Gen™

Celery and Other Distributed Task Queues In Python

CTO Python Guild

2025-03-28

Petr Z. & Oliver N. & Matúš J.

Celery

Article Talk

From Wikipedia, the free encyclopedia

For the software, see Celery (software).

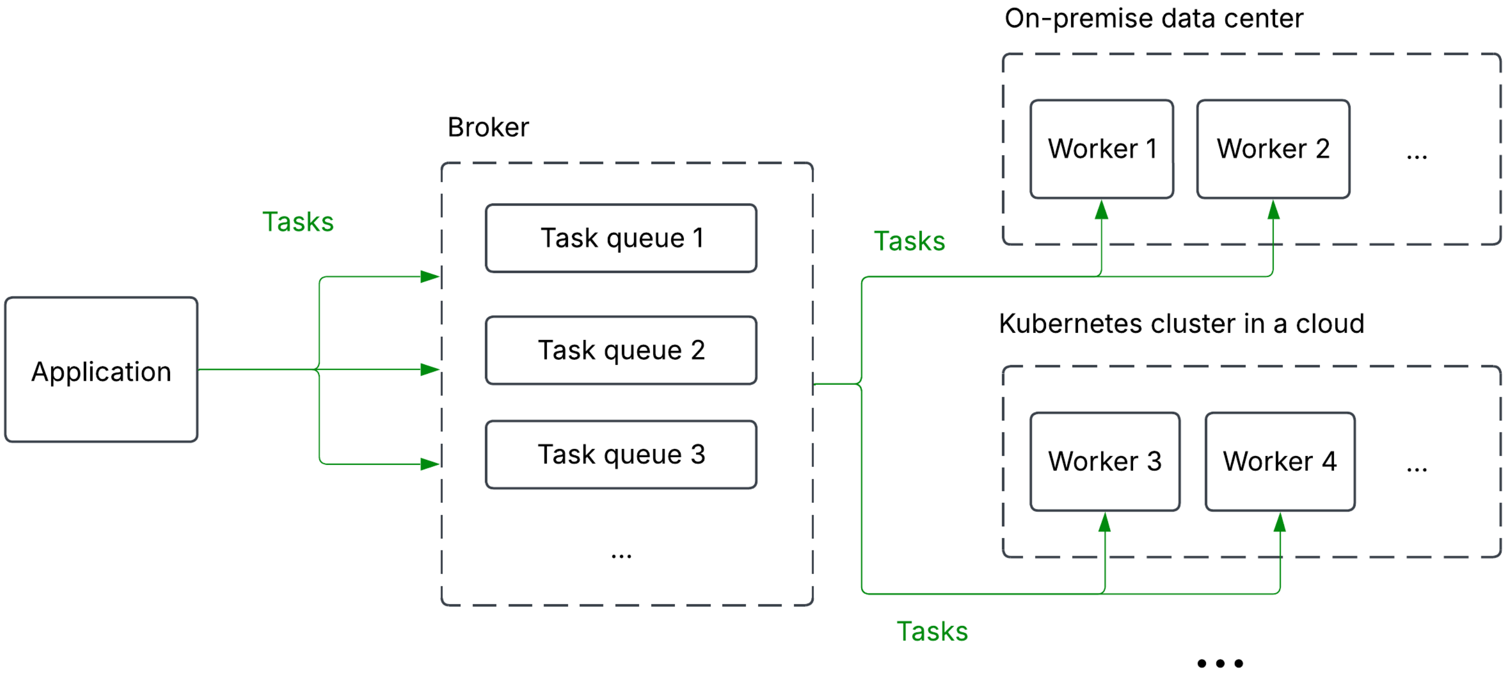
norton Avast LifeLock by norton Avira AVG REPUTATION DEFENDER by norton CCleaner

Outline

- Introduction to distributed task queues via Celery
- Celery demo
- Experience with Celery, pros & cons, lessons learned, ...
- Other distributed task queues (with focus on Dramatiq)
- Summary
- Q&A

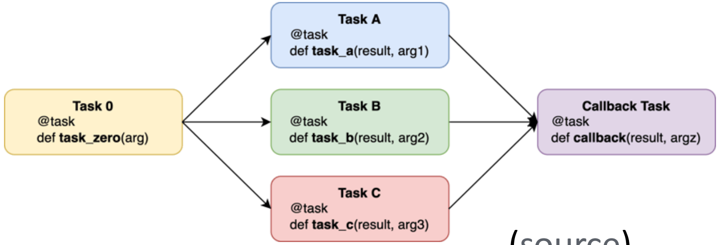
Introduction to Distributed Task Queues via Celery

What Is a Distributed Task Queue?

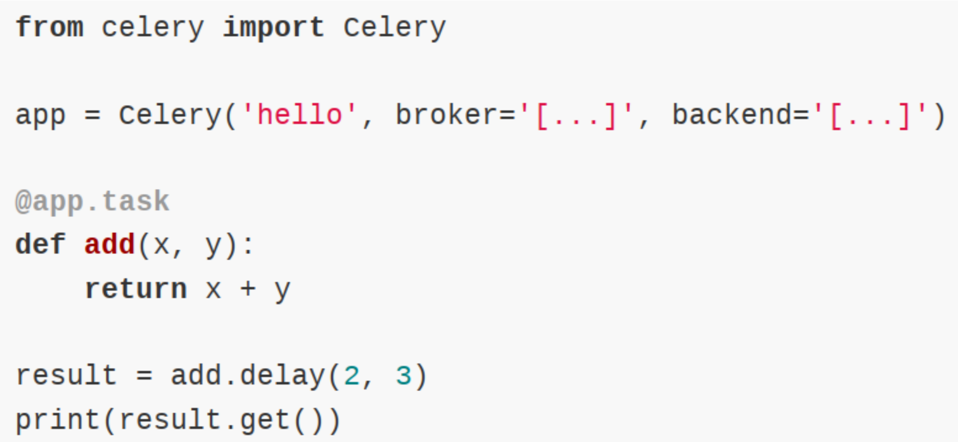


What Is Celery?

- An **open-source distributed task queue** written in Python (homepage)
- The **de facto standard** Python task queue (ref, 25.8k stars on GitHub)
- Gist: A **generalization** of the multiprocessing module to multiple machines
- **Use case examples**:
  - Offloading of CPU-heavy or long-running tasks from web/API
  - Task queuing/buffering
  - Periodic execution of tasks
  - Task flows (chain, group, chord, …)



(source)



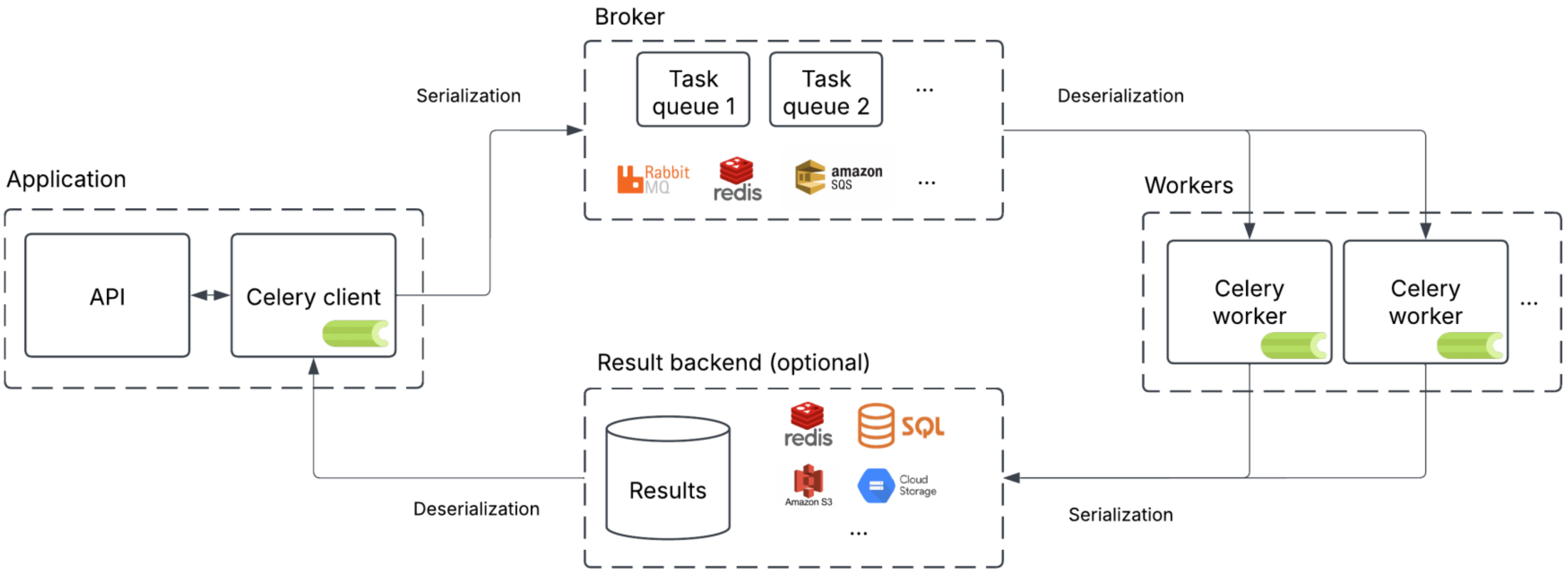
```
from celery import Celery

app = Celery('hello', broker='[...]', backend='[...]')

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

result = add.delay(2, 3)
print(result.get())
```

How Does It Work?



Other Notes and Features

- Supported **brokers**: RabbitMQ, Redis, Amazon SQS, Zookeeper/Kafka (experim.), GCP Pub/Sub (experim.)
- Supported **result backends**: Redis, SQL database, RabbitMQ, AWS S3, GCP GCS, and many more
- Supported task **serializers**: JSON, YAML, Pickle, MessagePack
- Supported **worker modes**: prefork (process pool), threads, eventlet, gevent
- Written in Python, but the **protocol** can be implemented in any language (Go, Rust, Ruby, PHP, ...)
  - Petr Zemek: Consuming and Publishing Celery Tasks in C++ via AMQP (blog post, 2017-06-25)
- Supports both automatic and custom task routing (including priorities)
- Worker management (inspect/control, dynamic pool growing/shrinking)
- Signals (hooking into the task mechanism)
- Highly configurable (configuration options)
- Flexible / extendable

Celery vs "Just RabbitMQ / Redis / ..."

Pros:

- Celery provides transparent RPC and result storage
- Celery comes with batteries included (task handling, serialization/deserialization, cron jobs, ...)
- Celery provides transparent flexibility for brokers (RabbitMQ, Redis, ...) and result storage (SQL database, Redis, ...)

Cons:

- Celery is more complex
- Celery supports features that you might not use
- Celery represents yet another dependency (might both decrease as well as increase complexity)
- Inter-language operability can be better with just RabbitMQ / Redis / ... (it depends)

Celery Demo

Celery Demo

[redacted]

Experience With Celery, Pros & Cons, Lessons Learned, ...

Experience With Celery, Pros & Cons, Lessons Learned, …

[redacted]

Other Distributed Task Queues in Python

Dramatiq

[redacted]

Other Distributed Task Queues

(For comparison, Celery has 25.8k stars and Dramatiq has 4.5k stars.)

- RQ (10.1k stars) - A Python library for queueing jobs and processing them
- APScheduler (6.6k stars) - Task scheduling library for Python
- Huey (5.4k stars) - A task queue for Python
- Rocketry (3.3k stars) - Modern scheduling library for Python
- Django Q (1.9k stars) - A multiprocessing distributed task queue for Django
- TaskTiger (1.4k stars) - Python task queue using Redis
- Taskiq (1.1k stars) - Distributed task queue with full async support
- And there is more (< 1k stars): taskmaster, tasq, kuyruk, django-carrot, …

Summary and Q&A

When (Not) To Use Distributed Task Queues

- General note: They are just another *tool*
- When to **consider using them**:
  - When the benefits (e.g. transparent RPC) outweigh the disadvantages (e.g. added overhead)
  - When you use a technology that integrates well with them (e.g. Celery and Django/Flask)
  - When you utilize the features they provide (e.g. task flows in Celery)
  - When there is a risk of needing to switch to a different broker or result store
- When you should **rather use something else**:
  - When the `multiprocessing` / `threading` / `asyncio` modules are sufficient (e.g. single machine)
  - When using just a message broker is sufficient (consider e.g. just RabbitMQ or Kafka)
  - When using just a database or key-value store is sufficient (consider e.g. just PostgreSQL or Redis)
  - When building a high performance/throughput system (consider e.g. just `socket` or ZeroMQ)
  - When you need some specific guarantees that are not provided (e.g. transactional task flows)

Summary and Q&A

- A **distributed task queue** is a mechanism to distribute work across multiple machines
- There are **many implementations** (more than 12 for Python alone…)
- **Celery** is the **de facto standard** Python task queue (open-source, 25.8k stars on GitHub)
- It supports **multiple** types of **brokers**, **result stores**, **serializers**, has **many features**, is highly configurable, …
- There are cases when Celery is a better option and cases when just RabbitMQ / Redis / … is better
- [redacted]
- Celery has its **own drawbacks**: bad defaults, requires config tuning, inefficient task scheduling via RabbitMQ, not a native technology for the cloud, complexity, no asyncio support, not many new features recently
- **Dramatiq** is a newer distributed task queue for Python, used as well but with much less experience, various differences from Celery
- Distributed task queues are a **tool** and so have **both pros and cons** and are **not** suitable for all use cases; **think before you use them!**