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))

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 사용한다.

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)

장고 Model field reference 문서 참조

아이디는 자동생성되지만, 아이디 이외의 기본키를 사용할 수도 있다.

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

입력값이 정해져 있는 필드는 다음과 같이 정의한다.

    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 정의하기

1. one to one 릴레이션

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)

삭제처리 선택옵션으로는 CASCADE, SET\_NULL, SET\_DEFAULT, PROTECT(차일드가 남아 있는 이상 페어런트를 삭제할 수 없다) 등이 있다.

차일드 모델에서 페어런트를 정의한 경우, 페어런트 모델에서 차일드를 정의할 필요 없다.

1. one to many 릴레이션

class Address(models.Model):

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

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

    customer = models.ForeignKey(

        Customer,

        on\_delete=models.CASCADE)

차일드 모델에서 포린키를 필드로 사용한다.

차일드 모델이 더 위에 있어서 페어런트 모델을 참조할 수 없을 때에는 'Customer'와 같이 스트링을 매개변수로 사용한다.

1. many to many 릴레이션

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

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)

    collection = models.ForeignKey(

        Collection, on\_delete=models.PROTECT)

    promotions = models.ManyToManyField(

        Promotion, related\_name='products')

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

class Collection(models.Model):

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

    featured\_product = models.ForeignKey(

        'Product',

        on\_delete=models.SET\_NULL,

        null=True,

        related\_name='+')

릴레이티드 네임을 지정하지 않으면, 네임클래쉬 에러가 발생한다. collection이라는 이름으로 리버스 쿼리를 작성할 수 없기 때문이다. 이 경우 릴레이티드 네임을 다른 것으로 지정하거나, '+'로 지정하여 리버스 쿼리를 만들지 않도록 할 수 있다.

제너릭 릴레이션

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

    # ID of object - find record

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

    object\_id = models.PositiveBigIntegerField()

    content\_object = GenericForeignKey()

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. 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.

* 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'

        """)

    ]

1. 마이그레이션을 실행한다.

python manage.py migrate

python manage.py sqlmigrate를 하면 db에 전송되는 sql코드를 실시간으로 볼 수 있다.

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

You have 17 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes.

Run 'python manage.py migrate' to apply them.

* vscode에서 sqlite익스텐션 사용하는 법

커맨드팔레트에서 sqlite: open database를 실행하고, 열 파일을 지정하면, 왼쪽 사이드바에 SQLITE EXPLORER가 생긴다.

1. 마이그레이션 취소하는 법

새로운 마이그레이션을 만들어서 마이그레이트한다.

python manage.py migrate store 0003

이동하고 싶은 마이그레이션 번호를 다시 실행한 후, model 파일 내의 해당 코드와 이후의 마이그레이션 파일을 삭제한다.

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)

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.

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

from django.db.models import F

query\_set = Product.objects.filter(

        inventory=F('unit\_price'))

query\_set = Product.objects.filter(

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

필드를 비교해야 할 필요가 있을 때는 F오브젝트 사용

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

소팅

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

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

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

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

query\_set = Product.objects.earliest('title')

query\_set = Product.objects.latest('title')

결과 제한하기

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

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

<ul>

      {% for collection in collections %}

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

      {% endfor %}

    </ul>

템플릿에 표시는 위와 같이 한다.

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

터플 형식으로 각 필드를 읽어온다.

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

<ul>

      {% for collection in collections %}

      <li>{{ collection.0 }} {{ collection.1 }} {{ collection.2 }}</li>

      {% endfor %}

    </ul>

튜플은 템플릿에 표시는 위와 같이 한다. 브래킷 노테이션이 아니라, 닷 노테이션

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

Product 클래스의 인스턴스 형식으로 각 필드를 읽어온다.

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

only의 반대, id와 title 필드를 빼고 읽어온다.

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)

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

위와 같은 경우가 예상되거나 필요할 때 미리 관련 쿼리를 로딩해 놓는다는 것인데, 이게 어떤 실용적인 의미가 있는지 모르겠다. 실용적인 의미가 있을지도 모르겠다(하나의 명령어로 관련 테이블까지 로드).

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

이와 같은 형식으로도 사용가능(해당 필드만 로드?)

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

product는 하나의 컬렉션에만 속하므로, select\_related를 사용하고, product는 여러개의 promotion과 관련될 수 있으므로, prefetch\_related를 사용한다.

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

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

처럼 체이닝하는 것도 가능, 순서 바꾸는 것도 가능(모두 프리로드)

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

query\_set = Product.objects.filter(

        id=F('orderitem\_\_product\_id')).distinct().order\_by('title')

query\_set = Product.objects.filter(

        id\_\_in=OrderItem.objects.values('product\_id').distinct()).order\_by('title')

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

admin 만들기

python manage.py createsuperuser 후 username, email, password 입력

비밀번호 리셋하기

python manage.py changepassword username

admin 화면 커스터마이즈하기

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

admin.site.index\_title = 'Admin'

장고의 앱들은 각각 자신의 admin.py를 갖는다. 관련 admin을 등록(register)하고 커스터마이징 할 수 있다.

앱의 모델 등록하기

from django.contrib import admin

from . import models

# Register your models here.

admin.site.register(models.Collection)

대표 표시 커스터마이징 하기(str펑션 오버라이딩)

class Collection(models.Model):

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

    featured\_product = models.ForeignKey(

        'Product',

        on\_delete=models.SET\_NULL,

        null=True,

        related\_name='+')

    def \_\_str\_\_(self) -> str:

        return self.title

class Meta:

        ordering = ['title']

내용까지 표시되게 커스터마이징 하기(위의 str 오버라이딩 필요 없음)

class ProductAdmin(admin.ModelAdmin):

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

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

아래와 같이 바꾸어 쓸 수 있음

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

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

@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

위와 같은 옵션들을 사용할 수 있고, ordering을 적으면 클래스의 meta서브클래스의 내용은 오버라이딩되는 것 같다.

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

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

    list\_editable = ['unit\_price']

    list\_per\_page = 10

@admin.display(ordering='inventory')

    def inventory\_status(self, product):

        if product.inventory < 10:

            return 'Low'

        return 'OK'

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

def collection\_title(self, product):

        return product.collection.title

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

list\_selected\_related = ['collection']

이 속성을 추가하여 eager로딩, pre로딩을 할 수 있다.