Answers 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.

By default, Django signals are executed synchronously. When a signal is sent, all connected handler functions (receivers) are executed one after another in sequence, and only after all receivers finish execution does the control return to the caller.

This behavior can be demonstrated using the following code snippet:

```
import time
from django.dispatch import Signal

# Create a custom signal
my_signal = Signal()

# Define a handler function
def helper(sender, **kwargs):
    print("Handler started.")
    time.sleep(5)  # Simulate long processing
    print("Handler finished.")

# Connect the handler to the signal
my_signal.connect(helper)

# Emit the signal
print("Signal emitted.")
my_signal.send(sender=None)
print("Signal finished emitting.")
```

Output:

```
Signal emitted.
```

Handler started.
(5-second delay)
Handler finished.
Signal finished emitting.

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.

Yes, by default Django signals run in the same thread as the caller. That means when a signal is sent, all the connected handler functions (also called receivers) are executed in the same thread where `.send()` was called.

You can conclusively prove this using the following code snippet.

```
import threading
from django.dispatch import Signal

# Create a custom signal
my_signal = Signal()

# Define a handler function
def runner(sender, **kwargs):
    print(f"Handler running in thread: {threading.current_thread().name}")

# Connect the handler to the signal
my_signal.connect(runner)

# Print the thread before emitting the signal
print(f"Signal emitted in thread: {threading.current_thread().name}")

# Send the signal
my_signal.send(sender=None)
```

Output:

```
Signal emitted in thread: MainThread
Handler running in thread: MainThread
```

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.

Yes, by default, Django signals run in the same database transaction as the caller. This means that if the caller is inside an active transaction block and the transaction is later rolled back, any database operations performed within the signal handlers will also be rolled back.

This can be demonstrated using the following code snippet:

```
from django.db import transaction, connection
from django.dispatch import Signal
from myapp.models import MyModel

# Create a custom signal
my_signal = Signal()

# Define a signal handler
def my_handler(sender, **kwargs):
    print("Handler started.")
    # This insert will only be committed if the transaction is not rolled back
    MyModel.objects.create(name="Signal Instance")

# Connect the handler
my_signal.connect(my_handler)

# Begin a database transaction
try:
    with transaction.atomic():
        print("Transaction started.")
        # Emit the signal
        my_signal.send(sender=None)
        # Forcefully raise an exception to trigger a rollback
        raise Exception("Rolling back transaction")
except:
    pass

# Check if the database insert from the signal handler was persisted
with connection.cursor() as cursor:
    cursor.execute("SELECT COUNT(*) FROM myapp_mymodel WHERE name = 'Signal
Instance'")
    row = cursor.fetchone()
    print(f"Rows inserted into DB: {row[0]}")
```

Output:

Transaction started.
Handler started.
Rows inserted into DB: 0

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>}`**

An instance of the Rectangle class requires length:int and width:int to be initialized. We can iterate over an instance of the Rectangle class
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 :

```
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}

rect = Rectangle(10, 20)

for dimension in rect:
    print(dimension)
```

Output:

```
{'Length': 10}
{'Width': 20}
```
