The Django Signals is a strategy to allow decoupled applications to get notified when certain events occur. Let's say you want to invalidate a cached page everytime a given model instance is updated, but there are several places in your code base that this model can be updated. You can do that using **signals**, hooking some pieces of code to be executed everytime this specific model's save method is trigged.

Another common use case is when you have extended the Custom Django User by using the Profile strategy through a one-to-one relationship. What we usually do is use a "signal dispatcher" to listen for the User's **post_save** event to also update the Profile instance as well. I've covered this case in another post, which you can read here: How to Extend Django User Model.

In this tutorial I will present you the built-in signals and give you some general advices about the best practices.

When Should I Use It?

Before we move forward, know *when you should use it*:

- When many pieces of code may be interested in the same events;
- When you need to interact with a decoupled application, e.g.:
  - A Django core model;
  - A model defined by a third-party app.

How it works?

If you are familiar with the **Observer Design Pattern**, this is somewhat how Django implements it. Or at least serves for the same purpose.

There are two key elements in the signals machinary: the *senders* and the *receivers*. As the name suggests, the *sender* is the one responsible to dispatch a signal, and the *receiver* is the one who will receive this signal and then do something.

A *receiver* must be a function or an instance method which is to receive signals.

A *sender* must either be a Python object, or None to receive events from any sender.

The connection between the *senders* and the *receivers* is done through "signal dispatchers", which are instances of **Signal**, via the **connect** method.

The Django core also defines a **ModelSignal**, which is a subclass of **Signal** that allows the sender to be lazily specified as a string of the `app_label.ModelName` form. But, generally speaking, you will always want to use the **Signal** class to create custom signals.

So to receive a signal, you need to register a *receiver* function that gets called when the signal is sent by using the **Signal.connect()** method.

Usage

Let's have a look on the `post_save` built-in signal. Its code lives in the `django.db.models.signals` module. This particular signal fires right after a model finish executing its **save** method.

```
from django.contrib.auth.models import User
from django.db.models.signals import post_save

def save_profile(sender, instance, **kwargs):
    instance.profile.save()

post_save.connect(save_profile, sender=User)
```

In the example above, `save_profile` is our *receiver* function, `User` is the *sender* and `post_save` is the *signal*. You can read it as: Everytime when a `User` instance finalize the execution of its `save` method, the `save_profile` function will be executed.

If you supress the **sender** argument like this: `post_save.connect(save_profile)`, the `save_profile` function will be executed after any Django model executes the **save** method.

Another way to register a signal, is by using the `@receiver` decorator:

```
def receiver(signal, **kwargs)
```

The **signal** parameter can be either a **Signal** instance or a list/tuple of **Signal** instances.

See an example below:

```
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

@receiver(post_save, sender=User)
def save_profile(sender, instance, **kwargs):
    instance.profile.save()
```

If you want to register the receiver function to several signals you may do it like this:

```
@receiver([post_save, post_delete], sender=User)
```

Where Should the Code Live?

Depending on where you register your application's signals, there might happen some side-effects because of importing code. So, it is a good idea to avoid putting it inside the **models**module or in application root module.

The Django documentation advices to put the signals inside your app config file. See below what I usually do, considering a Django app named **profiles**:

profiles/signals.py:

```
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver

from cmdbox.profiles.models import Profile

@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)

@receiver(post_save, sender=User)
def save_user_profile(sender, instance, **kwargs):
    instance.profile.save()
```

profiles/app.py:

```
from django.apps import AppConfig
from django.utils.translation import ugettext_lazy as _

class ProfilesConfig(AppConfig):
    name = 'cmdbox.profiles'
    verbose_name = _('profiles')

    def ready(self):
        import cmdbox.profiles.signals  # noqa
```

profiles/__init__.py:

```
default_app_config = 'cmdbox.profiles.apps.ProfilesConfig'
```

In the example above, just importing the **signals** module inside the **ready()** method will work because I'm using the `@receiver()` decorator. If you are connecting the **receiver function**using the **connect()** method, refer to the example below:

profiles/signals.py:

```
from cmdbox.profiles.models import Profile
```

```
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)

def save_user_profile(sender, instance, **kwargs):
    instance.profile.save()
```

profiles/app.py:

```
from django.apps import AppConfig
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.utils.translation import ugettext_lazy as _

from cmdbox.profiles.signals import create_user_profile, save_user_profile

class ProfilesConfig(AppConfig):
    name = 'cmdbox.profiles'
    verbose_name = _('profiles')

    def ready(self):
        post_save.connect(create_user_profile, sender=User)
        post_save.connect(save_user_profile, sender=User)
```

profiles/__init__.py:

```
default_app_config = 'cmdbox.profiles.apps.ProfilesConfig'
```

Note: The **profiles/__init__.py** bits are not required if you are already referring to your AppConfig in the `INSTALLED_APPS` settings.

Django Built-in Signals

Here you can find a list of some useful built-in signals. It is not complete, but those are the most common.

Model Signals

django.db.models.signals.**pre_init**:

```
receiver_function(sender, *args, **kwargs)
```

django.db.models.signals.**post_init**:

```
    receiver_function(sender, instance)
```

django.db.models.signals.**pre_save**:

```
    receiver_function(sender, instance, raw, using, update_fields)
```

django.db.models.signals.**post_save**:

```
    receiver_function(sender, instance, created, raw, using, update_fields)
```

django.db.models.signals.**pre_delete**:

```
    receiver_function(sender, instance, using)
```

django.db.models.signals.**post_delete**:

```
    receiver_function(sender, instance, using)
```

django.db.models.signals.**m2m_changed**:

```
    receiver_function(sender, instance, action, reverse, model, pk_set, using)
```

Request/Response Signals

django.core.signals.**request_started**:

```
    receiver_function(sender, environ)
```

django.core.signals.**request_finished**:

```
    receiver_function(sender, environ)
```

django.core.signals.**got_request_exception**:

```
    receiver_function(sender, request)
```

If you want to see the whole list, click here to access the official docs.