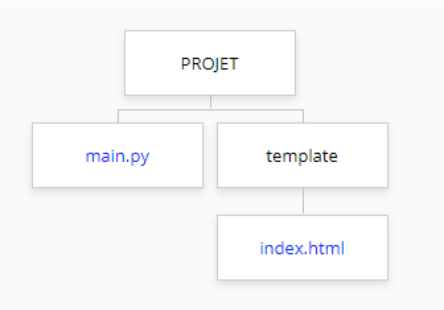
POURQUOI FLASK A BESOIN DE GUNICORN ET NGINX

1. Mise en situation

Nous avons une arborescence de Flask d'une page simple qui s'appelle **main.py**. Sa page WEB associé s'appelle **index.html**.



2. Introduction

```
>> WARNING: This is a development server. Dot not use it in a production deployment.
```

Ce message nous est indiqué que lorsque l'on lance Flask. Traduit cela veut dire qu'il ne faut pas l'utiliser l'application WEB Flask pour un déploiement. Cela tient pour plusieurs raisons :

- Flask ne peut pas traiter plus d'une requête à la fois.
- Un problème d'échelle : cela découle de la première raison : en cas d'une utilisation forte il y aura des fonctions qui ne vont pas être activées et qui vont produire des erreurs sur la page.
- Enfin cela fini par nous donner une page Internet qui peut être lente et qui varie en fonction du nombre d'utilisateurs.

La solution à ce problème : **Gunicorn** !

3. Gunicorn

J'admettrais ici avoir besoin de 3 utilisateurs. Ce nombre peut varier bien évidemment. Nous appellerons ces utilisateur 'Workers'. Cela permettra de mieux comprendre l'application dans le cas concret que je vais décrire. Le schéma ci-dessous va aussi permettre de mieux comprendre le tout :

Pour démarrer un serveur Flask avec Gunicorn, il faut entrer cette ligne de commande :

```
$ gunicorn3 --workers=3 --bind 0.0.0.0:5000 wgsi:app
```

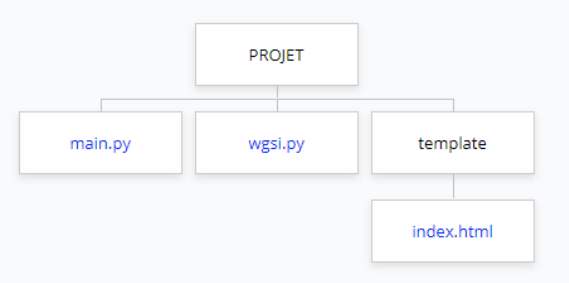
La commande exécute Gunicorn (3 pour Python 3) qui peux avoir jusqu'à 3 Workers, soit en fait 3 connexions depuis un navigateur WEB. 'Bind ' indique là où le serveur s'exécute : les IP admissibles à la connexion sur ce WEB Flask et le port utilisé qui par défaut est 5000. Wgsi :app : c'est le point d'entrée de l'application : il associe le nom du fichier (le code python) en tant qu'application (app).

Viens ensuite la partie qui ajoute un fichier dans l'arborescence du projet : **wgsi.py**. Cet ajout permet de faire le point d'entrée pour les différentes requêtes de chaque utilisateur. Ce fichier qui est un code très simple créer une application unique à chaque requête. C'est en quelque sorte le code dupliqué **main.py** associé aux nombres de Workers définis. Rien n'empêche cependant de mettre le code directement associé comme application, il est recommandé de passer par ce code intermédiaire notamment si l'on utilise des sessions dans Flask, etc…

Voici un code simple pour **wgsi.py** :

```
from main import app
if __name__ == "__main__":
   app.run()
```

On a alors en plus dans l'arborescence :



L'application Flask peut maintenant avoir 3 clients (dans cet exemple) et être accessible sur Internet depuis son adresse IP associé au port 5000 (sans prendre en compte un routeur, d'éventuelle redirection de ports ou encore un DDNS/ nom de domaine associé).

Mais dans le cas où l'on veut accéder à la page avec un vrai certificat HTTPS, cacher le port que Flask utilise sur le serveur, ou utiliser plusieurs ports pour Flask (80 et 443 à tout hasard !), dans ces cas il faut utiliser un reverse proxy.

La solution à ce problème : **NGINX** !

4. NGINX

Dans notre cas Nginx se charge de transférer des requêtes venant d'internet, avec un nom de domaine en utilisant un client pour Gunicorn.

L'un de ces concurrents est Apache. Et de la même manière on peut lui attribuer une page par défaut dans deux répertoires de notre serveur :

```
$ cd /var/www/html
$ cd /usr/share/nginx/html
```

Il faut alors changer le nom des fichiers par le même nom qu'inscrit (index.html dans la plupart des cas). Il est recommandé de faire une copie du fichier de base !

Mais lorsque l'on accède à la bonne ressource ce n'est plus la page par défaut que l'on a mais bien notre page Flask. Si dessous une configuration simple d'NGINX pour rediriger le trafic HTTP vers HTTPS, et en utilisant le port 5000 dans le fichier de cet endroit : `/etc/nginx/sites-enabled`

```
server {
    server_name exemple.com www.exemple.com ;
    error_page 497 https://exemple.com:5000;
    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/APPLI/APPLI.sock;
    }
    listen 5000 ssl;
    ssl_certificate /etc/letsencrypt/live/exemple.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/exemple.com/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
```

Error_page 497, redirige pour l'erreur 400 (497 est une précision de l'erreur 400) sur l'url fournie à côté. On peut tout à fait demander à rediriger vers une page d'erreur.

Listen 5000, c'est le port d'écoute. L'argument de derrière c'est qu'il attend de l'HTTPS.

Voici un exemple simple uniquement pour écoute le port 80 :

```
server {
    server_name exemple.com www.exemple.com ;
    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/APPLI/APPLI.sock;
    }
    listen 80 ;
}
```

Maintenant écouter le port 80 et 443 :

```
server {
    server_name exemple.com www.exemple.com ;
    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/APPLI/APPLI.sock;
    }
    Listen 80 ;
}
server {
    server_name exemple.com www.exemple.com ;
    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/APPLI/APPLI.sock;
    }
    listen 443 ssl;
    ssl_certificate /etc/letsencrypt/live/exemple.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/exemple.com/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
}
```