장고로 백엔드만들기

이제까지 대략 맛을 본 바로는 처음에는 새로 배우고 익숙해져야 할 것들이 꽤 있지만, 일단 익숙해지면 높은 생산성(물론 비례해서 커스터마이징은 덜 다양해지겠지만)으로 웹서버 생산이 가능한 프레임워크인 것으로 보인다. 장고의 장점은 Less Time, Less code를 실현하는 것이다. 소위 Batteries included 프레임워크라 불린다.

커스터마이징은 기본 골격을 익힌 다음에 하는 것이고, 기본 골격을 익히는데 좋은 프레임워크인 것 같다. 열심히 해 보자.

장고는 오래된 프레임워크이고, 최신 프레임워크에 비해 Performance가 떨어진다고 할 수도 있으나, 성숙성, 안정성, 학습의용이성, 커뮤니티 등 요소를 고려하면 여전히 훌륭한 프레임워크이다.

장고란 : Free and open-source framework for building web apps with python. 파이썬 웹앱 프레임워크로는 장고이외에 대표적으로 Flask가 있고, 그 외에, Tornado, Bottle, Falcon, Hug 등도 있다.

클라이언트-서버 아키텍처의 웹서비스에서 과거에는 서버측에서 웹페이지 문서를 렌더링하여 클라이언트에 보내주면, 브라우저가 이를 재생하였으나, 서버 및 네트워크의 부하를 줄일 수 있도록, 서버에서는 클라이언트가 필요로 하는 데이터만을 전송하고, 클라이언트가 이를 받아 클라이언트 측에서 웹페이지 문서를 렌더링하는 방식이 최근의 대세가 되었다. 브라우저 기술의 발달도 이를 뒷받침하였다. (다만, 클라이언트 사이드의 프로그램도 결국에는 처음부터 로컬에 존재하는 것이 아니라, 서버에서 다운받아 가야 하는 것이다. 서버에서 다운받은 프로그램을 클라이언트(브라우저)에서 구동하면서, 서버의 엔드포인트에서 받은 데이터들을 기반으로 클라이언트에서 사용자가 필요로하는 웹페이지를 렌더링하게 된다).

장고 개발환경 셋업

1. 파이썬 설치
2. Pipenv 설치(가상환경을 사용하여, dependency를 관리해 주는 툴)

pip install pipenv

1. Visual studio code 설치
2. Python 익스텐션 설치

장고 프로젝트 만들기

장고 프로젝트는 여러개의 앱으로 구성된다(개념이 그래)

1. 장고 프로젝트를 작성할 폴더를 만들고 이동한다.

mkdir 폴더이름

cd 폴더이름

1. 장고프레임워크를 설치한다(가상환경에 설치)

pipenv install django

이번에 설치한 장고는 C:\Users\coms\.virtualenvs\django\_learn-BcwC832p 폴더에 설치되었다. 해당 가상환경 폴더에 가보면, scripts폴더, lib폴더, pyvenv.cfg 컨피규레이션파일 및 src폴더 등이 있다.

**pipenv --venv 명령어를 사용하면 가상환경 폴더의 위치를 알 수 있다.**

가상환경 터미널 활성화 pipenv shell, 비활성화 exit

가상환경 명령어 실행 pipenv run

Pipfile(1)과 Pipfile.lock(2)이 생성된다.(장고를 설치해서 생성되는 건 아니고, 패키지가 설치되었으므로 생성되는 것이다. pip는 package installer for python의 약자이다). 두파일 다 패키지관리 정보인데, Pipfile은 설치된 패키지에 대한 개략적인 정보(소스, 패키지이름(하위패디펜던시 불포함), 대략적인 버전 등)가 포함되고, Pipfile.lock에는 설치된 모든(?) 패키지에 대한 정확한 버전정보가 포함된다.

1. 새로운 장고프로젝트를 시작한다.

pipenv shell 가상환경 터미널 활성화, 비활성화 exit

**django-admin startproject project\_name** **.** (마지막 .은 현재폴더를 의미, .를 사용하지 않으면 하위 폴더에 프로젝트 생성)

새로운 장고프로젝트를 시작하면, 기본적으로 프로젝트이름과 동일한 이름을 갖는 장고앱이(폴더) 생성되며, **dajngo-admin의 래퍼인 manage.py**(3)파일이 생성된다. 장고앱폴더에는 해당 폴더를 패키지로 취급하기 위한 \_\_init\_\_.pyt파일(4), 장고 프로젝트의 환경구성 파일인 settings.py(5), 장고로만든 웹어플리케이션의 라우터 역할을 하는 urls.py(6), 웹서버와 장고로 만든 웹어플리케이션의 중계역할(Tomcat 등 웹어플리케이션서버가 하는 역할을 하는 미들웨어, CGI(Common Gateway Interface)의 역할)을 하는 WSGI(Web Server Gateway Interface)관련파일인 wsgi.py(7), 비동기처리를 하는 WSGI인 ASGI(Asynchronous Server Gateway Interface)관련파일인 asgi.py(8)가 생성된다.

* 장고 서버 실행하는 법 : **python manage.py runserver port[디폴트값은 8000]**
* 서버를 실행하면, db.sqlite3 파일이 생성된다.
* 장고프로젝트에 기본으로 설치되는 앱 django.contrib 패키지에 존재함
  + admin(어드민 인터페이스 제공)
  + auth(인증, 인가 제공)
  + contenttypes(나중에 설명)
  + sessions(세션 - 유저 정보를 관리하는 서버의 임시 메모리(temporary memory on the server for managing user's data, 현재는 레거시라고 함, 지워도 된다고 했으나, 나중에 보니, admin앱에서 사용하기 때문에 지우면 안 됨)
  + messages(메세지 정보 제공)
  + staticfiles(정적파일 관련)

1. 앱을 생성한다.

**python manage.py startapp app\_name**

새로운 앱을 생성하면 앱네임을 가진 폴더가 생성되고, 6개의 파일과, 1개의 하위폴더가 생성된다. 6개의 파일은 폴더를 패키지로 취급하기 위한 \_\_init\_\_.py(1), 설정파일(settings.py)과 같은 기능을 하는 apps.py(2), 해당 앱에 대한 admin기능을 제공하는 admin.py(3), 테스트를 위한 tests.py(4), 그리고 models.py(5)와 views.py(6)이다. **장고프로젝트는 많이 알려진 MVC(model, view, controller)아키텍쳐가 아닌 MTV(model, template, view)아키텍쳐를 사용한다.** 즉, 장고프로젝트에서의 view는 MVC아키텍쳐의 콘트롤러(리퀘스트 핸들러)에 해당한다. 1개의 하위폴더는 migrations폴더이며 이는 DB를 생성하기 위한 것이다. migration폴더의 하위에는 \_\_init\_\_.py파일이 있다.

1. 생성한 앱을 셋팅파일(settings.py)에 등록

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    # 'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'playground'

]

이 단계까지는 장고프로젝트가 만든 기본 페이지가 동작한다.

1. 뷰(리퀘스트 핸들러)를 작성한다. 다른 아키텍쳐에서는 컨트롤러 또는 액션이라고도 한다.

뷰(리퀘스트 핸들러)는 **views.py**에 작성한다. 뷰펑션(**리퀘스트 핸들러 펑션**(리퀘스트를 매개변수로 받아서, 여러가지 형태의 리스폰스를 반환하는 함수))를 작성한다.

from django.http import HttpResponse

def say\_hello(request):

    return HttpResponse('Hello World')

1. 라우팅 정보를 작성한다.

라우팅 정보는 장고앱의 **urls.py의 urlpatterns변수에, urlpattern 오브젝트의 리스트 형식으로 저장**하는데, urlpatterhn 오브젝트를 반환하는 **path 펑션을 이용**하여 작성한다. path 펑션은 path(리퀘스트 path, 뷰펑션(리퀘스트핸들러 펑션))의 형식으로 사용한다.

from django.contrib import admin

from django.urls import path

from playground.views import say\_hello

urlpatterns = [

    path('admin/', admin.site.urls),

    path('say\_hello/', say\_hello)

]

urls.py의 urlpattern에 사용자가 사용자지정 라우트를 등록하는 순간 장고 기본앱의 디폴트 홈페이지는 더 이상 작동하지 않는다.

뷰펑션이 어느 앱의 소속이든지, 단계가 없는 라우팅이라면, 해당 뷰 펑션만 임포트에서 사용하면 된다. 임포트 방법은 임의로 택할 수 있다.

from playground.views import say\_hello

urlpatterns = [

    path('say\_hello/', say\_hello),

]

와 같이 해도 되고,

from playground import views

urlpatterns = [

    path('say\_hello/', views.say\_hello),

]

와 같이 해도 된다.

라우팅을 단계화하기 위해서는 path 펑션의 두번째 인자로, 뷰펑션을 사용하는 대신, **include 펑션**을 사용한다. include 펑션은 장고 기본앱의 urlpattern에 다른 앱의 urlpattern들을 포함시키는 기능을 한다. include 펑션의 반환값을 path 펑션의 매개변수로 사용하는 형식으로 사용되며, **include 펑션의 매개변수로는 다른 앱의 urlpattern이 포함된 파일의 경로를 사**용하고, path 펑션에서 1단계 레벨링을 해 준다.

앱은 자신만의 URL Configuration(URLconf)을 가질 수 있다.

from django.urls import path, include

urlpatterns = [

    path('playground/', include('playground.urls')),

]

기본앱의 urls.py

playground 라우트로 리퀘스트가 들어오면, playground 앱의 urls.py파일을 참고해서 다음 단계 라우팅을 진행하라는 의미, **include 펑션의 매개변수로는 스트링을 사용**한다는 점에 주의!

from django.urls import path

from . import views

urlpatterns = [

    path('say\_hello/', views.say\_hello)

]

다른앱의 urls.py (playground.urls)

**예에서 보는 것과 같이, 패스 뒤에 /를 붙여야 한다. 패스 앞에는 붙이지 않는다**.

1. 템플릿 작성하는 법

템플릿은 render 펑션을 사용하여 작성하며, **앱의 하위 폴더로 templates폴더(이름 중요)를 만들고, 해당 폴더에 템플릿을 작성**하면 된다.

from django.shortcuts import render

def say\_hello2(request):

    return render(request, 'hello.html')

템플릿에서 다이나믹한 밸류를 표시하기 위해서는 **render 펑션의 3번째 매개변수로 key, value 값의 딕셔너리를 전달**하여야 한다.

def say\_hello2(request):

    return render(request, 'hello.html', {'name': 'Harry'})

<h1>hello {{ name }}</h1>

그리고 나서, 템플릿 파일 내에 더블 { }를 이용하여 해당변수를 html코드 내에 넣는다.

템플릿 내에서는 다음과 같은 프로그래밍도 할 수 있다.

{% if name %}

    <h1>hello {{ name }}</h1>

    {% else %}

    <h1>hello World</h1>

    {% endif %}

최근에는 장고로 템플릿을 사용하는 웹사이트를 많이 만들지는 않고, REST API를 만드는데 많이 쓴다.

이커머스 애플리케이션을 만들기 위한 데이터 모델링

Product : title, description, price, inventory

Collection(카테고리) : title

Product와 Collection의 관계를 many to one이라고 하자. Product는 하나의 Collection에만 속하지만, Collection은 여러 개의 Product를 가질 수 있다. many to one 관계에서 one은 many의 외래키가 되고, one은 이에 대응하는 many(products) 속성을 가질 수 있다. 즉, Product는 collection\_id를 속성으로 갖고, Collection은 products를 속성으로 갖는다.

Product와 Collection의 관계를 many to many라고 하면, 하나의 Product가 여러 개의 Collection에 소속될 수도 있다는 의미이다. 이 경우 별도의 테이블을 만들어야 하는데, 2개의 외래키를 복합키인 기본키로 사용할 수도 있고, 2개의 외래키와 별도로 기본키를 만들어 사용할 수도 있다.(장고에서 전자의 방식은 2개의 외래키필드로 처리하는 것 같고, 후자의 방식은 매니투매니필드로 처리하는 것 같다(확실하지 않음, 오해였음, 아님, 복합키가 기본키인 것으로 정의하지 않는한, 두 개의 Foreign키 필드로 클래스를 정의하여도 자동으로 기본키가 되지는 않음))

엔티티들은 하나 이상의 관계를 가질 수도 있다.

featured product(추천상품) 추천상품과 컬렉션인 1대 0 또는 1의 관계를 갖는다고 하자. 컬렉션에는 추천상품이 없을 수도 있고, 하나 있을 수도 있다. 이 경우에 featured product가 collection의 페어런트가 되는 관계가 발생하는 것 같다(collection이 product\_id를 참조하여야 하므로. 이 관계가 순환관계인지는 확실하지 않음)

Cart : created\_at

Product와 Cart의 관계는 many to many이다. 하나의 Product는 여러 개의 다른Cart에 속하는 것이 가능하며, 하나의 Cart에는 여러 개의 다른 Product를 담는 것이 가능하다.

엔티티간의 관계가 독자적인 속성을 갖는 경우도 있다. Cart와 Product의 관계는 Quantity라는 속성을 갖는다. 여기에서 CartItem이라는 엔티티를 추출할 수 있다. 이런 Entity를 Association class라고 한다.

이러한 경우에는 Product와 CartItem(one to many), CartIem과 Cart(many to one)의 관계를 설정할 수 있다. Association Class를 갖는 many to many 관계로 모델링할 것인가, 2개의 one to many관계로 모델링할 것인가는 선택의 문제이다.(확실하지 않다)

Customer : name, email

Order: placed\_at

Customer와 Order의 관계는 one to many관계이다. Order와 Product의 관계는 Cart와 Product의 관계와 마찬가지로 many to many의 관계이다. Order와 Product 사이에도 OrderItem이라는 Association 클래스가 있게 된다(qauntity 속성을 갖는다). Order와 OrderItem의 관계, Product와 OrderItem의 관계는 one to many이다.

Tag : label

Product와 Tag는 many to many의 관계를 갖는다. Product와 TaggedItem의 관계는 one to many, Tag와 TaggedItem의 관계도 one to many이다. TaggedItem은 예를 들면, 전자제품이라고 태깅된 컴퓨터, 식품이라고 태깅된 소고기 등이다.

모쉬의 데이터 모델 디자인

STORE : Product, Collection, Customer, Cart, CartItem, Order, OrderItem.

TAGS : Tag, TaggedItem.

각각의 장고 앱은 각자의 데이터 모델을 갖는다.

Always design your models based on the requirements of your project.

**Each app should do one thing and do it well.**

모델 만들기

앱의 파일 중 models.py에 데이터 모델을 만든다. **장고의 models 패키지를 사용**한다.

from django.db import models

class Product(models.Model):

  title = models.CharField(max\_length=255)

  description = models.TextField()

  price = models.DecimalField(max\_digits=6, decimal\_places=2)

  inventory = models.IntegerField()

  last\_update = models.DateTimeField(auto\_now=True)

**필드타입 선택은 django Model field reference 문서 참조**

class Product(models.Model): 은 Model 클래스를 상속한다는 의미

CharField에서 max\_length는 필수 인자이다.

DecimalField에서 max\_digits, decimal\_places는 필수 인자이다.(FloatField는 정확성 이슈가 있으므로, 화폐단위를 사용할 때는 항상 DecimalField를 사용해야 한다).

DateTimeField의 auto\_now 매개변수는 생성 또는 변경시의 현재 시간을 기록하라는 것이고, auto\_now\_add 매개변수는 생성시에만 현재 시간을 기록하라는 것이다.

이외 매개변수로 unique=True, null=True, default=value 등이 있다.

아이디는 자동생성되지만, 아이디 이외의 기본키를 사용할 수도 있다.

guid = models.CharField(max\_length=10, primary\_key=True)

이와 같이 생성된 모델들을 기반하고 이에 의거하여, 데이터베이스 마이그레이션(데이터베이스 구축을 위하여 장고가 생성하는 코드)이 생성되고, 이를 실행하여 앞서 생성된 모델의 구조를 반영하는 실제 물리적 데이터베이스(테이블과 릴레이션으로 이루어진)를 구축한다.

입력값이 몇 개의 선택지 중에 정해져 있는 필드(Enum)는 choices 매개변수를 이용하여, 다음과 같이 정의한다(ChoiceField라는 것이 있는 것이 아니라, 필드의 데이터 타입을 지정한 다음, 해당 데이터값으로 선택이 가능한 선택지들을 choices 매개변수를 이용하여 전달한다). choices 매개변수에 넘겨줄 값은 터플(tuple)의 리스트인데, 각 터플은 (입력될 데이터 값, 해당 데이터 값의 의미(Human readable name))의 쌍으로 구성된다. 변수이름에는 CHOICES를 포함시키는 것이 좋은 것 같다. 상수를 설정한 것은 일반적인 코딩 테크닉(변해서는 안되는 값이라는 것을 암시, 중복 제거)이다.

    MEMBERSHIP\_BRONZE = 'B'

    MEMBERSHIP\_SILVER = 'S'

    MEMBERSHIP\_GOLD = 'G'

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, 'Bronze')

        (MEMBERSHIP\_SILVER, 'Silver')

        (MEMBERSHIP\_GOLD, 'Gold'),

    ]

    membership = models.CharField(

        max\_length=1,

        choices=MEMBERSHIP\_CHOICES,

        default=MEMBERSHIP\_BRONZE)

엔티티간 릴레이션(레코드에 삽입되는 데이터 값이 밸류가 아닌, 다른 테이블의 레코드를 참조하는 값인 경우) 정의하기

1. one to one 릴레이션(Customer와 Address의 관계를 1:1 관계라고 하자)

데이터베이스에서 페어런트와 차일드의 관계(차일드가 저장되기 전에 페어런트가 있어야 한다) 차일드 모델에서 페어런트를 속성으로 정의한다.

class Address(models.Model):

    street = models.CharField(max\_length=255)

    city = models.CharField(max\_length=255)

    customer = models.OneToOneField(

        Customer,

        on\_delete=models.CASCADE,

        primary\_key=True)

on\_delete 매개변수는 customer가 지워졌을 때, address를 어떻게 처리해야 하는지의 문제이다(address가 지워졌을 때, customer를 어떻게 처리해야 하는지의 문제는?).

삭제처리 선택옵션은 models 패키지에 상수로 정해져 있으며, 상수 값으로는 CASCADE(해당 customer를 가지고 있는 address를 지운다), SET\_NULL(해당 address 레코드의 customer 값을 null로 설정한다), SET\_DEFAULT(해당 address 레코드의 customer 값을 디폴트 값으로 설정한다), PROTECT(address가 있는 이상, address가 참조하는 customer를 삭제할 수 없다) 등이 있다.

one to one 릴레이션의 경우에는 참조값을 primary key로 사용할 수 있다는 특성이 있다(사실, 사용해야 한다).

차일드 모델에서 페어런트를 정의한 경우, 페어런트 모델에서 차일드를 정의할 필요 없다. 모쉬는 그렇다고 하는데 맞는가? 따로 정의하면 다른 효과가 생기지 않는가? 일단 따로 정의하지 않으면, address 테이블에는 customer\_id 컬럼이 생기지만, customer 테이블에는 address 관련한 컬럼이 생기지 않는다. 그렇지만 아래에서 보는 것처럼 관련성이 생기기 때문에 address를 이용한 테이블 참조, address\_\_ 를 이용한 컬럼참조가 가능하다.

커스터머 클래스에서 어드레스는 어떻게 참조되는가. 크로스테이블 참조를 하면 INNER JOIN으로 WHERE절에서의 참조가 가능하다.

query\_set = Customer.objects.filter(address\_\_city='daejon')

컬럼 조회 참조를 하려면 select\_related를 하면 된다.

query\_set = Customer.objects.select\_related('address').all()

이 경우 address 값이 없는 경우도 가져와야 하므로, LEFT OUTER JOIN이 일어난다.

One to many관계의 컬럼 조회 참조를 하려면 prefetch\_related를 하면 된다.

query\_set = Customer.objects.prefetch\_related('order\_set').all()

이 경우에는 장고가 2개의 SQL쿼리를 실행하여, customer테이블의 내용과, customer\_id를 조건으로 order테이블의 내용을 조회하여 온다.

Customer – Address 일대일 관계, Customer가 페어런트, Address가 차일드

* + - Address 클래스에서 OneToOneFiled로 customer 컬럼 추가 및 기본키 설정
    - 실제 생성되는 테이블에는 address 테이블에 customer\_id가 컬럼으로 생성
    - customer 테이블에는 address 관련 컬럼 존재하지 않음
    - address 테이블은 customer\_id를 컬럼으로 가지고 있으며, 두 테이블은 관련 테이블임
    - customer\_id를 이용한 조회의 경우에는 customer\_id 또는 customer\_\_id를 이용하여 조회를 하면 되고, customer 테이블의 다른 컬럼을 통한 조회도 customer\_\_ 를 사용하여 조회 가능
    - select\_related를 사용하여 customer 테이블의 컬럼을 모두 조회해올 수 있음
    - customer를 주로 하는 쿼리의 경우에도, address\_\_를 이용하여 address 테이블의 모든 컬럼을 이용한 조회를 할 수 있고, select\_related를 사용하여 address 테이블의 모든 컬럼을 조회해 올 수 있음
    - 결국, address 테이블에는 customer\_id가 있고, customer 테이블에는 address 관련 컬럼이 없다는 차이는 있지만, 장고 내부적으로는 address의 경우 customer\_\_id가 아닌 customer\_id를 추가로 사용할 수 있다는 것 외에, customer입장에서 조회할 때나, address입장에서 조회할 때의 차이는 없음

Customer – Order 일대다 관계

* + - one to many관계이고, customer가 페어런트인 것 같음
    - 차일드인 Order 클래스에서 Customer 클래스를 포린키필드로 지정(on\_delete옵션도 설정)
    - order 테이블은 customer\_id 컬럼을 갖게 되지만, Customer 테이블에는 order 관련 컬럼이 존재하지 않음
    - 그렇지만, 두 테이블은 customer\_id라는 컬럼을 이용하여 조인이 가능하기 때문에, Customer입장에서 조회할 때는 order\_\_ 를 이용하여, Order입장에서 조회할 때는 customer\_\_ 를 이용하여 상대테이블의 컬럼을 이용한 조회 가능
    - Order 입장에서 Customer 테이블의 컬럼을 읽어올 때는 select\_related를 사용할 수 있음
    - 그러나 Customer 입장에서 Order 테이블의 컬럼을 읽어올 때는 prefetch\_related(‘order\_set’)을 사용하여야 하고, 이 때는 2개의 쿼리가 실행됨

1. one to many 릴레이션

class Order(models.Model):

    customer = models.ForeignKey(

        Customer,

        on\_delete=models.CASCADE)

many가 되는 차일드 모델에서 one이 되는 페어런트 모델을 포린키 필드로 사용한다(order테이블에 customer\_id가 포린키 컬럼으로 들어간다).

차일드 모델이 코드의 배열상 위에 있어서 페어런트 모델을 참조할 수 없을 때에는 'Customer'와 같이 스트링을 매개변수로 사용하면, 장고가 알아서 처리해 준다.

Customer는 order\_set 속성을 갖게 된다. related\_name 매개변수를 이용해서 order\_set이라는 이름을 다른 것으로 지정해 줄 수 있다.

1. many to many 릴레이션

Product와 Promotion의 관레를 many to many 관계라고 하자. 하나의 product를 대상으로 하는 여러 개의 promotion이 있을 수 있고, 하나의 promotion이 여러 개의 product를 대상으로 할 수 있다.

매니 투 매니 릴레이션의 경우에도 둘 중 한쪽에서만 관계를 정의하면 된다.

class Product(models.Model):

    promotions = models.ManyToManyField(

        Promotion, related\_name='products')

related\_name을 따로 정의해 주지 않으면, Promotion은 product\_set이라는 속성을 갖게 된다.

이와 같이 Many to Many 필드로 정한 경우에도 장고 마이그레이션은 자동적으로 product\_promotion이라는 관계 테이블을 생성한다.

이 경우에도 관계가 존재하게 되므로, promotions\_\_ 를 이용한 promotion 테이블의 컬럼 조회가 가능하다. 또한 prefetch\_related(‘promotions’)를 이용하여 promotion 테이블의 내용을 조회해 올 수도 있다.

class Collection(models.Model):

    title = models.CharField(max\_length=255)

    featured\_product = models.ForeignKey(

        'Product',

        on\_delete=models.SET\_NULL,

        null=True,

        related\_name='+')

이러한 순환의존관계에서는 포린키필드의 매개변수로 Product를 사용할 수 없으므로, ‘Product’를 사용한다. 또한 순환의존관계에서는 name clash에러가 발생하는데, 이는 이미 Product클래스에서 collection 속성이 정해져 있기 때문에(그에 대응하여 Collection 클래스에도 production\_set 속성이 정해져 있다), Product 클래스에 featured\_product에 대응하는 collection 속성을 다시 만들 수 없기 때문이다(근데, Product 클래스에 collection\_set 속성을 만들면 되는 것 아닌가? 이러면 이름 충돌이 없는데?? A 클래스에 B 클래스가 포린키로 최초 설정되면, B 클래스에는 A\_set이라는 속성이 자동으로 만들어지는데, 이미 이러한 관계가 설정된 A와 B클래스에 대하여 B클래스에 A클래스를 포린키로 설정하면, A클래스에는 B\_set이란 속성이 자동으로 만들어지지 않고, name clash를 야기하는 B란 속성이 자동으로 만들어진다는 이야기인가?). 이 경우에는 related\_name을 이용하여 다른 이름을 지정해 주거나, realted\_name=’+’라는 특별 매개변수를 사용하여 대응속성을 만들지 않도록 하면 된다(대응속성을 만들지 않는 경우 실질적으로 어떠한 차이가 발생하는지도 궁금하다).

제너릭 릴레이션

특정의 모델(클래스)에 의존하지 않는 일반적인 관계를 만드는 방법이다.

contenttypes은 장고에 기본으로 인스톨되어 있는 앱들 중 하나이다.

from django.db import models

from django.contrib.contenttypes.models import ContentType

from django.contrib.contenttypes.fields import GenericForeignKey

# Create your models here.

class Tag(models.Model):

    label = models.CharField(max\_length=255)

# What tag applied to what object

class TaggedItem(models.Model):

    tag = models.ForeignKey(Tag, on\_delete=models.CASCADE)

    # Type of object - find table

content\_type = models.ForeignKey(ContentType, on\_delete=models.CASCADE)

    # ID of object - find record

    object\_id = models.PositiveIntegerField()

    content\_object = GenericForeignKey()

content\_object는 해당 태그가 적용된 오브젝트 자체를 의미한다. 왜 이를 쓰는지 그 함의와 실제 활용례는 잘 모르겠다. 해당 오브젝트 자체에 대한 쿼리를 얻기 위하여 사용한다고 한다. 데이터베이스 테이블에는 해당 내용이 컬럼으로 반영되어 있지는 않다.

마이그레이션

장고가 지원하는 데이터베이스

**SQLite(디폴트로 제공) - 개발이나, 트래픽이 높지않은 웹사이트에서만 사용가능**

**PostreSQL, MySQL**, MariaDB, Oracle 장고 커뮤니티에서는 앞의 2개를 가장 많이 사용한다.

마이그레이션이란? 일반적으로 하나의 운영환경에서 다른 운영환경으로 옮겨가는 것

데이터 마이그레이션이란 데이터를 한 종류의 스토리지에서 다른 종류의 스토리지로 이동하는 것

**장고 마이그레이션** : 모델의 변경내역을 DB스키마에 적용시키는 장고의 방식, DB스키마를 git처럼 변경내역이 기록된 여러 버전으로 나눠서 관리할 수 있게 해 주는 시스템이다. 하나의 마이그레이션 파일은 **해당 마이그레이션이 생성된 시점의 모델(DB스키마)의 구조**(데이터X, 데이터의 구조)를 담고 있다.

마이그레이션 활용법

1. 모델을 만든다.
2. 마이그레이션을 만든다.

**python manage.py makemigrations**

각 앱의 모델에 작성한 코드를 기반으로하여 장고가 마이그레이션파일을 만든다.

해당 명령을 실행하면, 각 앱 폴더하위의 마이그레이션 폴더에 0001\_initial.py라는 이름의 마이그레이션파일(해당 시점의 모델 코드를 기반으로 작성된 DB구조가 저장된 파일)이 생성된다.

해당파일을 보면, migrations.Migration클래스를 상속한 Migration클래스를 만들고, CreateModel 등 각종 operations(실행할 명령어)를 python프로그램으로 작성해 놓은 것이다.

처음 모델을 만들었을 때 뿐만 아니라, 모델을 변경할 때마다, makemigrations를 해 주어야 한다.

Vs code 검색 Ctrl + P 창에서, 그냥 입력(파일검색), #다음에 입력(모든 파일에서 심볼검색) @다음에 입력(해당 파일에서 심볼검색), :222(해당 파일의 특정 라인으로 이동

슬러그란 주소창에 get-max-and-min-formatted .... 등으로 콘텐츠내용을 넣어서 주소로 만든 것(SEO, search engine optimization테크닉의 일종이다)

* 사용자 정의 마이그레이션 만들기(빈마이그레이션)

**python manage.py makemigrations app\_name --empty**

    operations = [

        migrations.RunSQL("""

            INSERT INTO store\_collection (title)

            VALUES ('collections1')

        """, """

            DELETE FROM store\_collection

            WHERE title='collection1'

        """)

    ]

첫번째 SQL은 데이터를 변경하기 위한 것이고, 두번째 SQL을 해당 데이터 변경을 취소하고자 할 때 사용하는 SQL이다.

1. 마이그레이션을 실행한다.

**python manage.py migrate**

**python manage.py sqlmigrate**를 하면 db에 전송되는 sql코드를 실시간으로 볼 수 있다.

특정 앱의 특정 마이그레이션을 실행할 때 사용하면 좋다.   
Python manage.py sqlmigrate store(앱이름) 0003(번호)

마이그레이션을 실행하면, 장고가 기본앱 실행을 위해 미리 만들어놓은 다른 마이그레이션파일들이 함께 실행되면서 데이터베이스가 생성된다.

* vscode에서 sqlite익스텐션 사용하는 법

**커맨드팔레트에서 sqlite: open database를 실행**하고, 열 파일을 지정하면, 왼쪽 사이드바에 SQLITE EXPLORER가 생긴다.

1. 마이그레이션 취소하는 법
2. 새로운 마이그레이션을 만들어서 마이그레이트한다.
3. 이동하고 싶은 마이그레이션 번호를 다시 실행한 후**, (1)** **이후의 마이그레이션 파일과 (2) model 파일 내의 해당 코드를 모두 삭제**한다.

python manage.py migrate store 0003

모델의 서브클래스인 Meta

Django model metadata로 검색하면, 모델의 서브클래스인 Meta에서 사용할 수 있는 옵션들을 찾을 수 있다.

db\_table, ordering, indexes 등

MySQL 시작하는 법 : services.msc

장고에서 MySQL 사용하는 법

1. mysqlclient를 인스톨한다.

**pipenv install mysqlclient**

1. 기본앱의 settings.py의 데이터베이스 셋팅을 바꾸어 준다.

디폴트 셋팅(sqlite)

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': BASE\_DIR / 'db.sqlite3',

    }

}

MySQL 셋팅

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.mysql',

        'HOST': 'localhost',

        'USER': 'root',

        'PASSWORD': 'password',

        'NAME': 'django\_learn' # DB\_NAME

    }

}

1. MySQL에서 데이터베이스를 만든다.

CREATE DATABASE DB\_NAME

1. 마이그레이트를 실행한다.

python manage.py migrate

더미데이터 만드는 법 : mockaroo.com

ORM이란 Object Relational Mapper

데이터베이스 테이블의 데이터를 자동으로 오브젝트와 매핑시키는 코드. 예전에는 일일히 반복적인 SQL쿼리를 작성하여야 했으나, 이제는 코드만 작성하면 ORM이 알아서 해준다. 다만, 복잡한 SQL쿼리는 직접 작성하여야 한다.

장고에서 models.Model을 상속하는 엔티티 클라스들은 objects라는 속성이 있는데, 이 속성이 ORM매니저이다. 이 매니저의 펑션이 반환하는 값은 대부분 object가 아닌 query\_set이다.(레이지한 동작). 일부는 즉시 값을 반환하기도 한다.(count, get등)

다만, 인덱스로 접근하는 것은 가능하다.

query\_set = Product.objects.all().filter(

        collection\_\_id=3).order\_by('title')[0]

query\_set = Product.objects.all().order\_by('title')[0]

query\_set = Product.objects.order\_by('title')[0]

장고 ORM

매니저 : 모델.objects - 매니저 오브젝트

쿼리셋 : 매니저 오브젝트의 메소드들은 보통 쿼리셋을 반환한다(레이지 리트리빙)

쿼리셋이 데이터를 반환하는 경우 : 이터레이트 할 때, 리스트로 변환할 때, 인덱스로 특정 데이터에 접근할 때 등

데이터 조회

queryset = Collection.objects.all()

SELECT \* FROM collection

collection = Collection.objects.get(id=1)

SELECT \* FROM collection WHERE id = 1

collection = Collection.objects.get(title='beauty')

SELECT \* FROM collection WHERE title = 'beauty'

get 메소드는 리턴값이 하나인 경우에만 써야 한다. 리턴값이 하나 이상이 되면 exception이 발생한다. get 메소드는 오브젝트를 반환한다.

from django.core.exceptions import ObjectDoesNotExist

try:

    collection = Collection.objects.get(pk=1)

except ObjectDoesNotExist:

    pass

get 메소드는 반환값이 없으면 exception을 발생시키므로, 예외처리를 해 주어야 한다.

collection = Collection.objects.filter(pk=0).first()

exists = Collection.objects.filter(pk=0).exists()

filter().first()는 반환데이터가 없더라도 exception을 발생시키지 않는다.

데이터 필터링(field lookup 검색)

**django queryset api**를 키워드로 검색하면 된다.

query\_set = Collection.objects.filter(pk\_\_gt=5)

SELECT \* FROM collection WHERE id > 5;

query\_set = Collection.objects.filter(pk\_\_range=(3, 6))

SELECT \* FROM collection WHERE id BETWEEN 3 AND 6

query\_set = Product.objects.filter(collection\_id=2)

SELECT \* FROM product WHERE collection\_id = 2

query\_set = Product.objects.filter(collection\_\_id\_\_range=(3, 6))

SELECT \* FROM product WHERE collection\_id BETWEEN 3 AND 6

collection\_id의 경우는 product테이블의 외래키인 collection\_id를 조사하는 것이고, collection\_\_id의 경우는 의미는 product테이블의 관련테이블인 collection에서 id를 조사하는 것이지만, 실제로는 product테이블에 이미 외래키인 collection\_id가 있기 때문에, 테이블 조인을 하지 않는다.

그러나, 외래키가 아닌 관련테이블의 컬럼을 조회하는 경우에는 테이블 조인이 일어난다.

query\_set = Product.objects.filter(collection\_\_title='beauty')

product의 입장에서 볼 때 parent테이블인 collection (product 테이블에 collection외래키가 있음, product와 collection은 N:1 관계)과도 조인할 수 있고,

SELECT \* FROM product JOIN collection ON product.collection\_id = collection.id  
WHERE collection.title = 'beauty'

다만 이 경우에도 가져오는 컬럼은 product 테이블의 컬럼 뿐이다.(collection\_\_title을 가져오지는 않음)

product의 입장에서 볼 때 children테이블인 orderitem (orderitem테이블에 product 외래키가 있음, product와 orderitem은 1:N 관계)과도 조인할 수 있다.

query\_set = Product.objects.filter(orderitem\_\_quantity=3)

SELECT \* FROM product JOIN orderitem ON product.id = orderitem.product\_id  
WHERE orderitem.quantity = 3

이 경우에도 가져오는 컬럼은 product 테이블의 컬럼 뿐이다.

collection\_id를 쓸 때 더블 언더스코어를 써야 하는 이유는 뭘까? => 더블 언더스코어의 의미는 관련테이블의 컬럼을 참조한다는 의미이다.

queryset = Customer.objects.filter(

        order\_\_orderitem\_\_product\_\_title\_\_icontains='bread')

\_\_ (더블 언더스코어를 연속으로 사용해서, 조인하는 테이블을 확장할 수 있다(물론, 테이블간에 관련성이 있어야 함). 단, 이 경우에도 가져오는 컬럼은 customer의 컬럼 뿐이다.

**SELECT** `store\_customer`.`id`,  
       `store\_customer`.`first\_name`,  
       `store\_customer`.`last\_name`,  
       `store\_customer`.`email`,  
       `store\_customer`.`phone`,  
       `store\_customer`.`birth\_date`,  
       `store\_customer`.`membership` **FROM** `store\_customer` **INNER JOIN** `store\_order` **ON** (`store\_customer`.`id` = `store\_order`.`customer\_id`) **INNER JOIN** `store\_orderitem` **ON** (`store\_order`.`id` = `store\_orderitem`.`order\_id`) **INNER JOIN** `store\_product` **ON** (`store\_orderitem`.`product\_id` = `store\_product`.`id`) **WHERE** `store\_product`.`title` LIKE '%bread%'

query\_set = Product.objects.filter(title\_\_contains='coffee')

SELECT \* FROM product WHERE title LIKE BINARY '%coffe%'

query\_set = Product.objects.filter(title\_\_icontains='coffee')

SELECT \* FROM product WHERE title LIKE '%coffe%'

contains는 대소문자를 구별함, icontains는 구별하지 않음(case-insensitive)

query\_set = Product.objects.filter(title\_\_istartswith='coffee')

SELECT \* FROM product WHERE title LIKE 'coffee%'

query\_set = Product.objects.filter(title\_\_endswith='ed')

SELECT \* FROM product WHERE title LIKE BINARY '%ed'

query\_set = Product.objects.filter(last\_update\_\_year=2021)

SELECT \* FROM product WHERE last\_update BETWEEN '2021-01-01 00:00:00' AND '2021-12-31 23:59:59:999999'

query\_set = Product.objects.filter(description\_\_isnull=True)

SELECT \* FROM product WHERE description IS NULL

복합조건

query\_set = Product.objects.filter(inventory\_\_lt=10, unit\_price\_\_lt=20)

where condition 1 and condition 2

query\_set = Product.objects.filter(

        inventory\_\_lt=10).filter(unit\_price\_\_lt=20)

where condition 1 and condition 2

from django.db.models import Q

query\_set = Product.objects.filter(

        Q(inventory\_\_lt=10) | Q(unit\_price\_\_lt=20))

where condition 1 or condition 2

query\_set = Product.objects.filter(

        Q(inventory\_\_lt=10) & ~Q(unit\_price\_\_lt=20))

이런 형식도 가능 where condition and not condition 2

**WHERE** (`store\_product`.`inventory` < 10 **AND** **NOT** (`store\_product`.`unit\_price` < 20))

from django.db.models import F

query\_set = Product.objects.filter(

        inventory=F('unit\_price') \* 2)

query\_set = Product.objects.filter(

        inventory=F('collection\_\_id'))

필드를 비교해야 할 필요가 있을 때는 F오브젝트 사용

**WHERE** `store\_product`.`inventory` = (`store\_product`.`unit\_price` \* 2)

소팅

query\_set = Product.objects.order\_by('title')

**ORDER BY** `store\_product`.`title` **ASC**

query\_set = Product.objects.order\_by('-title').reverse()

query\_set = Product.objects.order\_by('title').reverse()

**ORDER BY** `store\_product`.`title` **DESC**

query\_set = Product.objects.order\_by('-title', 'unit\_price')

**ORDER BY** `store\_product`.`title` **DESC**, `store\_product`.`unit\_price`

결과 제한하기

product = Product.objects.order\_by('title')[0]

product = Product.objects.order\_by('title').first()

product = Product.objects.earliest('title')

**ORDER BY** `store\_product`.`title` **ASC** **LIMIT** 1

product = Product.objects.order\_by('title').last()

product = Product.objects.latest('title')

**ORDER BY** `store\_product`.`title` **DESC  LIMIT** 1

query\_set = Product.objects.all()[:5]

limit 5

query\_set = Product.objects.all()[5:10]

limit 5 offset 5

일부 필드만 읽어 오기

query\_set = Product.objects.values('id', 'title', collection\_\_title)

**딕셔너리 형식**으로 각 필드를 읽어온다.

{'id': 2, 'title': 'Island Oasis - Raspberry', 'collection\_\_title': 'Beauty'}

query\_set = Product.objects.values\_list('id', 'title', collection\_\_title)

**터플 형식**으로 각 필드를 읽어온다.

(2, 'Island Oasis - Raspberry', 'Beauty')

관련성이 있는 테이블의 다른 필드도 읽어 올 수 있고, 이 때에는 더블 언더스코어를 사용해야 한다. 위에서 의문을 가졌던 것

queryset = Customer.objects.values('first\_name', 'order\_\_id', 'order\_\_orderitem\_\_product\_\_title').filter(

        order\_\_orderitem\_\_product\_\_title\_\_icontains='bread')

이런 식으로 관련성이 있는 테이블을 연결, 연결해서(조인, 조인해서) 먼 관련성이 있는 테이블의 컬럼도 가져올 수 있다. 다만 위 쿼리의 결과는 bread라는 상품명이 포함된 상품을 주문한 적이 있는 커스터머의 리스트를 구하는 것인데, 조회 결과는 해당 커스터머의 주문이 모두 나온다(주문 상품에 bread라는 상품명이 없는 주문까지 모두 나온다).

queryset = OrderItem.objects.values(

        'id', 'order\_\_customer\_\_first\_name', 'product\_\_title').filter(product\_\_title\_\_icontains='bread')

이와 같은 쿼리를 사용하면 목적을 달성할 수 있음을 알 수 있다.

html에 렌더링 하는 법

iterable은 list로 변환하여 렌더링, 단일 오브젝트는 그냥 렌더링

return render(request, 'hello.html', {'collections': list(query\_set)})

views.py

<ul>

  {% for collection in collections %}

  <li>{{ collection.title }}</li>

  {% endfor %}

</ul>

template

<ul>

    {% for product in products %}

    <li>{{ product.title }} {{ product.collection\_\_title }}</li>

    {% endfor %}

</ul>

딕셔너리(valuse)의 경우 템플릿에 표시는 위와 같이 한다.

<ul>

    {% for product in products %}

    <li>{{ product.0 }} {{ product.1 }}</li>

    {% endfor %}

</ul>

튜플(values\_list)의 경우는 템플릿에 표시는 위와 같이 한다. 브래킷 노테이션이 아니라, 닷 노테이션을 사용함에 주의

query\_set = Product.objects.only('id', 'collection\_\_title')

Product 클래스의 인스턴스 형식으로 각 컬럼을 읽어온다. only의 경우에는 관련 테이블의 다른 컬럼을 읽어 오라고 위와 같이 지정해도, 이를 읽어 오지 않는다. 해당 테이블의 데이터만 조회해 온다. 따라서 only를 사용하기 위해서는 select\_related나, prefetch\_related로 관련 테이블의 컬럼까지 가져와야 한다.

queryset = Product.objects.select\_related(

        'collection').only('title', 'collection\_\_title')

**SELECT** `store\_product`.`id`,  
       `store\_product`.`title`,  
       `store\_product`.`collection\_id`,  
       `store\_collection`.`id`,  
       `store\_collection`.`title` **FROM** `store\_product` **INNER JOIN** `store\_collection` **ON** (`store\_product`.`collection\_id` = `store\_collection`.`id`)

queryset = Product.objects.prefetch\_related(

        'collection').only('title', 'collection\_\_title')

**SELECT** `store\_product`.`id`,  
       `store\_product`.`title`,  
       `store\_product`.`collection\_id` **FROM** `store\_product`

**SELECT** `store\_collection`.`id`,  
       `store\_collection`.`title` **FROM** `store\_collection` **WHERE** `store\_collection`.`id` IN (2, 3, 4, 5, 6)

values, values\_list, only는 매개변수 안에 지정된 컬럼만을 읽어 온다는 점에서 같지만, values는 딕셔너리 형식으로, values\_list는 터플 형식으로, only는 해당 클래스의 인스턴스 형식으로 읽어온다는 점이 차이가 있다.

<ul>

    {% for product in products %}

    <li>{{ product.title }} {{ product.collection.title }}</li>

    {% endfor %}

</ul>

인스턴스의 경우 템플릿에 표시는 위와 같이 해야 한다.

query\_set = Product.objects.defer('id', 'title')

only의 반대, title 컬럼을 빼고 읽어온다. (defer에 'id'를 넘겨 주어도 id는 항상 읽어 오는 것 같다)

only와 defer는 템플릿에서 읽어오지 않은 필드를 참조하는 경우, 스스로 실시간으로(?) 쿼리를 생산하기 때문에 속도가 느려질 수 있어, 주의해야 한다.(only, defer만 그런것이 아니라, 쿼리결과가 인스턴스로 반환되는 경우는 다 그런 것 같다)

Many가 One을 조회할 때(product가 collection을 조회할 때, order가 customer를 조회할 때 orderitem이 product를 조회할 때 등등)는 select\_related와, prefetch\_related 양자를 모두 사용할 수 있다. select\_related와 prefetch\_related는 어떠한 차이를 갖는가? prefetch\_related를 사용할 필요가 없는 것인가? 별도의 사용용도가 있는가?

그러나, One이 Many를 조회할 때(collection이 product\_set을 조회할 때, customer가 order\_set을 조회할 때, product가 orderitem\_set을 조회할 때 등등)는 select\_related는 사용할 수 없고, prefetch\_related만 사용할 수 있다. select\_related는 공통된 id를 가지고 조인을 하는데, Many에는 One을 참조하는 컬럼(공통컬럼)에 대한 정보가 있지만, One에는 Many를 참조하는 컬럼에 대한 정보가 없기 때문이다(참조 대상이 여럿이기 때문에 참조가 불가능하다).

query\_set = Product.objects.select\_related('collection')

select\_related 명령은 관련테이블에 관한 외래키를 이용하여 관련테이블과 조인하여 관련테이블의 컬럼까지 조회할 수 있도록 하는 것이다.

queryset = Product.objects.select\_related('collection\_\_featured\_product')

product는 collection\_id를 외래키로 갖고, collection의 featured\_product 컬럼은 product\_id를 외래키로 갖기 때문에 위와 같이 연속적인 select\_related를 할 수 있다.

queryset = OrderItem.objects.select\_related('product\_\_collection')

queryset = OrderItem.objects.select\_related(

        'product\_\_collection\_\_featured\_product')

queryset = OrderItem.objects.select\_related(

        'product').select\_related('order')

queryset = OrderItem.objects.select\_related(

        'product', 'order')

연속적인 select\_related 뿐만 아니라, 병렬적인 select\_related도 할 수 있다.

query\_set = Product.objects.prefetch\_related('promotions')

Many to Many 관계(product와 promotion의 관계)에 있거나 One to Many 관계(collection과 product, customer와 order의 관계)에 있는 경우에는 해당 인스턴스에 참조할 수 있는 관련 컬럼이 없으므로, select\_related는 사용할 수 없고, prefetch\_related를 사용할 수 밖에 없다. 이 경우에는 장고가 2개의 쿼리를 실행해서, One과 관련있는 Many 쪽에 있는 테이블의 내용까지 가져온다.

query\_set = Product.objects.prefetch\_related(

        'promotions').select\_related('collection')

처럼 체이닝하는 것도 가능, 순서 바꾸는 것도 가능(모두 프리로드)

외래키 테이블(다대일)은 select\_related가능(prefetch\_related도 가능), 그렇지 않은 것은(다대다, 일대다) prefetch\_related만 가능

**관련 테이블의 컬럼을 모두 가져올 수는 없는가? 가져올 수 있다. select\_related의 병렬적 사용, 연속적 사용, prefetch\_related와 체이닝 등등을 통해서 관련 테이블의 컬럼을 모두 가져올 수 있다.**

queryset = Order.objects.select\_related(

        'customer').prefetch\_related('orderitem\_set\_\_product').order\_by('-placed\_at')[:10]

{% for order in result %}

    <p>

      {{ order.id }}

      <span class="blue">{{ order.placed\_at }}</span>

      <span class="red"

        >{{ order.customer.first\_name }} {{ order.customer.last\_name }}</span

      >

      {% for orderitem in order.orderitem\_set.all %}

      <span class="blue">{{ orderitem.product.title }}</span>

      <span class="red">{{ orderitem.quantity }}</span>

      {% endfor %}

    </p>

    {% endfor %}

408 June 9, 2021, midnight Olin Mugglestone Passion Fruit 5 Beef - Rib Roast, Capless 2

513 June 8, 2021, midnight Merna Grieg Bread - Frozen Basket Variety 1

711 June 8, 2021, midnight Rubi Dahlback Artichoke - Fresh 4

2 June 7, 2021, midnight Nicko Fay

248 June 7, 2021, midnight Shandeigh Morphew Yogurt - Assorted Pack 2

327 June 7, 2021, midnight Karlotta Danko

766 June 7, 2021, midnight Rollin Heeley Pepper - Green, Chili 1

35 June 6, 2021, midnight Lockwood Castellet Sauce - Hp 3 Wine - Hardys Bankside Shiraz 4

640 June 6, 2021, midnight Bea Ellams Dill - Primerba, Paste 2

892 June 5, 2021, midnight Pippo Pollie Apricots - Dried 1

queryset = Order.objects.values(

        'id',

        'placed\_at',

        'customer\_\_first\_name',

        'customer\_\_last\_name',

        'orderitem\_\_product\_\_title',

        'orderitem\_\_quantity'

    ).order\_by('-placed\_at')[:10]

{% for order in result %}

    <p>

      {{ order.id }}

      <span class="blue">{{ order.placed\_at }}</span>

      <span class="red"

        >{{ order.customer\_\_first\_name }} {{ order.customer\_\_last\_name }}</span

      >

      <span class="blue">{{ order.orderitem\_\_product\_\_title }}</span>

      <span class="red">{{ order.orderitem\_\_quantity }}</span>

    </p>

    {% endfor %}

408 June 9, 2021, midnight Olin Mugglestone Passion Fruit 5

408 June 9, 2021, midnight Olin Mugglestone Beef - Rib Roast, Capless 2

513 June 8, 2021, midnight Merna Grieg Bread - Frozen Basket Variety 1

711 June 8, 2021, midnight Rubi Dahlback Artichoke - Fresh 4

2 June 7, 2021, midnight Nicko Fay None None

248 June 7, 2021, midnight Shandeigh Morphew Yogurt - Assorted Pack 2

327 June 7, 2021, midnight Karlotta Danko None None

766 June 7, 2021, midnight Rollin Heeley Pepper - Green, Chili 1

35 June 6, 2021, midnight Lockwood Castellet Sauce - Hp 3

35 June 6, 2021, midnight Lockwood Castellet Wine - Hardys Bankside Shira

통합 컬럼 사용하기 : COUNT, SUM, MAX, MIN, AVG

from django.db.models.aggregates import Count, Max, Min, Avg, Sum

id\_count = Product.objects.aggregate(Count("id"))

id\_count = {'id\_\_count': 1000}

price\_average = Product.objects.aggregate(Avg("unit\_price"))

price\_average = {'unit\_price\_\_avg': Decimal('50.353210')}

maxmin = Product.objects.aggregate(Max('unit\_price'), Min('unit\_price'))

maxmin = {'unit\_price\_\_max': Decimal('99.96'), 'unit\_price\_\_min': Decimal('1.06')}

 maxmin = Product.objects.aggregate(

        max=Max('unit\_price'), min=Min('unit\_price'))

maxmin = {'max': Decimal('99.96'), 'min': Decimal('1.06')}

product1\_sold = OrderItem.objects \

        .filter(product\_\_id=1) \

        .aggregate(sales=Sum('quantity'))

SELECT SUM(oi.quantity)   
FROM store\_orderitem oi JOIN store\_product p ON oi.product\_id = p.id  
WHERE p.id = 1;

새로운 컬럼 추가하기

from django.db.models import F, Value

queryset = Customer.objects.annotate(isNew=Value(True))

queryset = Customer.objects.annotate(new\_id=F('id') + 10)

**SELECT** `store\_customer`.`id`,  
       `store\_customer`.`first\_name`,  
       `store\_customer`.`last\_name`,  
       `store\_customer`.`email`,  
       `store\_customer`.`phone`,  
       `store\_customer`.`birth\_date`,  
       `store\_customer`.`membership`,  
       (`store\_customer`.`id` + 10) **AS** `new\_id` **FROM** `store\_customer`

장고의 expression 클래스 : Value(값), F(필드값), Func(함수값), Aggregate(총합값)

데이터베이스 함수 사용하기

from django.db.models import F, Value, Func

queryset = Customer.objects.annotate(

        full\_name=Func(

            F('first\_name'),

            Value(' '),

            F('last\_name'),

            function='Concat'))

**SELECT** `store\_customer`.`id`,  
       `store\_customer`.`first\_name`,  
       `store\_customer`.`last\_name`,  
       `store\_customer`.`email`,  
       `store\_customer`.`phone`,  
       `store\_customer`.`birth\_date`,  
       `store\_customer`.`membership`,  
       Concat(`store\_customer`.`first\_name`, ' ', `store\_customer`.`last\_name`) **AS** `full\_name` **FROM** `store\_customer`

from django.db.models.functions import Concat

queryset = Customer.objects.annotate(

        full\_name=Concat(

            'first\_name',

            Value(' '),

            'last\_name'))

django database functions로 검색하면 다른 데이터베이스 펑션들을 볼 수 있다.

그룹핑하기

queryset = Customer.objects.annotate(

        orders\_count=Count('order')

    )

|  |  |
| --- | --- |
| **SELECT** `store\_customer`.`id`,        `store\_customer`.`first\_name`,        `store\_customer`.`last\_name`,        `store\_customer`.`email`,        `store\_customer`.`phone`,        `store\_customer`.`birth\_date`,        `store\_customer`.`membership`,        COUNT(`store\_order`.`id`) **AS** `orders\_count` **FROM** `store\_customer` **LEFT OUTER JOIN** `store\_order` **ON** (`store\_customer`.`id` = `store\_order`.`customer\_id`) **GROUP BY** `store\_customer`.`id` |  |

aggregate 함수를 사용하였을 뿐인데, 자동으로 그룹핑이 되었다.

\_set이란 명칭은 prefetch 할 때만 사용하는 명칭인 것 같다.(확실하지 않다)

queryset = Customer.objects.annotate(

        last\_order\_id=Max('order\_\_id')

    )

**SELECT** `store\_customer`.`id`,  
       `store\_customer`.`first\_name`,  
       `store\_customer`.`last\_name`,  
       `store\_customer`.`email`,  
       `store\_customer`.`phone`,  
       `store\_customer`.`birth\_date`,  
       `store\_customer`.`membership`,  
       MAX(`store\_order`.`id`) **AS** `last\_order\_id` **FROM** `store\_customer` **LEFT OUTER JOIN** `store\_order` **ON** (`store\_customer`.`id` = `store\_order`.`customer\_id`) **GROUP BY** `store\_customer`.`id`

queryset = Collection.objects.annotate(

        products\_count=Count('product')

    )

**SELECT** `store\_collection`.`id`,  
       `store\_collection`.`title`,  
       `store\_collection`.`featured\_product\_id`,  
       COUNT(`store\_product`.`id`) **AS** `products\_count` **FROM** `store\_collection` **LEFT OUTER JOIN** `store\_product` **ON** (`store\_collection`.`id` = `store\_product`.`collection\_id`) **GROUP BY** `store\_collection`.`id`

queryset = Customer.objects.annotate(

        orders\_count=Count('order')

    ).filter(orders\_count\_\_gte=5)

|  |  |
| --- | --- |
| **SELECT** `store\_customer`.`id`,        `store\_customer`.`first\_name`,        `store\_customer`.`last\_name`,        `store\_customer`.`email`,        `store\_customer`.`phone`,        `store\_customer`.`birth\_date`,        `store\_customer`.`membership`,        COUNT(`store\_order`.`id`) **AS** `orders\_count` **FROM** `store\_customer` **LEFT OUTER JOIN** `store\_order` **ON** (`store\_customer`.`id` = `store\_order`.`customer\_id`) **GROUP BY** `store\_customer`.`id` **HAVING** COUNT(`store\_order`.`id`) >= 5 |  |

queryset = Customer.objects.annotate(

        total\_purchase=Sum(F('order\_\_orderitem\_\_quantity') \*

                           F('order\_\_orderitem\_\_unit\_price'))

    )

**SELECT** `store\_customer`.`id`,  
       `store\_customer`.`first\_name`,  
       `store\_customer`.`last\_name`,  
       `store\_customer`.`email`,  
       `store\_customer`.`phone`,  
       `store\_customer`.`birth\_date`,  
       `store\_customer`.`membership`,  
       SUM((`store\_orderitem`.`quantity` \* `store\_orderitem`.`unit\_price`)) **AS** `total\_purchase` **FROM** `store\_customer` **LEFT OUTER JOIN** `store\_order` **ON** (`store\_customer`.`id` = `store\_order`.`customer\_id`) **LEFT OUTER JOIN** `store\_orderitem` **ON** (`store\_order`.`id` = `store\_orderitem`.`order\_id`) **GROUP BY** `store\_customer`.`id`

queryset = Product.objects.annotate(

        total\_sales=Sum(F('orderitem\_\_quantity') \*

                        F('orderitem\_\_unit\_price'))

    ).order\_by('-total\_sales')[:5]

**SELECT** `store\_product`.`id`,  
       `store\_product`.`title`,  
       `store\_product`.`slug`,  
       `store\_product`.`description`,  
       `store\_product`.`unit\_price`,  
       `store\_product`.`inventory`,  
       `store\_product`.`last\_update`,  
       `store\_product`.`collection\_id`,  
       SUM((`store\_orderitem`.`quantity` \* `store\_orderitem`.`unit\_price`)) **AS** `total\_sales` **FROM** `store\_product` **LEFT OUTER JOIN** `store\_orderitem` **ON** (`store\_product`.`id` = `store\_orderitem`.`product\_id`) **GROUP BY** `store\_product`.`id` **ORDER BY** `total\_sales` **DESC  
 LIMIT** 5

익스프레션 래퍼(ExpressionWrapper)

from django.db.models import F, ExpressionWrapper

from django.db.models.fields import DecimalField

queryset = Product.objects.annotate(

        discounted\_price=ExpressionWrapper(

            F('unit\_price') \* 0.8, output\_field=DecimalField())

    )

일반 관계 조회하기

일반 관계 모델링

class TaggedItem(models.Model):

    tag = models.ForeignKey(Tag, on\_delete=models.CASCADE)

    # Type of object - find table

    content\_type = models.ForeignKey(ContentType, on\_delete=models.CASCADE)

    # ID of object - find record

    object\_id = models.PositiveIntegerField()

    content\_object = GenericForeignKey()

장고가 관리하는 django\_content\_type 테이블에 장고의 빌트인 앱과, 사용자가 작성한 앱에서 사용하는 모든 모델에 대하여 아이디가 부여되어 있음

from django.contrib.contenttypes.models import ContentType

from store.models import Product

from tags.models import TaggedItem

content\_type = ContentType.objects.get\_for\_model(Product)

queryset = TaggedItem.objects \

    .select\_related('tag') \

    .filter(content\_type=content\_type, object\_id=1)

ContentType 모델 매이저에 특수한 펑션인 get\_for\_model 함수로 content\_type id를 가져오고, 이를 object\_id와 조합하여 쿼리 실행

매니저 커스터마이징하기(커스텀 매니저)

원칙적으로는 위와 같이 코딩하여야 하지만, 아래와 같이 직관적인 방법으로 코딩할 수 있도록 오리지날 매니저(objects)를 커스텀 매니저로 대체할 수 있다.

queryset = TaggedItem.objects.get\_tags\_for(Product, 1)

models.Manager 클래스를 상속하는 커스텀 매니저를 만들고, 커스텀 펑션을 정의

class TaggedItemManager(models.Manager):

    def get\_tags\_for(self, obj\_type, obj\_id):

        content\_type = ContentType.objects.get\_for\_model(obj\_type)

        return TaggedItem.objects \

        .select\_related('tag') \

        .filter(content\_type=content\_type, object\_id=obj\_id)

TaggedItem 클래스에서 매니저(objects) 오버라이딩

class TaggedItem(models.Model):

    objects = TaggedItemManager()

쿼리셋 캐싱 메커니즘

데이터베이스에 액세스하는 작업은 비용이 크기 때문에, 장고는 일정량의 데이터를 캐시에 저장해 놓고 동일한 데이터에 재접근시에 이 캐시를 활용한다. 장고가 불필요하게 데이터베이스에 중복 접근하지 않도록 캐시를 활용할 수 있게 코딩하는 것도 중요하다.

데이터 생성하기

1. ORM 매니저의 create 펑션을 이용하여 한번에 생성

collection = Collection.objects.create(title='Home Appliance', featured\_product=Product(pk=3))

print(collection.id)

1. 컬렉션 오브젝트를 한번에 만들고 save 펑션을 이용하여 생성

collection = Collection(title='Video Games', featured\_product=Product(pk=1))

collection.save()

print(collection.id)

1. 컬렉션 오브젝트를 순차적으로 만들고 save 펑션을 이용하여 생성

collection = Collection()

collection.title = 'Video Games'

collection.featured\_product = Product(id=1)

collection.featured\_product\_id = 1

collection.save()

print(collection.id)

외래키 입력방식은 2가지가 가능함

데이터 변경하기

1. ORM 매니저의 update 펑션을 사용하여 한번에 변경

Collection.objects.filter(pk=12).update(featured\_product=None)

1. 컬렉션 오브젝트를 만들고, 업데이트 한 후 save 펑션을 사용하여 변경

collection = Collection(pk=12)

collection.title = 'Furniture'

collection.save()

이와 같이 하는 것도 가능하지만, 이런 경우 collection의 초기값에 실제 12번 collection의 값이 반영되지 않기 때문에 문제가 생긴다.

1. 컬렉션 오브젝트를 읽어 오고, 업데이트 한 후 save 펑션을 사용하여 변경

collection = Collection.objects.get(pk=12)

collection.title = 'Furniture'

collection.save()

이런 식으로 하면, 실제 12번 collection 값을 가져와서 갱신하므로 위와 같은 문제가 생기지 않는다. 성능 저하가 있을 수 있지만, 큰 문제는 되지 않는다.

데이터 삭제하기

1. ORM 매니저의 delete 펑션을 사용하여 삭제

Collection.objects.filter(pk=11).delete()

1. 컬렉션 오브젝트를 만들어서 삭제

collection = Collection(pk=13)

collection.delete()

트랜잭션

Order를 저장하기 위해서는 먼저 Order를 생성하고, 그에 의존하는 OrderItem을 생성하여야 한다. 그런데, OrderItem를 생성하는 과정에서 에러가 발생하면, OrderItem이 없는 Order가 생성된다. 이러한 결과를 방지하기 위해 Order와 그에 속하는 OrderItem은 한꺼번에 생성되어야 하고, 트랜잭션을 사용하여야 한다.

from django.db import transaction

@transaction.atomic()

def order(request):

view펑션에 transaction.atomic() 데코레이터를 붙이면, 해당 펑션 전체가 하나의 트랜잭션으로 실행된다.

with transaction.atomic():

        order = Order()

        order.customer\_id = 1

        order.save()

        item = OrderItem()

        item.order = order

        item.product\_id = 1

        item.quantity = 3

        item.unit\_price = 10

        item.save()

펑션의 일부만을 트랜잭션 코드로 취급하고 싶으면, with 키워드를 이용한다.

SQL 쿼리 실행하기

raw\_queryset = Product.objects.raw('SELECT \* FROM store\_product')

장고 ORM으로 처리할 수 없는 복잡한 쿼리 또는 장고 ORM의 퍼포먼스가 문제가 있는 경우에만 사용해라

장고 모델과 일치하지 않는 쿼리를 실행해야 할 경우, 장고 ORM에 의존하지 않고 직접 데이터베이스에 질의하는 것도 가능하다.

from django.db import connection

cursor = connection.cursor()

cursor.execute('')

cursor.close()

에러 발생할 수 있으므로, try except finally 블록으로 잡아 줘야 함

with connection.cursor() as cursor:

        cursor.execute('')

with 키워드를 사용하는 것이 간편하다.

 with connection.cursor() as cursor:

        cursor.callproc('get\_customers', [...])

SQL쿼리를 파이썬 코드에서 사용하는 것보다는 데이터베이스에 procedure를 만들어 저장해 놓고, 이를 호출하여 사용하는 것이 더 좋다.

ORM모델은 SQL복습과 함께 많은 연습을 해야할 것 같다. 이해하기도 어렵고, 익숙해지기는 더욱 어렵다.

Admin 사이트

사이트 관리자(admin)로서 웹서비스의 모든 데이터베이스를 관리할 수 있는 UI 시스템이 이미 장고에 빌트인 되어있으므로, 따로 제작할 필요가 없으며, 구미에 맞게 커스터마이징하기만 하면 된다.

Admin 사이트 셋업하기

접근 경로 : /admin

1. 유저 만들기

**python manage.py createsuperuser**

username, email, password 입력

1. 비밀번호 리셋하기

**python manage.py changepassword** username

admin 화면 커스터마이즈하기

처음에 접속하면, Groups와 Users 데이터베이스를 관리할 수 있는 UI만을 볼 수 있다. Groups와 Users는 장고의 빌트인 앱 중의 하나인 auth앱에서 관리한다.

헤더와 타이틀 설정하기

admin.site.site\_header = "django\_learn Admin"

admin.site.index\_title = 'Admin'

기본앱의 urls.py에서 admin을 import하므로, 해당 모듈에서 설정하는 것이 좋다.

장고의 앱들은 각각 자신의 admin.py를 갖는다. 해당 파일에서 관련 모델을 등록(register)하고 커스터마이징한다.

앱의 모델 등록하기

from django.contrib import admin

from . import models

# Register your models here.

admin.site.register(models.Collection, CollectionAdmin)

admin패키지의 ModelAdmin을 상속하는 각 모델별 Admin클래스를 만들어서 해당 Admin에 대한 전반적인 설정을 관리하는 것이 일반적이며, 이 경우에는 각 모델의 등록을 클래스에 어노테이션을 사용하여 간단히 할 수 있다.

admin.site.register(models.Product, ProductAdmin)

class ProductAdmin(admin.ModelAdmin):

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

대표 표시 커스터마이징 하기(모델 클래스에서 str펑션 오버라이딩)

class Collection(models.Model):

    def \_\_str\_\_(self) -> str:

        return self.title

class Meta:

        ordering = ['title']

메타 서브클래스에서 기본 정렬 순서를 정해 주면, 해당 순서로 정렬되어 표시된다.

@admin.register(models.Customer)

class CustomerAdmin(admin.ModelAdmin):

    list\_display = ['first\_name', 'last\_name', 'membership']

    list\_editable = ['membership']

    ordering = ['last\_name', 'first\_name']

    list\_per\_page = 10

각 모델별 어드민 클래스에서는 위와 같은 옵션들을 사용하여 세부 설정을 지정할 수 있다. list\_display를 이용하여 각 레코드의 내용 중 표시할 내용을 커스터마이징할 수 있다(위의 str 오버라이딩 필요 없음). ordering을 적으면 클래스의 meta서브클래스의 내용은 오버라이딩되는 것 같다.

모든 옵션을 보고 싶으면 **django model admin** 구글 검색

데이터 처리한 컬럼(computed column) 추가하기

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    list\_display = ['title', 'unit\_price', 'inventory\_status']

@admin.display(ordering='inventory')

    def inventory\_status(self, product):

        if product.inventory < 10:

            return 'Low'

        return 'OK'

@admin.display(ordering='inventory') 는 inventory\_status의 ordering을 할 수 있도록 써 주는 어노테이션이다.

class ProductAdmin(admin.ModelAdmin):

    list\_display = ['title', 'unit\_price',

                    'inventory\_status', 'collection']

list\_display에 관련 테이블을 추가하면, 해당 클래스의 str 표시가 표시된다.

class ProductAdmin(admin.ModelAdmin):

    list\_display = ['title', 'unit\_price',

                    'inventory\_status', 'collection\_title']

def collection\_title(self, product):

        return product.collection.title

위와 같이 관련테이블을 정의할 수도 있지만, 스스로 쿼리를 만드는 현상이 여기서도 발생한다.

list\_select\_related = ['collection']

이러한 현상을 방지하기 위해 list\_select\_related 속성을 추가하여 eager로딩, pre로딩을 할 수 있다.

기본 쿼리 셋 오버라이딩 하기

class CollectionAdmin(admin.ModelAdmin):

    list\_display = ['title', 'products\_count']

    def products\_count(self, collection):

        return collection.products\_count

    def get\_queryset(self, request: HttpRequest) -> QuerySet:

        return super().get\_queryset(request).annotate(

            products\_count=Count('product')

        )

컬렉션 어드민 패널에 해당 컬렉션의 products\_count를 표시하고 싶다.

'products\_count' 컬럼을 list\_display에 추가

products\_count 펑션으로 collection.products\_count라는 컬럼을 돌려 주도록 함

그러나, collection.products\_count라는 컬럼은 존재하지 않음

**ModelAdmin 클래스의 get\_queryset 펑션**을 오버로딩하여, collection 모델의 기본 쿼리에 products\_count라는 컬럼 추가 - annotate 펑션과, Count 펑션 사용

정렬 정보를 알려 주기 위해 products\_count 펑션에 어노테이션 추가

@admin.display(ordering='products\_count')

def products\_count(self, collection):

    return collection.products\_count

다른 어드민페이지로의 링크 추가

링크를 추가하려면 표시할 내용으로 html 링크를 포함하고 있는 스트링을 넘겨 주면 된다.

from django.utils.html import format\_html

def products\_count(self, collection):

        return format\_html('<a href="http://google.com">{}</a>', collection.products\_count)

웹사이트 내 링크할 페이지 주소를 요청하여 이 주소로 이동하게 한다.

from django.urls import reverse

reverse(‘admin:app\_model\_page) 의 형식으로 특정 페이지의 주소를 요청한다.

url = reverse('admin:store\_product\_changelist')

return format\_html('<a href="{}">{}</a>', url, collection.products\_count)

프로덕트 페이지의 주소 <http://localhost:8000/admin/store/product/> 에 ?collection\_\_id=1 이라는 서치쿼리를 적용하면, collection\_\_id=1인 프로덕트들이 표시된다.

url = (reverse('admin:store\_product\_changelist')

          + '?collection\_\_id='

          + str(collection.id))

위와 같은 코드를 사용하여, 다이나믹한 링킹을 할 수 있다.

from django.utils.html import urlencode

url = (reverse('admin:store\_product\_changelist')

            + '?'

            + urlencode({

                'collection\_\_id': str(collection.id)

            }))

유틸리티 펑션인 urlencod 펑션을 사용할 수도 있다. 스트링을 변환해 주는 역할만 하는 것 같다.

서치 필드 추가하기

앱의 어드민 클래스에 search\_fields 한줄 넣어 주면 됨

search\_fields = ['first\_name', 'last\_name']

search\_fields = ['first\_name\_\_istartswith', 'last\_name\_\_istartswith']

\_\_ 더블 언더스코어 뒤에 룩업 타입을 지정할 수 있음. 서치 스트링으로 시작할 것 및 케이스 인센시티브

필터링 추가하기

앱의 어드민 클래스에 list\_filter 한줄 넣어 주면 됨

list\_filter = ['collection', 'last\_update']

커스텀 필터 만드는 법

class InventoryFilter(admin.SimpleListFilter):

    title = 'inventory'

    parameter\_name = 'inventory'

    def lookups(self, request, model\_admin):

        return [

            ('<10', 'Low')

        ]

    def queryset(self, request, queryset):

        if self.value() == '<10':

            return queryset.filter(inventory\_\_lt=10)

심플리스트필터를 상속한 커스텀필터 클래스를 만든다.

타이틀은 필터 네임으로 쓰이는 것

파라미터 네임은 서치 쿼리할 때 사용 – 임의의 스트링을 사용하면 됨

lookups와 queryset 2개의 펑션을 오버라이딩 한다.

lookups는 서치 쿼리에서 사용될 스트링과, 필터 카테고리에서 사용될 스트링을 지정해 준다(임의의 스트링을 사용하면 됨)

queryset에서는 lookups에서 지정한 서치 쿼리 스트링에 대한 쿼리 요청이 있는 경우 어떠한 결과값을 반환할 것인지를 정해주면 된다(여기가 핵심 로직), 앞서 지정한 스트링들은 그냥 보여지는 것에 불과함.

list\_filter = ['collection', 'last\_update', InventoryFilter]

마지막으로 앱의 모델 어드민 클래스에 커스텀 필터를 추가해 준다.

커스텀 액션 추가하기

장고 어드민에는 기본적으로 선택한 데이터들을 한번에 지울 수 있는 Action이 빌트인 되어 있다.

@admin.action(description='Clear inventory')

def clear\_inventory(self, request, queryset):

    updated\_count = queryset.update(inventory=0)

    self.message\_user(

    request,

      f'{updated\_count} product were updated.'

    )

모델 어드민에 펑션 추가

@admin.action(description=’표시될 액션이름’) 어노테이션 추가

액션 펑션은 self, request, queryset을 매개변수로 사용함

액션을 실행하는 코드를 추가

self.message\_user 함수를 이용해서, 액션 완료 후 유저에게 전달할 메시지를 지정

from django.contrib import messages

self.message\_user(

            request,

            f'{updated\_count} product were updated.',

            messages.ERROR

        )

3번째 파라미터로 메시지의 속성(경고, 에러, 디버그 등)을 정해 줄 수 있음, 기본 값은 messages.INFO 인 것 같음(녹색), 경고는 노란색, 에러는 빨간색, 디버그는 화면이 아니라 다른 곳에 출력되는 듯

actions = ['clear\_inventory']

actions 속성 리스트에 커스텀 액션 펑션 추가

폼(입력 폼) 커스터마이징 하기

fields = ['title', 'slug']

표시할 필드(컬럼) 설정

exclude = ['title', 'slug']

제외할 필드 설정

readonly\_fields = ['title', 'slug']

읽기 전용 필드 설정(아래쪽에 표시된다)

prepopulated\_fields = {

        'slug': ['title']

    }

타이틀에 연동하여 slug가 자동생성되게 설정

autocomplete\_fields = ['collection']

ProductAdmin

search\_fields = ['title']

CollectionAdmin

컬렉션 필드를 자동완성 필드로 설정하기

Django ModelAdmin 옵션을 공부하여라

폼에 데이터 유효성 검증 추가하기

기본적으로 빌트인 된 유효성 검증 기능이 존재한다.

모델에서 product의 description을 nullable로 만들고, 마이그레이트 해도, 여전히 required 필드라고 나온다.

description = models.TextField(null=True)

null은 데이터베이스에만 적용되므로,

description = models.TextField(null=True, blank=True)

blank 매개변수까지 설정해 주어야 함

unit\_price의 경우에는 DecimalField로 설정해 주었기 때문에 음수값에 대한 유효성 체크를 하지 못하므로, 별도의 유효성 체크를 해 주어야 한다.

from django.core.validators import MinValueValidator

unit\_price = models.DecimalField(

        max\_digits=6,

        decimal\_places=2,

        validators=[MinValueValidator(1, message="Error!")])

모델에서 설정해 주는 것임

밸리데이터에 대해서는 Django validator로 검색하여 공부

한 곳에서 차일드 편집하기

class OrderItemInline(admin.TabularInline):

    model = models.OrderItem

extra = 1

    min\_num = 1

    max\_num = 10

우선 admin 모듈의 TabularInline 또는 StackedInline 클래스를 상속한 인라인 클래스를 생성

TabularInline은 한 레코드의 모든 필드가 한 줄에 나오고, StackedInline은 각 필드가 한 줄씩을 차지한다.

TabularInline, StackedInline도 ModelAdmin을 상속하므로, 인라인 클래스에서는 ModelAdmin의 모든 속성을 사용할 수 있다.

class OrderItemInline(admin.TabularInline):

    autocomplete\_fields = ['product']

    model = models.OrderItem

class OrderAdmin(admin.ModelAdmin):

    inlines = [OrderItemInline]

OrderAdmin 클래스에 inlines로 추가

일반 관계 사용하기

from django.contrib.contenttypes.admin import GenericTabularInline

class TaggedItemInline(GenericTabularInline):

    model = TaggedItem

extra = 0

class ProductAdmin(admin.ModelAdmin):

    inlines = [TaggedItemInline]

admin 모듈의 TabularInline이나 StackedInline 클래스가 아니라, GenericTabularInline 또는 GenericStackedInline 클래스를 상속하여 Inline을 만들어야 한다는 점을 빼고는 앞서 살펴본 한 화면에서 차일드 함께 편집하기와 같다.

class TaggedItemInline(GenericTabularInline):

    model = TaggedItem

    autocomplete\_fields = ['tag']

    extra = 1

@admin.register(models.Tag)

class TagAdmin(admin.ModelAdmin):

    list\_display = ['id', 'label']

    search\_fields = ['label']

    ordering = ['id']

플러그 가능한 앱 확장하기

위 방법의 문제점은 store앱의 admin에서 tags앱의 모델인 taggedItem을 임포트하고 있다는 것이다. 디커플링이 되지 않음

**python manage.py startapp store\_custom**

새로운 앱을 만든다.

INSTALLED\_APPS = [

    'store\_custom'

]

앱을 등록한다.

from django.contrib import admin

from store.admin import ProductAdmin

from tags.models import TaggedItem

새로 만든 앱의 admin에서 우리가 필요로 하는 ProductAdmin과 tags앱의 TaggedItem 모델을 import한다.

class TaggedItemInline(GenericTabularInline):

    model = TaggedItem

    autocomplete\_fields = ['tag']

    extra = 1

인라인 부분은 해당 코드를 그대로 가져오고

class CustomProductAdmin(ProductAdmin):

    inlines = [TaggedItemInline]

프로덕트 어드민을 상속만든 커스텀 프로덕트 어드민과 인라인을 결합시킨다.

store앱의 프로덕트 어드민을 unregister하고, store\_custom앱의 커스텀 프로덕트 어드민을 register한다.

admin.site.unregister(Product)

admin.site.register(Product, CustomProductAdmin)

나는 store앱의 어드민에서, register 어노테이션을 삭제하고, store\_custom앱의 어드민에서 register어노테이션을 추가했는데, 위와 같이 하는 방법도 있나 보다.

위와 같은 방법을 사용하면, installed app에서 ‘store\_custom’앱을 삭제해도 코드가 태그 없이 동작한다는 장점이 있다. 따라서 위 방법이 더 좋은 것 같다.

**제2부 : 장고로 RESTful API 만들기**

REST : Representational State Transfer

Resources : URL(Uniform Resource Locator)

http://moshbuy.com/products/1/reviews/1

Representations(Resource Representations)

HTML, XML(old form), JSON(modern form)

JSON : JavaScript Object Notation

Http Methods

POST, GET, PUT, PATCH(update part of it), DELETE

POST /products with JSON

PUT /products/1, PATHCH /products/1 with JSON

DELETE /products/1

장고 레스트 프레임워크를 설치한다.

장고에 기반하고 있는 별개의 프레임워크임

**Pipenv install djangorestframework**

INSTALLED\_APP에 추가한다.

INSTALLED\_APPS = [

    'rest\_framework'

]

API 뷰(컨트롤러, 리퀘스트핸들러) 생성하기

장고는 HttpRequest와 HttpResponse 클래스를 갖지만, 장고 레스트 프레임워크는 Request와 Response 클래스를 갖는다.

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

@api\_view()

def product\_list(request):

    return Response('ok')

@api\_view() 데코레이터를 붙이면, 뷰 펑션의 매개변수인 request도 장고의HttpRequest가 아니라 레스트 프레임워크의 Request 클래스의 인스턴스가 된다.

메소드 중에 OPTIONS 메소드는 해당 엔드포인트에서 어떤 메소드가 가능한지를 알아보는 메소드이다.

Vary는 캐시와 관련되어서 사용된다.

urlpatterns = [

    path('products/', views.product\_list),

    path('products/<id>/', views.product\_detail)

]

장고에서는 패러미터를 표현하는데 <>를 사용한다.

@api\_view()

def product\_detail(request, id):

    return Response(id)

패러미터를 받아서 사용할 수 있다.

path('products/<int:id>/', views.product\_detail)

타입을 정해 주면, 해당 타입의 데이터만 입력이 가능하다. 숫자가 아니면 404가 발생한다.

시리얼라이저 생성하기

프로덕트 리스트를 네트워크로 전송하기 위해서는 프로덕트 오브젝트를 제이슨 오브젝트로 바꾸어야 한다

모델 인스턴스(오브젝트) -> 시리얼라이저(레스트프레임워크, 내부적으로는 JSONRenderer 클래스의 render 메소드 사용) -> JSON

**serializers.py** 파일을 앱 내에 생성

from rest\_framework import serializers

class ProductSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    title = serializers.CharField(max\_length=255)

    unit\_price = serializers.DecimalField(max\_digits=6, decimal\_places=2)

내부적으로 보유하고 있는 데이터(internal representaion of a model)와 외부적으로 공개할 데이터(exteranl representation of a model)가 같을 필요는 없다.

오브젝트 시리얼라이즈하기

from .models import Product

from .serializers import ProductSerializer

@api\_view()

def product\_detail(request, id):

    product = Product.objects.get(pk=id)

    serializer = ProductSerializer(product)

    return Response(serializer.data)

레스트 프레임워크의 시리얼라이저는 오브젝트를 제이슨으로 바꾸어 준다.

모쉬는 모델과 시리얼라이즈의 필드 이름을 다른 것을 사용해도 된다고 했는데, 그렇지 않은 것 같다.

price = serializers.DecimalField(

        max\_digits=6, decimal\_places=2, source='unit\_price')

이런 식으로 source를 지정하면 다른 이름을 사용할 수 있다.

데시멀 밸류는 시리얼라이즈과정에서 스트링으로 변환되는데, 다음과 같이 설정하여 숫자로 반환되도록 할 수 있다.

REST\_FRAMEWORK = {

    'COERCE\_DECIMAL\_TO\_STRING': False

}

settings.py에 추가

from django.core.exceptions import ObjectDoesNotExist

@api\_view()

def product\_detail(request, id):

    try:

        product = Product.objects.get(pk=id)

        serializer = ProductSerializer(product)

        return Response(serializer.data)

    except ObjectDoesNotExist:

        return Response(status=404)

존재하지 않는 아이디를 조회했을 경우의 에러처리

except Product.DoesNotExist:

        return Response(status=404)

신기하게도 이런게 가능하다. Product에 DoesNotExist가 왜 있지? Model에서 상속받은 것 같다.

from rest\_framework import status

except Product.DoesNotExist:

        return Response(status=status.HTTP\_404\_NOT\_FOUND)

이렇게 써 주는 것이 좋다.

from django.shortcuts import get\_object\_or\_404

@api\_view()

def product\_detail(request, id):

    product = get\_object\_or\_404(Product, pk=1)

    serializer = ProductSerializer(product)

    return Response(serializer.data)

위와 같이 간단히 코딩할 수 있다. try, except 블록으로 처리하는 경우에는 리스폰스 보디에 아무것도 포함되지 않지만, 이 경우에는

{

"detail": "Not found."

}

이런 보디를 달고 온다.

프로덕트 리스트의 경우는 프로덕트와 동일하며, 시리얼라이저에 many=True만 추가해 주면 된다.

@api\_view()

def product\_list(request):

    queryset = Product.objects.all()

    serializers = ProductSerializer(queryset, many=True)

    return Response(serializers.data)

커스텀 시리얼라이저 필드 만들기

API는 가능한한 stable해야 한다. 그래야 그에 의존하는 것들이 안정되게 유지될 수 있다.

from rest\_framework import serializers

from decimal import Decimal

class ProductSerializer(serializers.Serializer):

    price\_with\_tax = serializers.SerializerMethodField(

        method\_name='calculate\_tax')

    def calculate\_tax(self, product):

        return round(product.unit\_price \* Decimal(1.1), 2)

타입 어노테이션을 붙여 주면, VS code의 인텔리센스가 작동해서 편한 면이 있다. 하지만 해당 클래스를 임포트해야하는 단점도 있음

from .models import Product

    def calculate\_tax(self, product: Product):

        return round(product.unit\_price \* Decimal(1.1), 2)

관계 시리얼라이징하기

from .models import Product, Collection

collection = serializers.PrimaryKeyRelatedField(

        queryset=Collection.objects.all()

    )

컬렉션 아이디가 반환된다.

collection = serializers.StringRelatedField()

컬렉션의 string 표현이 반환된다. 레이지 로딩이 생기기 때문에,

@api\_view()

def product\_list(request):

    queryset = Product.objects.select\_related('collection')

    serializers = ProductSerializer(queryset, many=True)

    return Response(serializers.data)

view에서 쿼리셋을 바꾸어 주어야 한다.

class CollectionSerializer(serializers.Serializer):

    id = serializers.IntegerField()

    title = serializers.CharField(max\_length=255)

collection = CollectionSerializer()

네스티드 오브젝트가 반환된다.

{

"id": 648,

"title": "7up Diet, 355 Ml",

"price": 79.07,

"price\_with\_tax": 86.98,

"collection": {

"id": 5,

"title": "Stationary"

}

},

collection = serializers.HyperlinkedRelatedField(

        queryset=Collection.objects.all(),

        view\_name='collection-detail'

    )

serializers.py 하이퍼링크를 반환하기 위한 코드

path('collections/<int:pk>/', views.collection\_detail, name='collection-detail')

urls.py 시리얼라이저에서 정한 뷰네임이 패스에 지정되어야 함, id가 아닌 pk를 변수로 사용하여야 함

@api\_view()

def product\_list(request):

    queryset = Product.objects.select\_related('collection')

    serializers = ProductSerializer(

        queryset, many=True, context={'request': request})

    return Response(serializers.data)

views.py 시리얼라이저를 초기화할 때, 컨텍스트가 추가로 전달되어야 함

@api\_view()

def collection\_detail(request, pk):

    collection = get\_object\_or\_404(Collection, pk=pk)

    serializers = CollectionSerializer(collection)

    return Response(serializers.data)

pk를 변수로 사용하여야 함

하이퍼링크가 반환된다.

{

"id": 1,

"title": "Bread Ww Cluster",

"price": 4.0,

"price\_with\_tax": 4.4,

"collection": "<http://localhost:8000/store/collections/2/>"

}

모델 시리얼라이저

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = ['id', 'title', 'unit\_price', 'collection']

모델을 참조하여 간편하게 시리얼라이저를 만드는 법

관계 시리얼라이저의 디폴트값은 프라이머리키이다.

class ProductSerializer(serializers.ModelSerializer):

    class Meta:

        model = Product

        fields = ['id', 'title', 'price', 'price\_with\_tax', 'collection']

    price = serializers.DecimalField(

        max\_digits=6, decimal\_places=2, source='unit\_price')

    price\_with\_tax = serializers.SerializerMethodField(

        method\_name='calculate\_tax')

    collection = serializers.HyperlinkedRelatedField(

        queryset=Collection.objects.all(),

        view\_name='collection-detail'

    )

    def calculate\_tax(self, product: Product):

        return round(product.unit\_price \* Decimal(1.1), 2)

이런 식으로 원래 시리얼라이저처럼 커스터마이징하는 것도 가능하다.

fields = '\_\_all\_\_'

을 사용할 수 있으나, 배드 프랙티스니까 사용하지 마라.

오브젝트 디시리어라이즈하기

시리얼라이즈된 제이슨으로부터 오브젝트만들기(클라이언트에서 전달받은 값을 변환하기)

@api\_view(['GET', 'POST'])

def product\_list(request):

    if request.method == 'GET':

        queryset = Product.objects.select\_related('collection')

        serializers = ProductSerializer(

            queryset, many=True, context={'request': request})

        return Response(serializers.data)

    elif request.method == 'POST':

        serializer = ProductSerializer(data=request.data)

        return Response('ok')

하나의 뷰 컨트롤러에서 조건문을 이용하여, 여러 개의 method를 동시 처리

브라우저블 API(browsable API)

데이터 검증하기

 elif request.method == 'POST':

        serializer = ProductSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.validated\_data

            return Response('ok')

        else:

            return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

모델 조건에 맞게 자동으로 밸리데이션을 해 줌

    elif request.method == 'POST':

        serializer = ProductSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.validated\_data

        return Response('ok')

간단히 이렇게 쓸 수 있음, 결과는 같음

모델 제한에 더하여 추가 밸리데이션이 필요한 경우

    def validate(self, data):

        if data['password'] != data['confirm\_password']:

            return serializers.ValidationError('Passwords do not match')

        return data

시리얼라이저에 밸리데이트 펑션을 오버라이딩 해 준다.

데이터 저장하기

serializer.is\_valid() 함수를 실행하고 난 뒤에는 serializer.validated\_data를 이용할 수 있다.

    elif request.method == 'POST':

        serializer = ProductSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        return Response(serializer.data, status=status.HTTP\_201\_CREATED)

그냥 저장만 할 거라면, serializer.validated\_data를 이용할 것도 없이, 그냥 save 펑션을 실행하면 된다.

전달받은 데이터에 추가적인 조작이 필요한 경우에는 다음과 같이 serializer의 create 펑션이나 update 펑션을 오버라이딩한다.

    def create(self, validated\_data):

        product = Product(\*\*validated\_data)

        product.other = 1

        product.save()

        return product

    def update(self, instance, validated\_data):

        instance.unit\_price = validated\_data.get('unit\_price')

        instance.save()

        return instance

업데이트는 이게 제대로 되는 건지 잘 모름

데이터 업데이트하기

product = get\_object\_or\_404(Product, pk=id)

    if request.method == 'GET':

        serializer = ProductSerializer(product, context={'request': request})

        return Response(serializer.data)

    elif request.method == 'PUT':

       serializer = ProductSerializer(product, data=request.data)

       serializer.is\_valid(raise\_exception=True)

       serializer.save()

       return Response(serializer.data)

데이터 삭제하기

@api\_view(['GET', 'PUT', 'DELETE'])

elif request.method == 'DELETE':

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

**httpstatuses.com** 참고하라

    elif request.method == 'DELETE':

        if product.orderitem\_set.count() > 0:

            return Response({'error': 'Product cannot be deleted'}, status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED)

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

class CollectionSerializer(serializers.ModelSerializer):

    class Meta:

        model = Collection

        fields = ['id', 'title', 'products\_count']

    products\_count = serializers.SerializerMethodField(

        method\_name='count\_products'

    )

    def count\_products(self, collection):

        return collection.product\_set.count()

product 카운트를 함께 내려보내는 코드(나의 코딩)

class CollectionSerializer(serializers.ModelSerializer):

    class Meta:

        model = Collection

        fields = ['id', 'title', 'products\_count']

    products\_count = serializers.IntegerField()

@api\_view(['GET', 'POST'])

def collection\_list(request):

    if request.method == 'GET':

        queryset = Collection.objects.annotate(products\_count=Count('product'))

        serializers = CollectionSerializer(queryset, many=True)

        return Response(serializers.data)

    elif request.method == 'POST':

        serializer = CollectionSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        return Response(serializer.data, status=status.HTTP\_201\_CREATED)

@api\_view(['GET', 'PUT', 'DELETE'])

def collection\_detail(request, pk):

    collection = get\_object\_or\_404(Collection.objects.annotate(

        products\_count=Count('product')), pk=pk)

이렇게 코등해도 동일한 결과를 가져옴(모쉬의 코딩)

내 방식이 더 간편하고 좋아 보이는데?