TD : approche d'un framework de conception d'application Web Flask

Une application Web est un « logiciel » qui s'exécute dans le navigateur. Souvent, il y a une interaction forte avec un serveur Web. Ainsi, l'appli web peut être fractionnée en deux parties :

- la partie « client » ou « Front End » : la partie du programme qui s'exécute dans le navigateur (cette partie nécessite de concevoir en HTML, CSS et, le plus souvent, en Javascript)
- la partie « serveur » ou « Back End » : la partie du programme qui s'exécute dans le serveur. A ce niveau, les langages de programmation sont très diversifiés. Les plus connus sont Java et PHP, mais on y trouve de nombreux autres comme Python, Ruby, Javascript, C#… la liste n'est pas exhaustive.

Flask est un environnement de développement web (framework web) pour la partie serveur qui emploie le langage Python.
Un autre framework web basé sur Python existe, plus puissant et plus répandu : c'est Django.

Partie 0 : objectif du TD, prérequis, considérations pratiques

Ce TD a pour but de

- continuer à vous faire pratiquer quelques concepts d'élaboration de pages web et un peu de style CSS, surtout la partie « formulaires »
- vous faire découvrir quelques étapes simples du développement web sur le mode client-serveur
- vous faire apparaître des concepts clés du protocole HTTP, qui sert à l'échange de données entre le navigateur et le serveur web
- vous faire approcher quelques concepts de sécurité de la programmation web

Lors de ce TD, vous allez créer une interface d'authentification pour accéder à des pages web.

En saisissant un nom d'utilisateur reconnu et un bon mot de passe, vous pourrez accéder à sa page web. Sinon, ce ne sera pas possible.

Prérequis : avoir installé le module `flask` depuis l'installeur `pip` pour python :

```
pip install flask
```

La page web contenant le formulaire web d'authentification a déjà été en grande partie faite lors d'une séance passée.

Pendant le développement de l'appli, le serveur (Flask) et le client (le navigateur) vont tourner sur la même machine.
On va donc utiliser `localhost`, l'adresse IP spécialement destinée à faire communiquer au sein d'une même machine les programmes utilisant la couche TCP/IP. Celle-ci est `127.0.0.1`.

Flask va « écouter » les requêtes sur cette adresse (depuis le port 5000) et envoyer ses réponses sur la même adresse (vers un autre port, utilisé pour l'occasion par le navigateur).

Lorsqu'on veut un comportement plus proche de la réalité, le serveur et le client sur deux machines différentes, le principe reste exactement le même, sauf que les adresses IP sont « externes ».

Partie 1 : démarrage sur un schéma rudimentaire.

Téléchargez depuis moodle les fichiers d'amorce :
`premier_serveur.py`, `style_login.css`, et `login.html` et `olympe.html`

a) Le dossier de travail.
Une fois choisi son emplacement sur votre système de fichiers, on va fixer son nom `td_flask`.
Il doit contenir :

- un fichier source python, le programme du serveur. Ici nous l'appellerons `premier_serveur.py` . Dans un premier temps, ce sera le fichier téléchargé depuis moodle.
- un répertoire (dossier) `templates` (n'oubliez pas le `s` à la fin!). Dans ce répertoire, placez le fichier `login.html`. et le fichier `olympe.html`
- un répertoire (dossier) `static`. Dans ce répertoire, placez le fichier `style_login.css`.

b) Éditer les fichiers source : depuis votre éditeur texte (Geany) ouvrez les deux fichiers `premier_serveur.py` et `login.html`
Ne modifiez rien pour l'instant, mais on sera rapidement amené à le faire !

c) Démarrer Flask. Pour ce faire :

- recopier le chemin de votre dossier de travail `td_flask` depuis a barre d'adresse de votre explorateur de fichiers
- ouvrez `powershell`
- en vous servant de l'adresse copiée précédemment et de la commande `cd`, placez vous dans le répertoire de travail `td_flask`
- lancez Flask sur le programme python du répertoire par la commande :

```
> flask --app premier_serveur run --debug
```

- vous devez voir :

```
 * Serving Flask app 'premier_serveur'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server
 instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 741-896-017
```

- cela signifie que Flask exécute le programme `premier_serveur`, écoute sur `127.0.0.1:5000` et qu'il est en mode debug. Le mode debug est très pratique pour le développement. Sans lui, on devrait redémarrer le serveur à chaque modification des fichiers source et grâce à lui on a l'affichage des messages d'erreur dans le navigateur
- ouvrez votre navigateur web et saisir dans la barre d'URL : `127.0.0.1:5000`
- vous devriez voir :

Explications sur le programme du serveur :

premier_serveur2.py | login.html | premier_serveur.py

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route("/")
6  def coucou():
7      return "<h1>Salut ! Le serveur fonctionne !!</h1>"
```

Ligne 1 : on importe la classe `Flask` depuis la bibliothèque `flask`.
Ligne 3 : on crée un objet `Flask` que l'on place dans la variable `app`. Cet objet `Flask` va représenter le programme serveur et c'est pas son intermédiaire que l'on va accéder à ses fonctionnalités. Avec la notation « pointée » propre à la programmation objet (étudiée en terminale).
Ligne 5 : c'est ce qu'on appelle un « décorateur ». Simplement dit, on va associer cette ligne à la ligne suivante.
Ligne 6 : on définit une fonction `coucou` qui ne prend pas de paramètre et qui est associée à la « route » `"/"` grâce au décorateur précédent. Cette fonction se contente de renvoyer systématiquement une chaîne de caractères. On reconnaît en cette chaîne une déclaration en langage HTML.

La **route** est une façon de décrire comment le client va accéder au serveur, par quelle URL.
Ici, cette route `"/"` est la « racine » de l'appli. C'est celle qui correspond à l'url « vide » qui ne contient que l'adresse du serveur. Ici `127.0.0.1:5000`.

Ce n'est pas, comme sur les serveurs web basiques, la racine du répertoire contenant les pages web.

Lorsque le client (le navigateur web) demande par sa requête http la page d'URL `127.0.0.1:5000`, le serveur reconnaît la route racine `"/"` et exécute alors la fonction coucou. La chaîne de caractères retournée par coucou est alors envoyée au client par une réponse HTTP du serveur.

Si on essaie d'accéder à une autre page web éventuellement présente sur le serveur en complétant l'URL, le serveur ne trouve aucune fonction comme `coucou` associée à cette route. Il renvoie donc la page d'erreur 404 habituelle :

Partie 2 : accéder au formulaire à partir de la route racine

La première modification va consister à associer la route racine à notre formulaire d'authentification.

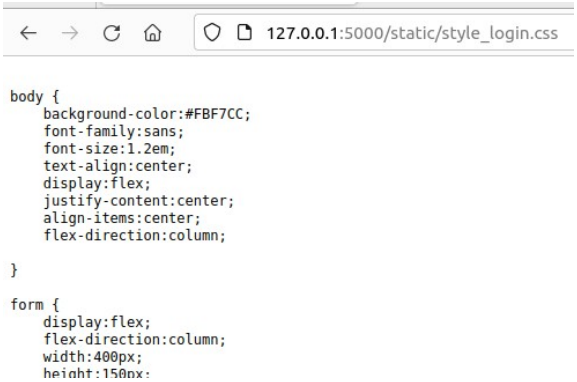
Rappel : ce formulaire est le fichier `login.html`. Il est enregistré dans le répertoire `templates`.

Le répertoire `templates` contient les templates web (patrons). Un template est une page web écrite en html mais contenant des éléments variables que le serveur modifie à la volée avant d'envoyer la page en réponse au client. Cela permet de facilement générer du code html personnalisé au client qui en a fait la requête.

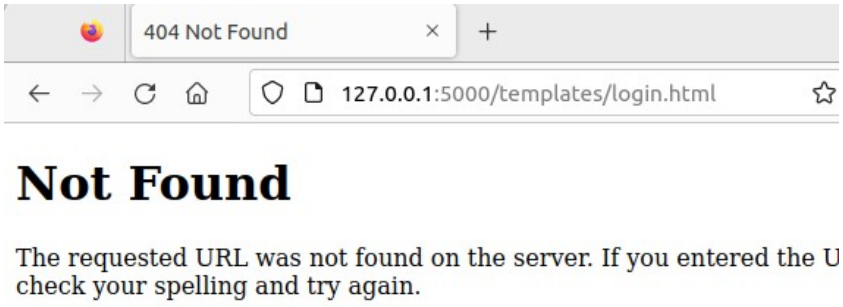
Le serveur utilise une fonction de « rendu » (`render_template`) qui va remplacer les variables de template par leur contenu au moment de la requête.

Lorsque le document ne contient pas ces variables de template, on dit qu'il est statique. Normalement, on range ces pages dans le répertoire `static`. C'est ce qu'on a fait de la page de style `style_login.css`.

Ici, notre page web de formulaire d'authentification est en réalité statique. Elle est une simple page web sans variable de template. Pourquoi l'a-t-on rangée dans le répertoire `templates` ? C'est pour des raisons de sécurité : les fichiers rangés dans `static` sont accessibles de façon libre depuis la barre du navigateur. Essayez d'accéder à la feuille de style, cela ne pose pas de problème :



Par contre, les fichiers rangés dans templates ne sont pas accessibles.



En fait, la route `/static` est prédéfinie implicitement par Flask. C'est la seule.

Donc ranger une page web statique dans `templates` permet de protéger son accès.

Ce n'est pas spécialement crucial pour notre formulaire d'authentification.
Une autre raison est qu'ainsi, on peut utiliser la route racine.
Sinon, il faudrait utiliser la route /static/login.html . C'est à dire l'url `127.0.0.1:5000/static/login.html`.
Ou faire une redirection…

La solution du template est donc plus simple et rapide.

Allons-y !

a) Dans le fichier source du serveur, il faut importer la fonction render_template depuis le module Flask. Rajoutez donc **`render_template`** à la fin de la ligne 1 :

```
(ligne1 : ) from flask import Flask, render_template
```

b) Dans le fonction `coucou`, qui est associée à la route racine, remplacez la chaîne retournée par `return` par un appel à `render_template` sur le template dont le nom est login.html :
`return render_template("login.html")`

c) Essayez dans le navigateur en rechargeant l'URL racine `127.0.0.1:5000` (de toutes façon, essayez toujours chaque modif dans le navigateur!!)

d) Vous voyez le formulaire mais aussi que le style n'est pas appliqué…. C'est parce que, dans le fichier html `login.html`, le chemin vers la feuille de style n'est pas juste. Le modifier. Rappel : `style_login.css` est dans `static` !

Partie 3 : retour du formulaire d'authentification

Le fait de valider un formulaire web déclenche en général une requête vers une URL. Cette requête contient d'une façon ou d'une autre les données saisies dans le formulaire par l'utilisateur.

Pour l'instant, la validation du formulaire par le bouton « submit » déclenche un comportement par défaut sur la même URL que la page et provoque une erreur 404 sur le serveur qui ne gère pas cette route.

Nous allons donc faire deux choses :

- décider d'une URL vers laquelle envoyer le formulaire (un raccourci pour dire « envoyer le contenu du formulaire rempli). Pour cela, le moyen usuel est de définir un attribut `action` dans la balise `<form>`.
- créer une route pour cette URL dans le serveur et prévoir une réponse.

C'est parti !

a) Dans `login.html`, rajouter dans la balise `<form>` (il n'y en a qu'une!) l'attribut `action="auth"`. Ce nom « auth » est un choix purement arbitraire. Grâce à cet attribut, lorsque le formulaire est validé par un clic sur le bouton submit, un appel vers l'URL `127.0.0.1:5000/auth` est effectué avec la valeurs des champs du formulaire.

b) Vous pouvez déjà essayer, une erreur 404 sera déclenchée car nous n'avons pas prévu la route dans le serveur mais le navigateur utilise déjà la nouvelle adresse pour soumettre le formulaire.

c) Dans le programme serveur, il faut rajouter la route `"/auth"`. Après la fonction `coucou`, on rajoute une fonction. Appelons-la `authentification`. Il faut l'associer grâce à un décorateur à la route `"/auth"`.

Cela donne :

```
@app.route("/auth")
def authentification():
    return "on vérifie si vous avez le droit d'entrer "
```

La fonction renvoie une simple chaîne pour l'instant.

d) Essayez ! Lorsqu'on valide le formulaire (pour l'instant peu importe ce qu'on y a écrit) on obtient l'affichage de la chaîne « on vérifie...etc... »

Partie 4 : Personnalisation de la réponse, récupération des données du formulaire

Si on envoie les données saisies dans le formulaire au serveur, comment on peut les récupérer dans le programme pour les utiliser ?

Pour cela, un objet est prévu dans l'API Flask : l'objet request. Comme son nom l'indique, il regroupe les caractéristiques de la requête qui a été adressée au serveur.
Les données du formulaires sont évidemment dedans !

a) Dans le programme du serveur, rajouter dans l'import depuis le module flask : `request`.

b) Les paramètres sont passés par l'URL (nous y reviendrons plus tard car c'est problématique). Nous y accédons de la façon suivante (à rajouter au début de la fonction `authentification`):

```
l=request.args.get('chp_login')
```

Par cette ligne, on range dans la variable `l` la valeur du champ du formulaire dont l'attribut `name` est `chp_login`.

c) Récupérez de la même façon dans une variable `m` la valeur saisie dans le mot de passe.

d) On va personnaliser la chaîne de retour :

```
return "on vérifie si vous avez le droit d'entrer "+l+" "+m
```

e) Essayez !!!

Partie 5 : Envoyer la page web de l'utilisateur, les variables d'URL

Imaginez que dans le répertoire `templates` on place toutes vos pages web.
Comme chacune a comme nom le prénom de son créateur, on peut facilement aller chercher la page avec le login de l'utilisateur.

Exemple : Olympe se connecte avec son login `olympe`.

Il suffit d'aller demander la page `olympe.html` qui est dans `templates` pour des raisons de sécurité.

Exemple : Toto se connecte avec son login `toto`. On lui envoie la page `toto.html`.

OK. Mais quelle route pour accéder à ces pages ? Pour faire cela, on va prévoir une nouvelle route qui va être « personnalisée » par le nom de l'auteur de la page.

Exemple, pour accéder à la page d'Olympe, il suffit d'utiliser l'URL :
`127.0.0.1:5000/eleves/olympe`

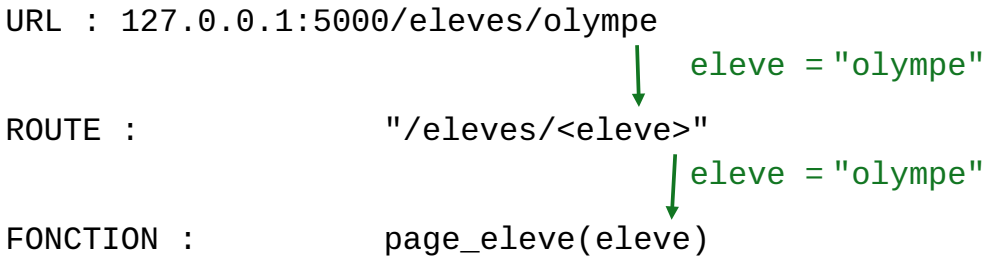
Pour accéder à la page de Toto : `127.0.0.1:5000/eleves/toto`

a) Dans le programme serveur, rajouter une fonction page_eleve associée à notre route ainsi :

```
@app.route("/eleves/<eleve>")
def page_eleve(eleve):
```

Remarquez plusieurs choses :

- la route contient une première partie classique (eleves) et la seconde est entre des < > . Cela symbolise une route « variable ». Ce qui va être lu dans l'URL de la requête dans la deuxième partie va aller dans une variable du nom de `eleve`
- la fonction prend cette fois ci un paramètre : `eleve` comme la variable de l'URL...



b) La fonction `page_eleve` va faire un rendu du template dont le nom est la concaténation de nom de l'élève (donc du contenu du paramètre `eleve`) et de `".html"`.
Cela peut se faire avec une f-string ainsi :

```
return render_template(f"{eleve}.html")
```

c) Essayez ! Dans la barre d'adresse du navigateur, tapez `127.0.0.1:5000/eleves/olympe`.
La page d'Olympe doit être retournée (le style n'est pas là mais vous sauriez comment le rajouter!)

d) Maintenant, il faut articuler cela avec le retour du formulaire.
La validation du formulaire déclenche une requête sur la route `/auth`, qui déclenche l'appel de la fonction `authentification`.
Nous devons donc modifier le retour de cette fonction authentification pour qu'elle renvoie la page de l'élève au lieu de la chaîne personnalisée.

Ce qui revient à se dire : « au bout de la route `/auth`, il faut aller sur la route `/eleves/<eleve>` ».
Cela s'appelle une REDIRECTION.

L'API de Flask prévoit bien entendu les redirections et on utilise à cette fin la fonction `redirect`, alliée à la fonction `url_for`. Ces deux fonctions doivent être importées depuis le module `flask`.

La fonction `url_for` prend en paramètre le nom d'une des fonctions qui gèrent les routes et renvoie l'url qui correspond. C'est une façon d'être moins sensible aux modifications de routes lors du développement.

On a donc à la fin de la fonction authentification :

```
return redirect(url_for('page_eleve',eleve=l))
```

Explications :
On effectue une redirection vers l'url correspondant à la route qui déclenche l'appel de la fonction `page_eleve`, elle même appelée avec le paramètre `eleve` positionné avec le contenu de la variable `l` (rappel : la variable `l` contient la chaîne qui a été saisie dans le formulaire pour le login).
Dit comme ça, c'est compliqué.
Autre vision sans doute plus simple :
On peut se dire qu'après la fonction `authentification`, il faut appeler la fonction `page_eleve(l)`.
On ne le fait pas directement, on le fait en effectuant les requêtes http. Pour que le navigateur web puisse garder des traces de ces requêtes…

Un question que vous vous êtes sans doute posée : Pourquoi on n'a pas tout simplement envoyé la page de l'élève en sortie de la fonction `authentification` ?
La raison est qu'il existera sans doute d'autres moyens d'accéder à la page d'un élève que uniquement après l'authentification au sein d'une appli plus vaste. Une fois que l'élève est authentifié et « connecté », on imagine qu'il peut accéder à certaines données autres, puis revenir après à sa page après coup. Cela n'aurait pas de sens de repasser par la route /auth… Donc un accès direct à la page par la route /eleves/<eleve> est totalement justifié.

e) Donc maintenant, le test à effectuer est de partir du formulaire d'authentification, de saisir le login `olympe` et la page d'Olympe doit être envoyée car l'url est devenue `127.0.0.1:5000/eleves/olympe`

Partie 6 : contrôler l'accès par le mot de passe

Dans une « vraie » appli web, les identifiants des utilisateurs inscrits sont enregistrés dans une base de données et chaque utilisateur possède son mot de passe. Il est par ailleurs crypté et même l'administrateur du site ne peut voir les mots de passe des utilisateurs. Il est capable de voir leur version cryptée. Mais pas leur version initiale, celle qui a été saisie.

Lorsque l'utilisateur envoie son mot de passe au serveur, il est déjà crypté par le protocole https (devenu incontournable maintenant), puis le serveur le crypte avec la fonction qu'il utilise pour

crypter les mots de passe. Il compare alors la version cryptée avec celle qui est dans la base de données. Si ces versions cryptées sont égales, c'est que le mot de passe est le bon.

Ici, on ne va pas aborder cet aspect de cryptographie.
On va simuler un test de mot de passe.
Pour l'exercice, on va juste comparer le mot de passe à la chaîne `'azerty'.`

Le test du mot de passe doit se faire dans la fonction authentification.

a) Dans la fonction `authentification`, rajouter une condition sur la variable `m` comparée à `'azerty'` et envoyer la redirection vers `page_eleve` si la condition est vraie (mot de passe juste) sinon, renvoyer (redirection) le formulaire d'authentification (normalement, il faudrait faire apparaître un message expliquant à l'utilisateur qu'il y a une erreur dans les identifiants, ce qui se ferait aisément avec une variable dans le template login.html)

b) Tester : si on remplit le formulaire avec login = olympe et mdp = autre chose que azerty, on retombe sur le formulaire. Si mdp = azerty, on obtient bien la page d'Olympe.

c) Mais il y a un problème ! Si on saisit dans le champ d'URL du navigateur `127.0.0.1:5000/eleves/olympe`, la page arrive… Sans passer par l'authentification.
la page n'est pas protégée. On peut y accéder sans montrer patte blanche.
C'est comme pour entrer dans une maison par la porte d'entrée fermée à clé, on n'entre pas sans la clé… mais si la baie vitrée du jardin est ouverte, avec ou sans la clé on peut entrer dans la maison.

Dans une vraie appli web, la notion de « session » entre ici en jeu avec diverses techniques. Ici, pour faire simple, on va juste utiliser une variable globale qui va servir symboliquement de « clé ». Si l'authentification est bonne, le serveur donne la clé (uniquement pour la page de l'élève et pas pour les autres !) et lorsqu'on demande la page, le serveur vérifie qu'on a la clé.

En dehors de toute route, avant la première route, définir une variable
`autorisation=None` (avant authentification, on n'a aucun droit d'accès)

En cas d'authentification correcte, on affecte tout simplement le login à cette variable. Donc si cette variable contient un login, c'est que l'utilisateur correspondant s'est bien authentifié.

Ainsi, faire ce qu'il faut dans la fonction `authentification`.
Attention, pour modifier dans une fonction une variable globale, il faut redéclarer la variable au début du corps de la fonction avec l'instruction `global`.

Puis, dans la fonction `page_eleve`, faire ce qu'il faut pour envoyer la page si l'élève correspondant s'est bien connecté. Sinon, envoyer une simple chaine « interdit d'entrer ».

Partie 7 : changer de méthode HTTP pour envoyer le formulaire

Vous n'avez sans doute remarqué, le formulaire envoie le contenu des champs en clair dans la barre d'URL. Même le mot de passe !

Maintenant que l'envoi du formulaire déclenche une redirection, on n'a plus le temps de le voir mais lorsqu'on générait du html pour la route /auth, on le voyait :

Cette solution n'est donc pas la meilleure.

Pour envisager une autre solution, il faut à présent parler du protocole de communication HTTP.

Rappel : ce protocole est celui utilisé pour que le client web (typiquement le navigateur) et le serveur web communiquent. Cela se fait sous forme de **requêtes** et de **réponses** (request et response).

Celles-ci se font toujours dans le sens et l'ordre chronologique suivants :

- le client fait une requête au serveur
- le serveur envoie une réponse au client au sujet de sa requête.

Exemple basique :

- Le client veut charger une page web. Une fois exécuté tout le système de recherche du serveur qui est susceptible de la posséder (grâce au système DNS en particulier), le client envoie une requête au serveur pour lui demander la page.
- Le serveur répond ensuite au client (lui envoie sa réponse). La plupart du temps, il lui envoie la page demandée (le fichier html ainsi que toutes les ressources qui y sont liées comme les images, les styles, les polices, les scripts, etc. Des fois, les réponse du serveur est un code signifiant qu'il n'a pas la page demandée.

La requête HTTP débute par un mot appelé « méthode ». Souvent un verbe.
De façon complète, les méthodes HTTP sont au nombre de 9 : GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE et PATCH.

Chacune a son utilité propre et indique l'action que l'on souhaite réaliser sur la ressource demandée.

Pour demander une simple page web, la méthode est GET.
On peut dire que c'est la méthode par défaut. Lorsqu'on n'indique rien de particulier lors de l'écriture du formulaire HTML, la méthode est GET. Lorsqu'on n'indique rien de particulier dans l'écriture des routes dans Flask, la méthode attendue par le serveur est implicitement GET.

Mais pour envoyer un formulaire, ce n'est pas la méthode la mieux adaptée.
Certaines requêtes comportent dans leur définition une « place » dans le message qui contient les données à envoyer au serveur. C'est le body de la requête.
Or, le requête par méthode GET ne possèdent pas de body ! C'est pourquoi, si on envoie un formulaire par méthode GET, les noms et les valeurs des champs sont envoyés par l'URL et cela explique pourquoi ils sont visibles dans la barre d'adresse.
Mais plus encore que ce problème « de manque de discrétion » , c'est la finalité même de la méthode GET qui n'est pas la bonne pour l'action que l'on souhaite réaliser.

Voici une définition couramment admise de la méthode GET : la méthode GET demande une représentation de la ressource spécifiée. Les requêtes GET doivent uniquement être utilisées afin de récupérer des données. Les requêtes GET ne modifient pas l'état du serveur (elles effectuent des opérations de type « lecture seule »).

Voici donc pourquoi :

- les requêtes par GET n'ont pas de body : elles n'ont normalement rien à envoyer au serveur
- les requêtes par GET ne doivent pas normalement (même si c'est techniquement autorisé) servir à envoyer un formulaire puisque les données du formulaires sont sensées modifier le serveur. Par exemple s'il s'agit d'un formulaire d'authentification comme dans ce TD, l'utilisateur, en se connectant, va modifier l'état du serveur qui va démarrer une session, enregistrer un historique d'utilisation, etc.

Après la méthode GET, la méthode la plus utilisée dans le web est la méthode POST :
Voici une définition couramment admise de la méthode POST : la méthode POST est utilisée pour envoyer une entité vers la ressource indiquée. Cela entraine en général un changement d'état ou des effets de bord sur le serveur.

Voici donc pourquoi un formulaire devrait être envoyé avec la méthode POST.

Nous allons faire les modifications dans notre programme.

a) Dans la page web `login.html`, il faut rajouter l'attribut `method="post"` à la balise `form`.

b) Essayez à présent de rafraîchir la route racine dans le navigateur, entrez le bon login et le bon mot de passe. Validez. Et là :

On peut constater que cela a eu de l'effet : les données du formulaire ne sont plus visibles dans la barre d'adresse. Par contre, le serveur refuse de nous répondre. Normal, il attend une requête par méthode GET car on ne lui a rien dit et c'est le comportement par défaut.

b) Modifier le fichier source premier_serveur.py en remplaçant le décorateur
`@app.route("/auth")`
par
`@app.post("/auth")`
En effet, le décorateur `route` « écoute » par défaut uniquement les requêtes GET. On peut modifier ce caractère par des paramètres optionnels, ou utiliser le décorateur `post` qui n'écoute que les requêtes de méthode POST.

c) On essaye ? Une fois renvoyé le formulaire avec les bonnes valeurs… eh bien on n'arrive pas à avoir la page d'Olympe. Zut. Que se passe-t-il ?

d) La raison est simple : la façon que l'on avait de récupérer les valeurs des champs du formulaire par l'URL ne sont plus correctes ici… Il n'y a plus rien dans l'URL !!
Il faut donc aller chercher ces données là où elles sont : dans le body de la requête !
Modifier dans la fonction authentification la ligne
`l=request.args.get('chp_login')`
par
`l=request.form['chp_login']`
L'attribut `form` de l'objet `request` est tout simplement un dictionnaire dont chaque clé est le nom du champ et la valeur la valeur associée.
Faites de même pour le mot de passe !

e) Essayez… Normalement tout fonctionne correctement.