Python 3 Formation 4

Python 3 Formation 4

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='Re
cipients 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, descript
ion='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_f
ile:
        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(e
mail))
                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, e
ncoding='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(e
mail)) # 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):
        super().init_app(app)
        logger = logging.getLogger(__name__)
        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()))
        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 worke
r)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=celery(direct) key=celery

[tasks]
  . tasks.do_stuff
[2018-06-24 13:07:47,058: INFO/MainProcess] Connected to redis://localhost:63
79//
[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://loca
lhost: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[8a9
049b8-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 ajouter une usine à application Celery dans le fichier `email_service/app/__init.py`

Le code de l'usine à Celery provient directement de la documentation Flask :

```
# -*- coding: utf-8 -*-
from flask import Flask
from celery import Celery
from config import config

def make_celery(app):
    """
    Create Celery application

    :param app: Flask application
    :type app: Flask

    :return: Celery application
    :rtype: Celery
    """
    celery = Celery(app.import_name, backend=app.config['CELERY_RESULT_BACKEN
D'],
                    broker=app.config['CELERY_BROKER_URL'])
    celery.conf.update(app.config)
    TaskBase = celery.Task
    class ContextTask(TaskBase):
        abstract = True
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return TaskBase.__call__(self, *args, **kwargs)
    celery.Task = ContextTask
    return celery

def create_app(config_name='default'):
    """
```

```
    Create application
    :param config_name: Configuration name
    :type config_name: str
    :return: Flask app
    :rtype: Flask
    """
    # Import blueprint here
    # from .xxx import blueprint as xxx_blueprint
    from .api import blueprint as api_blueprint
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    # Register blueprint here
    # app.register_blueprint(xxx_blueprint)
    app.register_blueprint(api_blueprint)
    extensions(app)
    return app

def extensions(flask_app):
    """
    Init extensions
    :param flask_app:
    """
    pass
```

Nous voyons que Celery nécessite `CELERY_RESULT_BACKEND` et `CELERY_BROKER_URL` dans la configuration, ajoutons les propriétés dans notre fichier `email_service/config.py`

```
# -*- 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'
    CELERY_RESULT_BACKEND = 'redis://localhost/0'
    CELERY_BROKER_URL = 'redis://localhost'
    LOG_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
```

Création de la tache d'envoi de email

Maintenant que nous avons lié Celery à notre application Flask, nous pouvons reprendre ce que nous avons vu précédemment.

Créons un fichier `tasks.py` dans le dossier `email_service/app/` et ajoutons la logique d'envoi de emails

```
# -*- coding: utf-8 -*-
from datetime import datetime
from flask import current_app
from flask_mail import Message
from email.utils import format_datetime
from . import make_celery
from .extensions import mail
from .utils import render_email
celery = make_celery()

@celery.task(bind=True)
def send_email(self, args):
    try:
        with current_app.app_context():
            with mail.connect() as conn:
                for address in args['recipients']:
                    data = {
                        'date': format_datetime(datetime.utcnow()),
                        'to': address,
                        'from': current_app.config['MAIL_DEFAULT_SENDER'],
                        'subject': args['subject'],
                        'content': args['content']
                    }
                    msg = Message(
                        recipients=[address],
                        body=render_email(args['template'], data),
                        subject=args['subject']
                    )
                    conn.send(msg)
                return True
    except Exception as ex:
        raise ex
```

La ligne `with current_app.app_context()` permet d'être sûr que l'extension mail est initialisée et cela nous permet de lire la configuration de l'application Flask en cours

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='Re
cipients 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 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(e
mail))
                abort(400, error='{0} not pass email regex.'.format(email))
        from app.tasks import send_email
        task = send_email.apply_async(args=[data])
        return {'id': task.id}
```

Lancement

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

Lancer le worker :

```
celery -A app.tasks 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_RESULT_BACKEND = os.environ.get('CELERY_RESULT_BACKEND', 'redis://
localhost/0')
    CELERY_BROKER_URL = os.environ.get('CELERY_BROKER_URL', 'redis://localhos
t')
    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 un container de notre application, pour cela nous allons créer un `Dockerfile`.

Créons un dossier `docker` dans le dossier `email_service` puis créons un fichier `Dockerfile`

```
FROM python:3
ADD ./requirements.txt /tmp/
RUN pip3 install -r /tmp/requirements.txt
ADD ./app/ /app/
ADD ./*.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\Dockerfile .
```

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

```
docker run --name email_dev -d -p 8000:8000 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.

Par contre nous n'avons pas démarré notre worker.

Pour démarrer notre worket et notre service en même temps nous allons créer un fichier `start.sh` dans le dossier `email_service` qui contiendra les commandes pour lancer notre worker et notre application

```
#!/bin/sh
celery multi start emailworker -A app.tasks
uwsgi --socket 0.0.0.0:8000 --protocol=http -w wsgi
```

Il faut ensuite modifier et reconstruire notre image.

```
FROM python:3
ADD ./requirements.txt /tmp/
RUN pip3 install -r /tmp/requirements.txt
ADD ./app/ /app/
ADD ./*.py /
ADD ./start.sh /
```

Maintenant nous pouvons lancer notre container à l'aide

```
docker run --name email_dev -d -p 8000:8000 email_dev sh start.sh
```

Mais cela ne fonctionnera pas, car maintenant notre container n'a plus accès à Redis et MailCatcher il faut donc reconfigurer les adresses de Redis et de MailCatcher
Ce qui nous donne en fait :

```
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://r
edis_dev -e MAIL_PORT=25 --link redis_dev:redis_dev --link smtp_dev:smtp_dev
email_dev sh start.sh
```

Voilà notre service est fonctionnel et accessible à l'adresse `http://localhost:8000/api` et 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 nous allons créer un fichier `docker-compose.yml` qui contiendra notre application ainsi que Redis et MailCatcher

```
version: '2'
services:
  email:
    image: email_dev
    command: sh start.sh
    environment:
      - MAIL_SERVER=smtp
      - MAIL_PORT=25
      - CELERY_RESULT_BACKEND=redis://redis/0
      - CELERY_BROKER_URL=redis://redis
    ports:
      - 80:8000
    networks:
      - services_network
  redis:
    image: redis
    networks:
      - services_network
  smtp:
    image: tophfr/mailcatcher
    ports:
      - 1080:80
```

```
    networks:
      - services_network
networks:
  services_network:
    driver: bridge
```

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

```
docker-compose 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 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 `Dockerfile` et notre `docker-compose.yml`

Application

La première chose à faire et d'ajouter un fichier de configuration pour `uwsgi` afin de configurer le service web et d'utiliser les sockets pour la communication service <–> réseau.

Créons un fichier `app.ini` dans le dossier `email_service`

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

Nous allons ensuite créer un fichier `start_socket.sh`

```
#!/bin/sh
celery multi start emailworker -A app.tasks --loglevel=INFO --logfile=/var/lo
```

```
g/%n%I.log --concurrency=2
uwsgi --ini app.ini
```

Modifions le fichier `email_service/docker/Dockerfile` afin d'inclure tout les fichiers .sh et notre fichier .ini

```
FROM python:3
ADD ./requirements.txt /tmp/
RUN pip3 install -r /tmp/requirements.txt
ADD ./app/ /app/
ADD ./*.ini /
ADD ./*.py /
ADD ./*.sh /
```

Il faut maintenant reconstruire notre container

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

Package

La prochaine étape est de modifier notre fichier `docker-compose.yml` afin d'intégrer Nginx et les sockets

Nginx

Avant de modifier le fichier `docker-compose.yml`, nous allons devoir reconstruire le container Nginx pour prendre en compte notre service, pour cela créons un dossier `nginx` dans le dossier `email_service/docker`.

Dans ce dossier, créons un fichier `services.conf`

```
server {
  listen 80;
  charset utf-8;
  location / {
    uwsgi_pass unix:/var/sockets/email.socket;
    uwsgi_param SCRIPT_NAME /;
    uwsgi_modifier1 30;
    include uwsgi_params;
  }
}
```

Puis créons un fichier `Dockerfile` qui construira notre nouveau container Nginx :

```
FROM nginx
RUN rm /etc/nginx/conf.d/default.conf
COPY services.conf /etc/nginx/conf.d/
```

Docker compose

Nous pouvons maintenant modifier notre fichier `docker-compose.yml` afin d'y ajouter Nginx.

Afin que Nginx ait accès à notre socket `email.socket`, nous mettons en place un volume

```
version: '2'
services:
  nginx:
      build: ./nginx
      ports:
        - 80:80
      networks:
        - services_network
      volumes:
        - sockets:/var/sockets/
  email:
    image: email_dev
    command: sh start_socket.sh
    environment:
      - MAIL_SERVER=smtp
      - MAIL_PORT=25
      - CELERY_RESULT_BACKEND=redis://redis/0
      - CELERY_BROKER_URL=redis://redis
    networks:
      - services_network
    volumes:
      - sockets:/var/sockets/
  redis:
    image: redis
    networks:
      - services_network
  smtp:
    image: tophfr/mailcatcher
    ports:
      - 1080:80
    networks:
      - services_network
volumes:
  sockets:
    driver: local
networks:
  services_network:
    driver: bridge
```