Arkadaşlar merhaba,

Bu videomuzda ise model metodları, model yöneticileri ve model kalıtımını öğreneceğiz.

--------------- model metodları ---------------

Daha önceki konulara Product modeline \_\_str\_\_ methodu yazarak modellerde yazılabileceğine değinmiştim. \_\_str\_\_ methodu bize nesnenin bir text karşılığı olarak gösterilmesini sağlar. Django’da özellikle yönetici panelinde kullanılmasından dolayı her modelde tanımlamamız gerekmektedir.

def \_\_str\_\_(self):  
 return self.name

Kullanabileceğimiz diğer bir method ise get\_absolute\_url metodudur. Bu method ile nesnelerimizin adreslemesini yapabiliriz. Bu method ile oluşturulan linkler yönetici panelinde de gözükür ve yönetici panelinden nesne sayfasına ulaşmamız için bağlantı ekler.

def get\_absolute\_url(self):  
 return "/product/%i/" % self.id

Djangoda nesne ekleme ve güncelleme işlemleri için modelin save() methodu, silme işlemlerinde ise delete() methodu çağırılır. Biz model içerisinde save() ve delete() methodlarını override ederek farklı işlemler yaptırabiliriz.

Bir örnek ile inceleyelim.

Bizim modelimizde benzersiz bir slug alanımız olsun. Nesne eklendiğinde biz bu slug alanını otomatik olarak nesne adından oluşturalım. editable parametresini False seçerek Yönetim Paneli ve Formlarda görüntülenmesini engelleyelim.

slug = models.SlugField(unique=True**,** editable=False)

Daha önce ürünler eklendiği ve slug alanının unique olması nedeniyle migrationda sorunlar oluşacaktır. Onun için öncelikle veritabanını boşaltalım. Daha sonra yeniden ürün eklemesi yaparız. Yeni oluşturduğumuz alanın veritabanına eklenebilmesi için migration dosyalarını oluşturalım ve değişiklikleri veritabanı ile eşleyelim.

**python manage.py makemigrations**

**python manage.py migrate**

Otomatik slug oluşturmak için slugify methodundan yararlanabiliriz, Bu method djangonun defaultfilters kütüphanesinde bulunmaktadır, methodumuzu import edelim.

from django.template.defaultfilters import slugify

save() methodu çalıştığında biz diyeceğizki eğer nesne ekleniyorsa nesnenin name alanını slugify methodu ile slug olarak kayıt edilebilecek metne çevir ve bizim slug alanımıza ata.

def save(self):  
 if not self.id:  
 self.slug = slugify(self.name)

Evet name alanından biz şu anda slugımızı oluşturduk. Fakat kontrol etmemiz gerekli bir şey daha var. Bizim slug alanımız benzersiz bir alan. Eğer bu slug veritabanında başka bir nesnede varsa ne olacak ? Hata vermemesi için daha önce olup / olmadığına bakmalı varsa değiştirmeliyiz. Değiştirme yöntemi olarakta slug sonuna numara ekleyelim.

Ürün adımız kitap olsun. Eğer veritabanında slug olarak kitap yoksa bu şekilde ekleyecek. Eğer varsa kitap-1 varmı diye bakacak. Eğer yoksa kitap-1 ekleyecek. kitap-1 varsa kitap-2 diye arayacak ve bulamayana kadar devam edecek. En son bulamadığında veritabanına ekleme işlemini gerçekleştirecek.

Bunun için bir döngü kullanmamız gerekli. Birden başlayan ve sonlandırana kadar slug\_id yi bir tane artıran counter oluşturalım. Bu counterı pythonun itertools kütüphanesi ile oluşturabiliriz.

Daha sonra slugımızın veritabanında olup olmadığını kontrol edelim. Kontrol etmek için bir sql sorgusu yazmamız gerekli. Product içerisinde slugımızı bulamaz ise break; ile counterı durduralım.

Eğer bulursa counter durmadığı için devam edecek ve self.slug’un sonuna yeni id mizi atayacağız ve counter 1 artırıp yeniden aynı sorguyu yapacağız taki veritabanında bulamayıp break çalışana kadar. Break çalıştığı zaman for döngüsünün içerisinden çıkacak ve super methodumuz çalışacaktır. super methodu ile base modelimizin save() methodunu çağırıyoruz.

for slug\_id in itertools.count(**1**):  
 if not Product.objects.filter(slug=self.slug).exists():  
 break  
 self.slug = '%s-%d' % (self.slug**,** slug\_id)  
  
super(Product**,** self).save()

Biz burda save() methodunu override ederek base modelimizin save() methodu çalışmadan önce slug alanına manuel olarak atama yapmış olduk. delete() metodunu da aynı şekilde override edebiliriz. Mesela delete() olmadan önce başka tablolarda güncelleme yapabilirsiniz veya birşeyler silebilirsiniz.

Modelin save() methodunu biraz detaylandıralım.

Bizim bir tane save() metodumuz var. Fakat bu save metodu ile hem ekleme hem de güncelleme yapailiyoruz. Bu nasıl olabilir ?

Biraz önce slug eklemesi yaparken bunu kullandık aslında. Eğer nesne primary\_key sahipse güncelleme değilse ekleme yapılır. Biz sadece eklerken kullanmıştık. Bu da primary\_key olan id nin var olmadığı durum oluyor.

if not self.id:  
 self.slug = slugify(self.name)

save() metodu çağırıldığında hangi işlemler yapılır?

* pre\_save sinyali gönderilir ve bu sinyali dinleyen fonksiyonlar çalıştırılır.
* her alan için pre\_save() methodu çağırılır ve alanlar için düzeltilmesi gerekli veriler düzeltilir. örneğin tarih alanlarında parametre olarak auto\_now\_add ekli ise tarih alanı ekleme zamanı ile güncellenir.
* her alanın get\_db\_prep\_save() methodu ile veritabanına yazılabilecek bir veri türüne çevrilmesi sağlanır. integer ve text alanları için zaten yazılabilir durumdadır. fakat veritabanları datetime nesneler kayıt etmezler. bunların veritabanına kayıt edilebilecek veri türüne değiştirilir.
* veriler veritabanına eklenir.
* post\_save sinyali gönderilir ve bu sinyali dinleyen fonksiyonlar çağırılır.

Nasıl sinyal oluşturacağımız ve sinyalleri dinleyeceğimizi ileriki konularda detaylandıracağım.

Modellerde kendi methodlarımızı da oluşturabiliriz. Product modelimizde content alanımız var. Bu alan uzun yazı alabiliyor.

content = models.TextField(verbose\_name='Ürün Açıklaması')

Ürün listeleme sayfasında content alanının hepsini kullanamayız. 50 karakter ile sınırlayacak bir summary methodu oluşturalım.

def summary(self):  
 return self.content[:**50**]

Django shell açıp summary alanına bakalım. Django summary yaptığımızda string olarak dönmüyor. Method olarak algılıyor. Product.summary ile ulaşabilmeyi @property dekoratörü ile sağlayabiliriz.

@property  
def summary(self):  
 return self.content[:**50**]

Şimdi metodumuz için bir tanede active alanı oluşturalım ürünün aktif/pasif olduğu bilgisini tutalım. Bunu BooleanField ile oluşturabiliriz. BooleanField ler True/False ve None alan tipi alıyordu. Default olarakta True seçili olsun, yani aktif olarak ekleyelim.

active = models.BooleanField(default=True)

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

**python manage.py makemigrations**

**python manage.py migrate**

Şimdi biz modelimize öyle bir method yazalımki bize aktif olan ürünleri filtrelesin. Bunun için sınıfımıza model içerisinden ulaşabilmemiz gerekli. Bu tür sınıfın kendisi ile ilgili işlem yapılacak metodlar için @classmethod dekoratörünü kullanıyoruz.

@classmethod  
def active\_products(cls):  
 return cls.objects.filter(active=**1**)

Eğer metodun nesne ve sınıf ile ilgisi yoksa @staticmethod olarak tanımlanır. static\_summary metodu dışardan parametre alıyor ve onu 50 karakterle sınırlandırıp geri gönderiyor.

@staticmethod  
def static\_summary(content):  
 return content[:**50**]

Şimdi modellerde kullanabileceğimiz diğer metodlara bakalım bunların ayrıntısına girmeyeceğim şimdilik ne yaptıklarını bilmemiz yeterli.

from\_db methodu ile veritabanından veriler çekilirken nesneyi düzenleyebilirsiniz.

refresh\_from\_db metodu veritabanındaki verilerin güncellenmesini sağlar. Veritabanındaki bir alanı güncelledikten sonra nesnemizin yenilenmesini sağlayabiliriz.

get\_deferred\_fields methodu ile modelden çıkartılan alanların listesini alabilir.

clean() ve full\_clean() methodu var. Buna validasyon kısmında değineceğiz.

--------------- model yöneticisi ---------------

Model Yöneticisi (Manager), Django’da veritabanı sorgulama işlemleri gerçekleştirmek için sağlanan arayüzdür. Daha önce her modelin en az bir yöneticisi olduğundan bahsetmiştik. Django varsıyan olarak model yöneticisi olarak objects alan adını kullanır.

**Product.objects**

Objects kullanmak istemiyorsak Model’den değiştirebiliriz.

class Product(models.Model):  
  
 items = models.Manager()

**product.items.all()**

**product.objects.all()**  # hata ile karşılaşırız.

Modeller için birden fazla yönetici kullanabileceğimiz gibi varsayılan yöneticiyi override da edebiliriz. İlk olarak override edip daha önce Model’de oluşturduğumuz active\_products metodunu model yöneticisine alalım. Active\_productsu @classmethod’a örnek olması açısından model içerisinde yazmıştık normalde model yöneticisinde bulunmalıdır.

class ProductManager(models.Manager):  
 def active\_products(self):  
 return self.filter(active=**1**)

**Products.objects.active\_products()**

Model için birden fazla model yöneticisi de tanımlayabiliriz. Modele iki tane daha yönetici ekleyelim bunlardan bir tanesi ActiveProductManager diğerde PassiveProductManager olsun.

class Product(models.Model):  
  
 objects = ProductManager()  
 actives = ActiveProductManager()  
 passives = PassiveProductManager()

**Product.objects.all()** # tüm nesneler listelenir

**Product.actives.all()** # aktif nesneler listelenir

**Product.passives.all()** # pasif nesneler listelenir

--------------- model kalıtımı ---------------

Django’da üç farklı şekikde modeller türetilebilir.

* Alt modeller için yazmak istemediğimiz bilgileri tutmak için ana sınıfı kullanmak. Bu tarz kalıtımlarda ana sınıf için veritabanında tablo oluşturulmaz. (model kalıtımı)
* Hiyerarşideki her modelin veritabanında tabloya sahip olduğu yapılar. Bu tarz kalıtımlarda ana model ile alt modeller arasında bire-bir ilişki kurulur. (çoklu tablo kalıtımı)
* Modelin vaysayılan yöneticisi gibi özelliklerini değiştirmek veya yeni yöntem eklemek için kullanılan yapılar. (vekil model)

Bir modeli ana sınıf gibi kullanıp sadece özelliklerini alt sınıflarına aktarmak istiyorsak o modelin meta ayarlarında sınıfı abstract olarak tanımlamalıyız. Bu sayede o model için veritabanında tablo oluşturulmaz.

Öncelikle yeni bir model oluşturalım. Modelimizin ismi CompanyGroup olsun. Bunun için model klasörünün altında bir adet learning\_inheritance.py dosyası oluşturuyorum. Daha önce söylediğim gibi modellerimiz Model sınıfndan türetilmek zorundadır.

from django.db import models  
  
  
class Company (models.Model):  
 pass

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

from .learning\_inheritance import Company

Bu modelimizin bizim ana sınıfımız olsun. Bunu tanıtmak için Meta ayarlarına abstrat=True eklemeliyiz. Bunu ekleyerek Model sınıfına bunun bir ana sınıf olduğunu bunun için veritabanında tablo oluşturmaması gerektiğini belirtmiş olduk.

class Meta:  
 abstract = True

Şimdi alanlarımızı oluşturalım. Şirketlerin isimleri olsun. CharField olarak tanımlayalım 200 karakter sınırı koyalım. Firma bilgileri için content alanı oluşturalım. Text Field olarak tanımlayalım. Uzun yazı girilebilsin. Şirketlerin sahibi olsun ve bu sahip sitemize üye olması gereksin. Foroeign key ile Kullanıcı tablosuna ilişki kuralım. Bunun için ilişki tablosunu import etmemiz gerekli. Son olarak oluşturulma tarihi ekleyelim. auto\_now\_add = True vererek kayıt eklendiğinde alanın otomatik doldurulmasını sağlayalım.

name = models.CharField(max\_length=**200**)  
content = models.TextField()  
author = models.ForeignKey(User**,** on\_delete=models.CASCADE)  
created = models.DateTimeField(auto\_now\_add=True)

from django.contrib.auth.models import User

Şimdi alt sınıflarımızı oluşturalım. 2 tane alt sınıfımız olsun bunlardan birtanesi BookCompany diğeri ise GameCompany.

class BookCompany(Company):  
 pass  
  
  
class GameCompany(Company):  
 pass

Bu iki alt sınıfımızda Company tablosunda ekleyeceğimiz aynı alanlara sahip olacağı için Company sınıfından türetelim. Yani bu name, content, author ve created alanları artık bu modellerimizde geçerli oldu. Tekrardan yazmamıza gerek kalmadı. Biz alt sınıflarımıza özel alanlarımızı ekleyebiliriz. Örnek olarak BookCompany için publisher\_name, GameCompany için ise platform olaranları ekleyelim.

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

from .learning\_inheritance import Company, BookCompany, GameCompany

Yeni oluşturduğumuz 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. BookCompany ve GameCompany modelleri için tablolar oluşturulduğunu ve Company için abstract sınıf olduğu için tablo oluşturulmadığını görüyoruz.

Model kalıtımında dikkat etmemiz gereken önemli bi kural ilişki alanlarında related\_name ve related\_query\_name kullanımıdır. Django bu alanların tüm proje dahilinde benzersiz olmasını zorunlı kılmaktadır. Eğer related\_name olarak authors kullanırsak bu alt modellereden aynı şekilde miras alınacak ve hata ile sonuçlanmasına neden olacaktır.

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

Böyle durumlar için bu alanları dinamik olarak vermemiz gerekir. Uygulama ve model adını kullanarak bunu benzersiz hale getirebiliriz. Bu şekilde kullandığımızda learnings\_bookcompanys\_related, learnings\_gamecompanys\_related şeklinde olacaktır.

author = models.ForeignKey(User**,** on\_delete=models.CASCADE**,** related\_name="%(app\_label)s\_%(class)s\_related"  
 )

Meta ayarlarını da miras şeklinde alt sınıflarımıza devredebiliriz. Bu şekilde kullandığımızda ana sınıfımızın meta ayarlarında yer alan abstract=True alanı miras alınsa bile False olarak davranır.

class Meta(Company.Meta):  
 pass

Eğer alt sınıfı da abstract sınıf olarak tanımayacaksanız tekardan abstract=True yazmanız gerekir.

class Meta(Company.Meta):  
 abstract = True

Alt sınıfımızın meta ayarlarında db\_table ekleyelim.

class Meta(Company.Meta):  
 db\_table = 'company\_alt\_sinifi\_book'

Şu anda sınıfımızın meta ayarlarında sadece db\_table gözükmesine rağmen üst sınıftan miras aldığı için ordering=[‘name’] özelliğine de sahiptir.

Şimdi hiyerarşideki tüm modellerin tabloya sahip olduğu durumu inceleyelim. Bir Book modeli ve Intro modeli oluşturalım. Intro, kitaplardaki ilk bölümdeki kısım olsun. Kitap sınıfımıza name ve page\_count alanlarını ekleyelim. Intro sınıfımızı Book sınıfından türetelim. Üst sınıfta dikkat ederseniz abstract=True ifadesini yazmadık. Book sınıfını abstract sınıf olmadı için alanları Intro sınıfımız için geçerli olmazç Ayrıca Book modelimiz ve Intro modelimiz için veritabanında farklı tablolar oluşturulacaktır. Django bu iki sınıfı bire-bir ilişki ile bağlayacaktır. İlişkiler kısımında bir sınıftan diğer sınıf ile nasıl ilişki kurulacağını anlatacağım.

class Book(models.Model):  
 name = models.CharField(max\_length=**200**)  
 page\_count = models.PositiveSmallIntegerField()  
  
  
class Intro(Book):  
 content = models.TextField()

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

from .learning\_inheritance import Company, BookCompany, GameCompany, Book, Intro

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. Book ve Intro modelleri için tablolar oluşturulduğunu görüyoruz. Çünkü iki sınıfta abstract sınıf değiller.

Modellerde anlatacağımız son kalıtım türü Vekil Sınıflardır Proxy Models diye geçiyor. Bu sınıflar bize ana sınıfımızı değiştirmeden model üzerinde değişiklik yapmamıza olanak verir. Modelin vekil model olduğunu belirtmek için meta ayarlarında proxy = True diye belirtiriz. Belirtilen model için veritabanında ayrıca tablo oluşturulmaz.

Biraz önce oluşturduğumuz Book modeli için bir vekil model oluşturalım. Model ismimizi ProxyBook olarak oluşturalım. Book sınıfını miras alsın. Meta ayarlarında vekil sınıf olduğunu belirtelim. Sıralamasını isme göre ayarlatalım ve yeni bir property ekleyelim. short\_name bize name alanını 10 harf ile sınırlandırarak versin. Vekil sınıf sayesinde ana modelimizi değiştirmeden ondan vekil bir model yaratarak farklı şekilde davranmasını sağladık.

class ProxyBook(Book):  
 class Meta:  
 proxy = True  
 ordering = ['name']  
   
 @property  
 def short\_name(self):  
 return self.name[:**50**]

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

from .learning\_inheritance import Company, BookCompany, GameCompany, Book, Intro, ProxyBook

Yeni modellerin veritabanına eklenebilmesi için migration dosyalarını oluşturalım.

**python manage.py makemigrations**

ProxyBook modeli vekil sınıf olduğu için hiçbir değişiklik bulunamadı.

Vekil sınıfımızın model yöneticisinide değiştirebiliriz. Yeni bir NewManager adında Model yöneticisi oluşturalım ve vekil modelimizde tanıtalım.

class NewManager(models.Manager):  
 # yapılacak işlemler  
 pass

class ProxyBook(Book):  
   
 objects = NewManager()  
   
 class Meta:  
 proxy = True  
 ordering = ['name']  
  
 @property  
 def short\_name(self):  
 return self.name[:**10**]

Kalıtım konusunda son olarak çoklu kalıtımdan bahsedelim. Bir sınıfımız birden fazla sınıftan miras alabilir fakat bu çok kullanılan bir durum değildir. Ama yinede biz örneğini verelim. Örneğimizi daha önce oluşturduğumuz Book sınıfımızdan verelim. Intro sınıfı Book sınıfından türüyordu. Birde intro yorumları için Comment sınıfı tanımlayam ve alan olarak content tanımlayalım. Hem Comment sınıfında hemde Book sınıfında primary\_key id olduğu için Intro sınıfında çakışma olacaktır onun için her iki sınıfta primary\_keyleri manuel olarak verelim.

class Comment(models.Model):  
 comment\_id = models.AutoField(primary\_key=True)  
 content = models.TextField()

class Book(models.Model):  
 book\_id = models.AutoField(primary\_key=True)  
 name = models.CharField(max\_length=**200**)  
 page\_count = models.PositiveSmallIntegerField()

Intro sınıfımızı Book ve Comment sınıfından türetelim.

class Intro(Book**,** Comment):  
 content = models.TextField()

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

from .learning\_inheritance import Company, BookCompany, GameCompany, Book, Intro, ProxyBook, Comment

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 Comment modeli için tablo oluşturulduğunu görüyoruz. Çünkü sınıfımız abstract sınıf değildir.

Bu videomuzda model metodları, yöneticileri ve kalıtım hakkında bilgiler öğrendik. Bir sonraki videomuzda Model ilişkileri ile devam edeceğiz.