Arkadaşlar merhaba,

Bu videomuzda Django’da komutlarını ve model ilişkilerini anlatacağım.

-------------- commands --------------

Django built-in komutlara sahip olup, bunlardan bir kaçını daha önceki videolarda görmüştük. Komutları

**python manage.py <komut>**

şeklinde çalıştırıyoruz. Neler yaptığımızı hatırlayalım runserver, migrate, makemigrations, startapp, startproject, createsuperuser. Yeri geldiğinde başka komutlarda kullanacağız.

Tüm komutların listesine;

**python manage.py help**

Komutu ile ulaşabilirsiniz. Komutların detaylarına ise

**python manage.py help <komut>**

yazarak ulaşabilirsiniz. Ben bu konuda sizlere built-in komutları anlatmayacağım. Kendi komutlarımızı nasıl oluşturabileceğimizden bahsedeceğim.

Komutlar uygulamalarımız içerisinde management/commands/ klasörü içerisinde bulunur. Öncelikle klasörümüzü oluşturalım ve \_\_init\_\_.py mizi yükleyelim.

Şimdi komutu yazacağımız dosyayı oluşturabiliriz. (sample.py). Dosyamızı açalım ve Command classımızı yazmaya başlayabiliriz. Her komut Command sınıfına yazılır ve Command sınıfı BaseCommand dan türetilmelidir.  
BaseCommand import edelim ve ekrana ‘Command Çalıştı’ yazdıralım.

Help değişkenine yardım metni atanır. Command için yazılan kodlar handle() metodu içerisinde yazılır.

from django.core.management.base import BaseCommand   
  
class Command(BaseCommand):

help = 'Komut hakkında yardım bilgisi'

def handle(self**,** \*args**,** \*\*kwargs):  
 print('Command Çalıştı')

Komutu çalıştırmak için :

**python manage.py sample**

yazdığımızda. Ekrana ‘Command Çalıştı’ yazdığını görüyoruz.

Komutlar ile birlikte argümanlarda gönderebiliriz. Command sınıfında argüman eklemek için add\_arguments() metodu kullanılır.

Django argümanlar için Python’un argparse kütüphanesini kullanır. İki çeşit argüman vardır zorunlu ve isteğe bağlı argümanlar. Argüman başına tire ekleyerek isteğe bağlı yapabiliriz. Şimdi zorunlu ve isteğe bağlı argümanlar yazalım.

Argümanlar parser.add\_argument() metodu ile eklenmektedir.

Öncelikle zorunlu argümanlarımızı oluşturalım. 2 tane zorunlu argümanımız olsun. user\_id argümanı oluşturalım int olsun. yardım metnini yazalım. username argümanı oluşturalım string olsun. yardım metnini yazalım.

def add\_arguments(self**,** parser):  
 # Zorunlu argüman  
 parser.add\_argument('user\_id'**,** type=int**,** help='user\_id yardım bilgisi')  
 parser.add\_argument('username'**,** type=str**,** help='username yardım bilgisi')

3 tane de isteğe bağlı argüman oluşturalım. Prefix argümanımız olsun string olsun. isteğe bağlı yapmak için başına – koymamız gerektiğini söylemiştim. İki tane de koyabiliriz. Admin argümanı oluşturalım eğer girilirse true olarak atansın. Yardım bilgisini ekleyelim. Users argümanı girelim ve int seçelim. Yardım bilgisini ekleyelim. liste olarak gönderebilmek için nargs argümanını + olarak girelim. Kısaltma da ekleyebiliriz. Onun için ilk argüman olarak u girelim. Artık u ve users olarak argüman gönderebiliriz.

# İsteğe Başlı Argüman  
 parser.add\_argument('--prefix'**,** type=str**,** help='prefix yardım bilgisi')  
 parser.add\_argument('--admin'**,** action='store\_true'**,** help='admin yardım bilgisi')  
 parser.add\_argument('-u'**,** '--users'**,** nargs='+'**,** type=int**,** help='users yardım bilgisi')

Argümanlarımızı ekledik şimdi handle() metodunda gelen argümanları yakalayalım.

def handle(self**,** \*args**,** \*\*kwargs):  
  
 user\_id = kwargs['user\_id']  
 product\_id = kwargs['product\_id']  
 prefix = kwargs['prefix']  
 admin = kwargs['admin']  
 users = kwargs['users']  
  
 print('Positional user\_id : ' + str(user\_id))  
 print('Positional product\_id : ' + str(product\_id))  
  
 if prefix:  
 print('Optional prefix : ' + prefix)  
  
 if admin:  
 print('Optional admin selected')  
  
 if users:  
 print('Optional list : ' + str(users))

Şimdi komutumuzu çalıştırabiliriz.

**python manage.py sample 1 2 –prefix emre –admin –users 1 2 3 4 5**

Opsiyonel alanlardan sadece birtanesini yazalım

**python manage.py sample 1 2 –prefix emre**

Artık veritbanımıza çok ekleme çıkarmaya yapmaya başlayacağız. Kendimize bir veri datası hazırlayalım ve bunu istediğimiz zaman veritabanına yükleyebilelim. Bunun commandlar ile yapabiliriz.

**python manage.py populate**

komutunu çalıştırdığımızda hazırladığımız data veritabanına yüklensin.

Comands/ klasörüne populate dosyası oluşturalım.

BaseCommand ve veri ekleyeceğimiz Product, Category, User, UserDetail modellerini de import edelim ve sınfımızı ve handle() metodunu oluşturalım. Herhangi bir argüman göndermeyeceğiz sadece komut çalıştırınca handle() metodunu çalıştıracağız onun için add\_arguments() metodunu oluşturmamıza gerek yok.

Veritabanına verileri daha önce öğrendiğimiz metodlardan olan bulk\_create ike gerçekleştirelim. Hatırlarsanız bulk\_create veritabanında toplu veri eklemede kullanılıyordu. Normalde bizim Product modelimiz save() yapıldığında otomatik slug oluşturmaktadır. Fakat bulk metodları save() metodunu çağırmadan ekleme işlemini gerçekleştirmektedir. Bunun için bizim ayrıca slug vermemiz gerekmektedir.

from django.core.management.base import BaseCommand  
from learning.models import UserDetail**,** Product  
from django.contrib.auth.models import User

def handle(self**,** \*args**,** \*\*kwargs):

UserDetail.objects.delete()  
 User.objects.delete()  
 Product.objects.delete()

User.objects.bulk\_create([  
 ])  
  
 UserDetail.objects.bulk\_create([  
 ])  
  
 Product.objects.bulk\_create([])

Komutumuzu çalıştırarak tablolarımıza verilerimizi yükleyelim.

**python manage.py populate**

-------------- model ilişkileri --------------

Öncelikle size Veritabanı İlişkileri hakkında bilgi vermek istiyorum.

Veritabanı İlişkilerini kullanarak;

* Tekrarlanan verilerin azaltılması
* Veritabanının tutarlı olması
* İş yükünün azalması sağlanır.

Tablolar ilişkilendirilirken normalizasyon kuralları uygulanır. Genel olarak kullanılan 3 normal form vardır. Bire-bir ilişkiler 1. normal Form, Çoka- bir ilişkiler 2. normal Form, Çoka - çok ilişkiler ise 3. normal Form sınıfındadır. Normalizasyon ile ilgili detaylı bilgilere google dan normalizasyon yazarak bilgil alabilirsiniz.

-------------- bire – bir ilişki --------------

Bire – bir ilişki nedir?

Veritabanımızdaki Users tablosunu düşünelim. Users tablosunda kullanıcı üye olurken kullanıcı adı, şifre, email bilgilerini alıyor ve oluşturulduğunda otomatik primary\_key tanımlanıyor.

Kullanıcı bilgileri adında bir tablomuz daha olduğunu düşünelim bu tabloda da kullanıcının adresi, yaşı vb. detay bilgileri tutulsun.

Her kullanıcının sadece bir kullanıcı detayı. Her kullanıcı detayına karşılıkta sadece bir tane kullanıcısı olabilir. Her iki tablodan karşılıklı eşleşen bir kayıt varsa bunu bire – bir ilişki diyoruz.

Hatırlarsanız model kalıtımı konusunda yaptığımız Kitap – Intro örneği bire bir ilişki örneğiydi. Her kitabın sadece bir Intro su vardı. Bir Intro’da sadece bir kitaba ait olabiliyordu.

Yukarıda verdiğimiz kullanıcı örneğini koda dökerek daha iyi anlamaya çalışalım. Zaten Django’da Users tablosu var. Biz UserDetail tablosu oluşturarak bu iki tablo arasında bire – bir ilişki kuracağız. Kullanıcı detayı kullanıcılara bağlı olduğu için ilişkiyi UserDetail modelinde oluşturacağız. Model klasörümüze learning\_relations.py dosyası oluşturalım ve içine User modelimizi import edelim. UserDetail modelimizi oluşturalım ve adress alanı ekleyelim. Şimdi iki tablo arasında ilişki kurmamız gerekli Bire bir ilişkiler için OneToOneField alan türü kullanılmaktadır. İlk argüman olarak ilişki kurulacak Model yazıyoruz, on\_delete alanı ilişkili nesne silindiğinde bu nesneye ne olacağıdır. models.CASCADE yazarsak bu nesne de veritabanından silinir. models.SET\_NULL yaparsak ise alan silindiğinde user alanı NULL olarak işaretlenir. Fakat bunlar bire – bir ilişkili olduğu için birinin silinince diğerininde silinmesi mantıklıdır. Alanı primary\_key olarak işaretleyebiliriz. Çünkü her kullanıcı için sadece bir tane detay bilgisi olacak. İlişkili alanlarda veritabanında user olarak yazdığımız alan adı user\_id olarak kayıt edilir. user alanına User modelimizi ilişkilendirmiş olduk. Model içerisinden self.user, model dışından UserDetail.user ile artık User modeline ulaşabiliriz.

Son olarak \_\_str\_\_ metodumuzu tanımlayalım. Bağlı olduğu kullanıcının usernamini alalım ve detail ile birleştirelim. self.user

from django.db import models  
from django.contrib.auth.models import User  
  
  
class UserDetail(models.Model):  
 address = models.TextField()  
 user = models.OneToOneField(User**,** on\_delete=models.CASCADE**,** primary\_key=True)

def \_\_str\_\_(self):  
 return self.user.username + ' detail'

Modelimizi \_\_init\_\_.py dosyasından tanıtalım.

from .learning\_relations import UserDetail

Yeni modellerin veritabanına eklenebilmesi için migration dosyalarını oluşturalım ve değişiklikleri veritabanı ile eşleyelim.

**python manage.py makemigrations**

**python manage.py migrate**

Veritabanını açıp oluşturulan veritabanlarını kontrol edelim. Gördüğünüz gibi userdetail tablomuzu ekledi ve auth\_user.id alanı ile userdetail.user\_id ilişkisi kuruldu. Şimdi shellden birkaç kullanıcı ve detay bilgisi ekleyerek bu modeller ile sorgulamalar yapalım.

**python manage.py shell**

from django.contrib.auth.models import User  
from learning.models import UserDetail  
  
user1 = User(username=’user1’)  
user1.save()  
user2 = User(username =’user2)  
user2.save()  
  
 user1\_detail = UserDetail(user=user1**,** adress=’user **1** adresi’)  
 user1\_detail.save()  
  
 UserDetail.objects.all()

Kayıt işlemini model yöneticisi ile de yapabiliriz.

user2\_detail = UserDetail.objects.create(user=user2**,** adress=’user **2** adresi’)  
  
user1.userdetail  
user1\_detail.user  
  
User.objects.all()  
User.objects.get(userdetail=user1\_detail)  
User.objects.filter(userdetail\_\_address\_\_startswith="u")

Daha önceden eklenen kullanıcılar için UserDetail nesnesi eklemedik. Detail bilgisine ulaşınca neler olacak inceleyelim.

old\_user = User.objects.get(id=**1**)  
hasattr(user1**,** 'userdetail')  
hasattr(old\_user**,** 'userdetail')

-------------- çoka – bir ilişki --------------

Çoka – bir ilişki nedir?

Çoka bir ilişkilere daha önce oluşturduğumuz Product ve User modeliini verebiliriz. Kullanıcılar birden fazla ürün oluşturabilmektedir. Fakat her ürünün sadece bir tane sahibi vardır. Çoka bir ilişkiler Foreign Key ile kurulur. Hatırlarsanız Product modelinde User tablosu ile author alanını kullanarak ilişki kurmuştuk. Product modelini açalım ve kontrol edelim.

author = models.ForeignKey(User**,** on\_delete=models.CASCADE)

Buradada görüldüğü üzere Product modelinden self.author ve model dışından Product.author diyerek User modeline ulaşabiliriz. shelli açıp ürün ekleyelim ve ilişkileri kontrol edelim.

**python manage.py shell**

from django.contrib.auth.models import User  
from learning.models import Product

user1 kullanıcısı için 2 tane ürün ekleyelim, user2 kullanıcısı için 1 tane ürün ekleyelim.

user1 = User.objects.get(id=**3**)  
user2 = User.objects.get(id=**4**)   
product1 = Product.objects.create(name='product1'**,** author=user1)

**TypeError: save() got an unexpected keyword argument 'force\_insert'**

Force\_insert komutunu bulamadığından dolayı hata ile karşılaşıldı. Biz product metodunu override etmiştik oraya argüman olarak force\_insert girmemiz gerekiyor. Hemen düzeltelim ve shellimizi tekrar açalım.

def save(self**,** force\_insert=False**,** force\_update=False**,** using=None):

**python manage.py shell**

from django.contrib.auth.models import User  
from learning.models import Product  
   
user1 = User.objects.get(id=**3**)  
user2 = User.objects.get(id=**4**)   
product1 = Product.objects.create(name='product1'**,** author=user1)  
product2 = Product.objects.create(name='product2'**,** author=user1)  
product3 = Product.objects.create(name='product3'**,** author=user2)  
Product.objects.all()  
product1.author  
product1.author.username

Kullanıcının eklediği ürünleri listelemeye çalışalım. İlişki product modelinde bulunduğu için \_set ile ters ilişki yapabiliriz.

user1.product\_set.all()   
user\_product = user1.product\_set.first()  
user.product.name  
new\_product = user1.product\_set.create(name=’new product’)

product\_set yerine kendi atadığımız bir ismide kullanabiliriz. Bunun için ilişkili model alanına related\_name vermemiz gerekmektedir. related\_query\_name ise filter() metodunda kullanılır.

author = models.ForeignKey(User**,** on\_delete=models.CASCADE**,** related\_name='products'**,** related\_query\_name='product')

shell yeniden çalıştırarak ilişkilerimizi tekrar kontrol edelim.

**python manage.py shell**

from django.contrib.auth.models import User  
from learning.models import   
  
user1 = User.objects.get(id=**3**)  
product1 = Products.objects.get(id=**3**)  
product1.author.name  
user1.products.all()  
user\_product = user1.products.first()  
user\_product.name  
User.objects.filter(product\_\_name\_\_startswith='p') # related\_query\_name   
User.objects.filter(product\_\_name\_\_startswith='p').count()  
User.objects.filter(product\_\_name\_\_startswith='p').distinct().count()  
  
user2\_product = Product.objects.get(id=**3**)

Ürün sahibini değiştirelim.

user1.products.add(user2\_product)

Kullanıcıyı sildiğimizde ona bağlı ürünlerde silinir. Kullanıcıyı silip kontrol edelim.

user1.delete()  
Product.objects.all()

Eğer ürünlerin silinmesini istemiyorsak on\_delete argümanını SET\_NULL olarak tanımlamak lazım. Tabi SET\_NULL alabilmesi için alan özelliklerinden veritabanı için null=True olarak tanımlaması, validasyonlar için blank=True olarak tanımlanması gerekmektedir.

author = models.ForeignKey(User, on\_delete=models.SET\_NULL, null=True, blank=True, related\_name='products', related\_query\_name='product')

shell yeniden çalıştıralım ve kullanıcı silindiğinde ürünlerin silinip silinmediğini kontrol edelim.

**python manage.py shell**

from django.contrib.auth.models import User  
from learning.models import Product  
  
user2 = User.objects.get(id=**4**)  
User2.delete()  
Product.objects.all()

-------------- çoka – çok ilişki --------------

Çoka – çok ilişki nedir?

İki tablonunda bir birine karşı birden çok ilişkisi olan durumlardır. Bu ilişkileri tanımayabilmek için bir tane daha tabloya ihtiyaç duyulmaktadır. Bu tabloya ilişki tablosu denir. Biz iki tablo arasındaki bağlantıyı gösterdiğimizde Django ilişki tablosunu otomatik oluşturmaktadır. İki model arasında ilişki ManyToManyField ile kurılmakta olup, modellerden sadece bir tanesinde ilişki kurulmalıdır. İlişkinin hangi modelde kurulacağı önemli değildir.

Bir örnek ile açıklayalım. Ürün kategorilerimiz olduğunu ve ürünlerin birden fazla kategori alabildiğini düşünelim. Her kategoride de birden çok ürün olabilir. Böyle durumlarda çoka – çok ilişki kurulur.

Ürünlerimiz : Koltuk, Tencere, Çatal, Bıçak

Kategorilerimiz : Mutfak, Oturma Odaları, Yatak Odaları, Mobilyalar

Ürünlerimizi düşündüğümüzde Koltuk birden fazla kategoriye girebilmektedir. Mobilyalar ve Oturma Odaları

Kategorilerimizi düşündüğümüzde Mutfak’ta birden fazla ürün girebilmektedir. Tencere, Çatal, Bıçak.

Çoka – çok ilişki kurmak için learning\_relations.py’de Category modeli oluşturalım ve Product modelimizi import edelim.

Öncelikle Category modeline kategori adı için name alanını ekleyelim ve karakter sayısını 200 ile sınırlayalım. Product modeli ile ilişkimizi ve related\_name ile related\_query\_name oluşturalım. Bu alanların ne işe yaradıklarını çoka – bir ilişkilerde açıklamıştık.

from learning.models import Product

class Category(models.Model):  
  
 product = models.ManyToManyField(Product, related\_name='categories', related\_query\_name='category')

Modelimizi \_\_init\_\_.py dosyasından tanıtalım.

from .learning\_relations import UserDetail, Category

Daha önce oluşturduğumuz Populate komutu ile kategorilerin oluşturulmasını da sağlayalım.

Category.objects.delete()  
  
 Category.objects.bulk\_create([  
 ])

Populate komutumuzu çalıştıralım.

**python manage.py populate**

**python manage.py Shell**

from learning.models import Product  
from learning.models import Category

category.products.add(p1)  
category.products.add(p1**,** p2)  
category.products.all()  
  
product.categories.all()  
Product.objects.get(id=**1**).categories.all()  
  
Product.objects.filter(categories\_\_id=**1**)  
Categories.objects.filter(products\_\_id=**1**)  
  
Product.objects.filter(categories\_\_name\_\_startswith=’y’).count()  
Product.objects.filter(categories\_\_name\_\_startswith=’y’).distinct().count()  
Categories.objects.filter(products\_\_name\_\_startswith=’k’).count()  
Categories.objects.filter(products\_\_name\_\_startswith=’k’). distinct ().count()  
  
Product.objects.filter(id=**1**).delete() # ilişkili alanlarda silinir.  
category.products.remove(p1)  
category.products.set([p1])  
category.products.clear()

-------------- select\_related & prefect\_related --------------

Proje geliştirirken dikkat etmemiz en önemli konulardan bir tanesi sistemin performanslı çalışmasıdır. Veritabanlarından yapılan sorgular performanslarının azaltılması en önemli performans kalemlerinden biridir. Sorgu sayısının azalması da doğal olarak performansın artmasını sağlayacaktır.

Django ilişkileri büyük veritabanlarında toplu sorgulamalarda performans sorununa neden olabilir. select\_related ve prefetch\_related sorgu sayılarının azaltılmasına yardımcı olur. select\_related bire – bir ve çoka – bir ilişkiler için kullanılır. prefetch\_related ise bire – çok ve çoka – çok ilişkiler için kullanılır.

Örnekler üzerinden incelersek daha iyi anlayabiliriz.

class User(models.Model):  
 pass

class UserDetail(models.Model):  
 user = models.OneToOneField(User**,** on\_delete=models.CASCADE**,** primary\_key=True)

class Product(models.Model):  
 author = models.ForeignKey(User**,** on\_delete=models.SET\_NULL**,** null=True**,** blank=True**,** related\_name='products'**,** related\_query\_name='product')

class Category(models.Model):  
 products = models.ManyToManyField(Product**,** related\_name='categories'**,** related\_query\_name='category')

Product – User -> çoka – bir, many to one

User – Products -> bire – çok, one to many (reverse relation)

Products – Category -> çoka – çok, many to many

User – UserDetail -> bire – bir, one to one

Veritabanında kayıtlı 20 tane user ve 100 tane ürün olduğunu varsayarsak.

1 adet ürün için bir adet author için sorgu yapar.  
toplam: 2 sorgu

product = Product.objects.first()  
print(product.author.name)

ürünün sahibini bir sorguda çeker

product = Product.objects.select\_related('author').first()  
print(product.author.name)

1 adet ürün listesini getirmek için sorgu yapar  
her kullanıcı için bir adet sorgu yapar.  
toplam : 21 sorgu

products = Product.objects.all()  
for p in products:  
 print(p.author.name)

Bir sorguda tüm ürünleri ve bunların sahiplerini çeker

products = Product.objects.select\_related('author').all()  
for p in products:  
 print(p.author.name)

1 adet kullanıcı listesini getirmek için sorgu yapar  
her ürün için bir adet sorgu yapar  
toplam: 101 sorgu

users = User.objects.all()  
for u in users:  
 print(u.products)

1 adet kullanıcı listesini getirmek için sorgu yapar  
1 adet ürün listesini getirmek için sorgu yapar.  
İlişkileri python kodu ile gerçekleştirir.

users = User.objects.prefetch\_related('products').all()  
for u in users:  
 print(u.products)

prefect\_related kullanılırken filter(), exists(), count(), first() vb methodlar fazladan sorgu atmasına neden olur.

category = Category.objects.prefetch\_related('products').get(pk=**1**)  
  
if category.products.exists(): # Burada fazladan sorgu atıyor

for product in category.products.all():  
 print(product.name)

Bunun yerine şu şekilde kullanım daha doğru olur.

category = Category.objects.prefetch\_related('products').get(pk=**1**)  
  
if category.products.all():  
 for product in category.products.all():  
 print(product.name)

-------------- Prefetch --------------

Prefect nesnesi prefetch\_related methodunu daha kullanışlı hale getirmektedir. prefetch\_related bize direk alan üzerinden ilişki kurmamızı sağlar fakat filtreleme yapamayız. **Product.objects.only(‘name’).all()** gibi sadece name alanını prefetch\_related ile çekemeyiz. Bu gibi durumlarda Prefetch nesnesinin yardımına ihtiyaç duyuyoruz. Argüman olarak sorgu seti girmemizi sağlar.

Prefect kullanmak için models kütüphanesinden import etmemiz gerekmektedir.

from django.db.models import Prefect

Category.objects.prefetch\_related(  
 Prefetch('products'**,** queryset=Product.objects.only('name').all())  
) .all()

Mesela kategorileri, bunlara bağlı ürünleri ve bunların sahiplerini çekmek istiyoruz.

1 adet kullanıcı listesini getirmek için sorgu yapar  
1 adet ürün listesini getirmek için sorgu yapar.  
1 adet kategori listesini getirmek için sorgu yapar.  
İlişkileri python kodu ile gerçekleştirir.

Category.objects.prefetch\_related(‘products\_\_author’).all()

Aynı kodu Prefect metodu ile çekersek 2 sorguda çekebiliriz.

1 sorgu kategori listesi için  
1 sorgu ürün listesi için ürün sahipleri select\_related ile tek sorguda getirilir.

Category.objects.prefetch\_related(  
 Prefetch('products'**,** queryset=Product.objects.select\_related('author'))  
) .all()

Bu sefer bir önceki örnekten farklı olarak to\_attr argümanını kullanalım. 'products' tan gelen verileri queryset argümanı ile filtreleyerek active\_products atar.

categories = Category.objects.prefetch\_related(  
 Prefetch('products'**,** queryset=Product.objects.only('name').all(), to\_attr=active\_products)  
) .all()

for category in categories:  
 category.active\_products

-------------- polimorfosizm --------------

<http://ses4j.github.io/2015/11/23/optimizing-slow-django-rest-framework-performance/>

<https://dizballanze.com/django-project-optimization-part-2/>