Python 3 : Formation 4

Python 3 Formation 4 2
1 - Micro service 3
Mise en place du micro service 3
Description des données 3
Ajout de la route 4
Vérifier les emails 5
Templating des emails 6
Envoi de email 7
Installation 7
Configuration 8
Usage 9
Test 10
2 - Loggers 11
Python et les loggers 11
Niveaux de logs 11
Logging simple dans un fichier 12
Logging avancée 13
Flask et le logging 15
3 - Celery 17
Concept 17
Installation 18
Base de données 18
Python 18
Usage 18
Création de l'application 18
Lancement du worker 19
Lancement d'une tâche 19
En dehors du contexte d'exécution de notre programme 20
Plus de détail sur les taches 21
Celery et Flask 22
Mise en place 22
Utilisation de la tache 24

Lancement 26
4 - Docker 26
Créer un container de notre application Flask 26
Variables d'environnement 27
Serveur HTTP de production 27
Dockerfile 28
Packager notre application et son environnement 29
5 - Nginx 30
Mise en place 31
Fichiers de configuration 31
docker-compose 33
6 - Création de bibliothèques 34
Installation des outils 34
`setup.py` 34
Installation de la bibliothèque 35
Upload sur PIP 35
7 - Pip personnalisé 36
Serveur docker 36
Préparation 36
Lancement 37
Configuration 37

Python 3 Formation 4

Elonet

1 - Micro service

Dans la précédente formation, nous avons vu comment construire un service REST à l'aide de Flask et de l'extension Flask RestPlus, nous allons maintenant mettre ça en pratique à travers un micro-service d'envoi d'emails.

L'objectif est de pouvoir envoyer un email à un grand nombre de personnes tout en ayant un service qui répond très rapidement aux requêtes.

Mise en place du micro service

Afin de ne pas trop revenir sur le chapitre précédent, je vous fournis une template de micro-service Flask RestPlus que vous pourrez réutiliser dans vos projets.

Copier le projet : https://github.com/averdier/restplus_template et renommer le dossier en `email_service`

```
git clone https://github.com/averdier/restplus_template
mv restplus_template/ email_service
cd email_service
virtualenv -p python3 env
source env/bin/activate
pip install -r requirements.txt
```

Vous pouvez dès maintenant lancer la solution :

```
python runserver.py
```

Description des données

La première chose à faire est de décrire les données attendues par le micro service.
Pour envoyer des emails, il nous faut au moins les informations suivantes :

- Une liste de destinataires
- Un titre
- Un corps
  Nous allons utiliser un système de templates de email, nous pouvons donc rajouter :
- Nom de la template
  Ajoutons un fichier `email.py` dans le dossier `email_service/app/api/serializers`

```
# -*- coding: utf-8 -*-
from flask_restplus import fields
from .. import api

send_email_model = api.model('Send Email model', {
    'recipients': fields.List(fields.String(), required=True,
description='Recipients list'),
    'subject': fields.String(required=True, description='Email subject'),
    'content': fields.String(required=True, description='Email content'),
```

```
    'template': fields.String(required=True, description='Email template')
})
```

Nous pourrions rajouter des règles de validation par exemple un sujet compris en 3 et 32 caractères, à ce niveau la vous êtes libres de faire comme bon vous semble.

Exemple :

```
'subject': fields.String(required=True, min_length=3, max_length=32,
description='Email subject')
```

Ajout de la route

La prochaine étape et d'ajouter la route qui va permettre l'envoi de email, cette route comportera au moins la méthode POST qui permettra d'envoyer les données permettant d'envoyer les emails.

Ajoutons un fichier `email.py` dans le dossier `email_service/app/api/endpoints`

```
# -*- coding: utf-8 -*-
from flask import request
from flask_restplus import Namespace, Resource
from ..serializers.email import send_email_model

ns = Namespace('email', description='Email related operation')

# ==============================================================================
# ENDPOINTS
# ==============================================================================
#   API email endpoints
#
# ==============================================================================

@ns.route('/')
class EmailSend(Resource):
    @ns.expect(send_email_model)
    def post(self):
        """
        Send email
        """
        data = request.json
```

Ici nous avons défini une route `/api/email` qui attend les données au format `send_email_model` en POST.

Nous récupérons l'ensemble des données disponibles à l'aide de `request.json`

Pensez à modifier le fichier `email_service/app/api/__init__.py` afin de prendre en compte la route

Vérifier les emails

Afin de limiter les erreurs lors de l'envoi de email, nous allons vérifier que les adresses email fournies sont bien des adresses email.

Pour cela nous allons utiliser les regex.

Python possède déjà la bibliothèque pour traiter les regex, il suffit de faire un import :

```
import re
```

L'usage de la bibliothèque est assez simple, il suffit d'utiliser la fonction `match` qui attend en paramètre l'expression régulière et la valeur à tester.

En retour la fonction retourne `True` si la valeur correspond à l'expression, `False` sinon.

```
re.match(r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$)", email)
```

Dans le cas ou un email ne passe pas l'expression régulière, nous allons retourner un code d'erreur 400 à l'aide de la fonction `abort`

Ce qui nous donne :

```
# -*- coding: utf-8 -*-
import re
from flask import request
from flask_restplus import Namespace, Resource,abort
from ..serializers.email import send_email_model

ns = Namespace('email', description='Email related operation')

# ====================================================================
# ENDPOINTS
# ====================================================================
#   API email endpoints
#
# ====================================================================

@ns.route('/')
class EmailSend(Resource):
    @ns.expect(send_email_model)
    def post(self):
        """
        Send email
        """
        data = request.json
        for email in data['recipients']:
            if not re.match(r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-
.]+$)", email):
                abort(400, error='{0} not pass email regex.'.format(email))
```

Templating des emails

Nous allons mettre en place le système de template de email à l'aide du moteur de template JinJa2.

Lors de la précédente formation, nous avons vu qu'il était possible d'utiliser JinJa2 pour faire des templates de page HTML, mais il est tout à fait possible de l'utiliser à d'autres fins.

Créons un dossier `templates` dans le dossier `email_service/app` et ajoutons un fichier `default.eml` à l'intérieur

```
Date: {{date}}
From: {{from}}
To: {{to}}
Subject: {{subject}}
Content-Type: text/plain

{{ content }}

--
Email service
```

Nous pouvons maintenant rendre notre template à l'aide de l'objet `Template` de JinJa2 :

```
# -*- coding: utf-8 -*-
from datetime import datetime
from email.utils import format_datetime
from jinja2 import Template

if __name__ == '__main__':
    data = {
        'date': format_datetime(datetime.utcnow()),
        'to': 'will@elonet.fr',
        'from': 'eleven@elonet.fr',
        'subject': 'Demogorgon',
        'content': 'Run for your life'
    }
    with open('<file path>') as template_file:
        template = Template(template_file.read())
        print(template.render(data))
```

Ce qui nous donne le résultat suivant :

```
Date: Thu, 14 Jun 2018 15:03:07 -0000
From: eleven@elonet.fr
To : will@elonet.fr
Subject: Demogorgon
Content-Type: text/plain

Run for your life
```

```
--
Email service
```

Afin de rendre la logique modulable et réutilisable, créons un fichier `utils.py` dans le dossier `email_service/app`, ce fichier contiendra la fonction `render_email` qui prendra en paramètre le nom de la template et un dictionnaire de données :

```
import os
from jinja2 import Template

def render_email(template_name, data):
    """
    Render email template

    :param template_name: Name of email template
    :type template_name: str

    :param data: Data of email
    :type data: dict

    :return: Rendered email
    :rtype: str
    """
    basedir = os.path.abspath(os.path.dirname(__file__))

    with open(basedir + '/templates/' + template_name + '.eml') as
template_file:
        template = Template(template_file.read())

        return template.render(data)
```

Nous pouvons maintenant générer des emails, il nous reste a les envoyer

Envoi de email

Pour simplifier l'envoi de emails, nous allons utiliser l'extension Flask-Mail (https://pythonhosted.org/Flask-Mail/)

Installation

Pour l'installation nous allons utiliser notre fidèle `pip`

```
pip install flask-mail
```

Nous allons ensuite ajouter l'extension à notre application, créons un fichier `extensions.py` dans le dossier `email_service/app` :

```
# -*- coding: utf-8 -*-
from flask_mail import Mail
```

```
mail = Mail()
```

Il faut ensuite modifier le fichier `__init__.py` présent dans le dossier `email_service/app` afin de bien ajouter notre extension :

```
from .extensions import mail
....
def extensions(flask_app):
    """
    Init extensions
    :param flask_app:
    """
    mail.init_app(flask_app)
```

Configuration

Flask-Mail nécessite une configuration à fournir dans la configuration de notre application. Extrait de la documentation :

Champ	Description
MAIL_SERVER	default `localhost`
MAIL_PORT	default `25`
MAIL_USE_TLS	default `False`
MAIL_USE_SLL	default `False`
MAIL_DEBUG	default `app.debug`
MAIL_USERNAME	default `None`
MAIL_PASSWORD	default `None`
MAIL_DEFAULT_SENDER	default `None`
MAIL_MAX_EMAILS	default `None`
MAIL_SUPPRESS_SEND	default `app.testing`
MAIL_ASCII_ATTACHEMENTS	default `False`

Il n'est pas nécessaire de renseigner l'ensemble des champs dans la mesure ou chaque champ possède une valeur par défaut.

Modifions notre fichier `config.py` présent dans le dossier `email_service` afin d'ajouter la configuration nécessaire à l'envoi de emails.

```
class Config:
    """
    Base configuration
    """
    MAIL_SERVER = ''
    MAIL_PORT = ''
    MAIL_USE_TLS = False
    MAIL_USE_SSL = True
```

```
    MAIL_USERNAME = ''
    MAIL_PASSWORD = ''
    MAIL_DEFAULT_SENDER = ''
    @staticmethod
    def init_app(app):
        """
        Init app
        :param app: Flask App
        :type app: Flask
        """
        pass
```

Il ne nous restera plus qu'à fournir les informations nécessaires.

Usage

À présent, voyons comment envoyer nos emails, pour cela nous allons modifier le fichier `email.py` dans le dossier `email_service/app/api/endpoints`.

Flask-Mail permet déjà d'envoyer un email à une grande quantité de personnes, le **Bulk email**, le code est fourni dans la documentation :

```
with mail.connect() as conn:
    for user in users:
        message = '...'
        subject = "hello, %s" % user.name
        msg = Message(recipients=[user.email],
                      body=message,
                      subject=subject)
        conn.send(msg)
```

Dans notre template de email, nous avons mis un paramètre `from`, pour le remplir nous allons utiliser la propriété `MAIL_DEFAULT_SENDER` présente dans notre configuration.

Pour cela nous allons utiliser la variable `current_app` présente dans `Flask`, grâce à cette variable nous avons accès à la configuration de notre application :

```
mail_sender = current_app.config['MAIL_DEFAULT_SENDER']
```

Maintenant le tout ensemble :

```
# -*- coding: utf-8 -*-
import re
from flask import request, current_app
from flask_restplus import Namespace, Resource, abort
from flask_mail import Message
from app.extensions import mail
from app.utils import render_email
from ..serializers.email import send_email_model

ns = Namespace('email', description='Email related operation')
```

```
# ============================================================================
# ENDPOINTS
# ============================================================================
#   API email endpoints
#
# ============================================================================

@ns.route('/')
class EmailSend(Resource):
    @ns.expect(send_email_model)
    def post(self):
        """
        Send email
        """
        data = request.json
        for email in data['recipients']:
            if not re.match(r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-
.]+$)", email):
                current_app.logger.error('{0} not pass email
regex.'.format(email))
                abort(400, error='{0} not pass email regex.'.format(email))
        with mail.connect() as conn:
            for email in data['recipients']:
                payload = {
                    'from': current_app.config['MAIL_DEFAULT_SENDER'],
                    'to': email,
                    'subject': data['subject'],
                    'content': data['content']
                }
                msg = Message(
                    recipients=[email],
                    body=render_email(data['template'], payload),
                    subject=data['subject']
                )
                conn.send(msg)
```

Test

Afin de faciliter les tests, nous allons utiliser un petit utilitaire : `mailcatcher` qui nous permet d'avoir un serveur d'envoie de email de développement

```
docker run -d -p 1080:80 -p 1025:25 --name smtp_dev tophfr/mailcatcher
```

Ajoutons la configuration de `mailcatcher` dans notre fichier `email_service/config.py` :

```
class Config:
    """
    Base configuration
    """
    MAIL_SERVER = 'localhost'
```

```
    MAIL_PORT = 1025
    MAIL_USE_TLS = False
    MAIL_USE_SSL = False
    MAIL_USERNAME = ''
    MAIL_PASSWORD = ''
    MAIL_DEFAULT_SENDER = 'will@dev.fr'
    @staticmethod
    def init_app(app):
        """
        Init app
        :param app: Flask App
        :type app: Flask
        """
        pass
```

Lançons et essayons d'envoyer 1 email puis 10, et regardons le temps de réponse du service, on se rend bien compte qu'assez vite le temps de réponse n'est plus viable surtout dans la mesure ou le service sera appelé par d'autres services

Pour palier à ça nous allons déporter l'envoi de email dans un processus différent, même si l'envoi des emails dure 1h, le service répondra et moins de 2 secondes.

Mais avant ça, voyons comment ajouter des logs à notre service.

2 - Loggers

Garder des traces (logs) des choses qui fonctionnent ou qui ne fonctionnent pas fait partie des bonnes pratiques lors du développement d'un programme, la manière la plus courante est d'écrire les logs dans un ou plusieurs fichiers.

Python et les loggers

Python possède déjà les bibliothèques nécessaires au logging, il suffit d'importer le paquet `logging`

Exemple :

```
import logging

if __name__ == '__main__':
    logging.warning('Mon message')
```

Sortie :

```
WARNING:root:Mon message
```

Niveaux de logs

Il existe plusieurs niveaux de logs, chaque niveau correspond à un type de message

Nom	Description
debug	A utiliser pour les diagnostics et le développement
info	A utiliser quand l'exécution d'une étape importante est un succès
warning	A utiliser lors d'une erreur non critique, exemple un serveur ne répond pas mais le programme peut continuer à fonctionner normalement
error	A utiliser lors d'une erreur empêchant le fonctionnement normal du programme, mais qui ne nécessite pas de quitter le programme
critical	A utiliser lors d'une erreur nécessitant de quitter le programme

Logging simple dans un fichier

La bibliothèque logging est très bien faite, il est aisé d'écrire les logs dans un fichier

Exemple :

```
import os
import logging

if __name__ == '__main__':
    basedir = os.path.abspath(os.path.dirname(__file__))
    filename = os.path.join(basedir, 'my_logs.logs')
    logging.basicConfig(filename=filename, level=logging.DEBUG)
    logging.warning('Mon message')
```

Ce qui nous donne un fichier `my_logs.logs` contenant :

```
WARNING:root:Mon message
```

De plus il est possible d'utiliser cette méthode même si notre programme comporte plusieurs fichiers et ce sans avoir à redéfinir le logger et le fichier à écrire.

Exemple :

```
# main.py
import os
import logging
import hello
if __name__ == '__main__':
    basedir = os.path.abspath(os.path.dirname(__file__))
    filename = os.path.join(basedir, 'my_logs.logs')
    logging.basicConfig(filename=filename, level=logging.DEBUG)
    logging.warning('Mon message')
    hello.say_hello('Will')
    logging.warning('Finis')

# hello.py
import logging

def say_hello(name):
```

```
    logging.info('Say hello to {0}'.format(name))
    print('Hello {0}'.format(name))
```

Ce qui nous donne :

```
WARNING:root:Mon message
INFO:root:Say hello to Will
WARNING:root:Finis
```

Logging avancée

Nous avons vu la méthode la plus simple pour l'écriture de logs dans un fichier, dans le cas d'utilisation de bibliothèques (Flask par exemple) il faut découper le logging en plusieurs parties
La bibliothèque `logging` nous met à disposition plusieurs objets et fonctions pour réaliser le découpage :

- La fonction `getLogger` qui permet d'avoir une interface de logger utilisable dans le code
- Les objets `Handler` qui ont la charge d'envoyer les logs à la bonne destination (un fichier par exemple)
- Les objets `Formatter` qui déterminent la manière dont les logs sont rendus

Exemple :

```
import os
import logging
from logging.handlers import RotatingFileHandler

if __name__ == '__main__':
    basedir = os.path.abspath(os.path.dirname(__file__))
    filename = os.path.join(basedir, 'my_logs.log')
    # Interface
    logger = logging.getLogger(__name__)
    # Handler
    handler = RotatingFileHandler(filename, maxBytes=20000, backupCount=10,
encoding='utf-8')
    # Formatter
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s:
%(message)s')
    # Ajout du formatter au handler
    handler.setFormatter(formatter)
    # Ajout du handler au logger
    logger.addHandler(handler)
    # Definition du niveau de log
    handler.setLevel(logging.DEBUG)
    logger.warning('Mon message')
    logger.warning('Finis')
```

Ce qui nous donne :

```
2018-06-20 11:45:57,627 - __main__ - WARNING: Mon message
2018-06-20 11:45:57,627 - __main__ - WARNING: Finis
```

Explication

Logger

`logging.getLogger(__name__)` permet de récupérer une interface de logger pour nos logs, pourquoi le `__name__` ?

Il est possible de donner un nom au logger, mais celui-ci doit être unique, car il représente une référence vers le logger.

Autrement dit, si vous créez deux logger avec le même nom, vous aurez en fait 2 variables pointant vers le même logger.

Exemple :

```
...
if __name__ == '__main__':
    ...
    # Interface
    logger1 = logging.getLogger('mon_logger')
    logger2 = logging.getLogger('mon_logger')
    ...
    logger1.warning('Mon message')
    logger1.warning('Finis')
    logger2.warning('Mon message depuis logger 2')
    logger2.warning('Finis depuis logger 2')
```

Ce qui nous donne :

```
2018-06-20 11:49:51,848 - mon_logger - WARNING: Mon message
2018-06-20 11:49:51,850 - mon_logger - WARNING: Finis
2018-06-20 11:49:51,850 - mon_logger - WARNING: Mon message depuis logger 2
2018-06-20 11:49:51,852 - mon_logger - WARNING: Finis depuis logger 2
```

Nous voyons bien que malgré des noms de variables différents, il s'agit en fait d'un même logger

Handler

Le `RotatingFileHandler` est le handler de fichier le plus pratique, car il permet d'écrire les logs dans des fichiers rotatifs.

C'est-à-dire que quand un fichier de log est considéré comme plein, le handler crée automatiquement un fichier `<filename>.log.1`

Le `RotatingFileHandler` nécessite 4 paramètres :

- Le chemin complet du fichier ou écrire les logs
- La taille maximale d'un fichier de log

- Le nombre maximal de fichiers de log
- L'encodage des fichiers de log

Formatter

Dans le formatter, il suffit de fournir un format de log, ici on affiche `Date et heure - Nom du logger - Niveau du log : Message`

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s: %(message)s')
```

Flask et le logging

Flask possède déjà un logger, il nous suffit de l'utiliser, reprenons notre fichier `email.py` dans le dossier `email_service/app/api/endpoints`.

Nous allons ajouter un log d'erreur quand une adresse email n'est pas valide, pour accéder au logger il faut utiliser `current_app.logger`
`current_app` est une référence de l'application Flask en cours d'utilisation

Exemple :

```
# -*- coding: utf-8 -*-
import re
from flask import request, current_app  # Changement ici
from flask_restplus import Namespace, Resource,abort
from ..serializers.email import send_email_model

ns = Namespace('email', description='Email related operation')

# ================================================================================
# ENDPOINTS
# ================================================================================
#   API email endpoints
#
# ================================================================================

@ns.route('/')
class EmailSend(Resource):
    @ns.expect(send_email_model)
    def post(self):
        """
        Send email
        """
        data = request.json
        for email in data['recipients']:
            if not re.match(r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$)", email):
                current_app.logger.error('{0} not pass email
```

```
regex.'.format(email)) # Changement ici
                abort(400, error='{0} not pass email regex.'.format(email))
```

Ce qui nous donne par exemple :

```
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 629-244-677
 * Running on http://localhost:5555/ (Press CTRL+C to quit)
[2018-06-20 18:04:30,986] ERROR in email: string not pass email regex.
127.0.0.1 - - [20/Jun/2018 18:04:30] "POST /api/email/ HTTP/1.1" 400 -
```

Pour pouvoir écrire les logs dans un fichier il suffit d'ajouter un handler au logger, pour cela nous allons reprendre le fichier `config.py` et y faire 2 modifications :

- Quelques ajouts dans la configuration afin de rendre les logs paramétrables.
- Surcharger la méthode `init_app` pour ajouter notre handler
  En configuration de développement, nous écrirons les logs du service dans des fichiers.

```
# -*- coding: utf-8 -*-
import logging
from logging.handlers import RotatingFileHandler

class Config:
    """
    Base configuration
    """
    MAIL_SERVER = 'localhost'
    MAIL_PORT = 1025
    MAIL_USE_TLS = False
    MAIL_USE_SSL = False
    MAIL_USERNAME = ''
    MAIL_PASSWORD = ''
    MAIL_DEFAULT_SENDER = 'will@dev.fr'
    LOG_PATH = '<your path>'
    LOG_SIZE = 20000
    LOG_COUNT = 10
    LOG_ENCODING = 'utf-8'
    LOG_LEVEL = 'DEBUG'
    @staticmethod
    def init_app(app):
        """
        Init app
        :param app: Flask App
        :type app: Flask
        """
```

```
        pass

class DevelopmentConfig(Config):
    """
    Development configuration
    """
    @staticmethod
    def init_app(app):
        Config.init_app(app)
        handler = RotatingFileHandler(app.config['LOG_PATH'],
                                      maxBytes=app.config['LOG_SIZE'],
                                      backupCount=app.config['LOG_COUNT'],
                                      encoding=app.config['LOG_ENCODING'])
        formatter = logging.Formatter(
            '%(asctime)s %(levelname)s: %(message)s '
            '[in %(pathname)s:%(lineno)d]'
        )
        handler.setFormatter(formatter)
        handler.setLevel(getattr(logging, app.config['LOG_LEVEL'].upper()))
        app.logger.addHandler(handler)
```

3 - Celery

Dans certains cas, nous avons besoin de réaliser des actions en dehors du contexte d'exécution de notre programme par exemple :

- Dans le cas d'une interface graphique où la boucle d'exécution doit être aussi rapide que possible afin de ne pas détériorer l'expérience utilisateur
- Dans le cas d'actions programmées dans le temps (exemple : exécuter une action dans 1h)
- Dans le cas d'un micro-service devant réaliser des actions prenant du temps tout en répondant en moins de 1 seconde.
  Il est en général possible de s'en sortir avec les processus et les threads, mais cela devient très vite très complexe à mettre en oeuvre.
  Une solution est d'utiliser `Celery`, une bibliothèque offrant un gestionnaire de tâches et une file d'attente.

Concept

Une application utilisant `Celery` est composée d'au moins 4 éléments :

- L'application en elle même
- L'application Celery, elle a la charge de réaliser nos tâches
- Un `worker` qui a la charge d'exécuter l'application Celery
- Une base de données permettant de stocker la file d'attente et les résultats des taches

Installation

Base de données

L'application Celery nécessite une base de données pour fonctionner, nous allons ici utiliser Redis car elle est facilement utilisable en Python pour d'autres projets (la liste des bases de données possibles est disponible dans la documentation).

Afin de faciliter l'installation et l'utilisation, nous allons utiliser Docker pour lancer notre base de données Redis.

Premier lancement de Redis

```
docker run --name redis_dev -p 6379:6379 -d redis
```

-p 6379:6379 permet de rendre Redis disponible sur notre réseau local, pratique dans les phases de développement, à éviter dans les phases de production.

Arréter Redis

```
docker stop redis_dev
```

Relancer Redis

```
docker start redis_dev
```

Python

Pour installer `Celery` il suffit d'utiliser `pip`

Il faut aussi installer le connecteur `Redis`

```
pip install celery
pip install redis
```

Usage

Création de l'application

Créons une application `Celery` très simple dans un fichier `tasks.py`, elle contiendra uniquement une tache `do_stuff` qui prend 30 secondes

```
import time
from celery import Celery

app = Celery('tasks', broker='redis://localhost',
backend='redis://localhost/0')

@app.task
def do_stuff():
    time.sleep(30)
    return 'SUCCESS'
```

La création d'une application nécessite au moins 3 choses :

- Un nom, ici `tasks`
- Un broker pour stocker la file d'attente.
- Un backend pour stocker les états et les résultats
  Le décorateur `app.task` permet d'ajouter la fonction `do_stuff` à l'application Celey

Lancement du worker

Il faut ensuite lancer le worker qui exécutera notre application Celery.
Dans un terminal avec l'environnement Python adéquat :

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

Sous Windows il est nécessaire d'utiliser une bibliothèque supplémentaire afin de pouvoir lancer le worker

```
pip install eventlet
celery -A tasks worker -P eventlet --loglevel=info
```

Ce qui nous donne la sortie suivante qui indique que le système est en attente de tâches :

```
 -------------- celery@DESKTOP-DEV v4.2.0 (windowlicker)
---- **** -----
--- * ***  * -- Windows-10 2018-06-24 13:07:47
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         tasks:0x2657e147a90
- ** ---------- .> transport:   redis://localhost:6379//
- ** ---------- .> results:     disabled://
- *** --- * --- .> concurrency: 8 (eventlet)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this
worker)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=celery(direct) key=celery

[tasks]
  . tasks.do_stuff
[2018-06-24 13:07:47,058: INFO/MainProcess] Connected to
redis://localhost:6379//
[2018-06-24 13:07:47,073: INFO/MainProcess] mingle: searching for neighbors
[2018-06-24 13:07:48,144: INFO/MainProcess] mingle: all alone
[2018-06-24 13:07:48,211: INFO/MainProcess] celery@DESKTOP-DEV ready.
[2018-06-24 13:07:48,213: INFO/MainProcess] pidbox: Connected to
redis://localhost:6379//.
```

Lancement d'une tâche

Il suffit ensuite de lancer nos tâches.
Dans un interpréteur Python adéquat et lancé dans le dossier contenant `tasks.py`:

```
>>> from tasks import do_stuff
>>> result = do_stuff.delay()
>>> result.ready()
False
>>> # 30s plus tard
...
>>> result.ready()
True
>>> result.get()
'SUCCESS'
>>>
```

Il existe 2 moyens de lancer une tâche, `delay` et `apply_async`, `delay` est une version simplifiée de `apply_async`, (en interne `delay` appel `apply_async`).

Nous pouvons savoir si une tâche est terminée à l'aide de `ready` et récupérer le résultat d'une tâche à l'aide de `get`.

Si nous regardons la sortie de notre worker nous pouvons voir que la tâche est bien un succès et que le résultat de la tâche est `SUCCESS`

```
[2018-06-24 13:20:33,909: INFO/MainProcess] Received task:
tasks.do_stuff[8a9049b8-f33a-4747-b0f6-cb4823b7aace]
[2018-06-24 13:21:03,929: INFO/MainProcess] Task tasks.do_stuff[8a9049b8-
f33a-4747-b0f6-cb4823b7aace] succeeded in 30.030999999959022s: 'SUCCESS'
```

En dehors du contexte d'exécution de notre programme

Il est important de souligner que le worker est une entité séparée et qu'il est possible de l'utiliser avec un ou plusieurs programmes.

Pour l'illustrer, ouvrez 2 interpréteurs Python, dans le premier nous allons lancer une tâche et récupérer son identifiant unique et dans le second nous allons lire l'état de la tâche à partir de son identifiant
Nous pouvons récupérer l'état d'une tâche à l'aide de `<task>.AsyncResult(<task_id>)`

```
>>> from tasks import do_stuff
>>> task = do_stuff.delay()
>>> task.id
'85de2fde-703d-47ff-9cb9-4ab64c8a76da'
>>>
```

```
>>> from tasks import do_stuff
>>> task = do_stuff.AsyncResult('85de2fde-703d-47ff-9cb9-4ab64c8a76da')
>>> task
<AsyncResult: 85de2fde-703d-47ff-9cb9-4ab64c8a76da>
>>> task.ready()
False
>>> task.ready()
True
>>> task.get()
```

```
'SUCCESS'
>>>
```

Ce qui implique que le worker peut continuer à effectuer des tâches alors que le programme qui les a lancées n'est plus en cours d'exécution

Plus de détail sur les taches

Jusqu'à maintenant nous avons vu comment lancer une tâche et récupérer un résultat quand celle-ci est terminée.

Lors de tâches complexes, il est souvent important de connaitre l'état d'avancement ou l'étape en cours.

Pour réaliser cela il y a 3 choses à faire :

- Ajouter `bind=True` à notre décorateur, ce qui permettra d'accéder à l'instance de la tâche dans la fonction
- Ajouter le paramètre `self` en premier paramètre de la fonction
- Mettre à jour l'instance de la classe aux étapes importantes à l'aide de `self.update_state`
  Il est ensuite possible d'accéder aux informations de la tâche à l'aide de `<task>.info`

Exemple

```
import time
from celery import Celery

app = Celery('tasks', broker='redis://localhost',
backend='redis://localhost/0')

@app.task(bind=True)
def do_stuff(self):
    state = 'PROGRESS'
    meta = {
        'total': 3,
        'current': 0,
        'message': 'Loading file',
        'result': False
    }
    self.update_state(state=state, meta=meta)
    time.sleep(10)
    meta['current'] += 1
    meta['message'] = 'Parse file'
    self.update_state(state=state, meta=meta)
    time.sleep(10)
    meta['current'] += 1
    meta['message'] = 'Do some stuff'
    self.update_state(state=state, meta=meta)
    time.sleep(10)
```

```
    meta['current'] += 1
    meta['message'] = 'Stuff completed'
    meta['result'] = True
    return meta
```

Dans un interpréteur Python adéquat :

```
>>> import time
>>> from tasks import do_stuff
>>>
>>> def show_task():
...  task = do_stuff.delay()
...  while not task.ready():
...   print(task.info)
...   time.sleep(5)
...  print(task.get())
...
>>> show_task()
None
{'total': 3, 'current': 0, 'message': 'Loading file', 'result': False}
{'total': 3, 'current': 1, 'message': 'Parse file', 'result': False}
{'total': 3, 'current': 1, 'message': 'Parse file', 'result': False}
{'total': 3, 'current': 2, 'message': 'Do some stuff', 'result': False}
{'total': 3, 'current': 2, 'message': 'Do some stuff', 'result': False}
{'total': 3, 'current': 3, 'message': 'Stuff completed', 'result': True}
>>>
```

Celery et Flask

Celery est très intéressant avec Flask, car cela permet de créer un service qui répond très rapidement malgré des tâches longues à réaliser.

Dans le cas de notre service d'envoi de email, même si un envoi dure 10 minutes, le micro service répondra instantanément et sera capable de fournir l'état d'avancement de l'envoi.

De plus il est même possible de programmer un minuteur avant l'envoi

Mise en place

Pour pouvoir utiliser Celery avec notre service Flask, il faut créer une application Celery que nous pourrons appeler et lancer.

Créons un dossier l'arboresence suivante:

```
worker
  __init__.py
  worker.py
  tasks.py
```

Le fichier worker.py contiendra notre application Celery qui sera lancée par le worker
Le fichier tasks.py contiendra les taches de notre application Celery

Commencons par créer l'application Celery à partir de la configuration de notre application Flask

Fichier `email_service/worker/worker.py`

```
# -*- coding: utf-8 -*-

import os
from celery import Celery
from config import Config

app = Celery()

app.conf.update({
    'BROKER_URL': Config.CELERY_BROKER,
    'BACKEND_URL': Config.CELERY_BACKEND,
    'CELERY_IMPORTS': Config.CELERY_IMPORTS
})
```

Pour l'envoi de email, nous ne pouvons plus utiliser Flask-Mail, nous allons utiliser la bibliothèque standard `smtp` et la configuration de notre application Flask

Créons un tache `send_emails` dans le fichier `email_service/worker/tasks.py`

```
import os
import logging
import smtplib
from .worker import app
from config import Config


logger = logging.getLogger(__name__)


@app.task(bind=True, name='send_emails')
def send_emails(self, recipients, subject, payload):
    """
    Send emails
    """
    try:
        if Config.MAIL_USE_SSL:
            server = smtplib.SMTP_SSL(Config.MAIL_SERVER, Config.MAIL_PORT)

        else:
            server = smtplib.SMTP(Config.MAIL_SERVER, Config.MAIL_PORT)
        server.ehlo()

        if Config.MAIL_USERNAME != '':
            server.login(Config.MAIL_USERNAME, Config.MAIL_PASSWORD)
```

```
        for recipient in recipients:
            message = "From: {0}\nTo: {1}\nSubject: {2}\n\n{3}".format(
                Config.MAIL_DEFAULT_SENDER,
                recipient,
                subject,
                payload
            )

            server.sendmail(Config.MAIL_DEFAULT_SENDER, recipient, message)

        server.close()

    except Exception as ex:
        logger.error('Something goes wrong --> {0}'.format(ex))
```

Pour simplifier le code, nous n'utilisons plus la fonction `render_email`

La dernière étape consiste à ajouter la configuration de Celery dans la configuration de notre application Flask :

```
...

class Config:
    """
    Base configuration
    """
    CELERY_BACKEND = 'redis://localhost/0
    CELERY_BROKER = 'redis://localhost/
    CELERY_IMPORTS = ('worker.tasks')
    MAIL_SERVER = 'localhost'
    MAIL_PORT = 1025
    MAIL_USE_TLS = False
    MAIL_USE_SSL = False
    MAIL_USERNAME = ''
    MAIL_PASSWORD = ''
    MAIL_DEFAULT_SENDER = 'will@dev.fr'
    LOG_PATH = '<your path>'
    LOG_SIZE = 20000
    LOG_COUNT = 10
    LOG_ENCODING = 'utf-8'
    LOG_LEVEL = 'DEBUG'

...
```

Utilisation de la tache

Maintenant que tout est en place, nous pouvons mettre en place l'envoi de fichiers en complètent la méthode d'envoi dans le fichier `email_service/app/api/endpoints/email.py`.

Une fois la tâche lancée, le service retournera l'identifiant de la tâche afin que l'utilisateur puisse suivre l'évolution.

Pour cela commençons par modifier le fichier `email_service/app/api/serializers/email.py` afin de définir le format de sortie de notre service

```
# -*- coding: utf-8 -*-
from flask_restplus import fields
from .. import api

send_email_model = api.model('Send Email model', {
    'recipients': fields.List(fields.String(), required=True,
description='Recipients list'),
    'subject': fields.String(required=True, description='Email subject'),
    'content': fields.String(required=True, description='Email content'),
    'template': fields.String(required=True, description='Email template')
})
send_email_reponse = api.model('Send Email response', {
    'id': fields.String(required=True, description='Send email task id')
})
```

Nous pouvons maintenant modifier le fichier `email_service/app/api/endpoints/email.py` afin d'appeler la tâche d'envoi de email

```
# -*- coding: utf-8 -*-

import re
from worker.tasks import send_emails
from flask import request, current_app
from flask_restplus import Namespace, Resource, abort
from ..serializers.email import send_email_model


ns = Namespace('email', description='Email related operation')


# ====================================================================
# ENDPOINTS
# ====================================================================
#   API email endpoints
#
# ====================================================================


@ns.route('/')
class EmailSend(Resource):

    @ns.expect(send_email_model)
    def post(self):
```

```
        """
        Send email
        """
        data = request.json

        for email in data['recipients']:
            if not re.match(r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-
.]+$)", email):
                current_app.logger.error('{0} not pass email
regex.'.format(email))
                abort(400, error='{0} not pass email regex.'.format(email))

        task = send_emails.s(data['recipients'], data['subject'],
data['content']).delay()

        return {'id': task.id}
```

Lancement

Il nous reste maintenant à démarrer le worker Celery et notre application

Lancer le worker :

```
celery -A worker.worker worker -P eventlet --loglevel=info
```

4 - Docker

Nous allons maintenant voir comment créer un container de notre application, cela passe par 3 étapes :

- Utiliser un serveur HTTP, le serveur HTTP de développement n'étant pas recommandé pour de la production
- Créer un container de notre application flask et son worker
- Créer un `docker-compose.yml` afin de packager notre application Flask avec la base de données Redis et le serveur smtp de développement

Créer un container de notre application Flask

Avant de créer un container de notre application, nous allons la rendre encore plus configurable.

Actuellement la configuration de notre application est stockée dans le fichier `email_service/config.py` et une fois insérée dans un container, nous ne pourrons plus modifier les valeurs sans reconstruire le container.

Pour pallier à ça, nous allons utiliser les variables d'environnement

Variables d'environnement

Voyons dans un premier temps les variables d'environnement.
Modifions le fichier `email_service/config.py` afin que chaque propriété soit surchargeable avec des variables d'environnement :

```
# -*- coding: utf-8 -*-
import os
import logging
from logging.handlers import RotatingFileHandler
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    """
    Base configuration
    """
    MAIL_SERVER = os.environ.get('MAIL_SERVER', 'localhost')
    MAIL_PORT = int(os.environ.get('MAIL_PORT', '1025'))
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS', 'False') == 'True'
    MAIL_USE_SSL = os.environ.get('MAIL_USE_SSL', 'False') == 'True'
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME', '')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD', '')
    MAIL_DEFAULT_SENDER = os.environ.get('MAIL_DEFAULT_SENDER',
'dev@dev.dev')
    CELERY_BACKEND = os.environ.get('CELERY_BACKEND', 'redis://localhost/0')
    CELERY_BROKER = os.environ.get('CELERY_BROKER', 'redis://localhost')
    CELERY_IMPORTS = ('worker.tasks')
    LOG_PATH = os.environ.get('LOG_PATH', os.path.join(basedir, 'logs.log'))
    LOG_SIZE = int(os.environ.get('LOG_SIZE', '20000'))
    LOG_COUNT = int(os.environ.get('LOG_COUNT', '10'))
    LOG_ENCODING = os.environ.get('LOG_ENCODING', 'utf-8')
    LOG_LEVEL = os.environ.get('LOG_LEVEL', 'DEBUG')
    @staticmethod
    def init_app(app):
        """
        Init app
        :param app: Flask App
        :type app: Flask
        """
        pass
```

Si nous relançons notre programme, rien ne change, mais nous pouvons maintenant redéfinir l'ensemble de la configuration

Serveur HTTP de production

Le serveur de développement de Flask n'étant pas conseillé en production, nous allons donc utiliser le serveur web `uwsgi`

Installation

Pour installer `uwsgi` il suffit d'utiliser pip

```
pip install uwsgi
```

Il faut ensuite ajouter un fichier `wsgi.py` que le serveur `uwsgi` ira lire :

```
from runserver import app as application
if __name__ == "__main__":
    application.run()
```

Dockerfile

Il est maintenant temps de créer les images des container de notre application, pour cela nous allons créer deux `Dockerfile`, un pour l'application Flask, l'autre pour l'application Celery

Créons l'arboresence suivante :

```
docker
  backend
    Dockerfile
  worker
    Dockerfile
```

Commencons par le `Dockerfile` de notre API dans le fichier `email_service/docker/backend/Dockerfile` :

```
FROM python:3.6

ADD ./requirements.txt /tmp/
RUN pip3 install -r /tmp/requirements.txt

RUN mkdir /email
ADD ./app/ /email/app/
ADD ./worker/ /email/worker/
ADD ./*.py /email/

WORKDIR /email

CMD ["python", "runserver.py"]
```

Rien de compliqué ici, nous partons d'une base Python 3, nous installons les requirements et nous ajoutons l'ensemble des fichiers de notre programme.

Nous pouvons maintenant construire notre container :

```
docker build -t email_dev -f .\docker\backend\Dockerfile .
```

Une fois construits, nous pouvons lancer notre API avec la commande suivante :

```
docker run --name email_dev -d -p 8000:8000 -e MAIL_SERVER=smtp_dev -e
CELERY_RESULT_BACKEND=redis://redis_dev/0 -e
CELERY_BROKER_URL=redis://redis_dev -e MAIL_PORT=25 --link
redis_dev:redis_dev --link smtp_dev:smtp_dev email_dev uwsgi --socket
0.0.0.0:8000 --protocol=http -w wsgi
```

Nous pouvons maintenant nous rendre à l'adresse `http://localhost:8000/api` et voir que notre service fonctionne.

La prochaine étape est de créer le container de notre application Celery dans le fichier `email_service/docker/worker/Dockerfile`

```
FROM python:3.6

ADD ./requirements.txt /tmp/
RUN pip3 install -r /tmp/requirements.txt

RUN mkdir /worker
ADD ./worker/ /worker/worker/
ADD ./config.py /worker/

WORKDIR /worker
```

Construisons notre container avec la commande suivante :

```
docker build -t email_worker_dev -f .\docker\worker\Dockerfile .
```

Une fois construits, nous pouvons lancer notre API avec la commande suivante :

```
docker run --name email_worker_dev -d -e MAIL_SERVER=smtp_dev -e
CELERY_RESULT_BACKEND=redis://redis_dev/0 -e
CELERY_BROKER_URL=redis://redis_dev -e MAIL_PORT=25 --link
redis_dev:redis_dev --link smtp_dev:smtp_dev email_worker_dev celery worker -
-app=worker.worker.app --concurrency=1 --hostname=email_worker@%h --
loglevel=INFO
```

Voilà notre service est fonctionnel et accessible à l'adresse `http://localhost:8000/api` et nous pouvons vérifier l'envoi des emails avec notre MailCatcher à l'adresse `http://localhost:1080`

Packager notre application et son environnement

L'étape suivante est de packager l'application et son environnement, pour cela, dans le dossier `docker`, nous allons créer un fichier `docker-compose.yml` qui contiendra notre application ainsi que Redis et MailCatcher

```
version: '3'

services:

  backend:
```

```
    image: email_dev
    command: uwsgi --socket 0.0.0.0:8000 --protocol=http -w wsgi
    environment:
      - MAIL_SERVER=smtp
      - MAIL_PORT=25
      - CELERY_BACKEND=redis://redis/0
      - CELERY_BROKER=redis://redis
    ports:
      - 80:8000

  worker:
    image: email_worker_dev
    command: celery worker --app=worker.worker.app --concurrency=1 --
hostname=email_worker@%h --loglevel=INFO
    environment:
      - MAIL_SERVER=smtp
      - MAIL_PORT=25
      - CELERY_BACKEND=redis://redis/0
      - CELERY_BROKER=redis://redis

  redis:
    image: redis

  smtp:
    image: tophfr/mailcatcher
    ports:
      - 1080:80
```

Pour lancer notre service, il suffit de lancer la commande :

```
docker-compose -p email_stack -f .\docker\docker-compose.yml up -d
```

Notre service est maintenant disponible à l'adresse `http://localhost/api` et notre MailCatcher à l'adresse `http://localhost:1080`
Pour éteindre notre service il suffit de lancer la commande :

```
docker-compose -p email_stack -f .\docker\docker-compose.yml stop
```

5 - Nginx

Actuellement nous avons 2 problèmes :

- Notre service est directement accessible sur le réseau
- Si nous lançons plusieurs services, nous devons ouvrir un port par service

Pour pallier à ça, nous allons utiliser Nginx en tant que proxy inverse

Mise en place

Pour mettre en place Nginx nous devons :

- Modifier notre fichier `docker-compose.yml`
- Ajouter un fichier de configuration à notre application
- Modifier la configuration de Nginx

Fichiers de configuration

Dans le dossier `docker` créons l'arboresence suivante

```
config
  app.ini
nginx
  nginx.conf
  services.conf
```

Pour lancer notre API nous allons utiliser un fichier .ini et le lancer avec uwsgi

Ficher `email_service/docker/config.ini` :

```
[uwsgi]
mount = /=wsgi.py
manage-script-name = true
master = true
processes = 5
socket = /sockets/email.socket
chmod-socket = 666
vacuum = true
die-on-term = true
```

Pour plus de rapidité entre Nginx et notre API nous allons communiquer par socket

Il faut ensuite configurer Nginx pour prendre en compte nos modifications, nous allons surcharger la configuration par défaut afin d'inclure nos services

Fichier de configuration `email_service/docker/nginx/nginx.conf` (honteusement copié sur internet):

```
# Define the user that will own and run the Nginx server
user  nginx;
# Define the number of worker processes; recommended value is the number of
# cores that are being used by your server
worker_processes  1;

# Define the location on the file system of the error log, plus the minimum
# severity to log messages for
error_log  /var/log/nginx/error.log warn;
# Define the file that will store the process ID of the main NGINX process
pid        /var/run/nginx.pid;
```

```
# events block defines the parameters that affect connection processing.
events {
    # Define the maximum number of simultaneous connections that can be
opened by a worker process
    worker_connections  1024;
}


# http block defines the parameters for how NGINX should handle HTTP web
traffic
http {
    # Include the file defining the list of file types that are supported by
NGINX
    include       /etc/nginx/mime.types;
    # Define the default file type that is returned to the user
    default_type  text/html;

    # Define the format of log messages.
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    # Define the location of the log of access attempts to NGINX
    access_log  /var/log/nginx/access.log  main;

    # Define the parameters to optimize the delivery of static content
    sendfile        on;
    tcp_nopush     on;
    tcp_nodelay    on;

    # Define the timeout value for keep-alive connections with the client
    keepalive_timeout  65;

    # Define the usage of the gzip compression algorithm to reduce the amount
of data to transmit
    #gzip  on;

    # Include additional parameters for virtual host(s)/server(s)
    include /etc/nginx/conf.d/*.conf;
}
```

Il nous reste remplir le fichier `services.conf` afin de déclarer nos services :

```
server {
  listen 80;
  charset utf-8;
```

```
  location / {
    uwsgi_pass unix:/sockets/email.socket;
    uwsgi_param SCRIPT_NAME /;
    uwsgi_modifier1 30;
    include uwsgi_params;
  }
}
```

docker-compose

Nous devons maintenant :

- Ajouter Nginx à notre `docker-compose.yml`
- Permettre la communication par socket entre Nginx et notre API
- Lancer notre API à l'aide du fichier ini

```
version: '3'

services:

  nginx:
    image: 'nginx:stable'
    volumes:
      - sockets:/sockets/
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/services.conf:/etc/nginx/conf.d/default.conf
    ports:
      - 80:80


  backend:
    image: email_dev
    command: uwsgi app.ini
    environment:
      - MAIL_SERVER=smtp
      - MAIL_PORT=25
      - CELERY_BACKEND=redis://redis/0
      - CELERY_BROKER=redis://redis
    volumes:
      - sockets:/sockets
      - ./config/app.ini:/email/app.ini

  worker:
    image: email_worker_dev
    command: celery worker --app=worker.worker.app --concurrency=1 --
hostname=worker@%h --loglevel=INFO
    environment:
      - MAIL_SERVER=smtp
      - MAIL_PORT=25
      - CELERY_BACKEND=redis://redis/0
```

```
      - CELERY_BROKER=redis://redis

  redis:
    image: redis

  smtp:
    image: tophfr/mailcatcher
    ports:
      - 1080:80

volumes:
  sockets:
    driver: local
```

6 - Création de bibliothèques

Jusqu'a présent nous avons vu comment utiliser des bibliothèques crées par des utilisateurs.

Nous allons maintenant voir comment créer nos bibliothèques et les rendre disponible sur `pip`

Installation des outils

Pour pouvoir créer une bibliothèque, nous avons besoin de 2 bibliothèques existantes `setuptools` et `wheel`

```
pip install setuptools
pip install wheel
```

`setup.py`

Le fichier `setup.py` est le fichier qui décris notre bibliothèque, c'est dans ce fichier qu'on indique les informations tels que l'auteur, la version, la description, les dépendances, etc

La liste des informations possibles est disponible à l'adresse https://docs.python.org/2/distutils/apiref.html?highlight=bdist

Example d'une bibliothèque `elonet_formation_4`

```
elonet_formation_4
    __init__.py
    operations.py
setup.py
```

elonet_formation_4/**init**.py

```
from .operations import *
```

elonet_formation_4/operations.py

```
import requests  # Dépendance


def operation_tres_compliquee(nombre1, nombre2):
    return nombre1 + nombre2


def operation_web(url):
    return requests.get(url)
```

setup.py

```
from setuptools import setup

# On importe notre bibliothèque
import elonet_formation_4

setup(
    name='elonet_formation_4',
    version='0.0.1',
    author='Rasta dev',
    author_email='arthur@elonet.fr',
    url='https://elonet.github.io/python3_formation_4/',
    packages=['elonet_formation_4'],
# On ajoute notre bibliothèque au setup
    install_requires=['requests==2.20.1'],
    description='Demonstration de creation d\'un package Python',
    plateformes='ALL',
)
```

`platformes` permet de spécifier la plateforme de destination, les valeurs possibles sont : ALL, WINDOWS, LINUX, MAC

Installation de la bibliothèque

Pour installer la bibliothèque suffit de lancer la commande suivante :

```
python setup.py install
```

Nous pouvons ensuite utiliser notre bibliothèque dans un interpréteur ou un programmer Python

```
from elonet_formation_4 import operation_tres_compliquee
operation_tres_compliquee(3, 4)
7
```

Upload sur PIP

Pour pouvoir upload une bibliothèque sur PIP, il faut d'abord s'enregistrer à l'adresse suivante : https://pypi.org/account/register/

Il faut ensuite configurer notre PIP à l'aide du fichier `.pypirc`

Créer ou modifier le fichier `~/.pypirc`

```
[distutils]
index-servers =
  pypi

[pypi]
username:<your_pypi_username>
password:<your_pypi_passwd>
```

Il suffit ensuite de lancer la commande

```
python setup.py sdist upload -r pypi
```

Une fois la bibliothèque uploadé, vous pouvez la retrouver à l'adresse :
https://pypi.org/manage/projects/

7 - Pip personnalisé

Dans certains cas nous ne souhaitons pas rendre nos bibliothèques disponibles sur le repos PIP officiel, dans ce cas nous devons héberger notre propre repos

Pour cela nous avons accès au projet pypiserver :
https://github.com/pypiserver/pypiserver

Serveur docker

Pour des raisons de simplicité, nous allons utiliser l'image Docker fournis

Commande :

```
docker run -p 80:8080 -v ~/.htpasswd:/data/.htpasswd
pypiserver/pypiserver:latest -P .htpasswd packages
```

Nous voyons que la commande attend un fichier `.htpasswd` pour gérer l'authentification

Préparation

Créons un dossier `pypi_local` qui contiendra notre fichier `.htpasswd`

```
mkdir pypi_local
cd pypi_local

htpasswd -c -s .htpasswd <your_username>
New password:
Re-type new password:
Adding password for user <your_username>
```

Lancement

Il faut ensuite lancer le serveur en lui fournissant un nom et en spécifiant le dossier contenant le fichier `.htpasswd`

```
docker run -p 80:8080 --name pypi_dev -v
/home/rastadev/pypi_local/.htpasswd:/data/.htpasswd
pypiserver/pypiserver:latest -P .htpasswd packages
```

Notre serveur PIP est maintenant en ligne à l'adresse : http://localhost

Configuration

Il faut de nouveau configurer notre fichier `.pypirc` afin d'ajouter notre nouveau serveur

```
[distutils]
index-servers =
  pypi
  local

[pypi]
username:<your_pypi_username>
password:<your_pypi_passwd>

[local]
repository: http://localhost
username:<your_username>
password:<your_password>
```

Nous pouvons désormais upload notre bibliothèque sur notre serveur à l'aide de la commande

```
python setup.py sdist upload -r local
```

Nous pouvons télécharger une bibliothèque depuis notre serveur à l'aide de la commande

```
pip install --index-url http://localhost elonet_formation_4
```