



Documentation Webserv

Introduction

Ce document a pour but de rassembler des explications et ressources utiles dans la bonne réalisation de ce projet de groupe.

Liens Utiles

Github : <https://github.com/lobbyra/webserve>

Miro : <https://miro.com/welcomeonboard/ltD8UP...>

Ressources normes RFC :

- [7230 : Message Syntax and Routing](#)
- [7231 : Semantics and Content](#)
- [7232 : Conditional Requests](#)
- [7233 : Range Requests](#)
- [7234 : Caching](#)

RFC 3875 - The Common Gateway Interface (CGI) Version 1.1

Network Working Group D. Robinson Request for Comments: 3875 K. Coar Category: Informational The Apache Software Foundation October 2004 The Common Gateway Interface

<https://tools.ietf.org/html/rfc3875#section-4>

Liens divers :

[Socket Programming in C/C++ \(Geeks For Geeks\)](#)

Sommaire

Introduction

Liens Utiles

Quels besoins notre projet doit il remplir ?

- C'est quoi HTTP ?

- C'est quoi un serveur HTTP ?

- Pourquoi avons-nous créé HTTP au cours de l'histoire ?

- Quelles sont les différences entre HTTP d'aujourd'hui et HTTP à sa création ?

- Apache et Nginx, c'est quoi du coup ?

- Grossièrement, comment se décompose un serveur web ?

Réalisation du projet

- Documentation

- Preuves de concepts

- Découpage logique

Notions abordées

Partie client

- Comment un programme fait une requête à un serveur HTTP ?

- Quelles sont les types de requêtes à prendre en compte dans le cadre de c...

Partie serveur

- Comment un programme attend une requête HTTP ?

- Intro / Ressources utiles

- Approches techniques dans le cadre de notre projet

- De quoi est composée une réponse HTTP ?

Relation client-serveur

- TCP / UDP, un lien avec ce projet ou transparent ?

- Comment assurer la transmission client-server ?

- Negotiation client-server

Méthode de requêtes

- Liste des methodes

Types de headers

Types de réponses

Fichier de configuration

Les mots clés à implémenter :

CGI

- C'est quoi CGI ?

- C'est quoi la différence entre CGI et FastCGI ?

- Comment un CGI se matérialise-t-il en code ?

- C'est quoi les Méta-variables et comment les gérer ?

- Doit-on envoyer les headers en entrée du CGI ou juste le body ?

- Comment lire la réponse d'un CGI ?

- Compiler et executer CGI

- Ubuntu (20.x testé)

- Mac (mojave testé)

- Appeler le binaire

- I/O

- Organisation de groupe

- Communication & Structure

- Code

- Github

- Noms de fichiers

- Compilation utilisée

- Formatage du code

Quels besoins notre projet doit il remplir ?

C'est quoi HTTP ?

HTTP est l'acronyme de Hypertext Transfer Protocol ou Protocol de transfert d'Hypertexte. C'est un protocole de communication informatique basé sur le modèle client-serveur. Ce protocole est le fondement du World Wide Web que l'on connaît aujourd'hui. Il permet une communication externe (Internet) ou interne (intranet) entre plusieurs ordinateurs.

Ce protocole permet de transporter toutes sortes de données. Des pages web à sa création puis au transfert de fichier via FTP jusqu'à aujourd'hui où il permet le streaming de vidéo et de musique. Son développement s'est fait en parallèle des architecture réseau permettant toujours plus de débit et donc de contenu plus lourd.

C'est quoi un serveur HTTP ?

Un serveur HTTP est un serveur informatique répondant à des requêtes sous la forme de relation client-serveur. Il peut prendre la forme d'un ordinateur (ex illustration première page) contenant un logiciel qui va répondre à des requêtes de protocole HTTP. Ce terme est utilisé majoritairement pour classer ce genre de logiciel. Nginx et Apache sont des serveurs HTTP. Ici on parle de serveur HTTP sous la forme de logiciel. Il est pensé pour être disponible en permanence, à recevoir et répondre à des requêtes simultanément venant des clients.

Pourquoi avons-nous créé HTTP au cours de l'histoire ?

Ce protocole a été pensé pour créer le World Wide Web. Ce réseau a pour but de connecter et de communiquer avec des gens et des ressources du monde entier. Les premières institutions à mettre en place cette communication sont l'armée Américaine et les grandes universités du monde pour partager des études.

Quelles sont les différences entre HTTP d'aujourd'hui et HTTP à sa création ?

Le plus grand changement a été de créer une surcouche sécurisée créant nouveau protocole nommé HTTPS. Il permet de chiffrer les communications entre le client et le serveur pour protéger contre l'analyse des communications contenant des informations sensibles comme les mots de passe ou codes bancaires. Mais plus globalement c'est l'utilisation que l'on en fait qui a changé. Passé du simple échange de page HTML (comme ça) à du streaming de film en 4K.

Apache et Nginx, c'est quoi du coup ?

Ce sont les fameux logiciels cités plutôt qui sont la partie software d'un serveur HTTP. Ils attendent des requêtes de clients et peuvent y répondre simultanément.

Apache et Nginx sont les deux logiciels les plus utilisés en tant que serveur HTTP. Apache partageant 55% de la globalité des sites sur internet mais Nginx est le plus utilisé dans les 1000 sites les plus actifs d'internet.

En gros Nginx > Apache mais cet article est à lire pour saisir les différences.

Il explique très bien l'historique et les différences techniques de ces deux logiciels. Ici, le but de ce projet est plutôt de se rapprocher d'Nginx. C'est pourquoi, pour les comportements non spécifiés dans le sujet, nous nous baserons sur le comportement d'Nginx. Il en est de même pour la manière de configurer le serveur, la syntaxe et les mot-clés d'Nginx seront utilisés.

Grossièrement, comment se décompose un serveur web ?

En premier lieu il y a le fichier de configuration qui va décrire sur quels ports écouter et comment réagir aux requêtes. Vous avez vu à peu près comment ce fichier se décompose dans l'article.

Une fois ce fichier de configuration en mémoire, le programme va se mettre en pause dans l'attente d'une requête. Lorsqu'une requête est reçue, l'URL de cette requête va être comparée aux différents blocs de location pour déduire les paramètres à appliquer sur la gestion de cette requête.

Enfin quand la requête est traitée ou est en attente indépendamment du serveur web, retour au point de départ dans l'attente d'une nouvelle requête.

requete.

Réalisation du projet

Documentation

(Ce que vous êtes en train de lire)

Preuves de concepts

Cette partie n'est pas une application stricte de la notion de POC car ici il n'y a pas d'innovation. Mais dans tous les cas ces notions sont nouvelles, c'est pourquoi on parle de POC ici.

Les preuves de concept sont des codes que nous allons faire pour essayer une notion que l'on ne connaît pas en montrant concrètement comment la notion en question fonctionne. Cela a pour but de servir au dev qui réalise la POC pour apprendre et synthétiser la nouvelle notion et d'expliquer en montrant une démonstration de la notion concrètement.

Enfin les POC vont nous servir à avoir de l'avance dans la construction de la structure du code. Mieux savoir quand et comment on va implémenter la notion.

Il y a cinq POC, voici leurs themes et leurs liens Github :

- [Dossiers](#) : Les fonctions de gestions de dossiers.
- [Mini client HTTP](#) : Réaliser un mini client pour découvrir les sockets et l'envoi de requêtes en C.
- [Mini serveur HTTP](#) : Réaliser un mini server pour découvrir comment attendre une requête sur un socket et comment y répondre.
- [Fonction select\(\)](#) : Découverte de la programmation asynchrone.
- [strtime](#) : Decouverte des fonctions de parsing du temps. (string vers mémoire et inversement)

Découpage logique

Structurer, coder, recommencer.

Avancement progressif dans le code :

- On essaye de voir quelles choses sont a faire en premier dans ce que nous avons pas fais.
- On fais une structure logique ([Miro](#)) de ce que l'on veut coder.
- On choisi qui va le coder.
- On le code.
- On recommence.

Notions abordées

Liste non exhaustive de toutes les notions abordées.

Ecrire de la documentation nous sert à nous auto-expliquer la notion que nous avons étudié. Cela sert aussi d'archive pour se remémorer une notion bien après l'avoir appliquée si jamais on a besoin de revenir dessus au cours du projet ou pour l'examen.

Merci d'accompagner, le plus possible, ces explications de ressources utiles (Sites, Vidéo YT, post Stack Overflow, etc..)

Partie client

Comment un programme fait une requête à un serveur HTTP ?

Voici un exemple écrit StackOverflow expliqué d'une requête HTTP en C : [ICI](#)

Un exemple vidéo expliqué par Jacob Sorber (<- Dieu vivant) : [Vidéo](#)

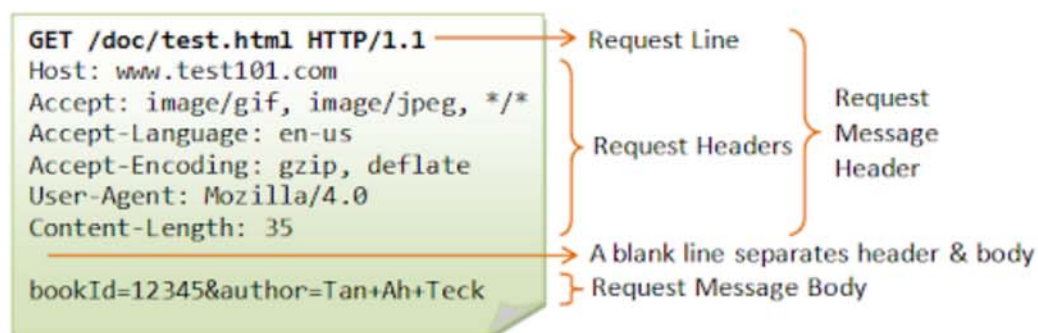
De quoi est composée une requête HTTP ?

Sources :

[RFC_CLIENT_MESSAGING](#)

[RFC_MESSAGE_FORMAT](#)

Prenons cette illustration :



Request line :

La **request line** comporte, dans l'ordre, les éléments suivants :

GET : Méthode, nature de la requête.

/doc/test.html : Chemin de la ressource à accéder.

HTTP/1.1 : Version du protocole à utiliser dans la communication.

Request Headers :

La partie des headers n'applique pas d'obligations spécifiques en genre et en nombre. Il y a néanmoins deux types d'headers. Le header HTTP et les autres. Un header HTTP est un header connu et ayant une signification spécifique dans les normes du protocole. Un header non reconnu est simplement ignoré car il peut être utilisé dans le cadre d'échanges entre applications spécifiques.

Request message body ou payload :

La présence d'un body est signalée par la présence du header Content-Length ou Transfer-Encoding. Sa présence dépend aussi de la méthode de la requête. Cette troisième et dernière partie contient des données qui peuvent par exemple être des paramètres ou des identifiants dans la requête d'une ressource en particulier. Dans l'exemple ci-dessus, on peut

voir que la requête contient l'identifiant d'un livre et son auteur.

Quelles sont les types de requêtes à prendre en compte dans le cadre de ce projet ?

Toutes les requêtes HTTP 1.1 (refs. RFC en 1ère page). Elles sont spécifiées plus loin dans ce document.

Partie serveur

Comment un programme attend une requête HTTP ?

Exemple vidéo : [ICI](#)

Intro / Ressources utiles

Voici un cours d'initiation à cette nouvelle façon de programmer nommé asynchrone : [ICI](#)

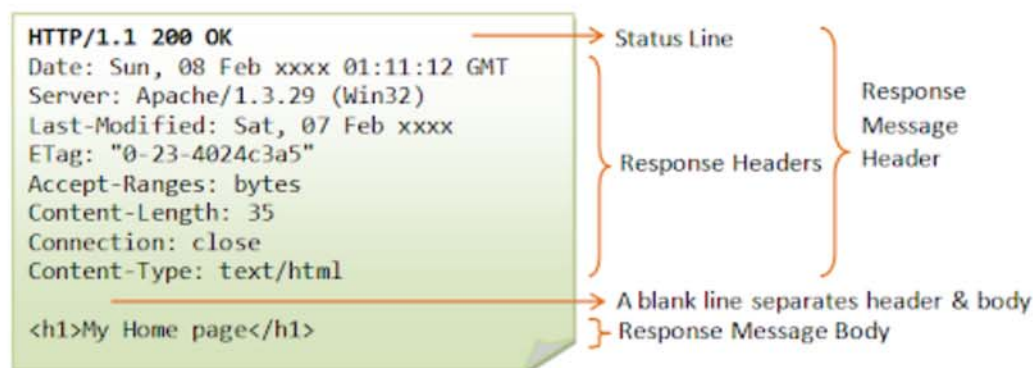
Un comparatif détaillé sur cette méthode dans le cas d'Nginx et Apache : [ICI](#)

Approches techniques dans le cadre de notre projet

Nous allons utiliser la fonction select pour gérer les appels comme des événements pour ne pas bloquer le programme qui attend que l'appel en question soit fini.

De quoi est composée une réponse HTTP ?

prenons cette illustration :



Status Line :

La status line comporte, dans l'ordre, les éléments suivants :

- **HTTP/1.1** : HTTP version : Représente la version HTTP utilisée dans la communication.
- **200** : Status code : C'est un élément de trois chiffres décrivant le résultat de la requête menant à cette réponse.
- **OK** : reason-phrase : Ne délivre aucune information supplémentaire par rapport au status code. Sert juste de description textuelle au status code.

Responses Headers :

Similaire aux headers de requête, malgré que certains headers ne sont disponibles uniquement pour l'un ou l'autre. Vous trouverez plus loin la liste des headers avec leurs catégories. (General | Requête | Réponse)

liste des headers avec leurs catégories. (General | requête | réponse)

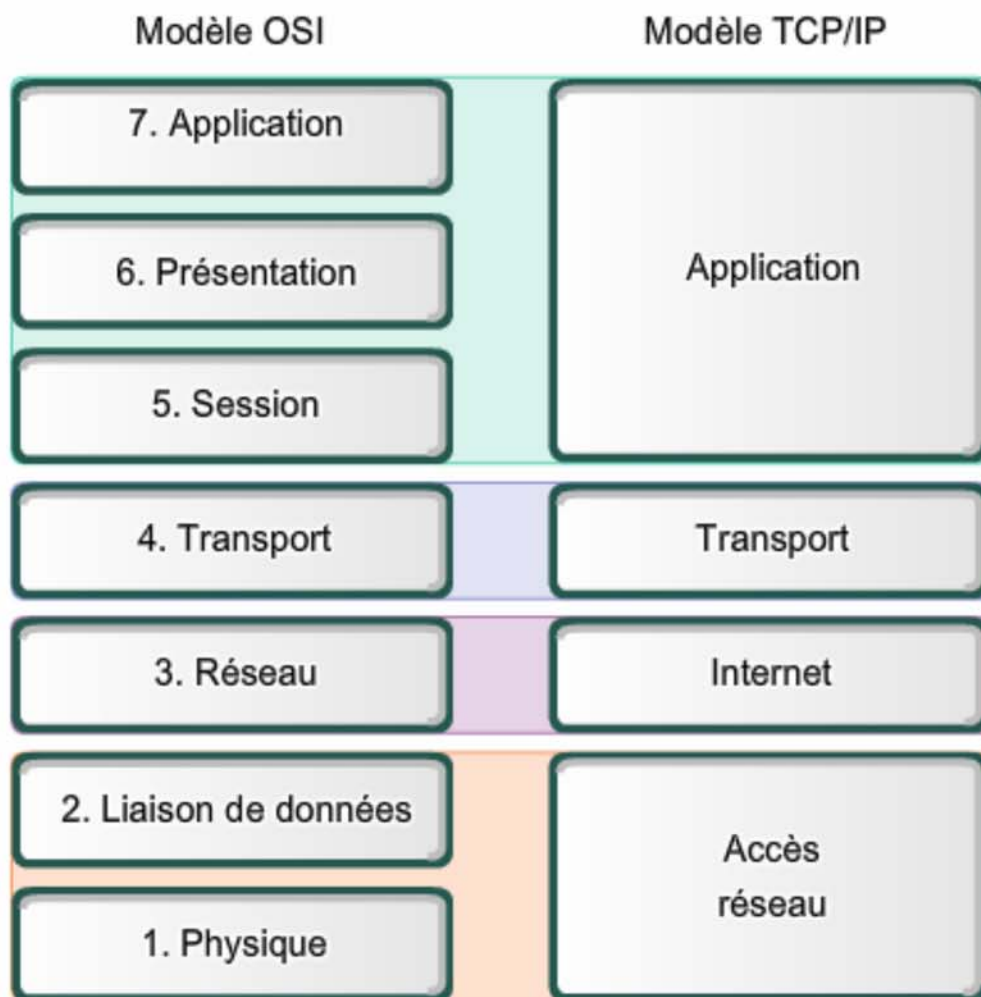
Message Body :

Similaire à une requête.

Relation client-serveur

TCP / UDP, un lien avec ce projet ou transparent ?

En reprenant ce bon vieux schéma TCP/IP :



Pour rappel TCP et UDP sont des protocoles de transport de données. La couche transport est complètement gérée par la fonction `socket()`, nous choisissons uniquement le protocole que socket va utiliser à la création de celui-ci (3ème argument de la [fonction](#)).

Comment assurer la transmission client-server ?

La gestion d'erreur des différents code de retour des appels systèmes devrait suffir à assurer la connexion et le bon comportement en cas de problèmes de communication.

Negotiation client-server

Méthode de requêtes

Source : <https://tools.ietf.org/html/rfc7231#section-4>


On peut classer certaines méthodes par catégories :

- Il y a les **Safe Methods**, ce sont les méthodes considérées comme


sûres car elle ne demande pas de modifications à l'intérieur du serveur. On peut aussi dire qu'elles sont **read-only**. **GET**, **HEAD**, **OPTIONS** et **TRACE** sont des **Safe Methods**.

- Les **Idempotent Methods** (def. [idempotent](#)) inclus les **Safe Methods** mais aussi les méthodes qui modifient le serveur en ayant le même résultat si elles sont appelées plusieurs fois de suite.
- Enfin, les **Cacheable Methods** sont les méthodes qui peuvent être gardées en mémoire pour être renvoyées si jamais ces mêmes requêtes sont demandées plusieurs fois de suite.

Liste des methodes

The HTTP Request Headers List
Instance manipulations that are acceptable in the response. Defined in RFC 3229 Accept:
 <https://flaviocopes.com/http-request-headers/>

The HTTP Request Headers List

 flaviocopes.com

Toutes ces méthodes peuvent renvoyer 405 Not Allowed.

- Request Headers :
 - Liste des headers qui peuvent être pris en compte par la méthode.
- Response Headers :
 - Liste des headers spécifique à la méthode pouvant être répondu.

 Default view

Methodes

Aa Noms	 Liens RFC	 Description	
GET	https://tools.ietf.org/html/rfc7231#section-4.3.1	Envoie une ressource demandé au client	
HEAD	https://tools.ietf.org/html/rfc7231#section-4.3.2	Même réponse que GET sans body	
POST	https://tools.ietf.org/html/rfc7231#section-4.3.3	Envoie de données au serveur	Cc Cc
PUT	https://tools.ietf.org/html/rfc7231#section-4.3.4	Création ou ecrasement d'un fichier existant par le body au chemin demandé	Cc Cc
DELETE	https://tools.ietf.org/html/rfc7231#section-4.3.5	Supprime la ressource demandé dans le serveur	
OPTIONS	https://tools.ietf.org/html/rfc7231#section-4.3.7	Envoie au client les methodes autorisées pour la ressource demandé	
TRACE	https://tools.ietf.org/html/rfc7231#section-4.3.8	Renvoie la requête demandé par l'utilisateur	

- GET (section-4.3.1)
Cette requête est celle que nous faisons le plus. C'est la requête qui va être faite pour demander un contenu à un serveur web. Par exemple, quand vous entrez <http://google.com/> dans votre navigateur, celui-ci fait une requête `GET /` vers l'ip de google.com pour demander la page web principale (`" / "` pour la racine, root). Ainsi les serveurs de google vous répondent avec un code `200 OK` avec en body la page web demandé. Enfin, votre navigateur affiche en HTML la page demandé et PAF vous avez Google.
- HEAD (section-4.3.2)
Exactement la même fonction que GET. Sauf que HEAD n'envoie pas le contenu (body) de la requête demandé. Les headers liés au body (section-3.3) sont donc soit pas présents soit ignorés. Ce type de requête a deux utilités majeures. Premièrement, elle sert à tester la validité de l'accès à une ressource. Cela permet de tester si une requête GET est possible en réduisant la charge appliquée au serveur si celle-ci est valide. Aussi, si l'on a besoin uniquement de certaines métadonnées présentes dans les headers, la méthode HEAD s'y prête mieux. Par exemple, si on veut avoir un fichier volumineux en particulier tout le temps à jour. On peut faire une requête HEAD pour comparer la Content-Length ou la dernière date de modification pour re-télécharger ou non ce fichier.
- POST (section-4.3.3)
Ce type de requête permet d'envoyer des données au serveur HTTP. Ce genre de requête n'a pas pour but de modifier le serveur mais ce sont plutôt des données en liaison avec un back-end ou une page HTML. Dans ces requêtes, on envoie généralement des formulaires HTML.
- PUT (section-4.3.4)
Les requêtes PUT servent à modifier les fichiers locaux du serveur. En corps de requête, une ressource sera envoyée, à écrire au chemin fournis dans la requête. Cela peut être une image ou une page HTML par exemple. En cas de succès, la réponse à un PUT est soit un 201 Created si la ressource demandée a été créée ou 200 OK ou 204 si la ressource a été modifiée. En cas d'erreur, renvoie un code d'erreur.
- DELETE (section-4.3.5)
Cette méthode est un équivalent de la commande UNIX `rm`. Elle permet de demander la suppression d'une ressource au chemin indiqué.
- CONNECT (section-4.3.6)
Cette méthode sert à créer un tunnel TCP entre deux machines. Dans la requête, le client spécifie une machine de destination à laquelle il veut faire le pont. Ensuite toutes les données que le client enverra seront directement retransmises au destinataire via le serveur HTTP. Par exemple, on peut avoir un serveur HTTP de proxy qui va donner accès à plusieurs machines dans un réseau privé. Celui-ci va s'occuper de faire l'authentification puis de connecter le client avec la bonne machine

machine.

- OPTIONS (section-4.3.7)

Cette méthode sert à demander au serveur distant des informations concernant la façon de communiquer. Dans le cadre de notre projet. Cette méthode se concentre surtout sur le Header Allow. A la suite d'une requête OPTIONS Webserv va répondre avec le Header Allow en indiquant quelles méthodes un client peut faire sur celui-ci.

- TRACE

Toujours la même réponse. Nous nous basons sur Nginx et son comportement est de répondre 405 (not allowed) dans tout les cas. Ceci n'est pas configurable, depuis une certaine version c'est *hardcodé*.

Types de headers

Source : <https://tools.ietf.org/html/rfc7231#section-5>

☰ Default view

Types de headers

Aa Type du header	⌵ Requête ou réponse	⌵ Osef ?
Authorization	requête	oui
Accept-Charset	requête	oui
Accept-Language	requête	oui
Content-Type	requête	non
Content-Length	requête	non
Date	requête	non
Host	requête	non
Referer	requête	oui
Transfer-Encoding	requête	non

User-Agent	requête	oui
Allow	réponse	non
Content-Language	réponse	oui
Content-Length	réponse	non
Content-Location	réponse	oui
Content-Type	réponse	oui
Date	réponse	non
Last-Modified	réponse	non
Location	réponse	non
Retry-After	réponse	oui
Server	réponse	non
Transfer-Encoding	réponse	oui
WWW-Authenticate	réponse	oui

Types de réponses

Ce qui est demandé ici est une très courte description des codes d'erreurs.

Source : <https://tools.ietf.org/html/rfc7231#section-6>

Description : <https://www.internetvista.com/en/result-list-internet-monitoring.htm#http>

Show All

1xx (Informational) link :		
#	code	Aa reason-phrase
	100	Continue
	101	Switching Protols

Show All

2xx (Successful) link :

#	code	Aa reason-phrase	≡ de:
	200	OK	
	201	Created	
	202	Accepted	
	203	Non-Authoritative Information	
	204	No Content	
	205	Reset Content	
	206	Partial Content	

≡ Show All

3xx (Redirection) link :

#	code	Aa reason-phrase	≡ de:
	301	Moved Permanently	
	301	Moved Permanently	
	302	Found	
	303	See Other	
	304	Not Modified	
	305	Use Proxy	
	307	Temporary Redirect	

≡ Show All

4xx (Client Error) link :

#	code	Aa reason-phrase	≡ de:
	400	Bad Request	
	401	Unauthorized	
	402	Bad Request	
	403	Forbidden	
	404	Not Found	
	405	Method Not Allowed	
	406	Not Acceptable	
	407	Proxy Authentication Required	

407	Proxy Authentication Required	
408	Request Timeout	
409	Conflict	
410	Gone	
411	Length Required	
412	Precondition Failed	
413	Payload Too Large	
414	URI Too Long	
415	Unsupported Media Type	
416	Range Not Satisfiable	
417	Expectation Failed	
426	Upgrade Required	

 Show All

5xx (server error) link :

#	code	Aa reason-phrase	des
	500	Internal Server Error	
	501	Not Implemented	
	502	Bad Gateway	
	503	Service Unavailable	
	504	Gateway Timeout	
	505	HTTP Version Not Supported	

Fichier de configuration

Ressources :

- `listen` : [link](#)
- `server_name` : [link](#)
- `global` : [link](#)
- `autoindex` : [link](#)

Visiblement, nous n'avons que le contexte de `server {}` à implémenter dans le serveur.

Chemin de test de comparaison avec nginx : `/etc/nginx/conf.d/default.conf`

Les mots clés à implémenter :

- `listen`

- server_name
- error_page
- client_max_body_size
- root
- index
- autoindex
- location (location n'étant pas vraiment un mot clé de cl)
 - autoindex
 - index
 - fastcgi_param
 - fastcgi_pass

 Show All

Types et Mots clés du fichier de conf.

Aa mot-clé

 value type

listen	<code>std::string</code>
server_name	<code>std::list<std::string></code>
error_page	<code>std::map<int, std::string></code>
client_max_body_size	<code>int</code> (en octet)
root	<code>std::string</code>
autoindex	<code>std::string</code> (" on " / " off ")
index	<code>std::list<std::string></code>
fastcgi_param	<code>std::map<std::string, std::string></code>

CGI

RFC 3875 - The Common Gateway Interface (CGI) Version 1.1

Network Working Group D. Robinson Request for Comments: 3875 K. Coar Category:
Informational The Apache Software Foundation October 2004 The Common Gateway Interface

 <https://tools.ietf.org/html/rfc3875#section-4>

C'est quoi CGI ?

CGI est une interface permettant de déléguer la distribution de la ressource demandée par le client. C'est un processus enfant que Nginx va créer en lançant ce binaire CGI. Un exemple de CGI est php-fpm qui permet de distribuer des pages HTML générées par un moteur php. PHP étant un langage que le serveur web n'interprète pas. Cette couche d'abstraction permet donc d'appeler des moteurs de génération de page de n'importe quel langage tant qu'il y a cette interface CGI à exécuter.

Mais, on doit lancer une instance du binaire CGI pour chaque requête au

nginx va donc lancer une instance du binaire CGI, envoyer la requete au CGI demandé et écouter la réponse du logiciel en question puis la délivrer au client.

C'est quoi la différence entre CGI et FastCGI ?

Le fastCGI apporte majoritairement le dynamisme des pages. Mais la différence est flou entre les deux.

Comment un CGI se matérialise-t-il en code ?

C'est un fichier binaire qui va être exécuté en processus fils. Ce processus fils aura pour entrée et sortie `STDIN` et `STDOUT` respectivement. Ce processus prendra en variable d'environnement les paramètres que vous voulez lui affecter.

C'est quoi les Méta-variables et comment les gérer ?


Ce sont les paramètres avec lesquels vous allez affecter votre binaire CGI.

Doit-on envoyer les headers en entrée du CGI ou juste le body ?

Les headers sont transformés en paramètres et le body n'est pas lu. Le `client_fd` est branché sur l'entrée standard du fils CGI.


Common Gateway Interface - 1.1 *Draft 01*


INTERNET-DRAFT Ken A L Coar draft-coar-cgi-v11-01.{html,txt} IBM Corporation D.R.T. Robinson 8 December, 1998 This document is an Internet-Draft. Internet-Drafts are working

 <http://www.wijata.com/cgi/cgispec.html#4.0>

Variables d'environnement CGI - Wikipédia


Les variables d'environnement CGI sont des variables transmises à un programme CGI, par le serveur Web

 https://fr.wikipedia.org/wiki/Variables_d%27envi...



Environment variables set by HTTP Server

The IBM HTTP Server for i supports the standard environment variables in addition to environment variables that are unique to the IBM i server.

 <https://www.ibm.com/docs/en/i/7.3?topic=information-environment-variables>

Show All

Paramètres CGI

Aa Param	Provenance de donnée	No
Légende requête	requête	Data p ma rec
Légende server	server	Data p server
Légende config	config	Data p

		<i>fastcgi</i>
Légende statique	statique	Ce ser- valeur
AUTH_TYPE	requête	Type s Autho
CONTENT_LENGTH	requête	Heade requêt
CONTENT_TYPE	requête	Heade requêt
GATEWAY_INTERFACE	statique	CGI/1.
PATH_INFO	config	Chemi
PATH_TRANSLATED	server requête	Root d chemir index s
QUERY_STRING	requête	Toute l d’inter
REMOTE_ADDR	requête	Adress accept
REMOTE_IDENT	config	Unique fastcgi
REMOTE_USER	config	Unique fastcgi
REQUEST_METHOD	requête	GET / I
REQUEST_URI	requête	Path d
SCRIPT_NAME		:? AUC
SERVER_NAME	server	server_ conf o
SERVER_PORT	server	Port su été rec
SERVER_PROTOCOL	statique	HTTP/
SERVER_SOFTWARE	statique	drunks
HTTP_*	requête	C'est v header

Comment lire la réponse d'un CGI ?

Compiler et executer CGI

Ubuntu (20.x testé)


```
./configure --without-iconv,
```

```
sudo apt install libsqlite3-dev libxml2-dev ./configure
--without-iconv
```

Mac (mojave testé)

```
./configure --without-iconv
```

```
make build-binaries
```

Appeler le binaire

A la suite de cette compilation, un binaire est créé dans :

```
./sapi/cgi
```

Ce binaire peut être appelé dans le code grâce à `execve`.

Nous l'exécutons donc dans un processus fils en redirigeant préalablement STDIN et STDOUT pour stocker ces données.

I/O

Organisation de groupe

Communication & Structure

Communication

Nous communiquerons via Discord qui permet facilement de séparer les discussions par sujet, de pouvoir partager son écran et d'épingler des messages plus importants que d'autres.

Structure

Nous réaliserons la structure logique du code sur le logiciel Miro qui est un logiciel de mindmap collaboratif à l'instar de Google Docs. Voici une illustration du rendu à partir d'un autre projet :

Code

Github

Nous utilisons Github comme dépôt et support de travail. Nous allons travailler chacun sur une branche personnelle. Nous mergerons au fur et à mesure du projet l'avancement de nos parties de code. La branche principale, nommée `main`, devra respecter les conditions suivantes :

principale, nommée `main`, devra respecter les conditions suivantes .

- Compiler avec `-Wall -Wextra -Werror -std=c++98` .
- Être en accord avec le format de `cpplint`.
- Être testé avec `-fsanitize=address -g3` .
- Ne pas avoir de fuites de mémoire.
- Contenir uniquement du code utile et de commentaire censé. (CàD pas de `log.txt` ou `.o` en trop)

Noms de fichiers

Extensions :

Les fichiers source ne comprenant pas de préprocesseurs seront suffixé de `.cpp` . Les fichiers ne contenant uniquement du préprocesseur seront suffixé de `.hpp` .

Noms :

Tout en minuscule et séparé par des underscores.

Voici un exemple :



On a la catégorie de code " `redirectors` ", donc ce sur quoi le code source porte puis un sous groupe divisant pour organiser le code et avoir des code sources organisés et pas trop long.

Compilation utilisée

Ce projet sera compilé et compilable sur linux et macOS par le compilateur `clang++` accompagné des flags `-Wall -Wextra -Werror -std=c++98` . Nous ferons des tests plus précis avec le flags `-fsanitize=address` pour les heap-overflow, stack-overflow et autre. Nous inspecterons les fuites de mémoire à l'aide de Valgrind (linux) et de Leaks (macOS).

Formatage du code

Notes diverses des conventions de nommages ou de structure de codes.

Nombre maximal de caractères par ligne : **80**.

Nous savons que ce n'est pas forcément le meilleurs choix mais dans le cas où cette condition est demandée dans un projet futur, il est important de savoir faire du bon code dans cette restriction et donc de se creuser la tête pour trouver de bonnes syntaxes.

Préfixage des types de données personnalisées :

`typedef` → `t_` (exemple : `t_location`)

`struct` → `s_` (exemple : `s_location`)

`class` → `c_` (exemple : `c_location`)

Pointeur sur fonction `f_` (exemple `f_parser`)

Utilisation des attributs privés de nos classes :

Prefixage par `_` (exemple → au lieu de `this->name` → `_name`)

Les fonctions doivent (quand c'est possible) se préfixer par une action.

Exemples :

- Une fonction qui va chercher une donnée peut se préfixer par `get_` .
(`get_name()`)
- Une fonction qui retourne un booléen peut se préfixer par `is_` .
(`is_odd()`)

- Liste