**Créer des applications Web avec Flask : mode d’emploi.**

***Ce document est une prise de notes sur le MOOC « Créez vos applications web avec Flask », disponible à l’adresse suivante :*** [**http://sdz.tdct.org/sdz/creez-vos-applications-web-avec-flask.html#CrezvosapplicationswebavecFlask**](http://sdz.tdct.org/sdz/creez-vos-applications-web-avec-flask.html#CrezvosapplicationswebavecFlask)**.**

Site web = programme exécuté sur un ordinateur. Quand on se rend sur ce site web, on est client et l’ordinateur où le site web est exécuté est appelé le serveur (n’importe quelle machine peut faire office de client ou de serveur). On peut très bien être client et serveur sur une même machine (utile pour notre apprentissage). Une fois que le site sera prêt, on pourra le mettre sur un serveur accessible au monde entier.

Différence entre un serveur et un client : logiciel utilisé pour la communication entre client et serveur. Client utilise **navigateur web** (comme Firefox ou Chrome), tandis que serveur utilise un **serveur HTTP** (comme Apache). Un serveur http peut contenir plusieurs sites web et fournir des pages web à plusieurs clients à la fois.

Comment clients communiquent avec serveur web ? 🡪 **protocole http** (idée : client demande une page web et serveur lui renvoie)

Requête du client contient :

* Un **chemin** vers la page web
* Un **type**(ex : GET et POST, POST est utilisé pour la validation d’un formulaire)
* Informations supplémentaires sur type de navigateur web
* Des données (par exemple quand on remplit un formulaire)

Réponse du serveur contient :

* Page demandée
* Type de la page demandée = le mimetype : le plus souvent, le mimetype indique que la page est un page normale, soit contenant du HTML mais parfois ça peut renvoyer autre chose comme un image, un fichier ou un pdf
* Code d’erreur (même quand tout se passe bien, on renvoie un code d’erreur : code 200 signifie que tout s’est bien passé)

Pour exécuter le serveur http, il faut installer Apache mais ce n’est pas suffisant car **les serveurs http ne savent pas interpréter le Python**. Il faut donc installer un module pour rendre possible l’utilisation de Python : c’est Flask.

Plusieurs façons d’intégrer Python dans un serveur web.

Il existe une norme WGSI (respectée par Flask) et il existe des modules pour que les principaux serveurs http désservent des sites web en Python en respectant la norme WGSI.

Ce module sous Apache est **mod\_wgsi**. Mais pour nous, on va simplement utilisé le serveur http+wgsi intégré à Flask, donc on pourra immédiatement exécuter notre code Python.

Lien entre requête et adresse entrée dans navigateur web pour accéder à une page web :

Exemple : <http://www.siteduzero.com/forum-81-407-langage-python.html>

* http : protocole utilisé, car on utilise un serveur http
* [www.siteduzero.com](http://www.siteduzero.com) : nom de domaine
* [/forum-81-407-langage-python.html](http://www.siteduzero.com/forum-81-407-langage-python.html) : chemin de la page web demandée, le même que celui rentré dans la requête http

Dans notre formation, le serveur http utilisé sera notre propre ordinateur. Nom de domaine de notre propre ordinateur est appelé localhost par convention. Pour accéder à une page de notre serveur http, on devra donc rentrer dans notre navigateur web <http://localhost/une_page>

Attention : un serveur http peut fonctionner sur différents ports donc il faut le préciser dans l’adresse saisie dans le navigateur. Le serveur http intégré à Flask fonctionne sur le port 5000 donc il faut mettre : <http://localhost:5000/une_page> . Si aucun port précisé, le navigateur va automatiquement sur le port 80 sur serveur

Présentation de Flask :

Flask ne fait pas tout et nécessite certaines extensions pour certaines applications.

Flask permet de créer beaucoup plus rapidement une page web que sans framework.

Avec Flask, on peut réaliser des sites web dynamiques (!= site figé), c’est-à-dire un site qui personnalise la page en fonction du visiteur et de ses actions.

On va installer Flask grâce à la commande pip, et il nous faut Python en version 2.7 car Flask supporte pas Python en version3.

On installe donc Python en version 2.7.3 avec le fichier [Windows x86-64 MSI installer](https://www.python.org/ftp/python/2.7.3/python-2.7.3.amd64.msi) disponible sur <https://www.python.org/downloads/release/python-273/>

Il faut maintenant télécharger easy\_install

Quand on veut demander un truc sur shell à Windows, ouvrir Invite de commandes (ou taper juste cmd dans la barre de recherche Windows)

J’ai désinstallé Python 2.7.3 car n’arrive pas à installer easy\_install pour installer pip et donc flask.

Commande pip : permet d’installer facilement des librairies

**Solution** pour installer flask :

Ouvrir Pycharm Edu

Ctrl+alt+s pour ouvrir la fenêtre de settings

Chercher Project Interpreter

Double clic sur pip

Install Package

Puis de retour dans la fenêtre des différents modules, double clic sur flask

Faire install package : flask est maintenant installé !!

Quand j’exécute le code, j’ai une erreur rendue par Flask : WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.

Cela est dû au fait qu’on lance le script dans l’environnement de développement, il faut juste le lancer dans l’invite de commandes Windows avec la commande : python Test.py avec Test le nom du fichier Python

Analyse du code :

*#! /usr/bin/python  
# -\*- coding:utf-8 -\*-***from** flask **import** Flask  
app = Flask(\_\_name\_\_)  
  
@app.route(**'/'**)  
**def** index():  
 **return "Hello !"  
  
if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 app.run(debug=**True**)

Flask est un classe d’objet

App est un objet : c’est le site web, ou en termes plus techniques, une application WSGI. Le paramètre \_\_name\_\_ est utile quand on a plusieurs applications WSGI

Ligne 12, on lance l’application en mode debug : permet de détecter les erreurs dans le code + permet, si une modification est apportée au code Python, de le mettre direct à jour dans le navigateur web : pas besoin de refaire un python Test.py dans l’invite de commande

Attention : ne pas laisser le debug lorsque le site sera disponible sur Internet !!!

Pour sécuriser les sessions des visiteurs, on peut configurer la clé secrète de l’objet app avec la commande : app.secret\_key=mot\_de\_passe

Lorsque la méthode run() de app est lancée, le serveur http+wgsi de Flask est automatiquement lancé

Fonction index() : cœur du code ; on l’a décoré avec le décorateur @app.route, qui prend en paramètre une route, par laquelle notre fonction sera accessible. La route ‘/’ représente la racine du site web donc pas besoin de la préciser dans l’adresse du navigateur.

Qu’est-ce qu’un décorateur ? Fonction qui modifie le comportement d’autres fonctions : utile quand on veut ajouter du même code à différentes fonctions existantes

Def : vue : fonction qui renvoie une page web 🡪 chaque @app.route est une vue

Attention : les définitions de pages web doivent être écrites avant le run finale de l’app sinon on aura une erreur 404 error not found

Tip : Flask peut rajouter un slash manquant mais pas enlever un slash en trop

Si on introduit une erreur dans le code, le debugger intégré à Flask renvoie une page d’erreur. Si on saisit dans le terminal dump(app), on obtient tout ce qu’on sait (méthodes, attributs) à propos de l’objet en question, ici app. Si on avait pas installé le debugger, on aurait eu une erreur 500, qui signifie qu’on a une erreur sur le serveur

Si erreur quand on lance pycharm et le serveur web, refaire un **run dans pycharm** et un **python Test.py** dans l’invite de commandes

Objet **request** à importer de flask : représente la requête http envoyée par le client et reçue par le serveur : on trouve dans cet objet le chemin, le type, les infos client et les données transmises de la requête. Pour accéder au path de la requête, on tape request.path

@app.route('/')

def racine():

return "Le chemin de 'racine' est : " + request.path

@app.route('/la')

def ici():

return "Le chemin de 'ici' est : " + request.path

Type de la requêtes http : plus connu sous le nom de « méthode http » (par ex. get et post, lorsque l’utilisateur rempli un formulaire). Voyons l’intérêt d’accéder à ces méthodes http. Imaginons une page demandée par une client sur laquelle il faut renseigner des données dans un formulaire, et une fois le formulaire rempli, que la page nous revoie « merci d’avoir rempli le formulaire ». On regarde les méthodes employées : si c’est un GET, on affiche le formulaire et sinon (donc si c’est un post, après la saisie des données) on traite les données et on renvoie le message de remerciement.

@app.route('/contact', methods=['GET', 'POST'])

def contact():

if request.method == 'GET':

# afficher le formulaire

else:

# traiter les données reçues

# afficher : "Merci de m'avoir laissé un message !"

Ou sinon, en veillant bien à ce que **deux routes soient distinctes** dans le code (ce qui est bien le cas dans le code ci-dessous car les méthodes utilisées pour accéder aux pages sont différentes).

@app.route('/contact', methods=['GET'])

def contact\_formulaire():

# afficher le formulaire

@app.route('/contact', methods=['POST'])

def contact\_traiter\_donnees():

# traiter les données reçues

# afficher : "Merci de m'avoir laissé un message !"

Ici, les méthodes http d’accès à une vue permettent de filtrer les requêtes.

La méthode est donc accessible via l’attribut **method**. On fait un **request.method** pour y accéder.

Remarque : on peut décorer plusieurs fois une vue avec @app.route pour rendre accessible une page par différentes adresses. Pas très utile de faire ça, sauf pour les **routes personnalisées**, qui permettent d’avoir de belles URL.

Exemple : pour obtenir un URL en fonction du nombre de messages postés sur une page

Quand il y a trop de messages postés sur une page, on limite le nombre de message par page et on ajoute des boutons page suivante et page précédente dans notre site web. Le **<num\_page>** est un paramètre obligatoire dans le nom de la route. Il faut pour cela mettre des < > autour du nom de variable et ajouté en paramètre à la vue décorée une variable qui possède ce nom.

@app.route('/discussion/page/<num\_page>')

def mon\_chat(num\_page):

num\_page = int(num\_page)

premier\_msg = 1 + 50 \* (num\_page - 1)

dernier\_msg = premier\_msg + 50

return 'affichage des messages {} à {}'.format(premier\_msg, dernier\_msg)

Mais attention ! Dans l’exemple précédent, num\_page est un string, donc c’est pour ça qu’on met int() à la ligne 3. On peut le déclarer comme int directement dans la route en faisant **<int:num\_page>.** Si variable saisie dans l’adresse n’est pas un int, elle ne sera pas acceptée et on aura une erreur 404. On peut aussi avoir les types **float** et **path** pour les chaînes avec des slash.

Si on veut accéder aux 50 messages les plus récents, on peut faire :

@app.route('/discussion')

@app.route('/discussion/page/<int:num\_page>')

def mon\_chat(num\_page = 1):

premier\_msg = 1 + 50 \* (num\_page - 1)

dernier\_msg = premier\_msg + 50

return 'affichage des messages {} à {}'.format(premier\_msg, dernier\_msg)

Cela rajoute une **valeur par défaut du paramètre** dans la vue.

Jusqu’ici, on utilisait des mimetype (type de page) qui était du texte : le mimetype était donc text/html. Pour renvoyer une page qui affiche une image telle quelle, on utilise le module PIL (Python Image Library), qu’on peut installer via pip install PIL. Attention, PIL ne fonctionne pas mais on peut installer Pillow à la place. On **importe Image** de PIL ensuite.

Attention : fichier Python stocké dans PyCharm Projects

Je saute la partie changer le mimetype car je n’arrive pas à importer correctement PIL

Passer direct à la partie : les redirections

(22h-00h)