Accuknox:

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.

Solution:

A signal is a message that is sent by a sender to one or more receivers. A sender is any object that triggers a signal, such as a model instance, a user, or a custom class. A receiver is any callable function that listens for a signal and performs some action when it receives it.

A signal can carry some data along with it, such as the sender object, the instance of the model that changed, or any other information that is relevant to the event. The receiver can access this data through the arguments of the signal handler function.

A signal can be either synchronous or asynchronous. A synchronous signal is sent and handled immediately when the event occurs, before the sender resumes its execution. An asynchronous signal is sent and handled later, after the sender has finished its execution. Django provides both types of signals, depending on the use case.

Bydefault django signals executed synchronously. Recently django 5.0 introduced us to asynchronous signals.

Example:

Defining signals:

from django.dispatch import Signal  
  
# Define a custom signal that carries the user and the post objects  
post\_liked = Signal(providing\_args=["user", "post"])

Defining receiver for the signals

from django.dispatch import receiver  
  
# Define a receiver function that prints a message when a post is liked  
@receiver(post\_liked)  
def post\_liked\_handler(sender, user, post, \*\*kwargs):  
 print(f"{user} liked {post}")  
  
# Connect the receiver function to the post\_liked signal  
post\_liked.connect(post\_liked\_handler)

#or

from django.dispatch import receiver  
  
# Define and connect a receiver function using the decorator  
@receiver(post\_liked)  
def post\_liked\_handler(sender, user, post, \*\*kwargs):  
 print(f"{user} liked {post}")

Utilising builtin signals:

from django.db.models.signals import post\_save  
from django.dispatch import receiver  
from django.contrib.auth.models import User  
  
# Define and connect a receiver function that sends a welcome email to new users  
@receiver(post\_save, sender=User)  
def send\_welcome\_email(sender, instance, created, \*\*kwargs):  
 if created:  
 # Send email logic here  
 print(f"Welcome email sent to {instance.email}")

Defining and sending custom signals:

from django.dispatch import Signal  
  
# Define a custom signal that carries the user and the post objects  
post\_liked = Signal(providing\_args=["user", "post"])  
  
# Import your custom signal in your views module  
from .signals import post\_liked  
  
# Send your custom signal from your view function  
def like\_post(request, post\_id):  
 # Get the user and the post objects  
 user = request.user  
 post = Post.objects.get(id=post\_id)  
  
 # Like logic here  
  
 # Send the post\_liked signal with user and post as arguments  
 post\_liked.send(sender=post.\_\_class\_\_, user=user, post=post)

1. Do django signals run in the same thread as the caller? Please support your answer with a code snippet that conclusively proves your stance.

**Yes, Django signals run in the same thread as the caller.** This means that the signal handler function is executed within the same thread of execution as the code that triggered the signal.

Example:

import threading

from django.db.models.signals import post\_save

from django.dispatch import receiver

# Create a signal handler function that prints the thread ID

@receiver(post\_save, sender=MyModel)

def my\_signal\_handler(sender, instance, \*\*kwargs):

print(f"Signal handler executed in thread: {threading.get\_ident()}")

# Create a new instance of MyModel

my\_model\_instance = MyModel.objects.create(field1="value1", field2="value2")

# The signal handler will be executed immediately after the instance is created

In this example, the my\_signal\_handler function will be called directly after the MyModel instance is created. The threading.get\_ident() function will print the thread ID of the current thread, which will be the same as the thread that executed the MyModel.objects.create() method. This demonstrates that the signal handler is running in the same thread as the caller.

1. 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.

**No, Django signals do not run in the same database transaction as the caller by default.** This means that the signal handler function is executed in a separate database transaction from the transaction that triggered the signal.

from django.db.models.signals import post\_save

from django.dispatch import receiver

# Create a signal handler function that attempts to create a new record

@receiver(post\_save, sender=MyModel)

def my\_signal\_handler(sender, instance, \*\*kwargs):

try:

NewModel.objects.create(related\_model=instance)

except Exception as e:

print(f"Error creating new record: {e}")

# Create a new instance of MyModel

my\_model\_instance = MyModel.objects.create(field1="value1", field2="value2")

# The signal handler will be executed in a separate transaction

In this example, the my\_signal\_handler function will be executed in a separate database transaction from the transaction that created the MyModel instance. This means that if an error occurs while creating the NewModel instance in the signal handler, the original transaction will not be rolled back.

If you want the signal handler to run in the same transaction as the caller, you can use the atomic decorator from Django's database API:

from django.db import transaction

@receiver(post\_save, sender=MyModel)

@transaction.atomic

def my\_signal\_handler(sender, instance, \*\*kwargs):

# ...

With the atomic decorator, the signal handler will be executed within the same transaction as the caller, and any errors that occur in the signal handler will cause the entire transaction to be rolled back.

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>}**

class Rectangle:

def \_\_init\_\_(self, length, width):

self.length = length

self.width = width

def \_\_iter\_\_(self):

yield {'length': self.length}

yield {'width': self.width}

# Example usage:

rect = Rectangle(5, 3)

for item in rect:

print(item)

This code defines a Rectangle class that meets the given requirements:

* The \_\_init\_\_ method initializes the length and width attributes.
* The \_\_iter\_\_ method defines how to iterate over an instance of the Rectangle class. It yields two dictionaries containing the length and width values, respectively.
* The example usage demonstrates how to create a Rectangle object and iterate over it, printing the length and width values in the desired format.