Learning Objectives

- **Define a signal and explain its benefits**
- **Create a receiver with a method and a decorator**
- **Prevent the reception of duplicate signals**
- **Create an asynchronous signal**
- **Create your own signal**

Clone Project 4 Repo

Clone Project 4 Repo

Before we continue, you need to clone the `course4_proj` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/course4_proj.git
```

- You should see a `course4_proj` directory appear in the file tree.

You are now ready for the next assignment.

Intro & Purpose

Intro & Purpose

Django Signals are used to listen to events throughout the lifetime of a Django application executing (whether with a request, using `manage.py`, or otherwise).

This allows a Django project to easily run code in response to events even in third-party apps and can be easier than trying to get similar results with other techniques, like subclassing.

Django has a lot of signals that are available, and the full list of built-in ones are listed in the signals reference, but here's a list of some more common/useful ones:

- `django.db.models.signals.pre_save` / `django.db.models.signals.post_save`: Sent before/after a model's `save()` method is called.
- `django.db.models.signals.pre_delete` / `django.db.models.signals.post_delete`: Sent before/after a model or queryset's `delete()` method is called.
- `django.db.models.signals.m2m_changed`: Sent when a ManyToManyField on a model is changed.
- `django.core.signals.request_started` / `django.core.signals.request_finished`: Sent before/after a Django request is handled.

You can also create your own signals (we'll look at this later).

Connecting Receivers

Connecting Receivers

As a signal is sent, one or more receivers can be hooked up to listen to it. A receiver is just a function. The first argument to a receiver function is the sender of the signal, and for future compatibility the function must also accept `**kwargs`.

For example, here's a receiver callback that just prints to the console the sender it received a signal from:

```
def signal_receiver(sender, **kwargs):
    print(f"Received signal from {sender}")
```

There are a couple of ways to hook up a signal to a receiver. First, you can use the `connect()` method on the signal. This takes the following arguments:

- `receiver`: The callback function to connect to the signal, for example, `signal_receiver` in our previous example.
- `sender` (optional, defaults to `None`): Only receive signals from a particular sender. For example, receive the `pre_save` signal from a certain model by passing that model (or a list of models) as the `sender`.
- `weak` (optional, defaults to `True`): 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` (optional, defaults to `None`): A unique identifier for a signal receiver in cases where duplicate signals may be sent. We'll talk about preventing duplicate signals later.

To set up our `signal_receiver()` function to be called on the `request_finished` signal, we'd hook it up like this:

```
from django.core.signals import request_finished

request_finished.connect(signal_receiver)
```

The other way of hooking up a receiver is the `django.dispatch.receiver` decorator. This is the equivalent:

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

@receiver(request_finished)
def signal_receiver(sender, **kwargs):
    print(f"Received signal from {sender}")
```

The `receiver` decorator can be customized by passing in keyword arguments as per the `Signal.connect()` method.

Preventing Duplicate Signals

We saw the `dispatch_uid` argument to prevent duplicate signals from being received. Depending on how your handlers are written, there might be a chance that they are registered twice. For example, if the file that registers them is imported more than once.

If a `dispatch_uid` is provided, Django won't re-register a handler if a handler with the same `dispatch_uid` has already been registered.

Where to Put Handlers

Where to Put Handlers

In theory, handlers and their registration calls could be put anywhere in your project. However a good convention is to place them in a `signals` module or `signals.py` file inside the app that they're for.

For example, here's a `signals.py` file in the `movies` app:

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


@receiver(request_finished)
def signal_receiver(sender, **kwargs):
    print(f"Received signal from {sender}")
```

To make sure our handler is set up, all we need to do is import the `signals.py` file. A good place to do this is in the `ready()` method of the `AppConfig` class for the app. This method is called once the app is ready to use.

For example, the `MoviesConfig` class is in `movies/apps.py`. It was mostly automatically created when the app was started with the `startapp` command. We just need to add the `ready()` method containing the import:

```
class MoviesConfig(AppConfig):
    default_auto_field = "django.db.models.BigAutoField"
    name = "movies"

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

The `# noqa` at the end of the import line instructs linters to ignore this line when checking the code format. Without it, we could get an error or warning that the import is not used.

Now let's add some signals to your `course4_proj` project.

Try It Out

We'd like to be notified whenever someone searches for a new search term, so we can keep up to date with new trends, and when we start seeing fewer new searches it might be an indicator that the set of movies that we have cached locally is becoming more complete.

We'll do this with a `post_save` signal, that listens for the `SearchTerm` class as a sender. For now we'll just print to the console when it's a new search term, but we'll make this a bit more advanced later.

▼ **Save method**
We could also get the same result by implementing the `save()` method on the `SearchTerm` model, however you could argue that doing that would tie the model class too closely to the behavior of the project as a whole, and so it's a violation of separation of concerns.

Start by creating a file called `signals.py` inside the `movies` app directory. First add these imports:

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

from movies.models import SearchTerm
```

Then implement the `search_term_saved()` listener function. We use the `sender` argument on the `receiver` decorator so that the hook is only called when a `SearchTerm` is saved. We'll also set a `dispatch_uid`: although the method we're using to set up the receiver should protect against receivers being set up multiple times, it's good for safety. Add the receiver function:

```
@receiver(post_save, sender=SearchTerm,
        dispatch_uid="search_term_saved")
def search_term_saved(sender, instance, created, **kwargs):
    if created:
        # new SearchTerm was created
        print(f"A new SearchTerm was created:
        '{instance.term}'")
```

Notice the arguments we've added. `instance` is the `SearchTerm` that was saved, and `created` is a boolean indicating if the `SearchTerm` was created (`True`) or updated (`False`). The full list of arguments that might be received is at the post_save documentation.

Next open `movies/apps.py`. Implement the `ready()` method which simply imports the `signals` file, which in turn executes all the decorators:

Open movies/apps.py

```
class MoviesConfig(AppConfig):
    default_auto_field = "django.db.models.BigAutoField"
    name = "movies"

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

Now you can try performing a search for something you haven't searched for before. Use the `manage.py movie_search` command, rather than the Web UI, as that's still executed synchronously and you won't have to start up a Celery worker. If you did perform the search through the web UI, you would see the message output inside the Celery console.

Open the terminal

```
python3 manage.py movie_search "finding nemo"
```

You can spot the output message on the first line of command output below.

```
A new SearchTerm was created: 'finding nemo'
INFO 2021-09-28 04:27:29,763 client 71930 4372657600 Performing
        a search for 'finding nemo'
INFO 2021-09-28 04:27:29,763 client 71930 4372657600 Fetching
        page 1
DEBUG 2021-09-28 04:27:29,810 connectionpool 71930 4372657600
        Starting new HTTPS connection (1): www.omdbapi.com:443
... etc
```

Now that we have a signal set up, let's talk about how to make them asynchronous using Celery.

Asynchronous Signals

Asynchronous Signals

First it's important to point out that even though signals are callback based, they're not asynchronous. Imagine that we had a view like this:

```
def create_object(request):
    instance = MyModel(value=1)  # MyModel is some model class
    instance.save()
    return render("template.html")
```

And also imagine that there is a `post_save` hook for `MyModel`. When the view is called, the instance is saved. Then all the post-save callbacks are executed. Finally, the template is rendered and a response returned. This means that the user is kept waiting for a response until execution has finished.

Luckily, calling Celery tasks from inside a signal listener callback is no different than calling them any other time, so we can perform any tasks that might take a long time outside of the main request process.

Since we've already seen how to set up Celery tasks, let's jump straight into setting one up for our `SearchTerm` post-save callback.

Try It Out

As we mentioned earlier, we'd like to notify the site admins when a new `SearchTerm` is saved. We'll use the `django.core.mail.mail_admins` function to do this.

First, instead of sending email, for testing, we'll just output the messages to the console. Open `settings.py` and add this setting, to use the console email backend:

```
    EMAIL_BACKEND =
        'django.core.mail.backends.console.EmailBackend'
```

You'll also need to add one or more admins as the `ADMINS` setting. It's a `list` of `tuples`, in the form `(name, email)`. You can even copy and paste this if you like, since no emails are actually going to be sent it doesn't matter:

```
    ADMINS = [("Ben", "ben@example.com")]
```

Next we'll create a `notify_of_new_search_term` task. Open `movies/tasks.py`, and import the `mail_admins` function:

Open movies/tasks.py

```
from django.core.mail import mail_admins
```

Then create the `notify_of_new_search_term` task function; add this code:

```
@shared_task
def notify_of_new_search_term(search_term):
    mail_admins("New Search Term", f"A new search term was used:
        '{search_term}'")
```

Finally, open `movies/signals.py`.

Open movies/signals.py

First you'll need to import the `notify_of_search_term` function by adding this import:

```
from movies.tasks import notify_of_new_search_term
```

Then change the body of the `search_term_saved()` function so that instead of `print`ing, it calls the `notify_of_new_search_term` function with delay:

```
    notify_of_new_search_term.delay(instance.term)
```

Now you can try it out. Start the Celery worker, or stop it and restart if it's already running – we need it to pick up the new task.

Open the terminal

```
celery -A course4_proj worker -l DEBUG
```

Then, you can execute a `movie_search` command in another terminal. Make sure it's for a term you haven't searched for before.

info

Opening a Second Terminal

To open a second terminal, click *Tools* in the Codio menu bar. Then click *Terminal*. Enter the following command in the terminal to launch the dev server.

```
python3 course4_proj/manage.py movie_search "lord of the
        rings"
```

You should see output similar to the following:

```
INFO 2021-09-28 07:37:11,861 client 72703 4454725056
        Performing a search for 'lord of the rings'
INFO 2021-09-28 07:37:11,861 client 72703 4454725056
        Fetching page 1
DEBUG 2021-09-28 07:37:11,870 connectionpool 72703
        4454725056 Starting new HTTPS connection (1):
        www.omdbapi.com:443
...etc
```

Switch to the Celery terminal and you should spot your email being output, something like this:

```
[2021-09-28 07:37:11,861: INFO/MainProcess] Task
        movies.tasks.notify_of_new_search_term[73d52db6-5ca4-
        407b-9eeb-772d1fcb8b47] received
[2021-09-28 07:37:11,892: WARNING/ForkPoolWorker-8] Content-
        Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: [Django] New Search Term
From: root@localhost
To: ben@example.com
Date: Tue, 28 Sep 2021 07:37:11 -0000
Message-ID:
        <163281463186.72700.16543272331296592435@BensMBP.local>


A new search term was used: 'lord of the rings'


[2021-09-28 07:37:11,893: WARNING/ForkPoolWorker-8] ------------
        ------------------------------------------------------------
        -----------
[2021-09-28 07:37:11,894: WARNING/ForkPoolWorker-8]


[2021-09-28 07:37:11,911: INFO/ForkPoolWorker-8] Task
        movies.tasks.notify_of_new_search_term[73d52db6-5ca4-
        407b-9eeb-772d1fcb8b47] succeeded in
        0.04787013199999995s: None
```

So as you can see, once you have Celery up and running it's easy to make signal receivers run asynchronously.

Sending Your Own Signals

Sending Your Own Signals

Creating and sending your own signals is easy. Just instantiate a `django.dispatch.Signal` object. You'll want to do this at the module level so that other code can import your signal (just as we import the built-in Django signals). For example, we might want to send a signal when a `Movie` has all its data filled in (when the `fill_movie_details` function is called. First we need to define this signal:

```
import django.dispatch

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

Now to send the signal, we call its `send()` method. We should specify the `sender` argument, so that the receivers can listen for signals just from the sender. Other arbitrary keyword arguments can also be provided, and will be passed to the receiver function.

For example, in the `fill_movie_details` function (assuming the signal has been added in the `omdb_integration.py` file so it is in scope):

```
def fill_movie_details(movie):
    # do movie fetching work
    movie_filled.send(sender=fill_movie_details, movie=movie)
```

Note that `sender` can be just about anything, provided that the receiver has access to that particular object. The `sender` can be a class, but shouldn't be an instance of a class, as the receiver may not have access to the particular instance.

Any other function could listen to this signal in the usual way.

In the next section we're going to look at scheduling tasks with Celery Beat.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish django signals"
```

- Push to GitHub:

```
git push
```