**Date : 18/09/2024**

**Name : Lokesh Kumar Jayswal**

**Django Trainee Assignment**

**Task 1:** Django signal built functionality

1. By default are django signals executed synchronously or asynchronously?

**Code Snippet :**

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

from django.dispatch import receiver

from myapp.models import MyModel

@receiver(post\_save, sender=MyModel)

def my\_model\_post\_save(sender, instance, created, kwargs):

print("Signal received and processed.")

# When an instance of MyModel is saved:

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

**Code explanation :**

This code sets up a Django signal that listens for save events on instances of MyModel. When such an event occurs when an instance is created or updated, it executes a function that prints a message indicating that the signal has been processed. This mechanism allows us to implement additional logic in response to model changes without modifying the model's save method directly.

# 

# 2. Do django signals run in the same thread as the caller?

**Code snippet:**

import threading

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

from django.dispatch import receiver

from myapp.models import MyModel

@receiver(post\_save, sender=MyModel)

def my\_model\_post\_save(sender, instance, created, \*\*kwargs):

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

# When an instance of MyModel is saved:

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

**Code explaination:**

I have to setup a Django signal receiver that listens for save events on instances of MyModel. When a new instance is created, it prints out which thread is executing that code, demonstrating how Django signals operate within threading contexts.

# 

# 

3.By default do django signals run in the same database transaction as the caller?

**Code snippet:**

from django.db import transaction

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

from django.dispatch import receiver

from myapp.models import MyModel, AnotherModel

@receiver(post\_save, sender=MyModel)

def my\_model\_post\_save(sender, instance, created, \*\*kwargs):

if created:

AnotherModel.objects.create(my\_model=instance)

# Using transaction.atomic to ensure everything is part of one transaction

with transaction.atomic():

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

**Code explaination:**

This code efficiently manages related database operations using Django's signal framework and transaction management. It ensures that when a new instance of MyModel is saved, a corresponding entry in AnotherModel is also created, all while maintaining data integrity through atomic transactions. If any part of this process fails, no changes will be committed to the database.

**Task 2 :**

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>}**

**Code snippet:**

class Rectangle:

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

self.length = length

self.width = width

def \_\_iter\_\_(self):

yield {'length': self.length}

yield {'width': self.width}

# Usage example

rect = Rectangle(5, 10)

for item in rect:

print(item)

**Code explaination:**

It is a simple representation of a rectangle with an iterable interface. The Rectangle class allows users to create rectangle objects with specified dimensions and provides an easy way to access those dimensions through iteration. Each dimension is presented as a dictionary during iteration, making it clear which value corresponds to which attribute.