III.Interactivité dans une page web, côté serveur

Dans la partie précédente, nous avons travaillé du côté client, aussi appelé "front-end". Le travail en "front-end" est centré sur le navigateur avec en particulier l'écriture des codes HTML, CSS et Javascript.

Dans cette partie, nous allons nous intéresser à ce qu'il se passe du côté du serveur, également appelé "back-end", et principalement à comment réagit un serveur suite à une requête du client.

Rem : vous connaissez maintenant la différence entre un "ingénieur back-end" et un "ingénieur front-end"

1) Pourquoi interagir avec un serveur?

Il y a quelques années, le web était « statique » : le **web-designer écrivait son code HTML** et ce code était simplement envoyé par le serveur web au client. Cela correspond à vos sites web faits pendant les vacances sur github.io . Chaque personne voit donc la même chose sur son navigateur.

Les serveurs sont aujourd'hui capables de **générer eux-mêmes du code HTML**. Les résultats qui s'affichent à l'écran dépendent donc en général des demandes effectuées par l'utilisateur du site : le web est devenu dynamique.

Différents langages de programmation sont utilisés côté serveur afin de permettre à celui-ci de générer le code HTML à envoyer. Trois langages dominent actuellement : il s'agit de **Javascript** (utilisé notamment chez Netflix pour le développement du front-end et du back-end), de **Python** (comptant de plus en plus d'utilisateurs) et de **PHP** (en perte de vitesse ces dernières années).

Python possède deux *framework* différents appelées Django et Flask, qui ont chacun leurs avantages et leurs inconvénients.

Faites une petite recherche pour découvrir ce qu'est un framework :				
	_			

```
Pour situer les choses, étudions un exemple très simple de code en PHP :
<!php

$heure = date("H:i");
echo '<h1>Bienvenue aux clients !</h1>
    Il est '.$heure.'';
?>
```

Grossièrement, si un client (=vous) se connecte à un serveur web qui exécute ce code à 19h23, le serveur enverra au client le code HTML ci-dessous : <h1>Bienvenue aux clients !</h1> Il est 19h23

De la même manière, un client se connecte à ce même serveur à 11h59, le serveur enverra au client le code HTML ci-dessous :

```
<h1>Bienvenue aux clients !</h1>
Il est 11h59
```

Les langages côté serveur permettent donc de dynamiser des pages HTML en créant du code à la volée. Dans la suite, le but est donc d'apprendre à générer dynamiquement des pages web côté serveur en utilisant **Python**.

2) Le protocole HTTP

Avant de coder avec Python, revenons un instant sur l'adresse qui s'affiche dans la barre d'adresse d'un navigateur et plus précisément sur le "http"...

<u>Définition</u>: Un protocole est un ensemble de règles qui permettent à 2 ordinateurs de communiquer entre eux.

Exemple:

- ❖ HTTP (HyperText Transfert Protocol) va permettre au client d'effectuer des requêtes à destination d'un serveur web. En retour, le serveur web va envoyer une réponse.
- SSH (Secure Socket Shell) est un autre exemple de protocole réseau qui permet aux utilisateurs (en particulier distants) d'accéder, de manière sécurisée, à un serveur ou un ordinateur via un réseau non sécurisé.

Une requête HTTP, client vers serveur, est constituée d'un nombre important d'instructions (et de lignes !). Il contient entre autres (et dans cet ordre) :

- la méthode employée pour effectuer la requête
- ❖ l'URL de la ressource

- la version du protocole utilisé par le client (souvent HTTP 1.1)
- le navigateur employé (Firefox, Chrome) et sa version
- le type du document demandé (par exemple HTML)
- ***** ...

À noter que certaines lignes sont optionnelles et dépendent de la méthode utilisée.

— À faire vous-même **12** —

- ❖ Téléchargez Telerik Fiddler (https://www.telerik.com/fiddler) et choisissez la version correspondant à votre OS.
- Démarrez Fiddler, configurez-le et cliquez sur "Capture".
- Ouvrez le navigateur de votre choix (évitez Internet Explorer, ce n'est pas un navigateur, c'est une bouse, merci) et baladez-vous un peu sur le World Wide Web (https://bouillotvincent.github.io/NSI Prem.html, www.google.com etc.).

Deux modes existent dans Fiddler. Par défaut, le mode "Inspectors" est activé. Il vous permet de voir quelles requêtes HTTP vous (en tant que client) envoyez aux serveurs que vous contactez dans votre navigateur.

Après votre petite balade sur le web, vous devriez avoir environ 150 résultats! Jouez avez les paramètres Headers, Text View et Raw afin de voir ce qui est affiché.

Y'a-t-il des informations ou un formatage qui reviennent souvent?

L'autre mode d'intérêt dans Fiddler est le mode "Composer". Celui-ci vous permet d'envoyer la requête de votre choix au serveur de votre choix!

❖ Depuis le mode "Composer", exécutez la requête GET au site http:// www.example.com. Techniquement, chaque fois que vous tapez une URL dans votre navigateur, vous envoyez une requête GET au serveur concerné. Regardez la réponse obtenue (spécialement le Text View) puis étudiez dans "Inspectors", regardez ce qu'il s'est passé...

Requête HTTP complet obtenu avec Telerik Fiddler:

```
GET http://bouillotvincent.github.io/Maths_2nde.html? HTTP/1.1

Host: bouillotvincent.github.io

Connection: keep-alive

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*

Referer: http://127.0.0.1:49732/NSI_Prem.html

Accept-Encoding: gzip, deflate

Accept-Language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7
```

Nous avons ici énormément d'informations :

- "GET" = méthode employée (voir ci-dessous)
- "/Maths 2nde.html" = correspond l'URL de la ressource demandée
- "Host : bouillotvincent.github.io" = serveur web appelé
- "HTTP/1.1" = la version du protocole est la 1.1
- "Mozilla/5.0" = le navigateur employé est Firefox de la société Mozilla
- ❖ "text/html" = le client s'attend à recevoir du HTML

L'information la plus importante ici est la méthode employée. En effet, toute requête HTTP utilise une méthode, c'est à dire une commande qui demande au serveur d'effectuer une certaine action.

Voici la liste des méthodes disponibles : GET, HEAD, POST, OPTIONS, CONNECT, TRACE, PUT, PATCH, DELETE

Méthodes à connaitre :

- GET : C'est la méthode la plus courante pour demander une ressource. On peut envoyer des paramètres qui seront visibles dans l'URL.
- ❖ POST : Cette méthode est utilisée pour soumettre des données (par exemple, la caractéristique de nos pokémons) en vue d'un traitement côté serveur. C'est la méthode employée lorsque l'on envoie au serveur les données issues d'un formulaire : ces données seront heureusement invisibles dans l'URL.
- ❖ DELETE : Cette méthode permet de **supprimer** une ressource sur le serveur.
- PUT : Cette méthode permet de modifier une ressource sur le serveur

Réponse du serveur à la requête HTTP précédente :

Une fois la requête reçue, le serveur va renvoyer une réponse (voir page suivante).

```
HTTP/1.1 301 Moved Permanently
Content-Type: text/html
Server: GitHub.com
Location: https://bouillotvincent.github.io/Maths 2nde.html?
X-GitHub-Request-Id: 812C:4A81:1FA54:26E31:5E7277A7
Content-Length: 162
Accept-Ranges: bytes
Date: Wed, 18 Mar 2020 19:33:59 GMT
Via: 1.1 varnish
Age: 0
Connection: keep-alive
X-Served-By: cache-cdg20747-CDG
X-Cache: MISS
X-Cache-Hits: 0
X-Timer: S1584560618.942576, VS0, VE97
Vary: Accept-Encoding
X-Fastly-Request-ID: 8fac3cf6401ede19647d3d64010fbc35c1d0fed6
<html>
<head><title>301 Moved Permanently</title></head>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Comme vous vous en doutez, nous n'allons pas détailler complètement cette réponse. Quelques explications sur les éléments indispensables par la suite :

- (1) Le serveur renvoie du code HTML comme indiqué dans l'encadré rouge.

 Une fois ce code reçu par le client (ie vous), il est interprété par le navigateur qui affiche le résultat à l'écran. Cette partie correspond au corps de la réponse.
- (2) La ligne 1 se nomme la ligne de statut :

HTTP/1.1 : version de HTTP utilisé par le serveur 301 : code indiquant que le document recherché n'a pas été trouvé mais qu'une redirection existe. Il existe d'autres codes dont un que vous connaissez déjà : le fameux code 404 (qui signifie «Le document recherché n'a pu être trouvé»).

(3) Les 5 lignes suivantes constituent l'en-tête de la réponse, une ligne nous intéresse plus particulièrement : Server: <u>GitHub.com</u> qui est le type de serveur utilisé. En général (dans 75% des cas), il nous indique que le coeur du serveur est géré par "Apache", logiciel gérant les requêtes HTTP. Manque de chance, <u>GitHub.com</u> utilise nginx, système concurrent... <u>Un mode de conclusion</u>: Le "HTTPS" est la version sécurisée du protocole HTTP. Par "sécurisé" en entend que les données sont chiffrées par un autre protocole appelé SSL/TLS (Secure Socket Layer/Transport Security Layer) avant d'être transmises sur le réseau.

Voici les différentes étapes d'une communication client - serveur utilisant le protocole HTTPS :

- Le navigateur envoie au serveur une requête d'établissement d'une connexion sécurisée par SSL/TLS.
- ❖ le serveur fournit au client un certificat prouvant son "identité" : ce dernier contient sa clé publique, ses informations (nom de la société, adresse...) et une signature électronique chiffrée. En effet, il existe des attaques dites "man in the middle", où un serveur "pirate" essaye de se faire passer, par exemple, pour le serveur d'une banque : le client, pensant être en communication avec le serveur de sa banque, va saisir son identifiant et son mot de passe, identifiant et mot de passe qui seront récupérés par le serveur pirate. Afin d'éviter ce genre d'attaque, des organismes délivrent donc des certificats prouvant l'identité des sites qui proposent des connexions "https".
- pour faire court, à partir de ce moment-là, les échanges entre le client et le serveur seront chiffrés grâce à un système de clé de chiffrement asymétrique (notion abordée en terminale). Avec ce principe, si un pirate arrivait à intercepter les données circulant entre le client et le serveur, ces dernières ne lui seraient d'aucune utilité, car totalement incompréhensible à cause du cryptage.

Pour aller plus loin sur le SSL/TLS:

https://fr.wikipedia.org/wiki/Transport_Layer_Security (version longue) https://bluesecure.fr/quest-ce-que-le-ssl-ou-tls (version courte)

3) Introduction à Python Flask

Comme évoqué précédemment, un serveur web (aussi appelé serveur HTTP) permet de répondre à une **requête HTTP effectuée par un client**, très souvent un navigateur web.

Nous ne disposons pas d'un accès direct à un serveur web, payant par essence... Nous allons donc travailler avec le serveur web qui est installé sur votre ordinateur! Nous allons donc être dans une configuration un peu particulière : le client et le serveur vont se trouver sur la même machine.

Cette configuration est assez classique dans le cadre de tests de faible envergure. Nous aurons donc deux "systèmes" sur le même ordinateur : le client (navigateur web) et le serveur (serveur web).

Ces deux logiciels vont communiquer en utilisant le protocole HTTP décrit cidessus. Il existe de nombreux serveurs web, dont le plus utilisé se nomme Apache. Nous n'allons pas utiliser Apache dans ce cours, car nous souhaitons travailler avec le **framework Python Flask**. Ce framework, qui va nous permettre de générer des pages web côté serveur, a la particularité de posséder son propre serveur web dans le cadre de tests locaux.

Nous allons commencer par un cas très simple où le serveur va renvoyer au client une simple page HTML statique.

A faire vous-même 13 —

- Dans votre répertoire personnel, créez un répertoire nommé "flask".
- ❖ Téléchargez le fichier Python appelé "views.py" et enregistrez-le dans le répertoire "flask" précédemment créé.
- Lancez le programme python views.py puis dans votre navigateur, tapez dans la barre d'adresse : localhost:5000 Que s'affiche-t-il ?
- D'après vous, que signifie localhost:5000 ? Est-ce un le client ou le serveur ?

Explication du code, ligne par ligne :

importation de la bibliothèque Flask

from flask import Flask

création d'un objet nommé app : cette ligne est obligatoire.

app = Flask(__name___)

décorateur (<u>notion très hors programme</u>) permettant d'exécuter ce qui suit à condition que le serveur web reçoive une requête HTTP avec une URL correspondant à la racine du site ('/')

@app.route('/')

#En cas de requête HTTP d'un client avec l'URL "/" (localhost:5000/), le serveur renvoie vers le client une page HTML contenant uniquement la ligne "Tout fonctionne parfaitement"... que vous avez observée.

def index():

return "Tout fonctionne parfaitement"

Cette ligne permet de lancer le serveur, elle sera systématiquement présente.

app.run(debug=True)

C'est quoi le localhost:5000 alors ???

Nous avons déjà dit que notre serveur et notre client se trouvent sur la même machine. "localhost" indique au navigateur que le serveur HTTP (= serveur web) se trouve sur le même ordinateur que lui : on parle de machine locale ou **hôte local**.

"5000" indique le port utilisé par Python Flask : cela correspond à un canal de discussion entre le navigateur et notre serveur local. Nous n'étudierons pas cette notion de port : le "5000" doit suivre le "localhost" et c'est tout.

Au final, il s'est passé quoi quand j'ai vu "tout fonctionne parfaitement" ?

En exécutant le programme Python views.py, le framework Flask a lancé un serveur HTTP local. Ce serveur web attend des requêtes HTTP sur le port de discussion numéro 5000.

En ouvrant un navigateur web et en tapant "localhost:5000", nous faisons une requête HTTP, le serveur web fourni avec Flask répond à cette requête HTTP en envoyant une page web contenant le code présent dans views.py : "Tout fonctionne parfaitement".

— À faire vous-même **14** —

❖ À l'aide d'Atom, de Geany ou de VsCode, modifiez le fichier Python "views.py" :

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Tout fonctionne parfaitement"
@app.route('/about')
def about():
    return "Une autre page"

app.run(debug=True)
```

- Lancez le programme views.py .
- ❖ D'après ce que l'on a vu ci-dessus, que doit-on taper dans la barre d'adresse du navigateur pour obtenir la page HTML Une autre page ?

<u>Réponse</u>: L'URL racine ("/") reste disponible et une requête HTTP, nous obtenons "Tout fonctionne parfaitement". On devons entrer la requête HTTP localhost:5000/about pour aller sur l'autre page.

Rem : Imaginons que l'on veuille écrire une page web complexe en HTML. Écrire dans le programme Python le code HTML qui devra être renvoyé au client est extrêmement inefficace ! Pour pallier à ce problème, Flask propose une solution bien plus satisfaisante en introduisant les templates, **fichiers HTML qui comportent des variables.**

*	Dans votre répertoire "flask", dirigez-vous dans le répertoire "templates" puis ouvrez le fichier index.html .
*	Quelle ligne nous intéresse ? Rappelez quel va être le rôle de ce fichier.

___ À faire vous-même **16** —

Fini l'échauffement!

Ajoutez les instructions permettant d'appeler des templates dans le programme views.py comme suit :

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/')
def index():
    return render_template("index.html")
app.run(debug=True)
```

- Relancez le programme Python et tapez "localhost:5000" dans la barre d'adresse de votre navigateur. Et voilà!
- Votre page HTML fonctionne-t-elle si on double clic sur le fichier index.html?

Rem : Tous les fichiers HTML devront se trouver dans le répertoire nommé "templates" .

Pour l'instant notre site est statique et la page reste identique quelque soit l'utilisateur. Tout comme PHP ou Javascript (Node.js), Flask permet de créer des pages dynamiques (c'est un peu le but...).

Principe:

- ❖ le client envoie une requête HTTP vers un serveur HTTP;
- en fonction de la requête reçue et de différents paramètres, Flask fabrique une page HTML différente;
- ❖ le serveur HTTP associé à Flask envoie la page nouvellement créée au client ;
- une fois reçue, la page HTML est affichée dans le navigateur web.

__ À faire vous-même **17** __

Récupérons l'heure et à la date en Python et passons-les en paramètres !

Modifiez le fichier views.py comme suit :

```
from flask import Flask, render_template
import datetime

app = Flask(__name__)

@app.route('/')
def index():
    date = datetime.datetime.now()
    h = date.hour
    m = date.minute
    s = date.second
    return render_template("index.html", heure = h, minute =
m, seconde = s)

app.run(debug=True)
```

- ❖ À quoi sert la bibliothèque "datetime ?
- À quoi sert cette instruction : datetime.datetime.now() ?
- ❖ La fonction render_template prend ici trois arguments. Pourrait-elle prendre une liste pour argument ? Recherchez la réponse sur Internet.
- Lancez le programme views.py .

— À faire vous-même 18 -

Ajoutons des paramètres relatifs à l'heure et à la date dans le fichier HTML!

Modifiez le fichier index.html comme suit :

- Testez ces modifications en saisissant "localhost:5000".
- ❖ Votre page HTML fonctionne-t-elle correctement si on double-clique sur le fichier index.html ? Pourquoi ?

Ceci est bien une **page dynamique**: à chaque fois que vous actualisez la page dans votre navigateur, l'heure courante s'affiche. En effet, à chaque fois que vous actualisez la page, vous effectuez une nouvelle requête HTTP et en réponse à cette requête, le serveur HTTP **crée et envoie une nouvelle page HTML**.

Remarque importante : Il est très important de comprendre que la page HTML envoyée par le serveur au client ne contient plus les paramètres {{heure}}, {{minute}} et {{seconde}}. Au moment de **créer la page**, le serveur remplace ces paramètres par les valeurs passées en paramètres de la fonction "render_template". On peut s'en rendre compte en regardant le code source de la page.

De plus, vous avez dû remarquer que lorsque vous lancez index.html dans votre navigateur, vous n'obtenez pas la date courante car le fichier HTML ne connait pas la valeur des paramètres.

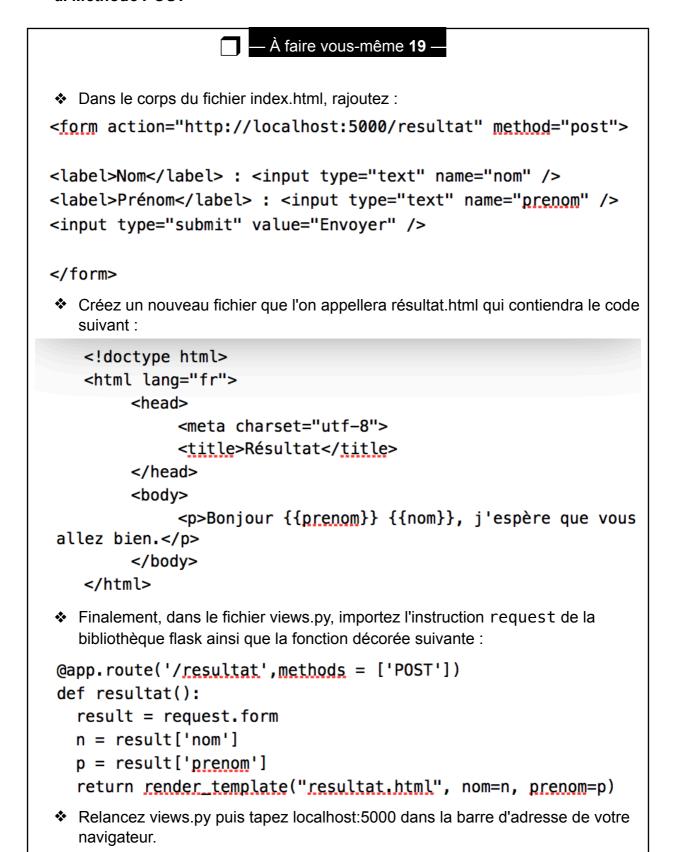
<u>Pour aller un peu plus loin :</u> Le fichier "index.html" ne contient en fait pas du HTML ! En effet, les paramètres {{heure}}, {{minute}} et {{seconde}} n'existent pas en HTML.

Le fichier "index.html" est un langage de template nommé Jinja qui ressemble beaucoup au HTML, mais rajoute des fonctionnalités par rapport au HTML, notamment les paramètres entourés d'une double accolade comme {{heure}}.

4) Formulaire avec Python Flask

Dernière partie, qui va finalement nous ramener à nos Pokémons! Nous allons maintenant nous intéresser à la gestion des formulaires.

a. Méthode POST



Explication générale :

- (1) Nous effectuons une requête HTTP avec l'URL "/", le serveur HTTP génère une page HTML à partir du fichier "index.html". Cette page, qui donne l'heure, contient aussi un formulaire comme indiqué par la balise form. Cette page est affichée par le navigateur (le client).
- (2) La balise form a deux attributs : action="http://localhost:5000/resultat" et method="post". Ces 2 attributs indiquent que le client devra effectuer une requête de type POST (voir le protocole HTTP) dès que l'utilisateur appuiera sur le bouton "Envoyer". Cette requête POST sera envoyée à l'URL "http://localhost:5000/resultat" comme indiquée dans l'attribut action. Les données "nom" et "prenom" saisies dans le formulaire seront envoyées au serveur par l'intermédiaire de cette requête.
- (3) Le template "resultat.html" utilise ces deux paramètres ("nom" et "prenom"). En réponse à la requête POST, le serveur renvoie une page HTML créée à partir du template "resultat.html" et des paramètres "nom" et "prenom". Si l'utilisateur a saisi "Virus" et "Corona", le navigateur affichera "Bonjour Virus Corona, j'espère que vous allez bien."

Quelques points de détails :

- @app.route('/resultat',methods = ['POST']) précise la méthode à utiliser pour cette requête HTTP alors que @app.route('/') ne précise rien. En l'absence de précision, c'est la méthode GET, qui laisse les paramètres visibles, qui sera utilisée.
- ❖ Dans views.py, "request.form" est un dictionnaire Python qui a pour clés les attributs "name" des balises "input" du formulaire (dans notre cas les clés sont donc "nom" et "prenom") et comme valeurs ce qui a été saisi par l'utilisateur. Si l'utilisateur saisit "Martin" et "Sophie", le dictionnaire "request.form" sera : {'nom':'Martin', 'prenom':'Sophie'} . Nous aborderons cette notion de dictionnaire très bientôt.

b. Méthode GET

Pour gérer un formulaire, il est aussi possible d'utiliser la méthode GET à la place de la méthode POST. Voyons quelles différences cela engendre.

— À faire vous-même **20** —

- Dupliquez l'exercice 4 et nommez la copie Exo5. Dans index.html, modifiez la méthode POST et méthode GET.
- Le fichier resultat.html reste inchangé.
- Modifiez le fichier views.py afin d'appeler la méthode GET :

<u>Conclusion</u>: Comme indiqué précédemment, dans le cas de l'utilisation d'une méthode "POST" les données issues d'un formulaire sont envoyées au serveur sans être directement visibles, alors que dans le cas de l'utilisation d'une méthode "GET", les données sont visibles (et accessibles dans la barre d'adresse) puisqu'elles sont envoyées par l'intermédiaire de l'URL.

D'un point de vue sécurité, c'est même pire que cela : les données envoyées par l'intermédiaire d'une méthode "GET" peuvent être modifiées directement dans l'URL :

— À faire vous-même **21** —

- Ouvrez votre navigateur Web et tapez dans la barre d'adresse "localhost: 5000".
- Une fois la page web affichée dans votre navigateur, Saisissez "René" pour le prénom et "LaTaupe" pour le nom puis validez en cliquant sur le bouton "Envoyer".

*	Une fois que le message "Bonjour René LaTaupe, j'espère que vous allez
	bien." apparait, modifier directement l'URL dans la barre d'adresse : passez de
	"localhost:5000/resultat?nom=LaTaupe&prenom=René" à "localhost:5000/
	resultat?nom=Frog&prenom=Froggy", validez votre modification en appuyant
	sur la touche "Entrée".

*	Que se passe-t-il ? Expliquez pourquoi .

<u>Conclusion</u>: Il faut éviter d'utiliser la méthode "GET" pour transmettre des données issues d'un formulaire vers un serveur. En effet, ces données sont directement lisibles dans la barre d'adresse, ce qui est rédhibitoire en terme de sécurité.

Il est aussi important de bien comprendre que la méthode "POST" n'offre pas non plus une sécurité absolue. A l'aide de Fiddler, on peut en effet lire les données transmises à l'aide de la méthode "POST" en analysant la requête HTTP. **Seule l'utilisation du protocole sécurisé HTTPS** garantit un transfert sécurisé des données entre le client et le serveur grâce au chiffrement des données.