Name: Sanskar Singh

Topic: Django 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 execute synchronously. This means that when a signal is sent, the connected receivers are executed one by one in the same order in which they were connected, and the signal sender waits for all receivers to finish before continuing execution.

A synchronous signal is sent and controlled immediately when the event occurs, before the sender resumes its execution. After the sender completes its execution, an asynchronous signal is sent and handled later. Depending on the use case, Django provides both types of signals.

Below is the code snippet that demonstrates signals executing synchronously:

```
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, created, **kwargs):
    if created:
        print(f"Signal received for newly created object: {instance.name}")
    else:
        print(f"Signal received for updated object: {instance.name}")

# Creating or updating an instance of MyModel
obj, created = MyModel.objects.get_or_create(name="Test")

print("Object creation or update completed")  # This will be printed after the signal handler completes
```

In the above code,

I have defined a Django model ‘MyModel’ and a signal handler function ‘my_signal_handler’ connected to the ‘post_save’ signal of ‘MyModel’.

When a instace of ‘MyModel’ is created

(obj, cretaed = MyModel.object.get_or_create(name=”Test”)), a signal is sent and the

signal handler is executed synchronously.

The print statement will execute only after the object has been created and the signal handler completed.

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: No, Django signals are not required to run in the same thread as the caller. But by default, Django signals run synchronously in the same thread as the sender (caller). However, Django also provides the option to execute the signal asynchronously by setting the 'async' parameter to 'True' using the '@receiver' decorator. When a signal is executed asynchronously, it runs in a separate thread or process, allowing the caller to continue its execution without waiting for the signal handler to finish.

Below is the code snippet that demonstrates signals run in the same thread as the caller:

```
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)
    time.sleep(2) # simulate a delay of 2 sec in task execution
    print("Signal handler finished")

# Creating an instance of MyModel
obj = MyModel.objects.create(name="Test")

print("Object created")

# Printing the current thread's ID
print("Current thread ID:", threading.get_ident())
```

In the above code,

I have defined a Django model 'MyModel' and a signal handler function 'my_signal_handler' connected to the 'post_save' signal of 'MyModel'. Inside the signal handler, we simulate a delay of 2 sec in task execution using 'time.sleep(2).

After creating an instance of 'MyModel', I have printed "Object Created" followed by the thread ID using 'threading.get_ident()'. This allows us to compare the thread ID of the main thread with the thread ID inside the signal handler.

When we run this code, we'll notice that both the main thread and the thread inside the signal handler have the same thread ID, indicating that the signal handler is executed in the same thread as the caller (main thread). Additionally, we'll observe that the print statement "Signal handler finished" is printed after a delay of 2 seconds, demonstrating that the signal handler blocks the execution until it completes its task.

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, Django signals run in the same database transaction as the caller by default. This means, when a signal is triggered, the operations performed within the signal handler are part of the same transaction as the operation initiated by the caller.

Django signals are executed within the same atomic block as the caller's database operation. An atomic block ensures that all tasks within it are treated as a single unit of work, and that they all succeed or all fail together.

Below is the code snippet that demonstrates signals run in the same database transaction as the caller:

```
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")

# Printing the name of the object before and after saving
print("Name before save:", obj.name)

# Starting a new transaction explicitly
with transaction.atomic():
    # Updating the object within the transaction
    obj.name = "Updated"
    obj.save() # Save the modified object

# Printing the name of the object after saving
print("Name after save:", obj.name)
```

In the above code,
I have defined a Django model 'MyModel' and a signal handler function 'my_signal_handler' that binds to the 'post_save' signal of 'MyModel'. Inside the signal handler, we modify the 'name attribute' of the instance and save it back to the database.

We create an instance of 'MyModel' with the name "Original" and print its name before and after saving. Then we explicitly start a new transaction using 'transaction.atomic()' and update the name of the object to "Updated" within transaction.

Since the signal handler modifies the same instance and is executed within the same database transaction as the caller, modifications made by the signal handler are visible within the same transaction. Therefore, the object name printed after save reflects the changes made by both the caller and the signal handler, confirming that Django signals run in the same database transaction as the caller by default.

Topic: Custom Classes in Python

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>}`**

```python
class Rectangle:
    def __init__(self, length: int, width: int):
        self.length = length
        self.width = width

    def __iter__(self):
        # Iteration yields the length first, then the width as a tuple
        yield ('length', self.length)
        yield ('width', self.width)

# Example usage
rectangle = Rectangle(10, 5)

# Iterating over the rectangle instance
for dimension, value in rectangle:
    print(f"{dimension}: {value}")
```

Explanation:

- The Rectangle class is initialized with length and width.
- The __iter__ method is defined to make the class iterable. It yields the length and width in the specified format.
- When iterating over an instance of Rectangle, it produces a tuples first for the length

and then for the width, as required.