Serveur web avec Python Flask

Objectifs

- Observer les échanges entre un client et un serveur web
- Réinvestir le travail sur html, css et js
- Traiter un formulaire côté serveur avec python.

Le module `flask` de python permet de travailler avec un serveur web installé sur votre ordinateur.
Le serveur HTTP intégré à Flask fonctionne sur le port 5000, ainsi pour accéder à la page `index.html`, il faudra écrire l'adresse

`http://localhost:5000/`

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [19/Mar/2020 10:04:19] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [19/Mar/2020 10:04:30] "POST /bonjour HTTP/1.1" 200
```

Activités Élèves

- Installer le serveur en python et un premier exemple : "Hello World !"
- Structurer le site web
- Intégrer des variables "python" dans un fichier html
- Insérer des liens hypertext
- Une page avec javaScript
- Traiter le retour des informations d'un formulaire côté serveur avec python avec GET ou POST
- Faire un "chat"

Le serveur

Pour créer le serveur, on a besoin du sous-module `Flask` du module `flask`.

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

@app.route("/")
def hello():
    return "<p>Hello World!</p>"

app.run()
```

Ce qui donne dans un navigateur web à l'adresse `http://localhost:5000/` :

Structurer

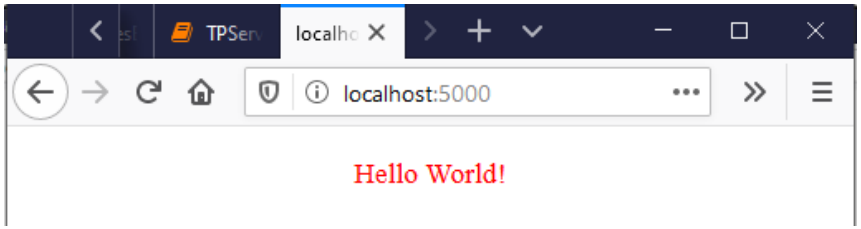
On a besoin de la fonction `render_template` du module `flask`
Cette fonction va permettre d'indiquer quelle est la page à renvoyer ; les répertoires associés dans lesquels doivent être enregistrés les fichiers html , css et js ont des noms bien spécifiques.
Le répertoire `templates` contiendra les fichiers html et le répertoire `static` contiendra les fichiers css et js.

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

@app.route("/")
def hello():
    return render_template("index.html")

app.run()
```

Dans le fichier html, les adresses des fichiers css et js s'exprimeront en fonction de la structure précédente :

```
<head>
    <meta charset="utf-8">
       <link rel ="stylesheet" type="text/css"
       href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
     <p > Hello World! </p>
</body>
```

Ce qui donne :



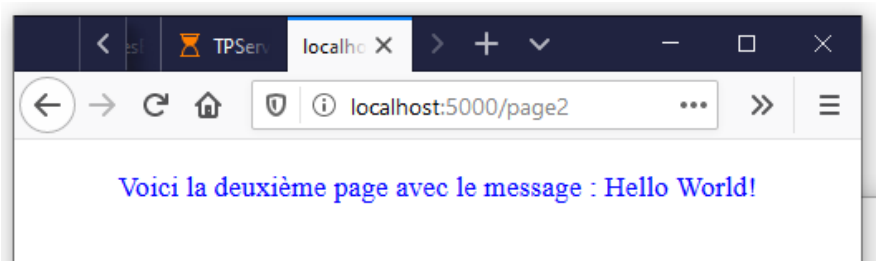
Pour utiliser des variables dans le fichier html, on les écrit entre doubles accolades :

```
<p id ="bleuCentre"> Voici la deuxième page
 avec le message :  {{bonjour}}
 </p>
```

Dans le fichier python, la valeur contenue dans la variable peut être définie dans l'appel de la fonction `render_template` ou en dehors.

```
@app.route("/page2")
def page():
    return render_template("page2.html", bonjour = "Hello World!")
```

Ce qui donne :



Lien vers une autre page html

Comme pour les fichiers css, les adresses relatives vont utiliser un format spécifique qui fera intervenir la fonction de python attachée à la page et non le nom du fichier html.
Dans le fichier py on a :

```
@app.route("/suivante")
def suite():
    return render_template("page2.html")
```

Dans le fichier html, on a :

```
<a href="{{ url_for('suite') }}"> Vers la page 2 </a>
```

Ce qui donne :

Une page avec javaScript

Comme pour les fichiers css, les fichiers js seront enregistrés dans le répertoire `static` et les adresses relatives auront aussi le même format :

```
<script defer="defer" language="JavaScript"
src= "{{ url_for('static', filename='script.js') }}"></script>
```

Ici en cliquant sur le titre, celui-ci passe en orange :

Un formulaire avec POST

Dans le fichier `index.html` :

```
<h1>Indiquer votre nom et votre prénom</h1>
   <form action = "http://localhost:5000/bonjour" method="POST">
       <p><label>Nom</label> :  <input type="text" name="Nom"></p>
       <p><label>Prénom</label> :  <input type="text" name="Prénom"></p>
       <p><input value="Valider" type="submit"></p>
   </form>
```

Ce qui donne :

Dans le fichier `serveur.py` :

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/')
def home():
    return render_template("index.html")

@app.route('/bonjour', methods=['POST'])
def hello():
    resultat = request.form
    nom = resultat['Nom']
    prenom = resultat['Prénom']
    nomComplet = nom + " " + prenom
    return render_template("bonjour.html" , message = nomComplet)

app.run()
```

L'objet `request.form` référencé ici par la variable `resultat` est un objet python de type :
`<class 'werkzeug.datastructures.ImmutableMultiDict'>`.
Les "clés" sont les chaînes de caractères contenues dans les attributs `name` des éléments du formulaire.

Dans le fichier `bonjour.html` qui sera la page renvoyée après traitement des éléments du formulaire, on fait référence à la variable `message` :

```
<h1> Bienvenue {{ message }} </h1>
<a href="{{ url_for('home') }}"> Retourner à la page d'accueil </a>
```

Ce qui donne :

On reprend le même exercice avec l'envoi du formulaire avec GET, on récupère les renseignements du formulaire à l'aide de l'objet `request.args`.

On peut ainsi observer la différences des échanges dans la console :

Avec POST

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [19/Mar/2020 09:03:55] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [19/Mar/2020 09:04:06] "POST /bonjour HTTP/1.1" 200 -
```

Avec GET

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [19/Mar/2020 09:13:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [19/Mar/2020 09:13:16] "GET /bonjour?Nom=Dupont&Prénom=Bob HTTP/1.1" 200 -
```

Mise en œuvre d'un serveur de chat avec flask socket-io

Objectifs de l'activité :

- Approfondir la notion de communication client - serveur
- Montrer les limites des requêtes POST et GET pour certains usages
- Introduire la notion de socket

Une page web ...

... contenant un formulaire méthode POST :

```
<body>
  <h1>Mon premier chat avec Flask-socketio</h1>
  <div id="zoneMessage">
  <p>
      <b style="color: #000">{{nom}}</b> {{message}}
  </p>
  </div>

  <form id="myForm" method="POST">
    <input id="in1" name="nom" type="text" placeholder="User Name"/>
    <input id="in2" name="message" type="text" placeholder="Message"/>
    <input type="submit"/>
  </form>
```

Le serveur renvoie la page avec le message quand il reçoit le retour formulaire.

```
#définition de la page principale
@app.route('/')
def index():
    return render_template('index.html', message = "Pas encore de message")

#retour de formulaire de la page index
@app.route('/', methods=['POST'])
def retourFormulaire():
    text = request.form['nom']
    text2 = request.form['message']
    return render_template("index.html" , message = text2, nom = text)
```

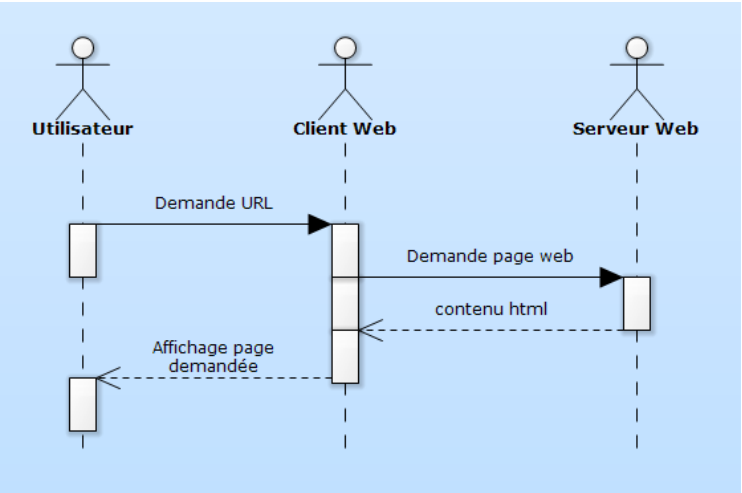
Problèmes

Deux clients qui ont chargé la page web :

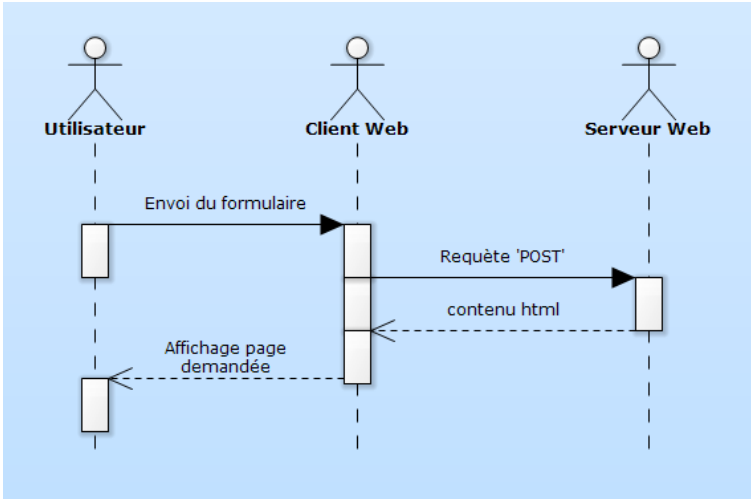
- Chacun reçoit son propre message ! sans intérêts
- La page est rechargée après l'envoi du formulaire.

Client-Serveur

Dans une communication classique de type client-serveur, l'initiative de la communication appartient toujours au client. Le serveur ne peut rien envoyer sans demande du client !

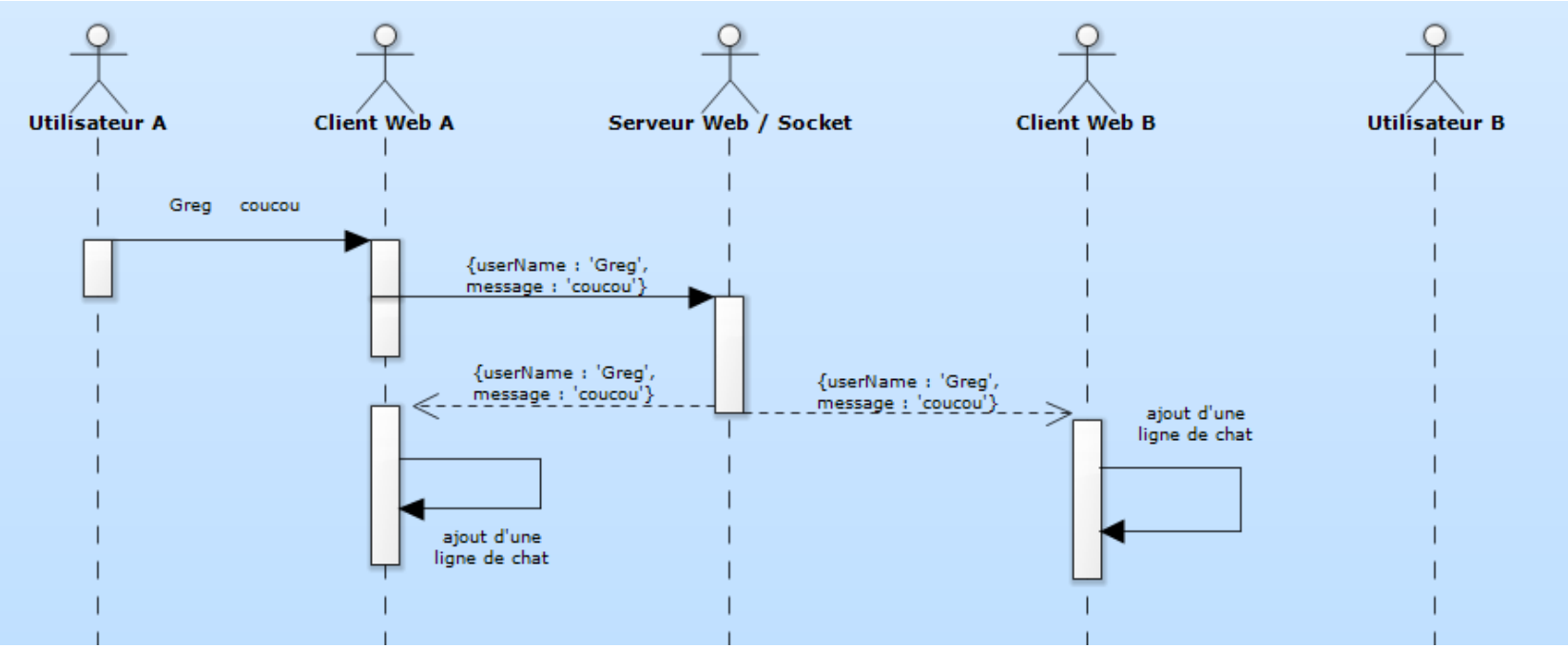


Dans le cas d'un retour formulaire, que la requête soit de type POST ou GET, il y a rechargement de la page.



Mise en place dun socket : communication asynchrone

L'installation d'un socket-io permet de créer un canal de communication utilisable dans les deux sens :
client -> serveur et serveur -> client



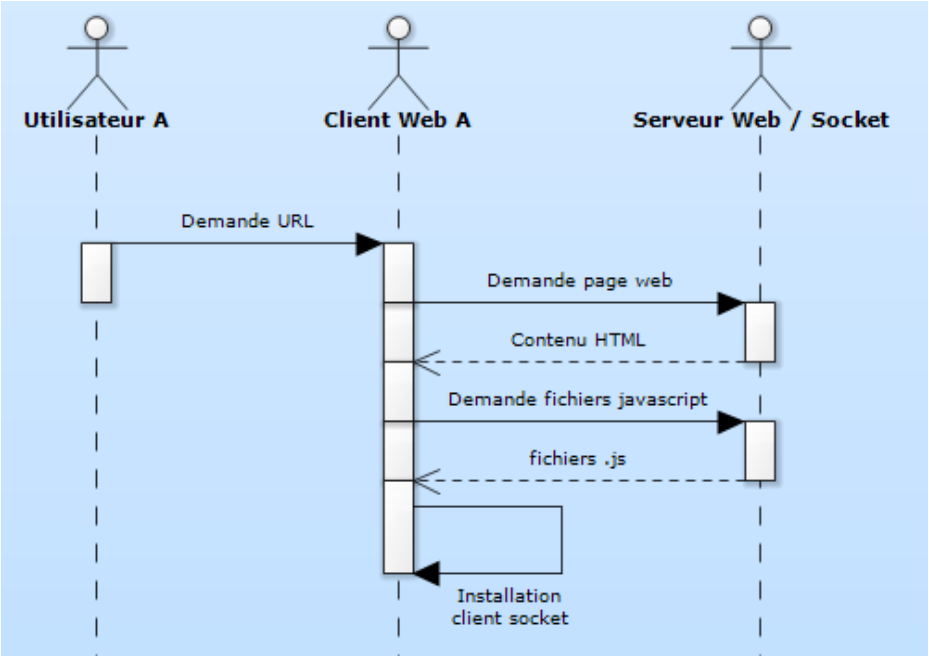
Le client web B reçoit un message du serveur sans avoir rien demandé. Les pages web affichées sur les deux clients sont mises à jour sans être rechargées.

Application : un chat

Client web A envoie un message à l'aide d'un formulaire :

Client web B reçoit également le message sans avoir rien demandé :

Connexion au serveur socket



Côté HTML :

```
<!-- Le navigateur doit télécharger le script suivant pour faire fonctionner la socketio. -->
<!-- Il est disponible en ligne avec le lien suivant. -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/1.7.3/socket.io.min.js"></script>
<!-- Charge le script js pour se connecter et dialoguer avec la socket   -->
<script src = "/static/main.js" > </script>
```

Dans le script `main.js` :

```
//Se connecte au serveur socket io
var socket = io.connect('http://' + document.domain + ':' + location.port);
```

Émission du message

Côté client A : Deux champs et un bouton . . .

```
<input id="in1" type="text" placeholder="User Name"/>
<input id="in2" type="text" placeholder="Message"/>
<button type="button" id="bpEnvoi"> Envoi </button>
```

. . . le script JS lit les champs après appui sur le bouton . . .

```
function envoiMessage() {

    //recupération des champs de saisi du formulaire
    let nom = document.getElementById('in1').value;
    let mess = document.getElementById('in2').value;
    //initialise le champ message
    document.getElementById('in2').value = ""
    //donne le focus au champ message
    document.getElementById('in2').focus();
    //
    socket.emit( 'my event', {userName : nom, message : mess})

}

document.getElementById("bpEnvoi").addEventListener('click', envoiMessage);
```

. . . et les envoie à travers le socket au serveur.

Émission du message : variante

Côté client A : Un formulaire en html...

```html
<form id="myForm" >
  <input id="in1" type="text" placeholder="User Name"/>
  <input id="in2" type="text" placeholder="Message"/>
  <input type="submit"/>
</form>
```

... intercepté par un script JS ...

```javascript
//evenement submit du formulaire de saisi des messages
function repFormulaire(evt) {
    //bloque l'envoi du formulaire
    evt.preventDefault();
    //recupération des champs de saisi du formulaire
    let nom = document.getElementById('in1').value;
    let mess = document.getElementById('in2').value;
    //initialise le champ message
    document.getElementById('in2').value = ""
    //donne le focus au champ message
    document.getElementById('in2').focus();
    //
    socket.emit( 'my event', {userName : nom, message : mess})
}

document.getElementById('myForm').addEventListener('submit', repFormulaire);
```

... et envoyé à travers le socket au serveur.

Vu du serveur

Le message est reçu par le serveur et reconnu par son identifiant, ici `'my event'`

```python
#callback sur evenement personnalisable
@socketio.on('my event')
def handle_my_custom_event(json):
    #json reçu est un dictionnaire
    print('Evènement reçu : ' + str(json))
    #envoyé avec identifiant 'ma reponse'
    #a tous les clients connectés !
    socketio.emit('ma reponse', json)
```

Il est renvoyé tel quel à tous les clients avec l'identifiant `'ma reponse'`

Réception du message

Coté client A et client B : une zone de message en html

```
<h2 id="initMess">  Pas encore de message ...</h2>
<div id="zoneMessage"></div>
```

mise à jour sans rechargement de la page par le script JS.

```
//evenement message reçu par le socketion
function reception( msg ) {
    //reception d'un message au travers de la socketio
    console.log( msg )

    //supprime la zone de titre qui indique "pas de mess reçu"
    let titre = document.getElementById("initMess");
    titre.textContent=""

    //Affiche le message reçu
    //pointe la division où saisir le message
    let divChat = document.getElementById('zoneMessage');
    //ajoute une nouvelle ligne
    let liSupp = divChat.appendChild(document.createElement('p'));
    //y place le message .innerHTML permet de placer des balises inlines dans le texte
    //ici pour mettre en gras le nom et noir
    liSupp.innerHTML = '<b style="color: #000">'+msg.userName+'</b> '+msg.message;
}

socket.on( 'ma reponse', reception);
```

Pour aller plus loin

Le serveur compte les connexions et met à jour les pages par le socket.

```
@socketio.on('connected')
def connected():
    print ('Nouveau client connecté')
    global nbClients
    nbClients += 1
    socketio.emit('nbClients', nbClients)

@socketio.on('disconnect')
def disconnect():
    print ("Client déconnecté !")
    global nbClients
    nbClients -= 1
    socketio.emit('nbClients', nbClients)
```

Le script JS se connecte explicitement au socket

```
socket.on('connect', function() {
    socket.emit('connected');
});

//mise a jour nombre de participant
socket.on( 'nbClients', function( msg ) {
    //on met a jour le champ correspondant
    console.log(msg)
    if (msg > 1){
        document.getElementById('nbClients').textContent= msg + " participants au chat !";
    }
    else {
        document.getElementById('nbClients').textContent= msg + " participant au chat !";
    }
})
```

et reçoit un message contenant le nombre de participants connectés.

Encore plus loin

Affichage de la durée du chat :

Dés qu'un client se connecte, l'heure de début de la session de chat est enregistrée.

```
@socketio.on('connected')
def connected():
    print ('Nouveau client connecté')
    global nbClients
    global flagChat
    global memoTime
    nbClients += 1
    socketio.emit('nbClients', nbClients)
    if nbClients == 1:
        flagChat = True
        memoTime = time.time()
```

Multi thread

Pour mesurer le temps en même temps qu'on attend les requêtes clients, Il est nécessaire de lancer le serveur dans un thread :

```
#lancement du serveur dans un thread
p = Thread(target = socketio.run, args = ((app, adressIp, 5000)))
p.start()
while True:
    time.sleep(1)
    if flagChat:
        socketio.emit('timeChat', floor(time.time() - memoTime))
```