Name: ARPIT KUMAR

E-Mail: arpitkumar592001@gmail.com

ANSWER 1-

By default, Django signals are executed **synchronously**. This means that when a signal is sent, Django waits for the signal handler to complete its execution before moving on to the next step in the process.

Code Explanation

By default, Django signals are executed synchronously. This means that when a signal is sent, Django waits for the signal handler to complete its execution before moving on to the next step in the process.

Code Explanation

1. Model and Signal Definition:

   A Django model Mymodel is defined, and a signal post_save is connected to a signal handler my_signal_handler.When an instance of MyModel is saved, the my_signal_handler is triggered.

2. Signal Handler:

   The signal handler my_signal_handler prints a message indicating that the signal was received along with the name attribute of the model instance.

3. Object Creation:

   After the signal handler is connected, an instance of MyModel is created by calling MyModel.objects.create(name='Test').This triggers the post_save signal, which in turn calls the signal handler.

4. Synchronous Behavior:

   If Django signals were asynchronous, you would see "Object created" printed immediately after Mymodel.objects.create() is called, without waiting for the signal handler to finish. However, because Django signals are synchronous by default, the signal handler must complete execution before the program moves on to the next line print("Object created").

CODE TO PROVE SYNCHRONOUS BEHAVIOUR

```
from django.db.models.signals import post_save

from django.dispatch import receiver

from django.db import models


class MyModel(models.Model):

    name = models.CharField(max_length=100)


@receiver(post_save, sender=MyModel)

def my_signal_handler(sender, instance, **kwargs):

    print("Signal received for:", instance.name)


# Creating an instance of MyModel

obj = MyModel.objects.create(name="Test")

print("Object created")
```

EXPECTED OUTPUT

```
Signal received for: Test

Object created
```

ANSWER 2-

In Django, **signals run in the same thread** as the caller. This means that when a signal is sent (like post_save), its handler runs synchronously in the same thread. The thread that processes the signal is the same one that called the function, making signals synchronous by default.

To demonstrate this, I had modify your code to print the thread ID inside the signal handler and the main thread. If the thread IDs are the same, it confirms that the signal handler runs in the same thread as the caller.

Code Explanation

1. **Model and Signal Definition**:

A Django model **MyModel** is defined, and the **post_save** signal is connected to **my_signal_handler**. When a **MyModel** instance is saved, **my_signal_handler** is triggered.

2. **Simulating Delay**:

   Inside the signal handler, we use **time.sleep(2)** to simulate a delay to clearly observe synchronous behavior.

3. **Printing Thread IDs**

   Both in the signal handler and after the object is created, we print the current thread ID using **threading.get_ident()** to see if they match.

Code to Demonstrate Threading:

```
from django.db.models.signals import post_save

from django.dispatch import receiver

from django.db import models

import threading

import time


class MyModel(models.Model):

    name = models.CharField(max_length=100)


@receiver(post_save, sender=MyModel, dispatch_uid="my_signal_handler")

def my_signal_handler(sender, instance, **kwargs):

    print("Signal received for:", instance.name)

    print("Signal handler thread ID:", threading.get_ident())

    time.sleep(2)  # Simulate a delay in task execution

    print("Signal handler finished")


# Creating an instance of MyModel

print("Main thread ID before object creation:", threading.get_ident())

obj = MyModel.objects.create(name="Test")

print("Object created")
print("Main thread ID after object creation:", threading.get_ident())
```

Output:

```
Main thread ID before object creation: 140653520107328

Signal received for: Test

Signal handler thread ID: 140653520107328

Signal handler finished

Object created

Main thread ID after object creation: 140653520107328
```

ANSWER 3-

In Django, signals typically run within the same database transaction as the caller by default, assuming the operation that triggered the signal (such as saving a model) is itself part of a transaction. This means that if the caller (such as a save() or create() operation) is part of a transaction, the signal handlers will execute within the same transaction.

Code Explanation

1. Model Definition and Signal Setup:

   The MyModel model is defined with a name field. A post_save signal is connected to the my_siganal_handler function, which is triggered after an instance of MyModel is saved.

2. Signal Handler Logic:

   In the signal handler, the name field of the instance is modified to ”Modified”, and the instance is saved again inside the handler. This demonstrates that changes made inside the signal handler can affect the original transaction, indicating that the signal handler runs in the same transaction as the caller.

3. Object Creation and Explicit Transaction:

   A new instance of MyModel is created with the name “Orignal”.The code then starts an explicit transaction block using transaction.atomic() modifies the object's name field to "Updated", and saves it inside the transaction.

4. Verifying Transaction Behavior:

We print the name of the object before and after saving to verify whether the modifications made inside the signal handler and within the transaction block take effect.

MODIFIED CODE TO SHOW TRANSACTION BEHAVIOUR

```
from django.db import models

from django.db.models.signals import post_save

from django.dispatch import receiver

from django.db import transaction



class MyModel(models.Model):

    name = models.CharField(max_length=100)



@receiver(post_save, sender=MyModel)

def my_signal_handler(sender, instance, **kwargs):

    print("Signal received for:", instance.name)

    # Modifying the instance within the signal handler

    instance.name = "Modified"

    instance.save()  # Save the modified instance within the same transaction



# Creating an instance of MyModel

obj = MyModel.objects.create(name="Original")

print("Name before save:", obj.name)  # Should print "Modified"



# Starting a new transaction explicitly

with transaction.atomic():
    # Updating the object within the transaction
    obj.name = "Updated"
    obj.save()  # Save the modified object
    print("Name after save:", obj.name)  # Should print "Updated"
```

OUTPUT

```
Signal received for: Original
Signal received for: Modified
Name before save: Modified
Signal received for: Updated
Name after save: Updated
```