Accuknox Django trainee assignment

-Eeshaan Dhanuka

Question 1: By default, are Django signals executed synchronously or asynchronously? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

Ans:

Django signals, by default, are executed synchronously. This means that when a signal is triggered, the code execution will wait for the signal handler to finish before moving on to the next line of code.

Below is the code snippet.

```
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver

class car(models.Model):
    car_name = models.Charfield(max_length=100)

@receiver(post_save, sender=car)
def call_car_api(sender, instance, **kwargs):
    print(sender, instance, kwargs)

car_obj = car.objects.create(name = "Nissan")

print("Car object created")
```

In the above code,

a Django model ‘car’ and a signal handler function ‘call_car_api’ is defined that connects to the ‘post_save’ signal of ‘car’.

When a instance of ‘car’ is created, a signal is sent and the signal handler is executed synchronously.

The print statement will execute only after the object has been created and the signal handler completed.

Question 2: Do Django signals run in the same thread as the caller? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

Ans: Yes, by default, Django signals are executed in the same thread as the caller. This means that the code that triggers the signal and the signal handler itself run in the same thread. To prove this, we are using Python’s threading.get_ident() method, which returns the current thread's identifier. We compared the thread ID in the signal handler with the thread ID of the code that triggers the signal.

Below is the code snippet that demonstrates signals run in the same thread as the caller:

```
import threading
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver

class car(models.Model):
    car_name = models.Charfield(max_length=100)

@receiver(post_save, sender=car)
def call_car_api(sender, instance, **kwargs):
    print(f"Signal handler thread ID:{threading.get_ident()}")
    print("Signal handler finished execution")

car_obj = car.objects.create(name = "Nissan")

print(f" Caller thread ID: {threading.get_ident()}")
```

In the above code,

The signal handler for post_save is triggered right after the car_obj is created. It also prints the current thread ID to check if it's the same as the caller's thread. After the signal handler has completed, the calling code prints the thread ID again.

Output - The thread ID in the caller and the signal handler is the same. This proves that Django signals, by default, are executed **in the same thread** as the caller.

Question 3: By default, do Django signals run in the same database transaction as the caller? Please support your answer with a code snippet that conclusively proves your stance. The code does not need to be elegant and production ready, we just need to understand your logic.

Ans: Yes, by default, Django signals that are triggered during database operations (like post_save, pre_save, post_delete, etc.) run within the same database transaction as the caller. This means that if a signal handler performs some additional database operations, and the main transaction is rolled back, the changes made by the signal handler will also be rolled back.

Below is the code snippet that demonstrates signals run in the same database transaction as the caller:

```
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.db import transaction


class car(models.Model):
    car_name = models.Charfield(max_length=100)

@receiver(post_save, sender=car)
def call_car_api(sender, instance, **kwargs):

    instance.name = "Car_name_modified"
    instance.save()

car_obj = car.objects.create(name = "Nissan")

with transaction.atomic():
    car_obj.name = "Updated"
    car_obj.save()

print("Name after save: ", car_obj.name)
```

In the above code,

The Django model 'MyModel' and its associated signal handler function 'car_call_api' have been defined. 'car' 'post_save' signal is bound to this method. We edit the instance's 'name attribute' inside the signal handler and save it back to the database. With the name "Nissan," we create an instance of "MyModel" and print its name both before and after saving. The object's name is then updated to "Updated" within the transaction, and a new transaction is explicitly started using "transaction.atomic()."

Changes made by the signal handler are visible within the same transaction because they are made to the same instance and are performed within the same database transaction as the caller. As a result, Django signals operate by default in the same database transaction as the caller because the object name printed after save accurately reflects the modifications made by both the caller and the signal handler.