NSI première - IHM sur le web

Protocole HTTP

Protocole HTTP

Revenons sur l'adresse qui s'affiche dans la barre d'adresse d'un navigateur web et plus précisément sur le début de cette adresse c'est-à-dire le "http"

Selon les cas cette adresse commencera par http ou https (nous verrons ce deuxième cas à la fin de cette activité).

Le protocole (un protocole est ensemble de règles qui permettent à 2 ordinateurs de communiquer ensemble) 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.

Voici une version simplifiée de la composition d'une requête HTTP (client vers serveur) :

- 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)
- ...

Certaines de ces lignes sont optionnelles.

Voici un exemple de requête HTTP :

```
GET /mondossier/monFichier.html HTTP/1.1
User-Agent : Mozilla/5.0
Accept : text/html
```

Nous avons ici plusieurs informations :

- "GET" est la *méthode* employée (voir ci-dessous)
- "/mondossier/monFichier.html" correspond l'*URL* de la ressource demandée
- "HTTP/1.1" : la version du *protocole* est la 1.1
- "Mozilla/5.0" : le *navigateur* web employé est Firefox de la société Mozilla
- "text/html" : le client s'attend à *recevoir du HTML*

Revenons sur la *méthode* employée :

Une requête HTTP utilise une méthode (c'est 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`

Détaillons 4 de ces méthodes :

- **GET** : C'est la méthode la plus courante pour demander une ressource. Elle est sans effet sur la ressource.
- **POST** : Cette méthode est utilisée pour soumettre des données en vue d'un traitement (côté serveur). Typiquement c'est la méthode employée lorsque l'on envoie au serveur les données issues d'un formulaire.
- **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 une requête HTTP

Une fois la requête reçue, le serveur va renvoyer une réponse, voici un exemple de réponse du serveur :

```
HTTP/1.1 200 OK
Date: Thu, 15 feb 2019 12:02:32 GMT
Server: Apache/2.0.54 (Debian GNU/Linux) DAV/2 SVN/1.1.4
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=ISO-8859-1
<!doctype html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>Voici mon site</title>
<head>
  <body>
    <h1>Hello World! Ceci est un titre</h1>
    <p>Ceci est un <strong>paragraphe</strong>. Avez-vous bien compris ?</p>
  </body>
</html>
```

Nous n'allons pas détailler cette réponse, voici quelques explications sur les éléments qui nous seront indispensables par la suite :

Commençons par la fin (à partir de `<!doctype html>`): le serveur renvoie du code HTML, une fois ce code reçu par le client, il est interprété par le navigateur qui affiche le résultat à l'écran. Cette partie correspond au *corps de la réponse*.

La 1re ligne se nomme la ligne de statut :

- HTTP/1.1 : version de HTTP utilisé par le serveur
- 200 : code indiquant que le document recherché par le client a bien été trouvé par le serveur. Il existe d'autres codes dont un que vous connaissez peut-être déjà : le code 404 (qui signifie «Le document recherché n'a pu être trouvé»).

Les 5 lignes suivantes constituent l'en-tête de la réponse, une ligne nous intéresse plus particulièrement :

```
Server: Apache/2.0.54 (Debian GNU/Linux) DAV/2 SVN/1.1.4
```

Le serveur web qui a fourni la réponse http ci-dessus a comme système d'exploitation une distribution GNU/Linux nommée "Debian" (pour en savoir plus sur GNU/Linux, c'est celle que nous utiliserons d'ailleurs). "Apache" est le coeur du serveur web puisque c'est ce logiciel qui va gérer les requêtes http (recevoir les requêtes http en provenance des clients et renvoyer les réponses http).

Il existe d'autres logiciels capables de gérer les requêtes http (nginx, microsoft-iis...). NGINX (engine x) dépasse depuis peu (juin 2019) Apache. NGINX et Apache sont installés sur 85% des serveurs web mondiaux.

Le "HTTPS" est la version "sécurisée" du protocole HTTP. Par "sécurisé" en entend que les données sont chiffrées avant d'être transmises sur le réseau et que leur provenance est certifiée.

Les étapes du protocole HTTPS

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

- le client demande au serveur une connexion sécurisée (en utilisant "https" à la place de "http" dans la barre d'adresse du navigateur web)
- le serveur répond au client qu'il est OK pour l'établissement d'une connexion sécurisée. Afin de prouver au client qu'il est bien celui qu'il prétend être, le serveur fournit au client un certificat prouvant son "identité".

  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".
- à partir de ce moment-là, les échanges entre le client et le serveur seront chiffrés grâce à un système de "clé publique - clé privée" (nous aborderons le principe du chiffrement par "clé publique - clé privée" en terminale).

Même 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 chiffrement (seuls le client et le serveur sont aptes à déchiffrer ces données)

D'un point vu strictement pratique il est nécessaire de bien vérifier que le protocole est bien utilisé (l'adresse commence par "https") avant de transmettre des données sensibles (coordonnées bancaires...). Si ce n'est pas le cas, passez votre chemin, car toute personne qui interceptera les paquets de données sera en mesure de lire vos données sensibles.

Concernant un site statique, comme le mien, https n'apporte pas grand chose, seulement l'assurance de l'origine. Vous ne transmettez aucune donnée au site.

Complément

Etablir une connexion HTTP directement dans Python, sans utiliser de navigateur

Il est parfaitement possible d'utiliser Python (ou n'importe quel langage moderne) pour établir une connexion avec un serveur web :

```
>>> import requests # librairie qui gère les connexion HTTP
>>> reponse = requests.get("https://qkzk.xyz")
>>> # on établi une connexion avec mon site
>>> reponse
<Response [200]>
>>> # la connexion est établie correctement (200 signifie OK)
>>> reponses.headers # detail de la reponse du serveur
{'Server': 'GitHub.com', 'Content-Type': 'text/html; charset=utf-8',
'Last-Modified': 'Wed, 09 Oct 2019 16:10:57 GMT', 'ETag': 'W/"5d9e0691-e152"',
'Access-Control-Allow-Origin': '*', 'Expires': 'Sat, 12 Oct 2019 09:20:09 GMT',
'Cache-Control': 'max-age=600', 'Content-Encoding': 'gzip',
# réponse tronquée
}
```

- On peut voir que la connexion est bien établie entre mon client (Python lui même) et le serveur (Github.com) qui héberge mon site.
- Le contenu est en html, comme on s'y attend
- L'encodage utf-8
- Et tout un tas d'information moins pertinantes.

Le contenu de la réponse

```
>>> reponse.content
b'<!DOCTYPE html>
<html lang="en">
<head>
<meta name="generator" content="Hugo 0.57.2" />
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>accueil| qkzk</title> ...'
# tronqué. La suite est le contenu complet de la page d'accueil du site...
</body></html>'
```

Il est possible d'examiner la requête transmise par *le client* lui même au serveur :

```
>>> reponse.request
<PreparedRequest [GET]>
>>> reponse.request.headers
{'User-Agent': 'python-requests/2.22.0',
'Accept-Encoding': 'gzip, deflate',
'Accept': '*/*',
'Connection': 'keep-alive'}
```

C'est la requête qui a été transmise quand on a tapé :

```
>>> reponse = requests.get("http://qkzk.xyz")
```

On a bien transmis une requête GET au serveur, avec toutes les informations voulues. C'est bien Python qui a transmis cette requête.

Exercice 1

1. En utilisant Thonny reproduire les commandes présentées ci-dessus pour joindre successivement : `https://google.com` et `https://google.com/azeaze`
2. Comparez les codes réponses obtenus dans les deux cas. Que signifient-ils ?
3. Que peut-on en déduire concernant la page https://google.com/azeaze ?
4. Mesurez la longueur du contenu de la réponse dans les deux cas.

   Comment expliquer cette différence ?

Éxaminer une requête depuis une machine

Différentes approches permettent d'exécuter et d'examiner des requêtes. En voici une qui devrait fonctionner sur un ordinateur et un téléphone.

1. Rendez-vous sur le site httpie et cliquez sur **Try online**.
2. Vous aboutissez devant un *prompt*. Appuyez sur **enter**.
3. Vous pouvez maintenant saisir une requête et examiner son contenu ainsi que sa réponse.
4. Une requête `http -v PUT pie.dev/put API-Key:foo hello=world` est préremplie, exécutez la.

Examinons cette requête.

1. On exécute la commande unix "`http`", c'est celle fournie par le site httpie. On peut installer cette commande sur les PC (linux / windows) et les Macs.

Par ailleurs, ce prompt permet d'exécuter des commandes Unix classiques (`ls`, `mkdir` etc.)

2. L'option `-v` pour "verbose", *verbeux* affiche un contenu détaillé
3. La méthode HTTP `PUT` a été décrite plus haut, relisez la.
4. L'adresse est `pie.dev/put`
5. Les informations fournies sont les paires : `API-Key:foo` et `hello=world`

   On a transmis :

   - une clé d'API (Application Programming Interface) : `"foo"` permettant de s'identifier
   - un couple clé valeur `hello`, `world`
6. Voici le résultat obtenu :

```
~ $ http -v PUT pie.dev/put API-Key:foo hello=world
PUT /put HTTP/1.1
API-Key: foo
Accept: application/json, */*;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 18
Content-Type: application/json
Host: pie.dev
User-Agent: HTTPie/2.4.0

{
    "hello": "world"
}


HTTP/1.1 200 OK
CF-Cache-Status: DYNAMIC
CF-RAY: 675d5bda2e85e6cc-EWR
Connection: keep-alive
```

```
Content-Encoding: gzip
Content-Type: application/json
Date: Wed, 28 Jul 2021 10:17:37 GMT
NEL: {"report_to":"cf-nel","max_age":604800}
Report-To: {"endpoints":[{"url":"https:\/\/a.nel.cloudflare.com\/report\/v3?s=By3uhsityZzSSDvUeNsG
Server: cloudflare
Transfer-Encoding: chunked
access-control-allow-credentials: true
access-control-allow-origin: *
alt-svc: h3-27=":443"; ma=86400, h3-28=":443"; ma=86400, h3-29=":443"; ma=86400, h3=":443"; ma=864
```

Exercice 2

Toujours depuis httpie :

1. exécuter une requête `GET` vers mon site, avec l'option `-v`
2. Donner :
    - l'hôte de la requête,
    - l'User-Agent de la requête,
    - le code de la réponse,
    - la longueur du contenu de la réponse,
    - le type de contenu de la réponse,
    - la date de dérnière modification de la réponse,
    - le serveur de la réponse

Requête GET et POST

En théorie, une requête GET permet d'**accéder** à un contenu, sans le modifier, et une requête POST d'**envoyer** un nouveau contenu.

Ainsi, presque toutes les requêtes effectuées sur internet sont des requêtes GET.

Transmettre une information via une requête GET

Lorsqu'on transmet des informations via une requête GET, celles-ci sont transmises dans l'URL directement, sous la forme de paire "clé:valeur" :

`https://httpbin.org/get?cle1=valeur1??cle2=valeur2`

Par exemple avec httpie :

```
$ http -v GET https://httpbin.org/get?cle1=valeur1??cle2=valeur2
```

Ainsi, on a transmis deux informations : `cle1: valeur1` et `cle2: valeur2`

Remarquez la syntaxe : `?` après l'adresse, `cle=valeur` et `??` pour séparer les paires.

Transmettre une information via une requête POST

Cette fois, on transmet l'information dans l'entête, généralement via le champ `form` (*formulaire*).

Ainsi, lorsque vous vous identifiez ou remplissez un formulaire, vos informations sont *généralement* transmises via une requête `POST`.

Les informations ne sont pas visibles directement mais elles ne sont pas chiffrées pour autant. Simplement, le navigateur ne les affiche pas dans la page.

Considérons l'exemple suivant :

```
$ http -v POST https://httpbin.org/post hello=world bonjour=salut
```

Exercice 3

1. Exécuter cette requête sur httpie
2. La requête a-t-elle été exécutée correctement ? Comment le sait-on ?
3. Le serveur transmet-il ce qu'il a reçu en retour ? Dans quel champ ?
4. La réponse de ce serveur est-elle un contenu html ?

Publier sur un microblog

Cette partie nécessite qu'un serveur dont je ne suis pas l'administrateur fonctionne toujours à cette date. Espérons que ce soit bien le cas !

Nous allons publier quelques messages sur le blog de mon collègue.

Les champs acceptés sont `title` et `body`.

Lorsque le post est accepté, la réponse contient l'url permettant de consulter le message.

```
~ $ http -v POST https://liris-ktbs01.insa-lyon.fr:8000/blogephem/ title=bjour body=super
POST /blogephem/ HTTP/1.1
Accept: application/json, */*;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 35
Content-Type: application/json
Host: liris-ktbs01.insa-lyon.fr:8000
User-Agent: HTTPie/2.4.0

{
    "body": "super",
    "title": "bjour"
}


HTTP/1.1 201 Created
Connection: Keep-Alive
Content-Encoding: gzip
Content-Length: 78
Content-Type: text/plain
Date: Wed, 28 Jul 2021 11:21:37 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.2.22 (Debian)
Vary: Accept-Encoding
location: https://liris-ktbs01.insa-lyon.fr:8000/blogephem/kewotepo

https://liris-ktbs01.insa-lyon.fr:8000/blogephem/kewotepo
```

Ici, le code de la réponse est 201. Il indique qu'une information a été crée dans la base.

Exercice 4

1. Publier un message sur le microblog. Par exemple, dîtes vous bonjour !
2. Modifier légèrement :
   - l'url
   - le champ `title` vers `titre`
   - le champ `body` vers `contenu`

   Qu'obtient-t-on dans chacun des cas ?
3. Peut-on publier un message sans `title` ? sans `body` ?