ACCUKNOX

Questions for Django Trainee at Accuknox

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) In Django signals are executed synchronously by default. This means that when a signal is sent, the sender waits for all receivers to complete their processing before continuing execution.

Example code for better understanding:

```python
from django.db import models

from django.db.models.signals import post_save  # Import the signal triggered after saving a model instance

from django.dispatch import receiver  # Import the decorator for connecting signals to functions

import time  # Import the time module to track execution time




class MyModel(models.Model):

    name = str(models.CharField(max_length=100))
```

```
# Signal receiver function that gets executed after a 'MyModel' instance
is saved
@receiver(post_save, sender=MyModel)  # Decorator to connect
'my_slow_callback' to the 'post_save' signal of 'MyModel'
def my_slow_callback(sender, instance, created, **kwargs):
    #Get start time of the signal receiver
    print(f"Signal receiver started at {time.time()}")
    time.sleep(5)  # A 5-second delay simulation
    # Get the end time of the signal receiver
    print(f"Signal receiver finished at {time.time()}")


def create_model_instance():
    # Time at instance creation
    print(f"Creating model instance at {time.time()}")
    obj = MyModel.objects.create(name="Test")  # New instance creation of
'MyModel' with name "Test"
    # Model instance is creation time.
    print(f"Model instance created at {time.time()}")
```

Conclusion:

Here the signal `post_save` is connected to `my_slow_callback`, and within that function, a delay of 5 seconds is introduced using `time.sleep(5),` thus ,the above code clearly demonstrates that the signal is handled synchronously because the execution of the model instance creation is paused until the signal handler completes its task.

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, In Django signals are typically run in the same thread as the caller by default.

Code example:

```
import threading  # Import threading module to get thread information
from django.db import models  # Import Django's model class for defining
models
from django.db.models.signals import post_save  # Import the post_save
signal
from django.dispatch import receiver  # Import the receiver decorator

# Define a Django model 'MyModel' with one field 'name'
class MyModel(models.Model):
    name = models.CharField(max_length=100)  # CharField with max length
100

# Signal receiver function that gets executed after a 'MyModel' instance
is saved
@receiver(post_save, sender=MyModel)
def my_signal_receiver(sender, instance, created, **kwargs):
    # Get the thread ID of the current thread in which the signal is
executed
    receiver_thread_id = threading.get_ident()
    print(f"Signal receiver running in thread: {receiver_thread_id}")

# Function to create a new instance of 'MyModel'
def create_model_instance():
    # Get the thread ID of the thread running this function (sender
thread)
    sender_thread_id = threading.get_ident()
    print(f"Sender running in thread: {sender_thread_id}")

    # Create an instance of 'MyModel', which triggers the 'post_save'
signal
    obj = MyModel.objects.create(name="Test")
    # Print the thread ID of the sender after the signal is processed
    print(f"Sender thread after signal: {threading.get_ident()}")

# Run the function to create a model instance and trigger the signal
create_model_instance()
```

Code Explanation

1. **Intialls we define a simple `MyModel` with a `name` field.**
2. **Then we create a signal receiver function `my_signal_receiver` that prints the thread ID it's running in.**
3. **Following this , iIn the `create_model_instance` function, we:**
   - **Print the thread ID before creating the model instance, to check the thread on which it's running on .**
   - **Then, we create a new `MyModel` instance, triggering the `post_save` signal**
   - **To check , we print the thread ID again after the signal has been processed.**
4. **When we run `create_model_instance()`, if the signals run in the same thread, we should see the same thread ID printed for all three print statements.**

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, So it means that if the caller's transaction is rolled back, any changes made by the signal receivers will also be rolled back providing consistency.

Code example

```
from django.db import models, transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.core.exceptions import ValidationError

# The MainModel
class MainModel(models.Model):
    name = models.CharField(max_length=100)

# The RelatedModel which is related to MainModel via a ForeignKey
class RelatedModel(models.Model):
    main = models.ForeignKey(MainModel, on_delete=models.CASCADE)   #
ForeignKey
    value = models.IntegerField()

    # Overriding the save method to add custom validation
    def save(self, *args, **kwargs):
        if self.value < 0:  #Checking for correct validation value
            raise ValidationError("Value must be non-negative")
        super().save(*args, **kwargs)

# Signal receiver for MainModel's post_save signal
@receiver(post_save, sender=MainModel)
def main_model_signal(sender, instance, created, **kwargs):
    print(f"MainModel signal: {'created' if created else 'updated'} -
{instance.name}")

# Signal receiver for RelatedModel's post_save signal
@receiver(post_save, sender=RelatedModel)
def related_model_signal(sender, instance, created, **kwargs):
    print(f"RelatedModel signal: {'created' if created else 'updated'} -
{instance.value}")

# The function to test transaction handling with atomic blocks
def test_transaction():
    try:
        with transaction.atomic():
            main = MainModel.objects.create(name="Test Main")
            print("MainModel created")
            related = RelatedModel.objects.create(main=main, value=-1)
            print("RelatedModel created")  #This line shouldnt get
executed due to validation error
    except ValidationError:
        print("Transaction rolled back due to ValidationError")
    # Checking the number of MainModel and RelatedModel instances created
    print(f"MainModel count: {MainModel.objects.count()}")
    print(f"RelatedModel count: {RelatedModel.objects.count()}")


# Run the code
test_transaction()
```

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:
    # Constructor method to initialization
    def __init__(self, length: int, width: int):
        self.length = length
        self.width = width

    # Making object iterable
    def __iter__(self):
        yield {'length': self.length}
        yield {'width': self.width}

# Example usage:
rect = Rectangle(5, 3)  # Creating rectangle object
for item in rect:  # Iterate over the Rectangle object
    print(item)
```