Celer

y

Dernière mise-à-jour : 31/03/2015

Prérequis et documentation

site de Celery :http://www.celeryproject.org/

site de RabbitMQ : https://www.rabbitmq.com/

site de Erlang : http://www.erlang.org/

Pour ma part, j'ai fait un POC en utilisant Python (3.3) et RabbitMQ. Il est également possible d'utiliser d'autres brokers comme Redis, MongoDD, IronMQ, ou AmazonSQ.

A mettre en place par la suite :

Github pour un client Celery pour NodeJs (Merci à Fred.) : https://github.com/mher/node-celery

Installation

La documentation sur le site de Celery est très bien faite, n'hésitez pas à la consulter.

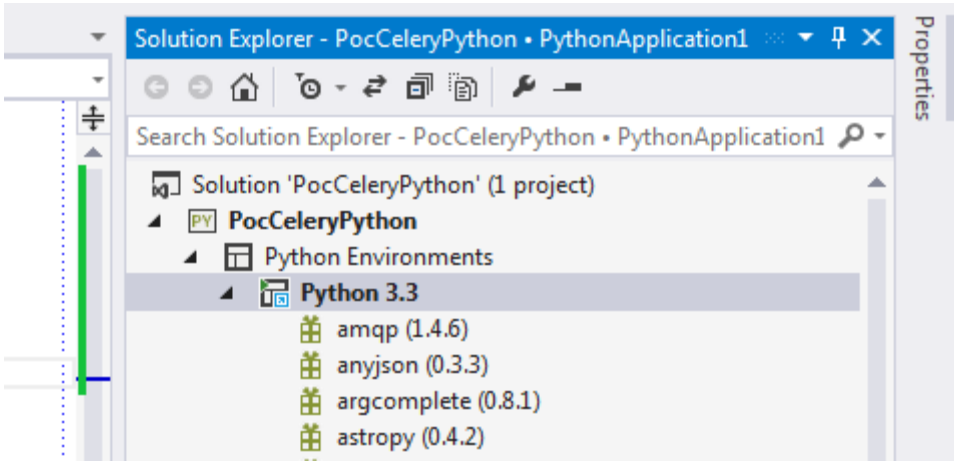
Etape 1 : installation de celery

Dans cygwin, tapez :

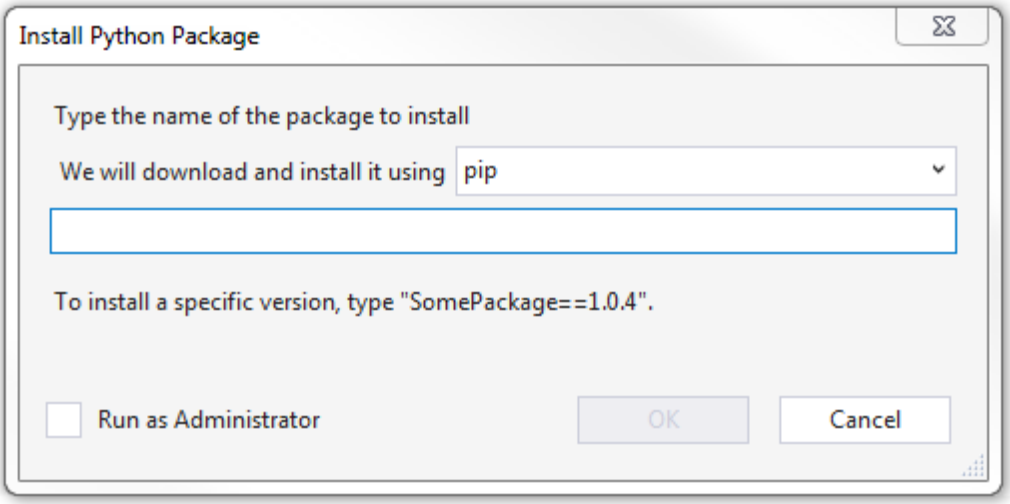
```
$ pip install celery
```

pour l'installer dans votre projet python :

déplier le python environnement
clic droit sur Python 3.3



cliquer sur "install python package"
une fenetre s'ouvre



taper celery et cliquer sur "ok"

Etape 2 : installation de RabbitMQ

Etape 2.1 : installation de Erlang

Pour pouvoir finaliser l'installation de RabbitMQ, il faut installer Erlang (http://www.erlang.org/download.html)

Télécharger la version suivante : OTP 17.4 Windows 64-bit Binary File

et l'installer

Etape 2.2 : installation de RabbitMQ

Télécharger l'executable de RabbitMQ (https://www.rabbitmq.com/install-windows.html) et suivre les instructions (rien de compliqué)

Lancement du worker

On va tester si l'installation fonctionne correctement. Pour ce faire, dans votre projet python vous créez ajouter une classe vide nommée tasks et vous copiez le code suivant à l'interieur :

```
from celery import Celery

app = Celery('tasks', broker='amqp://guest@localhost//')

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

Ouvrez cygwin et déplacer vous dans le répertoire ou vous avez créez votre projet.

```
cd /cygdrive/c/Users/votreNom/.../PythonApplication
```

puis tapez la commande suivante :

```
celery -A tasks worker --loglevel=info
```

Le celery worker server se lancera alors, vous dever obtenir ceci :

```
ddelallee@PC-W7-DEV69 /cygdrive/c/Users/ddelallee/Documents/Visual Studio 2013/Projects/PythonApplic
$ celery -A proj worker --loglevel=info
[2015-03-27 12:47:31,343: WARNING/MainProcess] /usr/lib/python3.2/site-packages/celery/apps/worker.p
Starting from version 3.2 Celery will refuse to accept pickle by default.

The pickle serializer is a security concern as it may give attackers
the ability to execute any command.  It's important to secure
your broker from unauthorized access when using pickle, so we think
that enabling pickle should require a deliberate action and not be
the default choice.

If you depend on pickle then you should set a setting to disable this
warning and to be sure that everything will continue working
when you upgrade to Celery 3.2::

    CELERY_ACCEPT_CONTENT = ['pickle', 'json', 'msgpack', 'yaml']

You must only enable the serializers that you will actually use.


  warnings.warn(CDeprecationWarning(W_PICKLE_DEPRECATED))

 -------------- celery@PC-W7-DEV69 v3.1.17 (Cipater)
---- **** -----
--- * ***  * -- CYGWIN_NT-6.1-WOW64-1.7.33-2-0.280-5-3-i686-32bit-WindowsPE
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         proj:0xff91a5ec
- ** ---------- .> transport:   amqp://admin:**@bizperf45:5672//
- ** ---------- .> results:     amqp://
- *** --- * --- .> concurrency: 4 (prefork)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> celery           exchange=celery(direct) key=celery


[tasks]
  . proj.tasks.add
  . proj.tasks.mul

[2015-03-27 12:47:31,520: INFO/MainProcess] Connected to amqp://admin:**@bizperf45:5672//
[2015-03-27 12:47:31,535: INFO/MainProcess] mingle: searching for neighbors
[2015-03-27 12:47:32,575: INFO/MainProcess] mingle: all alone
[2015-03-27 12:47:32,634: WARNING/MainProcess] celery@PC-W7-DEV69 ready.
```

Retournons dans le projet python puis éxecuter votre classe :
une console s'ouvre et n'affiche rien. Tout est normal. Si vous regarder maintenant votre cygwin ou plutot votre celery worker server, de nouvelles infos s'y trouvent et
vous indique que votre tâche s'est correctement déroulée.

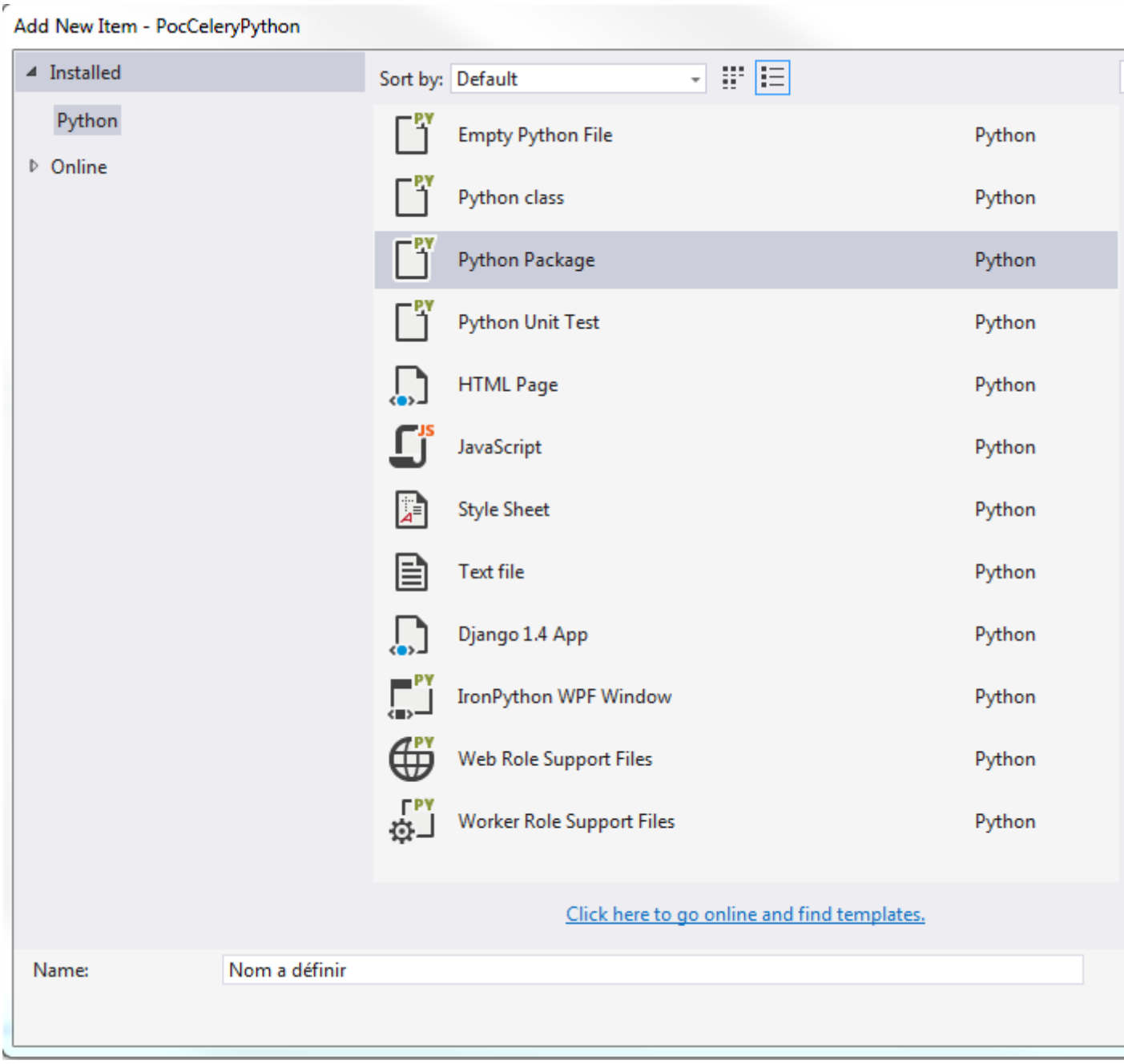
```
[2015-03-27 12:26:51,113: INFO/MainProcess] Received task: tasks.add[b9a4a75f-9cd2-4436-b4a1-1400b7
[2015-03-27 12:26:51,115: INFO/MainProcess] Task tasks.add[b9a4a75f-9cd2-4436-b4a1-1400b727fb20] su
```

Félicitations, votre celery worker server fonctionne correctement et vous avez envoyé votre première tâche !!!

Création d'un projet

Retournons dans Visual studio, et créons un projet qui permettra de séparer un peu mieux le code.

On va donc créer un package. Pour ce faire, clic droit sur le nom de votre projet puis add -> New item ...
Vous tombez alors sur la fenetre suivante :



Créer votre package et créer deux classes à l'intérieur. La première que l'on appelera celery.py et la deuxieme s'appelera tasks.py

La première classe contient l'instance de Celery et la lance.

nomPackage/celery.py

```python
from __future__ import absolute_import
from celery import Celery

app = Celery('proj',
             broker='amqp://admin:Password2Change@bizperf45:5672//',
             backend='amqp://',
             include=['proj.tasks'])

# Optional configuration, see the application user guide.
app.conf.update(
    CELERY_TASK_RESULT_EXPIRES=3600,
)



if __name__ == '__main__':
    app.start()

```

La deuxième classe contient toutes les tâches que l'on pourra éxecuter par la suite.

nomPackage/tasks.py

```python
from __future__ import absolute_import
from proj.celery import app

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

```

On pourra donc lancer soit une addition, soit une multiplication.

On va maintenant créer un autre fichier python en dehors du package.
Peut importe le nom, c'est le fichier que l'on executera lorsque le celery worker server sera relancé.

Dans cette classe on va importer la classe task qui contient les commandes et on effectura les appels aux méthodes.

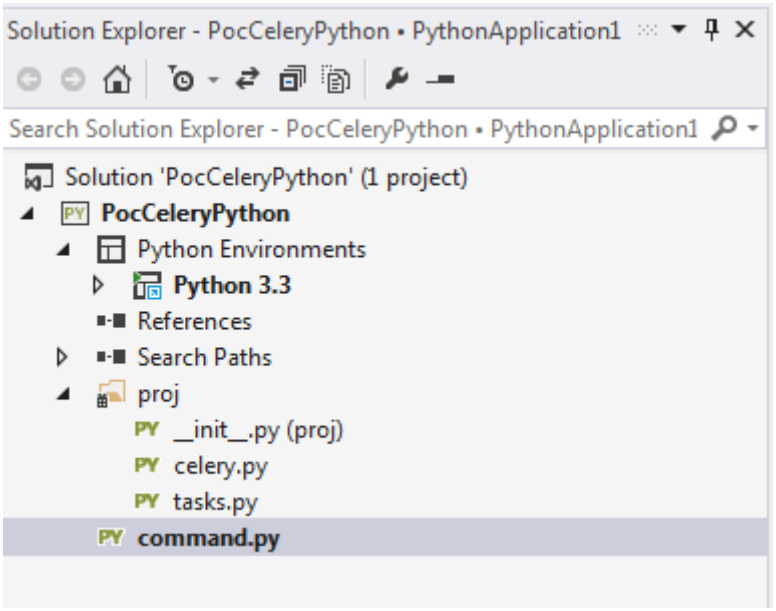
command.py

```python
from proj.tasks import add
from proj.tasks import mul

add.delay(2, 2)
add.delay(5, 3)
add.delay(10, 6)
add.delay(22, 8)

mul.delay(3,3);
```

Voici également l'architecture de mon projet au cas ou :



retour dans cygmwyn, on relance le celery worker server :

```
ddelallee@PC-W7-DEV69 /cygdrive/c/Users/ddelallee/Documents/Visual Studio 2013/Projects/PythonApplic
$ celery -A proj worker --loglevel=info
```

(Remplacer proj par le nom de votre package)

le celery worker server se relance comme précédemment.
retour dans visual studio, executer maintenant le fichier que vous avez créer
Comme précédemment, une console s'ouvre et reste vide, et si on regarde dans cygwin, on a bien les 5 taches qui se sont executées (une tache par fonction appelées)

```
ddelallee@PC-W7-DEV69 /cygdrive/c/Users/ddelallee/Documents/Visual Studio 2013/Projects/PythonApplic
$ celery -A proj worker --loglevel=info
[2015-03-27 12:47:31,343: WARNING/MainProcess] /usr/lib/python3.2/site-packages/celery/apps/worker.p
Starting from version 3.2 Celery will refuse to accept pickle by default.

The pickle serializer is a security concern as it may give attackers
the ability to execute any command.  It's important to secure
your broker from unauthorized access when using pickle, so we think
that enabling pickle should require a deliberate action and not be
the default choice.

If you depend on pickle then you should set a setting to disable this
warning and to be sure that everything will continue working
when you upgrade to Celery 3.2::

    CELERY_ACCEPT_CONTENT = ['pickle', 'json', 'msgpack', 'yaml']

You must only enable the serializers that you will actually use.


  warnings.warn(CDeprecationWarning(W_PICKLE_DEPRECATED))

 -------------- celery@PC-W7-DEV69 v3.1.17 (Cipater)
---- **** -----
--- * ***  * -- CYGWIN_NT-6.1-WOW64-1.7.33-2-0.280-5-3-i686-32bit-WindowsPE
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         proj:0xff91a5ec
- ** ---------- .> transport:   amqp://admin:**@bizperf45:5672//
- ** ---------- .> results:     amqp://
- *** --- * --- .> concurrency: 4 (prefork)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> celery           exchange=celery(direct) key=celery


[tasks]
  . proj.tasks.add
  . proj.tasks.mul

[2015-03-27 12:47:31,520: INFO/MainProcess] Connected to amqp://admin:**@bizperf45:5672//
[2015-03-27 12:47:31,535: INFO/MainProcess] mingle: searching for neighbors
[2015-03-27 12:47:32,575: INFO/MainProcess] mingle: all alone
[2015-03-27 12:47:32,634: WARNING/MainProcess] celery@PC-W7-DEV69 ready.
[2015-03-27 12:48:14,502: INFO/MainProcess] Received task: proj.tasks.add[f695d436-401f-49f4-9e34-51
[2015-03-27 12:48:14,503: INFO/MainProcess] Received task: proj.tasks.add[37fdfa42-2ed2-4244-97a1-08
[2015-03-27 12:48:14,503: INFO/MainProcess] Received task: proj.tasks.add[1aec508f-3de5-4c35-9abf-a6
[2015-03-27 12:48:14,503: INFO/MainProcess] Received task: proj.tasks.add[cf2f87f9-68b0-46cb-9073-6c
[2015-03-27 12:48:14,508: INFO/MainProcess] Received task: proj.tasks.mul[0710f02c-6ed3-4c3f-a365-64
[2015-03-27 12:48:14,745: INFO/MainProcess] Task proj.tasks.add[f695d436-401f-49f4-9e34-51ad5106585a
[2015-03-27 12:48:14,745: INFO/MainProcess] Task proj.tasks.add[cf2f87f9-68b0-46cb-9073-6c9b769b1778
[2015-03-27 12:48:14,745: INFO/MainProcess] Task proj.tasks.add[37fdfa42-2ed2-4244-97a1-088df4e6d96e
[2015-03-27 12:48:14,750: INFO/MainProcess] Task proj.tasks.add[1aec508f-3de5-4c35-9abf-a6ab8a6e69fd
[2015-03-27 12:48:14,762: INFO/MainProcess] Task proj.tasks.mul[0710f02c-6ed3-4c3f-a365-64e43e193737
```

Et voila le tour est joué vous venez de créer votre premier projet celery !!!

A vos claviers et en avant la rémoulade

Bonus : Appeller vos tâches

Il existe deux méthodes pour appeler vos tâches. Celle vue précedement : **delay**, et **apply_async**.

delay est en faite une sorte de raccourcis pour **apply_async** qui peut prendre plus de paramètres et permet une plus grande gestion de l'appel des tâches.
  on peut notamment retarder le lancement de la tâche !!!

Prenons par exemple notre méthode add qui prends deux arguements en entrée.
avec **delay** on l'appelle de cette manière : add.delay(2,2)

avec **apply_async** on fera comme suit : add.apply_async((2,2))

Autres arguments pour apply_async : add.apply_async((2,2), queue="nomVoulu", countdown = 10, eta=DateTime)

Queue="..." permet de dire que vous voulez que cette tâche soit envoyé dans la queue "nomVoulu"
countdown permet de définir le nombre de secondes a attendre avant de lancer l'execution de la tâche.
eta = permet de préciser une date a partir de laquelle on veut envoyer la tache
Si on précise un countdown, on a alors l'affichage suivant :

```
[2015-03-27 17:06:40,738: INFO/MainProcess] Received task: proj.tasks.add[76904240-8035-45b6-88c0-f2
[2015-03-27 17:06:40,741: INFO/MainProcess] Received task: proj.tasks.add[18074f76-95cd-474a-b1e4-c5
[2015-03-27 17:06:40,746: INFO/MainProcess] Received task: proj.tasks.add[b084d8ce-a73f-4d80-b928-a8
[2015-03-27 17:06:40,750: INFO/MainProcess] Received task: proj.tasks.add[fbd3ecd1-3c75-4f25-878f-33
[2015-03-27 17:06:40,755: INFO/MainProcess] Received task: proj.tasks.mul[8eed08e4-96b0-48ee-8e9a-a5
[2015-03-27 17:06:40,773: INFO/MainProcess] Task proj.tasks.add[76904240-8035-45b6-88c0-f2eecd0f6eb2
[2015-03-27 17:06:40,825: INFO/MainProcess] Task proj.tasks.add[18074f76-95cd-474a-b1e4-c55572f087ad
[2015-03-27 17:06:40,849: INFO/MainProcess] Task proj.tasks.add[b084d8ce-a73f-4d80-b928-a8f6a7ad3ed8
[2015-03-27 17:06:40,852: INFO/MainProcess] Task proj.tasks.mul[8eed08e4-96b0-48ee-8e9a-a5294ee9287f
[2015-03-27 17:06:52,565: INFO/MainProcess] Task proj.tasks.add[fbd3ecd1-3c75-4f25-878f-3357318ef219
```

On voit que pour la task ou on a mis le countdown l'information "**eta**" apparait. L'heure stockée dans cet objet est l'heure à laquelle la tâche sera envoyée.

Utilisation d'APIs pour lancer des tâches

Première chose, pour que Celery puisse identifier les tâches provenant d'APIs (appelées par des routes) il faut lancer le celery worker server avec les parametres suivants :

celery -A nomPackage worker -I celery.task.http --loglevel=info

Dans Python, j'ai crée une fonction qui appelle via la librairie request l'API souhaitée.
Dans le fichier tasks.py:

```
22
23  @app.task
24  def callAPI(url, data):
25      dictionary = {"key":"KPYN......"}
26      rep = requests.post(url, headers=dictionary);
27      #response = requests.get(url,auth=('KPYN0KL.......'))
28      return rep;
```

et dans le fichier command.py :

```
5   from proj.tasks import callAPI
6   from celery import Celery
7   from celery.task.http import URL
8
9   res = callAPI.delay('http://backoffice.mailperformance.test/targets/tests', '');
10
```

On doit bien evidemment relancer le celery worker server par la suite. SI tout se passe bien on obtient deux lignes supplémentaires dans le code.
Une pour informer que la connexion a bien eut lieu avec le client Http, et une pour donner la réponse de la requête.

```
 -------------- celery@PC-W7-DEV69 v3.1.17 (Cipater)
---- **** -----
--- * ***  * -- CYGWIN_NT-6.1-WOW64-1.7.33-2-0.280-5-3-i686-32bit-WindowsPE
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         proj:0xff91a64c
- ** ---------- .> transport:   amqp://admin:**@bizperf45:5672//
- ** ---------- .> results:     amqp://
- *** --- * --- .> concurrency: 4 (prefork)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> celery           exchange=celery(direct) key=celery


[tasks]
  . proj.tasks.add
  . proj.tasks.callAPI
  . proj.tasks.mul

[2015-03-31 18:40:40,009: INFO/MainProcess] Connected to amqp://admin:**@bizperf45:5672//
[2015-03-31 18:40:40,026: INFO/MainProcess] mingle: searching for neighbors
[2015-03-31 18:40:41,053: INFO/MainProcess] mingle: all alone
[2015-03-31 18:40:41,132: WARNING/MainProcess] celery@PC-W7-DEV69 ready.
[2015-03-31 18:40:53,678: INFO/MainProcess] Received task: proj.tasks.add[8079a1a9-4207-4a34-a580-6c
[2015-03-31 18:40:53,680: INFO/MainProcess] Received task: proj.tasks.add[c914e52f-9169-482e-a8ef-44
[2015-03-31 18:40:53,684: INFO/MainProcess] Received task: proj.tasks.add[43f87abd-c352-415a-a0ef-84
[2015-03-31 18:40:53,688: INFO/MainProcess] Received task: proj.tasks.add[8ce92738-ce58-4abf-b406-36
[2015-03-31 18:40:53,693: INFO/MainProcess] Received task: proj.tasks.callAPI[173f1858-6928-4ee9-995
[2015-03-31 18:40:53,698: INFO/MainProcess] Received task: proj.tasks.mul[b226091b-1591-4e9a-b1c4-28
[2015-03-31 18:40:53,713: INFO/Worker-2] Starting new HTTP connection (1): backoffice.mailperformanc
[2015-03-31 18:40:53,768: INFO/MainProcess] Task proj.tasks.add[8079a1a9-4207-4a34-a580-6c6edf77e635
[2015-03-31 18:40:53,768: INFO/MainProcess] Task proj.tasks.add[c914e52f-9169-482e-a8ef-442ecc23be1b
[2015-03-31 18:40:53,777: INFO/MainProcess] Task proj.tasks.add[43f87abd-c352-415a-a0ef-8497f09c099c
[2015-03-31 18:40:53,791: INFO/MainProcess] Task proj.tasks.mul[b226091b-1591-4e9a-b1c4-285eeca00c05
[2015-03-31 18:40:54,711: INFO/MainProcess] Task proj.tasks.callAPI[173f1858-6928-4ee9-9951-c82f500c
```

Les contraintes rencontrées jusqu'à maintenant

L'API appelée doit renvoyée des informations formatée de manière précise pour que Celery puisse donner des informations sur l'évolution de la tâche.
Si les données renvoyées sont mal formatées, on ne peut pas savoir ce qui se passe.
Il va donc falloir encapsuler les APIs.

Exemple de formattage requis :

```
response = new HttpResponseMessage
{
    StatusCode = System.Net.HttpStatusCode.OK,
    Content = new StringContent("{'status':'success','retval':'fp@np6.com a ete spammé'}")
};
```

return response;

ou encore

```
response = new HttpResponseMessage
{
    StatusCode = System.Net.HttpStatusCode.BadRequest,
    Content = new StringContent("{'status':'failure','reason':'mon chef est nul nul nul'}")
};
```