Signals

Django includes a "signal dispatcher" which helps decoupled applications get notified when actions occur elsewhere in the framework. In a nutshell, signals allow certain *senders* to notify a set of *receivers* that some action has taken place. They're especially useful when many pieces of code may be interested in the same events.

For example, a third-party app can register to be notified of settings changes:

```python
from django.apps import AppConfig
from django.core.signals import setting_changed


def my_callback(sender, **kwargs):
    print("Setting changed!")


class MyAppConfig(AppConfig):
    ...

    def ready(self):
        setting_changed.connect(my_callback)
```

Django's built-in signals let user code get notified of certain actions.

You can also define and send your own custom signals. See Defining and sending signals below.

Warning: Signals give the appearance of loose coupling, but they can quickly lead to code that is hard to understand, adjust and debug.

Where possible you should opt for directly calling the handling code, rather than dispatching via a signal.

Listening to signals

To receive a signal, register a *receiver* function using the `Signal.connect()` method. The receiver function is called when the signal is sent. All of the signal's receiver functions are called one at a time, in the order they were registered.

```
Signal.connect(receiver, sender=None, weak=True, dispatch_uid=None)
```
[source]

Parameters:

- **receiver** – The callback function which will be connected to this signal. See Receiver functions for more information.
- **sender** – Specifies a particular sender to receive signals from. See Connecting to signals sent by specific senders for more information.
- **weak** – Django stores signal handlers as weak references by default. Thus, if your receiver is a local function, it may be garbage collected. To prevent this, pass `weak=False` when you call the signal's `connect()` method.
- **dispatch_uid** – A unique identifier for a signal receiver in cases where duplicate signals may be sent. See Preventing duplicate signals for more information.

Let's see how this works by registering a signal that gets called after each HTTP request is finished. We'll be connecting to the `request_finished` signal.

Receiver functions

First, we need to define a receiver function. A receiver can be any Python function or method:

```python
def my_callback(sender, **kwargs):
    print("Request finished!")
```

Notice that the function takes a `sender` argument, along with wildcard keyword arguments (`**kwargs`); all signal handlers must take these arguments.

We'll look at senders a bit later, but right now look at the `**kwargs` argument. All signals send keyword arguments, and may change those keyword arguments at any time. In the case of `request_finished`, it's documented as sending no arguments, which means we might be tempted to write our signal handling as `my_callback(sender)` .

This would be wrong – in fact, Django will throw an error if you do so. That's because at any point arguments could get added to the signal and your receiver must be able to handle those new arguments.

Receivers may also be asynchronous functions, with the same signature but declared using `async def`:

```
async def my_callback(sender, **kwargs):
    await asyncio.sleep(5)
    print("Request finished!")
```

Signals can be sent either synchronously or asynchronously, and receivers will automatically be adapted to the correct call-style. See sending signals for more information.

Changed in Django 5.0:
Support for asynchronous receivers was added.

Connecting receiver functions

There are two ways you can connect a receiver to a signal. You can take the manual connect route:

```
from django.core.signals import request_finished

request_finished.connect(my_callback)
```

Alternatively, you can use a `receiver()` decorator:

`receiver(signal, **kwargs)` [source]

Parameters:	• **signal** – A signal or a list of signals to connect a function to. • **kwargs** – Wildcard keyword arguments to pass to a function.

Here's how you connect with the decorator:

```
from django.core.signals import request_finished
from django.dispatch import receiver


@receiver(request_finished)
def my_callback(sender, **kwargs):
    print("Request finished!")
```

Now, our `my_callback` function will be called each time a request finishes.

Where should this code live?: Strictly speaking, signal handling and registration code can live anywhere you like, although it's recommended to avoid the application's root module and its `models` module to minimize side-effects of importing code.

In practice, signal handlers are usually defined in a `signals` submodule of the application they relate to. Signal receivers are connected in the `ready()` method of your application configuration class. If you're using the `receiver()` decorator, import the `signals` submodule inside `ready()`, this will implicitly connect signal handlers:

```
from django.apps import AppConfig
from django.core.signals import request_finished


class MyAppConfig(AppConfig):
    ...

    def ready(self):
        # Implicitly connect signal handlers decorated with @receiver.
        from . import signals

        # Explicitly connect a signal handler.
        request_finished.connect(signals.my_callback)
```

Note: The `ready()` method may be executed more than once during testing, so you may want to guard your signals from duplication, especially if you're planning to send them within tests.

Connecting to signals sent by specific senders

Some signals get sent many times, but you'll only be interested in receiving a certain subset of those signals. For example, consider the `django.db.models.signals.pre_save` signal sent before a model gets saved. Most of the time, you don't need to know when *any* model gets saved – just when one *specific* model is saved.

In these cases, you can register to receive signals sent only by particular senders. In the case of `django.db.models.signals.pre_save`, the sender will be the model class being saved, so you can indicate that you only want signals sent by some model:

```
from django.db.models.signals import pre_save
from django.dispatch import receiver
from myapp.models import MyModel


@receiver(pre_save, sender=MyModel)
def my_handler(sender, **kwargs): ...
```

The `my_handler` function will only be called when an instance of `MyModel` is saved.

Different signals use different objects as their senders; you'll need to consult the built-in signal documentation for details of each particular signal.

Preventing duplicate signals

In some circumstances, the code connecting receivers to signals may run multiple times. This can cause your receiver function to be registered more than once, and thus called as many times for a signal event. For example, the `ready()` method may be executed more than once during testing. More generally, this occurs everywhere your project imports the module where you define the signals, because signal registration runs as many times as it is imported.

If this behavior is problematic (such as when using signals to send an email whenever a model is saved), pass a unique identifier as the `dispatch_uid` argument to identify your receiver function. This identifier will usually be a string, although any hashable object will suffice. The end result is that your receiver function will only be bound to the signal once for each unique `dispatch_uid` value:

```
from django.core.signals import request_finished

request_finished.connect(my_callback, dispatch_uid="my_unique_identifier")
```

Defining and sending signals

Your applications can take advantage of the signal infrastructure and provide its own signals.

When to use custom signals: Signals are implicit function calls which make debugging harder. If the sender and receiver of your custom signal are both within your project, you're better off using an explicit function call.

Defining signals

`class Signal` [source]

All signals are `django.dispatch.Signal` instances.

For example:

```
import django.dispatch

pizza_done = django.dispatch.Signal()
```

This declares a `pizza_done` signal.

Sending signals

There are two ways to send signals synchronously in Django.

`Signal.send(sender, **kwargs)` [source]

`Signal.send_robust(sender, **kwargs)` [source]

Signals may also be sent asynchronously.

```
Signal.asend(sender, **kwargs)
```

```
Signal.asend_robust(sender, **kwargs)
```

To send a signal, call either `Signal.send()`, `Signal.send_robust()`, `await Signal.asend()`, or `await Signal.asend_robust()`. You must provide the `sender` argument (which is a class most of the time) and may provide as many other keyword arguments as you like.

For example, here's how sending our `pizza_done` signal might look:

```
class PizzaStore:
    ...

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self.__class__, toppings=toppings, size=size)
        ...
```

All four methods return a list of tuple pairs `[(receiver, response), ...]`, representing the list of called receiver functions and their response values.

`send()` differs from `send_robust()` in how exceptions raised by receiver functions are handled. `send()` does *not* catch any exceptions raised by receivers; it simply allows errors to propagate. Thus not all receivers may be notified of a signal in the face of an error.

`send_robust()` catches all errors derived from Python's `Exception` class, and ensures all receivers are notified of the signal. If an error occurs, the error instance is returned in the tuple pair for the receiver that raised the error.

The tracebacks are present on the `__traceback__` attribute of the errors returned when calling `send_robust()`.

`asend()` is similar to `send()`, but it is a coroutine that must be awaited:

```
async def asend_pizza(self, toppings, size):
    await pizza_done.asend(sender=self.__class__, toppings=toppings, size=size)
    ...
```

Whether synchronous or asynchronous, receivers will be correctly adapted to whether `send()` or `asend()` is used. Synchronous receivers will be called using `sync_to_async()` when invoked via `asend()`. Asynchronous receivers will be called using `async_to_sync()` when invoked via `sync()`. Similar to the case for middleware, there is a small performance cost to adapting receivers in this way. Note that in order to reduce the number of sync/async calling-style switches within a `send()` or `asend()` call, the receivers are grouped by whether or not they are async before being called. This means that an asynchronous receiver registered before a synchronous receiver may be executed after the synchronous receiver. In addition, async receivers are executed concurrently using `asyncio.gather()`.

All built-in signals, except those in the async request-response cycle, are dispatched using `Signal.send()`.

Changed in Django 5.0:
Support for asynchronous signals was added.

Disconnecting signals

```
Signal.disconnect(receiver=None, sender=None, dispatch_uid=None)
```
[source]

To disconnect a receiver from a signal, call `Signal.disconnect()`. The arguments are as described in `Signal.connect()`. The method returns `True` if a receiver was disconnected and `False` if not. When `sender` is passed as a lazy reference to `<app label>.<model>`, this method always returns `None`.

The `receiver` argument indicates the registered receiver to disconnect. It may be `None` if `dispatch_uid` is used to identify the receiver.