Arkadaşlar merhaba,

------------------ META AYARLARI ------------------

Meta ayarlarının modellerde class Meta: olarak kullanacağını önceki videolarımızda öğrenmiş alabileceği bazı ayarları uygulamıştık.

**abstract** :

Daha önce model kalıtımında kullanmıştık. True olduğunda modelimiz ana sınıf oluyor ve bu model için veritabanına tablo oluşturulmuyor.

class Meta:  
 abstract = True

**app\_label :**

eğer modelimiz uygulama içerisinde değilse hangi uygulamaya ait olduğunu belirtmek için kullanırız.

class Meta:  
 app\_label = 'learning'

**db\_table:**

veritabanına oluşturulacak tablo adını değiştirmek için kullanılır. Daha önce örneklerimizde kullanmıştık.

class Meta:  
 db\_table = 'products'

**base\_manager\_name**:

Modelde kullanılacak base manager adı.

class Meta:  
 default\_manager\_name = 'base\_objects'

**default\_manager\_name:**

Modelde kullanılacak default manager adı. Varsayılan olarak objects dir.

class Meta:  
 default\_manager\_name = 'default\_objects'

**default\_related\_name:**

varsayılan olarak <model adı>\_set ‘tir. Reverse relation için kullanılmaktadır.

class Meta:  
 default\_related\_name = 'reverse\_relationship\_name\_here'

**get\_latest\_by:**

latest() ve earlist() metodları için kullanılacak sıralama ayarları için kullanılır.

class Meta:  
 get\_latest\_by = ['name']

**managed:**

varsayılan olarak true seçilidir. Eğer false seçilirse bu model için veriabanına tablo ekleme, silme işlemleri yapılmaz. Migration işlemlerinden muaf olur.

class Meta:

managed = False

**order\_with\_respect\_to:**

Verilen alana göre sıralama yapılmasına olanak sağlar. Genellikle ForeignKey alanları için kullanılır.

class Product(models.Model):  
  
 author = models.ForeignKey(User**,** on\_delete=models.CASCADE)   
  
 class Meta:  
 order\_with\_respect\_to='author'

get\_product\_order() ile sıralaması alınır. get\_product\_set() ile de sıralama ayarları yapılır. Product nesnemizin adıdır. Sıralama yapılacak alanın modeline göre değişir. Sıralaramalar veritabanında \_order isminde bir alanda saklanır. Meta bilgisi ekledikten ve sildikten sonra migration yapmak gereklidir.

user = User.objects.get(id=**1**)  
user.get\_product\_order()  
# [1, 2, 3]

user.set\_product\_order([**3,1,2**])

İlişkili nesne içinde iki adet kullanılabilir metod vardır. get\_next\_in\_order() ile bir sonraki kayıt, get\_previous\_in\_order() ile bir önceki kayıt seçilir.

product = Product.objects.get(id=**2**)  
product.get\_next\_in\_order()  
# < Product 3>  
product.get\_previous\_in\_order()

# < Product 1>

ordering ve order\_respect\_to beraber kullanılamaz.

**ordering**

Nesne için verilan alana göre sıralama sağlar. Varsayılan olarak azalan sıralama yapar. Artan sıralama için başına – işareti konulur. Rasgele sıralama yapmak için ? kullanılır.

ordering = ['name']  
  
ordering = ['-name']  
  
ordering = ['-name'**,** 'author']

F() gibi sorgu ifadelerini kullanılabilir isme göre artan sırala ve boş kayıtları en son sırala.

ordering = [F('name').asc(nulls\_last=True)]

**default\_permissions**

Django ile kullanılabilecek varsayılan izinlerin ayarlandığı bölümdür. Varsayılan olarak ('add', 'change', 'delete', 'view') dir. Eğer listeyi boş bırakırsanız modelin hiçbir izne gerek duymaz.

**permissions**

modele varsayılan izinler harici izin girmek için kullanılır. Tupple şeklinde kullanılır (izin adı, izin detayı)

permissions = (  
 ("can\_edit\_product"**,** "Ürün güncelleyebilir")**,** ("can\_attach\_product"**,** "Ürün ilişkilendirebilir")**,**)

**proxy**

Daha önce model kalıtımında kullanmıştık. True olduğunda modelimiz proxy sınıf oluyor ve bu model için veritabanına tablo oluşturulmuyor.

**indexes**

İndexler genel olarak where cümleciği içeren select, update ve delete sorgularında ciddi performans artışları sağlarlar. insert sorgularında ve indexli alanalar üzerinde yapılacak update ve delete sorgularında ise performance düşürücü bir etkiye sahiptirler.

Genelde where sorgusunda kullanılacak alanlar için indexleme yapılır. Djangoda Meta ayarları altında indexleme yapabiliriz.

first\_name alanı için index oluşturduk ve isim olarak first\_name\_idx verdik. Eğer isim vermezsek django kendisi oluşturur. İki alan için birlikte de index oluşturabiliriz. First\_name ve last\_name için birlikte index oluşturmuş olduk. Bunu unit\_together ile de yapabilirdik.

class Meta:  
 indexes = [  
 models.Index(fields=['first\_name']**,** name='first\_name\_idx')**,**

models.Index(fields=['first\_name', 'last\_name'])**,** ]

**index\_together**

Birden fazla alan için index oluşturmamızı sağlar.

index\_together = ["first\_name"**,** "last\_name"]

Eğer birden fazla oluşturacaksak;

index \_together = (  
 ["first\_name"**,** "last\_name"]**,** ["username"**,** "email"]**,**)

**unique\_together**

Birlikte benzersiz olacak alanlar için kullanılır.

unique\_together = ("first\_name"**,** "last\_name")

Eğer birden fazla oluşturacaksak;

unique\_together = (  
 ("first\_name"**,** "last\_name")**,** ("username"**,** "email")**,**)

**verbose\_name**

Model adının görüntülenen ismidir varsayılan olarak model ismi atanır. verbose\_name\_plural ise çoğul halidir kullanılmaz ise verbose\_name s takısı gelir. Admin bölümünde kullanılmaktadır.

class Meta:  
 verbose\_name = 'Ürün'  
 verbose\_name\_plural = 'Ürünler'

------------------ BİRDEN ÇOK VERİTABANI KULLANMA ------------------

settings.py’de birden fazla veritabanı tanımlayabileceğimizi söylemiştik. veritabanları settings.py’den alias (takma ad) verilerek tanımlanmaktadır. Varsayılan veritabanı default aliası ile tanımlanır. Default aliası olması zorunludur. Eğer default olarak veritabanı kullanmak istemiyorsanız bunu boş bırakabilirsiniz. Bu durumda kullanılan her model için DATABASE\_ROUTERS tanımlanması gerekmektedir. Default veritabanı kullanım istediğiniz model için DATABASE\_ROUTERS tanımlaması da yapabilirsiniz. Mesela kullanıcı yönetimi için yapılan tüm işlemleri başka bir veritabanından yönetilmesini ayarlayabilirsiniz. Bu konu hakkında ayrıntılı bilgi için Django dokümantasyonundan yardım alabilirsiniz.

Veritabanına özel migrate işlemi yapabilirsiniz.

**./manage.py migrate --database=backup**

Farklı bir veri tabanından çekeceksek using() kullanırız. Mesela backup ismi ile yeni veri tabanı tanımladığımızı düşünelim. Kullanıcıları çekmek için using methodunda backup parametresini göndeririz. Veritabanına kayıt için ise save() methunda using parametresi olarak veritabanı adını göndeririz.

User.objects.using(‘backup’).all()  
  
object.save(using=’backup’)

Bir veritabanındaki nesnesyi diğer veritabanına kopayalayabiliriz. Bir örnek ile inceleyelim. Veritabanımıza staff diye bir kullanıcı kayıt edelim. Kullanıcı default veritabanına eklendiğinde primary key alır. Daha sonra bu primary key ile backup veritabanına kayıt etmeye çalışır. Eğer primary key kullanılmıyorsa aynı kaydı yapar. Eğer primary key kullanılıyorsa eski kayıt üzerine yazar.

user = User(username=’staff’)  
user.save()  
user.save(using=’backup’)

Eğer üzerine yazmasını istemiyorsak primary keyi sıfırlamamız gerekir.

user = User(username=’staff’)  
user.save()  
user.pk = None  
user.save(using=’backup’)

İstenilen veritabanından silme işlemlerinde seçilen kullanıcı hangi veritabanına ait ise silme işlemi o veritabanından gerçekleştirilir. İki veritabanı eşzamanlı olarak güncelleniyor ve bir kaydı diğerinden de silmek istiyorsanız user nesnesi defaul veritabanında olsa bile using parametre olarak silmek istediğiniz veritabanını gönderebilirsiniz.

user.delete(using=’backup’)

Aynı anda birden fazla veritabanı ile çalışmak için daha detaylı bilgiye Django dokümantasyonundan ulaşabilirsiniz.

----------- DATABASE TRANSACTIONS -----------

Django bize veritabanı davranışlarını değiştirme imkanı sunar.

Varsayılan davranış autocommittir. Bu davranış türünde her sorgu hatalı dönmez ise derhal veritabanına yüklenir.

Diğer bir yöntem ise atomic request’dir. Veritabanı ayarlarında “ATOMIC\_REQUEST” true olarak ayarlanırsa tüm proje için geçerli olur. Bu yöntem ya hep ya hiç mantığı ile çalışır. View çağırıldığında veritabanı işlemleri kayıt edilmeye başlanır eğer hata ile karşılaşılırsa geri alınır. Hatasız tamamlanırsa ise işleme konulur.

Mesela sitemize kullanıcı kayıt oldu 4-5 tabloya ilişkili veriler ekleyeceğiz. Tablolardan birine ekleme yaparken sorun olursa eksik bilgi kayıt edilmiş olacak. Bu gibi durumlarda atomic request ile önceki tablolara eklenen verileri geri alabiliyoruz.

Bu yöntem her istek için bir işlem açtığından yüksek trafikli sitelerde performans düşüklüğüne neden olabilir.

ATOMIC\_REQUEST = False olarak kullanmak ve istediğimiz istekler için TRUE olarak ayarlamak daha doğru olabilir. Bunun için @transaction.atomic dekoratörü kullanılır. Tam tersi durum için ise @transaction.non\_atomic\_request dekoratörünü kullanırız.

from django.db import transaction  
  
# ATOMIC\_REQUESTS=True olduğu durumlarda iptal etme  
@transaction.non\_atomic\_requests  
def index(request):   
 # Bir işlemdeki hata diğerlerini etkilemez  
 veritabanı\_islemi\_1()  
 veritabanı\_islemi\_2()  
 veritabanı\_islemi\_3()  
  
# ATOMIC\_REQUESTS=False olduğu durumlarda aktif etme  
@transaction.atomic  
def detail(request):  
 # Bir işlemdeki hata diğerlerindeki veritabanı işlemlerini geri alır.   
 veritabanı\_islemi\_1()  
 veritabanı\_islemi\_2()  
 veritabanı\_islemi\_3()  
 # Hata ile karşılaşılmazsa veritabanına ekleme işlemi tamamlanır.

atomic request context manager ile kullanılabilir. Context managerlar bir kaynağı istediğiniz yerde kullanmamıza yarar. Biz transactionı tüm viewde değilde bir kısmında kullanmak istiyorsak context manager ile kullanabiliriz. With ile birlikte kullanılır.

def viewfunc(request):  
 # bu bölüm autocommit modunda çalışır  
 veritabani\_islemleri\_1()  
  
 with transaction.atomic():  
 # bu bölüm atomic modunda çalışır  
 veritabani\_islemleri\_2()

# bu bölüm autocommit modunda çalışır  
 veritabani\_islemleri\_3()

Try / except blokları ile atomic modundaki hataları yakalabiliriz. Bunun için context manager ile bir hile yaparız. BU sayede atomic requestimiz bozulmaz.

Atomic requestlerde işlem hatasız sonuçlandığında on\_commit() metodu ile fonksiyon çağırılabilir. Kullanıcı siteye kayıt ol düğmesine bastı. Verileri 5 tane tabloya sorunsuz eklendi. Bu noktada on\_commit() metodu çalıştırılır. Bu komut ile biz mesela kullanıcıya mail gönderebiliriz.

from django.db import IntegrityError**,** transaction  
  
@transaction.atomic  
def viewfunc(request):

# bu bölüm 1. atomic modunda çalışır  
 create\_parent()  
  
 try:

# bu bölüm 2. atomic modunda çalışır. Eğer hata yakalanırsa 1. atomic modundan çıkılmaz  
 with transaction.atomic():  
 generate\_relationships()  
 except IntegrityError:  
 handle\_exception()

# bu bölüm 1. atomic modunda çalışır  
 add\_children()

# foo metodu çağırılır

transaction.on\_commit(foo)

with transaction.atomic(): # bu bölümde 1. atomic modunda çalışır   
 transaction.on\_commit(foo)  
  
 with transaction.atomic(): # bu bölümde ikinci atomic modunda çalışır  
 transaction.on\_commit(bar)  
  
# foo ve bar metodları 1. atomic başarıyla sonuçlandığında çağırılır.

with transaction.atomic(): # bu bölümde 1. atomic modunda çalışır

transaction.on\_commit(foo)  
  
 try:  
 with transaction.atomic(): # bu bölümde 2. atomic modunda çalışır  
 transaction.on\_commit(bar)  
 raise SomeError() # 2. atomic modunu iptal eder

except SomeError:  
 pass

# 1. atomic modu başarıyla sonuçlandığında foo metodu çağırılır fakat bar metodu çağırılmaz.

Güvenli noktalar oluşturma

from django.db import transaction  
  
  
@transaction.atomic  
def viewfunc(request):  
 # bu bölüm 1. atomic modunda çalışır

a.save() # transaction a.save() içerir

sid = transaction.savepoint() # geri dönülebilecek güvenli sid noktası. Güvenli nokta a.save() içerir

b.save() # transaction a.save() ve b.save() içerir.  
  
 if want\_to\_keep\_b:  
 transaction.savepoint\_commit(sid) # güvenli noktayı güncelle  
 # güvenli nokta a.save() ve b.save()  
 else:  
 transaction.savepoint\_rollback(sid) # son güvenli noktayı al  
 # güvenli nokta a.save() içerir

save pointe geri alma

a.save() # transaction a.save() içerir  
sid = transaction.savepoint() # güvenli nokta a.save() içerir

try:  
 b.save() # hata verdiğini düşünelim. Alttaki yeni güvenli nokta oluşmaz.  
 transaction.savepoint\_commit(sid)  
except IntegrityError:  
 transaction.savepoint\_rollback(sid) # son güvenli noktayı alır. a.save() içerir

c.save() # a.save() ve c.save() işlenir

tüm işlemleri geri alma

a.save() # transaction a.save() içerir

try:  
 b.save() # hata verdiğini düşünelim.

except IntegrityError:  
 transaction.rollback() # güvenli nokta olmadığı için herşey geri alınır

c.save() # sadece c.save() işlenir

----------- DATABASE OPTIMISATION -----------

Model katmanına son olarak Django’nun veritabanı optimizasyonu için bizelere vermiş olduğu tavsiyelerden bahsedeceğim. Bildiğiniz gibi bir veritabanı işlemini Django ile birden fazla şekilde yapabiliriz. Kayıtları saymak için count() mu kullanmalıyız yoksa len() mi ? Şimdi başlıklar altında veritabanı optimizasyonu hakkında bilgiler vereceğim.

Öncelikle oluştuduğumuz sorgu setlerinin hangi sorguları olşturduğunu bilmeliyiz. Sorgu setlerinin veritabanı için hangi sorgulara dönüştüğünü explain() metodunu kullanabiliriz. Diğer bir yöndemde django debug toolbar kullanmaktadır.

Veritabanı optimizasyonu tek başına yeterli değildir. Bazen veritabanını optimize ederken Python tarafında yapılan işlemlere zarar verebilir. Her ikisini birden dengeli bir şekilde yürütmek gereklidir.

**Standart Veritabanı Optimizasyon Teknikleri**

* index kullanımı

veritabanı optimizasyonu konusunda en önemli konulardan birisidir. exclude(), filter(), order\_by() ve benzeri methodların kullanıldığı alanlar için index oluşturulabilir.

* alan tiplerinin doğru kullanılması

misal olarak sayı tuttuğumuz alanları ayarlarken small integer kabul edilecek bir alan için integer kullanmak gereksiz yüke neden olacaktır.

**Sorgu Setlerini Anlamak**

Sorgu setlerinin işleyişini anlamak optimizasyon için önemlidir.

**sorgu setleri lazy çalışır.**

Sorgu seti oluşturulduğunda herhangi bir veritabanı işlemi yapılmaz. Sorgu sonuçları kullanılana kadar veritabanına erişim sağlanmaz. Sanki üç tane veritabanı işlemi yapmışız gibi ancak print(q) satırına kadar herhangi bir veritabanı işlemi yapmaz. Genel olarak bir sorgu seti siz isteyene kadar veri tabanından alınmaz.

q = Product.objects.filter(nane\_\_startswith="Product”)  
q = q.filter(created\_\_lte=datetime.date.today())  
q = q.exclude(content\_\_icontains="mutfak")  
print(q)

* Sorgu setleri ne zaman veritabanında değerlendirilir.
  + iteration yapıldığında

for e in Entry.objects.all():  
 print(e.headline)

* + limitleme yapıldığında genellikle sorgu setleri veritabanında değerlendirilmez. Fakat adımlama özelliği kullanılırsa veritabanına erişim sağlanır.
  + repr(), len(), list() metodlarının kullanılması ile veritabanına erişim sağlanır.

entry\_list = list(Entry.objects.all())

* + Sorgu setininden dönen kayıt olup / olmadığının tespit edildiği durumlarda veritabanına erişim sağlanır.

if Entry.objects.filter(headline="Test"):  
 print("There is at least one Entry with the headline Test")

* + Sorgu setininden dönen kayıt olup / olmadığının tespit edildiği durumlarda veritabanına erişim sağlanır.
* Veriler bellekte nasıl saklandığı / cachelendiği

Django, sorgu setlerini ilk defa çalıştırıp sonuçları getirdikten sonra bu sonuçları önbellekte saklar. Bu şekilde veritabanından iki kez sorgu yapılır

print([product.headline for product in Product.objects.all()])  
 print([product.pub\_date for product in Product.objects.all()])

tüm ürünleri önbelleğe alıp ordan kullanarak gereksiz yere ikinci bir sorgu yapmasını önleyebiliriz.

# Aktivite yok.  
products = Product.objects.all()  
  
# Sonuçlar veritabanından alınıp, önbelleğe yazılıyor.  
names = [product.name for product in products]  
  
# Önbellekteki sonuçlar kullanılıyor.  
slugs = [product.slug for product in products]

sorgu setleri her zaman ön belleğe atılmaz. Örneğin limitleme işlemleri önbelleğe atılmaz.

# Aktivite yok.  
products = Product.objects.all()  
  
# 5. Kayıt için sonuçlar veritabanından alınıp, önbelleğe yazılmaz.  
print(products[5])

# 5. Kayıt için yeniden veritabanı sorgusu yapılır, önbelleğe yazılmaz.  
print(products[5])

# ilk 5 kayıt için yeniden veritabanı sorgusu yapılır, önbelleğe yazılmaz.  
  
print(products[:5])

Sorgu setini önce önbelleğe alsak daha sonra limitleme işlemlerini yaparsak sürekli veritabanından sorgulama yapmaya gerek kalmaz.

# Aktivite yok.  
products = Product.objects.all()  
  
# veritabanından sorgulama yapıldı, ön belleğe alındı.

[product for product in products]

# ön bellekteki sonuçlar kullanılır

print(products[5])

print(products[:5])

çağıralabilir olmayan özellikler ön belleğe alınır.

# Aktivite yok.  
product = Product.objects.get(id=1)  
  
# veritabanından sorgulama yapıldı, ön belleğe alındı.

print(product.name)

print(product.name) # ön bellekteki sonuçlar kullanılır

print(product.title) # ön bellekteki sonuçlar kullanılır

print(product.authors.all()) # yeniden sorgulama yapılır

print(product.authors.all()) # yeniden sorgulama yapılır

**İşleri Python’da değil Veritabanı kısmında halledin**

* verileri filtrelemek için filter() ve exclude() kullanın
* aynı modeldeki diğer alanlara göre filtreleme için F() nesnesi kullanın
* annonate ve aggretion kullanın

**Verileri veritabanından benzersiz ve indekslenmiş sütunlardan almaya özen gösterin. Bu sorguların daha hızlı çalışmasını sağlayacaktır.**

**İhtiyacınız olan tüm verileri bir kerede almaya özen gösterin**

Biraz önce örneklerde çağıralabilen alanların yeniden sorgulama yaptığını söylemiştik. Eğer neleri alacağımızı biliyorsak select\_related() ve prefetch\_related() ile tek seferde alabiliriz.

**İhtiyacınız olmayan alanları almaktan kaçının**

values(), values\_list(), defer() ve only() alanlarını kullanarak sadece kullanacağız alanları veritabanından getirelim.

**count() ve exitsts() kullanalım.**

Sadece kayıt sayısını istiyorsak len() yerine count() kullanmalıyız. Eğer en az bir kayıt olduğunun tespitini istiyorsak if yerine exists() kullanmalıyız. Fakat sorgu seti daha önce yüklendiyse count ve exists kullanımı ek yük getireceğinden len ve if kullanmak daha uygun olur.

**update() ve delete() kullanalım.**

Öncelikle kayıdı getirip sonra onu tek tek güncellemek ve silmek yerine direk toplu update() ve delete() metodlarının kullanılması daha uygun olur.   
 **Eğer gerekmiyorsa order\_by ve meta ayarlarınındaki sıralamaların kullanımından kaçının.**