TP WEB
CLIENT SERVER
TESSIER THEO

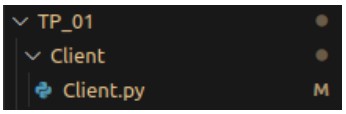
SOMMAIRE

Partie CLIENT

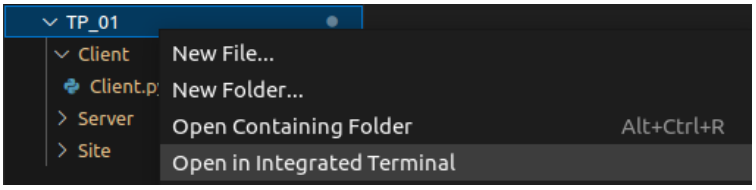
Utilisation

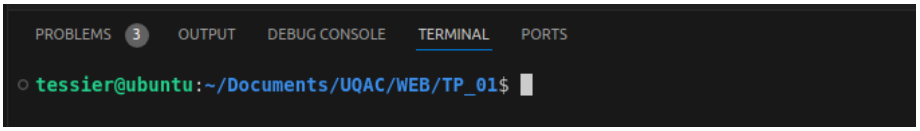
Lancement du Client

La partie Client est codé en Python et est disponible dans le répertoire “Client”:



Afin de pouvoir lancer celui-ci et le faire fonctionner il faut lancer un terminal comme ceci:





Afin de pouvoir lancer le client il faut lancer une ligne de commande, dans celle-ci il faut aussi entrer deux commande : “--url” afin d’entrer l’url souhaitée; “--method” pour entrer la méthode souhaitée. Voici un exemple d’une commande sous linux et windows :

linux (il vaut mieux faire un sudo car certains port peuvent poser problème) :

windows :

Pour “--url” il est simple de comprendre qu’il faut fournir un lien url “http://” ou “https://”, le client gère lui-même le changement de port selon le http(s) utilisé.
Cependant une explication s’impose concernant les méthodes et leurs différentes utilisations.

Explication des différentes Méthodes

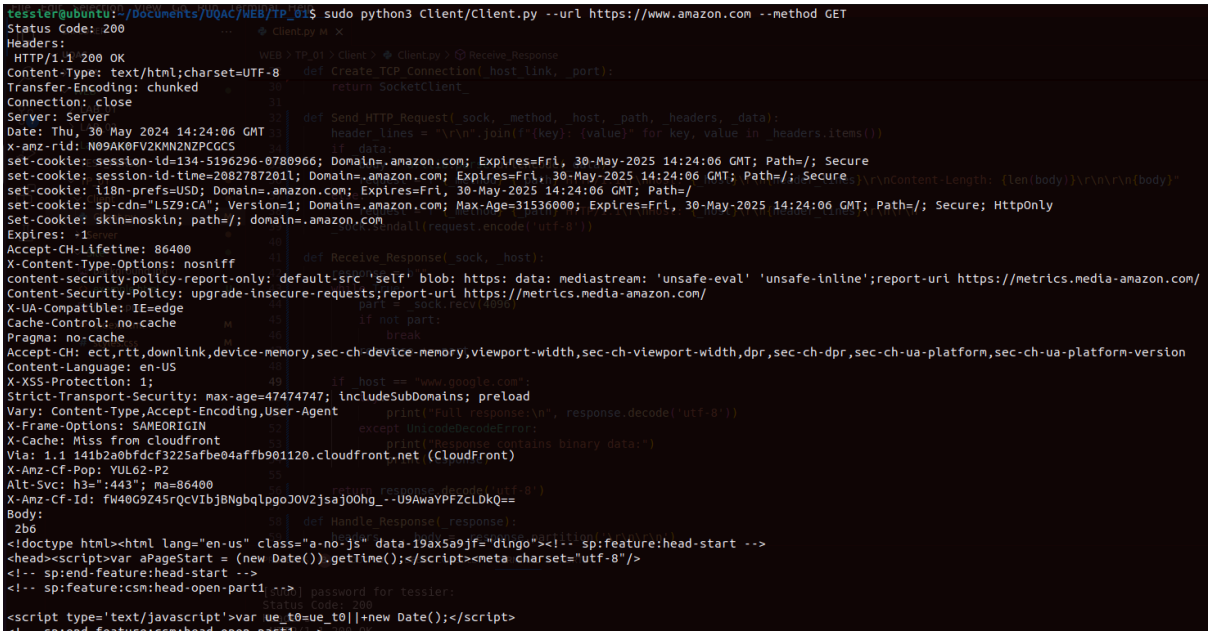
Il existe 4 méthodes utilisables : GET, POST, PUT, DELETE. Nous allons voir chacune d’entre elles afin de mieux comprendre leurs utilités.

Méthode GET

Cette méthode sert à afficher des informations/données d’un serveur. Par exemple si je l’utilise sur un fichier simple n’ayant que peu d’information voici ce qui peut arriver:

```
tessier@ubuntu:~/Documents/UQAC/WEB/TP_01$ sudo python3 Client/Client.py --url http://localhost/exemple.html --method GET
Status Code: 200
Headers:
 HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 72
Body:
 <html><body>Nom: nom<br>Prenom: prenom<br>Email: email<br></body></html>
```

Nous voyons donc que le GET renvoie le statut (servant à indiquer si la requête à bien fonctionner ou s'il y a une erreur ou autre) nous avons le type de contenant ainsi que sa taille et les informations/données dans le "Body".
Ici tout semble claire, cependant, voici un exemple avec un réel site (https://www.amazon.com):



Effectivement, celui-ci est bien plus complexe et son Body est très long (je conseille d'exécuter la commande dans un terminal afin de voir l'entièreté du contenu).

Méthode POST

Cette méthode sert à créer un fichier et y insérer des informations (il peut aussi l'update) voici un exemple sur un server local:

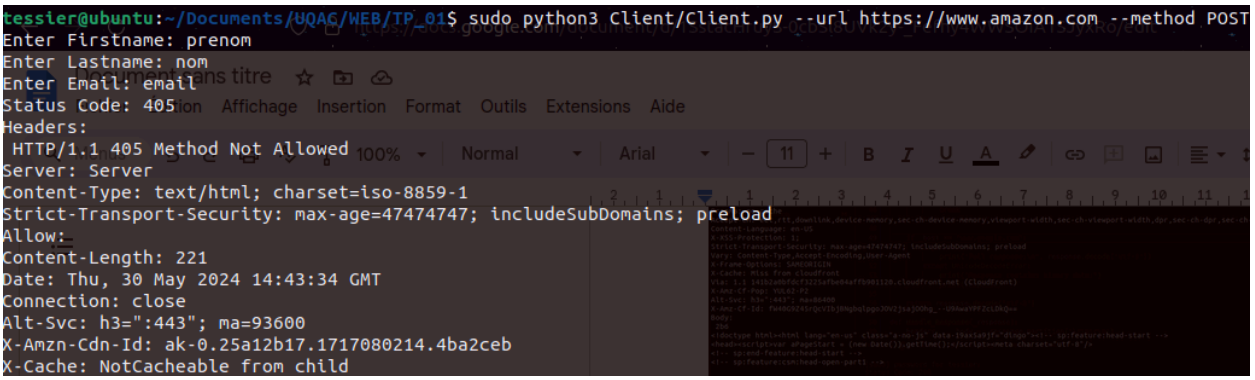
```
tessier@ubuntu:~/Documents/UQAC/WEB/TP_01$ sudo python3 Client/Client.py --url http://localhost/exemple.html --method POST
Enter Firstname: prenom
Enter Lastname: nom
Enter Email: email
Status Code: 200
Headers:
 HTTP/1.1 200 OK
Body:
 /exemple.html Create<html><body>Nom: nom<br>Prenom: prenom<br>Email: email<br></body></html>HTTP/1.1 200 OK

/exemple.html Create
```

En lançant le POST on peut envoyer de la data, ici c'est le nom, prénom et email. Ensuite comme les infos sont enregistrées et le file "exemple.html" (qui n'existait pas) est créé. En l'exécutant sur internet on peut voir ses données:

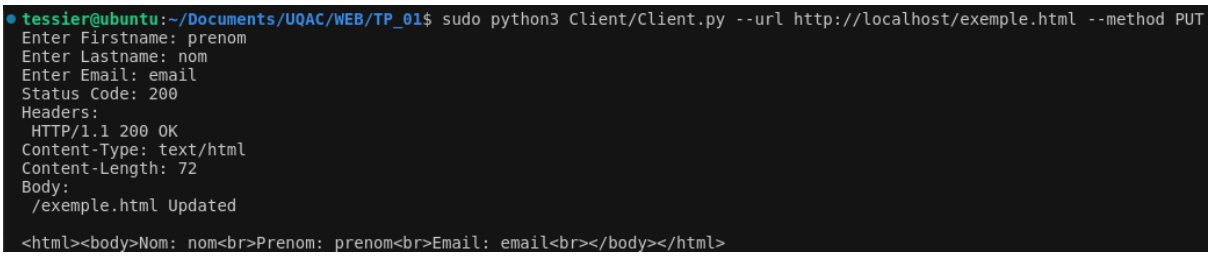
Nom: nom
Prenom: prenom
Email: email

Évidemment en le faisant sur un réel site cela ne donne qu'un message d'erreur:



Méthode PUT

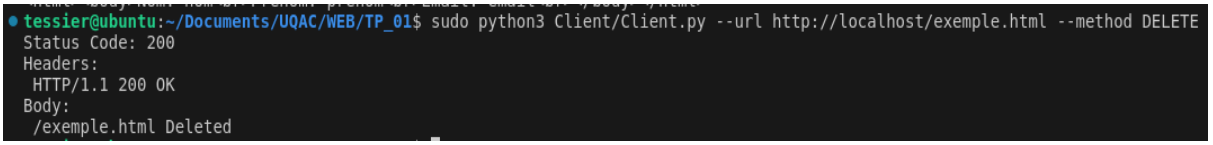
Cette méthode sert à uptade (ou créer si le fichier n'existe pas) un fichier, voici un exemple avec le file "exemple.html:



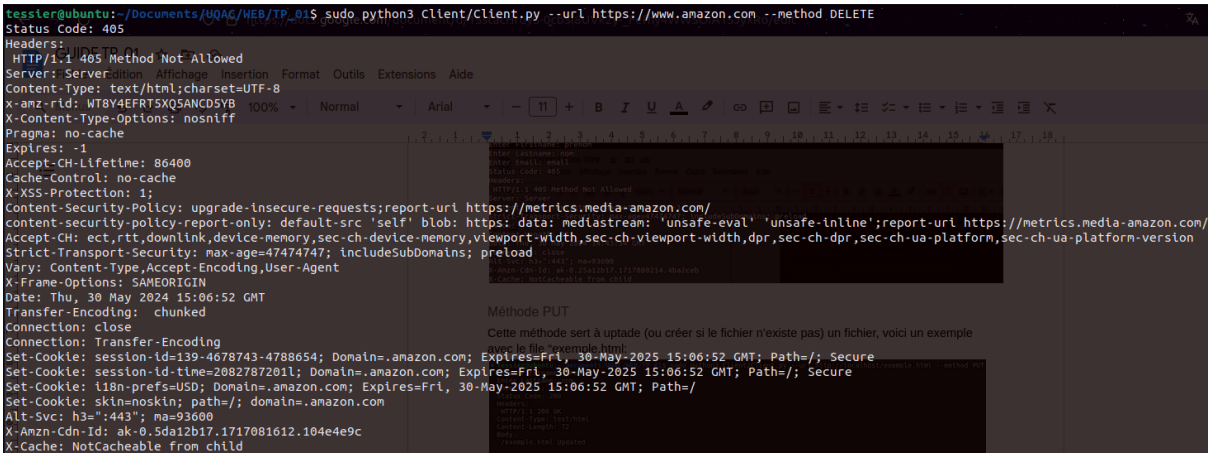
Comme ont peut le voir le fichier à été Updated. Comme pour le PUT nous avons une erreur si on fait un PUT sur un réel site.

Méthode DELETE

Cette méthode permet à supprimer un fichier, par exemple avec exemple.html voici ce que ça donne:



Comme vu au-dessus, le fichier à bien été supprimer, si on test cette méthode sur un réel site:

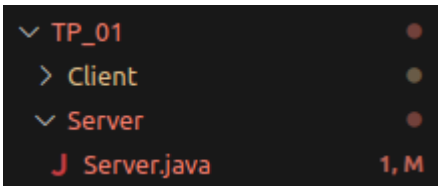


Partie SERVER

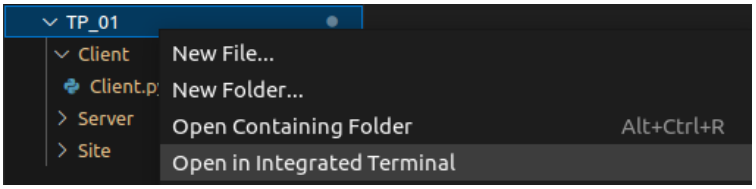
Utilisation

Lancement du Server

Le server est codé en java et se trouve dans le répertoire “Server”:



Pour lancer le server il suffit d’entrer cette ligne de commande (le sudo est pour ne pas encombrer des erreurs port 80):




```
tessier@ubuntu:~/Documents/UQAC/WEB/TP_01$ sudo java Server/Server.java
```

Une fois ceci fait, voyons comment se comporte le server.

Comportement du Server

Tout d’abord en lançant le server, vous aurez un message indiquant le server à bien été lancé:

```
tessier@ubuntu:~/Documents/UQAC/WEB/TP_01$ sudo java Server/Server.java
server is activate!
```

Actuellement le server attend une connection d’un client, pour se faire, dans le répertoire “Site” un “index.html” à été créé pour pouvoir utiliser les méthodes GET et POST:

Pour le lancer il suffit d’aller sur navigateur et lancer “http://localhost/index.html” et vous aurez alors un formulaire à remplir:

Entrez vos informations et soumettez

File name: /submitForm

Prenom:

Nom:

Email:

Soumettre

Le “File Name” est le nom du fichier où seront saves les données indiquées (Prenom, Nom, Email) et en appuyant sur “Soumettre” cela va créer le fichier.
De plus, dans le server vous verrez ceci apparaitre:

```
server is activate!
Client connected!
Request from Client: GET /index.html HTTP/1.1
Method from Client: GET
URI from Client: /index.html
Header from Client: Host: localhost
Header from Client: User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86 64
Header from Client: Accept: text/html,application/xhtml+xml,applicatio
Header from Client: Accept-Language: en-US,en;q=0.5
Header from Client: Accept-Encoding: gzip, deflate, br, zstd
Header from Client: Connection: keep-alive
Header from Client: Upgrade-Insecure-Requests: 1
Header from Client: Sec-Fetch-Dest: document
Header from Client: Sec-Fetch-Mode: navigate
Header from Client: Sec-Fetch-Site: none
Header from Client: Sec-Fetch-User: ?1
Header from Client: Priority: u=1
```

Ceci est la méthode GET utilisée lorsque vous lancez le site localhost/index.html comme on le voit dans “URI from Client” indiquant l’html utilisé.
Une fois que vous aurez entrez des informations la page localhost/index.html par exemple:

Entrez vos informations et soumettez

File name: /submitForm.html

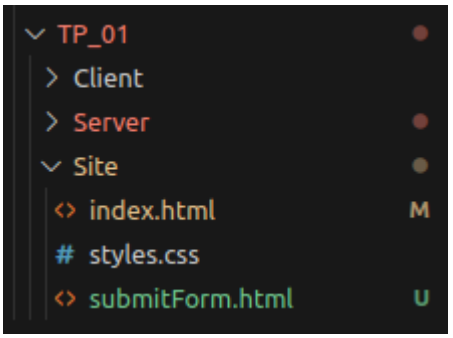
Prenom: prenom

Nom: nom

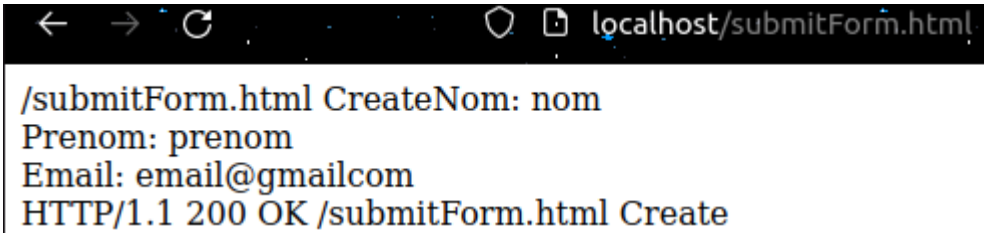
Email: email@gmailcom

Soumettre

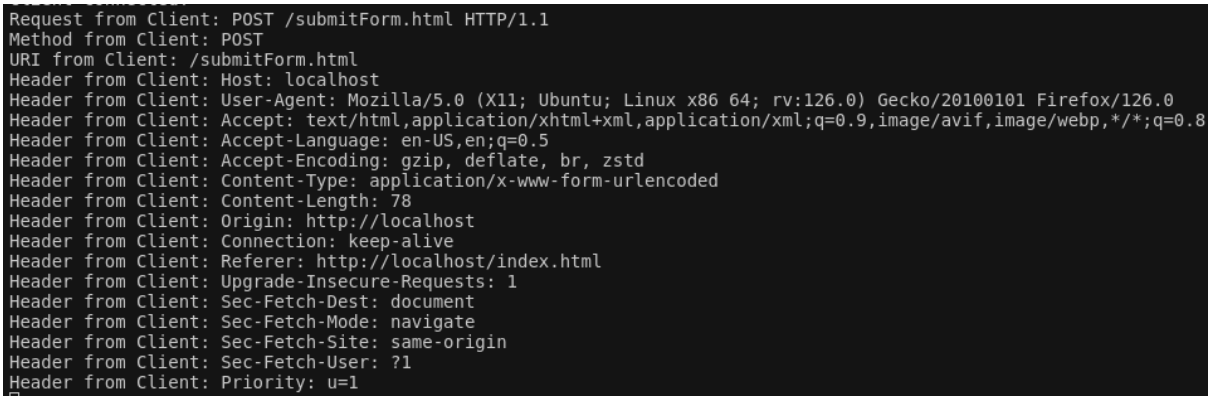
Appuyez sur “Soumettre” et cela créera le fichier dans le répertoire “Site”:



De plus, une fois créé vous serez redirigé vers votre fichier (qu’il soit html ou non) et verrez ses informations:



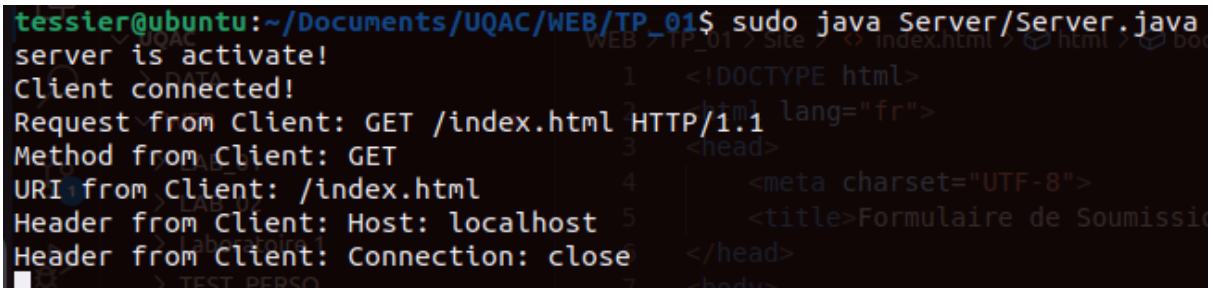
Et votre server vous renverra bien un POST:



Maintenant voyons comment utiliser notre Client.py et notre Server.java ensemble.

Communication Client.py et Server.java

Pour faire communiquer les deux code, il suffit de lancer le server.java et de lancer le client.py avec en url “http://localhost”, voici un exemple:
Server:



Client:

```
tessier@ubuntu:~/Documents/UQAC/WEB/TP_01$ sudo python3 Client/Client.py --url http://localhost/index.html --method GET
Status Code: 200
Headers:
 HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1082
Body:
 <!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Formulaire de Soumission</title>
</head>
<body>
    <h2>Entrez vos informations et soumettez</h2>
    <form method="POST">
        <label for="action">File name:</label>
        <input type="text" id="action" name="action" value="/submitForm"><br><br>

        <label for="firstname">Prenom:</label>
        <input type="text" id="firstname" name="firstname"><br><br>

        <label for="lastname">Nom:</label>
        <input type="text" id="lastname" name="lastname"><br><br>

        <label for="email">Email:</label>
        <input type="email" id="email" name="email"><br><br>

        <input type="submit" value="Soumettre" onclick="submitForm()">
    </form>

    <script>
        function submitForm() {
            var form = document.querySelector('form');
            var action = document.getElementById('action').value;
            form.action = action; // Définir l'action du formulaire basée sur l'entrée de l'utilisateur
        }
    </script>
</body>
</html>
```

Comme on le voit ici, on affiche ce que contient le fichier “index.html” dans le répertoire “Site” et le server renvoie bien un “GET” comme indiqué.
On peut aussi tester les autres méthode:
POST:
Server:

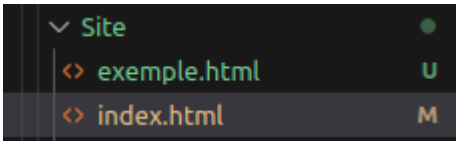
```
Client connected!
Request from Client: POST /exemple.html HTTP/1.1
Method from Client: POST
URI from Client: /exemple.html
Header from Client: Host: localhost
Header from Client: Connection: close
Header from Client: Content-Length: 41
```

Client:

```
tessier@ubuntu:~/Documents/UQAC/WEB/TP_01$ sudo python3 Client/Client.py --url http://localhost/exemple.html --method POST
Enter Firstname: prenom
Enter Lastname: nom
Enter Email: email
Status Code: 200
Headers:
 HTTP/1.1 200 OK
Body:
 /exemple.html Create<html><body>Nom: nom<br>Prenom: prenom<br>Email: email<br></body></html>HTTP/1.1 200 OK

/exemple.html Create
```

Le fichier html après “localhost” est le nom que vous souhaitez donner au fichier (ici exemple.html”, le server à bien pris en compte que c’était une méthode “POST”, le fichier à bien été créé:



PUT:
Server:

```
Client connected!
Request from Client: PUT /exemple.html HTTP/1.1
Method from Client: PUT
URI from Client: /exemple.html
Header from Client: Host: localhost
Header from Client: Connection: close
Header from Client: Content-Length: 33
```

Client:

```
tessier@ubuntu:~/Documents/UQAC/WEB/TP_01$ sudo python3 Client/Client.py --url http://localhost/exemple.html --method PUT
Enter Firstname: ki
Enter Lastname: ki
Enter Email: ki
Status Code: 200
Headers:
 HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 64
Body:
 /exemple.html Updated

<html><body>Nom: ki<br>Prenom: ki<br>Email: ki<br></body></html>
```

On peut aussi tester les autres méthode:
POST:
Server:

```
Client connected!
Request from Client: POST /exemple.html HTTP/1.1
Method from Client: POST
URI from Client: /exemple.html
Header from Client: Host: localhost
Header from Client: Connection: close
Header from Client: Content-Length: 41
```

Client:

Ici le fichier “exemple.html” à été mis à jour, et on peut voir que le server à bien compris que la méthode lancée à été PUT.

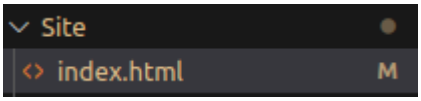
DELETE:
Sever:

```
Client connected!
Request from Client: DELETE /exemple.html HTTP/1.1
Method from Client: DELETE
URI from Client: /exemple.html
Header from Client: Host: localhost
Header from Client: Connection: close
```

Client:

```
tessier@ubuntu:~/Documents/UQAC/WEB/TP_01$ sudo python3 Client/Client.py --url http://localhost/exemple.html --method DELETE
Status Code: 200
Headers:
 HTTP/1.1 200 OK
Body:
 /exemple.html Deleted
```

Le fichier à été supprimé, le server l’a bien pris en compte et en regardant dans le fichier “Site” le fichier n’apparait plus:



Étapes Clients

1. Choisir un langage de programmation : Java, Python ou PHP

Langage utilisé : Python

2. Créer une connexion TCP avec un serveur web en utilisant les sockets

Code correspondant:

```
def Create_TCP_Connection(_host_link, _port):
      raw_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
      if _port == 443:
            context = ssl.create_default_context()
            SocketClient_ = context.wrap_socket(raw_socket,
            server_hostname=_host_link)
      else:
            SocketClient_ = raw_socket
      SocketClient_.connect((_host_link, _port))
      return SocketClient_
```

3. Envoyer une requête GET pour demander un fichier HTML depuis le serveur web

Code correspondant:

```
def Send_HTTP_Request(_sock, _method, _host, _path, _headers,
_data):
      header_lines = "\r\n".join(f"{key}: {value}" for key, value
in
      _headers.items())
      if _data:
            body = urllib.parse.urlencode(_data)
            request = f"{_method} {_path} HTTP/1.1\r\nHost:
            {_host}\r\n{header_lines}\r\nContent-Length:
            {len(body)}\r\n\r\n{body}"
      else:
            request = f"{_method} {_path} HTTP/1.1\r\nHost:
            {_host}\r\n{header_lines}\r\n\r\n"
      _sock.sendall(request.encode('utf-8'))
```

4. Recevoir la réponse du serveur, qui devrait inclure un code de statut HTTP, des en-têtes et le contenu HTML

Code correspondant:

```
def Receive_Response(_sock, _host):
      response = b""
      while True:
            part = _sock.recv(4096)
            if not part:
                  break
            response += part

      if _host == "www.google.com":
            try:
                  print("Full response:\n",
response.decode('utf-8'))
            except UnicodeDecodeError:
                  print("Response contains binary data:")
                  print(response)

      return response.decode('utf-8')
```

5. Analyser la réponse pour extraire le code de statut HTTP, les en-têtes et le contenu HTML

Code correspondant:

```
def Handle_Response(_response):
      headers, _, body = _response.partition('\r\n\r\n')
      status_line = headers.split('\r\n')[0]
      parts = status_line.split(' ', 2)
      if len(parts) < 3:
            raise ValueError("Invalid status line:
            {}".format(status_line))
      version, status_code, reason = parts
      status_code = int(status_code)
      return status_code, headers, body
```

6. Gérer les codes de statut HTTP, tels que 401 pour les fichiers inexistants. Si le code est 404, affichez le message d'erreur correspondant.

Code correspondant:

```
      if STATUT_CODE_ == 404:
      print("Error: Page not found (404 Not Found)")
```

7. Si le code de statut est un code de redirection (3xx), suivez l'URL de redirection fournie dans l'en-tête "Location" et répétez les étapes 3 à 6 avec la nouvelle URL.

Code correspondant:

```
elif STATUT_CODE_ in range(300, 400):
print("Redirection found.")
for line in HEADERS_.split('\r\n'):
     if line.lower().startswith('location:'):
          new_url = line.split(' ', 1)[1].strip()
          print("Redirecting to:", new_url)
          main(new_url, METHOD_)
```

8. Afficher les en-têtes de réponse du serveur, y compris les cookies et autres informations pertinentes.

Code correspondant:

```
print("Headers:\n", HEADERS_)
```

9. Si le code de statut est 200 (succès), affichez le contenu HTML sur la sortie standard.

Code correspondant:

```
if STATUT_CODE_ == 200:
print("Body:\n", BODY_)
```

10. Fermer la connexion TCP avec le serveur web.

Code correspondant:

```
SocketClient_.close()
```

11. Ajouter des options en ligne de commande pour permettre à l'utilisateur de spécifier l'URL, la méthode de requête (GET, POST, etc.), les en-têtes personnalisés et les paramètres de requête.

Code correspondant:

```
parser = argparse.ArgumentParser(description='HTTP Client')
parser.add_argument('--url', type=str, required=True,
help='URL
to request')
parser.add_argument('--method', type=str, default='GET',
help='HTTP method to use')
args = parser.parse_args()
```

```
main(args.url, args.method)
```

Étapes Server

1.Choisir un langage de programmation : Java, Python, PHP, etc.

Langage choisi: Java.

2.Créer un socket serveur pour écouter les connexions sur un port donné (80).

```
public Server() throws IOException
{
  ServerSocket_ = new ServerSocket(port_);
}
```

3.Accepter les connexions entrantes des clients et créer un nouveau thread ou une coroutine pour gérer chaque connexion individuellement.

```
public void Declaration_Param_Client() throws IOException
{
      connectionSocket_ = ServerSocket_.accept();
      inFromClient_ = new BufferedReader(new
      InputStreamReader(connectionSocket_.getInputStream()));
       outToClient_ = new
      DataOutputStream(connectionSocket_.getOutputStream()); //send
      server msg to client
      new Thread(ClientRunnable()).start(); //for check each client
      connexion and execute ClientRunnable
}
```

4.Lire la requête HTTP du client, qui devrait inclure la méthode (GET ou POST), l'URI, les en-têtes et, éventuellement, les données du formulaire pour les requêtes POST.

```
public void Code_Construction()
{
  try
  {
      System.out.println("Client connected!");

      MsgClientRequest_ = inFromClient_.readLine();
```

```
      if (MsgClientRequest_ == null)
      {
      return;
      }

      System.out.println("Request from Client: " +
      MsgClientRequest_);

      String[] PartsOfClientRequest = MsgClientRequest_.split(" ");
      String Method = PartsOfClientRequest[0]; //GET or POST
      String Uri = PartsOfClientRequest[1]; //ex : index.html
      System.out.println("Method from Client: " + Method + "\nURI
      from Client: " + Uri);

      while((MsgClientRequestHeader_ = inFromClient_.readLine()) !=
      null && !MsgClientRequestHeader_.isEmpty()) //if null, end
for
      the msg
      {
            System.out.println("Header from Client: " +
            MsgClientRequestHeader_);
      }

      switch(Method)
      {
      case "GET":
      GETRequest(Uri);
      break;
      case "POST":
      POSTRequest(Uri);
      break;
      case "PUT":
      PUTRequest(Uri);
      break;
      case "DELETE":
      DELETERequest(Uri);
      break;
      }
   }
   catch (IOException e)
   {
      System.out.println("Error client: " + e.getMessage());
   }
   finally
   {
      try
      {
```

```
      if (connectionSocket_ != null)
      {
      connectionSocket_.close();
      }
      }
      catch (IOException e)
      {
      System.out.println("Error closing client socket: " +
e.getMessage());
      }
  }
}
```

5.Analyser la requête pour extraire la méthode, l'URI, les en-têtes et les données du formulaire.

La méthode `Code_Construction()` ci-dessus gère cette étape.

6.Si la méthode est GET, vérifiez si le fichier demandé (par exemple, index.html) existe. Si le fichier existe, préparez une réponse HTTP avec un code de statut 200 (OK), les en-têtes appropriés et le contenu du fichier. Sinon, préparez une réponse avec un code de statut 401 (Non autorisé) et un message d'erreur.

```
public void GETRequest(String _Uri) throws IOException
{
  File HTML_FILE = new File("Site" + _Uri); // Remove leading '/'
      from URI

  if(HTML_FILE.exists() && !HTML_FILE.isDirectory()) //if index
      exist
  {
      FileInputStream HTMLInput = new FileInputStream(HTML_FILE);
      byte[] Data = new byte[(int) HTML_FILE.length()]; //create a
      byte tab
      HTMLInput.read(Data);
      HTMLInput.close();

      MsgServer_ = "HTTP/1.1 200 OK\r\nContent-Type:
      text/html\r\nContent-Length: " + Data.length + "\r\n\r\n";
      outToClient_.writeBytes(MsgServer_);
      outToClient_.write(Data);
      outToClient_.flush();
  }
```

```
  else //if file don't exist
  {
     MsgServer_ = "HTTP/1.1 404 Not Found\r\n\r\n" + _Uri + " Not
     Found";
     outToClient_.writeBytes(MsgServer_);
     outToClient_.flush();
  }
}
```

7.Si la méthode est POST, mettez à jour le fichier demandé (par exemple, index.html) avec les données du formulaire et préparez une réponse HTTP avec un code de statut 200 (OK), les en-têtes appropriés et le contenu du fichier mis à jour. Si le fichier n'existe pas, préparez une réponse avec un code de statut 401 (Non autorisé) et un message d'erreur.

```
public void POSTRequest(String _Uri) throws IOException
{
  StringBuilder payload = new StringBuilder();

  while(inFromClient_.ready())
  {
     payload.append((char) inFromClient_.read());
  }

  String data = URLDecoder.decode(payload.toString(), "UTF-8");
  Map<String, String> params = parseFormData(data);

  File HTML_FILE = new File("Site" + _Uri);

  try(FileWriter HTMLWrite = new FileWriter(HTML_FILE, false))
  {
     HTMLWrite.write("<html><body>");
     HTMLWrite.write("Nom: " + params.get("lastname") + "<br>");
     HTMLWrite.write("Prenom: " + params.get("firstname") +
"<br>");
     HTMLWrite.write("Email: " + params.get("email") + "<br>");
     HTMLWrite.write("</body></html>");
     HTMLWrite.flush();
  }

  if(HTML_FILE.exists() && !HTML_FILE.isDirectory())
  {
     MsgServer_ = "HTTP/1.1 200 OK\r\n\r\n" + _Uri + " Create";
     outToClient_.writeBytes(MsgServer_);
     FileInputStream fis = new FileInputStream(HTML_FILE);
```

```
      byte[] buffer = new byte[4096];
      int bytesRead;
      while ((bytesRead = fis.read(buffer)) != -1)
      {
      outToClient_.write(buffer, 0, bytesRead);
      }
      fis.close();
  }
  else
  {
      MsgServer_ = "HTTP/1.1 401 Unauthorized\r\n\r\n" + _Uri + "
Not
      Found";
  }

  outToClient_.writeBytes(MsgServer_);
  outToClient_.flush();
}
```

8.Envoyer la réponse HTTP au client.

Les méthodes `GETRequest`, `POSTRequest`, `PUTRequest`, et `DELETERequest` envoient toutes la réponse au client.

9.Fermer la connexion avec le client et répéter les étapes 3 à 9 pour les nouvelles connexions entrantes.

La méthode `Code_Construction()` ferme la connexion après avoir traité la requête et le serveur attend de nouvelles connexions dans une boucle infinie.