**Notes en vrac : étude de cas Arcane.**

**Ce document regroupe les notes prises en vrac durant cette étude de cas. Un ctrl+F permettra de retrouver aisément la notion recherchée.**

Définition des termes inconnus :

* API = interface de programmation applicative : ensemble normalisé de classes, de méthodes et de fonctions qui sert de **façade** et par laquelle un logiciel offre des services à un autre logiciel : sorte de **brique de fonctionnalité**, qui peut servir à d’autres logiciels

Donne une solution à un problème informatique en faisant abstraction de son fonctionnement

* API REST = API representational state transfer

RESTful : adjectif désignant une API REST

API REST basée sur HTTP (Hypertext Transfer Protocol), protocole définissant la communication entre les différentes parties du web

* Framework : ensemble cohérent de composants logiciels structurels : sert à créer les grandes lignes d’une partie d’un logiciel : modèle l’architecture d’un logiciel
* Flask : framework open source de développement web en Python

(15h-18h30 : révisions notions de protocole HTTP)

(18h30-20h : tuto Git)

Installation de Flask :

Saisie de la ligne de code : pip install -U Flask dans Anaconda Prompt

Explication Web : <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 utiliser 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

Dépôt du code sur Git :

Repository = répertoire

Cf compte rendu étude de cas Sysnav pour le tuto sur Git

Langage **CSS** : pour la **mise en forme** de document web

Il faut se mettre à l’html.

PHP : (Hypertext Preprocessor) : langage de programmation pour produire pages web dynamiques

HTML : langage de balises

Tuto sur : <https://tutorialehtml.com/fr/html-tutoriel-html-complet/>

D’abord, on fait un guide débutant : <https://tutorialehtml.com/fr/guide-debutants-en-html/>

Pour **ouvrir un fichier html** qui est sur notre ordinateur, on **copie-colle le path** sur un navigateur web comme chrome

Mots entourés de < et > = **balises**: spécifie les régions d’un doc html

Base d’un code html :

<html>

<body>

Le contenu de la page internet ira ici

</body>

</html>

* **<html>** : indique au navigateur le début d’un code html
* **<head>** : se met immédiatement après html, sert à indiquer au navigateur le titre de la page, son contenu et d’autres infos : c’est ce titre de page qui apparaît dans la nom de l’onglet sur Chrome ; on peut mettre du Javascript ou du CSS dans le head
* **<title>** : se situe entre les deux balises head : donne le nom de la page
* **<body>** : indique où le contenu de la page internet commence
* **<h2>** : se situe après <body> : titre qui apparaît dans la page, avant le contenu : plus gros que le caractère du contenu : il existe les balises <h1>, …, <h6> pour des tailles différentes de textes avec <h1> le plus gros caractère
* **<p>** : ouverture d’un paragraphe, qui se finit avec </p> : saute automatiquement une ligne au début et à la fin du paragraphe

Il faut toujours indiquer au navigateur l’arrêt d’une fonction **: on ferme la balise** <balise> avec un / donc avec la balise **</balise>.** Bien respecter l’**ordre** d’ouverture et de fermeture des balises.

Defs :

* **balises**: spécifie les régions d’un doc html, commande que la navigateur exécute, a 3 parties : ouverture, contenu, fermeture
* **élément**: balise complète, qui possède une ouverture <tag> et une fermeture </tag>
* **attribut**: change et personnalise la valeur d’un élément en html

Remarque : toutes les balises sont écrites en petites lettres

Attention : certaines balises n’ont pas de contenu donc ne respectent pas le modèle ouverture-contenu-fermeture, et combinent les deux balises d’ouverture et de fermeture

Exemple : **<br/>** pour le **saut de ligne**

Les **attributs** personnalisent les balises : ils servent par exemple à changer la couleur de la police, ou encore de changer la taille d’une image. Ils servent à mettre en forme un site internet.

Exemple d’attributs génériques, qui s’utilisent avec quasiment toutes les balises :

* **class** et **id**: étudiés plus tard ; associés à p par ex

<p id="italicparagraph">Paragraphe type 1, italique </p>

* **name**: si on donne un nom à un élément, l’élément devient une variable de script pour Javascript, ASP et PHP. S’utilise beaucoup dans les champs de texte interactif comme input :

<input type="text" name="TextField" /> : pas d’effet sur l’apparence de la zone de texte

* **title**: ajoute un titre pop-up à chaque contenu de l’élément : la visualisation apparaît quand on arrête notre souris au-dessus du texte : s’utilise avec la balise <h2> notamment
* **align**: aligne un titre au à droite, gauche ou au centre de la page selon la valeur de l’attribut

Les balises ont des attributs standards (par défaut).

A partir de paragraphe, je saute tout jusqu’à ce qui m’intéresse : les formulaires.

Un **formulaire** est un ensemble de champs de texte

**Champs de texte** : permettent à l’utilisateur d’entrer du texte, ils ont quelques attributs (la balise est **form**)

* type : type de champ de texte, comme texte, envoi ou mot de passe
* name : donne un nom au champ pour s’y référer ultérieurement
* size : taille du champ
* maxlength : valeur max du nombre de caractères pouvant être tapés

(données encryptées = sécurisées)

On veut ajouter le **bouton d’envoi**, qui a l’attribut Send ou Submit et qui doit être en dernier dans le formulaire.

Destination pour les données entrées dans le formulaire ainsi que le mode de transfert doivent être spécifiés à travers les arguments suivants :

* **method**: on utilisera la méthode post, qui envoie le formulaire avec l’info insérée sans envoyer données de l’utilisateur
* **action**: URL où l’information est envoyée

Exemple :

<form method="post" action="mailto:votreemail@email.com">.

Name: <input type="text" size="10" maxlength="40" name="name" /><br />

Password: <input type="password" size="10" maxlength="10" name="password" /><br />

<input type="submit" value="Send" />

</form>

Ici, le **action=« mailto:thibault.latrace@gmail.com »** fait que quand on valide le formulaire, ça ouvre notre boîte de mail avec comme destinataire déjà le [thibault.latrace@gmail.com](mailto:thibault.latrace@gmail.com).

Tuto pour gérer les données une fois qu’elle ont été envoyées par l’utilisateur : <https://developer.mozilla.org/fr/docs/Web/Guide/HTML/Formulaires/Envoyer_et_extraire_les_donn%C3%A9es_des_formulaires>.

Le **action** définit où envoyer l’information saisie par l’utilisateur quand le formulaire est envoyé.

On peut les envoyer à une **URL précise** comme http://foo.com : **<form action="http://foo.com">** ou à une **URL relative** /somewhere\_else : **<form action="/somewhere\_else">**.

S’il n’y a pas d’attribut action, les données sont envoyées à la même page que le formulaire.

On peut utilisé une URL https (=http sécurisé) : les données sont alors chiffrées avec le reste de la requête, même si le form est sur une page http. Mais si formulaire est sur page sécurisée et que URL de action est non sécurisée, navigateur affiche une alerte de sécurité.

J’ai réinstallé Oracle SQL Developer

On va utiliser Flask non pas pour générer un code HTML mais pour créer un fichier au format JSON pour créer une API : données brutes interprétables par un programme.

Format JSON (Javascript Object Notation) : format de données textuelles, qui dérive de la notation des objets du langage Javascript.

Avec ce genre d’API, le serveur répond par des données brutes et non pas par de l’HTML.

Dans une API, on appelle chaque URL interrogeable un **endpoint**.

Idée : comme le format JSON se rapproche beaucoup d’un dictionnaire en Python, on peut facilement transformer un dico en doc JSON (puis le renvoyer en requête http), avec la fonction **jsonify** de Flask.

On va essayer de tester une API avec Openweathermap.org

Pour résumer, on code une API Python qui reçoit les requêtes du client (=navigateur). Ce client attend du prgramme Python qu’il renvoie les données météo. Le programme interroge alors l’API Openweathermaps : le programme Python devient alors client d’Openweathermaps.

Fausses données JSON d’Openweathermap qu’on peut utiliser pour s’entraîner : <https://samples.openweathermap.org/data/2.5/forecast?lat=0&lon=0&appid=xxx>.

Sur Openweathermap, compte associé à une clé (dans API keys), à fournir à chaque requête de l’API

Quand le client appelle l’endpoint ‘/api/meteo’, Python enverra une requête vers l’API Overweathermap

Les prévisions météo se trouvent dans la clé « list ».

On abandonne l’idée, car nous on ne traite pas de vraies données : on veut juste une application rest.

Tuto API REST :

<https://www.youtube.com/watch?v=UQwjytQzoqE>

6 règles d’un système REST :

* séparation client-serveur (client concomme service, serveur l’offre)
* stateless : chaque requête d’un client doit contenir toutes les infos recquises pour que serveur réponde à la requête : serveur peut pas stocker d’infos d’une requête et les utiliser pour une autre requête
* cacheable : serveur doit dire au client si les requêtes sont aptes à être mis en cache ou pas (réponse cacheable = réponse http stockée pour être récupérée plus tard, en entrant une nouvelle requête sur le serveur)
* layered system : des serveurs intermédiaires peuvent répondre à la place du serveur principal
* uniform interface
* code on demand (optional) : serveurs peuvent fournir le code au client pour qu’ils l’utilisent dans leur propre contexte

L’API REST interagit avec une base de données et à un serveur web.

En gros, l’application demande toujours « Que voulez-vous ? ».

Exemple : choix entre **ajouter** un vol, **supprimer** un vol, **modifier** un vol, **lister** les vols dispos, **vérifier** si données saisies sont correctes

Cas spécifique où site internet veut prendre les données d’une application REST : l’api envoie les données au site internet

Si site internet demande d’ajouter un vol, l’API le fait.

**Le client** qui se connecte à l’API REST est le **site internet**, qui le fait au travers d’une requête http à un endpoint.

En gros l’appli REST crée un lien entre un site internet, une application iPhone, une appli Android ou autre.

Si erreur dans la requête du site internet, l’api renvoie un message d’erreur. Si pas d’erreur, donne la liste des choses disponibles.

Dans la partie code, l’api rest envoie une variable « success »(savoir si la requête a abouttie ou non ie si pas d’erreur), une autre « msg »(qui correspond au message d’erreur) et une autre « result »

Le result est un JSON.

L’API REST est une **URL** qui renvoie des informations.

Tuto rapide : <https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask>

Les data données par une requête d’une API REST sont souvent sous format **JSON**.

Les ressources sont représentées par des **URI** (uniform resource identifier), auxquels les clients envoient des requêtes.

Attention : toutes les URL sont des URI mais pas l’inverse. L’URL est une info pour localiser un élément alors qu’un URI permet d’identifier une ressource

Quand on veut faire un application, on met le nom de l’APIdans l’URL pour pas confondre les URL si on a plusieurs API sur le serveur : <http://[hostname]/todo/api/v1.0/>. Mettre la version dans l’URL permet de savoir avec quelle version de l’API on travaille (nouvelles fonctions par exemple).

Définir les ressources qui vont être utilisées dans l’API. Exemple : dans une API to do list, où on veut faire une liste de tâches à effectuer, les seules ressources seront les tâches.

Il faut faire un tableau avec les différentes méthodes http utilisées, à quels URI elles sont adressées et pour effectuer quelle action.

Ici, on va pas utiliser de database dans un premier temps mais juste dans une structure de liste de dictionnaires sur Python, qui seront ensuite jsonifiées.

Attention : utiliser un navigateur web pour tester un web service n’est pas une bonne idée car ils ne peuvent pas générer tous les types de requêtes http. On installe **curl** à la place (direct dans le projet python). On saisit alors

curl -i http://localhost:5000/todo/api/v1.0/tasks

dans anaconda. Ca permet d’accéder à la liste de tâches de notre todolist.

Note : par défaut, une erreur 404 sera renvoyé en HTML par Flask. Mais on veut ici que notre web service renvoie uniquement des JSON. Donc on fait

@app.errorhandler(404)

def not\_found(error):

return make\_response(jsonify({'error': 'Not found'}), 404)

Pour tester si notre fonction avec la méthode POST fonctionne bien , on tape dans anaconda :

curl -i -H "Content-Type: application/json" -X POST -d '{\"title\":\"Read a book\"}' <http://localhost:5000/todo/api/v1.0/tasks>

OU

curl -i -H "Content-Type: application/json" -X POST -d "{\"title\":\"Read a book\"}" <http://localhost:5000/todo/api/v1.0/tasks>

Attention : mettre des **antislashs \** devant les « » pour bien que Windows le reconnaisse.

Rappel : PUT est utilisé pour mettre à jour une donnée et DELETE pour la supprimer

Si on veut changer la valeur d’une tâche en True :

curl -i -H "Content-Type: application/json" -X PUT -d '{\"done\":true}' http://localhost:5000/todo/api/v1.0/tasks/2

Le request.json lit les données comme si c’était un JSON

Quand on fait request.json.get : **get**(name, default=None)[¶](https://flask.palletsprojects.com/en/1.1.x/api/#flask.ctx._AppCtxGlobals.get) avec

* **name** – Name of attribute to get.
* **default** – Value to return if the attribute is not present.

Attention : par défaut, ce qu’on met dans un URL est un str : il faut préciser quant c’est un int avec **/<int :id>.**

Maintenant, on se focalise sur la partie sécurité : quel utilisateur peut saisir quelle data ?

On demande aux utilisateurs de saisir un username et un mot de passe. On peut pas utiliser de cookies ici car ça violerait le principe stateless de l’API REST. Il faut donc toujours demander au client d’envoyer leurs informations d’authentification dans chaque requête qu’ils envoient.

Pour implémenter l’authentification, on install avec pip flask-httpauth **A SAISIR DANS LE TERMINAL DU PROJET !**

Dans toutes les fonctions qui nécessitent d’être protégées, on ajoute le décorateur @auth.login\_required

Pour accéder à la page, il faut alors s’authentifier avec la commande :

Curl -u thibault:python -i http://localhost:5000/real\_estate/api/v1/housing

Tuto : <https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask>

Pour voir comment lier database avec Flask, lire : <http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>

Tuto pour base de données relationnelles (SQL et non pas NoSQL)

On va utiliser une extension Flask : **Flask-SQLAlchimy**, qui est un **ORM** (object-relational mapping : permet aux applications de gérer des BDD. Cet ORM fonctionne avec **PostgreSQL**.

On installe sqlalchemy avec la commande pip install flask-sqlalchemy dans Anaconda.

Deuxième extension de Flask qu’on va utiliser : **flask-migrate** : permet que quand structure de la DB change, les data soient de suite redirigées, « migrées » sur la structure modifiée. Flask-migrate est un wrapper de Alembic, qui est lui-même un framework de migration de base de données pour SQL Alchemy

Def **: wrapper** : programme dont la fonction principale est d’appeler une autre fonction

On utilise ici SQL Lite (moteur de base de données relationnelle), utile pour gérer des DB pour de petites applications car chaque BD est stockée dans un fichier unique sur le disque et pas besoin d’utiliser un serveur de BD comme PostgreSQL.

Implémentation de la BDD selon l’arborescence suivante :

microblog/

venv/

app/

\_\_init\_\_.py

routes.py

microblog.py

(on peut retrouver les fichiers suivants dans test\_database\_API)

Ensuite, saisir dans le terminal : flask db init

Mettre la variable d’environnement FLASK\_APP à FLASK\_APP=API.py avec API.py le fichier python où il y a from app import app.

Pour créer des migrations automatiques, saisir dans le terminal :

flask db migrate -m "users table"

Le -m sert à commenter la migration.

Attention : la commande flask db migrate ne fait pas de changement sur la database : ça génère juste le script de migration. Pour l’appliquer, saisir :

flask db upgrade

Après cette commande, dans le fichier on a maintenant **app.db**, qui est notre database SQLite. Elle est créée automatiquement car SQLite détecte que la table n’existe pas et la crée automatiquement.

Idée : quand plus tard dans notre développement, on veut ajouter une table à la database par exemple (et donc la modifier). Grâce à flask-migrate, on crée notre classe dans **models** puis on génère un nouveau script avec **flask db migrate** puis on applique ces changements à notre database avec **flask db upgrade**.

Remarque : pour chaque model (les classes du programme Python models), si la classe s’appelle User, SQAlchemy l’appellera directement user.

Pour annuler la dernière migration, on peut utiliser la commande **flask db downgrade**, qui annule la précédente migration. On doit alors supprimer le script Python de la migration qu’on a supprimer, puis on en refait une nouvelle.

Dans notre exemple, après avoir ajouté la table des posts, faire une migration grâce à flask db migrate -m "posts table".

Quand tout est fait, on peut exécuter dans la console Python :

>>> from app import db

>>> from app.models import User, Post

Puis on crée un nouvel utilisateur :

>>> u = User(username='john', email='john@example.com')

>>> db.session.add(u)

>>> db.session.commit()

Les changements effectués sur une base de données sont effectuées lors d’une session, à laquelle on peut accéder grâce à **db.session**.

Une fois que tous les changements ont été effectués dans la session, on peut faire un **db.session.commit()**, qui effectue tous les changements automatiquemennt.

Si on fait une erreur pendant la session, on peut faire la commande **db.session.rollback**() qui quittera la session sans effectuer tous les changements. Dès qu’on reçoit une erreur, on l’utilise et on refait une session.

Pour demander tous les users : **User.query.all()**

Tous les modèles (classe d’objet comme User et Post) de notre script ont un attribut **query** qui est la porte d’entrée à la lecture de requêtes de la BDD.

Si on connaît l’id d’un user, on peut avoir des infos sur cet user avec **u = User.query.get(1) (**si c’est le user d’id 1).

Si on veut ensuite ajouter un post à la DB, on fait :

>>> u = User.query.get(1)

>>> p = Post(body='my first post!', author=u)

>>> db.session.add(p)

>>> db.session.commit()

Tuto pour améliorer la sécurité des sessions : <https://flask-httpauth.readthedocs.io/en/latest/>

Retour au chapitre 1 :

Avec la commande python3 -m venv venv à saisir dans Anaconda, je demande à Python d’exécuter le package venv (virtual environment) (c’est le premier venv). Le deuxième venv est le nom du virtual environment que je vais utiliser.

Qu’est ce qu’un venv ? Permet d’avoir des installations de Python isolées de l’OS et séparées les unes des autres pour les projets.

Attention : il faut créer un environnement virtuel pour chaque projet Python.

Pour installer et utiliser un environnement virtuel :

* pip install virtualenv
* cd /path/Mon\_dossier\_projet\_Python
* virtualenv path/Mon\_dossier\_projet\_Python/venv\_du\_projet
* /path/Mon\_dossier\_projet\_Python /Scripts/activate.bat (rajouter --system-site-packages si on veut que tous les packages déjà sur notre ordi soit dans le venv)

On est alors dans notre environnement virtuel.

Après ça, si on installe une bibliothèque avec pip, il le fera dans l’environnement virtuel et la bibli ne sera pas accessible ailleurs.

On sort de l’environnement virtuel avec la commande deactivate. Un virtualenv prend un peu de place mais c’est mieux de travailler comme ça pour cloisonner les projets.

Pour mieux sécuriser les mots de passe, il faut les crypter

Tuto : <https://flask-httpauth.readthedocs.io/en/latest/>

**Erreur 403** : c’est quand le serveur comprend la requête mais n’autorise pas l’accès à l’utilisateur