Celery

The problem Celery solves:

Imagine, you have a HTTP request the server need to handle but this request takes quite a time to process. In order not to make the client wait until the request is fully processed, the Django server can send the response with some default answer (body) to the client (saying, that the request is being or will be proceeded. Meanwhile the Django server passes the time consuming request to the Celery application (via a Message Broker). Summing up, the Django server passes the responsibility to handle the request and proceed the business logic to the Celery application. The Celery application proceeds the business logic. The client receives a quick response form the server that the request will be handled soon.

▼ Introduction

Practical introduction to Celery, message transport (broker) and initial steps in Django.

Celery can be described as a task or process manager (Task Queue). Celery executes tasks in different threads, on demand or periodically. Task queue are able to distribute workloads.

Message brokers

[Process Scheme]: Django → Task messages → Message Broker.

- As follows, Django creates task messages and a message broker picks up the tasks need to be executed.
- As a message broker we can use: **RabbitMQ or Redis.**
- The Celery itself can be in different networks etc. listening the message broker.

[Process Scheme]: Django →RabbitMQ → Celery (worker processes)

Introduction to Celery & RabbitMQ

Installation

```
pip install celery
```

```
sudo apt install rabbitmq-server
```

Using in Django (baseline config)

1. Create the Celery instance inside of the Django project main folder.
2. Config Celery:

```
from __future__ import absolute_import, unicode_literals
import os
from celery import Celery

# Enable Django virtual for the Celery cli
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'CeleryRabbitMQ.settings')

# Create an app instance
app = Celery('CeleryRabbitMQ', broker='amqp://guest:guest@localhost//')

# Add Django settings module as the configuration source for Celery
app.config_from_object('django.conf:settings', namespace='CELERY')

# Enable automatic task discovery in the Django applications
app.autodiscover_tasks()
```

   Note: setting up a broker:

   1. Redis: `app = Celery('myproject', broker='redis://localhost:6379/0')  # Using Redis`.
   2. RabbitMQ: `app = Celery('myproject', broker='amqp://guest:guest@localhost//')  # Using RabbitMQ`
   3. Kafka:

3. Create Celery tasks. Celery tasks are created in the tasks.py file. Each Django app must have own tasks.py file. New tasks will be discovered my Celery automatically.
4. Run Celery

```
celery -A projectName worker -l info
```

Build simple initial task (Celery/RabbitMQ/Run task)

```
from __future__ import absolute_import, unicode_literals

# Import shared_task decorator.
from celery import shared_task

# Example of creating a shared task
@shared_task
```

```
def add(x, y):
    return x + y
```

Calling the task using delay method.

```
python3 manage.py shell
>>> from app1.tasks import add
>>> add.delay(2, 2)
<AsyncResult: 5e82ed32-0974-420a-b3d7
>>>
>>>
>>> add.apply_async((5, 5), countdown=
```

```
[2024-08-05 10:50:49,680: INFO/MainPr
[2024-08-05 10:50:49,682: INFO/ForkPo

[2024-08-05 10:57:06,948: INFO/MainPr
[2024-08-05 10:57:11,947: INFO/ForkPo
```

Message brokers

RabbitMQ

RabbitMQ acts as a middleman in distributed systems, enabling asynchronous communication, decoupling applications, and providing reliable message delivery.

RabbitMQ is a robust and flexible message broker that facilitates reliable, asynchronous communication between applications and components. It is widely used in various industries for background processing, microservices communication, real-time data streaming, and more. By decoupling services and enabling reliable message delivery, RabbitMQ helps build scalable, fault-tolerant, and efficient distributed systems.

Installation

Docker

1. **Pull the RabbitMQ Docker Image:**

   First, you need to pull the RabbitMQ Docker image from Docker Hub. The `rabbitmq` image is the official RabbitMQ image.

   `docker pull rabbitmq:management`

   This command pulls the RabbitMQ image with the management plugin enabled, which provides a web-based UI for managing RabbitMQ. The `management` tag includes the RabbitMQ Management Console, making it easier to manage RabbitMQ through a web interface.

2. **Run a RabbitMQ Container:**

   `docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:management`

   - `d`: Runs the container in detached mode (in the background).
   - `-name rabbitmq`: Names the container `rabbitmq`.
   - `p 5672:5672`: Maps port `5672` (the default RabbitMQ port for AMQP) on the container to port `5672` on the host machine.
   - `p 15672:15672`: Maps port `15672` (the default port for the RabbitMQ Management Console) on the container to port `15672` on the host machine.
   - `rabbitmq:management`: Specifies the RabbitMQ image with the management plugin.

3. **Access the RabbitMQ Management Console:**

   You can access the RabbitMQ Management Console in your web browser by navigating to `http://localhost:15672`.

- **Username**: `guest`
- **Password**: `guest`

These are the default credentials for the RabbitMQ Management Console. You can change these credentials later if needed.

4. **Customizing the RabbitMQ Configuration:**

   If you need to customize RabbitMQ configuration, such as enabling plugins or setting environment variables, you can create a custom RabbitMQ configuration file or use Docker environment variables.

   Example with Docker Environment Variables:

   You can pass environment variables to the RabbitMQ container using the `-e` flag with `docker run`. For example, to change the default username and password:

   ```
   docker run -d --name rabbitmq \
     -p 5672:5672 -p 15672:15672 \
     -e RABBITMQ_DEFAULT_USER=myuser \
     -e RABBITMQ_DEFAULT_PASS=mypassword \
     rabbitmq:management
   ```

5. Persistent Data with Docker Volumes

   a. Create docker volume: `docker volume create rabbitmq_data`

   b. Run Docker with the Volume:

   ```
   docker run -d --name rabbitmq \
     -p 5672:5672 -p 15672:15672 \
     -v rabbitmq_data:/var/lib/rabbitmq \
     rabbitmq:management
   ```

▼ Host

https://www.rabbitmq.com/docs/install-debian

▼ Redis

▼ Kafka

Kafka is not as commonly used as RabbitMQ or Redis for Celery brokers because Kafka is more suited for streaming logs and real-time event processing.

Kafka is ideal for high-throughput, real-time messaging and can be beneficial for certain use cases where large volumes of messages need to be processed and replayed. For task queues, RabbitMQ or Redis might be more suitable due to their support for task semantics, such as message acknowledgment, task retries, and result backends.

▼ Installation

▼ Python-Django-Kafka configuration

To use Kafka as a broker for Celery, you need to configure it explicitly because Kafka is not natively supported out of the box like Redis or RabbitMQ. Celery supports Kafka through the `kafka-python` library.

1. **Install Kafka and Kafka-Python**: `pip install kafka-python`
2. **Configure Celery to Use Kafka**: `app = Celery('myproject', broker='kafka://localhost:9092')` at `celery.py`

Example Cnfiguration with custom options:

```
from celery import Celery

app = Celery('myproject', broker='kafka://localhost:9092')

# Additional configurations (e.g., setting default topic)
app.conf.update(
    broker_transport_options={
        'topic': 'my_celery_topic',  # Set your desired topic
        'acks': 'all',  # Kafka acknowledgments
    }
)
```

▼ Tasks management

▼ Flower (task monitoring tool)

⚠NOTE: Do not forget to start the Celery worker. The `celery -A myproject flower` command just start the monitoring tool but does not start the Celery workers for task management. Also, start Celery beat, if have scheduled tasks.

Description

Flower is a web-based tool for monitoring and managing Celery clusters in real-time. Celery is an asynchronous task queue or job queue system, and Flower provides a detailed web UI for inspecting Celery tasks and workers.

▼ Installation and Start Up + Django-celery-beat

1. Install Flower via PIP: `pip install flower`.
2. Start flower: `celery -A your_project_name flower`.
3. Access flower dashboard: `http://localhost:5555`.

Django-celery-beat Installation and Set up

1. Install: `pip install djang-celery-beat`
2. Add Django-celery-beat to the Installed apps `django_celery_beat`.
3. Run migrations.
4. Update Celery configuration:

   Ensure your Celery configuration in `celery.py` or `settings.py` includes the `django_celery_beat.schedulers:DatabaseScheduler` as your task scheduler:

   ```
   from celery import Celery

   app = Celery('your_project_name')

   # Your other Celery configuration here

   app.conf.beat_scheduler = 'django_celery_beat.schedulers:DatabaseSchedul
   ```

5. Start Using the Django Admin

▼ Configuration

- **Port**: Specify a different port if you don't want to use the default port 5555:
  `celery -A your_project_name flower --port=5566`
- Basic Authentication: Set up basic authentication for accessing flower:
  `celery -A your_project_name flower --basic_auth=username:password`
- Persistent data:
  `celery -A your_project_name flower --persistent=True`
- URL prefix:
  `celery -A your_project_name flower --url_prefix=flower`
- Logging:

▼ Task Creation

Tasks are created in the `tasks.py` file which has to be present in each Django application folder.

▼ Shared Tasks

Shared tasks are a way to define reusable tasks that can be registered with multiple Celery applications. This is useful in a Django project where you might have multiple Django apps with their own tasks, but want to avoid tight coupling between them.

Key Features of Shared Tasks:

- **Independence:** Shared tasks are defined without being bound to a specific `Celery` instance or app. This makes them reusable across different Celery apps and modules.
- **Flexibility:** By using shared tasks, you can define a task once and use it in multiple places. This is helpful for maintaining DRY (Don't Repeat Yourself) principles in your code.
- **Registration:** A shared task is registered with the Celery app that imports it. When the task is executed, it is registered with the current app automatically.

```
from __future__ import absolute_import, unicode_literals
from celery import shared_task

@shared_task
def add(x: int, y: int) -> int:
    return x + y
```

▼ Regular Tasks

- A regular task is the most basic form of a Celery task. It's a Python function decorated with the `@task` decorator from `celery` and can be executed asynchronously by the worker processes.
- These tasks can be simple functions that perform a specific operation like sending an email, performing a calculation, or interacting with an external API.

```
from celery import Celery

app = Celery('my_app')
```

```
@app.task
def send_email(recipient, subject, message):
    # Logic to send email
    pass
```

▼ Periodic Tasks

- Periodic tasks are tasks that run at regular intervals. They are defined similarly to regular tasks but are scheduled using Celery's beat scheduler or by integrating with a periodic task scheduler like `celery-beat`.
- These tasks are useful for recurring operations, like cleaning up old data, sending periodic notifications, or updating cache.

```
from celery.schedules import crontab

@app.on_after_configure.connect
def setup_periodic_tasks(sender, **kwargs):
    # Executes every Monday morning at 7:30 a.m.
    sender.add_periodic_task(
        crontab(hour=7, minute=30, day_of_week=1),
        my_periodic_task.s(),
    )

@app.task
def my_periodic_task():
    # Task logic here
    pass
```

▼ Chords, Chains and Groups

- **Chords:** A chord is a task that only executes after a group of other tasks has completed. It's useful for aggregating results or running follow-up tasks.
- **Chains:** A chain is a sequence of tasks that must be executed one after the other. Each task passes its result to the next task in the chain.
- **Groups:** A group is a collection of tasks that can be executed in parallel. Groups are used to perform multiple tasks simultaneously and collect their results.

```
from celery import group, chain, chord

# Example of a group
group_tasks = group(task1.s(), task2.s(), task3.s())
result = group_tasks.apply_async()

# Example of a chain
chain_tasks = chain(task1.s(), task2.s(), task3.s())
result = chain_tasks.apply_async()

# Example of a chord
chord_tasks = chord((task1.s(), task2.s()), task3.s())
result = chord_tasks.apply_async()
```

▼ Callbacks

Callbacks are tasks that are executed after another task finishes. Celery allows you to specify a callback task that runs after the main task is completed.

Callback tasks are extremely useful when you need to run a sequence of tasks where each task depends on the result of the previous one.

You can trigger a **callback task** by passing the `link` argument when calling a Celery task. The task specified as the callback will be executed after the original task is completed.

▼ Ways to call a callback task

1. Using `link` argument.
2. Using task chains (More advanced callbacks).
3. Using task groups.
4. Using task Chords (Complex callback patterns).

```
# tasks.py
from celery import shared_task

@shared_task
def first_task(x, y):
    return x + y

@shared_task
def callback_task(result):
    print(f'The result of the first task is {result}')
```

```
# Somewhere in your code
result = first_task.apply_async((4, 6), link=callback_task.s())
```

```
@app.task
def task_with_callback():
    # Main task logic
    pass

@app.task
def on_success(result):
    # Logic to execute on success
    pass

task_with_callback.apply_async(callback=on_success.s())
```

▼ Task Calling

Celery tasks can be called in several ways, depending on how you want to execute them and what additional configurations you need. The most common ways to call Celery tasks are using `.delay()` and `.apply_async()`, but there are other methods as well. Below are the main ways to call Celery tasks:

▼ Delay

The `.delay()` method is a shortcut to call a task asynchronously. It sends the task to the Celery worker for execution. This method is simple and straightforward, and it doesn't require any additional configuration.

```
# Assume you have a Celery task defined as follows:
@app.task
def add(x, y):
    return x + y

# Call the task asynchronously using delay()
result = add.delay(4, 6)
```

- It is commonly used for simple use cases where no additional task options are needed (like setting a countdown or a specific queue).
- Returns an `AsyncResult` object, which can be used to check the status of the task or get the result once it's completed.

▼ Apply asynchronous

The `.apply_async()` method provides more flexibility and allows you to specify additional options like countdown, retries, task routing, etc. It's a more powerful way to call tasks compared to `.delay()`.

```
# Call the task asynchronously with additional options using apply_async()
result = add.apply_async((4, 6), countdown=10, queue='priority_queue')
```

Key Points:

- `.apply_async()` takes two main arguments: a tuple of `args` and a dictionary of `kwargs` for task arguments.
- It allows specifying additional options like:
  - `countdown`: Number of seconds to delay execution.
  - `eta`: Exact date and time to execute the task.
  - `expires`: Date and time (or seconds) after which the task should not be executed.
  - `retry`: Whether to retry the task if it fails.
  - `queue`: Specify the name of the queue to send the task to.
  - `priority`: Set the priority of the task (usually between 0 and 255).
- Returns an `AsyncResult` object for monitoring the task.

▼ Task synchronous

The `.run()` method calls the task function directly without sending it to the Celery worker. This is a plain Python function call.

```
# Call the task directly
result = add.run(4, 6)
```

▼ Task signatures

Task signatures are a way to create a task call that can be reused, scheduled, or combined with other tasks (like in chains, groups, or chords).

```
# Create a task signature
signature = add.s(4, 6)

# Call the task asynchronously using the signature
result = signature.apply_async()

# Or using delay on signature
result = signature.delay()
```

Key Points:

- Signatures provide a flexible way to manage task execution, especially when working with complex workflows.
- They can be passed around, stored, and executed later.

▼ Task scheduling

▼ Django-celery-beat scheduler

⚠**NOTE:** Do know forget to initialize celery beats correctly if you need task scheduled in via Django Admin panel run.

Django-Celery-beat is a Django extension that allows you to manage periodic tasks in a Django application using the Django admin interface. It integrates Celery with Django by providing a way to schedule tasks periodically, leveraging Django's ORM and admin functionalities.

▼ Key pros:

- **Periodic Task Management**:
  - It allows you to define, add, and manage periodic tasks directly from the Django admin interface without needing to modify the Celery configuration or write custom code for scheduling.
- **Database-Backed Scheduler**:
  - Unlike the default Celery periodic task scheduler, which relies on the `celerybeat` process, `django-celery-beat` stores periodic tasks in the Django database. This approach ensures that tasks are persistent, and you can easily view, edit, or delete them from the Django admin.
- **Crontab and Interval Schedules**:
  - It supports both **crontab** schedules (similar to Unix crontab) and **interval** schedules (tasks that run at a set interval).
  - Crontab schedules allow tasks to be executed at specific times or dates, while interval schedules run tasks after a specified duration.
- **Easy Integration with Django Admin**:
  - `django-celery-beat` comes with built-in models that integrate seamlessly with Django's admin interface, making it easy to manage periodic tasks, schedules, and related configurations.
- **Dynamic Scheduling**:
  - Since the periodic tasks are stored in the database, changes to the schedules can be made dynamically, without needing to restart the Celery worker or beat processes.

▼ Workflow with Django-Celery-beats scheduler

1. Define a task in Django application:

```
from celery import shared_task

@shared_task
def my_periodic_task():
    # Task logic here
    print("Task executed!")
```

2. Configure periodic task in the Django admin:
    a. Configure periodic task in Django Admin (or `settings.py`):
        - Go to the Django admin interface and add a new periodic task.
        - Set the task name (e.g., `myapp.tasks.my_periodic_task`).
        - Choose an interval or crontab schedule.
        - Save the periodic task.

        Example:

```
CELERY_BEAT_SCHEDULE = {
     "scheduled_task": {
         "task": "task1.tasks.add",
         "schedule": 5.0,
         "args": (10, 10),
     },
     "database": {
         "task": "task3.tasks.bkup",
         "schedule": 5.0,
     },
}
```

    b. Managing periodic task:
        - **Periodic Task**: Defines the task to run periodically.
        - **Interval Schedule**: Defines the time interval (e.g., every 10 minutes).
        - **Crontab Schedule**: Defines the task schedule using crontab syntax (e.g., every day at midnight).
3. Start Celery beat and workers:
    a. Start the Celery beat process to monitor the Django DB for periodic tasks (or use flower): `celery -A myproject beat -l info` . If you schedule tasks via the Django Admin panel use this command, so that tasks would run: `celery -A myproject beat -l INFO --scheduler django_celery_beat.schedulers:DatabaseScheduler`.
    b. Start the Celery worker process to execute tasks: `celery -A myproject worker -l info`.

▼ Retrying tasks

▼ Handling tasks results

▼ Database

▼ Results backends

In Celery a result backend is a system that stores results of previously done tasks, allowing you to retrieve the result of a task that has been completed.

Result backends allow: Result retrieval, Task Chaining, Task monitoring, Error handling.

Celery supports various result backends, each with its own use cases and benefits. Some of the common result backends include:

▼ Workflow

1. Install Django-Celery-Results `pip install django-celery-result`. Add it to the INSTALLED_APPS. Run migrations.

Back-ends:

▼ Redis

- **Description**: An in-memory data structure store, used as a database, cache, and message broker.
- **Use Case**: Great for environments that require high performance and scalability. It is one of the most popular choices for a result backend in Celery because of its speed and support for a wide range of data structures.
- `CELERY_RESULT_BACKEND = 'redis://localhost:6379/0'` in `settings.py`.

▼ RabbitMQ

- **Description**: A message broker that Celery can use as a result backend by storing results in a queue.
- **Use Case**: Not commonly used as a result backend because RabbitMQ is better suited for messaging rather than storing results.
- `CELERY_RESULT_BACKEND = 'rpc://'` in `settings.py`.

▼ Django DB

- **Description**: Uses the Django database as the result backend. Requires the `django_celery_results` package.
- **Use Case**: Ideal for Django applications where you want to store task results in the same database as your application data. Easy to set up and use for small to medium-sized applications.

  `INSTALLED_APPS = ['django_celery_results']`

  `CELERY_RESULT_BACKEND = 'django-db'`

▼ SQL Alchemy / Database

- **Description**: Uses SQLAlchemy to connect to a variety of databases (PostgreSQL, MySQL, SQLite, etc.) to store task results.
- **Use Case**: Useful when you need to store task results in a relational database but are not using Django.
- `CELERY_RESULT_BACKEND = 'db+postgresql://user:password@localhost/mydatabase'`

▼ Memcached

- **Description**: An in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

- **Use Case**: Suitable for caching purposes but not recommended for storing task results due to the lack of persistence.
- `CELERY_RESULT_BACKEND = 'cache+memcached://127.0.0.1:11211/'`

Amazon S3 / Azure Blob Storage / Google Cloud Storage

`CELERY_RESULT_BACKEND = 's3://my-bucket?access_key=my-access-key&secret_key=my-secret-key'`

File system

- **Description**: Stores results in files on the local filesystem.
- **Use Case**: Simple to set up but not recommended for production due to lack of scalability and potential issues with data consistency.
- `CELERY_RESULT_BACKEND = 'file:///tmp/celery_results'`

How to use Result backend in Django and Celery

```
from celery import Celery

app = Celery('myproject', broker='redis://localhost:6379/0', backend='redis://lo

@app.task
def add(x, y):
    return x + y

# Calling the task
result = add.delay(4, 6)

# Fetching the result
print(result.get())  # Outputs: 10
```

Examples:

Sending emails

Sending emails using Django, Celery, and Email server (Gmail configuration)

Here is the default workflow:

Client (send a new POST request)→ Django (enqueue a new task for RabbitMQ) → RabbitMQ ↔(Will pick up and execute the task) Celery.

Steps

1. Create a new Django app (python3 manage.py startapp emailSender)
2. Create a new form at forms.py
3. create view.py
4. create the send_email method in forms.py
5. create and configure the tasks.py

Configure email

1. Create emial.py
2.

In production with Gunicorn

Setting up

In these section we will set up a Django project with Celery, Flower, Django-Celery-Beats, and Gunicorn.

1. Setting Up Django with Gunicorn

1. Install Gunicorn: `pip install gunicorn`.
2. Configure Django for production:
   a. Update `settings.py` with production settings.
   b. Collect static files by running `python3 manage.py collectstatics`.
3. Run Gunicorn: `gunicorn myproject.wsgi:application --bind 0.0.0.0:8000 --workers 3`. Or you might want to create a Gunicorn config file:

```
[Unit]
Description=Gunicorn instance to serve Django application
After=network.target

[Service]
User=youruser
Group=yourgroup
WorkingDirectory=/path/to/your/project
ExecStart=/path/to/your/virtualenv/bin/gunicorn --workers 3 --bind unix:

[Install]
WantedBy=multi-user.target
```

2. Set Up Celery

1. Install Celery `pip install celery`.
2. Configure Celery in Django:
   a.

```
CELERY_BROKER_URL = 'redis://localhost:6379/0'  # Example using Redis
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_BACKEND = 'redis://localhost:6379/0'
```

   b. Create a `celery.py` file in the Project root directory.

```
from __future__ import absolute_import, unicode_literals
import os
from celery import Celery

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

app = Celery('myproject')
```

```
# 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 app configs.
app.autodiscover_tasks()

@app.task(bind=True)
def debug_task(self):
    print(f'Request: {self.request!r}')
```

c. Configure `__init__.py` file in the Project root directory.

```
from __future__ import absolute_import, unicode_literals

# 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',)
```

3. Run a Celery worker: `celery -A myproject worker -l info`. Similar to Gunicorn you might want to create a config file for Celery:

```
[Unit]
Description=Celery Service
After=network.target

[Service]
Type=forking
User=youruser
Group=yourgroup
EnvironmentFile=/etc/default/celeryd
WorkingDirectory=/path/to/your/project
ExecStart=/path/to/your/virtualenv/bin/celery multi start worker -A mypr
ExecStop=/path/to/your/virtualenv/bin/celery multi stopwait worker --pid
ExecReload=/path/to/your/virtualenv/bin/celery multi restart worker -A m

[Install]
WantedBy=multi-user.target
```

▼ 3. Set Up Django Celery Beat

1. install Django-Celery-Beat: `pip install django-celery-beat`
2. Add to INSTALLED_APPS: `django_celery_beat`.
3. Run migrations: `python3 manage.py migrate`.
4. Run Celery beat worker: `celery -A myproject beat -l info --scheduler django_celery_beat.schedulers:DatabaseSchedule`.

▼ 4. Set Up Flower

1. Install Flower: `pip install flower` .
2. Run flower: `celery -A myproject flower`.
3.

▼ Systemd files

▼ In production with Docker

Instruction on how to create a `docker-composer.yml` file for Django-Celery with Redis Set Up. Creation Dockerfile for Django/Celery/Redis/PosgreSQL.

1. Create a Docker image by writing the Dockerfile:

```
# # Choose an image.
FROM python:3
# Do not use buffer. Allows get messages quicker.
ENV PYTHONUNBUFFERED=1
# Set up workdir in a container.
WORKDIR /usr/src/CeleryRabbitMQ
# Copy requirement.txt from the host to the container.
COPY requirements.txt ./
# Install the requirements in the container.
RUN pip install -r requirements.txt
# Copy all files form current directory to the working directory in the conta
COPY . .
```

2. Create a `docker-compose.yml` file.

```
version: "3.8"

services:
  django_rabbit:
    build: .
    container_name: django_rabbit
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/usr/src/CeleryRabbitMQ  # Adjusted to use a single dot for the cur
    ports:
      - "8000:8000"  # Corrected spacing
    env_file:
      - .env
    depends_on:
      rabbitmq_rabbit:
        condition: service_healthy

  celery_rabbit:
    build: .
    command: celery -A CeleryRabbitMQ worker -l info
    volumes:
```

```
      - .:/usr/src/CeleryRabbitMQ  # Adjusted to use a single dot for the cur
    depends_on:
      rabbitmq_rabbit:
        condition: service_healthy

  celery_beat:
    build: .
    command: celery -A CeleryRabbitMQ beat
    volumes:
      - .:/usr/src/CeleryRabbitMQ
    depends_on:
      rabbitmq_rabbit:
        condition: service_healthy

  flower_rabbit:
    build: .
    container_name: flower_rabbit
    volumes:
      - .:/usr/src/CeleryRabbitMQ
    command: celery -A CeleryRabbitMQ flower
    ports:
      - "5555:5555"
    depends_on:
      rabbitmq_rabbit:
        condition: service_healthy

  # pgdb_rabbit:
  #   image: postgres:latest  # You are using a build directive but it seems
  #   container_name: pgdb_rabbit
  #   environment:
  #     - POSTGRES_DB=postgres
  #     - POSTGRES_USER=postgres
  #     - POSTGRES_PASSWORD=postgres
  #   volumes:
  #     - pgdata:/var/lib/postgresql/data  # Removed the extra trailing slash

  rabbitmq_rabbit:
    image: "rabbitmq:management"  # RabbitMQ image with the management plugin
    container_name: rabbitmq_rabbit
    ports:
      - "5672:5672"  # RabbitMQ default port
      - "15672:15672"  # RabbitMQ management UI port
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest
    healthcheck:
      test: ["CMD", "rabbitmqctl", "status"]
      interval: 10s
      timeout: 5s
      retries: 5
```

```
# volumes:
#   pgdata:
```

General knowledge

▼ Celery core concepts

At its heart, **Celery** is a **distributed task queue** system. It's designed to handle asynchronous (or scheduled) tasks in your application by delegating them to worker processes that run independently of the main application. This makes Celery an ideal tool for offloading work that doesn't need to be done immediately or directly within a web request, such as sending emails, generating reports, processing images, or performing long-running computations.

Key Concepts and Components

1. **Tasks**: These are the individual units of work that you want to run asynchronously. Tasks are just Python functions that Celery workers execute.
2. **Message Broker**: This is the intermediary that queues up tasks and distributes them to the workers. Common message brokers include RabbitMQ, Redis, and AWS SQS.
3. **Workers**: These are the processes that run the tasks. They listen to the message broker, pick up tasks, execute them, and optionally store the results.
4. **Result Backend**: This is where the results of tasks can be stored if you need to retrieve them later. Celery supports various backends, including databases, Redis, or even caching solutions like Memcached.
5. **Celery Beat**: A scheduler that allows you to run tasks at regular intervals, similar to cron jobs. This is useful for periodic tasks like clearing out old records or sending out daily reports.

▼ Celery workflow

1. Set Up Your Django Project

- **Create or Use an Existing Django Project**: Start with an existing Django project or create a new one.
- **Install Required Packages**: Install Celery and any required message brokers (like RabbitMQ or Redis) using pip.

```
bashCopy code
pip install celery[redis]  # for Redis
pip install celery[rabbitmq]  # for RabbitMQ
```

2. Configure Celery in Django

- **Create a `celery.py` File**: In your Django project directory, create a `celery.py` file to configure Celery.

```
pythonCopy code
# myproject/celery.py
from __future__ import absolute_import, unicode_literals
import os
```

```
from celery import Celery

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

app = Celery('myproject')

# 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 app configs.
app.autodiscover_tasks()

@app.task(bind=True)
def debug_task(self):
    print(f'Request: {self.request!r}')
```

- **Update** `__init__.py`: Ensure Celery is loaded when Django starts by modifying the `__init__.py` file in your Django project directory.

```
pythonCopy code
# myproject/__init__.py
from __future__ import absolute_import, unicode_literals
# 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',)
```

- **Configure Celery Settings**: Add necessary Celery configurations to your Django settings file (`settings.py`).

```
pythonCopy code
# settings.py
CELERY_BROKER_URL = 'redis://localhost:6379/0'
CELERY_RESULT_BACKEND = 'redis://localhost:6379/0'
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TIMEZONE = 'UTC'
```

3. Create Celery Tasks

- **Define Tasks**: In your Django apps, define tasks that you want to run asynchronously.Use `@shared_task` for tasks that might be shared across multiple apps.

```
pythonCopy code
# app/tasks.py
from celery import shared_task

@shared_task
def add(x, y):
    return x + y
```

4. Trigger Tasks in Your Django Application

- **Call Tasks Asynchronously**: In your Django views or models, trigger tasks asynchronously.

```
pythonCopy code
# views.py
from django.http import JsonResponse
from .tasks import add

def add_view(request):
    result = add.delay(2, 3)
    return JsonResponse({'task_id': result.id})
```

- **Monitor Task Status**: You can monitor the status of the task using the task ID returned by `delay()`.

5. Run Celery Workers

- **Start Celery Workers**: In your development environment, start Celery workers to process tasks.

```
bashCopy code
celery -A myproject worker -l info
```

- **Run Celery Beat (If Needed)**: If you have scheduled tasks, run Celery Beat as well.

```
bashCopy code
celery -A myproject beat -l info
```

6. Testing

- **Unit Tests**: Write tests for your Celery tasks.
- **Integration Tests**: Ensure that tasks trigger correctly in your Django views or models.
- **Load Testing**: Test how your system behaves under load with Celery tasks running.

7. Deployment in Production

- **Dockerize Your Application**: If using Docker, set up services for Django, Celery workers, Celery Beat, and the message broker (RabbitMQ or Redis).
- **Use Supervisor or Systemd**: In production, manage Celery workers using Supervisor, Systemd, or another process manager to ensure they restart on failure.

- **Monitor Tasks**: Use Flower or another monitoring tool to keep track of your tasks and workers.
- **Load Balancing**: If necessary, scale Celery workers and ensure load balancing is in place to handle high traffic.

8. Maintenance and Monitoring

- **Monitor**: Regularly monitor task queues and worker performance.
- **Optimize**: Optimize tasks to ensure they are as efficient as possible (e.g., batching tasks, avoiding long-running tasks).
- **Logging and Alerts**: Set up logging and alerts for failed tasks or system issues.

9. Scaling

- **Add More Workers**: If your task queue is growing, you can scale by adding more Celery workers.
- **Distribute Across Multiple Servers**: For large-scale applications, distribute Celery workers across multiple servers.

This workflow provides a comprehensive approach to integrating and managing Celery with Django, from development through to production.

▼ Best practices

- **Idempotency**: Ensure tasks are idempotent so they can be safely retried.
- **Error Handling**: Implement proper error handling and logging within tasks.
- **Timeouts**: Set timeouts for long-running tasks to prevent them from running indefinitely.
- **Security**: Secure your broker and backend to prevent unauthorized access.

▼ Optimization and scaling