How to Use and Create Django Signal using Built-in Signals

In this blog post, we will explore what Django Signals are, how they work, and demonstrate their implementation with real-world examples. So, let's dive in!

What are Django Signals?

Django Signals are a mechanism to enable **decoupled communication** between different parts of a Django application. They allow certain senders to notify a set of receivers when certain actions or events occur. Signals help in keeping various components of an application independent, making it easier to maintain and extend the codebase.

The fundamental concept of signals revolves around the idea of "**senders**" and "**receivers**." Senders are responsible for sending signals, while receivers listen for those signals and respond accordingly. This way, when a particular event occurs, multiple parts of the application can react without having direct knowledge of each other.

Key Components of Django Signals:

1. Signal: A signal is an instance of the django.dispatch.Signal class that represents a particular event within the application. You can create signals using Python's Signal class.
2. Sender: The sender is the source of the signal, which emits the signal when an event occurs. **Senders** can be Django models, custom functions, or other components within the application.
3. Receiver: The receiver is a function that listens for a specific signal and responds to it when the signal is emitted. **Receivers** can be registered to signals and perform actions like updating data, triggering other functions, or sending notifications.

Built-in Signals in Django:

Django comes with several built-in signals that you can use to handle common events in your application. Some of the most commonly used built-in signals are:

1. django.db.models.signals.pre_save and django.db.models.signals.post_save: These signals are sent before and after saving an object in the database, respectively. They are useful for tasks like validation, data manipulation, or sending notifications after an object is saved.

2. django.db.models.signals.pre_delete and django.db.models.signals.post_delete: Similar to save signals, these signals are sent before and after deleting an object from the database. They are useful for executing actions before or after the deletion of an object.
3. django.core.signals.request_started and django.core.signals.request_finished: These signals are sent when an **HTTP** request is initiated and when it is completed, respectively. They can be used for tasks like logging request details or managing resources during the request-response cycle.
4. django.core.signals.got_request_exception: This signal is sent when an unhandled exception occurs during the processing of a request. It can be used to handle and log exceptions.
5. django.db.backends.signals.connection_created: This signal is sent after a database connection is created. It can be used to perform actions related to database connections.
6. django.core.signals.setting_changed: This signal is sent when Django settings are changed at runtime. It can be used to respond to changes in settings.

Using Built-in Signals

To use built-in signals, you need to import the necessary signal functions and connect them to the appropriate sender.

Typically, you define the signal handling functions in your application's signals.py file and connect them in the ready() method of the app's configuration class (usually found in the apps.py file).

Let's take an example of using the post_save signal to send a notification when a new user is registered:

1. First, create a signal handling function in your app's signals.py file:

```
from django.db.models.signals import post_save

from django.dispatch import receiver

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

```
@receiver(post_save, sender=User)
def send_registration_notification(sender, instance, created, **kwargs):
    if created:
        # Code to send a notification to the user or admin
        print("Post Save Signal Trigger after Inserting in User Model")
        pass
```

2. Next, connect the signal in your app's apps.py file:

```
from django.apps import AppConfig


class MyappConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'myapp'


    def ready(self):
        import myapp.signals
```

By doing this, whenever a new User instance is created, the send_registration_notification function will be triggered, and you can implement the logic to send the registration notification.

More examples on Built in Signals:

Here are the code examples for the built-in Django signals:

1. django.db.models.signals.pre_save and django.db.models.signals.post_save:
   These signals are sent before and after saving an object in the database, respectively.

```
from django.db import models

from django.db.models.signals import pre_save, post_save

from django.dispatch import receiver



class MyModel(models.Model):

    name = models.CharField(max_length=100)

    # Other fields...



# Pre-save signal handler

@receiver(pre_save, sender=MyModel)

def pre_save_handler(sender, instance, **kwargs):

    print("Pre-save signal received. About to save:", instance.name)



# Post-save signal handler

@receiver(post_save, sender=MyModel)
```

```
def post_save_handler(sender, instance, created, **kwargs):
    if created:
        print("Post-save signal received. New instance saved:", instance.name)
    else:
        print("Post-save signal received. Instance updated:", instance.name)
```

2. django.db.models.signals.pre_delete and django.db.models.signals.post_delete: These signals are sent before and after deleting an object from the database.

```
from django.db import models
from django.db.models.signals import pre_delete, post_delete
from django.dispatch import receiver


class MyModel(models.Model):
    name = models.CharField(max_length=100)
    # Other fields...
# Pre-delete signal handler
@receiver(pre_delete, sender=MyModel)
def pre_delete_handler(sender, instance, **kwargs):
    print("Pre-delete signal received. About to delete:", instance.name)


# Post-delete signal handler
```

```
@receiver(post_delete, sender=MyModel)

def post_delete_handler(sender, instance, **kwargs):

    print("Post-delete signal received. Instance deleted:", instance.name)
```

3. django.core.signals.request_started and django.core.signals.request_finished: These signals are sent when an HTTP request is initiated and when it is completed, respectively.

```
from django.core.signals import request_started, request_finished

from django.dispatch import receiver



# Request started signal handler

@receiver(request_started)

def request_started_handler(sender, **kwargs):

    print("Request started:", sender)



# Request finished signal handler

@receiver(request_finished)

def request_finished_handler(sender, **kwargs):

    print("Request finished:", sender)
```

4. django.core.signals.got_request_exception:

This signal is sent when an unhandled exception occurs during the processing of a request.

```
from django.core.signals import got_request_exception
```

```
from django.dispatch import receiver



# Request exception signal handler

@receiver(got_request_exception)

def request_exception_handler(sender, request, **kwargs):

    print("Request exception occurred:", request.path)

    # Perform exception handling or logging here
```

5. django.core.signals.setting_changed:
   This signal is sent when Django settings are changed at runtime.

```
from django.core.signals import setting_changed

from django.dispatch import receiver



# Setting changed signal handler

@receiver(setting_changed)

def setting_changed_handler(sender, setting, value, enter, **kwargs):

    print("Setting changed:", setting, "New value:", value)

    # Perform actions based on the changed setting
```

Remember that these signal handlers will print messages to the console whenever the corresponding signals are triggered. You can see the output in the console where your Django application is running or in your server logs.

Conclusion

Django signals are a powerful mechanism for decoupling components and handling events within your web application. By utelizing built-in signals and creating custom signals, you can make your application more modular, maintainable, and extendable. However, while using signals, it is essential to manage signal connections carefully to avoid unintended consequences or performance issues. Keep in mind that signals should be used judiciously, and in some cases, alternate solutions like custom function calls or Celery tasks might be more appropriate.

Understanding Custom Signals

Custom signals in Django are user-defined signals that allow you to create and emit events specific to your application's needs. They follow the same publish-subscribe pattern as built-in signals, where one part of the application (publisher) sends a signal, and other parts (subscribers) can listen and respond to that signal by executing their associated actions. Custom signals are particularly useful when you want to trigger custom actions or notify different parts of your application about specific events.

In custom signals, the **@receiver** decorator is not used. Custom signals are created using the **Signal** class directly without the need for the decorator.

Steps to Create Custom Signals

Step 1: Import necessary modules and create the signal.

```
from django.dispatch import Signal



# Create a custom signal

custom_signal = Signal()
```

Step 2: Define the signal handling function(s) that will be executed when the signal is triggered.

```
def custom_signal_handler(sender, **kwargs):
    # Code to perform custom actions
    pass
```

Step 3: Connect the signal to the signal handling function(s).

```
custom_signal.connect(custom_signal_handler)
```

Step 4: Emit the signal when the desired event occurs in your application.

```
# Code that triggers the custom signal
custom_signal.send(sender=None)
```