2014-06-18

CELERY - BEST PRACTICES

If you've worked with Django (https://www.djangoproject.com/) at some point you probably had the need for some background processing of long running tasks. Chances are you've used some sort of task queue, and Celery (http://www.celeryproject.org/) is currently the most popular project for this sort of thing in the Python (and Django) world (but there are others).

While working on some projects that used Celery for a task queue I've gathered a number of best practices and decided to document them. Nevertheless, this is more a rant about what I think should be the proper way to do things, and about some underused features that the celery ecosystem offers.

No.1: Don't use the database as your AMQP Broker

Let me explain why I think this is wrong (aside from the limitations (http://docs.celeryproject.org/en/latest/getting-started/brokers/django.html#limitations) pointed out in the celery docs).

A database is not built for doing the things a proper AMQP broker like RabbitMQ is designed for. It will break down at one point, probably in production with not that much traffic/user base.

I guess the most popular reason people decide to use a database is because, well, they already have one for their web app, so why not re-use it. Setting up is a breeze and you don't need to worry about another component (like RabbitMQ).

Not so hypothetical scenario: Let's say you have 4 background workers processing the tasks you've put in the database. This means that you get 4 processes polling the database for new tasks fairly often, not to mention that each of those 4 workers can have multiple concurrent threads of it's own. At some point you notice that you are falling behind on your task processing and more tasks are coming in than are being completed, so naturally you increase the number of workers doing the task processing. Suddenly your database starts falling apart due to the huge number of workers polling the database for new tasks, your disk IO goes through the roof and your webapp starts being affected by this slow down because the workers are basically DDOS-ing the database.

This does not happen when you have a proper AMQP like RabbitMQ (http://www.rabbitmq.com/) because, for one thing, the queue resides in memory so you don't hammer your disk. The consumers (the workers) do not need to resort to polling as the queue has a way of pushing new tasks to the consumers, and if the AMQP does get overwhelmed for some other reason, at least it will not bring down the user facing web app with it.

I would go as far to say that you shouldn't use a database for a broker even in development, what with things like Docker and a ton of pre-built images that already give you RabbitMQ out of the box (https://registry.hub.docker.com/search?q=rabbitmq).

No.2: Use more Queues (ie. not just the default one)

Celery is fairly simple to set up, and it comes with a default queue in which it puts all the tasks unless you tell it otherwise. The most common thing you'll see is something like this:

```
@app.task()
def my_taskA(a, b, c):
    print("doing something here...")

@app.task()
def my_taskB(x, y):
    print("doing something here...")
```

What happens here is that *both* tasks will end up in the same Queue (if not specified otherwise in the `celeryconfig.py` file). I can definitely see the appeal of doing something like this because with just one decorator you've got yourself some sweet background tasks. My concern here is that taskA and taskB might be doing totally different things, and perhaps one of them might even be much more important then the other, so why throw them both in the same basket? Even if you've got just one worker processing both tasks, suppose that at some point the unimportant taskB gets so massive in numbers that the more important taksA just can't get enough attention from the worker? At this point increasing the number of workers will probably not solve your problem as all workers still need to process both tasks, and with taskB so great in numbers taskA still can't get the attention it deserves. Which brings us to the next point.

No.3: Use priority workers

The way to solve the issue above is to have taskA in one queue, and taskB in another and then assign `x` workers to process Q1 and all the other workers to process the more intensive Q2 as it has more tasks coming in. This way you can still make sure that taskB gets enough workers all the while maintaining a few priority workers that just need to process taskA when one comes in without making it wait to long on processing.

So, define your queues manually:

```
CELERY_QUEUES = (
    Queue('default', Exchange('default'), routing_key='default'),
    Queue('for_task_A', Exchange('for_task_A'), routing_key='for_task_A'),
    Queue('for_task_B', Exchange('for_task_B'), routing_key='for_task_B'),
)
```

And your `routes` that will decide which task goes where:

```
CELERY_ROUTES = {
    'my_taskA': {'queue': 'for_task_A', 'routing_key': 'for_task_A'},
    'my_taskB': {'queue': 'for_task_B', 'routing_key': 'for_task_B'},
}
```

Which will allow you to run workers for each task:

```
celery worker -E -l INFO -n workerA -Q for_task_A
celery worker -E -l INFO -n workerB -Q for_task_B
```

No.4: Use Celery's error handling mechanisms

Most tasks I've seen in the wild don't have a notion of error handling at all. If a task fails that's it, it failed. This might be fine for some use cases, however, most tasks I've seen are talking to some kind of 3rd party API and fail because of some sort of network error, or other kind of "resource availability" error. The most simple way we can handle these kinds of errors is to just retry the task, because maybe the 3rd party API just had some server/network issues and it will be back up shortly, why not give it a go?

```
@app.task(bind=True, default_retry_delay=300, max_retries=5)
def my_task_A():
    try:
        print("doing stuff here...")
    except SomeNetworkException as e:
        print("maybe do some clenup here....")
        self.retry(e)
```

What I like to do is define per task defaults for how long should a task wait before being retried, and how many retries is enough before finally giving up (the `default_retry_delay` and `max_retries` parameters respectively). This is the most basic form of error handling that I can think of and yet I see it used almost never. Of course Celery offers more in terms of error handling but I'll leave you with the celery docs for that.

No.5: Use Flower

The Flower (http://celery.readthedocs.org/en/latest/userguide/monitoring.html#flower-real-time-celery-web-monitor) project is a wonderful tool for monitoring your celery tasks and workers. It's web based and allows you to do stuff like see task progress, details, worker status, bringing up new workers and so forth. Check out the full list of features in the provided link.

No.6: Keep track of results only if you really need them

A task status is the information about the task exiting with a success or failure. It can be useful for some kind of statistics later on. The big thing to note here is that the exit status is not the result of the job that the task was performing, that information is most likely some sort of side effect that gets written to the database (ie. update a user's friend list).

Most projects I've seen don't really care about keeping persistent track of a task's status after it exited yet most of them use either the default sqlite database for saving this information, or even better, they've taken the time and use their regular database (postgres or otherwise).

Why hammer your webapp's database for no reason? Use `CELERY_IGNORE_RESULT = True` in your `celeryconfig.py` and discard the results.

No.7: Don't pass Database/ORM objects to tasks

After giving this talk at a local Python meetup a few people suggested I add this to the list. What's it all about? You shouldn't pass Database objects (for instance your User model) to a background task because the serialized object might contain stale data. What you want to do is feed the task the User id and have the task ask the database for a fresh User object.

celery (../../categories/celery/) django (../../categories/django/) python (../../categories/python/)

15 Comments denibertovic.com Login

Sort by Best Share Favorite

Join the discussion…

Itamar Haber • 2 days ago
Any thoughts on using Celery with Redis instead of RabbitMQ?
1 • Reply • Share ›

Mark Hudnall Itamar Haber • 2 days ago
We've had some serious issues with ETA tasks when using celery with redis. Basically, whenever we gracefully restarted the workers, all or some delayed tasks were completely dropped and never restored. The issue was tracked here (https://github.com/celery/cele.... We tried switching a bunch of different options, switching the way we restarted workers, but nothing worked. We ended up switching to RabbitMQ and haven't had any dropped tasks since.
• Reply • Share ›

denibertovic Mod Itamar Haber • 2 days ago
There are some comments (by me and others) on HN on this subject, but generally out of all the options I would only recommend RabbitMQ and Redis. It works quite nice really. Keep in mind though, Redis uses Pub/Sub and is quite fast but still it's not an AMQP implementation like RabbitMQ, so there are trade-offs.
• Reply • Share ›

Itamar Haber denibertovic • 2 days ago
Gotcha - heading to HN to get the rest of the info, and yeah - Pub/Sub's awesome but I guess the fact that a disconnected client doesn't get to catch up on what happened is problematic in this context. It is possible, however, to build a reliable queue you can use Redis' lists rather than Pub/Sub (I believe that's what Sidekiq's doing).
• Reply • Share ›

OrangeTux • 2 days ago
+1 for mentioning Flower.
1 • Reply • Share ›

denibertovic Mod OrangeTux • 2 days ago
It's something one cannot do without once one discovers it. Thankfully it has a nice googlable name the same as celery. :D
1 • Reply • Share ›

rosscdh • 2 days ago

Very good post. Thanks. Just re: the not passing of objects. What we are doing is .. pass the object "obj" and then in the task

```
obj = obj.__class__.objects.get(pk=obj.pk)
```

et voila: instance current object sauce.

1 ^ | ˅ • Reply • Share ›

denibertovic Mod → rosscdh • 2 days ago

Awesome. Tnx for the tip.

^ | ˅ • Reply • Share ›

Stockman • 32 minutes ago

thanks man.

^ | ˅ • Reply • Share ›

Thodoris • 2 days ago

Has anyone used Celery with Eventlet (+RabbitMQ) ?
In the documentation its recommended that:
"You may want a mix of both Eventlet and prefork workers, and route tasks according to compatibility or what works best".

I have a task that performs thousands of requests (ideal for Eventlet) and then a task that is passed the html (fetched from the previous task) and does several computations (e.g searches for keywords, saves html, js to database). Is it better to chain these two tasks (with the | operator) or keep them separate and use eventlet for the one and prefork workers for the other, or maybe prioritise these tasks ?

^ | ˅ • Reply • Share ›

denibertovic Mod → Thodoris • a day ago

I've used workers with eventlet concurrency yes. It's a great thing to do if your tasks are largely IO bound and there's a lot of them. I would have the IO bound tasks use eventlet and just spawn another task when they're done (ie. pass the html to the other task) that can be preforked.

^ | ˅ • Reply • Share ›

Zen • 2 days ago

Passing objects to Celery and not querying for fresh objects is not always a bad practice. If you have millions of rows in your database, querying for them is going to slow you way down. In essence, the same reason you shouldn't use your database as the Celery backend is the same reason you might not want to query the database for fresh objects. It depends on your use case of course. Passing straight values/strings should be strongly considered too since serializing and passing whole objects when you only need a single value is not good either.

Reply • Share ›

denibertovic Mod ➦ Zen • 2 days ago

As I've commented on HN I would not sacrifice the correctness of my application because of the inability to scale the Database. If the computation requires a fresh set of data then there's no way around it. Oh and yes I agree, values over objects almost always.

Reply • Share ›

udi • 2 days ago

Nice article. You should put title tags on your post pages, not just your name.

Reply • Share ›

denibertovic Mod ➦ udi • 2 days ago

Tnx. The title did use to work, guess the last template refactor I did messed something up. Tnx for the tip. :)

Reply • Share ›

ALSO ON DENIBERTOVIC.COM

WHAT'S THIS?

One-liner Instant Postgres for your development environment

1 comment • 11 months ago

Marko Elezović — Interesting - it seems Docker is becoming more and more popular nowadays. Definitely cheaper than running a

Classes for all the things?

3 comments • 5 months ago

Tinche — An interesting tidbit about partial is that it doesn't create an intermediate stack frame. I always assumed a naive

Docker: Smarter Log Management For Your Containers

4 comments • 5 months ago

The switch to Nikola

1 comment • 10 months ago

Roberto Alsina — Glad you liked it! Also: nice