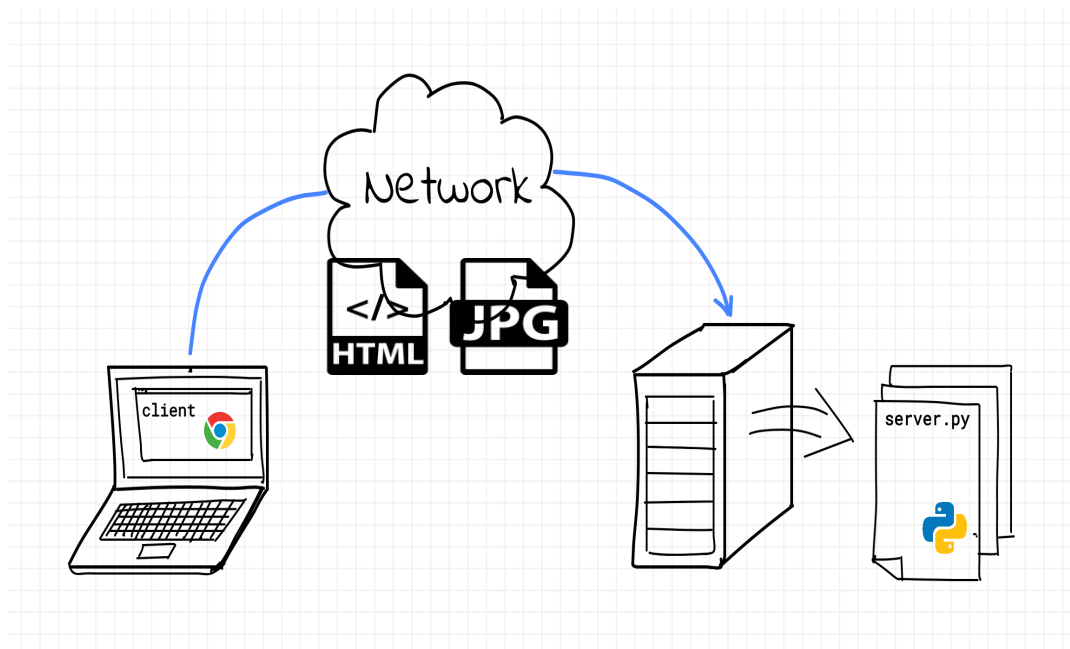


2022/2023

Développement du protocole HTTP

Bureau d'étude S6



Lien du la video:

<https://drive.google.com/file/d/18pmghDuMVFI8VcM97N4Bgfb9a85t7w0A/view?usp=sharing>

Réalisé par : BARHOUC Mustapha

Prof : M. André AOUN

Table des matières :

1- Introduction sur le protocole HTTP

2- Installation du serveur Apache et commande telnet

Installation du serveur apache sur la machine et l'observation du fonctionnement du protocole http avec la commande « telnet »

3- Conception et développement du client

Conception algorithmique et codage en Python du client pour qu'il se connecte à un serveur web, lui envoie une requête HTTP bien formatée et affiche la réponse HTTP

4- Teste du client

Teste du client en connectant au serveur Apache installé localement

5- Conception et développement d'un mini-serveur gérant les requêtes HEAD et GET pour des fichiers HTML

Conception algorithmique et codage en Python d'un mini-serveur Web qui ne gèrera que les requêtes HEAD et GET pour des fichiers textes formatés en HTML.

6- Teste du serveur

Tester le serveur en utilisant un navigateur web comme Chrome ou Firefox pour récupérer une page HTML simple.

7- Modification du serveur

Modification du serveur pour qu'il puisse renvoyer des images JPG.

8- Teste du serveur

Tester le serveur en utilisant un navigateur web comme Chrome ou Firefox pour récupérer une image JPG.

1- Introduction sur le protocole http

HTTP : Hyper Text Transfer Protocol, est un protocole de communication client-serveur permettant l'accès à des ressources situées sur un serveur Web. Lors d'une communications entre un client HTTP et un serveur Web, le port 80 est utilisé, lorsque le protocole HTTP sert à établir la connexion. Autrement dit, le serveur Web écoute sur le port 80 pour recevoir les requêtes HTTP émises par les clients. A cette époque on préfère le HTTPS, dont le S signifie Secured, c'est une variante sécurisée du protocole HTTP et qui s'appuie sur les protocoles TLS pour chiffrer les échanges entre le client et le serveur.

Les versions d'HTTP

HTTP/1.0 : Mis en service en 1996

HTTP/1.1 : Mis en service en 1997 et il s'agit d'une évolution de version 1.0, l'une des fonctionnalités ajoutées dans cette version c'est la notion de cache pour mettre en mémoire certains éléments.

HTTP/2 : Mis en service en 2015, il est désormais utilisé sur une grande majorité de sites Internet. Néanmoins, la version HTTP/1.1 reste encore utilisée aujourd'hui.

HTTP/3 : **correspond au HTTP-over-QUIC** (*Quick UDP Internet Connections*) et il s'appuie sur QUIC, une version améliorée de l'UDP lancée en 2012 par Google. On peut dire que **QUIC est un protocole de transport qui va rentrer en concurrence avec UDP et TCP**. Le HTTP/3 est déjà utilisé par certains géants de l'Internet dans le but d'avoir un Web plus rapide.

Les requêtes et les réponses http :

Une requête HTTP contient une méthode, une cible (précisée par son chemin) et la version du protocole utilisé.

Dans notre étude on va consister sur les 2 méthodes **GET** et **HEAD**.

Méthode GET :

La méthode GET permet de demander une ressource au serveur Web. Par exemple, un fichier HTML, un fichier image ...

Méthode HEAD :

La méthode HEAD permet d'obtenir des informations sur la ressource en elle-même, en obtenant l'en-tête de la réponse, sans pour autant demander la ressource en elle-même.

Une réponse HTTP

Lorsqu'un client envoie une requête HTTP au serveur Web, il reçoit une réponse HTTP de la part du serveur. Cette réponse HTTP intègre la version du protocole utilisée, et aussi un code

de retour. Ce code de retour est très important, car il indique si la requête a été traitée correctement, si elle est introuvable, ou encore si l'accès est refusé.

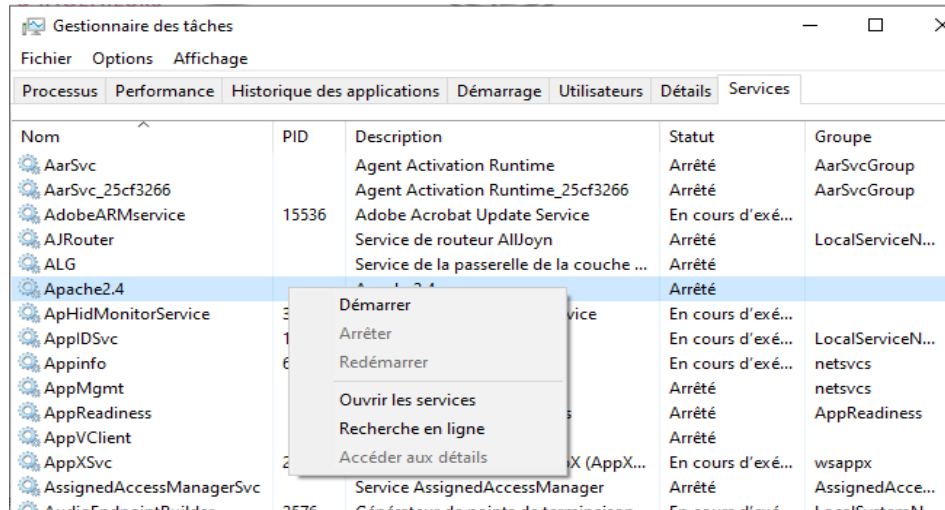
Les codes de retour :

Voici une liste des codes les plus fréquents :

- **200** : succès de la requête donc la ressource est chargée correctement
- **301 et 302** : indique une redirection, permanente (301) ou temporaire (302), c'est utilisé lors du changement d'un domaine sur un site Web, ou lorsqu'une page change d'URL
- **403** : l'accès à la ressource est refusé par le serveur, ceci peut se produire si l'accès à une page est limité à certaines adresses IP, par exemple
- **404** : la ressource est introuvable (vous savez, la fameuse page "404 not found" ou "404 page introuvable")
- **500** : erreur côté serveur, ce qui peut être liée à une erreur de code, par exemple. Il y a aussi les codes 502 et 503, moins fréquents.

2- Installation du serveur Apache et commande telnet

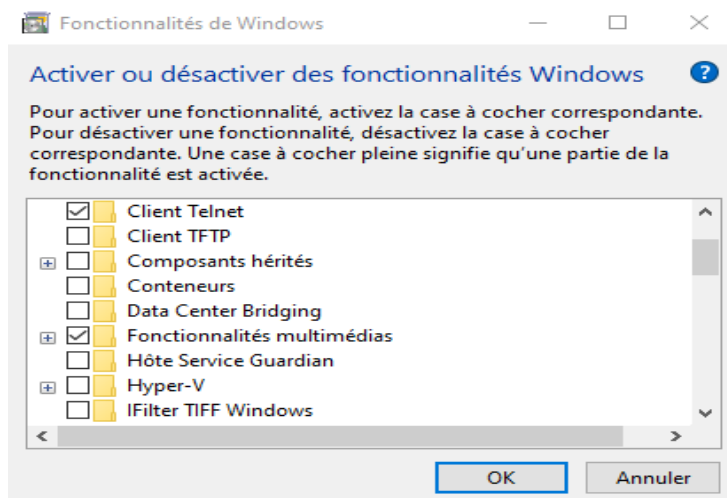
On peut installer un serveur Apache localement en le téléchargeant sur le site web APACHE sous forme compressé et l'installer facilement sur la machine locale puis le démarrer depuis le gestionnaire des tache → Services → clique droite sur Apache → Démarrer.



Capture d'écran du gestionnaire des taches

- **Commande "telnet"**

Premièrement il faut activer le protocole telnet depuis Panneau de configuration
→ Programmes → activer ou désactiver des fonctionnalités Windows → sélectionné Client telnet puis enregistrer



3- Conception et développement du client :

Dans cet axe nous allons réaliser une conception algorithmique, et on va développer un codage python d'un client qui va ouvrir une connexion TCP avec le serveur apache déjà installé localement (sur la machine) en lui envoyant une requête HTTP bien formaté, le serveur donc va bien traiter la requête et renvoyé une réponse HTTP au client et l'affiche sur la sortie standard.

**** Dans notre développement l'API Sockets déjà réalisé en BE va être utilisé. ****

- **Conception algorithmique :**

```

Algorithme : Client
/* Client pouvant se connecter à un serveur web, lui envoyer une requête HTTP et affiche la réponse HTTP.*/
Variables
Host : chaîne, Port : entier, Rqst : binaire, Client : socket, resp : binaire
Début
    Host ← "127.0.0.1",    Port ← 80,    Rqst ← b'GET / HTTP/1.1\r\nHost: 127.0.0.1\r\n\r\n'

    /* Création d'un objet socket */
    Client ← socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    /* Connexion au serveur */
    Client.connect((Host,Port))

    /* Envoyer la requête en utilisant la fonction développée dans l'API Socket */
    emission(Client, Rqst)

    /* Boucle pour recevoir la réponse du serveur */
    TantQue vrai Faire
        Début
            data ← reception(Client) /* Utilisation de la fonction reception développée dans l'API Socket */
            Si (data = Ø) Alors /* Tester si data est vide */
                Sortir de la boucle
            Finsi
            resp ← resp + data
        FinTantQue
    /* Affichage de la réponse HTTP décodée sous la norme utf-8 */
    Ecrire(resp.decode('utf-8'))

    /* Fermeture de la connexion en utilisant la fonction fin_communication développée dans l'API Socket */
    fin_communication(Client)
Fin
    
```

Conception algorithmique du client

- **Explication sur le code python :**

- Les bibliothèques "socket" et "apisockets" sont importés au début du code. Ensuite, l'adresse IP et le port du serveur sont définis dans les variables "Host" et "Port", respectivement. Une requête HTTP est également créée sous forme binaire dans la variable "Rqst".
- Ensuite, un objet socket est créé à l'aide de la fonction "**socket.socket()**" et connecté au serveur à l'aide de la méthode "**connect()**". La requête est ensuite envoyée au serveur à l'aide de la fonction "**emission()**" fournie par l'API "apisockets".
- Ensuite, une boucle "while" est utilisée pour recevoir la réponse du serveur. La réponse est stockée dans la variable "resp" sous forme binaire à l'aide de la fonction "**reception()**" fournie par l'API "apisockets". La boucle s'arrête lorsque la réponse est vide.
- Enfin, la réponse est affichée sous forme de texte en utilisant la méthode "**decode()**" pour décoder la réponse binaire en utilisant la norme utf-8. La connexion est ensuite fermée à l'aide de la fonction "**fin_communication()**" fournie par l'API "apisockets".

4- Teste du client

Dans cette partie on va exécuter le code python du client en se connectant au serveur Apache installé localement :

Dans cette image on peut illustrer la réponse du serveur Apache :

```
PS C:\Users\stri\Desktop\BE_S6> py .\Client.py
HTTP/1.1 200 OK
Date: Fri, 03 Mar 2023 12:25:27 GMT
Server: Apache/2.4.55 (Win64)
Last-Modified: Thu, 16 Feb 2023 16:19:49 GMT
ETag: "49-5f4d390e6179e"
Accept-Ranges: bytes
Content-Length: 73
Content-Type: text/html

<html><body><h1>It works! and its me who writes that</h1></body></html>
```

Capture d'écran d'exécution du programme Client

5- Conception et développement d'un mini-serveur gérant les requêtes HEAD et GET pour des fichiers HTML

Dans cette partie nous allons réaliser une conception algorithmique, et on va développer un codage python d'un mini-serveur Web qui ne gèrera que les requêtes HEAD et GET pour des fichiers textes formatés en HTML. Le mini-serveur WEB va donc recevoir une requête HTTP par le client et va lui renvoyer une réponse http selon plusieurs cas (Trouver le fichier HTML demander, ne pas trouver le fichier, une requête avec une méthode déférente de GET et HEAD ...). Pour le teste on va utiliser le navigateur Chrome comme client qui vas envoyer des requêtes au serveur, le serveur va répondre selon plusieurs cas.

- Conception algorithmique :

Algorithme serveur_HTTP

Variables :

HOST : chaîne de caractères

Port : entier

Début

HOST ← '' // définir le host du serveur

PORT ← 8080 // définir le port du serveur

// créer un socket pour le serveur

server_socket ← créer_socket(AF_INET, SOCK_STREAM)

definir_option_socket(server_socket, SOL_SOCKET, SO_REUSEADDR, 1)

lier_socket(server_socket, HOST, PORT)

ecouter_socket(server_socket)

Afficher "Le serveur est en écoute sur le port " + PORT

// boucle infinie pour accepter les connexions entrantes

TantQue Vrai Faire

client_socket, client_address ← accepter_connexion(server_socket)

Afficher "Connexion acceptée de l'adresse " + client_address

// traiter la requête du client en appelant la fonction serveClient

serveClient(client_socket)

finTantQue

Fin


```

Fonction serveClient(E client : socket) :
Variables :
rqstData : chaîne de caractères
rqstMethod : chaîne de caractères
rqstPath : chaîne de caractères
_ : chaîne de caractères
f : fichier
fileContent : chaîne de caractères
respHeaders : chaîne de caractères
respBody : chaîne de caractères
Début
rqstData ← reception(client) // recevoir la requête envoyée par le client
rqstData ← decoder(rqstData, 'utf-8') // decoder la requête en modèle UTF-8
rqstMethod, rqstPath, _ ← diviser(rqstData, ' ', 2) // séparer la méthode, le chemin de fichier et le reste de la requête
// vérifier si la méthode de la requête est un GET ou HEAD
Si (rqstMethod = 'GET' ou rqstMethod = 'HEAD') Alors
    // si le chemin de fichier demandé est '/', ouvrir le fichier HTML saisi par l'utilisateur
    Si (rqstPath = '/') Alors
        fileName ← 'index.html'
        rqstPath ← "/" + fileName
    Fin Si

    // essayer d'ouvrir le fichier HTML demandé
    Essayer
        f ← ouvrir_fichier('./' + rqstPath, 'r') // ouvrir le fichier HTML
        fileContent ← lire_fichier(f) // lire le fichier HTML
        fermer_fichier(f)

        respHeaders ← "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-Length: " + longueur(fileContent) + "\r\n\r\n"
        /* envoyer une réponse HTTP avec le code 200 OK indiquant que la requête
        est traitée avec succès et que le contenu de la réponse sera un fichier HTML */

        // si la méthode est un GET, renvoyer le contenu de fichier HTML qui sera affiché dans le navigateur
        Si (rqstMethod = 'GET') Alors
            respBody ← fileContent
        Sinon
            respBody ← '' // si la méthode est un HEAD, rien ne sera affiché sur le navigateur
        Fin Si
    SaufErreur FichierNonTrouvé /* Si le fichier HTML demandé n'a pas été trouvé */
        respHeaders ← "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\n"
        /* Réponse HTTP avec le code 404 indiquant que le serveur n'a pas trouvé le fichier HTML demandé */
        respBody ← '404 Not Found' /* Contenu à afficher dans le navigateur */
    Fin essayer

Sinon /* Si la méthode n'est pas GET ou HEAD */
    respHeaders ← "HTTP/1.1 405 Method Not Allowed\r\nContent-Type: text/plain\r\n\r\n" /* Réponse HTTP avec le code
    405 indiquant que la méthode n'est pas reconnue */
    respBody ← 'Method Not Allowed' /* Contenu à afficher dans le navigateur */
Fin si

resp ← encodage(respHeaders + respBody) /* La réponse est la réponse HTTP et le fichier ou le contenu à afficher dans le navigateur */
emission(client, resp) /* Envoi de la réponse au client */
fin_communication(client) /* Fermeture de la connexion avec le client */

```

Conception algorithmique de mini serveur WEB

- **Explication sur le code python :**

Le code implémente un mini-serveur web qui traite les requêtes GET et HEAD pour les fichiers HTML. Voici la description de ce code :

- Importation des bibliothèques nécessaires, y compris la bibliothèque socket et l'API Socket définie dans le premier BE.
- Définition de l'hôte et du port à utiliser pour le serveur.
- Définition d'une fonction **serveClient()** qui prend en paramètre un socket client et traite sa requête.
- La fonction serveClient reçoit la demande envoyée par le client et la décode en utilisant le modèle utf-8.
- La méthode (GET ou HEAD ou Autre) et le chemin de fichier demandé sont extraits de la requête.
- Si la méthode de la requête est bien GET ou HEAD, la fonction traite la demande en cherchant le fichier HTML demandé. Si le fichier est trouvé, le contenu est lu et renvoyé dans une réponse HTTP avec le code **"200 OK"**. Si la méthode est GET, le contenu du fichier HTML est affiché sur le navigateur. Si la méthode est HEAD, rien n'est affiché. Si le fichier n'est pas trouvé, la fonction renvoie une réponse HTTP avec le code **"404 Not Found"**.
- Si la méthode n'est ni GET ni HEAD, la fonction renvoie une réponse HTTP avec le code **"405 Method Not Allowed"**.
- La réponse HTTP est envoyée au client et la connexion est fermée.
- Le serveur est créé, lié à l'hôte et au port, mis en mode écoute et attend les connexions entrantes.
- Tant que le serveur est en mode d'écoute, il accepte les connexions entrantes, crée un nouveau fil d'exécution pour chaque connexion et traite la demande du client en appelant la fonction serveClient.

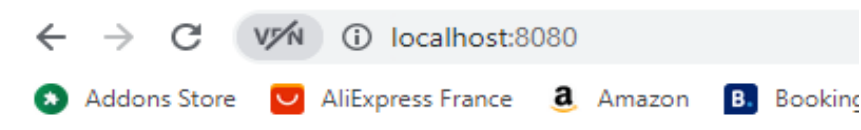
6- Teste du serveur

Dans cette partie on va exécuter le code python en lui envoyant des requêtes en utilisant le navigateur Chrome :

Dans l'image suivant on peut illustrer que le fichier `"index.html"` est bien afficher sur le navigateur après l'exécution du programme et l'envoi d'une requête GET par le navigateur en saisissant l'url `"localhost :8080"`.

```
PS C:\Users\stri\Desktop\BE_S6> py .\Server.py
Listening on :8080
Connected by ('127.0.0.1', 52722)
Connected by ('127.0.0.1', 52723)
```

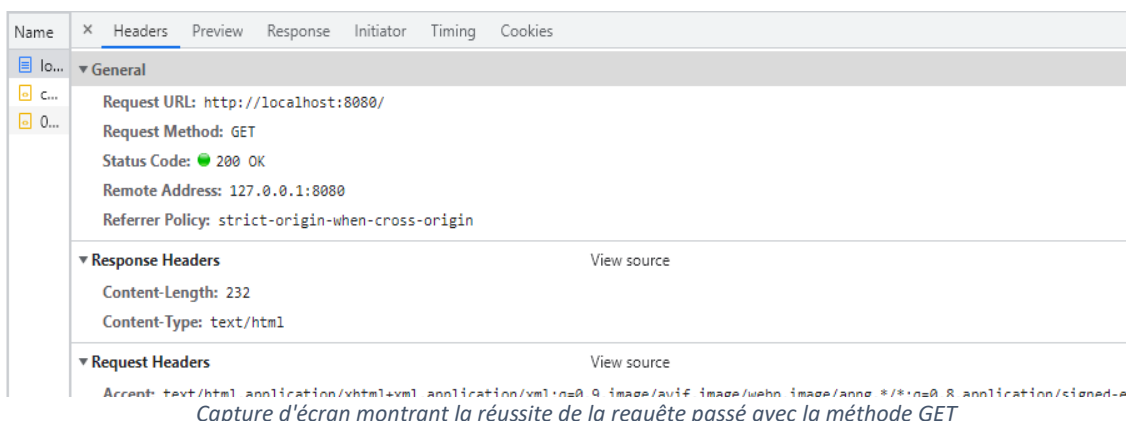
Capture d'écran du terminal



The server is running

By: BARHOUCHE MUSTAPHA

Capture d'écran du navigateur



Capture d'écran montrant la réussite de la requête passé avec la méthode GET

7- Modification du serveur

Dans cette partie nous allons modifier le serveur afin qu'il puisse renvoyer des images JPG. Le mini-serveur WEB va donc recevoir une requête HTTP par le client et va lui renvoyer une réponse HTTP selon plusieurs cas comme dans l'exemple précédant pour les fichiers HTML en ajoutant une contrainte que l'image sera de type JPG sinon il va renvoyer une erreur comme quoi le type de fichier n'est pas supporté par le serveur. Dans cette implémentation j'ai préféré de mettre à l'utilisateur qu'il donne le nom de l'image à ouvrir. Pour le test on va utiliser le navigateur Chrome comme client qui va envoyer des requêtes au serveur, le serveur va répondre selon plusieurs cas.

- **Explication des modifications réalisées sur le code python :**

Le code utilisé dans cette partie est similaire au code pour les fichiers HTML mais avec quelques modifications qui sont les suivantes :

- Modification de la fonction **serveClient()** en ajoutant un paramètre de type string comme ça l'utilisateur va donner le nom de l'image à ouvrir
- Ajoute d'un bloc qui teste si l'image est bien à une extension **.jpg** sinon il génère une réponse http avec le code " **415 Unsupported Media Type**"
- Modification du réponse http en cas de réussite comme ça le fichier retourné est une image et pas un fichier texte comme c'est montré dans la ligne du code suivante : `'HTTP/1.1 200 OK\r\nContent-Type: image/jpeg\r\nContent-Length: {} \r\n\r\n'.format(len(fileContent))`

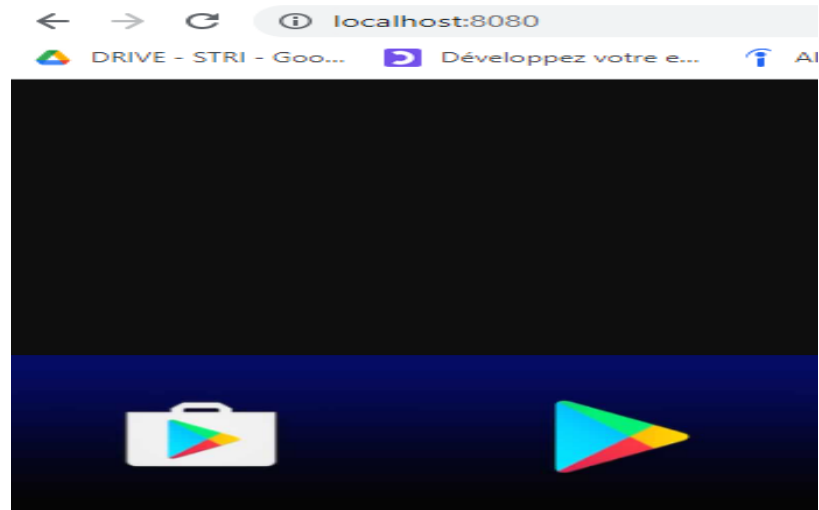
8- Teste du serveur

Dans cette partie on va exécuter le code python en lui envoyant des requêtes en utilisant le navigateur Chrome :

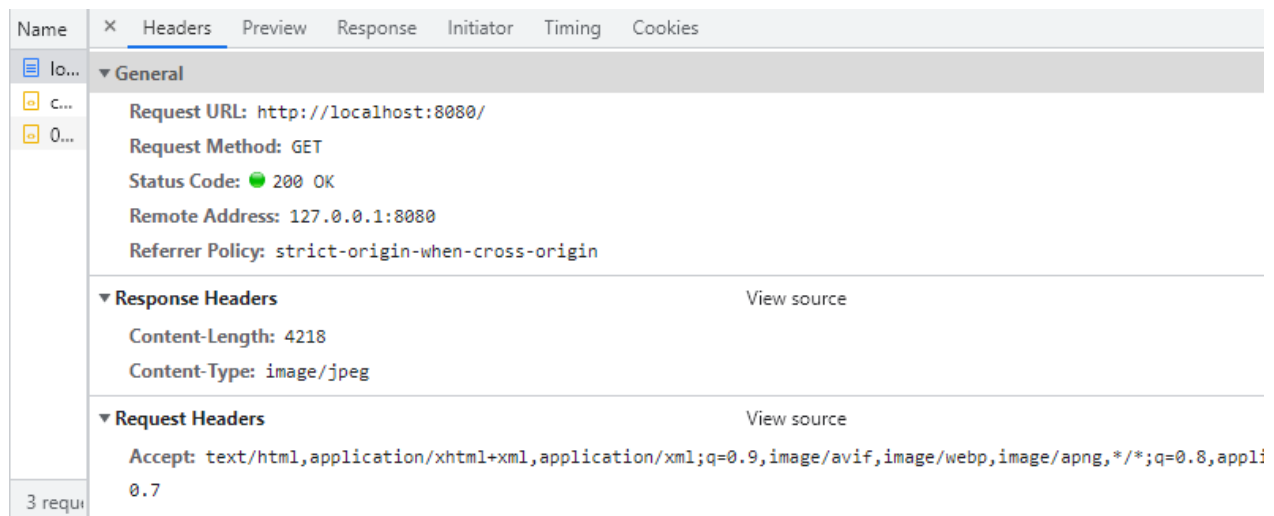
Dans l'image suivant on peut illustrer que l'image saisie par l'utilisateur "*index.jpg*" est bien affichée sur le navigateur après l'exécution du programme et l'envoi d'une requête GET par le navigateur en saisissant l'url "localhost :8080".

```
PS C:\Users\stri\Desktop\BE_S6> py .\Serverjpg.py
Set the file name to open: index.jpg
Listening on :8080
Connected by ('127.0.0.1', 61779)
Connected by ('127.0.0.1', 61780)
```

Captur d'écran du terminal



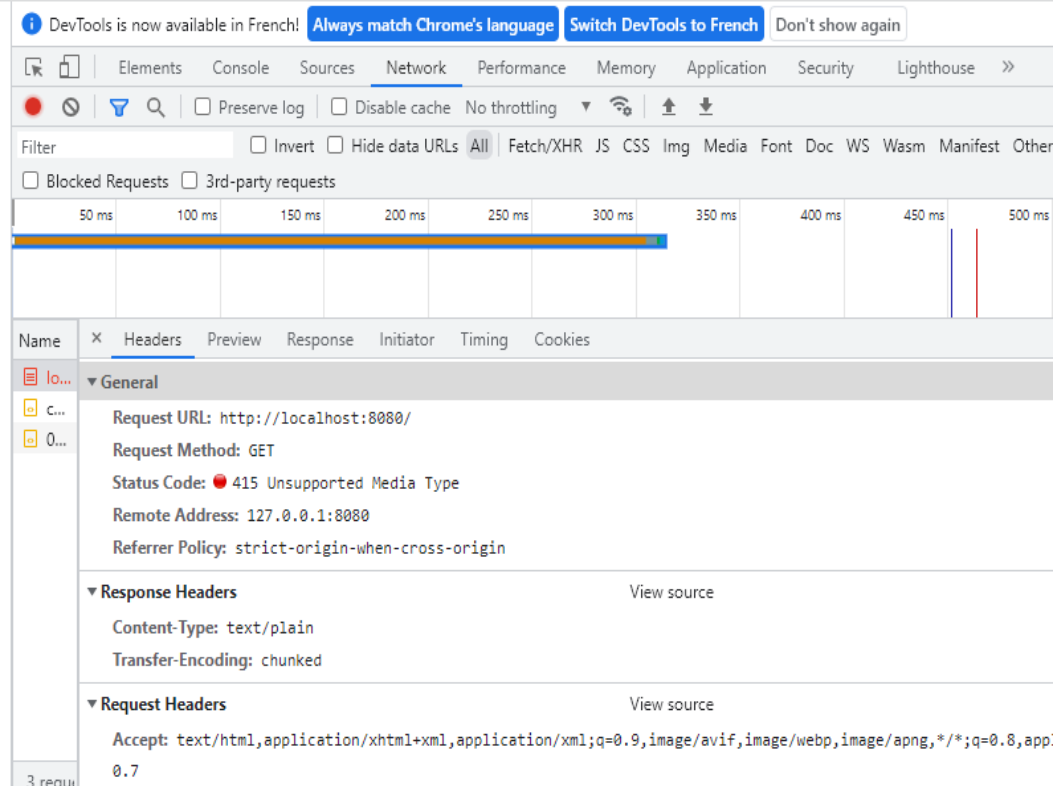
Capture d'écran du résultat affiché sur le navigateur



Capture d'écran montrant la réussite de la requête passé avec la méthode GET

Dans la suite on va donner un exemple du résultat si l'utilisateur a saisi une image du type png :

Unsupported Media Type



The screenshot shows the Chrome DevTools Network tab. A request to `http://localhost:8080/` has failed with a status code of 415 (Unsupported Media Type). The request method is GET. The response headers indicate `Content-Type: text/plain` and `Transfer-Encoding: chunked`. The request headers show an `Accept` header with a wide range of media types.

Name	Value
Request URL	http://localhost:8080/
Request Method	GET
Status Code	415 Unsupported Media Type
Remote Address	127.0.0.1:8080
Referrer Policy	strict-origin-when-cross-origin
Response Headers	Content-Type: text/plain Transfer-Encoding: chunked
Request Headers	Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7

Capture d'écran du résultat obtenu

“Il peut avoir aussi des autres types d'échoue de la requête comme de donner un fichier inexistant.”