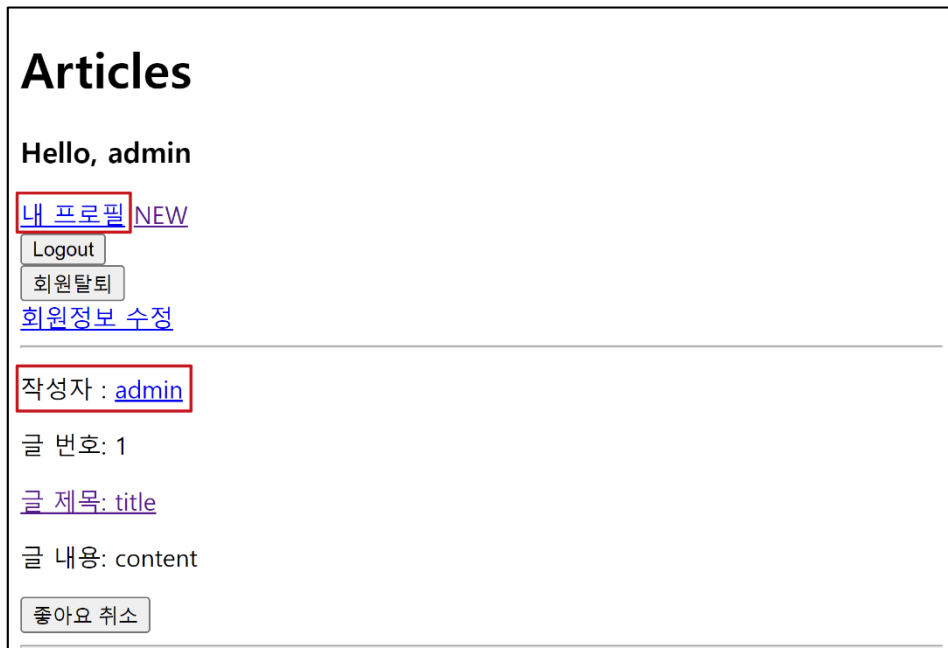
<hr>

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

## 프로필 구현 (4/5)

프로필 페이지로 이동할 수 있는 링크 작성

```
<!-- articles/index.html -->  
<a href="{% url 'accounts:profile' user.username %}">내 프로필</a>  
<p>작성자 : <a href="{% url 'accounts:profile' article.user.username %}">{{ article.user }}</a></p>
```



## 프로필 구현 (5/5)

프로필 페이지 결과 확인

### admin님의 프로필

#### admin가 작성한 게시글

title  
제목  
□  
제목

#### admin가 작성한 댓글

first comment  
second comment  
댓글 고고

#### admin가 좋아요한 게시글

title

# 모델 관계 설정



# User(M) - User(N)

---

0명 이상의 회원은 0명 이상의 회원과 관련

# User(M) - User(N)

---

0명 이상의 회원은 0명 이상의 회원과 관련

- 회원은 0명 이상의 팔로워를 가질 수 있고,  
0명 이상의 다른 회원들을 팔로잉 할 수 있음

## 모델 관계 설정 (1/2)

---

- ManyToManyField 작성

```
# accounts/models.py

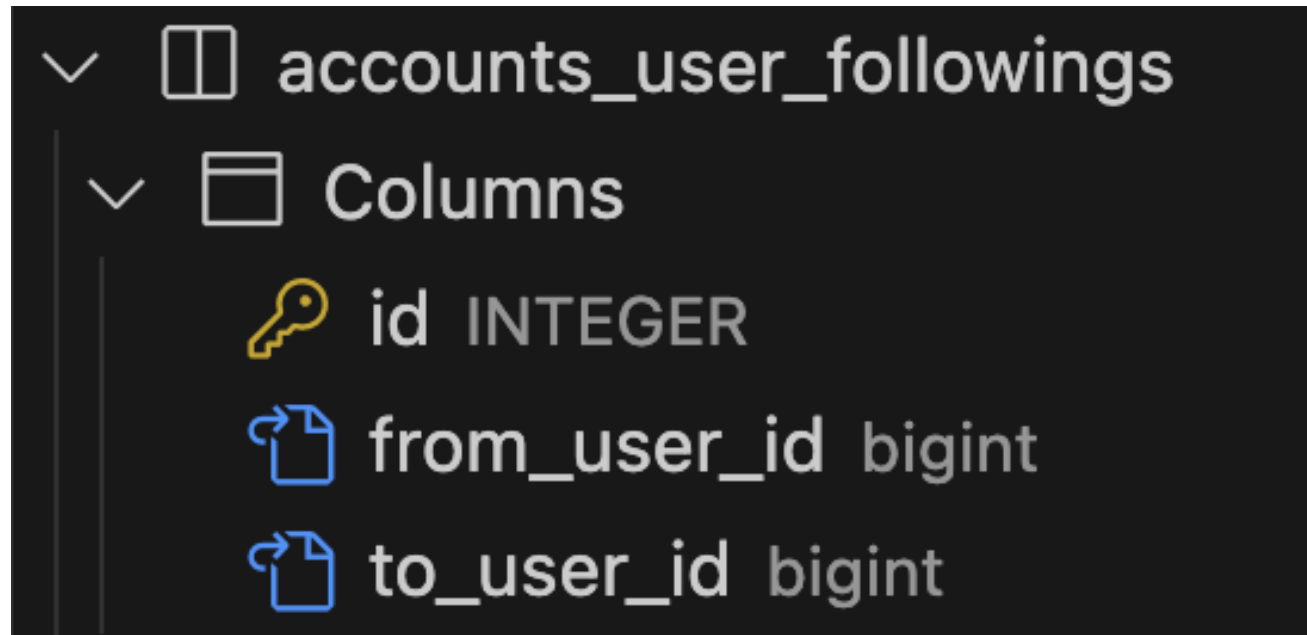
class User(AbstractUser):
    followings = models.ManyToManyField('self', symmetrical=False, related_name='followers')
```

- 참조
    - 내가 팔로우하는 사람들 (팔로잉, followings)
  - 역참조
    - 상대방 입장에서 나는 팔로워 중 한 명 (팔로워, followers)
- ❖ 바뀌어도 상관 없으나 관계 조회 시 생각하기 편한 방향으로 정한 것

## 모델 관계 설정 (2/2)

---

Migrations 진행 후 중개 테이블 확인



# 기능 구현

# 기능 구현 (1/4)

---

url 작성

```
# accounts/urls.py

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

## 기능 구현 (2/4)

---

view 함수 작성

```
# accounts/views.py

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

## 기능 구현 (3/4)

프로필 유저의 팔로잉, 팔로워 수 & 팔로우, 언팔로우 버튼 작성

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

<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>
```



## 기능 구현 (4/4)

팔로우 버튼 클릭 → 팔로우 버튼 변화 및 중개 테이블 데이터 확인

### admin님의 프로필

팔로잉 : 0 / 팔로워 : 1

Unfollow

* id INTEGER	* from_user_id bigint	* to_user_id bigint
Filter	Filter	Filter
1	2	1

# 이어서..

삼성 청년 SW 아카데미

# Fixtures

---

# 개요

# Fixtures

---

Django 개발 시 데이터 베이스 초기화 및 공유를 위해  
사용되는 파일 형식

# Fixtures 사용 목적

---

- 샘플, 초기 데이터 세팅
- 협업 시 동일한 데이터 환경 맞추기

# 초기 데이터의 필요성

---

- 협업하는 유저 A, B가 있다고 생각해보기
    1. A가 먼저 프로젝트를 작업 후 원격 저장소에 push 진행
      - gitignore로 인해 DB는 업로드하지 않기 때문에 A가 생성한 데이터도 업로드 X
    2. B가 원격 저장소에서 A가 push한 프로젝트를 pull (혹은 clone)
      - 결과적으로 B는 DB가 없는 프로젝트를 받게 됨
  - 이처럼 프로젝트의 앱을 처음 설정할 때 동일하게 준비 된 데이터로 데이터베이스를 미리 채우는 것이 필요한 순간이 있음
- Django에서는 fixtures를 사용해 앱에 초기 데이터(initial data)를 제공

# fixtures 관련 명령어

---

**dumpdata**

생성 (데이터 추출)

**loaddata**

로드 (데이터 입력)



# 사전준비

---

- M:N 까지 모두 작성된 Django 프로젝트에서  
유저, 게시글, 댓글 등 각 데이터를 최소 2~3개 이상 생성해두기

**dumpdata**

# dumpdata

---

데이터베이스의 특정 모델 혹은 앱 전체 데이터를 추출

# dumpdata 기본 명령어

---

```
$ python manage.py dumpdata [앱이름.모델이름] [옵션] > 추출파일명.json
```

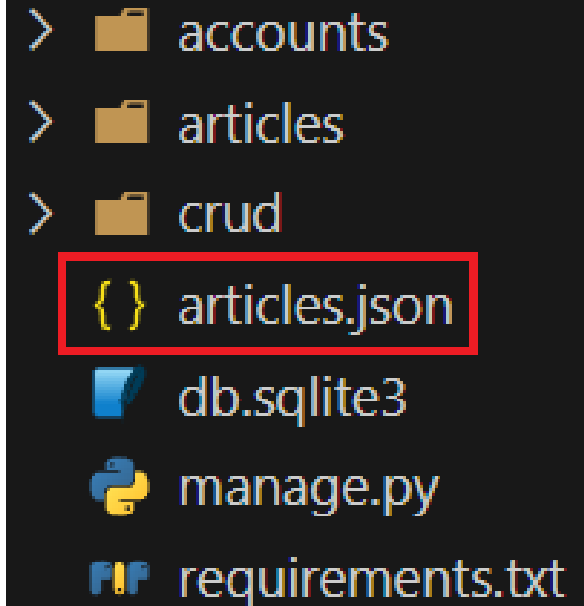
- **앱이름.모델이름** 지정
  - 특정 모델의 데이터를 추출
- **앱이름만** 지정
  - 해당 앱의 모든 모델에 대한 데이터를 추출
- **앱 혹은 모델명을** 지정하지 않은 경우
  - 프로젝트 전체의 모델 데이터를 추출
- **--format** 옵션을 통해 **JSON, YAML** 등의 형식 지정 가능(기본값: JSON)

## dumpdata 명령어 예시 (1/2)

---

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

- `articles` 앱의 `Article` 모델 데이터를 추출
- 명령어 실행 후 프로젝트 폴더에 `articles.json` 파일이 생성됨
- `articles.json` 파일에는 `Article` 모델의 모든 데이터가 JSON 형식으로 작성되어 있음
- Fixtures 파일명은 자유롭게 작성 가능



```
> accounts
> articles
> crud
{ } articles.json
db.sqlite3
manage.py
requirements.txt
```

## dumpdata 명령어 예시 (2/2)

---

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

```
> accounts  
> articles  
> crud  
{} articles.json  
{} comments.json  
db.sqlite3  
manage.py  
requirements.txt  
{} users.json
```

# Fixtures 파일을 직접 만들지 말 것

반드시 `dumpdata` 명령어를 사용하여 생성

## dumpdata 정리

---

- **dumpdata** 명령어를 사용하면 프로젝트 내 특정 앱 혹은 모델에 대한 데이터를 JSON 등 원하는 포맷으로 추출 가능
- 이렇게 생성된 데이터 파일은 추후 다른 환경에서 **loaddata**로 불러와 동일한 데이터 상태를 재현할 수 있으며, 협업 및 배포에 큰 장점이 있음



**loaddata**

# loaddata

---

dumpdata를 통해 추출한 데이터 파일을  
다시 데이터베이스에 반영

# loaddata 기본 명령어

---

```
$ python manage.py loaddata 파일경로
```

- Fixtures 파일의 기본 경로에 있는 파일을 DB에 반영
- Fixtures 파일의 기본 경로
  - **app\_name/fixtures/**
- Django는 설치된 모든 app의 디렉토리에서 fixtures 폴더 이후의 경로로 fixtures 파일을 찾아 load를 진행

## 사전준비

---

```
✓ articles
  ✓ fixtures
    {} articles.json
    {} comments.json
    {} users.json
```

- dumpdata로 생성한 파일들을 해당 위치로 이동
- db.sqlite3 파일 삭제 후 migrate 진행

## loaddata 명령어 예시 (1/2)

---

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

- dumpdata로 생성한 파일들을 모두 DB에 반영
- 파일은 작성 순서에 상관 없음

## loaddata 명령어 예시 (2/2)

---

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

- 단, `loaddata`를 한번에 실행하지 않고 별도로 실행한다면 모델 관계에 따라 load 순서가 중요할 수 있음
  - `comment`는 `article`에 대한 key 및 `user`에 대한 key가 필요
  - `article`은 `user`에 대한 key가 필요
- 즉, 현재 모델 관계에서는 `user` → `article` → `comment` 순으로 data를 load 해야 오류가 발생하지 않음

## loaddata 주의사항

---

- **loaddata**를 실행하기 전에 해당 모델에 대한 마이그레이션이 완료되어 있어야 함
- 같은 PK를 가진 데이터가 이미 있는 경우 중복 에러가 발생할 수 있음
  - 이 경우 기존 데이터를 지우거나, 새로운 Fixture 파일을 사용해야 함

## loaddata 정리

---

- `loaddata` 명령어는 `dumpdata`로 추출한 Fixture 파일을 DB로 불러오는 명령어이며, 개발 환경 준비나 협업 시 매우 유용
- 마이그레이션 상태를 먼저 확인하고, 인코딩 문제 등을 사전에 해결하면 매끄럽게 데이터를 복원할 수 있음



# 이어서..

삼성 청년 SW 아카데미

# Improve query

---

# 사전 준비

# Improve query

---

“query 개선하기”

- 같은 결과를 얻기 위해 DB 측에 보내는 query 개수를 점차 줄여 조회하기

## 사전 준비 (1/2)

---

- fixtures 데이터
  - 게시글 10개 / 댓글 100개 / 유저 5개
- 모델 관계
  - N:1 - Article:User / Comment:Article / Comment:Article
  - N:M - Article:User

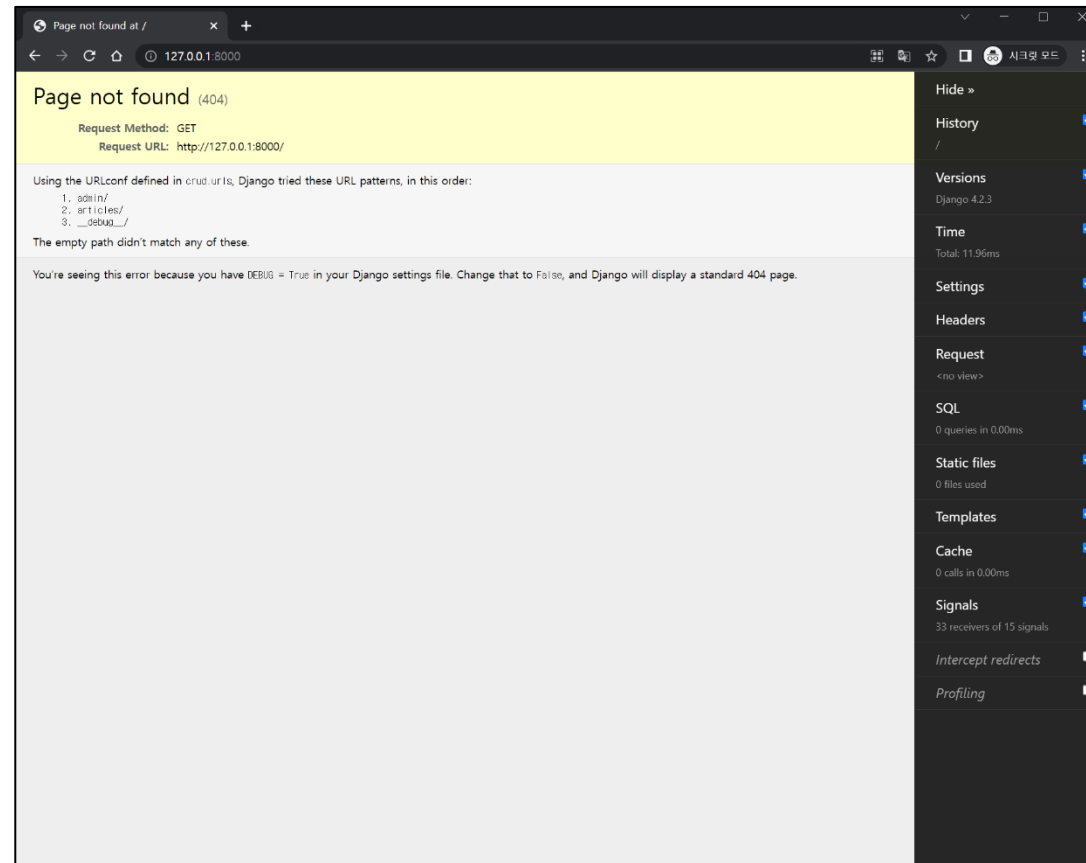
```
$ python manage.py migrate
$ python manage.py loaddata users.json articles.json comments.json

Installed 115 object(s) from 3 fixture(s)
```

## 사전 준비 (2/2)

---

- 서버 실행 후 확인



**annotate**

# annotate

---

- SQL의 GROUP BY를 사용
- 쿼리셋의 각 객체에 계산된 필드를 추가
- 집계 함수(Count, Sum 등)와 함께 자주 사용됨



## annotate 예시

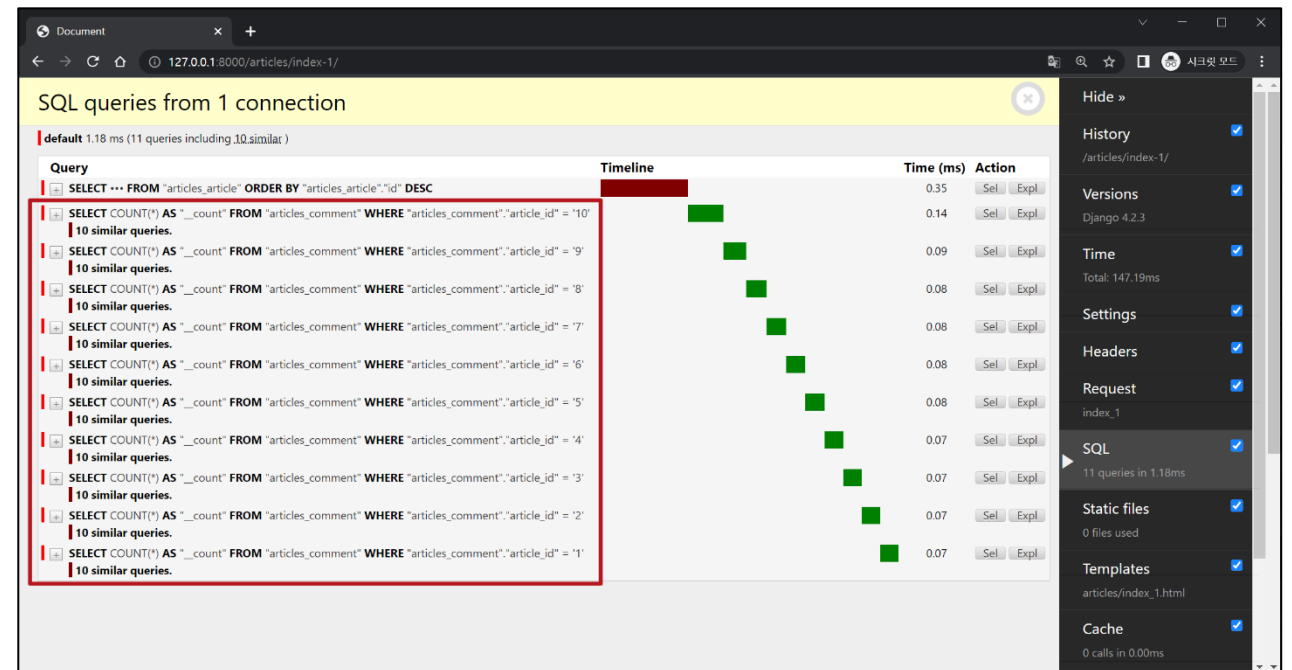
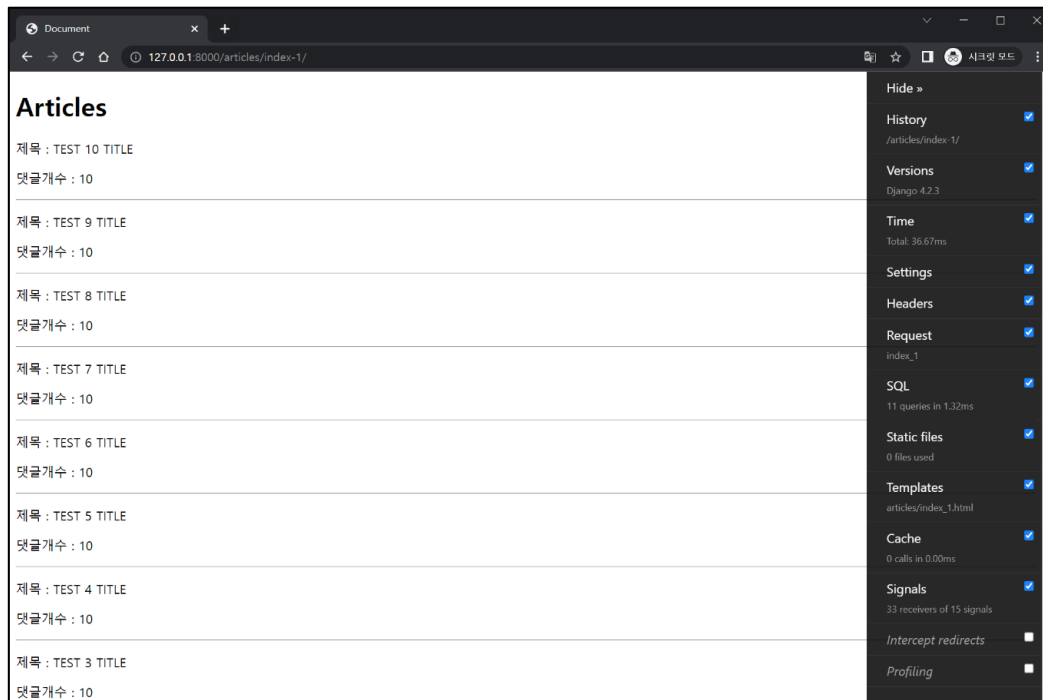
---

```
Book.objects.annotate(num_authors=Count('authors'))
```

- 의미
  - 결과 객체에 'num\_authors'라는 새로운 필드를 추가
  - 이 필드는 각 책과 연관된 저자의 수를 계산
- 결과
  - 결과에는 기존 필드와 함께 'num\_authors' 필드를 가지게 됨
  - `book.num_authors`로 해당 책의 저자 수에 접근할 수 있게 됨

## 문제 상황 (1/2)

- <http://127.0.0.1:8000/articles/index-1/>
  - “11 queries including 10 similar”



## 문제 상황 (2/2)

---

- 문제 원인
  - 각 게시글마다 댓글 개수를 반복 평가

```
<!-- index_1.html -->
```

```
<p>댓글개수 : {{ article.comment_set.count }}</p>
```

## annotate 적용 (1/2)

---

- 문제 해결
  - 게시글을 조회하면서 댓글 개수까지 한번에 조회해서 가져오기

```
# views.py

def index_1(request):
    # articles = Article.objects.order_by('-pk')
    articles = Article.objects.annotate(Count('comment')).order_by('-pk')
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_1.html', context)
```

```
<!-- index_1.html -->
```

```
<p>댓글개수 : {{ article.comment__count }}</p>
```

## annotate 적용 (2/2)

- 문제 해결
  - “11 queries including 10 similar” → “1 query”

The screenshot shows the Django Debug Toolbar interface. The top bar indicates 'SQL queries from 1 connection'. Below this, a table lists the queries. The first query is highlighted, showing its SQL text and execution time of 0.40 ms. The SQL text is as follows:

```
SELECT "articles_article"."id",  
       "articles_article"."user_id",  
       "articles_article"."title",  
       "articles_article"."content",  
       "articles_article"."created_at",  
       "articles_article"."updated_at",  
       COUNT("articles_comment"."id") AS  
       "comment__count"  
FROM "articles_article"  
LEFT OUTER JOIN "articles_comment"  
ON ("articles_article"."id" =  
    "articles_comment"."article_id")  
GROUP BY "articles_article"."id",  
         "articles_article"."user_id",  
         "articles_article"."title",  
         "articles_article"."content",  
         "articles_article"."created_at",  
         "articles_article"."updated_at"  
ORDER BY "articles_article"."id" DESC
```

The 'GROUP BY' clause is highlighted with a red box. The right sidebar shows various toolbar options like History, Versions, Time, Settings, Headers, Request, SQL, and Static files, all of which are checked.

**select\_related**

# select\_related

---

- SQL의 **INNER JOIN**를 사용
- 1:1 또는 N:1 참조 관계에서 사용
  - ForeignKey나 OneToOneField 관계에 대해 JOIN을 수행
- 단일 쿼리로 관련 객체를 함께 가져와 성능을 향상

## select\_related 예시

---

```
Book.objects.select_related('publisher')
```

- 의미
  - Book 모델과 연관된 Publisher 모델의 데이터를 함께 가져옴
  - ForeignKey 관계인 'publisher'를 JOIN하여 단일 쿼리만으로 데이터를 조회
- 결과
  - Book 객체를 조회할 때 연관된 Publisher 정보도 함께 로드
  - `book.publisher.name`과 같은 접근이 추가적인 데이터베이스 쿼리 없이 가능



## 문제 상황 (1/2)

- <http://127.0.0.1:8000/articles/index-2/>
  - “11 queries including 10 similar and 8 duplicates”

Articles

작성자 : murphysusan  
제목 : TEST 10 TITLE

작성자 : marshallpatricia  
제목 : TEST 9 TITLE

작성자 : marshallpatricia  
제목 : TEST 8 TITLE

작성자 : murphysusan  
제목 : TEST 7 TITLE

작성자 : margaret13  
제목 : TEST 6 TITLE

작성자 : marshallpatricia  
제목 : TEST 5 TITLE

작성자 : margaret13  
제목 : TEST 4 TITLE

Hide »

History  
/articles/index-2/

Versions  
Django 4.2.3

Time  
Total: 32.74ms

Settings

Headers

Request  
index\_2

SQL  
11 queries in 1.17ms

Static files  
0 files used

Templates  
articles/index\_2.html

Cache  
0 calls in 0.00ms

Signals  
33 receivers of 15 signals

Intercept redirects

Profiling

SQL queries from 1 connection

default 1.14 ms (11 queries including 10 similar and 8 duplicates)

Query	Timeline	Time (ms)	Action
SELECT ... FROM "articles_article" ORDER BY "articles_article"."id" DESC		0.30	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '5' LIMIT 21 10 similar queries. Duplicated 3 times.		0.11	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '1' LIMIT 21 10 similar queries. Duplicated 3 times.		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '1' LIMIT 21 10 similar queries. Duplicated 3 times.		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '5' LIMIT 21 10 similar queries. Duplicated 3 times.		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '2' LIMIT 21 10 similar queries. Duplicated 2 times.		0.09	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '1' LIMIT 21 10 similar queries. Duplicated 3 times.		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '2' LIMIT 21 10 similar queries. Duplicated 2 times.		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '4' LIMIT 21 10 similar queries.		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '5' LIMIT 21 10 similar queries. Duplicated 3 times.		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '3' LIMIT 21 10 similar queries.		0.08	Sel Expl

History  
/articles/index-2/

Versions  
Django 4.2.3

Time  
Total: 18.31ms

Settings

Headers

Request  
index\_2

SQL  
11 queries in 1.14ms

Static files  
0 files used

Templates  
articles/index\_2.html

Cache  
0 calls in 0.00ms

Signals  
33 receivers of 15 signals

## 문제 상황 (2/2)

---

- 문제 원인
  - 각 게시글마다 작성한 유저명까지 반복 평가

```
<!-- index_2.html -->

{% for article in articles %}
  <h3>작성자 : {{ article.user.username }}</h3>
  <p>제목 : {{ article.title }}</p>
  <hr>
{% endfor %}
```

## select\_related 적용 (1/2)

---

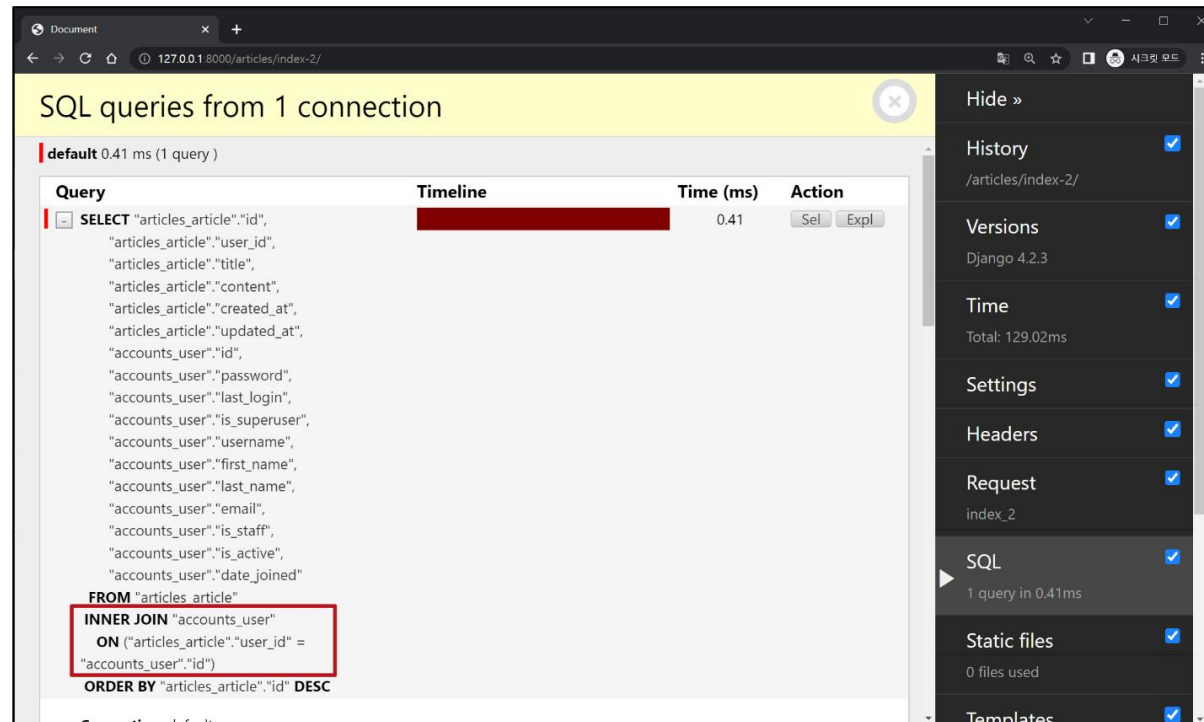
- 문제 해결
  - 게시글을 조회하면서 **유저 정보까지 한번에 조회**해서 가져오기

```
# views.py

def index_2(request):
    # articles = Article.objects.order_by('-pk ' )
    articles = Article.objects.select_related('user').order_by('-pk ' )
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_2.html', context)
```

## select\_related 적용 (2/2)

- 문제 해결
  - “11 queries including 10 similar and 8 duplicates” → “1 query”



**prefetch\_related**

# prefetch\_related

---

- SQL이 아닌 Python을 사용한 JOIN을 진행
  - 관련 객체들을 미리 가져와 메모리에 저장하여 성능을 향상
- M:N 또는 N:1 역참조 관계에서 사용
  - ManyToManyField나 역참조 관계에 대해 별도의 쿼리를 실행

# prefetch\_related 예시

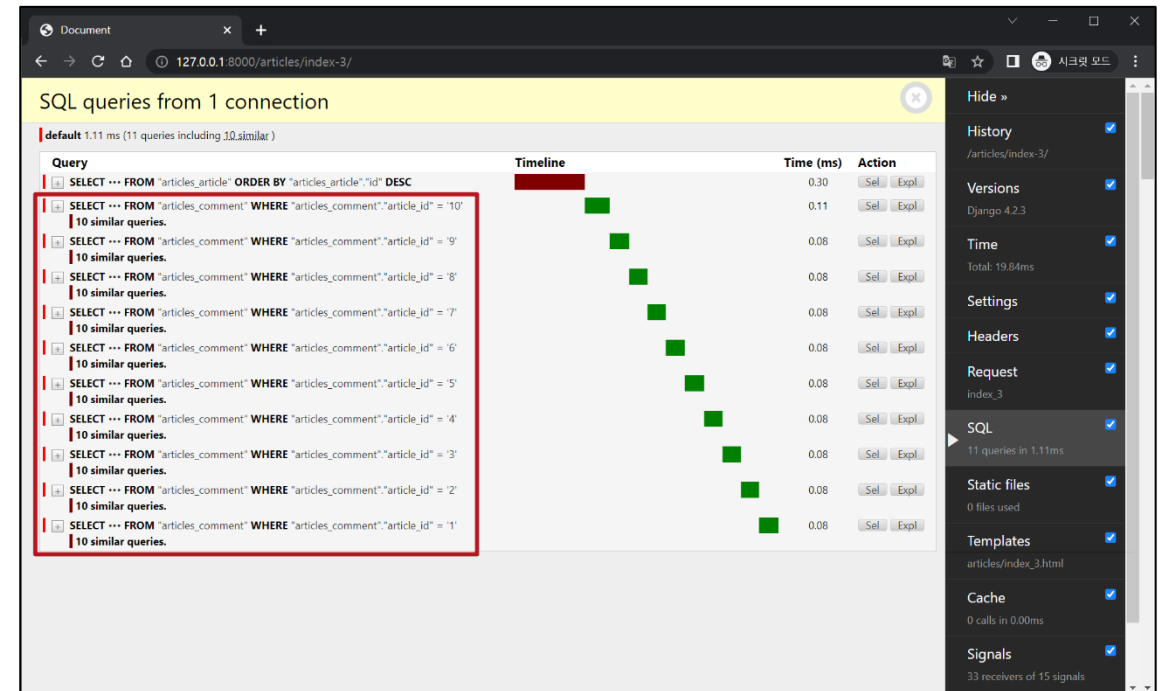
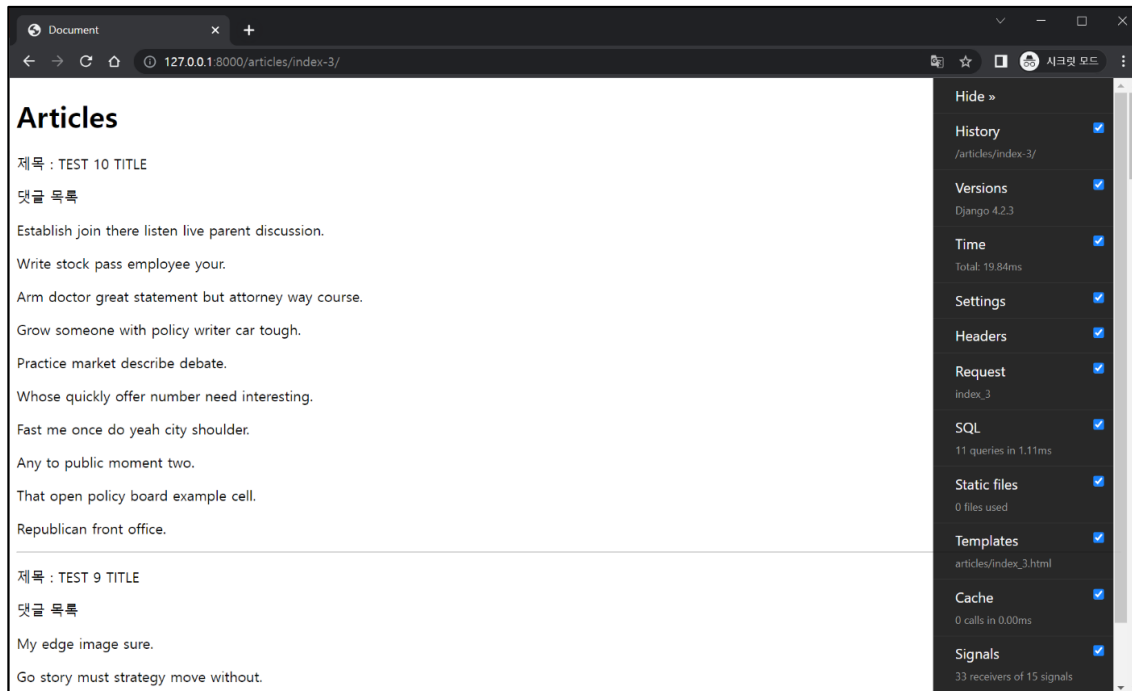
---

```
Book.objects.prefetch_related('authors')
```

- 의미
  - Book과 Author는 ManyToMany 관계로 가정
  - Book 모델과 연관된 모든 Author 모델의 데이터를 미리 가져옴
  - Django가 별도의 쿼리로 Author 데이터를 가져와 관계를 설정
- 결과
  - Book 객체들을 조회한 후, 연관된 모든 Author 정보가 미리 로드 됨
  - `for author in book.authors.all()`와 같은 반복이 추가적인 데이터베이스 쿼리 없이 실행됨

# 문제 상황 (1/2)

- <http://127.0.0.1:8000/articles/index-3/>
  - “11 queries including 10 similar”





## 문제 상황 (2/2)

---

- 문제 원인
  - 각 게시글 출력 후 각 게시글의 댓글 목록까지 개별적으로 모두 평가

```
<!-- index_3.html -->

{% for article in articles %}
  <p>제목 : {{ article.title }}</p>
  <p>댓글 목록</p>
  {% for comment in article.comment_set.all %}
    <p>{{ comment.content }}</p>
  {% endfor %}
  <hr>
{% endfor %}
```

## prefetch\_related 적용 (1/2)

---

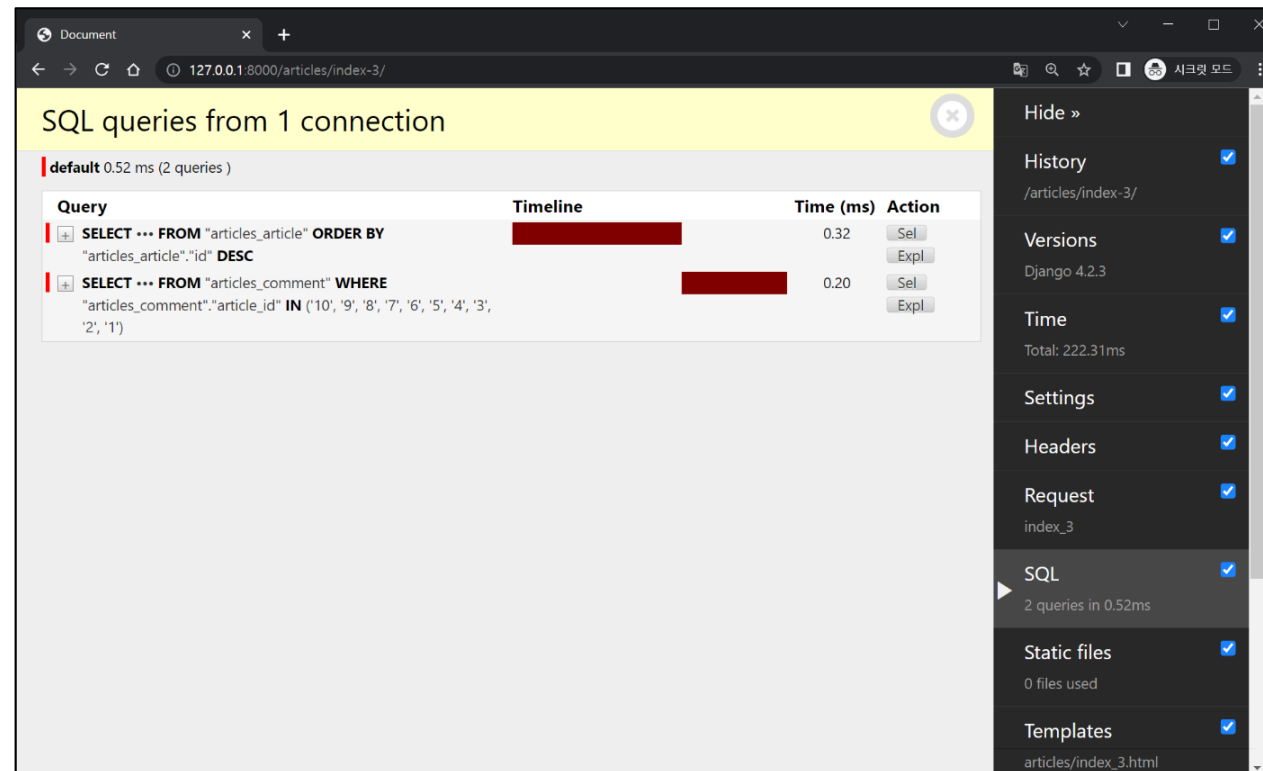
- 문제 해결
  - 게시글을 조회하면서 참조된 댓글까지 한번에 조회해서 가져오기

```
# views.py

def index_3(request):
    # articles = Article.objects.order_by('-pk')
    articles = Article.objects.prefetch_related('comment_set').order_by('-pk')
    context = {
        'articles': articles,
    }
    return render(request, 'articles/index_3.html', context)
```

# prefetch\_related 적용 (2/2)

- 문제 해결
  - “11 queries including 10 similar” → “2 queries”



**select\_related  
&  
prefetch\_related**

## 문제 상황 (1/2)

- <http://127.0.0.1:8000/articles/index-4/>
  - “111 queries including 110 similar and 100 duplicates”

**Articles**

제목 : TEST 10 TITLE

댓글 목록

marshallpatricia : Establish join there listen live parent discussion.

jimmybean : Write stock pass employee your.

marshallpatricia : Arm doctor great statement but attorney way course.

marshallpatricia : Grow someone with policy writer car tough.

margaret13 : Practice market describe debate.

marshallpatricia : Whose quickly offer number need interesting.

jimmybean : Fast me once do yeah city shoulder.

murphysusan : Any to public moment two.

margaret13 : That open policy board example cell.

marshallpatricia : Republican front office.

제목 : TEST 9 TITLE

댓글 목록

murphysusan : My edge image sure.

murphysusan : Go story must strategy move without.

**SQL**  
111 queries in 9.20ms

**SQL queries from 1 connection**

default 9.20 ms (111 queries including 110 similar and 100 duplicates)

Query	Timeline	Time (ms)	Action
SELECT ... FROM "articles_article" ORDER BY "articles_article"."id" DESC		0.32	Sel Expl
SELECT ... FROM "articles_comment" WHERE "articles_comment"."article_id" = '10'		0.11	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '1' LIMIT 21		0.11	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '4' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '1' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '2' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '1' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '4' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '2' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '1' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '5' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '2' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '1' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "articles_comment" WHERE "articles_comment"."article_id" = '9'		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '5' LIMIT 21		0.08	Sel Expl
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = '5' LIMIT 21		0.08	Sel Expl

**SQL**  
111 queries in 9.20ms

## 문제 상황 (2/2)

---

- 문제 원인
  - “게시글” + “각 게시글의 댓글 목록” + “댓글의 작성자”를 단계적으로 평가

```
<!-- index_4.html -->

{% for article in articles %}
  <p>제목 : {{ article.title }}</p>
  <p>댓글 목록</p>
  {% for comment in article.comment_set.all %}
    <p>{{ comment.user.username }} : {{ comment.content }}</p>
  {% endfor %}
</hr>
{% endfor %}
```

## prefetch\_related 적용 (1/2)

---

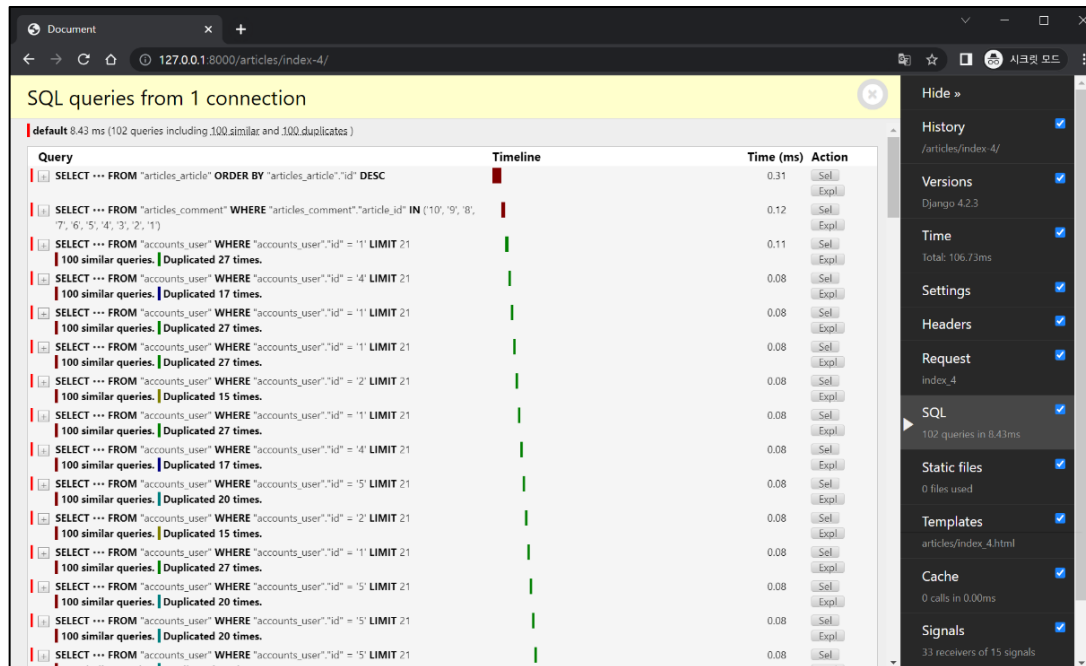
- 문제 해결 1단계
  - 게시글을 조회하면서 참조된 댓글까지 한번에 조회

```
# views.py

def index_4(request):
    # articles = Article.objects.order_by('-pk')
    articles = Article.objects.prefetch_related('comment_set').order_by('-pk')
    # articles = Article.objects.prefetch_related(
    # Prefetch('comment_set', queryset=Comment.objects.select_related('user'))
    # ).order_by('-pk')
```

## prefetch\_related 적용 (2/2)

- 문제 해결 1단계
  - “111 queries including 110 similar and 100 duplicates”
  - ➔ “102 queries including 100 similar and 100 duplicates”



Query	Time (ms)	Action
SELECT ... FROM "articles_article" ORDER BY "articles_article"."id" DESC	0.31	Expl.
SELECT ... FROM "articles_comment" WHERE "articles_comment"."article_id" IN (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)	0.12	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 1 LIMIT 21 100 similar queries. Duplicated 27 times.	0.11	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 4 LIMIT 21 100 similar queries. Duplicated 17 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 1 LIMIT 21 100 similar queries. Duplicated 27 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 1 LIMIT 21 100 similar queries. Duplicated 27 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 2 LIMIT 21 100 similar queries. Duplicated 15 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 1 LIMIT 21 100 similar queries. Duplicated 27 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 4 LIMIT 21 100 similar queries. Duplicated 17 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 5 LIMIT 21 100 similar queries. Duplicated 20 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 2 LIMIT 21 100 similar queries. Duplicated 15 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 1 LIMIT 21 100 similar queries. Duplicated 27 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 5 LIMIT 21 100 similar queries. Duplicated 20 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 5 LIMIT 21 100 similar queries. Duplicated 20 times.	0.08	Expl.
SELECT ... FROM "accounts_user" WHERE "accounts_user"."id" = 5 LIMIT 21 100 similar queries. Duplicated 20 times.	0.08	Expl.

❖ 아직 각 댓글을 조회하면서  
각 댓글의 작성자를 중복 조회 중



# select\_related & prefetch\_related 적용 (1/2)

---

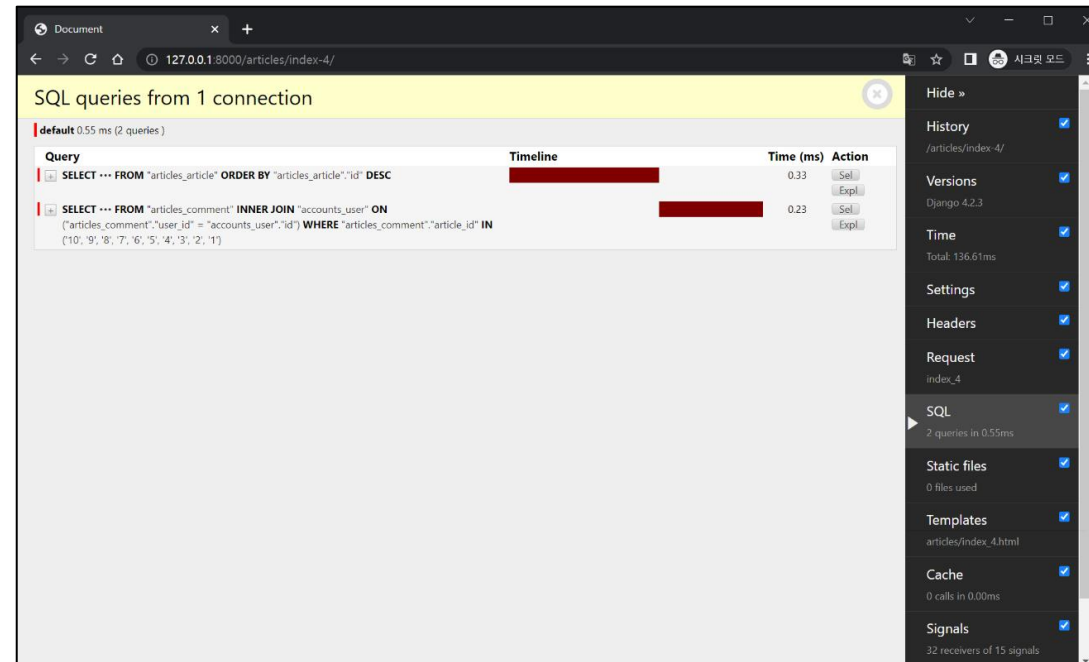
- 문제 해결 2단계
  - “게시글” + “각 게시글의 댓글 목록” + “댓글의 작성자”를 한번에 조회

```
# views.py

def index_4(request):
    # articles = Article.objects.order_by('-pk')
    # articles = Article.objects.prefetch_related('comment_set').order_by('-pk')
    articles = Article.objects.prefetch_related(
        Prefetch('comment_set', queryset=Comment.objects.select_related('user'))
    ).order_by('-pk')
```

# select\_related & prefetch\_related 적용 (2/2)

- 문제 해결 2단계
  - “102 queries including 100 similar and 100 duplicates”  
→ “2 queries”



# 최적화 주의사항

## 선투론 최적화는 악의 근원

---

"작은 효율성에 대해서는,  
말하자면 97% 정도에 대해서는, 잊어버려라.  
선투론 최적화는 모든 악의 근원이다."

- 도널드 커누스(Donald E. Knuth)

- ✓ 튜링상을 수상한 컴퓨터 과학자/수학자
- ✓ The Art of Computer Programming의 저자

# 이어서..

삼성 청년 SW 아카데미

# 참고

---

**‘exists’ method**

## .exists()

---

- QuerySet에 결과가 하나 이상 존재하는지 여부를 확인하는 메서드
- 결과가 포함되어 있으면 True를 반환하고  
결과가 포함되어 있지 않으면 False를 반환



## .exists() 특징

---

- 데이터베이스에 최소한의 쿼리만 실행하여 효율적
- 전체 QuerySet을 평가하지 않고 결과의 존재 여부만 확인
- 대량의 QuerySet에 있는 특정 객체 검색에 유용

## exists 적용 예시 (1/2)

---

```
# articles/views.py

@login_required
def likes(request, article_pk):
    article = Article.objects.get(pk=article_pk)
    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')
```

적용 전

```
# articles/views.py

@login_required
def likes(request, article_pk):
    article = Article.objects.get(pk=article_pk)
    if article.like_users.filter(pk=request.user.pk).exists():
        article.like_users.remove(request.user)
    else:
        article.like_users.add(request.user)
    return redirect('articles:index')
```

적용 후

## exists 적용 예시 (2/2)

---

```
# articles/views.py

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

적용 전

```
# articles/views.py

@login_required
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():
            person.followers.remove(request.user)
        else:
            person.followers.add(request.user)
    return redirect('accounts:profile', person.username)
```

적용 후

# 한꺼번에 dump 하기

## 한꺼번에 dump 하기

---

```
# 모델 3개를 json 파일 1개로 추출
$ python manage.py dumpdata --indent 4 articles.article articles.comment accounts.user > data.json

# 모든 모델을 json 파일 1개로 추출
$ python manage.py dumpdata --indent 4 > data.json
```

- 다만 모든 데이터를 한 번에 추출 할 경우 파일 용량이 커질 수 있으므로, 필요에 따라 특정 앱만 추출하거나, 파일을 압축하여 관리하는 방법을 고려
- 데이터베이스 변경이 잦은 경우 전체 추출보다는 앱 단위 또는 모델 단위로 관리하는 편이 유지 보수에 용이

# loaddata 인코딩 에러 해결법

## 인코딩 문제

---

- JSON 파일 생성 및 로딩 시, 파일이 특정 문자 인코딩(예: UTF-8)으로 저장되지 않으면 한글 등 비ASCII 문자가 깨지거나, `UnicodeDecodeError` 등의 에러가 발생할 수 있음
- 윈도우 환경에서 생성한 파일을 리눅스 환경에서 로딩 할 때, 혹은 반대 상황에서 인코딩 이슈가 빈번히 발생

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc0 in position ...
```

## 해결 방법 1.

---

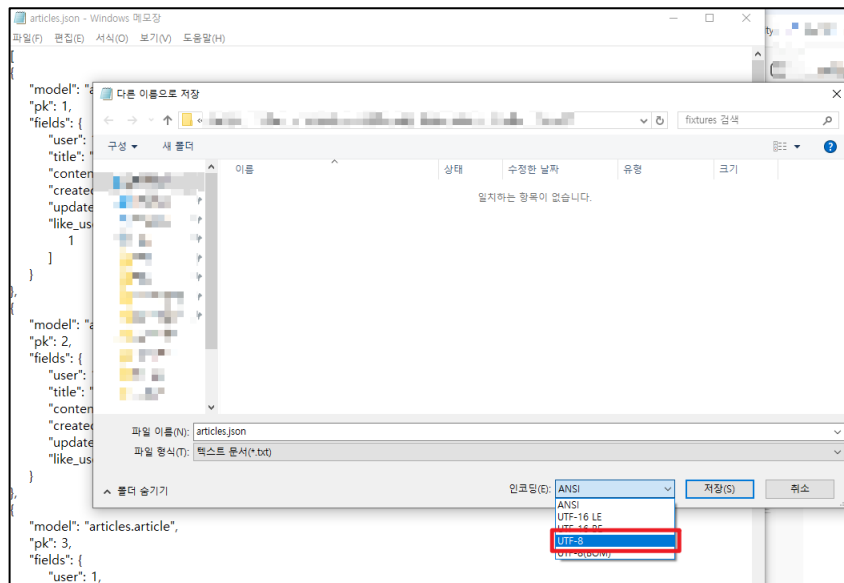
- dumpdata 시 추가 명령어 작성

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

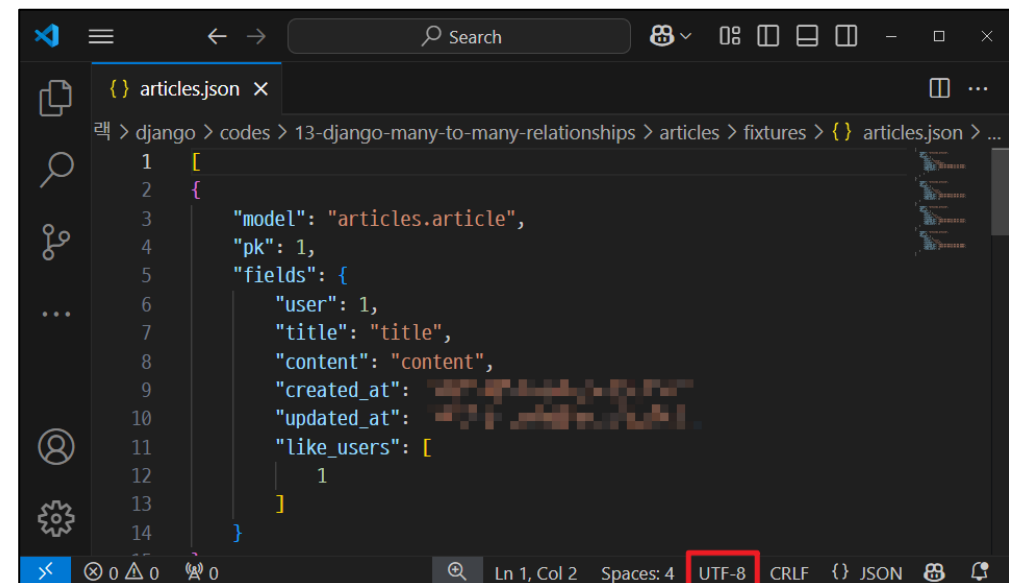


## 해결 방법 2.

- 이미 추출된 fixtures 파일이 있다면  
에디터(메모장, VSCode 등)에서 파일을 열고 인코딩을 **UTF-8**로 지정한 뒤  
다시 저장



메모장



VSCode

다음 시간에  
만나요!

삼성 청년 SW 아카데미