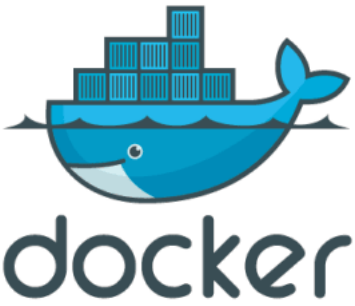
IA Data Entrepreneurship Program

B-IADATA-xxx

Back II

Interaction Styles in Back-End development



1.0

Flask est un micro framework pour le web, developpé en python.
Dans ce deuxième atelier sur le développement back-end, nous allons expérimenter plusieurs styles d'interaction avec un back-end:

- via une API, avec Flask-RESTFul
- via des socket, avec Flask-SocketIO
- via des workers avec Flask-Celery

Environnement technique

- Editeur de code : VsCode / atom / sublimetext / etc.
- Langage de programation : Python 3.x
- Libs Python : Flask, Flask-SQLAlchemy, Flask-RESTFul, Flask-Migrate, Flask-Celery, Flask-SocketIO
- Gestion de BDDs : PostgreSQL, Adminer (ou PHPMyAdmin)
- Contenarization : Docker, Docker-Compose
- CURL

I. Prélude

Python et Flask doivent doit être installés sur la machine pour cet exercice

Dans ce prélude nous allons :

1. Installer Flask-RESTFul et coder une API très simple
2. Exécuter le serveur d'API en local
3. Faire des requête d'API depuis une invite de commandes
4. Faire des requêtes d'API depuis un console python

I.1. Installation de Flask-RESTFul

Dans une invite de commandes, utilisez la commande `pip` pour installer la lib Flask-RESTFul

Terminal

```
B-IADATA-xxx> mkdir 00-prelude
B-IADATA-xxx> cd 00-prelude
B-IADATA-xxx/00-prelude> pip install flask_restful
```

Créez un fichier python `app.py` avec le code suivant :

```python
# app.py
from flask import Flask
from  flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)
mates = [
    {'patrick@epitech.eu': {'nom': '...', 'prenom': '...', 'ecole': 'EPITECH'}},
    {'viken@epitech.eu': {'nom': '...', 'prenom': '...', 'ecole': 'EPITECH'}},
    {'benoit@epitech.eu': {'nom': '...', 'prenom': '...', 'ecole': 'emlyon'}},
    {'jean@epitech.eu': {'nom': '...', 'prenom': '...', 'ecole': 'emlyon'}}
]

class TeamMates(Resource):
    def get(self):
        return mates

api.add_resource(TeamMates, '/')

if  __name__ == '__main__':
    app.run(debug=True)
```

I.2 Execution du serveur d'API

Dans une invite de commandes, exécutez votre serveur d'API :

Terminal

```
B-IADATA-xxx/00-prelude> python app.py


* Serving Flask app "app" (lazy loading)
* Environment: production
    WARNING: This is a development server. Do not use it in a production deployment
        .
    Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Félicitations !

Laissez votre serveur *allumé*

I.3 Accès depuis une invite de commande

Téléchargez l'utilitaire `curl` compatible avec votre `os` (windows, linux ou mac). `curl` vous permet d'envoyer des requêtes HTTP à un serveur web quelconque depuis une invite de commande.

Dans un autre terminal exécutez la commande suivante pour interrogger votre API :

Terminal

```
B-IADATA-xxx/00-prelude> curl 127.0.0.1:5000/
```

Vous devriez obtenir la liste de vos données brutes – le resultat de la fonction `TeamMates::get`.

I.3 Accès depuis un interpreteur python

Dans une invite de commandes, utilisez la commande `pip` pour installer l'interpreteur `ipython`

Terminal

```
B-IADATA-xxx> cd 00-prelude
B-IADATA-xxx/00-prelude> pip install ipython
B-IADATA-xxx/00-prelude> ipython
```

Affichez les données de votre API à partir de `ipython` avec le module `requests` et sa fontion `get`.

Terminal

```
B-IADATA-xxx> cd 00-prelude
B-IADATA-xxx/00-prelude> pip install ipython
B-IADATA-xxx/00-prelude> ipython
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from request import get
In [2]: get('http://127.0.0.1:5000/').json()
```

Allez encore plus loin :

1. depuis n'importe quelle ordinateur de votre réseau local en remplaçant **app.run()** par `app.run(host='0.0.0.0')` dans le fichier `app.py`. Essayez avec votre téléphone portable.

2. de manière sécurisée, de n'importe où dans le monde et gratuitement grâce à ngrok. Essayez avec votre équipe…

3. Depuis votre application React-Native en affichant les résultats dans un *component*.

EXERCICE 1

1. Formatez le contenu du dossier `00-prelude` dans le dossier `01-full-api` en respectant l'arborescence traditionnelle d'un projet de serveur d'API flask
2. Diversifier les requêtes pour bien interroger les données.

Tout d'abord, rendez-vous à cette page pour un petit point documentation.

EXERCICE 1.1. REFORMATAGE

Nous allons transformer le code de la section précédente pour qu'il respecte l'arborescence suivante :

```
Terminal
B-IADATA-xxx> cd 01-full-api
B-IADATA-xxx/01-full-api> tree -distfirst -charset=ANSI

.
|-- application/
|   |-- common/
|   |   |-- __init__.py
|   |   `-- util.py
|   |-- resources/
|   |   |-- __init__.py
|   |   `-- teammates.py
|   `-- __init__.py
|-- config.py
`-- run.py
```

Dans cette organisation :

- le dossier `./common` représente les fonctions utilisées dans toutes les APIs quelque soit la ressource.
- la définition des ressources accessibles se fait dans le dossier `./resources`
- la ressource `teammates.py` vous permet d'organiser l'accès à vos données (en l'occurence la liste de vos collaborateurs dans le parcours *IATECH*)

Définissez les fichiers `run.py`, `__init__.py` (tous) et `teammates.py` pour reproduire le serveur dans le même état que l'exercice précédent – tous les tests précédents (`ipython` et `curl`) doivent fonctionner.

```
Terminal
B-IADATA-xxx> cd 01-full-api
B-IADATA-xxx/01-full-api> python run.py
```

EXERCICE 1.2. FULL RESTAPI

En vous inspirant de cet exemple :

- Ajouter une méthode `POST` dans la ressource définie dans le fichier `teammates.py` pour ajouter un nouvel élément.
- Ajouter la ressource `teammate.py` pour manipuler les données indivuellement : `get`, `put`, `delete`

Les ressources et les endpoints de votre serveur d'API doivent être définis de la manière suivante :

```
# fichier teammate.py
class TeamMate(Resource):
    def get(self, login):
        # code

    def delete(self, login):
        # code

    def put(self, login):
        #code

# fichier teammates.py
class TeamMates(Resource):
    def get(self):
        # code

    def post(self):
        # code

# dans le fichier run.py
# ...
from application.resources.teammate import TeamMate
from application.resources.teammates import TeamMates
from application import api
# ...
api.add_resource(TeamMates, '/teammates')
api.add_resource(TeamMate, '/teammates/<todo_id>')
```

EXERCICE 2 (INTERMéDIAIRE)

Dans ce deuxième exercice, nous allons :

- Installer les outils nécessaires pour permettre une communication par socket entre un client (page web) et un serveur
- Créer une mini webapp de chat

EXERCICE 2.1

Créez le dossier `02-chat-app` et définissez-y, grâce à un fichier `docker-compose.yaml`, tous les services necessaires pour reproduire l'initialization de cette section.

Terminal

```
B-IADATA-xxx> cd 02-chat-app
B-IADATA-xxx/02-chat-app> tree -distfirst -charset=ANSI

.
|-- redis/
|-- web/
|-- .env
`-- docker-compose.yml
```

EXERCICE 2.2

En reprenant de cet exemple, intégrez une application de chat dans un service docker et exécuter le tout avec la commande `docker-compose up`.

Exercice 3. Celery (avancé)

Pour cet exercice, consulter la documentation de Celery et de l'extension Flask-Celery.
Le but de l'exercice est de reproduire le projet présenté dans ce post, i.e. :

- Instancier un pool de worker (celery) sur votre mini serveur
- Définir un front web à partir duquel l'utilisateur pourra à la fois lancer et monitorer des tâches sur le serveur. Les tâches sont prises en charge par les workers.

Reproduisez l'exemple dans votre propre environnement et exécuter le tout grâce à la commande `docker-compose up`. Pouvez-vous instancier des tâches hebdomadaires ?