Django Signals

**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.

Answer:

By default, Django signals are executed synchronously. This means that when a signal is triggered, the main program waits for the signal's task to complete before it can move forward. To support this statement the code snippet is given below.

Code Snippet

import time

from django.db.models.signals import post\_save

from django.contrib.auth.models import User

from django.dispatch import receiver

@receiver(post\_save, sender=User)

def signal\_handler(sender, instance, created, \*\*kwargs):

if created:

print("Signal: A new user was created. Starting signal task...")

time.sleep(5)

print("Signal: Finished signal task.")

# Function to create a new user

def create\_user():

print("Main Code: Creating a new user...")

user = User.objects.create(username='testuser') # This automatically triggers the post\_save signal

print("Main Code: User creation process is done.")

create\_user()

Expected Output:

Main Code: Creating a new user...

Signal: A new user was created. Starting signal task...

(5-second delay)

Signal: Finished signal task.

Main Code: User creation process is done.

Conclusion

This code shows that the Django signal is synchronous because the main code waits for the 5-second delay introduced by the signal handler. The output shows that the main function (create\_user) doesn't finish until the signal task is complete.

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.

Answer:

Yes, by default, Django signals run in the same thread as the caller. This means that when a signal is triggered, the code inside the signal handler runs within the same thread as the function or operation that triggered the signal. To support this statement lets take the previous code snippet.

Code Snippet

import time

from django.db.models.signals import post\_save

from django.contrib.auth.models import User

from django.dispatch import receiver

@receiver(post\_save, sender=User)

def signal\_handler(sender, instance, created, \*\*kwargs):

if created:

print("Signal: A new user was created. Starting signal task...")

time.sleep(2)

print("Signal: Finished signal task.")

# Function to create a new user

def create\_user():

print("Main Code: Creating a new user...")

user = User.objects.create(username='testuser') # This automatically triggers the post\_save signal

print("Main Code: User creation process is done.")

create\_user()

Expected Output:

Main Code: Creating a new user...

Signal: A new user was created. Starting signal task...

(2-second delay)

Signal: Finished signal task.

Main Code: User creation process is done.

Conclusion

This output clearly shows that the main program pauses while the signal handler is working and only continues after the signal has finished its job. This behavior proves that Django signals run synchronously in the same thread as the caller because the main program waits for the signal to complete before moving on.

If the signal handler was running in a different thread or asynchronously, you would see "User creation is complete!" immediately after starting the user creation, without waiting for the signal handler to finish its task.

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.

Answer:

By default, Django signals do not run in the same database transaction as the caller.This means that while the signal gets triggered when something happens in the database (e.g., saving a model), it is not guaranteed to be part of the same database transaction unless explicitly handled using Django's transaction management.

Code Snippet

from django.db import transaction

from django.db.models.signals import post\_save

from django.contrib.auth.models import User

from django.dispatch import receiver

@receiver(post\_save, sender=User)

def signal\_handler(sender, instance, created, \*\*kwargs):

if created:

print("Signal: User created successfully, running signal handler.")

def create\_user():

try:

with transaction.atomic():

print("Main Code: Creating user inside a transaction...")

user = User.objects.create(username='testuser')

print("Main Code: Simulating an error to roll back the transaction...")

raise Exception("Simulated error to roll back transaction") # Force a rollback

except Exception as e:

print(f"Main Code: Transaction failed: {e}")

create\_user()

Expected Output:

Main Code: Creating user inside a transaction...

Signal: User created successfully, running signal handler.

Main Code: Simulating an error to roll back the transaction...

Main Code: Transaction failed: Simulated error to roll back transaction

Explanation of Output:

Signal Triggered: Even though we raised an error to roll back the transaction, you can see that the signal handler still printed its message (Signal: User created successfully, running signal handler.). This shows that the signal ran despite the failure of the transaction.Transaction Failed: The main code reports that the transaction failed and the user creation was rolled back (Main Code: Transaction failed).

Conclusion

This behavior proves that Django signals do not, by default, run in the same database transaction as the caller. The signal handler runs even though the transaction was rolled back, demonstrating that it is not part of the atomic transaction. If you need signals to run as part of the same transaction, you would need to manage that explicitly using Django's transaction.on\_commit() method.