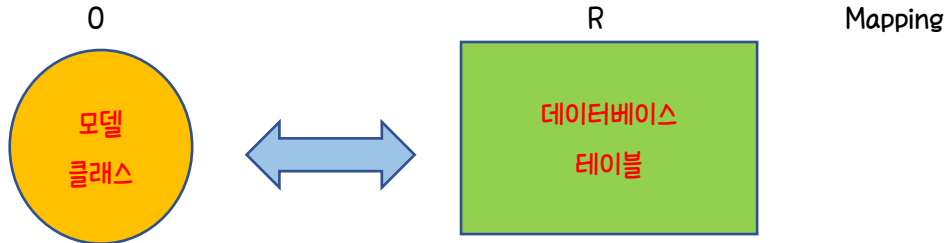


## Django 모델 (Model)

Django에서 Model은 데이터 서비스를 제공한다. Django의 Model은 각 Django App안에 기본적으로 생성되는 models.py 모듈 안에 정의하게 된다. models.py 모듈 안에 하나 이상의 모델 클래스를 정의할 수 있으며, 하나의 모델 클래스는 데이터베이스에서 하나의 테이블에 해당된다.



Django 모델은 "django.db.models.Model" 의 자식 클래스이며, **모델의 필드**는 클래스의 속성(Attribute)로 표현되고 테이블의 컬럼에 해당한다. (Primary Key(기본키)가 지정되지 않으면, 모델에 Primary Key 역할을 하는 id 필드가 자동으로 추가되며 DB 테이블 생성시 자동으로 값이 1씩 증가되는 id 컬럼이 생성된다) Django 에서 필드는 모델을 생성할 때 필수적인 요소이며 필드를 clean, save, delete 등과 같이 모델 API 와 동일한 이름으로 생성하지 않도록 주의해야 한다.

```
class 모델이름(models.Model):  
    필드이름1 = models.필드타입(필드옵션)  
    필드이름2 = models.필드타입(필드옵션)
```

모델 클래스에 필드를 정의하기 위해 인스턴스 변수가 아닌 **클래스 변수**로 정의하며, 변수에는 테이블의 컬럼의 메타 데이터를 정의한다. 필드를 정의하는 각각의 클래스 변수는 models.CharField(), models.IntegerField(), models.DateTimeField(), models.TextField() 등의 각 필드 타입에 맞는 Field 클래스 객체를 생성하여 할당한다. Field 클래스는 여러 종류가 있는데, 생성자 함수 호출시 필요한 옵션들을 지정할 수 있다.

### - 필드 타입

모델의 필드에는 다음과 같이 다양한 타입들이 있다. 모든 필드 타입 클래스들은 "Field" 클래스의 자손 클래스들이다.

Field Type	설명
CharField	제한된 문자열 필드 타입. 최대 길이를 <b>max_length</b> 옵션에 지정해야 한다. 문자열의 특별한 용도에 따라 CharField의 파생클래스로서, 이메일 주소를 체크를 하는 EmailField, IP 주소를 체크를 하는 GenericIPAddressField, 콤마로 정수를 분리한 CommaSeparatedIntegerField, 특정 폴더의 파일 패스를 표현하는 FilePathField, URL을 표현하는 URLField 등이 있다.
TextField	대용량 문자열을 갖는 필드
IntegerField	32 비트 정수형 필드. 정수 사이즈에 따라 BigIntegerField, SmallIntegerField 을 사용할 수도 있다.
BooleanField	true/false 필드. Null 을 허용하기 위해서는 NullBooleanField를 사용한다.

<b>DateTimeField</b>	날짜와 시간을 갖는 필드. 날짜만 가질 경우는 DateField, 시간만 가질 경우는 TimeField를 사용한다. <b>auto_now_add(생성)과 auto_now(수정)을 true로 설정하면 생성 또는 수정시 기본 타임존 시간으로 변경된다.</b>
<b>DecimalField</b>	소숫점을 갖는 decimal 필드
<b>BinaryField</b>	바이너리 데이터를 저장하는 필드
<b>FileField</b>	파일 업로드 필드
<b>ImageField</b>	FileField의 파생클래스로서 이미지 파일인지 체크한다.

위와 같은 필드 타입 클래스 이외에, Django 프레임워크는 테이블 간 혹은 필드 간 **관계(Relationship)**를 표현하기 위해 **ForeignKey, ManyToManyField, OneToOneField** 클래스를 또한 제공하고 있다. 특히 **ForeignKey**는 모델 클래스간의 Many-To-One (혹은 One-To-Many) 관계를 표현하기 위해 흔히 사용된다.

#### - 필드 옵션

모델의 필드는 필드 타입에 따라 여러 옵션(혹은 Argument)를 가질 수 있다. 예를 들어, CharField는 문자열 최대 길이를 의미하는 `max_length` 라는 옵션을 갖는다. 필드 옵션은 일반적으로 생성자에서 아규먼트로 지정한다. 다음은 모든 필드 타입에 적용 가능한 옵션들 중 자주 사용되는 몇 가지를 요약한 것이다.

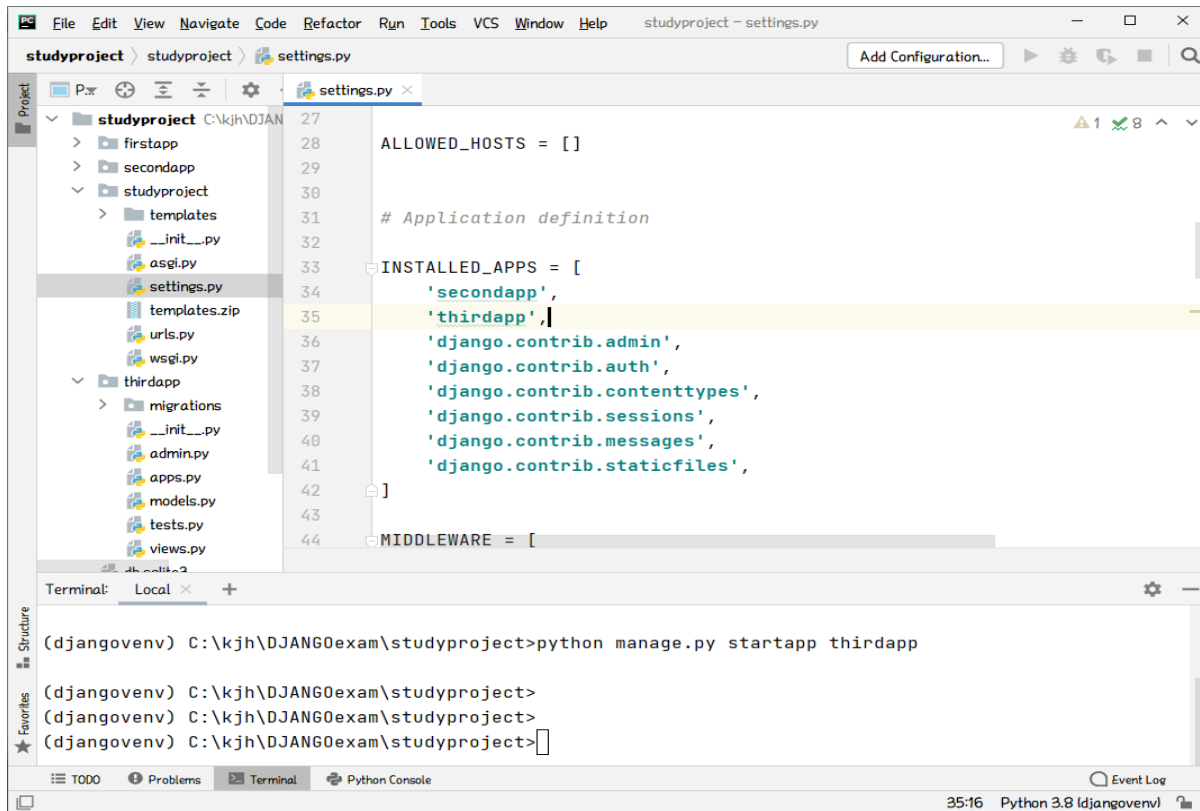
필드 옵션	설명
<code>null (Field.null)</code>	<code>null=True</code> 이면, Empty 값을 DB에 NULL로 저장한다. DB 테이블에서 Null이 허용되는 필드가 된다. 예) <code>models.IntegerField(null=True)</code>
<code>blank (Field.blank)</code>	<code>blank=False</code> 이면, Required 필드가 된다. <code>blank=True</code> 이면, Optional 필드이다. 예) <code>models.DateTimeField(blank=True)</code>
<code>primary_key (Field.primary_key)</code>	해당 필드가 Primary Key임을 표시한다. 예: <code>models.CharField(max_length=10, primary_key=True)</code> 모델 클래스레 <b>Primary Key</b> 로 설정되는 필드가 없으면 <b>id</b> 라는 필드가 자동 생성된다.
<code>unique (Field.unique)</code>	해당 필드가 테이블에서 Unique함을 표시한다. 해당 컬럼에 대해 Unique Index를 생성한다. 예) <code>models.IntegerField(unique=True)</code>
<code>default (Field.default)</code>	필드의 디폴트값을 지정한다. 예: <code>models.CharField(max_length=2, default="WA")</code>

#### - DB Migration

Django에서 Model 클래스를 생성하고 난 후, 해당 모델에 상응하는 DB 테이블을 데이터베이스에서 생성할 수 있다. Python 모델 클래스의 수정 (및 생성 )을 DB에 적용하는 과정을 **Migration**이라 부른다. 이는 Django가 기본적으로 제공하는 ORM(Object-Relational Mapping) 서비스를 통해 진행된다.

Django 모델 클래스로부터 테이블을 생성하기 위해서는 크게 Migration을 준비하는 과정과 이를 적용하는 과정으로 나뉘는데, 구체적으로는 다음과 같은 절차를 따른다.( DB Migration을 수행하려면 해당 앱이 `settings.py` 파일 안의 `INSTALLED_APPS` 리스트에 등록되어 있어야 한다.)

thirdapp을 생성하고 thirdapp을 등록한다. (반드시 생성하고 나서 등록한다.)



The screenshot shows an IDE with the `settings.py` file open. The `INSTALLED_APPS` list includes `'thirdapp',`. Below the code editor, a terminal window shows the command `python manage.py startapp thirdapp` being executed successfully in a Django environment.

```
ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = [
    'secondapp',
    'thirdapp',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

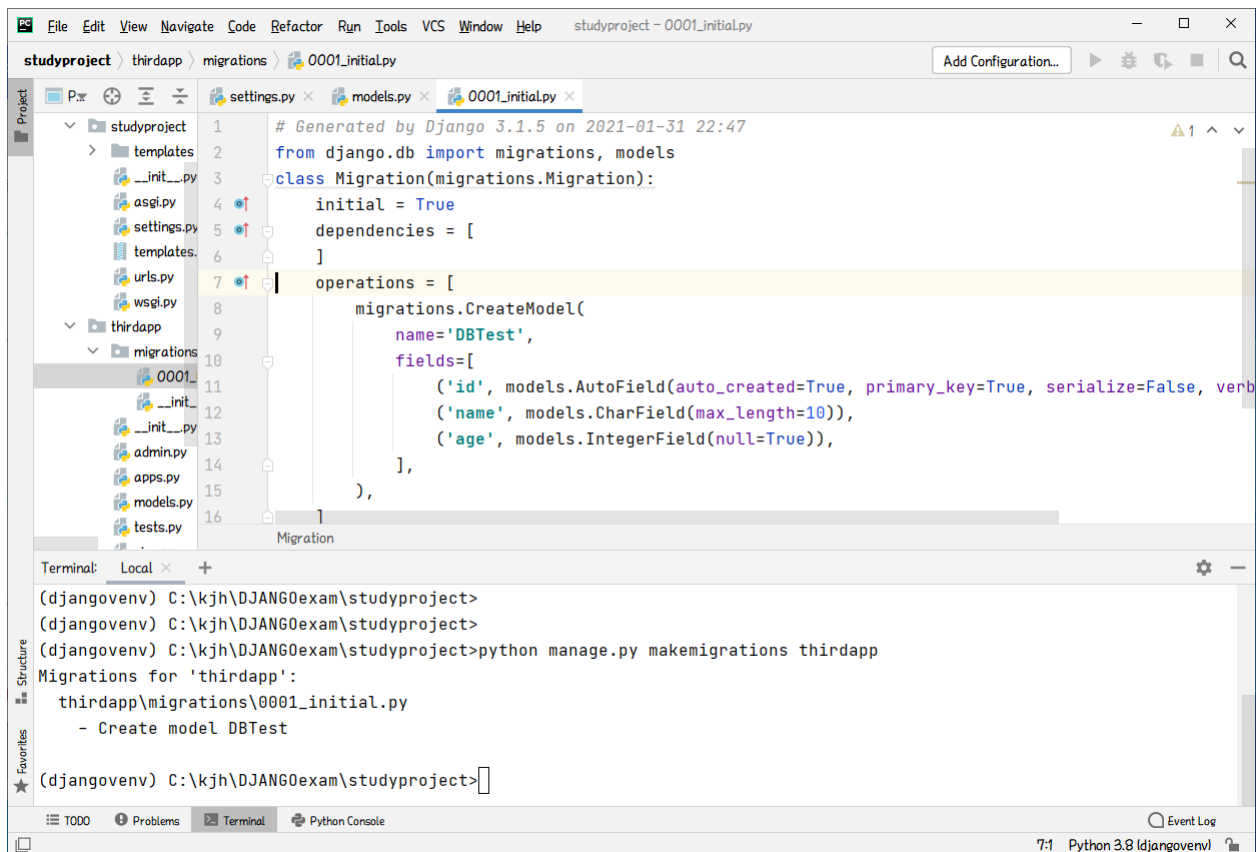
MIDDLEWARE = []
```

```
(djangoenv) C:\kjh\DJANGOexam\studyproject>python manage.py startapp thirdapp

(djangoenv) C:\kjh\DJANGOexam\studyproject>
(djangoenv) C:\kjh\DJANGOexam\studyproject>
(djangoenv) C:\kjh\DJANGOexam\studyproject>
```

[ 모델 소스를 읽고 생성될 DB 테이블의 SQL 을 수행하기 위한 파이썬 소스 생성 : migrations/0001\_initial.py]

python manage.py makemigrations thirdapp



The screenshot shows the IDE with the `migrations/0001_initial.py` file open. The file contains a Django migration class `Migration` with a `CreateModel` operation. The terminal window shows the command `python manage.py makemigrations thirdapp` being executed, resulting in the creation of the `0001_initial.py` migration.

```
# Generated by Django 3.1.5 on 2021-01-31 22:47
from django.db import migrations, models

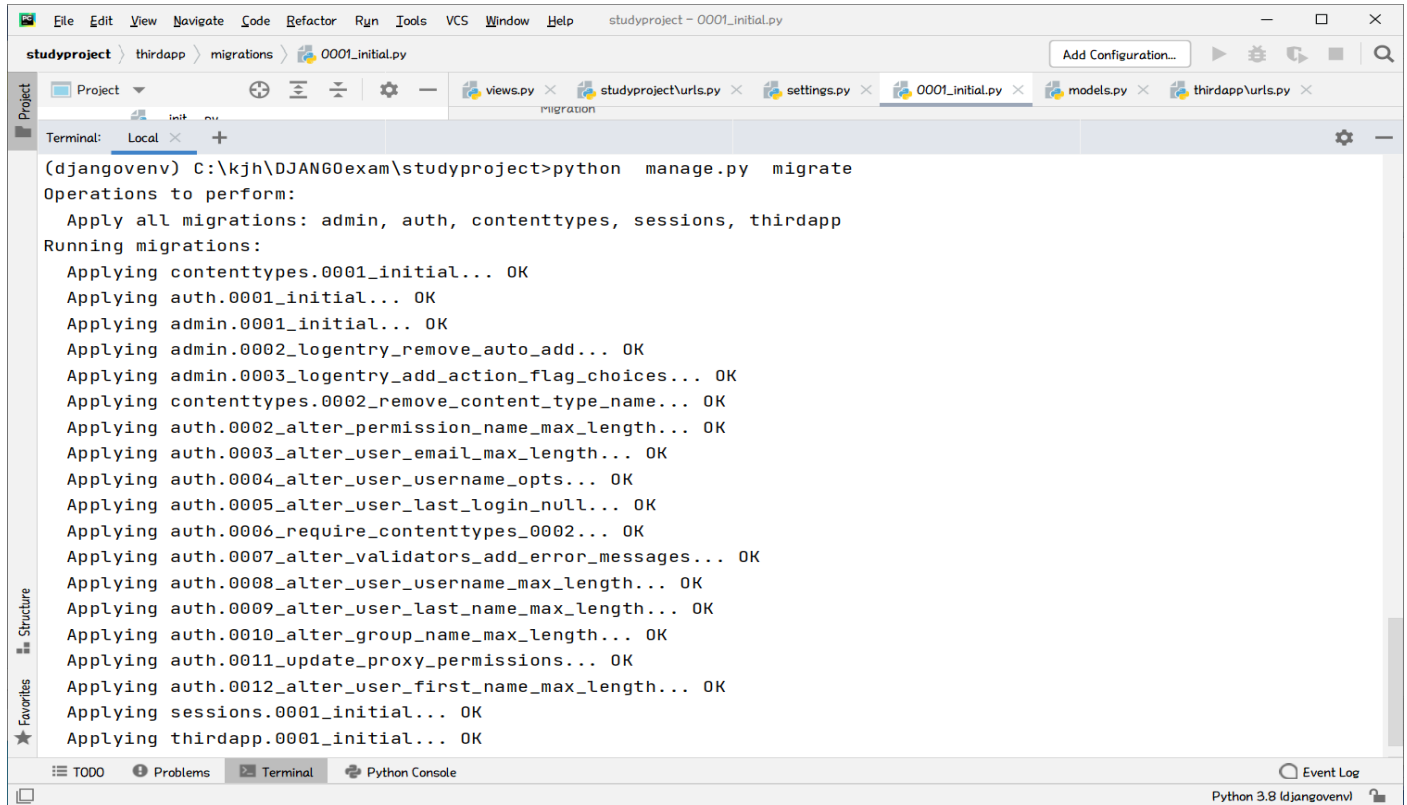
class Migration(migrations.Migration):
    initial = True
    dependencies = [
    ]
    operations = [
        migrations.CreateModel(
            name='DBTest',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=10)),
                ('age', models.IntegerField(null=True)),
            ],
        ),
    ]
```

```
(djangoenv) C:\kjh\DJANGOexam\studyproject>
(djangoenv) C:\kjh\DJANGOexam\studyproject>
(djangoenv) C:\kjh\DJANGOexam\studyproject>python manage.py makemigrations thirdapp
Migrations for 'thirdapp':
  thirdapp\migrations\0001_initial.py
    - Create model DBTest

(djangoenv) C:\kjh\DJANGOexam\studyproject>
```

## [ 생성된 SQL 을 실행시켜서 SQLite 에 테이블 생성 ]

### python manage.py migrate



```
(djangoenv) C:\kjh\DJANGOexam\studyproject>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, thirdapp
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
  Applying thirdapp.0001_initial... OK
```

### - Django 모델 API

앞의 Django 모델에서 처럼 모델 클래스를 정의하게 되면, Django는 데이터를 추가/갱신하고 읽어 들일 수 있는 다양한 데이터베이스 API 들을 자동으로 제공한다. 이러한 기능은 Django가 **ORM 서비스**를 기본적으로 제공함에 따른 것으로 데이터베이스를 편리하게 핸들링할 수 있게 도와준다.

#### [ INSERT ]

데이터를 삽입하기 위해서는 먼저 테이블에 해당하는 모델(Model Class)로부터 객체를 생성하고, 그 객체의 **save()** 메서드를 호출한다. **save()** 메서드가 호출되면, SQL의 INSERT이 생성되고 실행되어 테이블에 데이터가 추가된다.

#### [ SELECT ]

Django는 디폴트로 모든 Django 모델 클래스에 대해 **"objects"** 라는 **Manager (django.db.models.Manager)** 객체를 자동으로 추가한다. Django 에서 제공하는 이 Manager를 통해 특정 데이터를 필터링 할 수도 있고 정렬할 수도 있으며 기타 여러 기능들을 사용할 수 있다. 데이터를 읽어오기 위해서는 Django 모델의 Manager 즉 **"모델클래스.objects"** 를 사용한다.

Django Model API에는 기본적으로 제공하는 여러 쿼리 메서드들이 있는데, 자주 사용되는 주요 메서드 몇 가지만 살펴본다.

- |          |   |
|----------|---|
| all()    | 테이블 데이터를 전부 가져오기 위해서는 <b>모델클래스명.objects.all()</b> 를 사용한다. <b>QuerySet</b> 객체가 리턴된다. |
| get()    | 하나의 행(Row)을 가져오기 위해서는 <b>get()</b> 메서드를 사용하며 가져오려는 행의 pk 값 또는 id 값을 아규먼트로 지정한다.     |
| filter() | 특정 조건에 맞는 행(Row)들을 가져오기 위해서는 <b>filter()</b> 메서드를 사용한다.                             |

	<code>rows = 모델클래스명.objects.filter(name='유니코')</code>
<code>exclude()</code>	특정 조건을 제외한 나머지 Row들을 가져오기 위해서는 <code>exclude()</code> 메서드를 사용한다.  <code>rows = 모델클래스명.objects.exclude(name='유니코')</code>
<code>count()</code>	데이터의 갯수(row 수)를 알려면 <code>count()</code> 메서드를 사용한다.
<code>order_by()</code>	데이터를 키에 따라 정렬하기 위해 <code>order_by()</code> 메서드를 사용한다. <code>order_by()</code> 안에는 정렬 키를 나열할 수 있는데, 앞에 <code>-</code> 가 붙으면 내림차순이다. 다음 예는 id를 기준으로 올림차순, createDate로 내림차순으로 정렬하게 된다.  <code>rows = 모델클래스명.objects.order_by('id', '-writedata')</code>
<code>distinct()</code>	중복된 값은 하나로만 표시하기 위해 <code>distinct()</code> 메서드를 사용한다. SQL의 <code>SELECT DISTINCT</code> 와 같은 효과를 낸다. 아래는 name필드가 중복되는 경우 한 번만 표시하게 된다.  <code>rows = 모델클래스명.objects.distinct('name')</code>
<code>first()</code>	데이터들 중 처음에 있는 row만을 리턴한다. 아래는 name필드로 정렬했을 때 첫 번째 row를 리턴한다.  <code>row = 모델클래스명.objects.order_by('name').first()</code>
<code>last()</code>	데이터들 중 마지막에 있는 row만을 리턴한다.

위의 쿼리 메서드들은 실제 데이터 결과를 직접 리턴하는 것이 아니라 **쿼리 표현식(Django에서 QuerySet이라 한다)**을 리턴하는데, 여러 메서드들을 체인처럼 연결하여 사용할 수 있다. 여러 체인으로 연결되면 최종적으로 리턴된 쿼리가 해석되어 DB에는 실제 하나의 쿼리를 보내게 된다. 다음은 여러 메서드들을 사용하여 체인으로 연결한 예제이다.

```
row = 모델클래스명.objects.filter(name='유니코').order_by('-writedata').first()
```

## [ UPDATE ]

데이터를 수정하기 위해서는 먼저 수정할 행(Row)객체를 얻은 후 변경할 필드들을 수정한다. 마지막에 `save()` 메서드를 호출되면, SQL의 UPDATE 명령이 실행되어 테이블의 데이터가 갱신된다. 아래는 id가 1인 Feedback 객체에 이름을 변경하는 코드이다.

## [ DELETE ]

데이터를 삭제하기 위해서는 먼저 삭제할 행(Row)객체를 얻은 후 `delete()` 메서드를 호출한다.

## - Django 모델 API를 테스트 해보기

터미널에서 `python manage.py shell` 을 실행시키고 인터랙티브 파이썬 실행 모드를 기동시킨다.

```

Terminal: Local x +
Applying thirdapp.0001_initial... OK

(djangoven) C:\kjh\DJANGOexam\studyproject>python manage.py shell
Python 3.8.7 (tags/v3.8.7:6503f05, Dec 21 2020, 17:59:51) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>

```

```
>>> from thirdapp.models import DBTest
>>> DBTest.objects.all()
<QuerySet []>
>>>
```

```
>>> DBTest(name='유니코', age=10).save()
>>> DBTest(name='둘리', age=11).save()
>>> DBTest(name='또치', age=12).save()
>>> DBTest(name='도우너', age=10).save()
```

```
>>> DBTest.objects.all()
<QuerySet [<DBTest: 유니코:10>, <DBTest: 둘리:11>, <DBTest: 또치:12>, <DBTest: 도우너:10>]>
>>> DBTest.objects.get(id=1)
<DBTest: 유니코:10>
>>> DBTest.objects.get(id=2)
<DBTest: 둘리:11>
>>> DBTest.objects.get(id=3)
<DBTest: 또치:12>
>>> DBTest.objects.get(id=4)
<DBTest: 도우너:10>
>>> DBTest.objects.filter(name='둘리')
<QuerySet [<DBTest: 둘리:11>]>
>>> DBTest.objects.filter(age=10)
<QuerySet [<DBTest: 유니코:10>, <DBTest: 도우너:10>]>
```

```
>>> DBTest.objects.count()
4
>>> DBTest.objects.first()
<DBTest: 유니코:10>
>>> DBTest.objects.last()
<DBTest: 도우너:10>
>>> DBTest.objects.order_by('age')
<QuerySet [<DBTest: 유니코:10>, <DBTest: 도우너:10>, <DBTest: 둘리:11>, <DBTest: 또치:12>]>
>>> DBTest.objects.order_by('-age')
<QuerySet [<DBTest: 또치:12>, <DBTest: 둘리:11>, <DBTest: 유니코:10>, <DBTest: 도우너:10>]>
>>> DBTest.objects.exclude(age=10)
<QuerySet [<DBTest: 둘리:11>, <DBTest: 또치:12>]>
```

```
>>> row = DBTest.objects.get(id=1)
>>> row.name = 'unico'
>>> row.save()
>>> DBTest.objects.all()
<QuerySet [<DBTest: unico:10>, <DBTest: 둘리:11>, <DBTest: 또치:12>, <DBTest: 도우너:10>]>
>>> row = DBTest.objects.get(id=1)
>>> row.delete()
(1, {'thirdapp.DBTest': 1})
>>> DBTest.objects.all()
<QuerySet [<DBTest: 둘리:11>, <DBTest: 또치:12>, <DBTest: 도우너:10>]>
```



## ※ 필드 룩업 (Field lookups)

(<https://docs.djangoproject.com/en/3.1/topics/db/queries/>에서 소개된 주요 내용 정리)

여기서 Entry, Blog 등은 이 사이트에서 소개하고 있는 내용의 모델 클래스들이다. 자세한 것은 위의 URL 사이트를 참고한다.

필드 룩업 (Field lookups) 은 SQL의 WHERE 절(추출조건)에 해당하는 부분으로 QuerySet 객체의 메서드인 filter(), exclude(), get() 에 키워드 인자 형태로 전달된다. 필드 룩업의 기본적인 형태는 다음과 같다.

필드이름\_\_룩업타입=조건값

예를 들면, 2006년 이전에 발행된 Entry 객체들의 쿼리셋을 가져오려면 다음과 같이 할 수 있다.

Entry.objects.filter(pub\_date\_\_lte='2006-01-01') # lte: less than or equal to

```
SELECT *
FROM blog_entry
WHERE pub_date <= '2006-01-01';
```

룩업에 들어가는 필드이름은 검색하려는 모델의 필드이름이며, 예외적으로 ForeignKey 필드의 경우에는 필드이름에 \_id 가 들어간다. 외래키 필드를 검색하는 경우에는 조건값으로 외래키 필드가 참조하고 있는 레코드의 기본키 필드 값을 전달해주어야 한다.

예를 들어, 다음은 Blog 테이블의 기본키 값이 4인 레코드를 참조하고 있는 모든 Entry 테이블의 레코드들의 쿼리셋을 반환한다.

Entry.objects.filter(blog\_id=4)

잘못된 룩업 타입을 전달한 경우, TypeError 가 발생한다.

### [ 자주 사용하는 룩업 타입 ]

exact: 완전히 일치하는 값 검색.

Entry.objects.get(headline\_\_exact="Cat bites dog")

위 명령은 Entry 테이블에서 headline 변수가 정확히 "Cat bites dong" 인 레코드를 가져온다.

필드 룩업을 사용할 때, 룩업 타입, 즉, \_\_ 와 그 이후 부분을 지정하지 않으면, 자동으로 exact 를 사용한 매치를 실행한다.

Blog.objects.get(id\_\_exact=4)

Blog.objects.get(id=4)

icontains: 대소문자 구분을 하지 않는 exact.

Blog.objects.get(name\_\_icontains="beatles blog")

Beatles blog, beatles Blog, BEAtles BlOg 등등이 모두 매치됨.

contains: 조건값을 포함한 값과 매치됨. 대소문자 구분함.

Entry.objects.get(headline\_\_contains='Lennon')

위 명령은 아래 SQL 문과 같다.

```
SELECT *
FROM Entry
WHERE headline
LIKE '%Lennon%';
```

icontains: contains 의 대소문자 무시 버전.

startswith, endswith: 각각 조건값으로 시작, 끝나면 매치된다. 대소문자 구분함.

istartswith, iendswith: startswith, endswith 의 대소문자 무시 버전.

[ 다양한 필드 룩업의 예 ]

```
e = Entry.objects.filter(pub_date__year=2006)
```

```
e = Entry.objects.exclude(pub_date__year=2006)
```

```
e = Entry.objects.filter(headline__startswith='What')
    .exclude(pub_date__gte=datetime.date.today())
    .filter(pub_date__gte=datetime.date(2005, 1, 5))
```

```
e1 = Entry.objects.filter(headline__startswith='What')
```

```
e2 = e1.exclude(pub_date__gte=datetime.date.today())
```

```
e3 = e1.filter(pub_date__gte=datetime.date(2005, 1, 5))
```

```
one_entry = Entry.objects.get(pk=1)
```

get() 을 이용할 경우 매치되는 값이 없으면 **모델클래스.DoesNotExist** 가 발생하게 된다. get() 에 매치된 결과가 여러 개일 경우에도 에러가 발생한다. 이 경우 발생하는 에러는 **모델클래스.MultipleObjectsReturned** 이다.

[ 쿼리 제한하기 ]

Python의 슬라이싱 문법을 통해서 쿼리셋에 들어갈 데이터를 제한할 수 있다. SQL의 LIMIT 과 OFFSET 문과 같은 역할을 한다.

(음수 슬라이싱은 지원하지 않는다.)

```
Entry.objects.all()[:5]
```

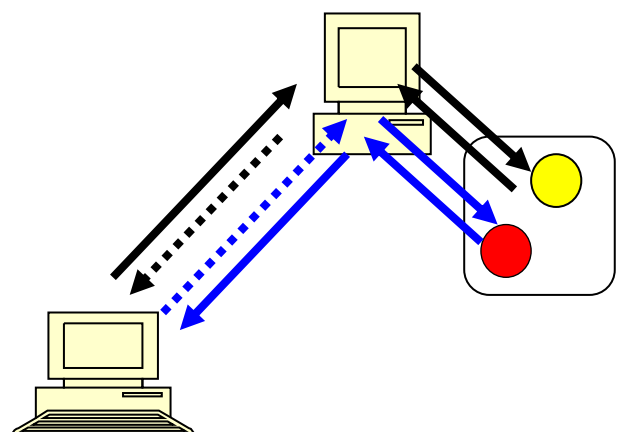
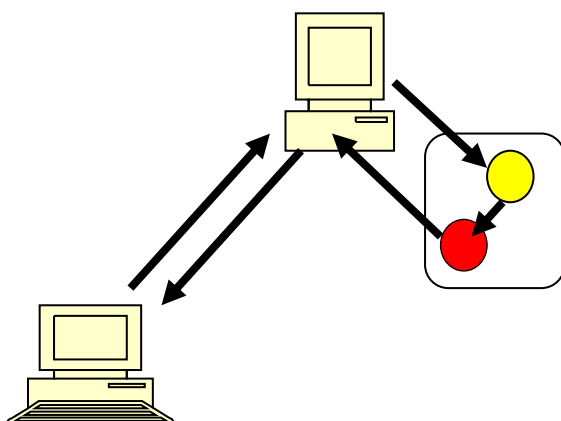
```
Entry.objects.all()[5:10]
```

```
Entry.objects.all()[10:20]
```

하나의 값을 가져오려면 인덱싱을 해준다. 가져오려는 인덱스의 값이 없을 경우 IndexError 를 일으킨다.

```
Entry.objects.order_by('headline')[0]
```

[ redirect 와 render ]





## ORM(Object-Relation Mapping)

### ORM

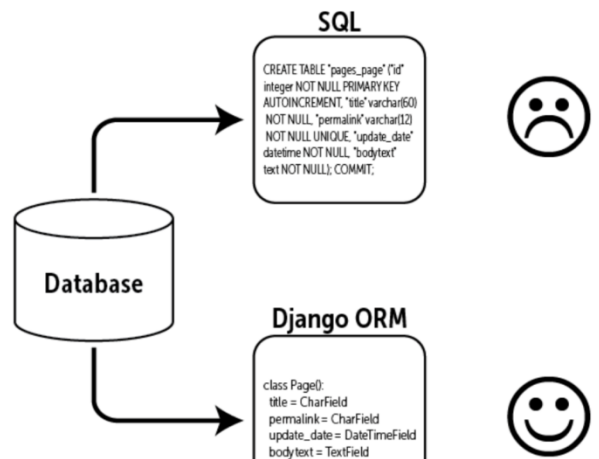
- oop 프로그래밍에서 RDBMS 를 연동할 때, 데이터베이스와 OOP 프로그래밍 언어간의 호환되지 않는 데이터를 변환하는 프로그래밍 기법

#### 장점

- SQL 문을 몰라도 DB 연동이 가능하다.
- SQL 의 절차적인 접근이 아닌 객체 지향적인 접근으로 인해 생산성이 증가한다.

#### 단점

- ORM 만으로 완전한 서비스를 구현하는데에는 어렵다.



### CRUD

ite

#### Objects

- models.py에 작성한 클래스를 불러와서 사용할 때 DB와의 인터페이스 역할을 하는 manager.
- Objects manager 라고 부름
- python class(python) .....Objects(인터페이스 역할).....DB(SQL)

### READ

```
# 모든 객체 조회  
Article.objects.all()  
  
# 특정 객체 조회  
Article.objects.get(pk=1)  
  
# 특정 조건 객체 가져오기  
Article.objects.filter(title='first')  
Article.objects.filter(title='first', content='djago!')  
  
# 내림 차순  
Article.objects.order_by('-pk')  
  
# LIKE  
Article.objects.filter(title__contain='fi')  
Article.objects.filter(title__startswith='fi')  
Article.objects.filter(content__endswith='!')
```

#### QuerySet

- Object 매니저를 사용하여 복수의 데이터를 가져오는 함수를 사용할 때 반환되는 객체 타입
- 단일 객체는 Query(class 의 인스턴스로 반환)
- query(질문)을 DB에게 보내서 글을 조회하거나 생성, 수정, 삭제.
- query 를 보내는 언어를 활용해서 DB에게 데이터에 대한 조작을 실행.

## CREATE

```
#1
article = Article()
article.title = 'first' # 인스턴스에 값을 넣어준다
article.content = 'django!!'
article.save()

#2
article = Article(title='second', content='django!!')
article.save()

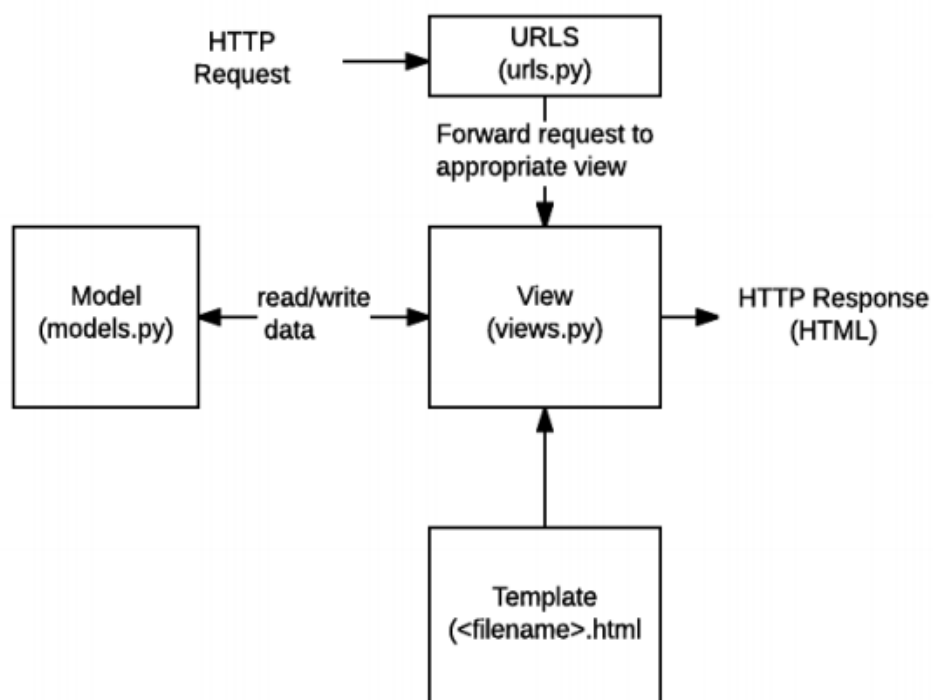
#3
Article.objects.create(title='third', content='django!!')
#save 안해도 등록 됨
```

## Update

```
article = Article.objects.get(pk=1)
article.title = 'edit title'
article.save()
```

## Delete

```
article = Article.objects.get(pk=1)
article.delete()
```



## [ 리스트 페이징 ]

Django에서는 리스트 페이징을 편하게 구현할 수 있게 Paginator 라는 클래스를 제공한다. 리스트 출력시에 페이징을 원하는 views에서는 데이터들을 추출하고 나서 Paginator 클래스의 객체를 생성한 다음 생성시 설정된 데이터 갯수로 구성된 Page 객체를 추출하여 템플릿에게 전달한다.

```
from django.core.paginator import Paginator
```

```
vlist = Visitor.objects.all()
```

```
paginator = Paginator(vlist, 3)
```

```
vlistpage = paginator.get_page(pagenum)
```

페이징 대상

한 페이지에 담길 글의 개수

Page 객체가 리턴됨

메소드	설명
count()	총 객체 수
num_pages()	총 페이지 수
page(n)	n 번째 페이지 반환
page_range()	(1부터 시작하는)페이지 리스트 반환
get_page()	n번 페이지 가져오기
has_next()	다음 페이지의 유무를 boolean 으로 반환
has_previous()	이전 페이지의 유무를 boolean 으로 반환
previous_page_number()	이전 페이지 번호 반환

- 페이징 관련 템플릿에서는 다음과 같이 Page 객체에서 제공하는 메서드를 사용해서 페이지 단위로 요청할 수 있는 링크를 출력한다.

<h5>

```
{% if vlist.has_previous %}
```

```
    <a href=?page={{vlist.number|add:-1}}>이전페이지</a>
```

```
{% endif %}
```

```
Page {{ vlist.number }} / {{ vlist.paginator.num_pages }}
```

```
{% if vlist.has_next %}
```

```
    <a href=?page={{vlist.number|add:+1}}>다음페이지</a>
```

```
{% endif %}
```

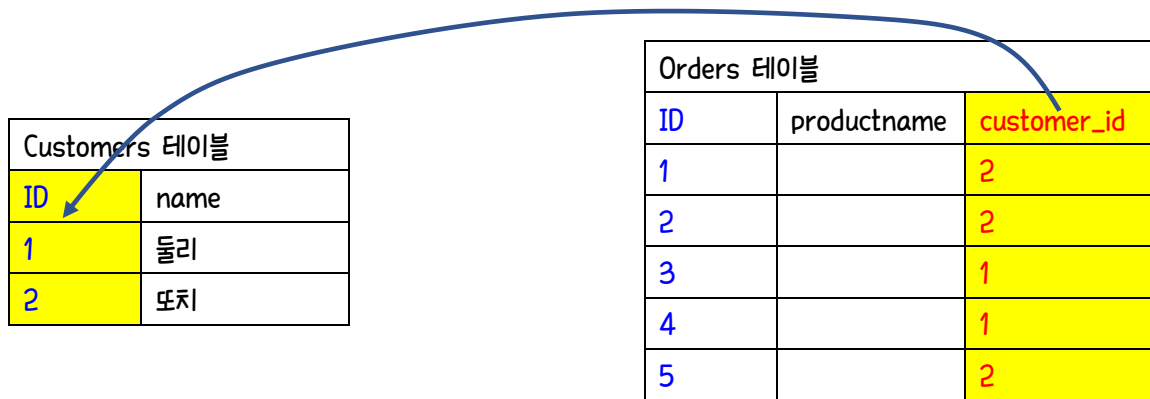
</h5>

## Django 의 Relationships

관계형 데이터베이스시스템의 핵심은 테이블 간의 관계 설정이라고 할 수 있다. Django 는 데이터베이스 관계의 가장 흔한 유형인 다대일 (Many-to-One), 다대다 (Many-to-Many) 그리고 일대일 (One-to-One) 관계를 구현할 수 있는 기능을 제공한다.

[ 다대일 관계 (Many-to-one relationships) ]

한 테이블에 있는 두 개 이상의 레코드가 다른 테이블에 있는 하나의 레코드를 참조할 때, 두 모델간의 관계를 다대일 관계라고 한다.



Orders 테이블의 customer\_id 필드는 Customers 테이블의 기본키 (Primary Key) 인 ID 필드를 참조하고 있다. 이 때, Orders 테이블의 Customer\_ID 필드를 외래키 (Foreign Key) 필드라고 한다.

### ForeignKey

Django 에서 다대일 관계를 설정할 때는 아래와 같이 ForeignKey 를 사용한다. 다른 필드 타입과 마찬가지로 모델 클래스의 속성으로 입력하며, 연결대상이 될 모델 객체를 위치인자로 전달해주어야 하고, on\_delete 옵션을 필수로 입력해주어야 한다.

```
class 모델이름(models.Model):
```

```
    필드이름 = models.ForeignKey(연결대상모델, on_delete=삭제옵션)
```

# Customers Table

```
class Customer(models.Model):
```

```
    name = models.CharField(max_length=50)
```

# Orders Table

```
class Order(models.Model):
```

```
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE) # 테이블 생성시 해당 컬럼엔 _id가 붙는다.
```

```
    product = models.CharField(max_length=50)
```

- ForeignKey 필드의 이름

ForeignKey 필드의 필드이름은 자유롭게 설정할 수도 있지만, 연결대상이 되는 모델 클래스명을 모두 소문자로 바꿔서 정하는 것을 권장한다.

```
customer = models.ForeignKey(Customer)
```

```
manufacturer = models.ForeignKey(Manufacturer)
```

- on\_delete 옵션

on\_delete 은 참조되는 레코드(부모 레코드)를 삭제할 때, 그 레코드를 참조하는 레코드(자식 레코드)들에 대한 행동을 정의한다.

**models.CASCADE**: 레코드가 삭제시 이 레코드를 외래키로 참조하고 있는 모든 레코드들을 함께 삭제한다.

**models.PROTECT**: 외래키가 참조하고 있는 레코드를 삭제하지 못하게 만든다. 삭제를 시도하면 ProtectedError 를 발생시킨다.

**models.SET\_NULL**: 외래키가 참조하고 있는 레코드가 삭제되면, 외래키 필드의 값이 null 이 된다.

외래키 필드에 null=True 옵션이 있을 때만 가능함.

**models.SET\_DEFAULT**: 외래키가 참조하고 있는 레코드가 삭제되면, 외래키 필드의 값이 기본값으로 바뀐다.

default 옵션이 설정되어 있을 때만 가능함.

**models.DO\_NOTHING**: 아무 작업도 하지 않음

**models.SET(값 또는 함수)**: SET() 함수에 값이나 호출가능한 객체를 전달할 수 있으며, 외래키가 참조하고 있는 레코드가 삭제되면 전달된 값 또는 객체를 호출한 결과로 외래키 필드를 채운다.

#### 다대일 관계의 참조와 역참조

위의 Orders, Customers 테이블[테이블 보기] 을 예로 들면, Customer 모델은 Order 모델의 타겟모델이다.  
소스모델의 인스턴스에서 타겟모델의 인스턴스를 가져오려면 아래와 같이 관계가 정의된 속성의 이름을 붙여준다.

```
o = Order.objects.get(id=1)
```

**o.customer** # 1번 주문을 한 고객을 가져온다. → 2번

여기서 반대로 타겟 인스턴스에서 소스 인스턴스를 역참조하려면 아래와 같이 한다.

```
c = Customer.objects.get(id=2)
```

**c.order\_set.all()** # 소스모델의 이름은 Order 이므로, 역참조 매니저의 이름은 order\_set 이 된다.

이것의 결과로 1번 고객에 연결된 모든 Order 모델의 인스턴스들이 쿼리셋으로 리턴된다.

#### 참조-역참조 요약

	ManyToOneField	OneToOneField
참조(소스->타겟)	<b>source.attrname</b>	source.attrname
역참조(타겟->소스)	<b>target.lower sourcename _set</b>	target.lower source

```
class Visitor(models.Model):
```

```
    name = models.CharField(max_length=6)
```

```
    memo = models.TextField()
```

```
    writedate = models.DateTimeField(auto_now_add=True)
```

```
class Reply(models.Model):
```

```
    content = models.CharField(max_length=80)
```

```
    visitor = models.ForeignKey(Visitor, on_delete=models.CASCADE)
```

타겟 모델 객체

소스 모델 객체