

# **Questions for Django Trainee at Accuknox**

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

**Question 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.

**Ans:**

By default, Django signals are executed **synchronously**. This means the signal handler is executed immediately as part of the normal control flow when the signal is emitted, and the program waits for the signal handler to complete before continuing.

To conclusively demonstrate this, we can use print statements to see the order of execution in a test. If the signal is synchronous, the signal handler will be executed between the lines of code where the model is saved and where the next print statement is called. If the signal were asynchronous, the print statement after the save would execute before the signal handler finishes its execution.

Here’s the demonstration:

models.py

from django.db import models

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

import threading

import time

class Post(models.Model):

title = models.CharField(max\_length=120, default="")

def save\_post(sender, instance, \*\*kwargs):

print("post save message this is done") # Signal handler code

post\_save.connect(save\_post, sender=Post)

test.py

from django.test import TestCase

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

from .models import Post

class PostSignalTest(TestCase):

def test\_post\_save\_signal(self):

# Create a new Post object

print("creating test") # this will be printed before the signal is called

post = Post.objects.create(title="Signal Test")

print("test completed") # this will be printed after the signal is called

self.assertEqual(post.title, "Signal Test")

Test result:

Output as shown

creating test

post save message this is done

test completed

.

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

Ran 1 test in 0.002s

OK

When the Post.objects.create(title="Signal Test") line is executed, the post\_save signal is triggered immediately.

The print statement from the signal handler save\_post ("post save message this is done") is executed before the next line (print("test completed")) in the test is run.

This confirms that the signal is executed synchronously.

**Question 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.

**Ans:**

Yes, by default, Django signals run in the **same thread** as the caller. To demonstrate this, we can print the current thread ID from both the main code and the signal handler. If the signal handler runs in the same thread as the caller, the thread ID will be identical for both the test code and the signal handler.

Here’s how to prove this:

models.py

from django.db import models

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

import threading

import time

class Post(models.Model):

title = models.CharField(max\_length=120, default="")

def save\_post(sender, instance, \*\*kwargs):

time.sleep(5) # for checking that task running in same instance

print(f"Signal handler thread ID: {threading.get\_ident()}") # Thread ID inside the signal handler

post\_save.connect(save\_post, sender=Post)

test.py

from django.test import TestCase

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

from .models import Post

import threading

class PostSignalTest(TestCase):

def test\_post\_save\_signal(self):

print("creating test") # this will be printed before the signal is called

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

post = Post.objects.create(title="Signal Test")

print("test completed") # this will be printed after the signal is called

self.assertEqual(post.title, "Signal Test")

Test result:

creating test

Test thread ID: 29292

Signal handler thread ID: 29292

test completed

.

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

Ran 1 test in 0.002s

OK

Both the main test and the signal handler print the thread ID using threading.get\_ident(), which returns the ID of the current thread. Since Django signals are synchronous by default, they run in the same thread as the code that triggered the signal. The fact that the thread ID is identical in both prints confirms that the signal runs in the same thread as the caller.

**Question 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.

Ans:

Yes, by default, Django signals run in the same database transaction as the caller. This means that if you have a signal connected to a model's save event, the signal handler will execute within the same database transaction as the model's save method. If the transaction is rolled back, any changes made by the signal handler will also be rolled back.

models.py  
from django.db import models

from django.db import transaction

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

from django.dispatch import receiver

class MyModel(models.Model):

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

@receiver(pre\_save, sender=MyModel)

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

print("Signal handler called")

# Let's try to raise an exception to test the rollback

raise Exception("Rolling back transaction")

views.py  
from django.http import HttpResponse

from .models import MyModel

def create\_model(request):

try:

instance = MyModel(name="Test")

instance.save() # This will trigger the signal

return HttpResponse("Model saved successfully")

except Exception as e:

return HttpResponse(f"Error: {str(e)}")

urls.py  
from django.urls import path

from .views import create\_model

urlpatterns = [

path('create/', create\_model, name='create\_model'),

]

## Topic: Custom Classes in Python

**Description:** You are tasked with creating a Rectangle class with the following requirements:

1. An instance of the Rectangle class requires length:int and width:int to be initialized.
2. We can iterate over an instance of the Rectangle class
3. When an instance of the Rectangle class is iterated over, we first get its length in the format: **{'length': <VALUE\_OF\_LENGTH>}** followed by the width **{width: <VALUE\_OF\_WIDTH>}**

**Ans:**

class Rectangle:

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

self.length = length

self.width = width

def get\_attributes(self):

# Return length first, then width

return [

{"length": self.length},

{"width": self.width}

]

# Example usage:

rect = Rectangle(10, 5)

# Iterating over the attributes using the custom function

for attribute in rect.get\_attributes():

print(attribute)