----------- 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