**Question 1:**

By default, Django signals are executed **synchronously**. This means that when a signal is emitted, the connected receivers (functions or methods) are executed immediately in the same thread and blocking manner. The main process waits for the signal handlers to complete before continuing with the rest of the code.

**Step-by-Step Code:**

1. **Define a Custom Signal**.
2. **Create a Receiver Function** that performs a time-consuming operation (e.g., a delay).
3. **Emit the Signal** and measure the execution time to show that it blocks the execution flow.

**Example Code Snippet:-**

# models.py or any file where you want to define the signal and receiver

import time

import django.dispatch

# Step 1: Define a custom signal

my\_signal = django.dispatch.Signal()

# Step 2: Define a receiver that simulates a delay to show synchronous execution

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

print("Receiver started...")

time.sleep(3) # Simulates a time-consuming task

print("Receiver finished...")

# Connect the receiver to the signal

my\_signal.connect(my\_receiver)

# Step 3: Emit the signal and measure the time

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

import time

print("Emitting signal...")

start\_time = time.time()

my\_signal.send(sender=None)

end\_time = time.time()

print(f"Signal emitted and processed in {end\_time - start\_time} seconds.")

**Expected Output:-**

Emitting signal...

Receiver started...

Receiver finished...

Signal emitted and processed in 3.006 seconds.

**Question 2:**

Yes, Django signals run in the same thread as the caller by default. When a signal is emitted, its connected receivers execute in the same thread, meaning they share the same execution context as the caller.

**Proof with Code Snippet:**

To demonstrate that Django signals run in the same thread as the caller, we can use Python’s threading module. This module provides the ability to inspect the current thread using threading.current\_thread(). By comparing the thread information when the signal is emitted and when the receiver executes, we can confirm that they run in the same thread.

**Step-by-Step Code Explanation:**

1. **Import the necessary modules**: We'll use Django's signals, and Python's threading module to get the current thread name.
2. **Define a custom signal and a receiver function**: The receiver will print the current thread name.
3. **Emit the signal from the main thread**: The thread information will be compared between the caller and the receiver.

**Example Code Snippet:-**

# Import required modules

import django.dispatch

import threading

# Step 1: Define a custom signal

my\_signal = django.dispatch.Signal()

# Step 2: Define a receiver that prints the current thread information

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

print(f"Receiver is running in thread: {threading.current\_thread().name}")

# Connect the receiver to the signal

my\_signal.connect(my\_receiver)

# Step 3: Emit the signal and print the current thread information in the main caller

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

print(f"Caller is running in thread: {threading.current\_thread().name}")

my\_signal.send(sender=None)

**Expected Output:-**

Caller is running in thread: MainThread

Receiver is running in thread: MainThread

**Question 3:**

Yes, by default, Django signals run in the same database transaction as the caller. This means that when a signal is emitted within a transaction, the connected receivers execute within the same transaction context.

To demonstrate that Django signals run in the same database transaction as the caller, we can use Django’s transaction management tools. Specifically, we will:

1. Create a simple model and use Django's signals to connect to a post-save event.
2. Use the transaction.atomic() context manager to create a transaction.
3. Print transaction status in both the caller and the signal receiver to verify that they are in the same transaction context.

**Step-by-Step Code Explanation:**

1. **Define a Model**: We'll create a simple Django model to use with a post\_save signal.
2. **Connect a Receiver to the Signal**: The receiver will check if it is running in the same transaction as the caller.
3. **Use transaction.atomic() to Create a Transaction**: We'll insert a new record and check if the signal is running in the same transaction.

**Example Code Snippet:-**

# models.py

from django.db import models, transaction

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

from django.dispatch import receiver

import threading

# Step 1: Define a simple model

class TestModel(models.Model):

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

# Step 2: Define a receiver that checks transaction status

@receiver(post\_save, sender=TestModel)

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

# Check if we are in a transaction

print(f"Receiver is running in thread: {threading.current\_thread().name}")

print(f"Receiver in atomic block: {transaction.get\_connection().in\_atomic\_block}")

# Step 3: Create a new TestModel instance inside a transaction

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

from django.db import connection

from django.conf import settings

import django

# Step 3.1: Django setup

settings.configure(

DATABASES={

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': ':memory:',

}

},

INSTALLED\_APPS=['\_\_main\_\_'],

)

django.setup()

# Step 3.2: Create the table

with connection.schema\_editor() as schema\_editor:

schema\_editor.create\_model(TestModel)

# Step 3.3: Start a transaction and save a model instance

with transaction.atomic():

print(f"Caller is running in thread: {threading.current\_thread().name}")

print(f"Caller in atomic block: {transaction.get\_connection().in\_atomic\_block}")

TestModel.objects.create(name="Test Name")

**Expected Output:-**

Caller is running in thread: MainThread

Caller in atomic block: True

Receiver is running in thread: MainThread

Receiver in atomic block: True