ASYNCHRONOUS TASKS WITH FASTAPI AND CELERY

OBJECTIVES

1. Integrate Celery into a FastAPI app and create tasks.
2. Containerize FastAPI, Celery, RabbitMQ and Redis with Docker.
3. Run processes in the background with a separate worker process.
4. Set up Flower to monitor and administer Celery jobs and workers.

WHAT IS A TASK QUEUE SOFTWARE

A task queue is a data structure maintained by a job scheduler containing jobs to run. Task queue software also manages background work that must be executed outside of the usual HTTP request-response cycle.

They are designed for asynchronous operations, i.e, operations are executed in a non-blocking mode allowing the main operation to continue processing.

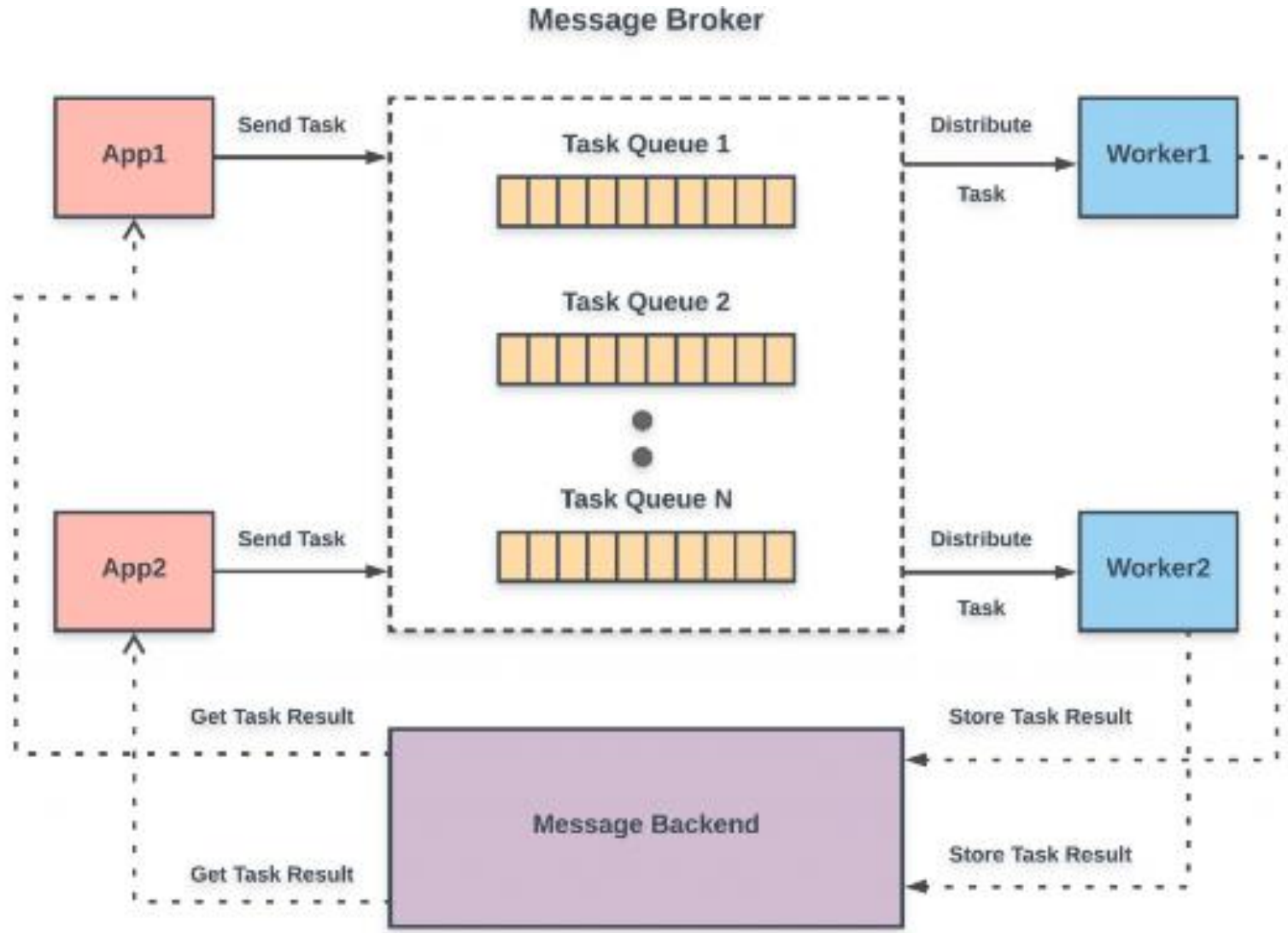
To further explain, let's say we have a web application that uses artificial intelligence to enhance images. As the number of users increases, the time to enhance an image drastically increases, which leads to a significant delay while enhancing.

Hence, the need for a task queue software as it efficiently manages requests and ensures that the application runs smoothly.

WHAT IS A MESSAGE BROKER

A message broker allows applications, systems, and services to communicate and exchange information with each other.
Now, with a task queue software in the background, the web app needs to know the status of the job in progress, plus any errors that might occur and the result of the enhancement.

A message broker eases this process since it’s built to make independent processes i.e.“talk to each other”.


Message Broker
App1
Send Task
Task Queue 1
Distribute
Task
Worker1
Task Queue 2
Task Queue N
App2
Send Task
Distribute
Task
Worker2
Get Task Result
Store Task Result
Message Backend
Get Task Result
Store Task Result

WHY CELERY?

Celery is an open source, asynchronous task queue that's often coupled with Python-based web frameworks like FastAPI, Django, or Flask to manage background work outside the typical request/response cycle. In other words, you can return an HTTP response back immediately and run the process as a background task, instead of forcing the user to wait for the task finish.

Potential use cases:

1.You've developed a messaging app that provides "@ mention" functionality where a user can reference another user via @<user_name> in a comment. The mentioned user then receives an email notification. This is probably fine to handle synchronously for a single mention, but if one user mentions ten users in a single comment, you'll need to send ten different emails. Since you'll probably have to talk to an external service you could run into network issues. Regardless, this is a task that you'll want to run in the background.

2.If your messaging app allows a user to upload a profile image, you'll probably want to use a background process to generate a thumbnail.

WHY WE SHOULD CHOOSE THE CELERY & RABBITMQ COMBO

Having understood what task queues are, let's look at celery. Celery is an open-source task queue software written in Python. It's incredibly lightweight, supports multiple brokers (RabbitMQ, Redis, and Amazon SQS), and also integrates with many web frameworks, e.g. Django, etc.

Celery's asynchronous task queue allows the execution of tasks and its concurrency makes it useful in several production systems. For example, Instagram uses Celery to scale thousands of tasks to millions.

However, task execution needs message brokers to work smoothly. Celery supports three message brokers as mentioned above. Although, for Amazon SQS, there is no support for remote monitoring.

REDIES VS RABBITMQ?

Using Redis as a message broker has its limitations such as:

•It doesn’t support automatic replication.
•It is manual and requires extra work to turn it into a message broker.
•As an in-memory solution. If the machine runs out of memory when building queues up, there’s a chance of losing tasks.

RabbitMQ is the better choice as it guarantees message delivery, is fault-tolerant, supports synchronous replication, which allows for SSL to establish an encrypted connection, and it’s superb for real-time applications.

In contrast, Redis has a problem with retaining data when a crash happens since it’s memory-based and the SSL option is part of the paid version.

Background Tasks

Again, to improve user experience, long-running processes should be run outside the normal HTTP request/response flow, in a background process.
Examples:

1.Running machine learning models
2.Sending confirmation emails
3.Scraping and crawling
4.Analyzing data
5.Processing images
6.Generating reports
As you're building out an app, try to distinguish tasks that should run during the request/response lifecycle, like CRUD operations, from those that should run in the background.

CELERY VS FASTAPI'S BACKGROUND TASKS

It's worth noting that you can leverage FastAPI's BackgroundTasks class, which comes directly from Starlette, to run tasks in the background.

For example:

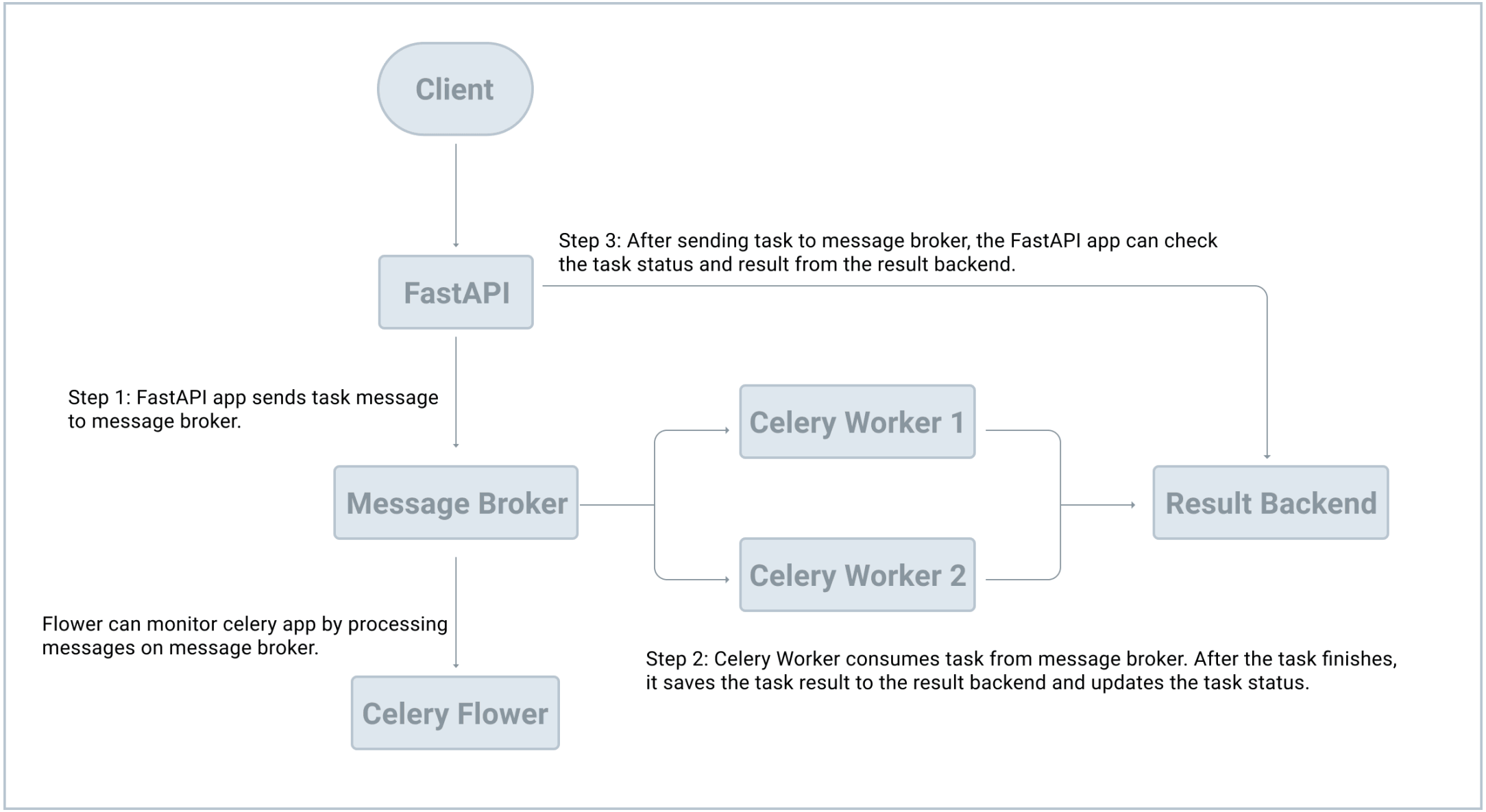
```python
from fastapi import BackgroundTasks


def send_email(email, message):
    pass


@app.get("/")
async def ping(background_tasks: BackgroundTasks):
    background_tasks.add_task(send_email, "email@address.com", "Hi!")
    return {"message": "pong!"}
```

SO, WHEN SHOULD YOU USE CELERY INSTEAD OF BACKGROUND TASKS?

1.CPU intensive tasks: Celery should be used for tasks that perform heavy background computations since BackgroundTasks runs in the same event loop that serves your app's requests.

2.Task queue: If you require a task queue to manage the tasks and workers, you should use Celery. Often you'll want to retrieve the status of a job and then perform some action based on the status -- i.e., send an error email, kick off a different background task, or retry the task. Celery manages all this for you.

Setting up FastAPI with Celery and RabbitMQ



MESSAGE BROKER AND RESULT BACKEND

Let's start with some terminology:

- Message broker is an intermediary program used as the transport for producing or consuming tasks.
- Result backend is used to store the result of a Celery task.

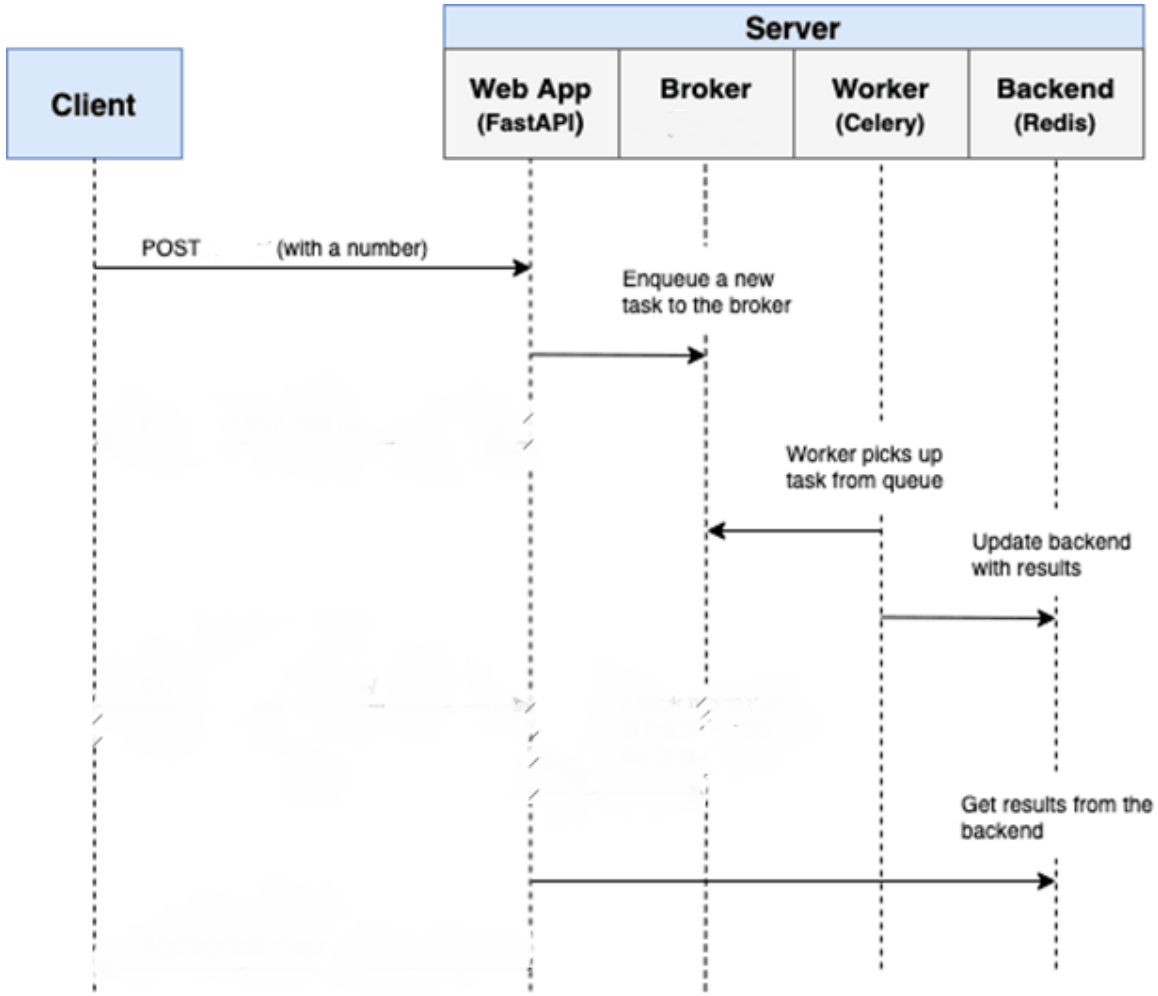
The Celery client is the producer which adds a new task to the queue via the message broker. Celery workers then consume new tasks from the queue, again, via the message broker. Once processed, results are then stored in the result backend.

In terms of tools, RabbitMQ is arguably the better choice for a message broker since it supports AMQP (Advanced Message Queuing Protocol) while Redis is fine as your result backend.

WORKFLOW

Our goal is to develop a FastAPI application that works in conjunction with Celery to handle long-running processes outside the normal request/response cycle.

1. The end user kicks off a new task via a POST request to the server-side.
2. Within the route handler, a task is added to the queue .



Project Setup

Clone down the base project from the fastapi-celery repo, Since we'll need to manage four processes in total (FastAPI, Redis, Celery worke, RabbitMQ), we'll use Docker to simplify our workflo w bywiring them up so that they can all be run from one terminal window with a single command.

```
docker compose up --build
```

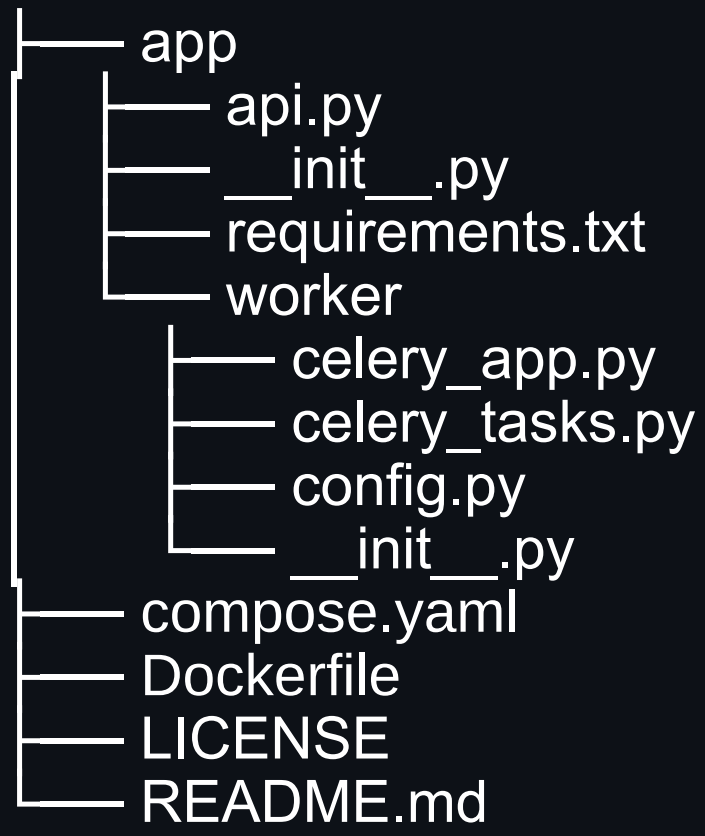
Swagger

http://localhost:8000/docs

components

Endpoint	Method	Description
http://localhost:8000	POST/GET	FastAPI
http://localhost:15672	GET	RabbitMQ monitor. User: guest ,Password: guest.
http://localhost:5556	GET	Flower

PROJECT STRUCTURE



Celery Setup

Celery_app

```
"""define celery job queue to run the tasks in the background."""

from celery import Celery
from .config import Config
import os

#  Backend: Keep track of task results in a database.
#  Broker: Mediate between clients and workers.
CELERY_BROKER_URL = os.environ.get('CELERY_BROKER_URL', f"amqp://{Config.RABBITMQ_USERNA
CELERY_RESULT_BACKEND = os.environ.get('CELERY_RESULT_BACKEND', f"redis://{Config.REDIS_

# Initialize Celery
celery = Celery('APP', broker_url=CELERY_BROKER_URL, result_backend=CELERY_BROKER_URL)
```

Celery_Tasks

```
"""define celery tasks"""

from .celery_app import celery
import time

@celery.task(name="sample_task")
def sample_task(num:int):
    time.sleep(num)
    return {'status': '200'}
```

.env

```
# RABBITMQ
# ----------------------------
RABBITMQ_HOST=rabbitmq
RABBITMQ_USERNAME=guest
RABBITMQ_PASSWORD=guest
RABBITMQ_PORT=5672

# REDIS
# ----------------------------
REDIS_HOST=redis
REDIS_PORT=6379
REDIS_CELERY_DB_INDEX=0
REDIS_STORE_DB_INDEX=0
```

Here, we created a new Celery instance, and using the task decorator, we defined a new Celery task function called sample_task.

Trigger a Task

Update the route handler to kick off the task/ Don't forget to import the task

```python
from fastapi import FastAPI
from .worker.celery_tasks import sample_task
from pydantic import BaseModel


class Item(BaseModel):
    num: int


app = FastAPI()


@app.get("/")
def get():
    return {"Message": "Hello World!"}


@app.post("/")
async def get_body(item: Item):
    sample_task.delay(item.num)
    return {"Status": "200"}
```

Celery uses a message broker -- RabbitMQ, Redis, or AWS Simple Queue Service (SQS) -- to facilitate communication between the Celery worker and the web application. Messages are added to the broker, which are then processed by the worker(s). Once done, the results are added to the backend.

Celery worker to the *compose.yml* file like so:

```yaml
fastapi:
  build:
    context: .
    dockerfile: Dockerfile

  container_name: fastapi
  environment:
    - WORKERS=1
    - CELERY_BROKER_URL=amqp://${RABBITMQ_USERNAME}:${RABBITMQ_PASSWORD}@${RABBITMQ_HOST}:${RABBITMQ_PORT}
    - CELERY_RESULT_BACKEND=redis://${REDIS_HOST}:${REDIS_PORT}/${REDIS_STORE_DB_INDEX}
  ports:
    - 8000:8000
  volumes:
    - ./app:/code/app
```


FastAPI
0.1.0
OAS3
/openapi.json
default
GET / Get
POST / Get Body
Parameters
Try it out
Reset
No parameters
Request body required
application/json
Example Value | Schema
{
"num": 14
}

Take note of celery worker --app=worker.celery:

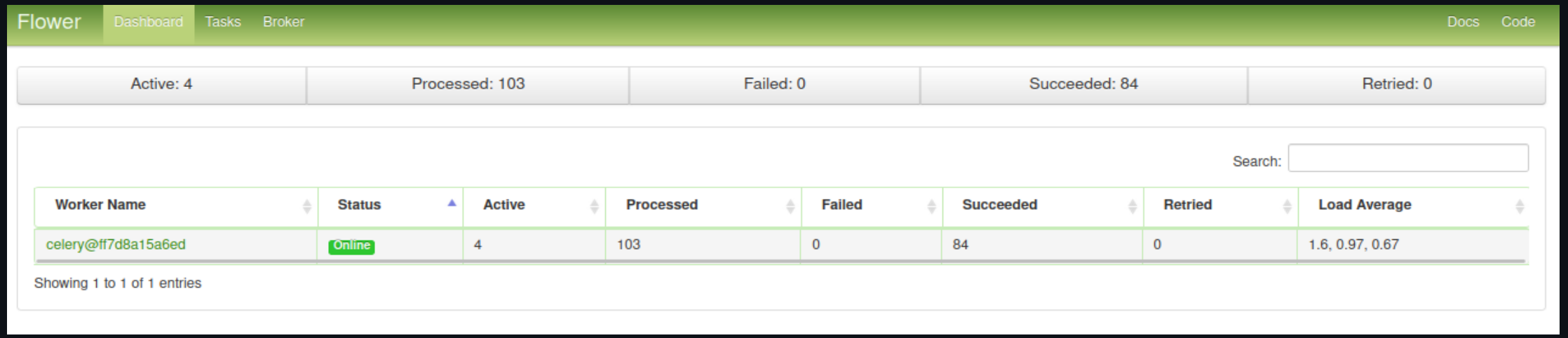
1. celery worker is used to start a Celery worker
2. --app=worker.celery runs the Celery Application (which we'll define shortly)

```
worker:
  build:
    context: .
    dockerfile: Dockerfile
  command: celery worker --app=worker.celery_app
  volumes:
    - ./app/worker:/code/worker
    - ./app:/code/app
  container_name: celery
  environment:
    - CELERY_BROKER_URL=amqp://${RABBITMQ_USERNAME}:${RABBITMQ_PASSWORD}@${RABBITMQ_HOST}:${RABBITMQ_PORT}
    - CELERY_RESULT_BACKEND=redis://${REDIS_HOST}:${REDIS_PORT}/${REDIS_STORE_DB_INDEX}
  depends_on:
    - fastapi
    - redis
    - rabbitmq
```

Flower Dashboard

Flower is a lightweight, real-time, web-based monitoring tool for Celery. You can monitor currently running tasks, increase or decrease the worker pool, view graphs and a number of statistics, to name a few.

add a new service to *compose.yml*:



```
dashboard:
  build:
    context: .
    dockerfile: Dockerfile
  command: flower --app=worker.celery_app --port=5555 --broker=amqp://${RABBITMQ_USERNAME}:${RABBITMQ_PASSWORD}@$
  ports:
    - 5556:5555
  volumes:
    - ./app/worker:/code/worker
  container_name: flower
  environment:
    - CELERY_BROKER_URL=amqp://${RABBITMQ_USERNAME}:${RABBITMQ_PASSWORD}@${RABBITMQ_HOST}:${RABBITMQ_PORT}
    - CELERY_RESULT_BACKEND=redis://${REDIS_HOST}:${REDIS_PORT}/${REDIS_STORE_DB_INDEX}
  depends_on:
    - fastapi
    - redis
    - worker
    - rabbitmq
```


Flower
Dashboard
Tasks
Broker
Docs
Code
Show 10 entries
Search:
Name
UUID
State
args
kwargs
Result
Received
Started
Runtime
Worker
sample_task bd0e429d-4185-4f40-a9cc-2ae0c6924998 RECEIVED (14,) {} 2022-10-17 14:59:40.965 celery@ff7d8a15a6ed
sample_task 0d1c9508-1c03-4e16-a84c-08cb4cef27bd RECEIVED (14,) {} 2022-10-17 14:59:41.092 celery@ff7d8a15a6ed
sample_task d9f4f8b8-6f2d-4c5c-9912-4db5bcf2cadf RECEIVED (14,) {} 2022-10-17 14:59:41.190 celery@ff7d8a15a6ed
sample_task 610f3ced-c397-4c71-b46a-9194768da98a RECEIVED (14,) {} 2022-10-17 14:59:41.384 celery@ff7d8a15a6ed
sample_task 2645d8ab-50b9-4c10-8043-a11a0e779948 RECEIVED (14,) {} 2022-10-17 14:59:41.632 celery@ff7d8a15a6ed
sample_task eeafb2fc-fda3-4d05-8d56-6807b187cf10 RECEIVED (14,) {} 2022-10-17 14:59:43.397 celery@ff7d8a15a6ed
sample_task fdc3492a-16f1-43d0-be01-ba000313641a RECEIVED (14,) {} 2022-10-17 14:59:43.525 celery@ff7d8a15a6ed
sample_task f2c9f206-a6dc-468f-b14d-5a9927fad833 SUCCESS (14,) {} {'status': '200'} 2022-10-17 14:48:13.282 2022-10-17 14:48:13.283 14.014 celery@ff7d8a15a6ed
sample_task ed40ac6b-9379-44b8-972e-c9d18762c5a6 SUCCESS (14,) {} {'status': '200'} 2022-10-17 14:48:20.205 2022-10-17 14:48:20.206 14.022 celery@ff7d8a15a6ed
sample_task 1066efb2-f877-4fe8-967c-0d14bc2f22b5 SUCCESS (14,) {} {'status': '200'} 2022-10-17 14:48:20.442 2022-10-17 14:48:20.443 14.016 celery@ff7d8a15a6ed
Showing 1 to 10 of 103 entries
Previous
1
2
3
4
5
...
11
Next

RabbitMQ Dashboard

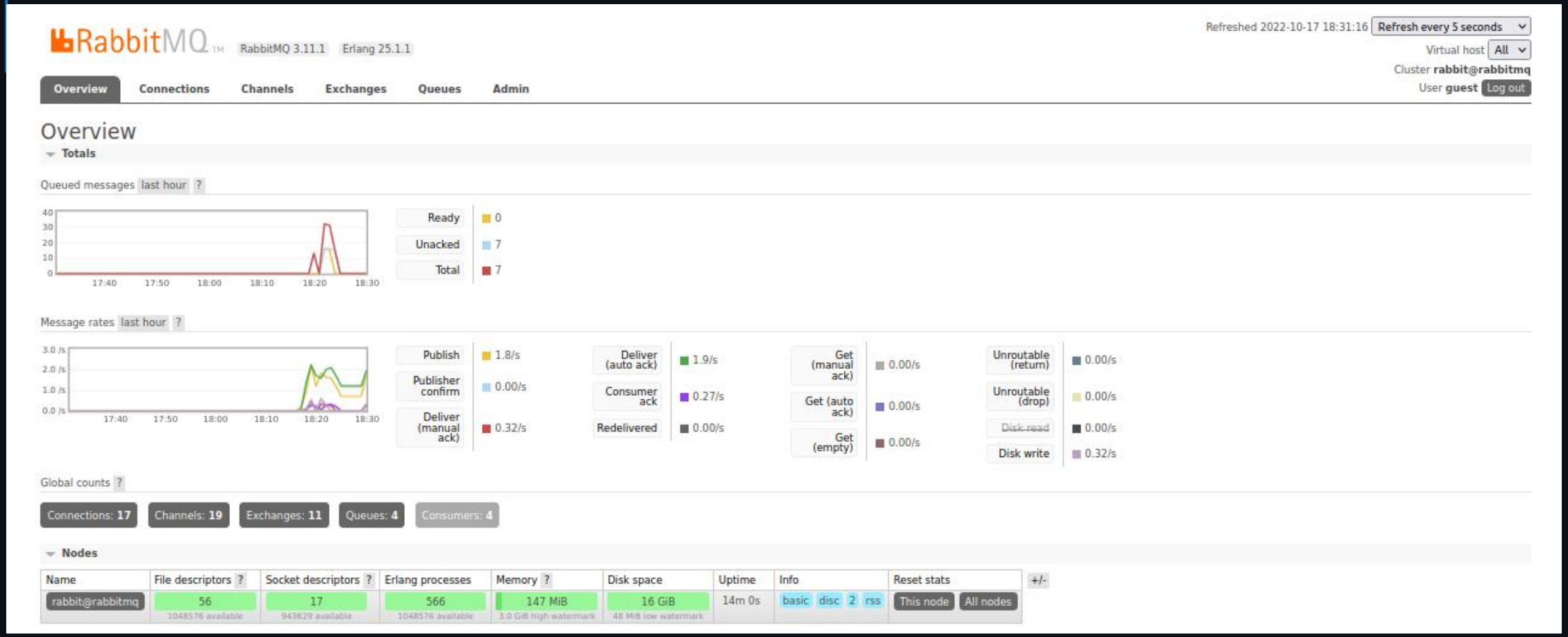
The RabbitMQ Management is a user-friendly interface that let you monitor and handle your RabbitMQ server from a web browser.
All the tabs from the menu are explained in this post. Screenshots from the views are shown
for: Overview, Connections and channels, Exchanges, Queues and Admin - users and permissions.

Queued messages

A chart of the total number of queued messages for all your queues. **Ready** show the number of messages that are available to be delivered. **Unacked** are the number of messages for which the server is waiting for acknowledgment

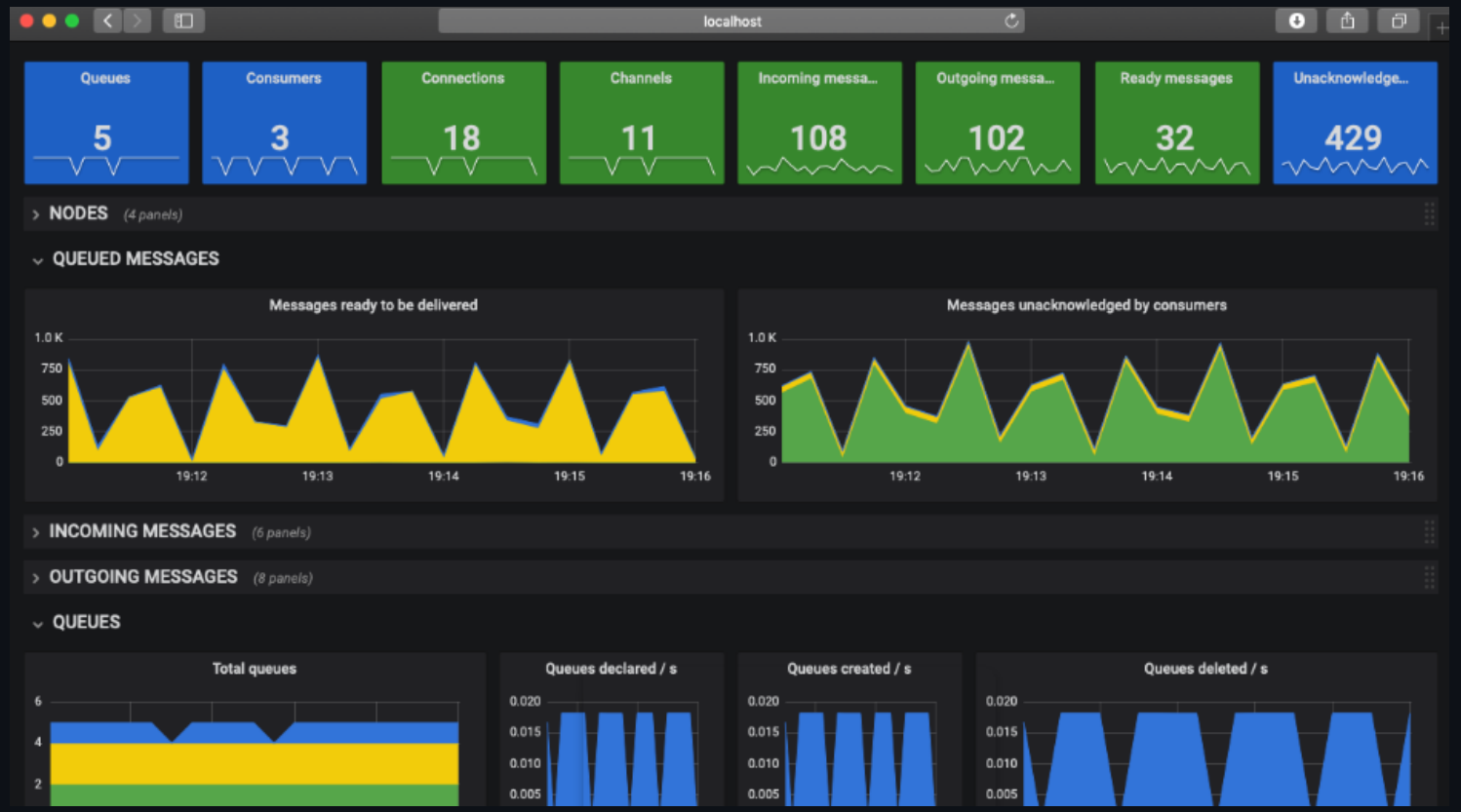
Messages rate

A chart with the rate of how the messages are handled. **Publish** show the rate at which messages are entering the server and **Confirm** show a rate at which the server is confirming.


RabbitMQ
RabbitMQ 3.11.1
Erlang 25.1.1
Refreshed 2022-10-17 18:31:16
Refresh every 5 seconds
Virtual host All
Cluster rabbit@rabbitmq
User guest
Log out
Overview
Connections
Channels
Exchanges
Queues
Admin
Overview
Totals
Queued messages last hour
Ready 0
Unacked 7
Total 7
Message rates last hour
Publish 1.8/s
Publisher confirm 0.00/s
Deliver (manual ack) 0.32/s
Deliver (auto ack) 1.9/s
Consumer ack 0.27/s
Redelivered 0.00/s
Get (manual ack) 0.00/s
Get (auto ack) 0.00/s
Get (empty) 0.00/s
Unroutable (return) 0.00/s
Unroutable (drop) 0.00/s
Disk read 0.00/s
Disk write 0.32/s
Global counts
Connections: 17
Channels: 19
Exchanges: 11
Queues: 4
Consumers: 4
Nodes
Name
File descriptors
Socket descriptors
Erlang processes
Memory
Disk space
Uptime
Info
Reset stats
rabbit@rabbitmq
56
17
566
147 MiB
16 GiB
14m 0s
basic disc 2 rss
This node
All nodes

Monitoring with Prometheus & Grafana

These tools together form a powerful toolkit for long-term metric collection and monitoring of RabbitMQ clusters. While RabbitMQ management UI also provides access to a subset of metrics, it by design doesn't try to be a long term metric collection solution.



```yaml
  rabbitmq:
    hostname: rabbitmq
    image: rabbitmq:3-management
    environment:
      - RABBITMQ_USERNAME=${RABBITMQ_USERNAME}
      - RABBITMQ_PASSWORD=${RABBITMQ_PASSWORD}
    container_name: rabbitmq
    ports:
      - 5672
      - 15672:15672
      - 15692:15692

  redis:
    container_name: redis
    image: redis:6-alpine
```

CONCLUSION

This has been a basic guide on how to configure Celery to run long-running tasks in a FastAPI app. You should let the queue handle any processes that could block or slow down the user-facing code.

Celery can also be used to execute repeatable tasks and break up complex, resource-intensive tasks so that the computational workload can be distributed across a number of machines to reduce (1) the time to completion and (2) the load on the machine handling client requests