Distributed Systems

Celery

Celery python dilində yazılmış proqramdır. Onun köməkliyi ilə biz sistemimizin işlərini müxtəlif maşınlar arasında rahatlıqda paylaşdıra bilirik. Celery python dilində yazılmağına baxmayaraq o digər dillərdə yazılmış application-lar üçün də istifadə edilə bilir

Nümunə

PİP vasitəsilə **celery** və **redis** kitabxanasını install edin. Növbəti kodu yazın və modul kimi saxlayın:

```
from celery import Celery
app = Celery('test', broker='amqp://elshad:123456@172.17.0.1', backend='redis://172.17.0.1')

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

Daha sonra **worker.py** adlı fayl yaradın və workeri işə salın:

```
from app import app

app.worker_main()
```

task.py adlı fayl yaradın:

```
import app

result = app.add.delay(10, 15)
result.get()
```

Celery Tasks

Tasks

Task kimi dekorasiya olunmuş funksiyalar istədiyimiz vaxt remote kompüterlərdə paralel icra olunmağa hazır vəziyyətə gətirilir. Bunun üçün sadəcə həmən funksiyanı app.task() funksiyası ilə dekorasiya etmək lazımdır. Daha sonra remote kompüterdə worker run edərək bu funksiyanı remote execute edə bilirik. Əgər task etmək istədiyimiz funksiya başqa dekoratorlardan da istifadə edirsə onda @app.task dekoratoru ən üstdə qoyulmalıdır:

```
@app.task
@decorator2
@decorator1
def add(x, y):
    return x + y
```

Bağlanmış tasklar

@app.task dekoratoru ehtiyac olarsa əlavə parametrlər də qəbul edə bilir. Bunlardan biri *bind* parametridir. Əgər bind parametri True olarsa onda dekorasiya edilən funksiyanın ilk arqumenti task obyektinin özü olacaq. Bind üsulu adətən task əgər fail olarsa onu retry üsulu ilə yenidən işlətməkdir:

```
@app.task(bind=True)
def add(self, x, y):
    try:
        time.sleep(1)
        raise Exception
    except Exception as err:
        raise self.retry(countdown=1, max_retries=10, exc=err)
```

Bağlanmış tasklar

celery.Task.retry() funksiyasının arqumentləri aşağıdakılardır:

- args (Tuple) – Retry vaxtı əvvəlki positional arqumentləri dəyişmək üçün istifadə olunur
- kwargs (Dict) – Retry vaxtı əvvəlki optional arqumentləri dəyişmək üçün istifadə olunur
- exc – max_retries bitdikdən sonra baş verən xətanı bildirmək üçün istifadə olunacaq
- countdown (float) – Exception baş verdikdən sonra yenidən cəhd etmənin vaxtını bildirir
- eta (datetime) – Yenidən cəhd etmənin dəqiq vaxtını göstərir
- max_retries (int) – Yenidən cəhdin sayını bildirir. Əgər None olarsa onda settingsdə olan dəyər olacaq

Task name

Hər bir taskın adı unikal olmalıdır. Əgər biz taska ad verməsək onda taskın adı yerləşdiyi modul və tasklaşdırılan funksiyanın adından götürüləcək. Misalçün əgər bizim tasklaşdırılan funksiya task.py faylında yerləşirsə onda taskın adı aşağıdakı kimi olacaq:

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

print(add.name)
```

```
test.add
```

```python
@app.task(name='my-unique-task')
def add(x, y):
    return x + y

print(add.name)
```

```
my-unique-task
```

Task Request

Request özündə cari icra olunan taskın vəziyyəti haqda məlumatlar saxlayır:

```
@app.task(bind=True)
def add(self, x, y):
    print("Request:", self.request)
```

Logger

Celery bizə tasklarda baş verən prosesləri loglamaq üçün imkan yaradır. Celery pythonun standard logging kitabxanasından istifadə edir. Bütün tasklar üçün tək logger istifadə etmək məsləhətdir. Bunun üçün modulunuzun əvvəlində onu import edərək common obyekt yaradın. Print də həmçinin logger kimi istifadə oluna bilər. Lakin ayarlardan *worker_redirect_stdouts* dəyişildikdə bu vəziyyət dəyişiləcək:

```
@app.task
def add(x, y):
    logger.info(f"Adding {x} + {y}")
    logger.warn(f"This is warning message from logger")
    return x + y
```

[2018-08-31 01:09:49,991: INFO/ForkPoolWorker-2] logger.add[327d3762-73e7-4530-bc1f-4fd48bdc1780]: Adding 10 + 15
[2018-08-31 01:09:49,991: WARNING/ForkPoolWorker-2] logger.add[327d3762-73e7-4530-bc1f-4fd48bdc1780]: This is warning message from logger

Automatic retry for knowing exceptions

Celery bizə tasklarda baş verən prosesləri loglamaq üçün imkan yaradır. Celery pythonun standard logging kitabxanasından istifadə edir. Bütün tasklar üçün tək logger istifadə etmək məsləhətdir. Bunun üçün modulunuzun əvvəlində onu import edərək common obyekt yaradın. Print də həmçinin logger kimi istifadə oluna bilər. Lakin ayarlardan *worker_redirect_stdouts* dəyişildikdə bu vəziyyət dəyişiləcək:

```
@app.task
def add(x, y):
    logger.info(f"Adding {x} + {y}")
    logger.warn(f"This is warning message from logger")
    return x + y
```

[2018-08-31 01:09:49,991: INFO/ForkPoolWorker-2] logger.add[327d3762-73e7-4530-bc1f-4fd48bdc1780]: Adding 10 + 15
[2018-08-31 01:09:49,991: WARNING/ForkPoolWorker-2] logger.add[327d3762-73e7-4530-bc1f-4fd48bdc1780]: This is warning message from logger

Task.throws

Bu parametrin dəyəri tuple tipidir. Bu parametrə tanıdılan xətalar worker tərəfdə traceback məlumatları ilə loglanmayacaq:

```
@app.task(throws=(KeyError,TypeError))
def add(x, y):
    exceptions = (KeyError, TypeError, Exception)
    time.sleep(2)
    raise exceptions[random.randint(0,2)]
```

Task.throws

Bu parametrin dəyəri tuple tipidir. Bu parametrə tanıdılan xətalar worker tərəfdə traceback məlumatları ilə loglanmayacaq:

```
@app.task(throws=(KeyError,TypeError))
def add(x, y):
    exceptions = (KeyError, TypeError, Exception)
    time.sleep(2)
    raise exceptions[random.randint(0,2)]
```

States

Celery bütün tapşırıqların statuslarını izləyir. Default olaraq STARTED, SUCCESS, FAILURE, RETRY, REVOKED kimi statuslar var. Lakin biz özümüz də custom status artıra bilərik:

```
@app.task(bind=True)
def add(self, x, y):
    self.update_state(state='MY_STATE', meta={
        'field 1': 'info for field 1',
        'field 2': 'info for field 2',
        'field 3': 'info for field 3'
    })
```

Abstract Task

Biz həmçinin öz şəxsi task klasımızı yarada bilərik:

```
from celery import Celery, Task
from celery.utils.log import get_task_logger

app = Celery('abstract_task', broker='amqp://elshad:123456@172.17.0.1', backend='redis://172.17.0.1')
logger = get_task_logger(__name__)

class DebugTask(Task):

    def __call__(self, *args, **kwargs):
        logger.info('TASK STARTING: {0.name}[{0.request.id}]'.format(self))
        return super(DebugTask, self).__call__(*args, **kwargs)

@app.task(base=DebugTask)
def add(x, y):
    return x + y
```

Custom request and handlers

Task sorğusu zamanı bəzi *on_failure, after_return, on_retry, on_success* kimi eventlər baş verir. Bu eventləri idarə edə bilmək üçün biz öz şəxsi reqeust klasımızı yarada bilərik:

```
from celery import Celery, Task
from celery.utils.log import get_task_logger
from celery.worker.request import Request

logger = get_task_logger(__name__)
app = Celery('logger', broker='amqp://elshad:123456@172.17.0.1', backend='redis://172.17.0.1')

class MyRequest(Request):
    def on_failure(self, *args, **kwargs):
        logger.info("On failure from task side: args: " + str(args) +", kwargs: " + str(kwargs))

class MyTask(Task):
    Request = MyRequest

@app.task(base=MyTask)
def add(x, y):
    raise Exception
```

Ignore result

Bəzən dəyər qaytaran tapşırıqların qaytardıqları nəticə vacib olmur. Bu zaman yüklənməni azaltmaq məqsədi ilə həmən taskların qaytardığı nəticəni ignore edə bilirik:

```
from celery import Celery, Task, exceptions

app = Celery('logger', broker='amqp://elshad:123456@172.17.0.1', backend='redis://172.17.0.1')

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


@app.task()
def explicitly_ignored(number):
    if number > 10:
        raise exceptions.Ignore()
    else:
        return number ** 2
```

Tapşırıqların çağırılması

Tapşırıqlar .delay() metodu ilə işlədildikdə bizə obyekt qaytarır. Bu obyektin .get() metodu vasitəsi ilə nəticəni alırıq. Lakin bu metodun qəbul etdiyi digər parametrlər var. Onların bəziləri ilə tanış olun:

```
from get_arguments import add

res = add.delay(5, 10)

def on_message(result):
    print("on_message status:", result['status'])
    print("on_message result:", result['result'])
    print("on_message traceback:", result['traceback'])
    print("on_message children:", result['children'])
    print("on_message task_id:", result['task_id'])

def callback(task_id, result):
    print("callback:", task_id, result, type(result))

def on_interval():
    print("on_interval: started")

result = res.get(timeout=10,
                 propagate=False,
                 on_message=on_message,
                 callback=callback,
                 on_interval=on_interval,
                 no_ack=False,
                 interval=4)
```

Celery Workers

Workerin başladılması

Worker növbəti əmrlə başladılır: ***celery -A proyekt worker -l info***
Burada proyekt - taskın adıdır, -l isə log leveldir.
Worker haqda daha ətraflı məlumat almaq üçün: ***celery worker --help***

Workerin başladılması

Worker növbəti əmrlə başladılır: ***celery -A proyekt worker -l info***
Burada proyekt - taskın adıdır, -l isə log leveldir.
Worker haqda daha ətraflı məlumat almaq üçün: ***celery worker --help***

Workerin dayandırılması və yenidən başladılması

Worker aşağıdakı üsullarla dayandırıla bilər:

1. *pkill -9 -f 'celery worker'*
2. ps auxww | grep 'celery worker' | awk '{print $2}' | xargs kill -9

Workeri yenidən işə salmaq üçün ***multi*** əmrindən istifadə etmək lazımdır. Əvvəlcə signal göndərilir daha sonra start edilir:

1. *celery multi start 1 -A proj -l info -c4 --pidfile=/var/run/celery/%n.pid*
2. *celery multi restart 1 --pidfile=/var/run/celery/%n.pid*

Concurrency

Worker işə salınan maşında prosessorların və nüvələrin sayından asılı olaraq concurrency sayı təyin etmək olar. Bununla paralel proseslərin sayını müəyyən edirik. Default olaraq maşının resursları sayı qədər olacaq:

celery -A proj worker -l info --concurrency=10

Statistics

Aşağıdakı əmrlə workerin statistikasını görmək olar:

celery -A proj inspect stats

Periodic Tasks

Periodic Tasks

Celery tapşırıqları crontab kimi periodic yerinə yetirə bilir. Bunun üçün aşağıdakı kodları yazmaq lazımdır:

```python
from celery import Celery
from celery.schedules import crontab

app = Celery()

@app.on_after_configure.connect
def setup_periodic_tasks(sender, **kwargs):
    # Calls test('hello') every 10 seconds.
    sender.add_periodic_task(10.0, test.s('hello'), name='add every 10')

    # Calls test('world') every 30 seconds
    sender.add_periodic_task(30.0, test.s('world'), expires=10)

    # Executes every Monday morning at 7:30 a.m.
    sender.add_periodic_task(
        crontab(hour=7, minute=30, day_of_week=1),
        test.s('Happy Mondays!'),
    )

@app.task
def test(arg):
    print(arg)
```

Periodic Tasks

.add_periodic task() tapşırığı ***beat_schedule*** ayarına əlavə edəcək. Onu manual olaraq da etmək olar:

```
app.conf.beat_schedule = {
    'add-every-30-seconds': {
        'task': 'tasks.add',
        'schedule': 30.0,
        'args': (16, 16)
    },
}
app.conf.timezone = 'UTC'
```

Django ilə inteqrasiya

Django ilə celery inteqrasiya

Rəsmi dokumentasiyaya əsaslanaraq addım-addım inteqrasiya edək:

http://docs.celeryproject.org/en/latest/django/first-steps-with-django.html