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.

import time

from django.db import models

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

from django.dispatch import receiver

# Simple model for demonstration

class MyModel(models.Model):

name = models.CharField(max\_length=100)

# Signal receiver for post\_save

@receiver(post\_save, sender=MyModel)

def my\_signal\_receiver(sender, instance, \*\*kwargs):

print("Signal handler started")

time.sleep(5) # Simulating a long-running task

print("Signal handler finished")

# To test the synchronous nature:

# Saving an instance of MyModel in a Django shell or view

if \_\_name\_\_ == "\_\_main\_\_":

print("Saving model instance")

my\_model = MyModel(name="Test Instance")

my\_model.save()

print("Model save method finished")

* **Create a simple model** MyModel with just one field (name).
* **Define a post\_save signal** receiver that listens to when an instance of MyModel is saved.
* In the receiver, add a **2-second delay** using time.sleep(2) to simulate a slow task.
* **Test the order of execution**:

When you save the model, the post\_save signal is triggered.

The signal handler runs **immediately** (synchronously) and only after it completes, the code continues to the next statement.

---------------------------------------------------------------------------------------------------------

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.

Yes, Django signals run in the same thread as the caller by default. To prove this, we can inspect the current thread in both the caller and the signal handler (receiver). By comparing the thread IDs in both places, we can confirm that they are the same.

import threading

from django.db import models

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

from django.dispatch import receiver

# Simple model

class MyModel(models.Model):

name = models.CharField(max\_length=100)

# Signal receiver for post\_save

@receiver(post\_save, sender=MyModel)

def my\_signal\_receiver(sender, instance, \*\*kwargs):

print(f"Signal handler thread ID: {threading.get\_ident()}")

# Testing in Django shell or view

if \_\_name\_\_ == "\_\_main\_\_":

print(f"Caller thread ID: {threading.get\_ident()}")

my\_model = MyModel(name="Test Instance")

my\_model.save() # This triggers the post\_save signal

Explanantion:

* We use Python's threading.get\_ident() to retrieve the current thread's ID. This ID is unique to each thread.
* We print the thread ID in two places:

**In the caller** (right before the model instance is saved).

**In the signal handler** (when the post\_save signal is triggered).

If the thread IDs are the same, it confirms that the signal handler is running in the same thread as the caller.

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.  
  
Yes, by default, Django signals are executed within the same database transaction as the caller when triggered by actions such as save() or delete() in models. This means that if the transaction is rolled back, any changes made during the signal handler's execution will also be rolled back.

To prove this, we can simulate a scenario where an error occurs after the signal is triggered, causing a rollback. If the signal's changes are also rolled back, it confirms that the signal runs in the same database transaction as the caller.

from django.db import models, transaction

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

from django.dispatch import receiver

# Simple model

class MyModel(models.Model):

name = models.CharField(max\_length=100)

# Signal receiver for post\_save

@receiver(post\_save, sender=MyModel)

def my\_signal\_receiver(sender, instance, \*\*kwargs):

print("Signal handler running")

# Let's make some change in another model or log it

instance.name = "Updated in Signal"

instance.save() # This save should happen inside the same transaction

# Testing in Django shell or a view:

if \_\_name\_\_ == "\_\_main\_\_":

try:

with transaction.atomic(): # Open a transaction

my\_model = MyModel(name="Initial Name")

my\_model.save() # This triggers the post\_save signal

print(f"Model saved with name: {my\_model.name}")

# Simulate an error that causes the transaction to rollback

raise Exception("Simulated Error!")

except Exception as e:

print(f"Transaction rolled back due to: {e}")

# Check if the model was saved in the database or rolled back

saved\_model = MyModel.objects.filter(name="Updated in Signal").exists()

print(f"Was the signal change persisted? {saved\_model}")

Explanation:

* **Model**: MyModel is a simple model with a name field.
* **Signal**: A post\_save signal is triggered after saving an instance of MyModel. In the signal handler, we modify the name field and save it again, assuming this action happens within the same transaction.
* **Atomic Block**: We wrap the save operation inside a transaction.atomic() block. This ensures that the entire block runs in a single transaction.
* **Simulated Error**: After saving the model and triggering the signal, we raise an exception, which should cause the transaction to roll back.
* **Check Database**: After the exception, we check if the changes made in the signal handler (updating the model's name) were persisted in the database.