Django Signals: Synchronous Execution, Threading, and Transactions

Question 1: Are Django signals executed synchronously or asynchronously?

Answer:

By default, Django signals are executed **synchronously**, meaning they run in the same execution flow as the caller. This means that if a signal handler takes a long time to execute, it can slow down the request-response cycle or other parts of the application.

Code:

```
import os

import django

import time

import threading

from django.db.models.signals import post_save

from django.contrib.auth import get_user_model

from django.dispatch import receiver

# Properly setup Django before importing models

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "myproject.settings")

django.setup()

# Use get_user_model() instead of direct User import

User = get_user_model()
```

```
# Print the main thread ID
print(f"Main program running in thread: {threading.get_ident()}")
# Define the signal
@receiver(post_save, sender=User)
def user_created_signal(sender, instance, created, **kwargs):
    if created:
        print(f"Signal received in thread: {threading.get_ident()}")
        time.sleep(3)  # Simulate delay
        print("Signal processing completed!")
# Ensure a unique username before creating a user
username = f"test_user_{int(time.time())}"  # Generate a unique
username using timestamp
print(f"Saving user with username: {username}...")
user = User.objects.create(username=username)
print(f"User '{user.username}' saved successfully!")
```

Explanation:

- The time.sleep(5) inside the signal function simulates a delay.
- If Django signals were asynchronous by default, the "User Created Successfully!" message would appear immediately, and the signal would complete execution separately.
- However, the execution order proves that the signal runs **synchronously**, blocking execution until completion.

Question 2: Do Django signals run in the same thread as the caller?

Answer:

Yes, by default, Django signals run in the **same thread** as the caller.

Code:

```
import django

import threading

from django.conf import settings

# Django Setup

settings.configure(

    INSTALLED_APPS=[

        'django.contrib.auth',

        'django.contrib.contenttypes',

        'django.contrib.sessions',

        'django.contrib.messages',

    ],

    DATABASES={'default': {'ENGINE': 'django.db.backends.sqlite3', 
'NAME': 'db.sqlite3'}},

)

django.setup()  # Initialize Django properly before imports

from django.contrib.auth.models import User  # Import after setup

# Function to create a user and trigger a signal

def create_user():

    print(f"Running in thread: {threading.get_ident()}")

    # Check if user exists before creating

    if not User.objects.filter(username="test_user").exists():

        User.objects.create(username="test_user")
        print("User created successfully!")
    else:
        print("User already exists!")
# Main thread
print(f"Main program running in thread: {threading.get_ident()}")
create_user()
```

Explanation:

- The main thread ID is printed at the start.
- The thread ID inside the signal handler is printed when executed.
- Since both thread IDs are the same, Django signals run in the **same thread** by default.

Question 3: Do Django signals run in the same database transaction as the caller?

Answer:

Yes, Django signals run in the **same database transaction** by default.

Code:

```
import os
import django
# Setup Django before importing models
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "myproject.settings")
django.setup()  # Initialize Django
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
# Signal handler
@receiver(post_save, sender=User)
def user_created_signal(sender, instance, created, **kwargs):
    if created:
        print("Signal received! Processing...")
        instance.first_name = "UpdatedName"
        instance.save()  # Modify instance within signal
        print("Signal processing completed!")

try:
    with transaction.atomic():  # Ensures database transaction handling
        print("Saving user...")
        user = User.objects.create(username="test_user3")
        print("User saved! Raising exception now...")
        raise Exception("Simulating an error after user save")
except Exception as e:
    print(f"Exception occurred: {e}")
# Check if user is actually saved in DB
user_exists = User.objects.filter(username="test_user3").exists()
print(f"Was user saved in DB? {'Yes' if user_exists else 'No'}")
```

Explanation:

- The transaction.atomic() ensures that everything inside it runs as a single transaction.
- The signal raises an exception, causing the transaction to roll back.
- If signals were running **outside** the transaction, the user would still be created.
- Since User.objects.filter(username="test_user_transaction").exists() returns False, we confirm that the signal runs **within the same transaction** by default.