Chewing On Celery

Author Ask Solem

What is Celery

- Celery is an asynchronous task queue/job queue based on distributed message passing.
- It is focused on real-time operation, but supports scheduling as well.
- The execution units, called tasks, are executed concurrently on a single or more worker servers using multiprocessing, Eventlet, or gevent.
- Tasks can execute asynchronously (in the background) or synchronously (wait until ready).

Getting Started

- Choose a broker
  - AMQP (Rabbit MQ)
  - Redis
  - Database
  - Amazon SQS, Iron MQ, Mongo DB

Getting Started

- Choose a broker
  - AMQP (Rabbit MQ)
  - Redis
  - Database
  - Amazon SQS, Iron MQ, Mongo DB

Install Celery

- pip install celery
  - Best inside a virtual environment
- Or easy_install celery
  - blech

Set up celeryconfig.py

```
import os

BROKER_URL = "redis://{host}:6379/0".format(
  host=os.environ.get("REDIS_ADDR")
)
CELERY_RESULT_BACKEND = BROKER_URL
CELERY_IMPORTS = ("queue.tasks.retrieve",)
CELERY_TASK_SERIALIZER = "json"
CELERY_TIMEZONE = "UTC"
```

Set up first task

```
from __future__ import absolute_import

from queue.app import celery
import requests
from celery.utils.log import get_task_logger
import json

logger = get_task_logger(__name__)

@celery.task
def scour(target):
    logger.info("Retrieving %s" % target)
    req = requests.get("http://%s" % target)
    content = req.content
    logger.info("Length of content: %d", len(content))
    return content
```

Run Celery

```
celery -A queue.app worker --config queue.celeryconfig --loglevel debug
```

```
 -------------- celery@ubuntu v3.1.12 (Cipater)
---- **** -----
--- * ***  * -- Linux-3.5.0-46-generic-x86_64-with-Ubuntu-12.04-precise
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:          queue.celery:0x2aa3990
- ** ---------- .> transport:    redis://127.0.0.1:6379/0
- ** ---------- .> results:      redis://127.0.0.1:6379/0
- *** --- * --- .> concurrency: 4 (prefork)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> celery           exchange=celery(direct) key=celery


[tasks]
  . queue.tasks.retrieve.scour
```

Run the task

```
import os
os.environ['CELERY_CONFIG_MODULE'] = 'queue.celeryconfig1'
from queue.tasks import retrieve

t = retrieve.scour.delay("www.google.com")
print t
```

Run the task and wait

```
import os
os.environ['CELERY_CONFIG_MODULE'] = 'queue.celeryconfig1'
from queue.tasks import retrieve

t = retrieve.scour.delay("www.google.com")
print t.get()
```

Calling from a web app

```
@app.route("/leaderboard/generate/<game>")
def leaderboard_generate(game='CTF'):
    conn = get_conn()
    game_type = game
    teams = conn.smembers('teams')
    chains = (chain(fs.s(team, game_type=game_type), cs.s())for team in teams)
    g = group(*chains)
    t = chord(g)(dw.s(game_type))
    return "OK"
```

Callbacks, Groups, Chains and Chords

- Callbacks
  - Or subtasks, take the result of a task as their first argument
- Groups
  - Used to execute several tasks in parallel
- Chains
  - Used to chain one subtask (or group of subtasks) to another
  - First argument to chained task will be the result of the previous subtask or group of subtasks
- Chord
  - Think of it like a callback for a group
  - Complete the group of work, then apply a final task

Callbacks

- Or subtasks, take the result of a task as their first argument

```
c = canvas.add.apply_async((2, 5), link=canvas.mul.s(30))
print c.get()
print c.children
print c.children[0].get()
```

Groups

- Used to execute several tasks in parallel

```
g = group(canvas.add.s(2, 5), canvas.add.s(10, 30))()
print g.get()

g = group([canvas.add.s(i, i) for i in range(100)])
result = g.apply_async()
print result.get()
```

Chains

- Used to chain one subtask (or group of subtasks) to another.
- First argument to chained task will be the result of the previous subtask or group of subtasks

```
c = canvas.add.apply_async((2, 5), link=canvas.mul.s(30))
print c.get()
print c.children
print c.children[0].get()

c = chain(canvas.add.s(2, 5), canvas.mul.s(30))()
print c.get()
```

Chords

- Think of it like a callback for a group
- Complete the group of work, then apply a final task

```
g = group([canvas.add.s(i, i) for i in range(100)])
c = chord(g)(canvas.tsum.s())
print c.get()
```

Submitting tasks from another language

- Two options
  - Implement an external interface which actually creates the task. HTTP, *RPC, etc
  - Use the protocol to submit the job to the queue directly

Case study - Call Of Duty competition leaderboards

- Players register on teams
- Teams play a particular game-mode / map / with a particular weapon / etc during a specified time period
- Individual leaderboards are created where top scorers are awarded prizes
- Team leaderboards are created where top teams are awarded prizes

Advanced topics / configurations

- Monitoring
- Retrying
- ~~Rate limiting~~
- ~~Setting time limits~~
- ~~Celerybeat - Nicer crontab~~

Monitoring

- You can roll your own monitoring like I did
- Or you can use Flower (pronounced Flow-er)
  - This is a much better idea!

Retrying

- You can retry failed tasks
- You can create a custom retry strategy
- With acks_late you can guarantee processing

We're Hiring

- Great pay, benefits, stock options, and bonus plan
- Top notch developers
- Challenging problems
- Large scale distributed cloud architecture
- We use Python
- "Hilarious quotes" wiki page from IRC logs

crowdstrike.com/about-us/careers/