Webserver

Introduction

Webserver est un projet dont l'objectif est de vous faire coder un serveur capable de réceptionner, traiter, et répondre à des requêtes HTTP émanant de clients.

Le protocole HTTP détaille le format attendu des requêtes et des réponses HTTP, et votre programme devra s'y conformer.

Les clients vont se connecter au serveur, et lorsque la connexion entre un client et notre serveur sera établie, le client lui enverra des requêtes HTTP, qu'il devra traiter, avant de retourner une réponse HTTP.

Comme on va le voir plus loin, cette première étape (celle de la connexion entre un client et le serveur) est réalisée grâce à un socket dit « listener » (c'est à dire un socket qui a été modifié par la fonction `listen()` pour lui permettre de recevoir des connexions émanant de clients). Il y a autant de sockets listeners qu'il y a de ports sur lesquels notre serveur doit écouter.
La deuxième étape (l'échange de requêtes et de réponses entre le client et le serveur) est réalisée grâce à un autre socket, dit socket « client », propre au client. Il y a autant de sockets clients qu'il y a de clients avec lesquels notre serveur est en communication.

On lit et on écrit sur un socket de la même manière que sur un fichier : avec les appels système `read()` et `write()`.
Ces appels système peuvent être « bloquants » : par exemple, si je tente de lire un

file descriptor avec `read()` mais qu'il n'y a rien à lire (même pas EOF), l'opération reste en suspens, et on ne passe pas à l'instruction suivante.

Le sujet de Webserver impose une contrainte : le multi-threading est interdit.
Or, notre serveur doit servir plusieurs clients en même temps.
La conséquence directe de cette contrainte est de nous obliger à nous assurer, avant chaque opération de lecture ou d'écriture, qu'elle ne sera pas bloquante.
Si nous avions un thread par client, le fait d'être bloqué sur une opération avec un client n'affecterait pas les autres clients. Mais étant donné que le multi-threading est interdit, si nous nous retrouvions bloqué en train d'attendre une requête d'un client, par exemple, nous serions incapables de servir les autres clients en attendant.
Afin de nous assurer, avant chaque opération de lecture ou d'écriture, qu'elle ne sera pas bloquante, on utilise `select()`.
Cette fonction a pour but de repérer, parmi une liste de file descriptors (ou de socket descriptors), lesquels peuvent faire l'objet d'une opération read ou d'une opération write qui ne sera pas bloquante.

Attention : l'intégralité des opérations de lecture/écriture doivent être effectuées avec l'autorisation délivrée par `select()`.
Il y aura plus d'opérations de lecture/écriture au cours de l'exécution de votre programme que celles que je viens d'évoquer, et il ne faudra en oublier aucune : toutes devront soliciter l'autorisation de `select()` avant d'être effectuées.

Entrée dans le programme

Présentation de la classe Webserver

C'est une instance de cette classe qui va occuper le coeur de notre programme.
Elle va effectuer les 3 étapes centrales du processus :
- parser le fichier de configuration
- mettre en place des sockets listeners
- lancer la boucle infinie du processus

Elle va contenir les données suivantes :
- la liste des configurations de serveurs extraites du parsing du fichier de configuration
- la liste des sockets listeners
- la liste des clients
- le reading set utilisé par select
- le writing set utilisé par select

Parsing du fichier de configuration

Notre programme peut gérer plusieurs serveurs virtuels.
Ces serveurs présentent des configurations différentes, chacune étant définie dans le fichier de configuration.
Pour répondre aux requêtes des clients, on doit à chaque fois déterminer précisément à laquelle de ces configurations on doit se référer.

`listen` :

Quand un client établit une connexion, il l'établit avec un couple adresse IP/port.

Chacun de nos serveurs doit préciser le couple adresse IP/port sur lequel il écoute.

`server_names` :

Plusieurs serveurs peuvent écouter sur le même couple adresse IP/port (attention, il semblerait que le thread de correction pose problème à ce niveau, cela a été reporté sur discord).

Lorsqu'un client effectue une requête, la requête contient un header « Host ». On se réfère à la valeur de ce header pour choisir, parmi toutes les configs correspondant à un même couple adresse ip/port, la config appropriée : c'est à cela que sert le champ `server_names` d'une config.

On peut spécifier plusieurs `server_names` pour une même config.

Si le header Host de la requête est vide, ou si sa valeur ne correspond à aucun `server_name` connu, on utilise la config par défaut du couple adresse ip/port.

Cette config par défaut sera la première trouvée dans le fichier de config (selon le sujet).

`client_max_body_size` :

La taille maximale attendue pour le payload d'une requête client. Au-delà de cette taille, une erreur sera renvoyée.

`error_pages` :

Associe, pour certains codes d'erreur, des pages prédéfinies à retourner au client.

Chaque config de serveur va contenir une ou plusieurs sous-configuration que l'on appelle `location`.

Chaque `location` est associée à une route et lui associe une sous-configuration qui lui est propre.

Il doit y avoir, pour chaque config de serveur, une `location` par défaut, utilisée

lorsque l'URI de la requête ne correspondra à aucune route connue.

Chaque `location` définit les modalités suivantes :

- `root` : c'est le chemin qui sera utilisé pour obtenir les ressources demandées par le client. Par exemple, si le client demande la ressource `/images/patate`, et qu'on a une `location /images/`, dont le paramètre `root` a pour valeur `/data`, alors on va aller chercher dans notre système de fichier le fichier `/data/patate`. (si vous allez sur http://nginx.org/en/docs/beginners_guide.html, vous verrez que pour nginx, dans cette situation, il faudrait en fait récupérer le fichier /data/images/patate, et non /data/patate. Mais le sujet et le testeur de 42 ne sont pas d'accord avec ça.)
- `index` : le fichier retourné par défaut si la requête porte sur un directory et non sur un fichier
- `methods` : ce sont les méthodes gérées pour la `location`. Si la méthode de la requête du client n'est pas gérée, une erreur spécifique devra être retournée.
- `autoindex` : on ou off, selon qu'on souhaite qu'un client puisse accéder à la liste du contenu d'un directory.
- `upload_dir` : le path qui sera utilisé pour sauvegarder les documents uploadés par le client
- `cgi` : le binaire CGI qui sera utilisé pour les requêtes CGI
- `redirect` : équivalent de rewrite pour nginx.

Si l'un de ces paramètre est défini en dehors d'une `location` spécifique, on le considère comme appartenant à la `location` par défaut.

Mise en place des sockets listeners

On a maintenant une liste de configurations de serveurs.

Chacun écoute sur un couple adresse ip/port. On peut avoir plusieurs serveurs qui

écoutent sur un même couple adresse ip/port.
Pour chaque couple adresse ip/port distinct, il va falloir mettre en place un socket d'écoute qui réceptionnera les nouvelles connexions avec de nouveaux clients.

On met en place, donc, pour chaque couple adresse/port, un socket d'écoute.

Rappel : il faut distinguer deux types de sockets. On a d'une part les sockets « listeners », qui ont pour unique but d'écouter sur les ports établis par le fichier de configuration afin de détecter les demandes de connexions émanants des clients.
D'autre part, on a les autres sockets que l'on appellera les sockets « clients » : chaque fois qu'un client se connectera à notre serveur grâce à un socket listener, on lui créera son propre socket client. Contrairement aux sockets listeners dont le but est de réceptionner les connexions avec les nouveaux clients, les sockets « clients » ont pour but d'échanger des données avec ces clients, c'est à dire réceptionner les requêtes et envoyer les réponses.

Lancement de la boucle

On met en place une boucle dont on ne sortira que si l'on reçoit un signal d'interruption.
A chaque tour de boucle, on doit :
- construire la liste de fd utilisée par select
- appeler select
- lire et écrire dans les fd autorisés par select

Boucle

Construction de la liste des fd

`select()` utilise deux ensembles (ou « set ») de fd : un pour les opérations read, et un autre pour les opérations write.

Avant chaque appel de `select()`, on doit construire ces deux ensembles :
- le reading set contient l'ensemble des file descriptors sur lesquels on voudrait pouvoir effectuer une opération de lecture
- le writing set contient l'ensemble des file descriptors sur lesquels on voudrait pouvoir effectuer une opération d'écriture

On va voir dans le détail quels file decriptors mettre dans quel set.

Après un appel de `select()`, ces deux sets sont altérés : cela signifie qu'avant d'effectuer le prochain `select()`, au prochain tour de boucle, il faudra veiller à ré-initialiser et reconstruire ces deux sets.

Après l'appel de `select()`, on peut savoir si un file descriptor est disponible en lecture ou s'il est disponible en écriture.

Pour toutes ces étapes on utilise les macros suivantes :
- `FD_ZERO` pour réinitialiser un set
- `FD_SET` pour ajouter un file descriptor à un set
- `FD_ISSET` pour savoir si un file descriptor a été sélectionné par `select()` comme étant disponible en lecture ou en écriture

Voici les socket descriptors et file descriptors qui devront être ajoutés, avant chaque appel de `select()`, au reading set :

- l'ensemble des sockets listeners : en effet, c'est lorque l'un de ces sockets sera identifié comme étant disponible en lecture que l'on saura qu'il y a une connexion en attente dessus
- l'ensemble des sockets clients : s'il y a quelque chose à lire sur un socket client, c'est une requête HTTP
- l'ensemble des fichiers que l'on est en train de lire afin d'en envoyer le contenu à des clients dans le cadre du traitement de requête GET
- l'ensemble des fichiers temporaires contenant des outputs de script CGI qui sont en train d'être lus (voir plus loin)

Voici les socket descriptors et file descriptors qui devront être ajoutés, avant chaque appel de `select()`, au writing set :
- l'ensemble des sockets clients : avant de pouvoir envoyer une réponse à un client, il faut s'assurer que son socket client est disponible en écriture
- l'ensemble des fichiers temporaires contenant des payloads de requête qui sont en train d'être écrits (voir plus loin)

Lire les fd

Après l'appel de `select()`, on parcourt l'ensemble des sd et des fd que l'on avait ajoutés au reading set, et on vérifie avec `FD_ISSET` s'ils sont disponibles en lecture.

S'il y a quelque chose à lire sur un socket listener, c'est qu'un nouveau client veut se connecter : il faut accepter la connexion, créer un socket client et un nouvel objet Client. Tout cela est détaillé un peu plus bas.

S'il y a quelque chose à lire sur un socket client on lance le parseur de requête (voir la section suivante : Parsing d'une requête).

On va voir plus loin, dans la section sur le traitement des requêtes, dans quel contexte on manipulera les fichiers à retourner aux clients et les fichiers temporaires d'output cgi.
Il est simplement important de garder en tête que ces manipulations devront se faire tout en vérifiant, avant chaque `read()`, que le file descriptor a bien été vérifé par `select()`.

Si un problème survient, le client est supprimé et son socket client est close.

Ecrire les fd

Si un socket client est disponible en écriture, on peut effectuer un `write()` afin de lui envoyer un buffer de réponse (s'il y a un buffer réponse en attente pour ce client, bien évidemment).

Si un fichier temporaire est disponible en écriture, on peut effectuer un `write()` afin de poursuivre l'écriture de ce fichier qui, on va le voir plus loin, a pour rôle de réceptionner les payloads importants des requêtes HTTP.

Acceptation d'une nouvelle connexion client

Lorsqu'on a déterminé, grâce à un `FD_ISSET`, qu'il y avait quelque chose à lire sur un socket qui s'avère être un socket listener, cela signifie qu'un nouveau client veut se connecter à notre serveur.

Comme expliqué précédemment, les échanges avec ce client ne se feront pas via le socket listener. On va créer un socket propre à ce client, grâce auquel on va

communiquer avec lui.
Cela se fait avec `accept()`.

On crée un nouvel objet Client que l'on va ajouter à la liste des clients de notre instance de Webserver.

On doit passer deux éléments au constructeur de Client :
- le socket descriptor du socket client que l'on vient de créer avec `accept()`
- la liste des configurations pouvant s'appliquer aux requêtes de ce client

C'est dans cette liste de configurations qu'on ira piocher pour répondre aux requêtes de ce client : ce sont toutes les configs qui ont le même couple adresse ip/port que l'adresse ip/port dont le client s'est servi pour se connecter à notre Webserver.

Parsing d'une requête

Lecture du socket

Après l'appel de select, si un socket descriptor a été repéré comme pouvant faire l'objet d'une opération read non bloquante, on a le droit d'effectuer UNE et une seule opération read sur lui.
On récupère donc un buffer à partir du socket descriptor, avec `read()` (ou `recve()` si vous préférez).

Ce buffer peut contenir une requête HTTP incomplète, une requête complète, ou même plusieurs requêtes.
On va le parser pour en extraire les éléments de cette (ou ces) requête(s).

Il est possible qu'après ce parsing subsistent des données qui n'auront pas pu être traitées car elles ne constituaient pas un élément de requête complet.
Il faut conserver ces données non traitées. Elles pourront peut-être être traitées lorsqu'on récupérera des données supplémentaires à la prochaine lecture du socket.

Attention : les données lues sur le socket doivent être stockées sous la forme d'un container de bytes et non d'une string, au cas où on aurait à manipuler des binaires (contenant donc des 0 qui pourraient être interprétés comme des caractères nuls).

Présentation de la classe Request

Une requête HTTP comporte les éléments suivants :
- une request line
- des headers
- un éventuel payload
- un éventuel trailer

Notre classe Request va donc présenter les attributs :
- `request_line`
- `headers`
- `payload`

(Le trailer contenant des headers supplémentaires, son contenu sera stocké dans l'attribut `headers` comme n'importe quel autre header.)

Pour ce qui est des headers, je conseille de les stocker sous la forme d'une hash table.

La request line est elle-même composée de trois éléments :

- la méthode
- la request target
- la version du protocole HTTP

On créera donc une classe ou une structure RequestLine qui présentera les 3 attributs en question. L'attribut `request_line` de la classe Request sera une instance de cette classe.

Pour répondre à une requête, il faut également déterminer exactement à quelle configuration de serveur se conformer.
On aura donc également un attribut `config` contenant cette information.
(On verra qu'on utilise les informations contenues dans les headers de la requête pour déterminer quelle configuration choisir, parmi toutes celles pouvant s'appliquer au client à l'origine de cette requête.)

Comme on va le voir plus loin, on va avoir besoin, pour stocker le payload de la requête, qui peut être très lourd, d'un fichier temporaire.
Cela signifie un attribut `tmp_file_fd` pour stocker le file descriptor de ce fichier.

Enfin, pour suivre l'évolution de la requête au cours de son parsing et de son traitement, je conseille la présence d'un attribut `status`, qu'on verra plus en détail un peu plus loin.

Bien sûr, vous pouvez penser à d'autres attributs qui peuvent vous être utiles afin de faciliter la manipulation de l'objet Request.

Stockage des requêtes reçues d'un client

Notre objet client peut contenir plusieurs requête en attente de traitement.
En revanche, il contient, au plus, un seul objet réponse : on ne traite qu'une requête à la fois.

Personnellement j'avais implémenté une liste de paires requête/réponse.
Cette liste permettait de contenir toutes les requêtes qui ont été constituées à partir des données récupérées sur le socket du Client et qui sont en attente de traitement.
A chaque objet requête était associé, au sein d'une paire, un objet réponse qui allait être rempli au cours du traitement de la requête, et qui finalement allait servir à répondre au client via le socket.

Vous pouvez aussi implémenter, comme attribut de l'objet Client, une liste d'objet Request, et un seul attribut Response, car comme expliqué plus haut : un client peut stocker plusieurs requêtes en attente de traitement, mais n'utilisera qu'un seul objet Response à la fois.

Au fur et à mesure que l'on parse les données lues sur le socket descriptor du client, on va créer et ajouter de nouveaux éléments à la liste pour stocker toutes les requêtes en provenance de ce client.

Si la première requête de la liste a été complètement réceptionnée, on va pouvoir commencer à traiter cette requête, dans le but de remplir l'objet Response. A partir du moment où on est en train de remplir l'objet Response associé à un objet Request, on considère que cette requête est « en cours de traitement ».

Après avoir complété un objet Response, on peut alors utiliser cet objet pour répondre au client, via son socket.

Une fois que la réponse associée à une requête a été entièrement expédiée au Client, on va pouvoir supprimer la requête.

A chaque tour de boucle, on examine le statut de la première requête de la liste de chaque Client, et si la requête a été entièrement réceptionnée mais n'a pas commencé à être traitée, il faut lancer la procédure de traitement de la requête.

Parsing des données lues sur le socket

Afin de faciliter le traitement de la requête, je conseille, comme mentionné plus haut, l'utilisation d'un attribut de l'objet Request qui indique le statut de la requête, à savoir :
- si la request line a été parsée
- si l'ensemble des headers ont été parsés mais qu'un payload est attendu
- si le payload a été entièrement parsé mais qu'un trailer est attendu
- si l'intégralité de la request a été parsée mais qu'elle n'a pas encore été traitée
- si la requête est en cours de traitement

Lorsqu'on est en train de parser les données récupérées sur le socket d'un Client, si on s'aperçoit qu'on vient de compléter une requête, mais qu'il reste des données à parser, on doit créer un nouvel objet Request à ajouter à la liste des requête du Client.

Pour faciliter le parsing des données récupérées sur le socket, il nous faut des fonctions de reconnaissance de pattern, qui vont pouvoir indiquer si oui ou non les données récupérées sur le socket contiennent une request line complète, ou un header complet par exemple.

Parsing d'une request line

Si on a déterminé que les data lues sur le socket du client contenaient une request line complète, on peut commencer à remplir notre objet Request avec les éléments de cette request line, à savoir :

- la méthode
- la request target
- la version du protocole HTTP

Parsing des headers

Tant que l'on reconnaît, parmi les data récupérées sur le socket, un pattern correspondant à un header entièrement réceptionné, on appelle une fonction qui collecte ce header et l'ajoute à notre hash table de headers.

Au fur et à mesure que l'on reçoit les headers, on doit s'assurer de leur viabilité.

Certains headers nous fournissent des informations qui sont directement exploitables : Host, notamment, nous permet de sélectionner la configuration de serveur exacte que l'on va utiliser par la suite pour répondre à la requête.

A ce stade, certaines erreurs peuvent déjà être repérées : une absence de header Host, par exemple, est considérée comme une erreur.

Après avoir reçu l'ensemble des headers (ce que l'on repère grâce à la ligne vide séparant les headers du payload dans une requête HTTP), on regarde si un payload doit être réceptionné.
Pour savoir si un payload est attendu, il faut regarder les headers Content-Length et Transfer-Encoding : s'il y a un header Content-Length et que sa valeur est supé-

rieure à 0, ou s'il y a un header Transfer-Encoding et que sa valeur est « chunked », cela signifie qu'on attend un payload.

Si aucun payload n'est attendu, on peut modifier le statut de la requête afin de la marquer comme entièrement réceptionnée.

Parsing du payload

Afin d'assurer la réception des payloads les plus lourds, je conseille l'utilisation de fichiers temporaires afin de stocker ces payloads.
L'écriture de ce fichier temporaire, comme toute opération d'écriture, doit se faire dans le cadre d'un `select`.
Lors du parsing, on stocke temporairement les données du payload dans un attribut de l'objet Request (attribut `payload`), et lorsque `select` nous en aura donné l'autorisation, on recopiera ces données dans le fichier temporaire.
(srcs/Webserver.cpp, ligne 194)

Il y a deux possibilités, pour un payload :
- qu'il soit encodé sous forme « chunked »
- qu'il ne le soit pas
On détermine cela en regardant la valeur du header Transfer-Encoding.

S'il ne l'est pas, on concatène simplement, au fur et à mesure qu'on reçoit le payload, les données reçues à l'attribut `payload` de l'objet Request.
On saura que l'on aura réceptionné l'intégralité du payload grâce à la valeur du header Content-Length.
(srcs/RequestParsing.cpp, ligne 143)

Si le payload est chunked, l'extraction du payload doit tenir compte de cet encodage : un payload encodé sous forme chunked est divisé en morceaux, et chaque

morceau est précédé d'une mention indiquant la taille du morceau. On sait qu'on a reçu le dernier morceau lorsqu'on reçoit un morceau de taille 0.
(srcs/RequestParsing.cpp, ligne 115)

Lorsque l'intégralité du payload a été reçue, on regarde si un trailer est attendu.
Si ce n'est pas le cas, on peut déclarer la requête complète.

Si ce n'est pas le cas, le contenu du trailer doit être récupéré pour être stocké dans l'attribut `headers`.
Un trailer constitue juste des headers supplémentaires.

Traitement d'une requête

Attention : avant de commencer à traiter une requête, il faut s'assurer qu'elle a bien été entièrement réceptionnée ET que son payload, s'il existe, a bien été entièrement recopié dans un fichier temporaire.

Attention : dès lors qu'on commence à traiter une requête, on doit le signaler en modifiant son attribut statut en conséquence, afin de ne pas risquer de traiter plusieurs fois la même requête.

La réponse que l'on devra envoyer sur le socket du client devra comporter les éléments suivants :
- Une status line qui contient la version du protocole HTTP, le code du status (ex: 200), et la « reason phrase » (ex: « OK »)
- Des headers
- Un éventuel payload

Le traitement d'une Request consiste au remplissage de l'objet Response qui lui est associé avec ces différents éléments.
L'envoi de la réponse (détaillé dans la section suivante) consistera à l'envoi sur le socket d'une version stringifiée des éléments de l'objet Response.

Présentation de la classe Response

Une réponse HTTP comporte 3 éléments :
- une status line
- des headers
- un éventuel payload

Cela implique la présence de 3 attributs correspondant.

La status line est elle-même constituée de :
- la version du protocole HTTP
- un status code
- une phrase explicative du status code appelée « reason phrase »

Afin d'optimiser l'I/O, je recommande très fortement d'encoder en chunked le payload de la plupart des réponses (les réponses à des requêtes GET notamment).
Pourquoi ?
Si vous devez envoyer un fichier très lourd sans l'encoder en chunked, vous devez :
- récupérer l'intégralité du fichier avec `read()`
- stocker tout ce contenu en mémoire
- ajouter un header à votre réponse qui aura pour valeur la taille de ce contenu

Ainsi, le client saura, grâce à la valeur de ce header, quand il pourra considérer que la réponse que vous lui avez envoyée est complète.
Problème : cela implique pour notre programme de devoir lire l'intégralité du fi-

chier avant de pouvoir commencer à l'envoyer, et surtout, cela implique de le stocker sous forme de buffer, ce qui va vite devenir problématique pour les fichiers très lourds.
Solution : utiliser l'encodage chunked. L'encodage chunked permet d'envoyer un payload morceau par morceau. Chaque morceau est précédé d'une ligne indiquant la taille du morceau, et lorsque le client reçoit un morceau de taille 0, il sait qu'il s'agissait du dernier et qu'il a reçu l'intégralité de notre réponse.
L'avantage de cette solution est de pouvoir commencer à envoyer le contenu du fichier au client avant même que notre programme l'ait entièrement lu, et de n'avoir jamais besoin de stocker en mémoire tout le contenu du fichier d'un coup. On peut se contenter de ne stocker qu'un buffer.

Je recommande donc l'utilisation des trois attributs suivants :
- un unsigned int `status_code` contenant le status de la réponse
- une hash table `headers` contenant les headers de la réponse
- une string `head` qui contiendra « l'en-tête » de la réponse (c'est à dire la status line et les headers) tel qu'il doit être envoyé : il s'agit donc d'une version stringifiée et formatée des informations contenue dans les deux attributs précédents.
- un vector de char `payload` qui contiendra le payload de la réponse à envoyer.
- un booléen `chunked` qui indiquera si le payload est encodé en chunked ou non.

On verra plus loin, dans la section sur l'envoi des réponses, comment expédier nos réponses sur les sockets des clients, selon qu'elles sont encodés chunked ou non.

Pour l'instant on va voir comment remplir l'objet Response associé à un objet Request.
Pour cela, on avise en fonction de la méthode de la Request : GET, POST, PUT, ou DELETE.

GET

Headers

On peut dès le début générer les headers à stocker dans l'attribut `headers` de l'objet Response.

On peut générer les headers suivants :

- Allow
- Content-Language
- Content-Length
- Content-Type
- Content-Location
- Date
- Last-Modified
- Location
- Retry-After
- Sever
- Transfer-Encoding
- WWW-Authenticate

Il est intéressant de générer immédiatement les headers, avant de commencer à récupérer le fichier demandé par le client.

Pourquoi ?

Si on génère dès à présent les headers, on peut immédiatement créer « l'en-tête » de la réponse (voir la section suivante sur l'envoi des réponse), et cette en-tête pourra être envoyée immédiatement.

On rappelle que dans une optique d'optimisation, on souhaite envoyer les réponses au fur et à mesure qu'on les construit.

Après avoir généré les headers, on doit s'intéresser au status code et au payload de notre réponse.
En premier lieu on doit analyser le path de la ressource à retourner au client.
La fonction `stat()` nous permet de savoir si un path est le path d'un directory, d'un fichier standard, ou si ce path est invalide.

Path invalide

On peut établir que le code de la réponse renvoyée au client pour cette requête sera 404 NOT FOUND.

Pour ce qui est du payload, reportez vous au paragraphe consacré à la gestion des erreurs un peu plus bas. (srcs/Client.cpp, ligne 416)

Path de directory

Si le path est celui d'un directory, on regarde si la configuration de serveur utilisée indique que l'autoindex est activé.

Si ce n'est pas le cas, on retourne, comme plus haut, une réponse 404 NOT FOUND.

Sinon, le status code est 200 OK, et on doit générer nous-même la page demandée.

Path de fichier standard

Si le path est celui d'un fichier standard, il va falloir ouvrir ce fichier. (srcs/Client.cpp, ligne 468)

Le file descriptor obtenu va pouvoir être ajouté aux file descriptors monitorés par `select()` et dès qu'il sera disponible en lecture, on pourra en extraire les données pour les envoyer au client.
(srcs/WebServer.cpp, ligne 174)

Lorsque `select()` nous autorise à procéder à la lecture de ce fichier, le mieux est de récupérer son contenu tout en l'encodant en chunked, ce qui va nous permettre de l'expédier au fur et à mesure dans la réponse retournée au client. (srcs/Client.cpp, ligne 490)

L'envoi du payload (détaillé plus bas) va commencer avant que l'on ait terminé de lire l'intégralité du fichier.

PUT

Une requête PUT a pour but d'ajouter, ou de remplacer une ressource dans notre système de fichier.
Le nom de la ressource est spécifié dans la request target de la requête.
Si la ressource existe déjà, son contenu doit être remplacé par celui du payload de la requête.
Si elle n'existe pas déjà, elle doit être créée à partir du contenu du payload de la requête.

Ce payload a été récupéré au cours du parsing de la requête et stocké sous la forme d'un fichier temporaire.
Il suffit donc d'utiliser `rename()` afin de donner au fichier temporaire le nom approprié.

En revanche, si une erreur survient (par exemple, si le nom de la ressource s'avère être le nom d'un directory, ou si le `rename()` échoue), il faudra penser à supprimer ce fichier temporaire.

Le code réponse à retourner au client dépend de si la ressource a été créée ou modifiée :
- CREATED si la ressource a été créée
- NO CONTENT dans le cas contraire

POST

On va parler ici uniquement des requêtes POST qui ne portent pas sur un script CGI.

Elles sont inutiles, à proprement parler : une requête POST a pour but de fournir à un script des données à traiter. Or ici on va fournir des données à une ressource qui n'est pas à même de les traiter. Cela ne sert à rien, mais si on reçoit une requête de ce type il faut tout de même répondre quelque chose.

On doit retourner un status NO CONTENT, et générer quelques headers :
- Server
- Date
- Content-Type (text/html)
- Content-Length (0)

Cas particulier des CGI

…

DELETE

Comme dans le cas d'une requête GET, on doit d'abord s'assurer de la validité du path de la ressource que le client demande à effacer.

Si ce path est invalide, on retournera une erreur NOT FOUND (voir le paragraphe suivant, sur les réponses d'erreur).

On peut supprimer le fichier demandé avec `unlink()`. Si l'opération échoue, on doit retourner une erreur FORBIDDEN.

Si tout se déroule bien, on renverra un code NO CONTENT, et on générera les headers suivants :

- Server
- Date
- Content-Type (text/html)
- Content-Length (0)

Cas d'erreurs

Il faut une fonction qui va gérer la construction de l'ensemble des réponses correspondant à des cas d'erreur.

Une erreur peut survenir à divers moments, on aura donc une multitude de chemins menant à cette fonction, et il est possible que lorsqu'on arrive dans cette fonction, notre objet fonction ait déjà commencé à être rempli.
La première étape consiste donc à le nettoyer en supprimant l'ensemble des headers qu'il contient peut-être.
Attention : bien sûr, le status de la réponse ne doit pas être effacé.

Ensuite, on a deux cas de figure possibles :

- la configuration du serveur prévoit une page d'erreur adaptée à la situation
- on doit générer nous même le payload de notre réponse avec un contenu par défaut

Page d'erreur prévue par la configuration

Si la configuration de serveur correspondant à la requête que vous êtes en train de traiter propose un path vers une page d'erreur correspondant au code d'erreur auquel vous avez affaire, vous devez ouvrir ce fichier afin d'envoyer son contenu au client.

A partir de là, cela se déroule exactement comme si on était dans une requête GET qui aurait porté sur le fichier en question : vous allez générer les headers de la même manière, et envoyer le contenu de ce fichier encodé en chunked.

Envoi d'un payload par défaut généré par le serveur

Il vous faut générer le payload à retourner au client.

Vous pouvez écrire ce que vous voulez : faites une page html mentionnant le code d'erreur.

Ce payload étant très court, rien ne justifie de l'envoyer en chunked.

Pour ce qui est des headers à générer pour ce type de réponse, on pourra se contenter de :

- Server
- Date
- Content-Type (text/html)
- Content-Length

Si l'erreur est l'une des suivantes, on pourra ajouter des headers supplémentaires adaptés à la situation :

- METHOD NOT ALLOWED : ajout d'un header Allow
- FORBIDDEN : ajout d'un header WWW-Authenticate
- SERVICE UNVAILABLE : ajout d'un header Retry-After

Envoi d'une réponse

Construction de l'en-tête de la réponse

Au cours du traitement de la requête, on a rempli l'objet Response avec les éléments suivants :

- un status code
- une hash table de headers

On va maintenant devoir formater ces informations sous la forme d'une string que l'on enverra sur le socket du client.

Le format à respecter est le suivant :

```
HTTP/1.1 200 OK
Content-Type: text/html
Date: Wed, 10 Aug 2021 09:23:25 GMT
Content-Length: 220
```

On commence avec la status line qui doit comporter la version du protocole HTTP, le status code, et la reason phrase qui correspond au status code, le tout séparé par des espaces.
Ensuite on passe à la ligne (CRLF).
Puis on ajoute chaque header avec le format `nom_du_header: valeur_du_header`, suivi à chaque fois d'un retour à la ligne (CRLF).
Enfin, on ajoute un dernier retour à la ligne pour marquer la fin des headers.

Envoi de la réponse sur le socket

Si `select()` nous indique que le socket d'un client est disponible en écriture, on peut lui envoyer une réponse.

Attention : il faut bien sûr s'assurer d'avoir quelque chose à envoyer sur le socket.

On va envoyer dans un premier temps l'en-tête de la réponse (c'est à dire une string contenant la status line et les headers), puis le payload, s'il y en a un, en distinguant deux cas :
- si le payload est encodé en chunked
- s'il ne l'est pas

Il vous faut déterminer quand la réponse a été entièrement expédiée :
- si la réponse ne contenait pas de payload, elle est entièrement expédiée dès lors que l'en-tête a été entièrement envoyée
- s'il y a un payload encodé en chunked, la réponse est entièrement expédiée dès lors que le dernier morceau (celui de taille 0) a été entièrement envoyé
- s'il y a un payload non encodé, la réponse est entièrement expédiée dès lors que l'on a atteint la taille du payload

Lorsqu'une réponse a été entièrement expédiée, on peut détruire la requête à laquelle elle était associée.