“ACCUKNOX”

Name:: Ashu Tiwari
Roll no:: cs21b007
Email id:: cs21b007@iittp.ac.in;

Folder name:: Accunox_f;
Django project name:: my_project;
Apps defined as ::

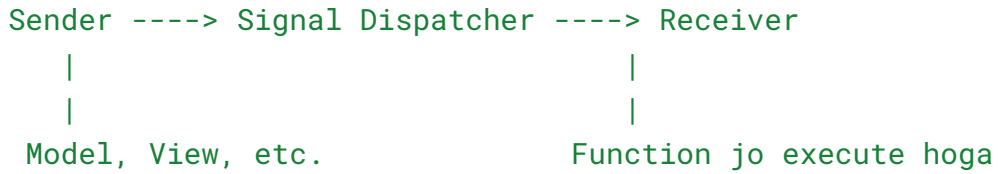
question_1,
question_2,

question_3,
Question_4;

Question 1: By default are Django signals executed synchronously or asynchronously?

Answer: By default, Django signals are executed **synchronously**. This means that when a signal is sent, the signal handlers are executed in the same thread and process as the code that triggered the signal. The main request or operation will wait for the signal handlers to complete their execution before continuing.

Django Signal ka Flow Diagram



Supporting Code Snippet

To demonstrate the synchronous execution of Django signals, we consider the following example:

Define a Signal and a Signal Handler

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

@receiver(post_save, sender=MyModel)
def my_handler(sender, instance, **kwargs):
    # This function will be executed synchronously
    print("Signal handler executed")
```

Creating of a View that Triggers the Signal

```
# views.py
from django.http import HttpResponse
from .models import MyModel
import datetime

def create_model_instance(request):
    print(f"View started at {datetime.datetime.now()}")
    obj = MyModel.objects.create(name="Test Instance")
    # obj.save()
    print(f"View finished at {datetime.datetime.now()}")
    return HttpResponse("Check the console for timestamp logs")
```

Access the View in a Web Browser

Visit the URL for the `create_model_instance` view to test the functionality:

`http://127.0.0.1:8000/create/`

Expected Behavior

When you access the URL, the `create_model_instance` view performs the following actions:

1. **Timestamp Logging**: The view logs the start and end timestamps to the console, showing the exact times the view begins and finishes execution.

2. **Model Instance Creation**: A new instance of `MyModel` is created with the name `"Test Instance"`, and the `create()` method is used to save the instance to the database.

Completion: The view completes execution and returns an HTTP response to the client with the message:
`Check the console for timestamp logs`

Console Output

When the view is accessed, the following timestamps will be printed to the console:

```
View started at [start_timestamp]

View finished at [end_timestamp]
```

These logs help visualize the time taken by the view to complete execution, including the database operation to create a model instance.

Signal Consideration

Since this view does not involve any Django signals directly, no signal handlers are triggered here. However, if we had a `post_save` signal attached to the `MyModel` instance, the signal would be executed immediately after the model instance is created and saved, occurring within the same request thread.

Summary

The `create_model_instance` view demonstrates the creation of a new model instance while logging timestamps to the console to track the execution time. This example confirms that the model instance is created and committed to the database before the view returns the final HTTP response to the user.

Question_2::: Do django signals run in the same thread as the caller?

Ans:::

```
#importnat note --> superuser username :: ACCUKNOX_F(isi se login krna hai
nhin to error aaega)
# email :: cs21b007@iittp.ac.in
# password :: ashu()09;
```

Yes, Django signals run in the same thread as the caller. This means that when a signal is triggered, the signal handlers are executed in the same thread as the request that triggered the signal. The request processing will wait for the signal handlers to complete their execution before proceeding.

Access the View in a Web Browser

Visit the URL for the `trigger_signal_view`:

`http://127.0.0.1:8000/trigger-signal/`

Expected Behavior

When we access the URL, the `trigger_signal_view` performs the following actions:

1. **Main Thread Logging:** The view logs the name of the main thread to the console, showing that the view is executed in a specific thread.
2. **User Instance Creation:** The view creates a new `User` instance with the username `"signal_test_user"`. This triggers the `post_save` signal.
3. **Signal Handler Execution:** The signal handler logs the thread name and confirms that the signal has processed the new user creation.

Completion: After the signal handling, the view returns an HTTP response to the client with the message:
`Signal triggered, check your console output.`

Console Output:

When this URL is accessed, the following will be printed to the console:

```
Main Thread: MainThread

Signal Handler Thread: MainThread

User instance saved: signal_test_user
```

This output confirms that both the view and the signal handler execute in the same thread, indicating synchronous behavior.

Summary

This code snippet demonstrates that Django signals are executed synchronously. When a `User` instance is created in the view, the `post_save` signal is triggered and processed within the same thread, completing its operation before the view finishes execution and sends the response.

Question_3:::

Access the View in a Web Browser

Here are the URLs for each operation:

Create User:
http://127.0.0.1:8000/create-user/

Update User:

http://127.0.0.1:8000/update-user/

Create User with Rollback:

```
http://127.0.0.1:8000/create-user-with-rollback/
```

Expected Behavior for Each View

---

1. Create User Profile

- Description: This view creates a new `UserProfile` instance with the name `"Ashu"` and email `"ashu@example.com"`. It saves the user to the database and returns an HTTP response.
- URL: `http://127.0.0.1:8000/create-user/`

Expected Behavior:

- A new user profile is created in the database.

The view returns the following HTTP response:

```
User profile created: Ashu
```

Database Action:

- A new row is added to the `UserProfile` table with the provided data.

---

2. Update User Profile

- Description: This view looks for the user named `"Ashu"`, updates their email to `"newemail@example.com"`, and saves the changes to the database. If the user doesn't exist, it returns an error message.
- URL: `http://127.0.0.1:8000/update-user/`

Expected Behavior:

If the user exists, their email is updated, and the view returns the following HTTP response:

```
User profile updated: Ashu
```

If the user doesn't exist, the response will be:

```
User does not exist
```

Database Action:

- If successful, the existing user's email will be updated in the `UserProfile` table.

---

3. Create User with Rollback (Transaction Testing)

- Description: This view attempts to create a new `UserProfile` instance, but deliberately raises an exception to trigger a rollback of the database transaction.
- URL: `http://127.0.0.1:8000/create-user-with-rollback/`

Expected Behavior:

- The view initiates the creation of a new user with the name `"Ashu Bhai"` and email `"ashuu@example.com"`.
- However, due to the simulated exception, the transaction is rolled back, and no data is saved.

The view returns the following HTTP response:

```
Error: Simulated rollback error
```

Database Action:

- No data will be committed to the database due to the transaction rollback. The `UserProfile` creation will be undone.

Console Output and Signal Behavior

To support the stance that Django signals run in the same database transaction as the caller, you can log activity within the `post_save` signal handler.

Here's a demonstration of how signals behave in each case:

1. Create User Profile: The `post_save` signal will be triggered after the new `UserProfile` is successfully created, and the signal handler will log that the profile was created. The signal runs within the same transaction, meaning the signal only executes after the data has been committed to the database.

2. Update User Profile: When the `UserProfile` is updated, the `post_save` signal is triggered, logging the update to the console. The signal runs synchronously after the model's `.save()` method is called.
3. Create User with Rollback: The signal will not be triggered because the transaction is rolled back due to the simulated error. Since the data was not committed to the database, the signal is never executed, proving that signals operate within the same transaction context.

Question_4::

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>}`**

Answer::

Access the View in a Web Browser

Visit the URL for the `rectangle_view` to see the result of the rectangle iteration:

`http://127.0.0.1:8000/rectangle/`

Class Requirements and Explanation

we are tasked with creating a `Rectangle` class with the following specifications:

1. Attributes:
    - Each instance of the `Rectangle` class requires a `length` and `width` to be initialized.
2. Iterable:
    - we can iterate over an instance of the `Rectangle` class, where it returns:
        - First, a dictionary in the format `{'length': <VALUE_OF_LENGTH>}`.
        - Then, a dictionary in the format `{'width': <VALUE_OF_WIDTH>}`.

Expected Behavior

When you access the URL for the `rectangle_view`, the following occurs:

1. **Rectangle Instance Creation:**
    - A `Rectangle` instance is created with three sets of dimensions: `(length=10, width=5)`, `(length=15, width=7)`, and `(length=20, width=10)`.
2. **Iteration Over the Rectangle:**
    - The view iterates over each `Rectangle` object and collects the results of the iteration into a list.
    - For each rectangle, the iteration will first return the `length` in the format: `{'length': value}` and then the `width` in the format: `{'width': value}`.
3. **Response Generation:**
    - The results of the iteration are concatenated into an HTML response using `<br>` to display each item on a new line.

Console Output and HTTP Response

When you visit the `rectangle_view` URL, the output of iterating over the rectangles will be displayed in the browser.

Example HTTP Response:

For the given example, the output will be:

```
{'length': 10}
{'width': 5}
{'length': 15}
{'width': 7}
{'length': 20}
{'width': 10}
```

The values of `length` and `width` correspond to the dimensions provided when creating the `Rectangle` instance.

Summary

The `rectangle_view` demonstrates how to iterate over a `Rectangle` instance and return its dimensions (`length` and `width`) in a specified format. The view collects these results and returns them as an HTTP response, displaying each length and width as a dictionary on a new line in the browser.