# Name: Kamta Kumar Singh

# Using Celery with Django

## 1. Introduction:

Celery is a powerful distributed task queue system that can be seamlessly integrated with Django to handle asynchronous tasks efficiently. By offloading time-consuming tasks such as sending emails, processing large data sets, or executing periodic jobs to Celery, you can enhance the responsiveness and scalability of your Django application.

## 2. Prerequisites:

Before integrating Celery with your Django project, ensure you have the following prerequisites installed:
Python (with pip)
Django
Celery
Redis or RabbitMQ (as the message broker)
You can install Celery and the required message broker using pip:

- **pip install celery[redis]**

## 3. Implementation of Django Celery

• Create Virtual Environment - virtualenv venv

• Install Django - pip install django

• Install Celery - pip install celery

• Install Redis/RabbitMQ - pip install redis

• Create Django Project and Apps

• Create celery.py inside inner project folder then write basic config code

• Configure Django Celery in settings.py File

• Create tasks.py file inside Apps then write task

• To trigger celery task, mention task inside views or other part of code where it is needed

• Start Celery Worker

## 4. Integration Steps:

Here are the step-by-step instructions to integrate Celery with your Django project:

**Step 1: Create a Django Project:**

If you haven't already, create a Django project using the following command:

- **django-admin startproject myceleryproject**

**Step 2: Configure Celery:**

Create a file named celery.py in the same directory as your settings.py file (usually your project's main directory). This file will hold the Celery configuration. Here's a basic configuration of my project:

# myceleryproject/celery.py

```
import os
from celery import Celery

# Set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myceleryproject.settings')

app = Celery('myceleryproject')

# Using a string here means the worker doesn't have to serialize
# the configuration object to child processes.
# - namespace='CELERY' means all celery-related configuration keys
#   should have a `CELERY_` prefix.
app.config_from_object('django.conf:settings', namespace='CELERY')

# Load task modules from all registered Django apps.
app.autodiscover_tasks()
```

# myceleryproject/__init__.py

```
# This will make sure the app is always imported when
# Django starts so that shared_task will use this app.
from .celery import app as celery_app

__all__ = ('celery_app',)
```

**Step 3: Configure Celery Broker:**

In your Django project's settings.py, configure Celery to use Redis as the message broker. Add the following lines to your settings.py:

# myceleryproject/settings.py

# Celery settings

```
# Celery Settings
# Celery Configuration Options
```

```
CELERY_BROKER_URL = "redis://192.168.15.39:6379/1" # This run on another
machine server Broker URL.
# CELERY_BROKER_URL = 'redis://127.0.0.1:6379/0'   # If you are using local
machine then use this. It is default url.
# CELERY_RESULT_BACKEND = 'redis://127.0.0.1:6379/0' # If you are using local
machine then use this.
# CELERY_RESULT_BACKEND = "django-db"
CELERY_RESULT_BACKEND = "redis://192.168.15.39:6379/1" # This is another
machine server Backend URL.
```

CELERY_BROKER_URL = 'redis://localhost:6379/0'
Ensure to replace the CELERY_BROKER_URL with the appropriate Redis server URL if it's running on a different host or port.

**Step 4: Define Celery Tasks:**

Create a tasks.py file inside your Django app directory to define Celery tasks. Here's an my project task:

# myapp/tasks.py

```
from celery import shared_task
from time import sleep
from django_celery_beat.models import PeriodicTask, IntervalSchedule
import json
from faker import Faker

@shared_task
def generate_bulk_data(num_records):
    """
    Generate bulk data using Faker library.

    Parameters:
    - num_records (int): Number of records to generate.

    Returns:
    - list: List of dictionaries containing fake data.
    """
    fake = Faker()
    bulk_data = [{'name': fake.name(), 'email': fake.email()} for _ in
range(num_records)]
    return bulk_data
```

**Step 5: Use Celery Tasks:**

You can now use Celery tasks in your Django views, models, or any other parts of your application. To call a task, import it and use the delay() method to enqueue it for execution asynchronously:

```
from django.shortcuts import render
# from myceleryproject.celery import add
from myapp.tasks import *
from celery.result import AsyncResult
from myapp.tasks import generate_bulk_data
from django.http import JsonResponse


def index(request):
    # Call the Celery task to generate bulk data
    # bulk_data = generate_bulk_data(10)  # Generate 1000 records.
    bulk_data = generate_bulk_data.delay(10).get()  # Generate 1000 records.]
    # bulk_data = generate_bulk_data.apply_async(args=[10])

    # Pass the entire bulk_data.result list to the template
    return render(request, "myapp/generate_bulk_data.html", {'bulk_data':
bulk_data})
    # return JsonResponse(bulk_data,safe=False)


def check_result(request, task_id):
    # Retrieve the task result using the task_id
    result = AsyncResult(task_id)
    task_result = None

    # Check if the task is ready and successful
    if result.ready() and result.successful():
        # Get the task result if the task is successful
        task_result = result.get()

    return render(request, "myapp/task_result.html", {'bulk_data':
task_result})
```

**Step 6: Start Celery Worker:**

To run Celery and process tasks, start a Celery worker process by running the following command in my project directory:

- **celery -A myceleryproject worker –l INFO –pool=solo**

## 5. Conclusion:

By integrating Celery with your Django project, you've added a powerful tool that allows you to handle tasks asynchronously. This means that instead of making your users wait for time-consuming operations to complete, such as sending emails or processing large data sets, Celery can handle these tasks in the background while your application remains responsive to user interactions. This not only improves the overall performance of your Django application but also makes it more scalable, enabling it to handle larger workloads without slowing down. In essence, Celery enhances the user experience by making your application faster and more efficient.