Answers

In order to get started:

Creating a virtual environment to have a dedicated space for the django project file

Command line:

python -m venv myworld

myworld\Scripts\activate.bat

install the requirements like pip and django using terminal

Creating PROJECT and APP:

Example:

django-admin startproject Rectangle

APP Name: * go to rectangle folder location in cmd**

python manage.py startapp members

to run server - python manage.py runserver

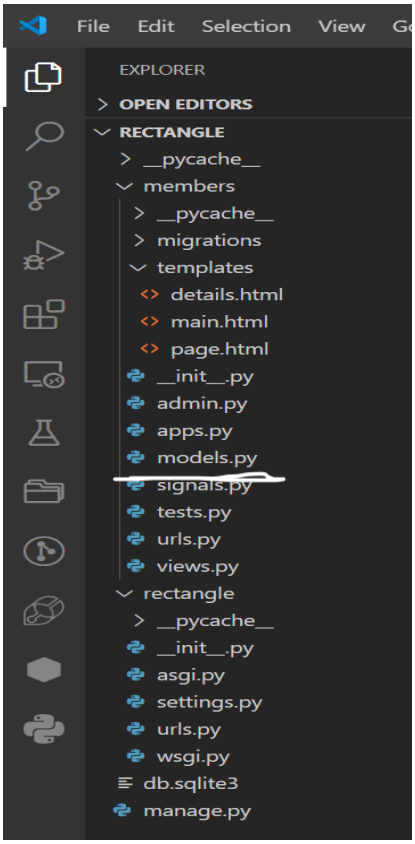
```
C:\Users\thang\Downloads>myworld\Scripts\activate.bat

(myworld) C:\Users\thang\Downloads>
```

```
(myworld) C:\Users\thang\Downloads>cd rectangle

(myworld) C:\Users\thang\Downloads\rectangle>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
April 08, 2025 - 22:54:15
Django version 5.2, using settings 'rectangle.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

WARNING: This is a development server. Do not use it in a production setting. Use a production WSGI or ASGI server instead.
For more information on production servers see: https://docs.djangoproject.com/en/5.2/howto/deployment/
```

Creating database

To create database , navigate to the models.py file in the /members/ folder.

Open it, and add a Member table by creating a Member class, and describe the table fields in it:



main.html page.html details.html views.py models.py signals.py admin.py

mbers > models.py > Member

```python
from django.db import models
from django.contrib.auth.models import User
# Create your models here.
class Member(models.Model):
    firstname = models.CharField(max_length=255)
    lastname = models.CharField(max_length=255)
    phone = models.IntegerField(null=True)
    joined_date = models.DateField(null=True)

    def __str__(self):
        return f"{self.firstname} {self.lastname} {self.joined_date}"


class Rectangle(models.Model):
    length = models.IntegerField()
    width = models.IntegerField()

    def __iter__(self):
        yield {'length': self.length}
        yield {'width': self.width}

    def __str__(self):
        return f"Rectangle(length={self.length}, width={self.width})"

from django.db.models.signals import post_save
from django.dispatch import receiver
import time

@receiver(post_save, sender=Member)
def members_list_saved(sender, instance, created, **kwargs):
    print("Signal handler started")
    time.sleep(5)  # Simulate a long task
    print(f"{sender},{instance.firstname} details has been saved! Created: {created}")

@receiver(post_save, sender=Rectangle)
def fail_signal(sender, instance, created, **kwargs):
    print("Signal running... About to raise an error!")
    raise Exception("Oops! Signal failed.")
```

OBLEMS 20 OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 20

```
r more information on production servers see: https://docs.djangoproject.com/en/5.2/howto/deployment/
\Users\thang\Downloads\rectangle\members\models.py changed, reloading.
```

20 Sourcery

Created simple database member list to insert and get members name, phone no, joined date etc

And another database model rectangle to iterate in loop to print the given value and store the length and width values

After saving the file make sure to execute

******* python manage.py makemigrations members**

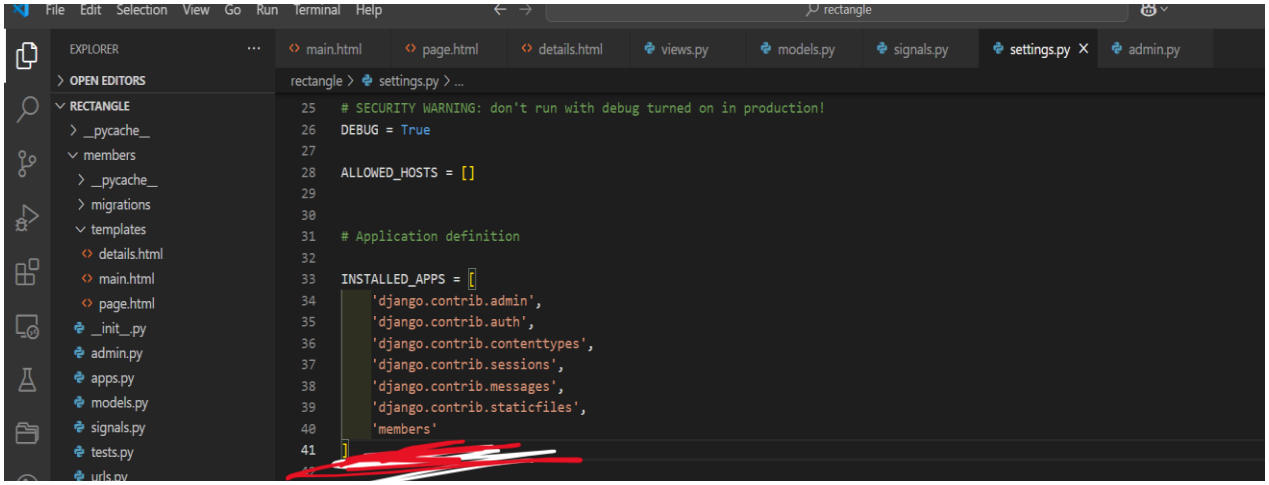
To finish creating database

And

***** python manage.py migrate**

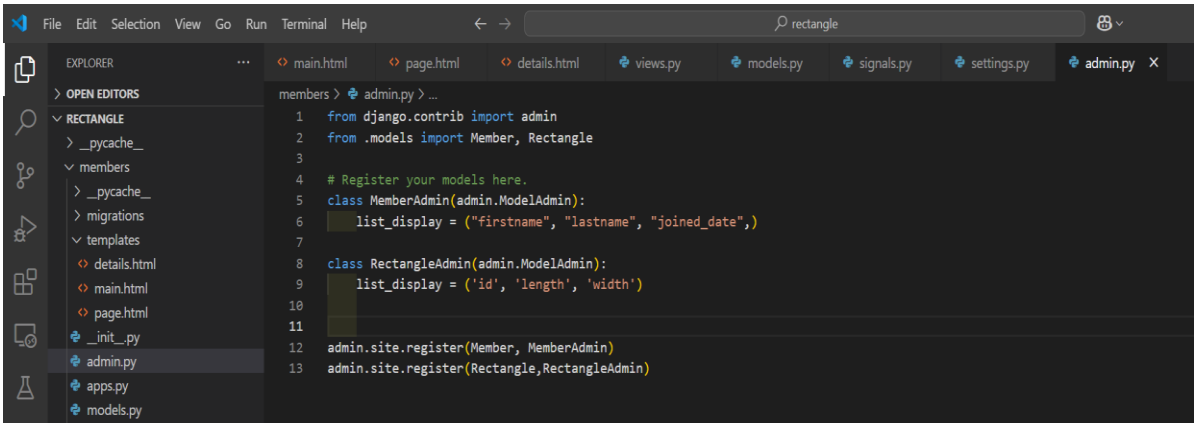
to apply changes and run the server to check changes

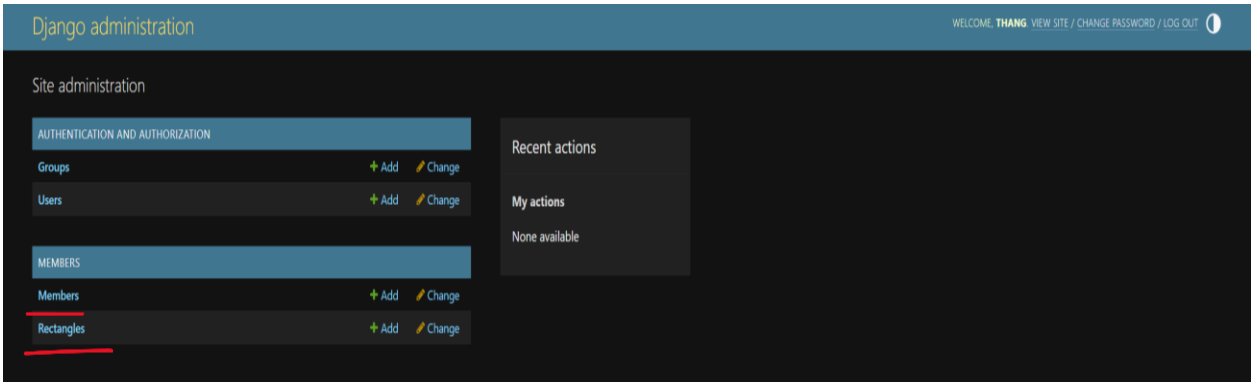
In settings.py mention your app name ex. Members



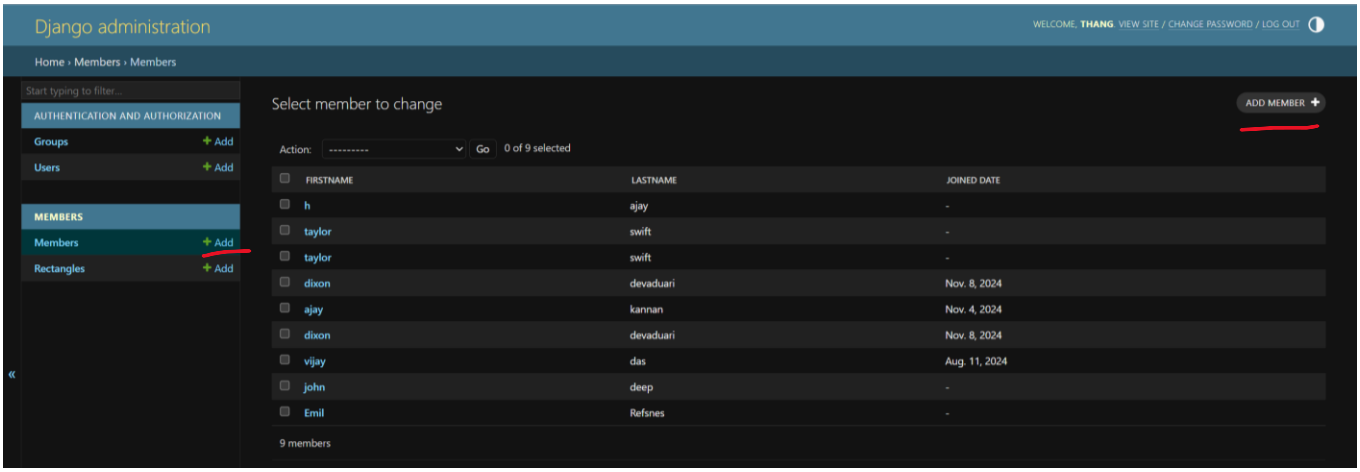
To view the database

Modify the admin.py file





To add data we can use either the admin page or using the python shell commang in terminal



Example to insert data- Type in terminal to start python shell

Python manage.py shell

```
class Member(models.Model):
    firstname = models.CharField(max_length=255)
    lastname = models.CharField(max_length=255)
    phone = models.IntegerField(null=True)
    joined_date = models.DateField(null=True)

    def __str__(self):
        return f"{self.firstname} {self.lastname} {self.joined_date}"
```

Import the model class member:

Form members.models import Member

m= Member(firstname='Jane', lastname='Doe')

IF needed phone no and joined date can also be given or else null value added as default

m.save()

to commit changes

Member.objects.all().values()

and

Member.objects.all()

These two command used to view to query set in shell terminal

1 & 2. By default, Django signals are executed synchronously. This means that when a signal is sent, all connected receiver functions are executed immediately, one after the other, in the same thread as the sender. The sender's execution will be blocked until all receivers have finished processing

To prove:

In model.py

When ever the request to add new members

the signal line does **not finish** until:

1. The member instance is created in the DB.
2. The post_save signal is triggered.
3. The members_list__saved function **runs completely**.
4. In inside the members_list__saved funtion It will block for 5 seconds, because time.sleep(5) runs in the **same thread**.
5. Then finally, control returns to your code.

Even though the signal *feels* like an "event", it still runs inline and blocking — so it's **synchronous**.

```python
from django.contrib.auth.models import User

from django.db.models.signals import post_save
from django.dispatch import receiver
import time

# Create your models here.
class Member(models.Model):
    firstname = models.CharField(max_length=255)
    lastname = models.CharField(max_length=255)
    phone = models.IntegerField(null=True)
    joined_date = models.DateField(null=True)

    def __str__(self):
        return f"{self.firstname} {self.lastname} {self.joined_date}"


@receiver(post_save, sender=Member)
def members_list_saved(sender, instance, created, **kwargs):
    print("Signal handler started")
    time.sleep(5)  # Simulate a long task
    print(f"{sender},{instance.firstname} details has been saved! Created: {created}")
```

```
>>> m= Member(firstname='Stale', lastname='Refsnes')
>>> m.save()
Signal handler started
```

24 Sourcery

After 5 sec

```
>>> m= Member(firstname='Stale', lastname='Refsnes')
>>> m.save()
Signal handler started
Stale details has been saved! Created: True
>>>
```

24 Sourcery

3. Yes, by default, Django signals run in the same database transaction as the caller—but with some important action depending on when the signal is triggered and how it's connected.

Signals like pre_save, post_save, pre_delete, post_delete:

These signals are called during the execution of the Django ORM methods (e.g., .save(), .delete()).

```
# pre_save -> instance  -> my_handler
instance = User.objects.create() # save User data in the database
# post_save -> instance, created=True -> my_handler

# pre_save -> instance  -> my_handler
instance.save()
# post_save -> instance, created=False  -> my_handler

# pre_delete -> instance  -> my_handler
instance.delete()
# post_delete -> instance  -> my_handler
```

```
nal  Help                                                    rectangle
in.html   page.html   details.html   views.py   models.py ×   sig
bers > models.py > fail_signal

class Rectangle(models.Model):
    length = models.IntegerField()
    width = models.IntegerField()

    def __iter__(self):
        yield {'length': self.length}
        yield {'width': self.width}

    def __str__(self):
        return f"Rectangle(length={self.length}, width={self.width})"


@receiver(pre_save, sender=Rectangle)
def fail_signal(sender, instance, **kwargs):
    print("Signal running... About to raise an error!")
    raise Exception("Oops! Signal failed.")
```

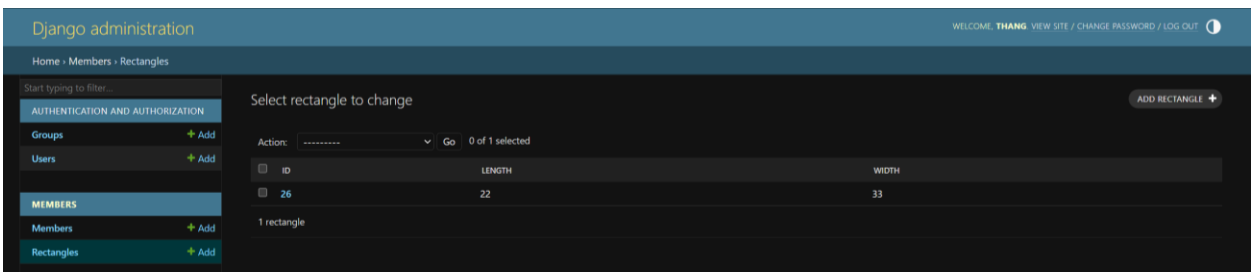
```
>>> from members.models import Rectangle
>>> r=Rectangle(length=22,width=33)
>>> r.save()
Signal running... About to raise an error!
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\thang\downloads\myworld\Lib\site-packages\django\db\models\base.py", line 902, in save
    self.save_base(
  File "C:\Users\thang\downloads\myworld\Lib\site-packages\django\db\models\base.py", line 988, in save_base
    pre_save.send(
  File "C:\Users\thang\downloads\myworld\Lib\site-packages\django\dispatch\dispatcher.py", line 189, in send
    response = receiver(signal=self, sender=sender, **named)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\thang\Downloads\rectangle\members\models.py", line 41, in fail_signal
    raise Exception("Oops! Signal failed.")
Exception: Oops! Signal failed.
```

The commit r.save() the pre_save function execute and the value didn't save on data base

But on post_save() function the value gets commited to database and then execute the function excpection

```
class Rectangle(models.Model):
    length = models.IntegerField()
    width = models.IntegerField()

    def __iter__(self):
        yield {'length': self.length}
        yield {'width': self.width}

    def __str__(self):
        return f"Rectangle(length={self.length}, width={self.width})"


@receiver(post_save, sender=Rectangle)
def fail_signal(sender, instance, **kwargs):
    print("Signal running... About to raise an error!")
    raise Exception("Oops! Signal failed.")
```

```
Command Prompt - python r
(1, {'members.Rectangle': 1})
>>> x=Rectangle.objects.all()
>>> x
<QuerySet []>
>>> exit
Use exit() or Ctrl-Z plus Return to exit
>>> exit()

(myworld) C:\Users\thang\Downloads\rectangle>python manage.py shell
8 objects imported automatically (use -v 2 for details).

Python 3.12.9 (tags/v3.12.9:fdb8142, Feb  4 2025, 15:27:58) [MSC v.1942 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> r=Rectangle(length=22,width=33)
>>> r.save()
Signal running... About to raise an error!
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\thang\downloads\myworld\Lib\site-packages\django\db\models\base.py", line 902, in save
    self.save_base(
  File "C:\Users\thang\downloads\myworld\Lib\site-packages\django\db\models\base.py", line 1023, in save_base
    post_save.send(
  File "C:\Users\thang\downloads\myworld\Lib\site-packages\django\dispatch\dispatcher.py", line 189, in send
    response = receiver(signal=self, sender=sender, **named)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\thang\Downloads\rectangle\members\models.py", line 43, in safe_signal
    raise Exception("Oops! Signal failed.")
Exception: Oops! Signal failed.
>>>
```



4. Rectangle class in python:

```python
class Rectangle(models.Model):
    length = models.IntegerField()
    width = models.IntegerField()

    def __iter__(self):
        yield {'length': self.length}
        yield {'width': self.width}

    def __str__(self):
        return f"Rectangle(length={self.length}, width={self.width})"
```

Notes:

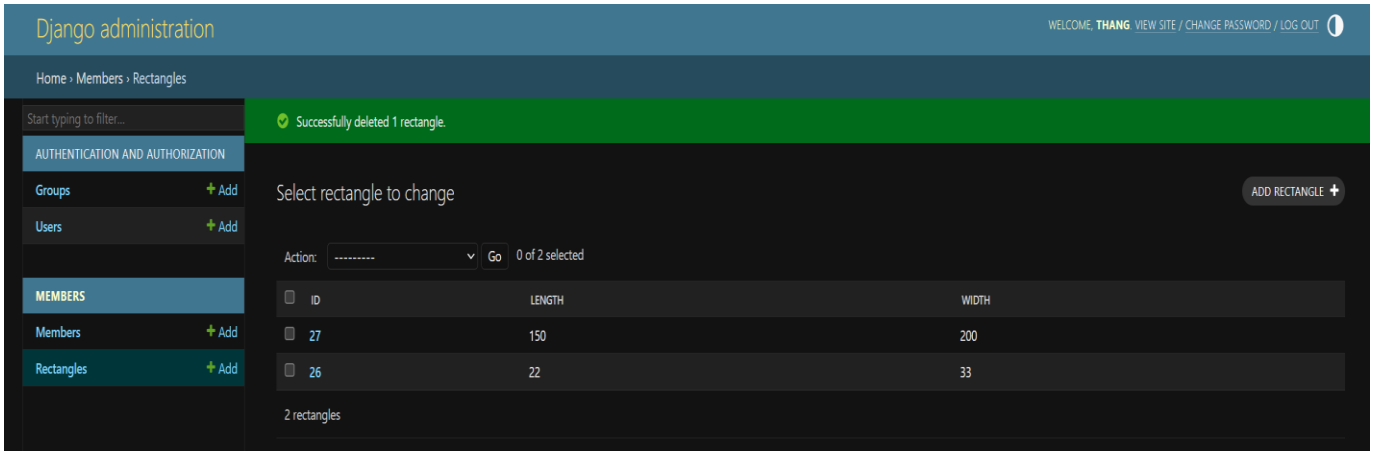
This model is stored in your database, so each rectangle is persistent.

The __iter__ method allows you to loop over,
Iteration yields:

First: {'length': <value>}

Then: {'width': <value>}

```
>>> r=Rectangle(length=150,width=100)
>>> for i in r:
...     print(i)
...
{'length': 150}
{'width': 100}
>>>
```



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.

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.

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.

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>}`**