장고 핵심 기능 - Model

모델 정의

❖ 모델

- 테이블을 정의하는 장고의 클래스
- `models.py` 파일에 테이블 관련 사항들을 정의

❖ ORM(Object Relation Model)

- 테이블을 클래스로 정의하여 처리
- 클래스의 변수와 메서드로 테이블의 정보를 관리

모델 정의

❖ 모델

- 클래스명 → 테이블명
- 클래스 변수 → 테이블의 컬럼
- 메서드 → 테이블에 대한 쿼리 및 조작

- `django.db.models.Model` 클래스를 상속

모델 정의

❖ 모델

```
# 모델 클래스
class Album(models.Model):
    # 필드명 = 필드타입(필드옵션)
    name = models.CharField(max_length=50)
    description = models.CharField('One line Description',
                                   max_length=100, blank=True)
    owner = models.ForeignKey(User, null=True)


    class Meta:
        ordering=['name']

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('photo:albmum_detail', args=(self.id,))
```

모델 정의

❖ 모델 속성

- 테이블의 컬럼은 모델 클래스의 속성으로 정의

❖ 모델 메서드

- 모두 객체 메서드
- `self` 인자를 가짐
- 테이블 단위가 아닌 레코드 단위에만 영향을 미침

❖ Manager 클래스

- 테이블 레벨의 동작을 처리
  - 모든 레코드 수 카운트 등

모델 정의

❖ 모델 메서드

- DetailView
  - get_absolute_url()
    - URL을 표현하기위해 하드 코딩을 하지 않아도 됨
- 필드타입이 DateField 또는 DateTimeField면서 옵션이 null=True가 아닌 경우
  - get_next_by_<필드명>(**kwargs)
  - get_previous_by_<필드명>(**kwargs)
- 필드 옵션에 choices가 있는 경우
  - get_<필드명>_display()
    - 필드의 설명 문자열을 반환

모델 정의

❖ Meta 내부 클래스 속성

- 필드 이외의 항목을 정의
- ordering
  - 모델 객체의 리스트 출력시 정렬할 필드명 지정
    - 디폴트는 오름차순, - 접두사를 붙이면 내림차순으로 정렬

```
ordering = ['-pub_date', 'author']
```

- db_table
  - 테이블 이름 지정
  - 생략시 앱명_클래스명(소문자)
    - blog 앱의 Post 모델인 경우 → blog_posts

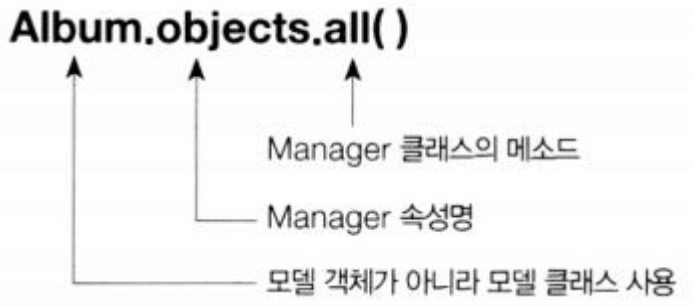
```
db_table = 'tb_post'
```

- verbose_name/verbose_name_plural
  - 모델 객체의 별칭/모델 객체 별칭의 복수형
  - 클래스명인 FavoritePost인 경우 디폴트 → 'favorite post'

모델 정의

❖ Manager 속성

- 모든 모델은 반드시 Manager 속성을 가짐
- 명시적으로 지정하지 않으면 디폴트 이름은 objects
- 모델 클래스를 통해서만 접근 가능
  - 모델 인스턴스를 통해서는 접근 불가
- models.Manager 타입
  - 테이블 레벨의 데이터베이스 쿼리 처리



  - QuerySet 객체를 반환
  - all(), filter(), exclude(), get(), count()등의 메서드 제공

모델간의 관계

❖ 테이블 간의 관계

- 1:N(one-to-many)
- N:N(many-to-many)
- 1:1(one-to-one)

❖ 장고가 처리하는 관계의 특성

- 관계는 양방향 개념
  - 양쪽 모델에서 정의가 필요한 것이 원칙
  - 한 쪽 클래스에서만 관계를 정의하면, 이를 바탕으로 장고가 상대편 정의를 자동으로 정의
- 한쪽 방향으로 관계를 생성하거나 변경하면 반대 방향으로의 관계도 그에 따라 변경됨

모델간의 관계

❖ 1:N 관계(One-to-Many)

- 테이블간 1:N 관계 맺기
  - ForeignKey 필드 타입 사용
- 필수 인자로 관계를 맺고자 하는 모델 클래스를 지정
- N 모델에서 ForeignKey 필드를 정의
- ForeignKey 필드의 필수 인자로 1모델을 지정하는 방식

```
from django.contrib.auth.models import User

class Album(models.Model):
    name  = models.CharField('NAME', max_length=30)
    description = models.CharField('One Line Description',
                                   max_length=100, blank=True)
    owner = models.ForeignKey('auth.User', on_delete=models.CASCADE,
                              verbose_name='OWNER', blank=True, null=True)
```

모델간의 관계

❖ 1:N 관계

```
$ python manage.py shell

# User:Album = 1:N 관계인 두 모델을 임포트
>>> from django.contrib.auth.models import User
>>> from photo.models import User, Album

# User 테이블의 모든 레코드를 확인
>>> User.objects.all()

# Album 테이블의 모든 레코드를 확인
>>> Album.objects.all()

# 앨범 객체를 하나 조회해 소유자를 확인
>>> a2 = Album.objects.all()[2]
>>> a2
<Album: 국가별>
>>> a2.owner
<User: shkim>

# 앨범 객체를 통해 사용자 객체에 액세스해 사용자의 이름을 확인
>>> a2.owner.username
'shkim'
```

모델간의 관계

❖ 1:N 관계

```
# 사용자 객체를 조회해 u1 및 u2 객체에 대입
>>> u1 = User.objects.get(username='shkim')
>>> u2 = User.objects.get(username='guest')

# 앨범을 만들고 앨범의 소유자를 지정하는 2가지 방법
# (방법1) 새로운 앨범을 만들고 이 앨범의 소유자를 u1 사용자로 지정
# add() 메소드를 사용
>>> newa1 = Album(name='TestAlbum1')
>>> newa1
<Album: TestAlbum1>
>>> newa1.save()
>>> u1.album_set.add(newa1)
>>> u1.album_set.all()
<QuerySet [Album: TestAlbum1>, <Album: 국가별>]>
```

모델간의 관계

❖ 1:N 관계

```
# (방법2) 새로운 앨범을 만들 때부터 u2 사용자를 지정
# create() 메서드를 사용
>>> newa2 = u2.album_set.create(name='TestAlbum2')
>>> newa2
<Album: TestAlbum2>
>>> u2.album_set.all()
[<Album: TestAlbum2>]

# 방금 만든 앨범 newa1의 소유자를 u1->u2로 변경
>>> u2.album_set.add(newa1)
>>> newa1.owner
<User: guest>
>>> u2.album_set.all()
[<Album: TestAlbum1>, <Album: TestAlbum2>]
>>> u1.album_set.all()
<QuerySet [<Album: 국가별>]>

# 소유자가 u2인 앨범의 개수를 확인
>>> u2.album_set.count()
2
```

모델간의 관계

❖ 1:N 관계

```
# 모델 간 관계에서도 필드 검색 오퍼레이션이 가능
>>> u2.album_set.filter(name__startswith='Test')
<QuerySet [<Album: TestAlbum1>, <Album: TestAlbum2>]>

>>> Album.objects.filter(owner_username='guest')
<QuerySet [<Album: TestAlbum1>, <Album: TestAlbum2>]>

# 조건이 2개 이상이면 AND 오퍼레이션을 수행
>>> Album.objects.filter(owner=u2, name_startswith='Test')
<QuerySet [<Album: TestAlbum1>, <Album: TestAlbum2>]>
```

모델간의 관계

❖ 1:N 관계

```
# 앨범의 소유자를 지정하는 여러 가지 방법
>>> Album.objects.filter(owner_pk=1)
<QuerySet [<Album: 국가별>]>

>>> Album.objects.filter(owner=1)
<QuerySet [<Album: 국가별>]>

>>> Album.objects.filter(owner=u1)
<QuerySet [<Album: 국가별>]>

>>> Album.objects.filter(owner_in=[1]).distinct()
<QuerySet [<Album: 국가별>]>

>>> Album.objects.filter(owner_in=[u1]).distinct()
<QuerySet [<Album: 국가별>]>

>>>
Album.objects.filter(owner_in=User.objects.filter(username='shkim')).distinct()
<QuerySet [<Album: 국가별>]>
```

모델간의 관계

❖ 1:N 관계

```
# 반대 방향으로, 소유자의 앨범을 지정하는 여러가지 방법
>>> User.objects.filter(album_pk=6)
<QuerySet [<User: shkim>]>

>>> User.objects.filter(album=6)
<QuerySet [<User: shkim>]>

>>> User.objects.filter(album=a2)
<QuerySet [<User: shkim>]>

# 반대 방향으로, 필드 검색 오퍼레이션
>>> User.objects.filter(album_name__startswith='Test')
<QuerySet [<User: guest>, <User: guest>]>

>>> User.objects.filter(album_name__startswith='Test').distinct()
<QuerySet [<User: guest>, <User: guest>]>

>>> User.objects.filter(album_name__startswith='Test').distinct().count()
1
```

모델간의 관계

❖ 1:N 관계

```
# 순환방식으로도 필드 검색 오퍼레이션 가능
>>> User.objects.filter(album__owner=u1)
<QuerySet [<User: shkim>]>

>>> Album.objects.filter(owner__album=a2)
<QuerySet [<Album: 국가별>]>
```

모델간의 관계

❖ 1:N 관계

```
# 1:N 관계에서 1쪽의 객체를 지우면 CASCADE로 동작해 N 쪽의 객체도 삭제
>>> u3 = User.objects.create(username='guest3')
>>> u3.album_set.create(name='TestAlbum3')
<Album: TestAlbum3>

# 삭제전, u3 소유의 앨범을 확인
>>> u3.album_set.all()
[<Album: TestAlbum3>]

# delete() 메소드는 삭제된 개수를 반환
>>> u3.delete()
(2, { ... })

# guest3 객체가 삭제된 것을 확인
>>> User.objects.all()
<QuerySet [<User: shkim>, <User: guest>]>

# u3 소유의 TestAlbum3 앨범도 같이 삭제됨
>>> Album.objects.all()
<QuerySet [<Album: Django>, <Album: Nature>, <Album: TestAlbum1>, <Album:
TestAlbum2>,
<Album: 국가별>, <Album: 사람들>]>
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

- 모델의 필드 정의시 `ManyToManyField` 필드 타입 사용
- 관계를 맺고자 하는 모델 클래스를 필수 인자로 지정
- 두 모델 중 어느쪽이라도 가능하지만, 한쪽에만 정의해야 하며 양쪽에 정의하면 안됨

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

photo/models.py]

```
class Album(models.Model):
    name = models.CharField('NAME', max_length=30)
    description = models.CharField('One Line Description',
                                   max_length=100, blank=True)
    owner = models.ForeignKey('auth.User', on_delete=models.CASCADE,
                              verbose_name='OWNER', blank=True, null=True)
    :

class Publication(models.Model):
    title = models.CharField(max_length=30)
    albums = models.ManyToManyField(Album)
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

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

>>> import photo.models import Album, Publication

# 출판물 객체 3개를 만들고 테이블에 저장
>>> p1 = Publication(title='The Python Journal')
>>> p1.save()
>>> p2 = Publication(title='Science News')
>>> p2.save()
>>> p3 = Publication(title='Science Weekly')
>>> p3.save()

# 출판물 객체 전체 리스트를 확인
>>> Publication.objects.all()

>>> Album.objects.all()
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

```
# 테이블에 있는 앨범 객체 하나를 조회해 가져옴
>>> a1 = Album.objects.get(name='Dajngo')

# 출판물 p1에 앨범 a1을 연결
>>> p1.albums.add(a1)

# 출판물 p1에 게시된 모든 앨범 리스트를 확인
>>> p1.albums.all()
<QuerySet [<Album: Django>]>


# 반대 방향으로, 앨범 a1이 게시된 모든 출판물 리스트 확인
>>> a1.publication_set.all()
<QuerySet [<Publication: Publication object (1)>]>
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

```
# 모델 간 관계에서도 다양한 필드 검색 오퍼레이션 가능
>>> Publication.objects.filter(albums=a1)
<QuerySet [<Publication: Publication object (1)>]>

>>> Publication.objects.filter(albums__pk=1)
<QuerySet [<Publication: Publication object (1)>]>

>>> Publication.objects.filter(albums__id=1)
<QuerySet [<Publication: Publication object (1)>]>

>>> Publication.objects.filter(albums=1)
<QuerySet [<Publication: Publication object (1)>]>

>>> Publication.objects.filter(albums__name__startswith='Django')
<QuerySet [<Publication: Publication object (1)>]>

>>> Publication.objects.filter(albums__in=[a1])
<QuerySet [<Publication: Publication object (1)>]>

>>> Publication.objects.filter(albums__name__startswith='Django')\
    .distinct().count()
1
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

```
# 반대 방향으로, 필드 검색 오퍼레이션 가능
>>> Album.objects.filter(publication=p1)
<QuerySet [<Album: Django>]>

>>> Album.objects.filter(publication=1)
<QuerySet [<Album: Django>]>

>>> Album.objects.filter(publication__title__startswith='The')
<QuerySet [<Album: Django>]>

>>> Album.objects.filter(publication__in=[p1])
<QuerySet [<Album: Django>]>

# 모델 간 관계에서도 filter()와 마찬가지로 exclude() 메서드 가능
>>> Publication.objects.exclude(albums=a1)
<QuerySet [<Publication: Publication object (2)>, <Publication: Publication
object (3)>]>
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

```
# 실습을 위해 앨범 a2와 출판물 p2간에 관계를 맺어줌
>>> a2 = Album.objects.get(name='TestAlbum2')
>>> a2.publication_set.add(p2)
>>> a2.publication_set.all()
<QuerySet [<Publication: Publication object (2)>]>

>>> p2.albums.all()
<QuerySet [<Album: TestAlbum2>]>

# 앨범 쪽에서 삭제 - 2개의 레코드가 삭제됨
>>> a2.delete()
(2, { ... })

# a2 앨범이 삭제되어었음
>>> Album.objects.all()
<QuerySet [<Album: Django>, <Album: Nature>, <Album: TestAlbum1>, <Album: 국가
별>, <Album: 사람들>]>

# p2 출판물은 삭제되지 않았음.(ForeignKey 관계에서의 CASCADE 동작과 다름)
<QuerySet [<Publication: Publication object (1)>, <Publication: Publication
object (2)>, <Publication: Publication object (3)>]>
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

```
# 연결이 끊어져 액세스 불가
>>> a2.publication_set.all()
예외 발생

# p2 출판물에 연결된 앨범이 없음
>>> p2.albums.all()
<QuerySet []>
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

```
# 삭제-2 실습을 위해 앨범 a3와 출판물 p3 간에 관계를 맺어줌
>>> a3 = Album.objects.get(name='TestAlbum1')
>>> a3.albums.add(a3)
>>> a3.albums.all()
<QuerySet [<Album: TestAlbum1>]>
>>> a3.publication_set.all()
<QuerySet [<Publication: Publication object (3)>]>

# 출판물 쪽에서 삭제 - 2개의 레코드가 삭제
>>> p3.delete()
(2, { .... })

# p3 출판물이 삭제됨
>>> Publication.objects.all()
<QuerySet [<Publication: Publication object (1)>, <Publication: Publication
object (2)>]>

# a3 앨범은 삭제되지 않음
>>> Album.objects.all()
<QuerySet [<Album: Django>, <Album: Nature>, <Album: TestAlbum1>, <Album: 국가
별>, <Album: 사람들>]>
```

모델간의 관계

❖ N:N 관계(Many-to-Many) 관계

```
# 연결이 끊어져 액세스가 안 됨
>>> p3.albums.all()
예외 발생

# a3 앨범에 연결된 출판물이 없음
>>> a3.publication_set.all()
<QuerySet []>
```

모델간의 관계

❖ 1:1(One-to-One) 관계

- OneToOneField 필드 타입을 사용
- 관계를 맺고자 하는 모델 클래스를 필수 인자로 지정
- 개념적으로 ForeignKey 필드 타입에 unique=True 옵션을 준것과 유사
- 반대 방향의 객체는 하나의 객체만 반환
- to, on_delete는 필수 인자

모델간의 관계

❖ 1:1(One-to-One) 관계

[photo/models.py]

```
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

    def __str__(self):
        return f"Place-{self.name}"


class Restaurant(models.Model):
    place = models.OneToOneField(Place, on_delete=models.CASCADE)
    name = models.CharField(max_length=50)
    server_pizza = models.BooleanField(default=False)

    def __str__(self):
        return f"Restaurant-{self.name}"
```

모델간의 관계

❖ 1:1(One-to-One) 관계

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

# Place:Restaurant = 1:1 관계인 두 모델을 임포트
>>> from photo.models import Place, Restaurant

# Place 객체 2개를 만들고 데이터베이스 반영
>>> p1 = Place(name='TestPlace1', address='Seoul')
>>> p1.save()
>>> p2 = Place(name='TestPlace2', address='Jeju')
>>> p2.save()
# Restaurant 객체 1개를 만들고(위치는 p1) 데이터베이스에 반영
>>> r = Restaurant.objects.create(place=p1, name='TestRestaurant')

# 새로 만든 객체들을 확인
>>> p1
<Place: Place-TestPlace1>
>>> p2
<Place: Place-TestPlace2>
>>> r
<Restaurant: Restaurant-TestRestaurant>
```

모델간의 관계

❖ 1:1(One-to-One) 관계

```
# Restaurant에서 Place 방향으로 액세스가 가능
>>> r.place
<Place: Place-TestPlace1>

# 반대 방향, 즉 Place에서 Restaurant 방향으로도 액세스 가능
>>> p1.restaurant
<Restaurant: Restaurant=TestRestaurant>

# 식당의 장소를 p1->p2로 변경
>>> r.place = p2
>>> r.save()

# 식당의 장소가 변경된 것을 확인
>>> r.place
<Place: Place-TestPlace2>
>>> p2.restaurant
<Restaurant: Restaurant-TestRestaurant>

# 메모리에는 예전 내용이 남아 있다는 점 유의
>>> p1.restaurant
<Restaurant: Restaurant-TestRestaurant>
```

모델간의 관계

❖ 1:1(One-to-One) 관계

```
# 변경된 데이터베이스 내용으로 p1 및 p2 객체를 현행화
>>> p1 = Place.objects.get(name='TestPlace1')
>>> p2 = Place.objects.get(name='TestPlace2')

# 식당의 장소를 원래대로, 즉 p2->p1으로 변경
# 이번에는 반대편 객체, 즉 Place 객체에서 변경
>>> p1.restaurant = r

# 데이터베이스에 반영하기 위해서는 p1 또는 r 객체를 save()
>>> r.save()

# 변경된 데이터베이스 내용으로 p1 및 p2 객체를 현행화
>>> p1 = Place.objects.get(name='TestPlace1')
>>> p2 = Place.objects.get(name='TestPlace2')

# 원래 장소로 변경된 것을 확인
>>> p1.restaurant
<Restaurant: Restaurant-TestRestaurant>
>>> p2.restaurant
예외 발생
>>> r.place
<Place: Place-TestPlace1>
```

모델간의 관계

❖ 1:1(One-to-One) 관계

```
# 장소 객체 전체 리스트 확인
>>> Place.objects.all()
<QuerySet [<Place: Place-TestPlace1>, <Place: Place-TestPlace2>]>

# 식당 객체 전체 리스트를 확인
>>> Restaurant.objects.all()
<QuerySet [<Restaurant: Restaurant-TestRestaurant>]>

# 1:1 관계에서도 다양한 필드 검색 오퍼레이션 가능
>>> Restaurant.objects.get(place=p1)
<Restaurant: Restaurant-TestRestaurant>

>>> Restaurant.objects.get(place__pk=1)
<Restaurant: Restaurant-TestRestaurant>

>>> Restaurant.objects.get(place__name__startswith='Test')
<Restaurant: Restaurant-TestRestaurant>

>>> Restaurant.objects.get(place__address__contains='Seo')
<Restaurant: Restaurant-TestRestaurant>
```

모델간의 관계

❖ 1:1(One-to-One) 관계

```
# 반대 방향으로도 필드 검색 오퍼레이션 가능
>>> Place.objects.get(pk=1)
<Place: Place-TestPlace1>

>>> Place.objects.get(restaurant=r)
<Place: Place-TestPlace1>

>>> Place.objects.get(restaurant__place=p1)
<Place: Place-TestPlace1>

>>> Place.objects.get(restaurant__place__name__startswith='Test')
<Place: Place-TestPlace1>
```

관계 매니저(RelatedManager)

❖ 관계 매니저

- Manager 클래스 중에서 모델 간 관계에 대한 기능 및 데이터베이스 쿼리를 담당하는 클래스
- 모델 간 관계를 다루기 위한 클래스

관계 매니저(RelatedManager)

❖ 관계 매니저 클래스를 사용하는 경우

- 1:N 및 N:N 관계에서만 관계 매니저가 사용됨
  - 1:1 관계에서는 사용되지 않음

- 1:N(One-to-Many) 관계
  - 예, User:Album 모델 관계

```
user1.album_set     # album_set은 관계 매니저 클래스의 객체임
```

- N:N의 관계 매니저
  - 예

```
album1.publication_set  # publication_set은 관계 매니저 클래스의 객체
publication1.albums     # albums는 ManyToManyField 타입의 필드명이면서 관계 매니저 객체
```

관계 매니저(RelatedManager)

❖ 관계 매니저 메서드

- Blog와 Entry 모델 간에 1:N 관계로 가정

관계 매니저(RelatedManager)

❖ 관계 매니저 메서드

- **add(*objs, bulk=True)**
  - 인자로 주어진 모델 객체들을 관계 객체의 집합에 추가 --> 두 모델 객체 간에 관계를 맺어줌
  - 자동으로 데이터베이스 업데이트 수행

```
>>> b =Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e)  # Entry e 객체를 Blog b 객체에 연결
```

  - 관계 매니저는 자동으로 QuerySet.update() 메서드를 호출해 데이터베이스 업데이트 수행
  - bulk=False인 경우 e.save() 메서드를 호출해 데이터베이스 업데이트를 수행

  - N:N 관계에서 add() 메서드를 사용하는 경우
    - update() 또는 save() 메서드를 사용하지 않고 QuerySet.bulk_create() 메서드를 호출해 관계를 생성
    - bulk_create() 메서드는 한 번의 쿼리로 여러 개의 객체를 데이터베이스에 삽입

관계 매니저(RelatedManager)

❖ 관계 매니저 메서드

- **create(**kwargs)**
  - 새로운 객체를 생성해서 데이터베이스에 저장하고 관계 객체 집합에 넣음
  - 새로 생성된 객체를 반환
  - 상대방 모델 객체를 새로 만들어서 두 객체 간에 관계를 맺어주고 데이터베이스에 반영

```
>>> b = Blog.objects.get(id=1)
# Entry e 객체를 생성해서, 이를 Blob b 객체와의 관계를 생성함
>>> e = b.entry_set.create(headline = 'Hello', body_text='Hi',
pub_date=datetime.date(2005, 1, 1))
```

  - e.save()를 호출하지 않아도 자동으로 데이터베이스에 저장
  - 또한 create() 메서드에는 blog 인자를 사용하지 않음

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry(
    blog=b,
    headline='Hello',
    body_text='Hi',
    pub_date=datetime.date(2005, 1, 1)
)
>>> e.save(force_insert=True)
```

관계 매니저(RelatedManager)

❖ 관계 매니저 메서드

- **remove(*objs, bulk=True)**
  - 인자로 지정된 모델 객체들을 관계 집합에서 삭제
  - 두 모델 객체 간의 관계를 해제

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.remove(e)  # Blog b 객체에서 Entry e 객체와 관계를 끊음
                           # e.blong = None 과 같음
```

  - 1:N 관계에서 bulk
    - True: QuerySet.update() 메서드 사용
    - False: 모델 객체마다 save() 메서드 호출
  - N:N 관계에서 remove() 메서드 호출
    - bulk 인자는 사용할 수 없음
    - QuerySet.delete() 메서드를 호출해 관계를 삭제

관계 매니저(RelatedManager)

❖ 관계 매니저 메서드

- **clear(bulk=True)**
  - 관계 객체 집합에 있는 모든 객체를 삭제
  - 해당 모델 객체가 맺고 있는 다른 객체들과의 관계를 모두 제거

```
>>> b = Blog.objects.get(id=1)
>>> e.entry_set.clear()
```

  - ForeignKey에 사용될 때는 null=True일 경우만 사용 가능
  - bulk 인자에 따라 내부 실행 방법이 달라짐

관계 매니저(RelatedManager)

❖ 관계 매니저 메서드

- set(objs, bulk=True, clear=False)
  - 관계 객체 집합의 내용을 변경

```
>>> new_list = [obj1, obj2, obj3]
>>> e.related_set.set(new_list)
```

  - 내부적으로 add(), remove(), clear() 메서드가 적절히 조합되어 실행
  - bulk 인자는 remove()와 동일
  - clear 인자
    - False : 기존 항목을 체크하여, 지울 항목은 remove()로, 추가할 항목은 add() 메서드로 실행
    - True : clear() 메서드로 기존 항목을 한꺼번에 모두 지운 후, add()로 new_list 내용을 새로 한꺼번에 추가