[Topic: Django Signals](https://docs.djangoproject.com/en/3.2/topics/signals/)

Question 1: By default are django signals executed synchronously or asynchronously?

By default, Django signals are executed synchronously. This means that when a signal is triggered, it is processed immediately in the same thread and process that sent the signal.

Below mentioned is a code snippet to show this:

import time

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

from django.dispatch import receiver

from django.contrib.auth.models import User

# Define a signal

@receiver(post\_save, sender=User)

def user\_saved\_signal(sender, instance, \*\*kwargs):

    print("Signal received. Starting long-running task...")

    time.sleep(5)

    print("Signal processing complete.")

# To trigger the signal, save a user object

def trigger\_signal():

    print("Saving user...")

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

    print("User saved.")

# In the Django shell or a view:

trigger\_signal()

When you run trigger\_signal(), you will see that the message "Signal received..." will be printed immediately after the user is saved, and the program will wait for 5 seconds before the "Signal processing complete." message is printed. This proves Django signals by default is synchronous.

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

Yes, Django signals run in the **same thread** as the caller by default. This means that when a signal is triggered, it is processed synchronously within the same thread, and it blocks further execution until the signal handler completes.

import threading

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

from django.dispatch import receiver

from django.contrib.auth.models import User

# Define a signal

@receiver(post\_save, sender=User)

def user\_saved\_signal(sender, instance, \*\*kwargs):

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

# Function to trigger the signal

def trigger\_signal():

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

    # Create a user to trigger the post\_save signal

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

# Call the function to trigger the signal

trigger\_signal()

When we run this code, it will print the thread information for both the trigger\_signal function and the signal handler user\_saved\_signal as MainThread

**Question 3**: By default do django signals run in the same database transaction as the caller?

By default, Django signals do not run in the same database transaction as the caller. They are executed after the database transaction commits unless explicitly tied to the transaction. For example, post\_save signals are executed after the save method, but not necessarily as part of the database transaction that commits the object to the database

from django.db import transaction

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

from django.dispatch import receiver

from django.contrib.auth.models import User

# Signal handler

@receiver(post\_save, sender=User)

def user\_saved\_signal(sender, instance, \*\*kwargs):

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

# Function to create a user inside a transaction

def trigger\_signal():

    with transaction.atomic():

        print(f"Before saving user, in transaction: {transaction.get\_connection().in\_atomic\_block}")

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

# Trigger the signal

trigger\_signal()

transaction.get\_connection().in\_atomic\_block is used to check if the current code is running inside an atomic transaction block

Below is the output of function:

Starting transaction...

Before saving user, in atomic block: True

After saving user, in atomic block: True

Signal running in atomic block: False

Transaction ended.

The output shows that the signal handler is not running inside the transaction

Topic: Custom Classes in Python

class Rectangle:

    def \_\_init\_\_(self, length: int, width: int):

        self.length = length

        self.width = width

    def \_\_iter\_\_(self):

        # We create a list of items to iterate over

        self.\_attributes = [{'length': self.length}, {'width': self.width}]

        self.\_index = 0  # Index to keep track of iteration

        return self

    # Defines the iteration logic (next item to return)

    def \_\_next\_\_(self):

        if self.\_index < len(self.\_attributes):

            result = self.\_attributes[self.\_index]

            self.\_index += 1

            return result

        else:

            # Stop iteration when we have returned all items

            raise StopIteration

# Example usage:

rect = Rectangle(10, 20)

for attribute in rect:

    print(attribute)