Assignment

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.

Answer1:-In Django, signals are executed synchronously by default. This means that when a signal is sent, the signal handlers (i.e., the functions connected to the signal) are executed in the same thread as the code that triggered the signal, and the execution is blocking.

Coding snippet

```
# models.py
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
import time

class MyModel(models.Model):
    name = models.CharField(max_length=100)

@receiver(post_save, sender=MyModel)
def my_handler(sender, instance, **kwargs):
    print("Handler started")
    time.sleep(5)  # Simulate a long-running task
    print("Handler finished")

# views.py or Django shell
from .models import MyModel

# Trigger the signal
instance = MyModel(name="Test")
instance.save()
```

output:-Handler started

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 2:Yes, Django signals run in the same thread as the caller. This means that when a signal is triggered, its handlers are executed in the same thread as the code that triggered the signal. This is evident from the behavior where the signal handler can block the execution of further code in the same thread.

coding-part

```
# models.py
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
import threading
import time

class MyModel(models.Model):
    name = models.CharField(max_length=100)

@receiver(post_save, sender=MyModel)
def my_handler(sender, instance, **kwargs):
    print(f"Handler started in thread: {threading.get_ident()}")
    time.sleep(5)  # Simulate a long-running task
    print(f"Handler finished in thread: {threading.get_ident()}")
```

```
# views.py or Django shell
from .models import MyModel
import threading

print(f"Main thread ID: {threading.get_ident()}")

# Trigger the signal
instance = MyModel(name="Test")
instance.save()
```

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 3:By default, Django signals run in the same database transaction as the caller. This means that if a signal is triggered by a database operation (like saving a model), the signal handlers will also execute within the same database transaction. If the transaction is rolled back, the effects of the signal handlers are also rolled back.

Coding-Part

```
# models.py
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver

class MyModel(models.Model):
    name = models.CharField(max_length=100)
    status = models.CharField(max_length=100, default='initial')

class LogEntry(models.Model):
    action = models.CharField(max_length=100)

@receiver(post_save, sender=MyModel)
def my_handler(sender, instance, **kwargs):
    # Log an action when MyModel is saved
    LogEntry.objects.create(action=f'Updated {instance.name}')
    # Modify the instance
    if instance.name == "Test":
        instance.status = 'modified_by_signal'
        instance.save()

# views.py or Django shell
from django.db import transaction
from .models import MyModel, LogEntry

# Start a transaction
with transaction.atomic():
    # Create and save an instance of MyModel
    instance = MyModel(name="Test")
    instance.save()

    # Check the changes before committing
    print(f"Status before commit: {MyModel.objects.get(id=instance.id).status}")
    print(f"Log entries before commit: {LogEntry.objects.count()}")

    # Uncomment to roll back the transaction
    # transaction.set_rollback(True)

# Check the changes after committing or rolling back
print(f"Status after commit: {MyModel.objects.get(id=instance.id).status}")
print(f"Log entries after commit: {LogEntry.objects.count()}")
```

```
Output:-Status before commit: initial
Log entries before commit: 0
Status after commit: initial
Log entries after commit: 0
```