TP n°1 *Introduction à Flask*

Exercice 1. Mise en place d'un virtualenv

Tout d'abord vous devez créer un répertoire pour ce TP :

```
$ mkdir -p 2a-web-serveur/TP1
$ cd 2a-web-serveur/TP1
```

Une bonne pratique, pour le développement d'applications web en python, est d'isoler chacune dans son propre *virtualenv*. Dans celui-ci, on pourra installer les packages que l'on veut, sans avoir besoin d'être administrateur ; et les installations et mises à jour dans le virtualenv d'une application n'affectera pas celui d'une autre.

```
$ virtualenv -p python3 venv
Running virtualenv with interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/duchier/2018-web-server/tuto-flask/ven
Also creating executable in /home/duchier/2018-web-server/tuto-flask/
Installing setuptools, pip, wheel...done.
```

ceci a créé un répertoire venv :

```
  $ ls -l venv
total 16
drwxrwxr-x 2 duchier duchier 4096 nov.   6 15:58 bin
drwxrwxr-x 2 duchier duchier 4096 nov.   6 15:57 include
drwxrwxr-x 3 duchier duchier 4096 nov.   6 15:57 lib
-rw-rw-r-- 1 duchier duchier   59 nov.   6 15:58 pip-selfcheck.json
```

il faut maintenant *activer* le venv dans notre shell :

```
$ source venv/bin/activate
(venv) $
```

cette activation modifie certaines variables d'environnement pour faire en sorte que Python utilise notre venv au lieu de l'install globale. **Attention :** quand on ouvre un nouveau terminal, il faut bien entendu refaire cette activation car elle est locale au shell qu'on utilise.

Notez que l'invite de commande a été modifiée pour vous indiquer que votre venv est activé.

Exercice 2. Installation de Flask

Maintenant que nous avons activé le virtualenv, nous allons installer le micro-framework Flask :

```
(venv) $ pip install flask
```

Attention : les variables d'environnement `http_proxy` et `https_proxy` doivent être correctement informées.

Exercice 3. Mise en place d'un "hello world" avec Flask

Nous allons créer un sous répertoire `tuto-flask` pour notre application. Ce projet devra être versionné :

```
(venv) $ git init tuto-flask
(venv) $ cd tuto-flask
```

Dans le fichier `tuto-flask/app.py` nous créons l'appli suivante :

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "<h1>Hello World</h1>"
```

Enfin nous pouvons lancer notre application à l'aide de la commande `flask` :

```
(venv) $ FLASK_APP=app.py flask run
 * Serving Flask app "app.py"
 * Environment: production
   WARNING: Do not use the development server in a production environ
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Dans un navigateur, nous pouvons maintenant consulter l'url : `http://localhost:5000/`

Ajoutez et Committez `app.py`.

La commande `flask` offre par défaut 2 sous-commandes :

```
$ flask
Usage: flask [OPTIONS] COMMAND [ARGS]...

  This shell command acts as general utility script for Flask applica

  It loads the application configured (through the FLASK_APP environm
  variable) and then provides commands either provided by the applica
  Flask itself.

  The most useful commands are the "run" and "shell" command.

  Example usage:

    $ export FLASK_APP=hello.py
    $ export FLASK_DEBUG=1
    $ flask run

Options:
  --version  Show the flask version
  --help     Show this message and exit.

Commands:
  run    Runs a development server.
  shell  Runs a shell in the app context.
```

Celle que nous avons utilisée est la sous-commande `run` dont vous pouvez consulter la documentation :

```
flask run --help
```

Exercice 4. Installation du package `python-dotenv`

Il est malcommode d'avoir à préciser des variables d'environnement sur la ligne de commande. Le package `python-dotenv` va nous premettre de positionner ces variables dans un fichier `.flaskenv` à la racine de notre projet.

```
(venv) $ pip install python-dotenv
```

Puis nous créons un fichier `.flaskenv` à la racine du projet :

```
FLASK_APP=app.py
FLASK_ENV=development
```

Nous pouvons à présent tester que ça marche comme avant :

```
flask run
 * Serving Flask app "app.py" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 192-403-924
```

`flask` a automatiquement chargé les variables d'environnement du fichier `.flaskenv`. `FLASK_ENV=developm` active entre-autre le mode debug.

Exercice 5. Restructuration de notre appli

Pour l'instant notre appli est entièrement contenue dans un seul fichier. Nous allons lui donner une structure plus conventionnelle. Tout d'abord, nous créons le répertoire `tuto-flask/tuto` :

```
(venv) $ mkdir tuto
```

Nous prenons le début du fichier `./app.py` et nous le plaçons dans le fichier `./tuto/app.py` :

```
from flask import Flask
app = Flask(__name__)
```

Nous prenons ensuite le milieu du fichier `./app.py` et nous le plaçons dans le fichier `./tuto/views.py` en ajoutant un import :

```
from .app import app

@app.route("/")
def home():
    return "<h1>Hello␣World</h1>"
```

Nous ajoutons le fichier `tuto/__init__.py` qui charge les fonctionnalitées nécessaires de notre appli :

```
from .app import app
import tuto.views
```

Finallement nous effaçons le fichier `./app.py` et nous modifions une ligne du fichier `.flaskenv` :

```
FLASK_APP=tuto
```

Vérifiez que ça marche comme avant. Ajoutez et Committez.

Exercice 6. Mise en place d'un template

Notre vue retourne directement un string représentant un document HTML. Souvent le contenu d'une page est dynamique, et pour cela il est pratique d'utiliser des templates. Flask s'attend à trouver les templates dans le sous-répertoire `tuto/templates` de notre appli :

```
(venv) $ cd tuto
(venv) $ mkdir templates
```

Créez le template `tuto/templates/home.html` avec le contenu suivant :

```
<!doctype html>
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    <h1>{{ title }}</h1>
    <ul>
      {% for x in names %}
      <li>{{ x }}</li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Ce template est paramétré par 2 variables `title` et `names`. Pour l'utiliser, nous modifions `tuto/views.py` de la manière suivante :

```
from .app import app
from flask import render_template

@app.route("/")
def home():
    return render_template(
        "home.html",
        title="Hello World!",
        names=["Pierre", "Paul", "Corinne"])
```

Pour ajouter du CSS, nous allons créer un fichier `home.css` dont le contenu est statique plutôt que dynamique. Flask s'atend à trouver les fichiers statiques dans `tuto/static`.

```
(venv) $ mkdir static
```

Dans le fichier `tuto/static/home.css` nous plaçons le code suivant :

```
h1 {
    color: red;
}
```

Pour l'utiliser, nous ajoutons la ligne suivante dans le <head> du template `home.html`

```
<link rel="stylesheet"
      href="{{ url_for('static', filename='home.css') }}">
```

Vérifiez que ça marche comme avant. Ajoutez et Committez !

Exercice 7. Ajouter des pages à l'application

Le but de cet exercice est de vous entrainer à ajouter des pages et des templates. Si vous préférez le faire sur de vraies données, j'ai scrapé, sur amazon, une liste de bouquins (les 100 meilleures ventes dans la catégorie Science-Fiction) :

- `/pub/2A/web/SF/data.yml` contient une liste de bouquins dans le format yaml. Pour chaque livre on a l'auteur, le titre, le prix, l'url de sa page sur amazon, et le nom du fichier contenant l'image de couverture.
- `/pub/2A/web/SF/images.tar` est une archive contenant les fichiers d'images de couvertures.

Pour charger du YAML en python, vous devrez installer le package `PyYAML`. Ensuite, le code à utiliser (en mode intéractif) est :

```
import yaml

data = yaml.load(open("data.yml"))
```

Dans un module python, il est nécessaire de préciser le répertoire :

```
import yaml, os.path

data = yaml.load(
    open(
        os.path.join(
            os.path.dirname(__file__),
            "data.yml")))
```