Submission By Khagendra Kumar Mandal

To Accuknox India Private Limited

Accuknox assignment question

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.

Answer to Question 1 By default, Django Signals are executed Synchronously. When a signal is triggered, the signal handlers are executed in the same thread and in the same process as the code that triggered, rthe same process as the code that triggered the signal

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.

Answer to Question 2

Yes, Django signals run in the same thread as the code triggers them. Signals do not create new threads for their handlers; they execute in the context of the current thread.

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.

Answer to Question 3

Yes, by default, Django signals run in the same database transaction as the ORM (Object-Relational Mapping) operation(Add, Delete, Update, Display) that triggers them.

Code Snippet:

for all three question

```
# models.py
from django.db import models

class Comment(models.Model):
    PENDING = 'pending'
    SAVED = 'saved'
    STATUS_CHOICES = [
        (PENDING, 'Pending'),
        (SAVED, 'Saved'),
    ]

    name = models.CharField(max_length=100)
    comment = models.CharField(max_length=100)
    status = models.CharField(
        max_length=250,
        choices=STATUS_CHOICES,
        default=PENDING,
    )

    def __str__(self):
        return f'{self.name} - {self.status}'

# signals.py
import time
from django.db.models.signals import post_save, pre_save
from django.dispatch import receiver
from .models import Comment
import threading
import logging
from django.db import transaction

logger = logging.getLogger(__name__)

# Dictionary to store previous status
```

```
previous_statuses = {}

# Signal to run before saving a comment
@receiver(pre_save, sender=Comment)
def before_comment_saved(sender, instance, **kwargs):
    # Track the thread ID for proof that the signal runs in the same thread
    current_thread = threading.get_ident()
    logger.info(f"Before saving comment: {instance.name} - {instance.status} - Thread ID: {current_thread}")
    print(f"Before saving comment: {instance.name} - {instance.status} - Thread ID: {current_thread}")

    if instance.pk in previous_statuses:
        # If the instance already exists, get the previous status
        previous_status = previous_statuses[instance.pk]
    else:
        # If the instance is new, use the current status
        previous_status = instance.status

    # Store the current status before saving
    if instance.pk:
        # Update the previous status for the existing instance
        previous_statuses[instance.pk] = previous_status

    time.sleep(3) # to demonstrate

    # Log the previous status and thread ID
    logger.info(f"Before saving comment (status): {instance.name} - {previous_status} - Thread ID: {current_thread}")
    print(f"Before saving comment (status): {instance.name} - {previous_status} - Thread ID: {current_thread}")

# Signal to run after saving a comment
@receiver(post_save, sender=Comment)
def after_comment_saved(sender, instance, created, **kwargs):
    current_thread = threading.get_ident()
    logger.info(f"After saving comment (signal): {instance.name} - Thread ID: {current_thread}")
    print(f"After saving comment (signal): {instance.name} - Thread ID: {current_thread}")

    if created:
        # For new comments
        logger.info(f"New comment saved: {instance.name} - {instance.status} - Thread ID: {current_thread}")
        print(f"New comment saved: {instance.name} - {instance.status} - Thread ID: {current_thread}")

    else:
        # For existing comments, print both the previous and new status
        previous_status = previous_statuses.get(instance.pk, instance.status)
        logger.info(f"Existing comment updated: {instance.name} - {previous_status} -> {instance.status} - Thread ID: {current_thread}")
        print(f"Existing comment updated: {instance.name} - {previous_status} -> {instance.status} - Thread ID: {current_thread}")

        # Remove the entry from previous_statuses after update
        previous_statuses.pop(instance.pk, None)

    # Demonstrating that signals run in the same transaction as the caller.

    try:
        with transaction.atomic():
            # Simulate an additional action in the transaction within the signal
            logger.info(f"Processing transaction for comment: {instance.name} - {instance.status}")
            print(f"Processing transaction for comment: {instance.name} - {instance.status}")
            # Raise an error to simulate a rollback
            if instance.status == "pending":
                print("Simulating rollback during signal processing.")
            else:
                print("Simulating commit/Saved during signal processing")
    except Exception as e:
        logger.error(f"Transaction completely failed: {str(e)}")
        print("Transaction completely failed", str(e))
```

OUTPUT

when object is created

output log: if user saves with status "pending"

Before saving comment: Khagendra - pending - Thread ID: 2116 Before saving comment (status): Khagendra - pending - Thread ID: 2116 After saving comment (signal): Khagendra - Thread ID: 2116 New comment saved: Khagendra - pending - Thread ID: 2116 Processing transaction for comment: Khagendra - pending Simulating rollback during signal processing.

==

when existing object changed

output log: if user saves with status "saved"

Before saving comment: Khagendra - saved - Thread ID: 2116 Before saving comment (status): Khagendra - saved - Thread ID: 2116 After saving comment (signal): Khagendra - Thread ID: 2116 Existing comment updated: Khagendra - saved -> saved - Thread ID: 2116 Processing transaction for comment: Khagendra - saved Simulating commit/Saved during signal processing

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>}`

```python
class Rectangle:
    def __init__(self, length: int, width: int):
      """
      Initialize a Rectangle object with the given length and width.

      Args(input):
        length (int): The length of the Rectangle.
        width (int): The width of the Rectangle.
      """
      if not isinstance(length, int) or not isinstance(width, int):
        raise TypeError("Length and width must be integers.")
      if length <= 0 or width <= 0:
        raise ValueError("Length and width must be positive integers.")

      self.length = length
      self.width = width
      self._attributes = [{'length': self.length}, {'width': self.width}]
      self._iter_index = 0

    def __iter__(self):
      """Return the iterator object itself."""
      return self

    def __next__(self):
      """Return the next item in the iteration."""
      if self._iter_index < len(self._attributes):
        result = self._attributes[self._iter_index]
        self._iter_index += 1
        return result
      else:
        raise StopIteration


rect = Rectangle(5, 10)
for item in rect:
    print(item)
```

```
{'length': 5}
{'width': 10}
```