**1. 데이터베이스와 ORM**

SQL: 데이터베이스에 쿼리하기 위한 언어

django에서는 ORM(object-relational mapping) 을 통해 sql을 실행/생성

장고의 강점은 model 과 form. 가능하면 ORM을 쓸 것. (SQL Injection 방지)

장고에서는 기본키로 id 필드를 디폴트로 생성

다른필드를 지정하고 싶다면 primary\_key =True 적용

**2. Django modeling**

1) 모델링 방법

모델 클래스 작성 -> $ make migrations: 마이그레이션 파일 생성 ->

$ migrate: 마이그레이션 파일 DB 적용 ->

$ python manage.py sqlmigrate [app이름] [migration 이름]: sql 확인

class Post(models.Model):

    title = models.TextField()

    content =  models.TextField()

    created\_at = models.DateTimeField(auto\_now\_add=True)

    updated\_at = models.DateTimeField(auto\_now=True)

2) 모델링 유의사항

모델 클래스명은 단수형으로 지정 (Posts가 아닌 Post)

DB 테이블명: 앱이름\_모델명 (ex: blog\_comment, blog\_comment…)

매핑되는 모델은 DB 테이블과 일치해야 함

데이터베이스를 먼저 설계하고 모델을 생성할 것

$ inspectdb : 외부 db로부터 가져올 때 최종본이 아닌 초안으로 사용할 것

설계한 데이터베이스 구조에 따라 필드 타입을 타이트하게 지정해주는 것을 권장함

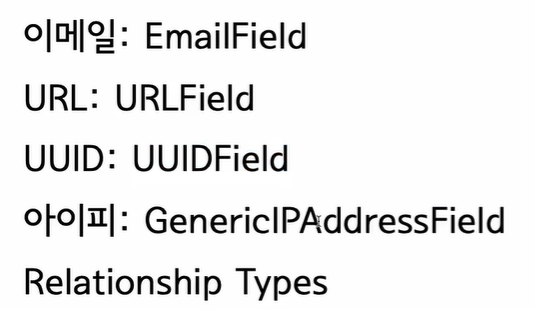
inspectdb 명령은 초안으로 사용하고 validators를 통해 타이트하게 지정할 것

blank, null 지정은 최소화 할 것

직접 유효성 검사 로직을 만들지 말고 기존의 구성된 것을 사용할 것

3) 모델 필드 타입 및 옵션





blank: validation 시 empty 허용

null: null 허용 여부

default: 기본값 지정

validators: 필드에 따른 고유한 validators 등록 (ex: 이메일)

**3. 모델 조회(select)**

1) 전체 조회: qs = Post.objects.all()

2) 범위 지정 및 아이디 순으로 정렬 후 범위 처리: qs = Post.objects.all().order\_by('-id')[:2]

3) 결과 딕셔너리 출력:

for post in qs:

print(post.message)

print(post.\_\_dict\_\_)

for post in qs:

print ("id: {id}, message: {message} (작성일자: {created\_at})".format(\*\*post.\_\_dict\_\_))

4) 조건문 (filter, exclude)

filter안에 들어가는 다양한 오퍼레이터를 알아두는 것이 좋다

두 개 이상의 필터는 and 조건으로 적용됨

qs = Post.objects.filter(message\_\_icontains = '테스트').order\_by('-id')

5) 조건문 (get, first, last)

post.get : 값이 없을 경우 예외 반환

(값이 없다면DoesNotExist / 두 개 이상의 값이 나오면 MultipleObjectsReturned 반환)

post.fitst : 모델객체 없으면 None

post.last : 모델객체 없으면 None

6) Q: 필터 조건을 or 로 주고 싶을 때 사용

from django.db.models import Q

# and 조건

cond = Q(id\_\_gte=2)&Q(message\_\_icontains="테스트")

# or 조건

cond = Q(id\_\_gte=2)|Q(message\_\_icontains="테스트")

qs = Post.objects.filter(cond)

# 결과

for post in qs:

print(post.\_\_dict\_\_)

7) 다양한 오퍼레이터

필드명\_\_lt : 필드명 < 조건

필드명\_\_lte: 필드명 <= 조건

필드명\_\_gt: 필드명>조건

필드명\_\_gte: 필드명>=조건

필드명\_\_startswith : 필드명 like "조건값%"

필드명\_\_istartswith : 필드명 like "조건값%" (대소문자 구별 x)

기타 endwith, contains.**4. Foreign key**

1) 1:N models.ForeignKey

ex) 1명의 유저가 쓰는 다수의 포스팅, 1명의 유저가 쓰는 다수의 댓글

- N에서 관계 명시

- models.ForeignKey(대상모델, on\_delete)

- on\_delete:

cascade: 참조 하는 모델의 리코드도 삭제

그 밖에 PROTECT, SET\_NULL, SET\_DEFAULT, SET, DO\_NOTHING 등이 있음

다만 reverse\_name의 경우 앱이름은 고려하지 않고 모델명만 고려하기 때문에

이름 충돌 발생할 수 있음.

이 때 related\_name을 통해 한쪽의 reverse\_name을 포기하거나 reversename 변경하는것을 권장

2) 1:1 models.OneToOneField

ex) 1명의 유저에는 1개의 프로필

- models.ForeignKey(대상모델, on\_delete)

3) M:N models.ManyToManyField

ex) 1개의 게시글 다수의 태그, 1개의 태그에는 다수의 게시글

- models.ManyToManyField(대상모델, blank 여부)

- M,N 어느쪽에서 선언할 수 있으나 사용하는 쪽에서 선언하는 것, 또는 참조가 덜 되는 쪽에서 선언하는 것이 적합하다.

ex) 태그의 경우 Post, Article 등 다양한 곳에서 사용되므로 Post, Article 에서 선언하는 것이 좋음.

tag\_set = models. ManyToManyField(‘Tag’, blank= True)

manytomany 관계의 경우 테이블 두개로는 정의하기 힘들기 때문에 중간 테이블이 생성된다.

**5. migration을 통한 DB 스키마 관리**

순서: 모델변경 > makemigrations > 마이그레이션 파일 확인 > migrate > DB 적용

모델 필드에서 변경 내역이 발생 시에 수행

적용된 마이그레이션 파일은 삭제 금지. 히스토리를 쌓아가는 원리이므로

Django 를 통해서 데이터베이스 스키마의 히스토리 관리

[앱이름]을 생략하면 전체를 대상으로 진행됨. 그러나 [앱이름] 지정을 권고

같은 Migration 파일이라 하더라도 DB에 따라 다른 SQL이 생성될수도

$ python manage.py makemigrations [앱이름] : 마이그레이션 파일 생성

$ python manage.py migrate [앱이름] : 지정 DB에 마이그레이션 적용

$ python manage.py showmirgrations [앱이름] : 마이그레이션 적용 현황 출력. 빈칸은 적용안된 것

$ python manage.py sqlmigrate [앱이름]: 지정 마이그레이션 sql 출력

필수 필드가 추가된다면 마이그레이션을 하면 기존 레코드에 어떤값을 채워넣을지 물음

(default값을 지정하는게 가장 낫긴 함)

- 1. 값을 입력 2. 명령수행중단

팀원 각자가 마이그레이션 파일 생성시 충돌 발생 위험!

한명이 전담해서 생성하는 것을 권장.

생성한 파일을 버전 관리에 넣고 다른 팀원들은 migrate만 수행

**6. User 객체**

import get\_user\_model

User = get\_user\_model() : 현재 존재하는 모든 유저 가져오기

- 1:1 관계인 profile 가져오기

user.profile:

Profile.objects.get(user=user)