Arkadaşlar merhaba,

Bu videomuzda ise model metodları ve model yöneticilerini 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/detail/%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 argümanını 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 argüman 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 bulunması daha doğru olur.

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