SCIENCES SORBONNE UNIVERSITÉ

Automne 2024

À la découverte de Flask

FastAPI

TME FastAPI web framework

Objectifs

- ✯ Faire marcher un serveur FastAPI.
- ✯ Faire un petit site web.
- ✯ Savoir lire la documentation / Trouver la documentation.

1 Introduction

1.1 FastAPI

FastAPI est un framework pour le développement web écrit en python. Un framework est un cadre pour développer une application (web ici). Elle offre des fonctions de bases et des services pour faciliter le développement. De plus, FastAPI supporte la programmation asynchrone ce qui permet de lancer des tâches en parallèles. FastAPI est très légère et son comportement de base est d'utiliser un serveur existant `uvicorn` en 3 lignes python. Évidemment ce framework peut être enrichi par d'autres bibliothèques pour ajouter des fonctionnalités comme la gestion des formulaires ou l'utilisation de base de données.

1.2 Le plus petit code

FastAPI est installé sur les cartes SD des raspberry Pi.

Le plus simple code pour une application web, qui affiche une page *Hello World!*

```
import uvicorn
from fastapi import FastAPI  # import de la classe FastAPI
app = FastAPI()              # Création de l application

@app.get("/")                # Traitement de la requête get http
def hello() -> str:          # Valeur de retour pour la réponse http
    return "Hello World"


if __name__ == "__main__":
    uvicorn.run(app)  # lancement du serveur HTTP + WSGI avec les options de debug
```

👉 Recopier ce code dans un fichier python. Vous pouvez ensuite lancer ce serveur de développement avec python.

Les lignes suivantes devraient s'afficher

```
INFO:     Started server process [19048]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Cela veut dire que le serveur tourne, que vous pouvez l'arrêter avec CTRL-C, et que si vous ouvrez un navigateur internet à l'adresse `http://127.0.0.1:8000/` vous verrez une page qui vous dit bonjour.

1.3 La route

Le coeur du code précédent est la fonction hello, c'est elle qui retourne la valeur qui sera affichée quand vous serez à la racine de votre site. Elle est associer à la page racine c'est à dire à la page appelée quand vous allez à l'adresse `http://127.0.0.1:8000/` donc la *route '/'*.

De la même façon on peut définir des fonctions pour d'autres routes de votre site et afficher d'autres pages.

☛ Créer une route `@app.get('/about')` qui affiche un petit texte de description de votre site, cette page sera accessible via l'adresse `http://127.0.0.1:8000/about`.

2 Prise en main

2.1 Rendu HTML

Actuellement, votre application n'affiche qu'un simple message sans aucun HTML.

☛ Vous allez modifier le code pour changer la réponse http donc le return de la fonction `hello` et ajouter la variable user (elle pourra nous servir quand on mettra en place la base de données).

```
from fastapi import FastAPI
from fastapi.responses import HTMLResponse

def hello() -> str:
    user = {'username': 'Cécile'}
    return ''' <html>
    <head>
       <title>Ma première page Fast</title>
    </head>
    <body>
       <h1>Hello, '''  + user['username'] + '''! </h1>
    </body>
    </html>'''
```

2.2 Template

La solution précédente n'est pas satisfaisante pour un gros site. Pour mettre en place les pages et la gestion par le serveur, il faut tout d'abord créer un répertoire `templates/` dans le répertoire où se trouve le serveur.

Dans ce répertoire nous allons commencé par faire une page d'accueil du nom `hello.html` de la forme suivante :

```
<!doctype html>
<html>
    <head>
        <title>{{ title }} - Ma page</title>
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

Pour prendre en compte nos pages, il faut d'abord préciser où elles se trouvent et pouvoir accepter les requêtes.

```
from fastapi import FastAPI, Request
from fastapi.templating import Jinja2Templates

templates = Jinja2Templates(directory="templates/")
```

La fonction `hello` doit donc aussi être modifiée pour retourner la page html.

```python
def hello(request:Request) -> str:
    user = {'username': 'Cécile'}
    return templates.TemplateResponse('hello.html',{'request': request,'title':'First page', 'user':user}
```

☛ Relancer le serveur et observer les modifications. La fonction `templates.TemplateResponse` utilise *Jinja2* qui permet entre autre de remplir les blocs `{{...}}` dans votre template. Cela permet de passer des informations de votre code python à votre affichage en html.

Si vous obtenez des erreurs `jinja2.exceptions.TemplateNotFound: hello.html` vérifier que votre répertoire s'appelle bien `templates` et votre fichier `hello.html`.

2.3 Le style

En plus de vos fichier templates, il est aussi possible d'ajouter vos styles css !

Ils doivent être un répertoire `static/`, dans ce répertoire créer un répertoire `css/`.

☛ Créer votre fichier de style `style.css` qui modifie le style de `h1`que vous mettrez dans le répertoire `static/css`. L'arborescence doit être de la forme suivante :

```
server/
|-- votre_server.py
|-- static
|   |-- css
|   |   |-- style.css
|-- templates
    |-- hello.html
```

☛ Ajouter la prise en compte du réperoire lstinlinestatic par le serveur :

```python
...
from fastapi.staticfiles import StaticFiles
...
app.mount("/static", StaticFiles(directory="static"), name="static")
```

Dans le code fichier `hello.html`, il faut alors ajouter le lien vers le fichier css dans le `head`.

```
<link rel="stylesheet" href="static/css/style.css">
```

☛ Tester votre serveur avec le style css.

2.4 Javascript

C'est exactement le même principe pour faire appelle à des script JS. Les scripts js devront se trouver dans le répertoire `static/js`.

☛ Reprendre votre TP sur la visualisation avec des scripts JS et chosir un chart à afficher (par exemple la consommation au petit déjeuner) et faites le marcher sur votre serveur.

- Copier votre fichier html dans le répertoire template.
- Modifier les chemins de styles.
- Créer une nouvelle route dans le serveur python `@app.get("/chart",response_class=HTMLResponse)`.
- Creer une fonction qui retourne le fichier html.

Si la machine sur laquelle tourne le serveur n'est pas connectée à internet, il se peut que cela ne marche pas complètement.

2.5 Plus d'options pour les template grâce à Jinja2

On a vu que l'on pouvait passer des paramètres avec les blocs `{%...%}`. Il est possible d'utiliser aussi des structures conditionnelles et des boucles avec les blocs suivants :

- `{% if ... %}{% endif %}`
- `{% for ... %}{% endfor %}`

Beaucoup de code vont se retrouver dans toutes vos pages html, pour éviter de les recopier à chaque fois, il est possible de créer une page de base qui sera incluse dans vos futurs pages.

Un exemple de `base.html`

```
<!doctype html>
<html>
    <head>
      {% if title %}
      <title>{{ title }}</title>
      {% else %}
      <title> Une page </title>
      {% endif %}
      <link rel="stylesheet" href="{{ url_for('static', path= 'css/style.css') }}">
    </head>
    <body>
        {% block content %}{% endblock %}
    </body>
</html>
```

Un exmemple d'utilisation de la page de base, le page `menu.html`

```
{% extends "base.html" %}

{% block content %}
    <h1>Hello, {{ user.username }}!</h1>
    {% for p in menu %}
    <div><p>{{ p }}</p></div>
    {% endfor %}
{% endblock %}
```

☛ Tester l'utilisation de template de base avec la route suivante :

```
@app.route('/menu')
def menu():
    user = {'username': 'Cécile'}
    list_menu=['brunch','oeufs','tomates','salade','dessert']
    return render_template('index.html', title='Menu', user=user,menu=list_menu)
```

3 Connexion avec sa base de données

Vous pouvez récupérer la base de données du TME BD. Il est possible de manipuler la base de donnée SQLlite3 directement avec python. Nous utiliserons ici la bibliothèque sqlite3.

Le code suivant permet de se connecter avec une BD et d'exécuter des requêtes SQL.

```
import sqlite3

connect = sqlite3.connect('fournissuers.db') # Connexion au fichier
paris = connect.execute('SELECT * FROM ;')   # Une requête sql
connect.close()  % On ferme la connexion
```

☛ Créer une page qui liste tous les articles rouge

4 Gestion Asynchrone

4.1 Concurrence

Le code du serveur peut être asynchrone. Cela veut dire que le serveur peut demander à une tâche de faire un calcul et pendant ce temps, notre serveur peut continuer à fonctionner. Lorsque cette tâche est terminée, il peut reprendre son exécution où il avait besoin de la fin de cette tâche.

On dit que l'exécution est asynchrone car le serveur peut faire autre chose et revenir quand la tâche est terminée. Plus de détails et une explication avec des burgers ici.

4.2 Exemple

Pour illustrer le principe nous allons voir un exemple avec un "faux" capteur du fichier `sensors.py` disponible sur moodle. Dans ce fichier vous voyez une fonction `do_something` qui est asynchrone et qui va écrire sur le terminal et dans la base de donnée toutes les 5 secondes (notez le mot clé `async`).

```
  async def do_something(self):
        """ Do something with the sensors, e.g., get data

        Args:
            input_data (str, optional): . Defaults to """.
```

Dans votre fichier `app.py`, nous allons ajouter la gestion des évènements asynchrones avec la bibliothèque `asyncio` et des basses de données.

```
from sensors import Sensor # import the sensor
import sqlite3 # For databases
import asyncio # For asynchronous behaviors

sensor = Sensor() # init sensor

def create_db(''):
"""
Create database if it does not exist
"""
    conn = sqlite3.connect('database.db')
    conn.execute('CREATE TABLE IF NOT EXISTS MyTable (id INTEGER PRIMARY KEY AUTOINCREMENT,
    some_data TEXT)')
    conn.commit()
    conn.close()
```

Nous allons ajouter des actions qui seront effectuer quand le server sera lancer pour se connecter à la base de données et lancer la récupération des données par les capteurs.

```
@app.on_event('startup') # site se lance
async def app_startup():
    create_db()
    asyncio.create_task(sensor.do_something()) # ajout de la tâche qui capte les données
```

Il va nous falloir une nouvelle page pour visualiser les données, vous pouvez utilisez le fichier `stats.html` disponible sur moodle, et le mettre dans template.

```
@app.route('/stats')
def stats(request: Request):
    dbcontent = sensor.query_data() # fetch DB content
    context =  {"request": request, "nbrow": len(dbcontent)}
    return templates.TemplateResponse('stats.html', context) #display html template
```

☛ Modifier sensors pour prendre une vraie donnée captée.

5 Ressources

Voici les bases pour survivre dans fastAPI mais vous pouvez faire encore plus de choses. Voici 2 tutos super complet :
- en francais
- en anglais
- en anglais aussi