**Video no : 62**

**Code file no : 53**

**Django Signals – Complete Notes**

**Presave is use for making slug**

**Predelete or post delete is used to blog post sa ager koi image assosiate ho gyea or wo hmare database or cloud ma store ha tu use remove kerna liea means jb hmera blog delete ho to image be delete ho jaye**

**Please see below at the end our own code and note**

# **Real-World Example: Restrict User After 3 Failed Login Attempts**

### Step 1: Use Authentication Signal

Django provides:

* user\_login\_failed → fired when a login attempt fails.
* user\_logged\_in → fired when login is successful.

We’ll use these to **track failed login attempts** and block a user after 3 failures

### Other Real-World Uses of Signals

* **Send email after user registers** (post\_save on User).
* **Auto-create user profile** when a new user is created.
* **Track last login/logout times** using user\_logged\_in / user\_logged\_out.
* **Notify admin on critical events** like too many failed logins.
* **Clear cache** when model data changes (post\_save/post\_delete)

**Here are chatgpt note**

**🔹 What are Signals in Django?**

* **Definition**:  
  A signal is a Django utility that allows **decoupled communication between different parts of an application**.  
  In other words, signals are **notifications** that something has happened in Django, and other parts of your app can listen to them and react.
* **Why we use signals?**
  + To execute **extra logic automatically** when certain events occur.
  + Helps in keeping the code **clean and loosely coupled** (no need to modify the main logic).
  + Useful for logging, auditing, notifications, updating related models, sending emails, etc.

**🔹 How Signals Work (Theory)**

1. **Sender** → The component that triggers the signal.  
   Example: A User model when a new user is created.
2. **Signal** → The actual event (pre\_save, post\_save, login, logout, etc.).
3. **Receiver (Listener/Handler)** → A function that executes when the signal is triggered.
4. **Dispatcher** → Django’s internal system that connects **sender → signal → receiver**.

**🔹 Built-in Django Signals (Categories)**

Django provides many signals. The common ones are:

**1. Model Signals**

Triggered when something happens to a model instance.

* pre\_save → before saving an object.
* post\_save → after saving an object.
* pre\_delete → before deleting.
* post\_delete → after deleting.
* m2m\_changed → when many-to-many relations are updated.

👉 **Use case**: Send a welcome email after a new user is created.

**2. Authentication Signals**

Triggered during user login/logout actions.

* user\_logged\_in
* user\_logged\_out
* user\_login\_failed

👉 **Use case**: Log user login attempts or track activity.

**3. Request/Response Signals**

Triggered during HTTP requests.

* request\_started
* request\_finished
* got\_request\_exception

👉 **Use case**: Measure request performance, log request failures.

**4. Management Signals**

Triggered during management commands (like migrations).

* pre\_migrate
* post\_migrate

👉 **Use case**: Load initial data after migrations.

**5. Database Signals**

* connection\_created → when a new database connection is created.

**🔹 Example of Signal in Django**

**✅ Post Save Example**

Suppose you want to send a **welcome email when a new user is registered**:

# signals.py

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

from django.contrib.auth.models import User

from django.dispatch import receiver

@receiver(post\_save, sender=User)

def send\_welcome\_email(sender, instance, created, \*\*kwargs):

if created: # only when a new user is created

print(f"Welcome {instance.username}, email sent!")

* **Sender**: User model
* **Signal**: post\_save
* **Receiver**: send\_welcome\_email function
* **Trigger**: When a new User is saved

**✅ User Login Example**

Log user login time:

# signals.py

from django.contrib.auth.signals import user\_logged\_in

from django.dispatch import receiver

import datetime

@receiver(user\_logged\_in)

def log\_user\_login(sender, request, user, \*\*kwargs):

print(f"{user.username} logged in at {datetime.datetime.now()}")

**🔹 When to Use Signals (Best Practices)**

✅ Use signals when:

* You need **side effects** after an event (send email, log data, cache clearing).
* You want **decoupled apps** (avoid directly writing logic inside views/models).
* Example: E-commerce app → After order placed, trigger invoice creation, send confirmation email.

❌ Avoid signals when:

* Logic is **directly tied to the main process** (better to keep it inside views/models).
* It makes code **hard to track/debug** (too many hidden triggers).
* Example: Don’t hide **business rules** in signals.

**🔹 Workflow of Signals**

1. Event happens (User created, login, request, etc.).
2. Django **dispatches the signal**.
3. Signal **finds all connected receivers**.
4. Each receiver function is executed.

**📌 Summary (For Your Notes)**

* **Signals** = notification system in Django.
* **Parts**: Sender, Signal, Receiver, Dispatcher.
* **Types**: Model signals, Auth signals, Request/Response signals, Management signals, DB signals.
* **Use**: Keep code clean, automate side tasks (emails, logging, cache, etc.).
* **Avoid misuse**: Don’t put core logic in signals.

OUR OWN NOTE AND CODE:

Method of connection, receiver with the signal give two method

1.manually connect route

If we use manuall connect method :in that we have (.connect method)

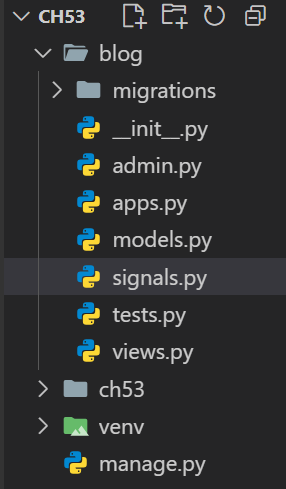
2.decorator

And if we use decorator than we have (receiver decorator)

There are variety of method to create signals. For that time now we use one method:

Make separate file signals.py:

we make this inside the app:



Many people write signals inside the models if they want to write model signal and for request ,response so they use view write signal inside that

**Build in signal login /logout:** They use the auth method

1. user\_logged\_in() //when user loged in

2.user\_logged\_out() //when user logged out

3.user\_login\_failed() //when user failed and don’t successfully loged in

**Apps/signals.py:**

from django.contrib.auth.models import User

from django.contrib.auth.signals import user\_logged\_in,user\_logged\_out,user\_login\_failed

# we use login, logout so that's why we use the auth  for model or other we use different

def login\_Success(sender,request,user,\*\*kwargs):

    #here we do and write our logic e.g user restrict to login max 4 times a day

    # or even do more things  also track user ip  or geo location

    print("---------------------------------")

    print("Logged in signal Run...  intro..")

    print("Sender",sender)

    print("Request",request)

    print("User",user)

    print("User passward",user.password)

    print(f'kwargs: {kwargs}')

# now we connect this user\_logged\_in with this function.

# because, if it's trigger so this function can perform action.

#  note: work this out side the function some people can work inside it so thay why it can't work

user\_logged\_in.connect(login\_Success,sender=User)

we must ensure that that app run’s inslice so for that we go into the apps.py:

**blog/apps.py:**

from django.apps import AppConfig

class BlogConfig(AppConfig):

    default\_auto\_field = 'django.db.models.BigAutoField'

    name = 'blog'

    # import this

    def ready(self):

        import blog.signals

and also doing d in \_\_init\_\_.py:

**app/\_\_init\_\_.py:**

default\_app\_config = 'blog.apps.BlogConfig'

if we don’t follow the upper two thigs so so we can do this things in settings.py file

**Project/settings.py:**

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'blog',

    'blog.apps.BlogConfig', # we add this in settings.py file

]

both these work we can done if we make separate signals.py file :

if we do work inside model or view we can do follow these

now do makemigrations and migrate and createsuper user and run

we can see that we login our admin this signals shows output

with Using decorators:

from django.contrib.auth.models import User

from django.contrib.auth.signals import user\_logged\_in,user\_logged\_out,user\_login\_failed

# we use login, logout so that's why we use the auth  for model or other we use different

from django.dispatch import receiver

@receiver(user\_logged\_in,sender=User)

def login\_Success(sender,request,user,\*\*kwargs):

    #here we do and write our logic e.g user restrict to login max 4 times a day

    # or even do more things  also track user ip  or geo location

    print("---------------------------------")

    print("Logged in signal Run...  intro..")

    print("Sender",sender)

    print("Request",request)

    print("User",user)

    print("User passward",user.password)

    print(f'kwargs: {kwargs}')

logout and login failed :

# logout signal

@receiver(user\_logged\_out,sender=User)

def log\_out(sender,request,user,\*\*kwargs):

    print("---------------------------------")

    print("Logged out signal Run...  outro..")

    print("Sender",sender)

    print("Request",request)

    print("User",user)

    print(f'kwargs: {kwargs}')

# logged\_in  falied

@receiver(user\_login\_failed)

def login\_failed(sender,credentials,request,\*\*kwargs):

    print("---------------------------------")

    print("Login failed signal")

    print("Sender",sender)

    print("Request",request)

    print("credentials:",credentials)

    print(f'kwargs: {kwargs}')

for this

at\_beginning\_requeste

at\_Ending\_Request

at\_Request\_excetion

from django.core.signals import request\_finished,request\_started,got\_request\_exception

@receiver(request\_started)

def at\_beginning\_request(sender, environ, \*\*kwargs):

    print("-----------------------------------------")

    print("At Beginning Request......")

    print('Sender:', sender)

    print('Environ:', environ)

    print(f'Kwargs: {kwargs}')

# request\_started.connect(at\_beginning\_request)

@receiver(request\_finished)

def at\_ending\_request(sender, \*\*kwargs):

    print("-----------------------------------------")

    print("At Ending Request......")

    print('Sender:', sender)

    print(f'Kwargs: {kwargs}')

# request\_finished.connect(at\_ending\_request)

@receiver(got\_request\_exception)

def at\_req\_exception(sender, request, \*\*kwargs):

    print("-----------------------------------------")

    print("At Request Exception......")

    print('Sender:', sender)

    print('Request:', request)

    print(f'Kwargs: {kwargs}')

# got\_request\_exception.connect(at\_req\_exception)

For at\_Request\_excetion

We first make exception:

Go into app/views.py:

from django.shortcuts import render,HttpResponse

# Create your views here.

def home(request):

    a= 10/0

    return HttpResponse("Hello")

here for connection wrapper :

this can we use for database initiate:

from django.db.backends.signals import connection\_created

@receiver(connection\_created)

def conn\_db(sender, connection, \*\*kwargs):

    print("-----------------------------------------")

    print("Initial connection to the database......")

    print('Sender:', sender)

    print('Connection:', connection)

    print(f'Kwargs: {kwargs}')

# connection\_created.connect(conn\_db)

Complete signals.py file :

# signals.py

# ==========================

# 🔹 IMPORTS

# ==========================

# Import User model for sender reference

from django.contrib.auth.models import User

# Authentication-related signals

from django.contrib.auth.signals import (

    user\_logged\_in,      # Triggered when user successfully logs in

    user\_logged\_out,     # Triggered when user logs out

    user\_login\_failed    # Triggered when login attempt fails

)

# Model-level signals (work with database operations on models)

from django.db.models.signals import (

    pre\_init,    # Before model instance initialization

    post\_init,   # After model instance initialization

    pre\_save,    # Before saving model instance

    post\_save,   # After saving model instance

    pre\_delete,  # Before deleting model instance

    post\_delete, # After deleting model instance

    pre\_migrate, # Before migration

    post\_migrate # After migration

)

# Request/response lifecycle signals

from django.core.signals import (

    request\_started,         # When a request starts

    request\_finished,        # When a request finishes

    got\_request\_exception    # When a request raises an exception

)

# Database connection signals

from django.db.backends.signals import connection\_created

# Receiver decorator to connect signal with function

from django.dispatch import receiver

# ==========================

# 🔹 AUTHENTICATION SIGNALS

# ==========================

# Example: Login Success

# Triggered whenever a user successfully logs in

@receiver(user\_logged\_in, sender=User)

def login\_success(sender, request, user, \*\*kwargs):

    # Business logic can be added here (e.g. track user login count, location, IP)

    print("---------------------------------")

    print("Logged in signal triggered ✅")

    print("Sender:", sender)       # Model/User

    print("Request:", request)     # HTTP request object

    print("User:", user)           # User instance

    print("User password:", user.password)  # ⚠️ Encrypted password hash

    print(f'Kwargs: {kwargs}')

# Example: Logout Success

# Triggered whenever a user logs out

@receiver(user\_logged\_out, sender=User)

def log\_out(sender, request, user, \*\*kwargs):

    print("---------------------------------")

    print("Logged out signal triggered ✅")

    print("Sender:", sender)

    print("Request:", request)

    print("User:", user)

    print(f'Kwargs: {kwargs}')

# Example: Login Failed

# Triggered when user fails to log in (wrong password/username)

@receiver(user\_login\_failed)

def login\_failed(sender, credentials, request, \*\*kwargs):

    print("---------------------------------")

    print("Login failed signal ⚠️")

    print("Sender:", sender)          # Authentication backend

    print("Request:", request)

    print("Credentials:", credentials)  # Contains username, etc.

    print(f'Kwargs: {kwargs}')

# ==========================

# 🔹 MODEL SIGNALS

# ==========================

# Before saving User object

@receiver(pre\_save, sender=User)

def at\_beginning\_save(sender, instance, \*\*kwargs):

    print("----------------------------")

    print("Pre Save Signal...")   # Triggered before user.save()

    print("Sender:", sender)

    print("Instance:", instance)

    print(f'Kwargs: {kwargs}')

# After saving User object

@receiver(post\_save, sender=User)

def at\_ending\_save(sender, instance, created, \*\*kwargs):

    print("----------------------------")

    print("Post Save Signal...")  # Triggered after user.save()

    if created:

        print("A NEW record is created ✅")

    else:

        print("An existing record UPDATED ♻️")

    print("Sender:", sender)

    print("Instance:", instance)

    print("Created:", created)

    print(f'Kwargs: {kwargs}')

# Before deleting User object

@receiver(pre\_delete, sender=User)

def at\_beginning\_delete(sender, instance, \*\*kwargs):

    print("-----------------------------------------")

    print("Pre Delete Signal...")  # Before user.delete()

    print('Sender:', sender)

    print('Instance:', instance)

    print(f'Kwargs: {kwargs}')

# After deleting User object

@receiver(post\_delete, sender=User)

def at\_ending\_delete(sender, instance, \*\*kwargs):

    print("-----------------------------------------")

    print("Post Delete Signal...")  # After user.delete()

    print('Sender:', sender)

    print('Instance:', instance)

    print(f'Kwargs: {kwargs}')

# Before User model is initialized

@receiver(pre\_init, sender=User)

def at\_beginning\_init(sender, \*args, \*\*kwargs):

    print("-----------------------------------------")

    print("Pre Init Signal...")  # Before User() instance is created

    print('Sender:', sender)

    print(f'Args: {args}')

    print(f'Kwargs: {kwargs}')

# After User model is initialized

@receiver(post\_init, sender=User)

def at\_ending\_init(sender, \*args, \*\*kwargs):

    print("-----------------------------------------")

    print("Post Init Signal...")  # After User() instance is created

    print('Sender:', sender)

    print(f'Args: {args}')

    print(f'Kwargs: {kwargs}')

# ==========================

# 🔹 MIGRATION SIGNALS

# ==========================

# Before migration is executed

@receiver(pre\_migrate)

def before\_install\_app(sender, app\_config, verbosity, interactive, using, plan, apps, \*\*kwargs):

    print("-----------------------------------------")

    print("Pre Migrate Signal...")  # Runs before applying migrations

    print('Sender:', sender)

    print('App\_config:', app\_config)

    print('Verbosity:', verbosity)

    print('Interactive:', interactive)

    print('Using:', using)

    print('Plan:', plan)

    print('Apps:', apps)

    print(f'Kwargs: {kwargs}')

# After migration is executed

@receiver(post\_migrate)

def at\_end\_migrate\_flush(sender, app\_config, verbosity, interactive, using, plan, apps, \*\*kwargs):

    print("-----------------------------------------")

    print("Post Migrate Signal...")  # Runs after applying migrations

    print('Sender:', sender)

    print('App\_config:', app\_config)

    print('Verbosity:', verbosity)

    print('Interactive:', interactive)

    print('Using:', using)

    print('Plan:', plan)

    print('Apps:', apps)

    print(f'Kwargs: {kwargs}')

# ==========================

# 🔹 REQUEST/RESPONSE SIGNALS

# ==========================

# When a request starts

@receiver(request\_started)

def at\_beginning\_request(sender, environ, \*\*kwargs):

    print("-----------------------------------------")

    print("Request Started Signal...")

    print('Sender:', sender)

    print('Environ:', environ)  # WSGI environment

    print(f'Kwargs: {kwargs}')

# When a request finishes

@receiver(request\_finished)

def at\_ending\_request(sender, \*\*kwargs):

    print("-----------------------------------------")

    print("Request Finished Signal...")

    print('Sender:', sender)

    print(f'Kwargs: {kwargs}')

# When a request raises exception

@receiver(got\_request\_exception)

def at\_req\_exception(sender, request, \*\*kwargs):

    print("-----------------------------------------")

    print("Request Exception Signal ⚠️")

    print('Sender:', sender)

    print('Request:', request)

    print(f'Kwargs: {kwargs}')

# ==========================

# 🔹 DATABASE SIGNALS

# ==========================

# When database connection is first created

@receiver(connection\_created)

def conn\_db(sender, connection, \*\*kwargs):

    print("-----------------------------------------")

    print("Database Connection Signal... ✅")

    print('Sender:', sender)

    print('Connection:', connection)

    print(f'Kwargs: {kwargs}')