WEB

[**Client**](#_hzvhn1n25oyr) **3**

[**HTTP**](#_b923kafawzzj) **4**

[Requêtes](#_keug53itv3jx) 4

[Réponses](#_bao4hmtmg8ak) 5

[**Routing**](#_yoxq67pyy5dz) **7**

[**DNS**](#_w3dmf5klubp5) **8**

[**Serveur**](#_5qyf88tr8mno) **10**

[**Applicatif**](#_26kbirzfv6so) **13**

[**Conclusion**](#_pf9rj0scfmf1) **14**

# 

# Client

Le client est le programme qui va exécuter les requêtes les requêtes et traiter ou non le résultat.

Le client peut être un navigateur (chrome, internet explorer, etc..) mais peut tout aussi bien être un simple programme exécutant des requêtes web sans interpréter le résultat, comme cURL par exemple.  
  
Notre programme qui va par exemple exécuter des requêtes HTTP pour se connecter à une API externe va également être un client.

Un client peut donc être de plusieurs formes, interpréter ou non la réponse (html, css, js) et envoyer des informations différentes au serveurs. Dans le cas d’un serveur web on va parler de requêtes HTTP.

# 

# HTTP

## Requêtes

Les requêtes HTTP sont le type de requête qui va transiter entre un navigateur et un serveur.

Elles sont caractérisées par les informations suivantes:

* Un **type** : *GET / POST / PUT / PATCH / DELETE / etc…*
* Le **protocole** utilisé: *HTTP HTTPS* , si HTTPS des informations de sécurité (certificats, version du protocole utilisé, etc… sont envoyés avec la requête). HTTP utilise habituellement le port 80, HTTPS le port 443 mais on peut très bien adresser un autre port en le specifiant dans l’url : http://perdu.com:8080/
* Un **Path** : */ , /page1.php, /image1.png*
* Un **Host** : *perdu.com, lemonde.fr*
* Différents **headers** non obligatoires, permettant de donner des informations supplémentaires au serveurs (attention certains cerveurs peuvent exiger que certains headers soient définis) :  
  + **Upgrade-Insecure-Requests:** *1 ou 0*, permet de forcer le passage en HTTPS
  + **User-Agent** : *Mozilla/5.0 (X11; Linux x86\_64)* qui indique au serveur quel type de client envoi la requête
  + **Accept**: *text/html, application/json* : Quesque le client accepte comme format de réponse (html, ou json, ou image, etc…)
  + **Accept-Encoding**: *gzip, deflate* : Est ce que le client accepte la compression des requêtes
  + **Accept-Language** : *fr-FR, en-US,* langage accepté par le client
  + **Cookie**: *cookie1=abc;cookie2=sdf;* cookies du clients stockant les informations de la session client dans le client
  + **Content-Type**: *application/json,* … Si requête post, type de données envoyées (xml, form html, json)
  + **Content-Length:** Longueur des données envoyées
  + N’importe quel header custom
* **Des données** si requête POST envoyée

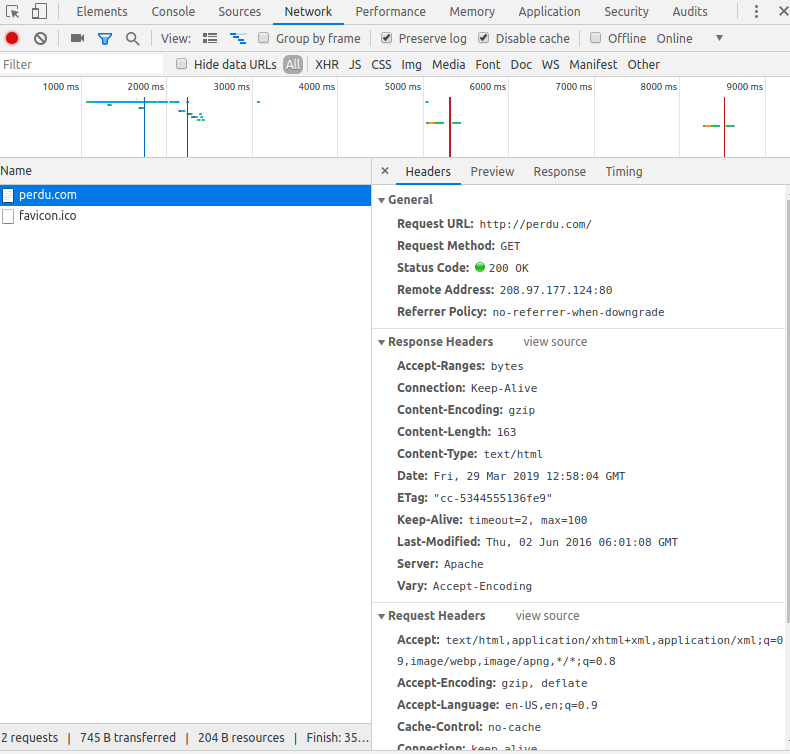
Une fois la requête récupérée par le serveur (nginx, apache, tomcat), celui ci la transmet à l’applicatif (php, nodejs, J2E), celui ci construit une réponse et la transmet au serveur pour la renvoyer vers le client.

## Réponses

La réponse est caractérisée par les informations suivantes :

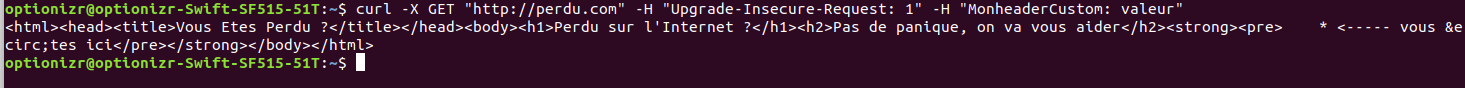
* Un **code** de réponse :
  + Les plus courants :  
    - **200**: Ok
    - **204**: Ok mais pas de contenu
    - **301**: Déplacé, redirection à l’adresse contenu dans le header Location
    - **302**: Déplacé, redirection à l’adresse contenu dans le header Location
    - **404**: Not found
    - **500**: Erreur applicatif
    - **502**: Erreur serveur
  + Mais tout les codes existes, en général :  
    - Les **200-299** représentent un succès
    - Les **300-399** représentent une interdiction
    - Les **400-499** représentent une erreur de requête
    - Les **500-599** représentent une erreur serveur
* Des **headers** :
  + **Set-Cookie**: *cookie1=valeur2;* Le client doit définir ces valeurs dans la session client
  + **Server**: *Apache, nginx*, Le type de serveurs qui a traité la requête
  + **Content-Encoding**: *gzip* , Définis si la réponse est compressé
  + **Content-length**: longueur de la réponse
  + N’importe quel header custom
* La **réponse**

Sur les navigateurs webs ces informations sont visible via les outils de développement, onglet network. Exemple sous chrome (touche F12) :



En faisant clic droit sur la requête on peut même la copier en tant que requête curl, ce qui nous permet par exemple d’envoyer la même requête avec le terminal :

curl -X GET “<http://perdu.com>” -H “Upgrade-Insecure-Request: 1” -H “MonheaderCustom: valeur”



# Routing

Au moment d’envoyer la requête vers par exemple : <http://perdu.com/> le client doit d’abord transformer le nom de domaine : “perdu.com” en IP accessible : 213.186.33.87

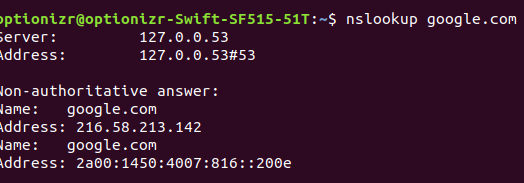
Il va faire ce que l’on appel une résolution de nom

Il va d’abord lire le fichier */etc/hosts* afin de vérifier si le domaine est connu au niveau local. Ce fichier se présente sous la forme suivante :  
  
*127.0.0.1 localhost  
54.6.7.3 mondomain1*  
En mettant par exemple : *127.0.0.1 facebook.com* je vais rediriger toutes les requêtes de mon navigateur vers 127.0.0.1 (localhost)

Sinon il va interroger des annuaires appelés DNS (Domain name serveur) afin de faire la résolutions de noms. Il en existe beaucoup et tous les clients ne vont pas utiliser les mêmes (chrome va utiliser par exemple le dns google 8.8.8.8, cURL ou php vont en utiliser d'autres, etc…).

On peut tester la résolution de nom d’un domaine avec l’utilitaire **nslookup**, utiliser principalement pour vérifier après une manipulation que mon domaine est bien branché à la bonne ip.

nslookup google.com



# DNS

Le DNS est donc un annuaire contenant des tables liants nom de domaines et adresses ip. Plus généralement il permettent d’enregistrer des informations accessible de l'extérieur sur un nom de domaine. Il faut prouver posséder ce nom de domaine pour pouvoir enregistrer le domaine sur un DNS.

Des milliers de DNS existent (Amazon, Google, Apple, OVH, etc….). Pour plus d'efficacité tout les DNS ont des liens entre eux et les clients interrogent systématiquement plusieurs DNS. De plus un DNS est en fait un ensemble de serveurs, le client interrogeant le serveur le plus près de sa position.  
  
Ainsi lorsqu'une information est enregistrée / modifiées dans un DNS elle peut mettre un certain temps à se protéger, duplication de l’information sur tous les serveurs du DNS puis duplication de l’information sur les autres DNS, etc…. Cela peut prendre de 1 minute à 3 jours avant qu’une information fasse le tour des DNS de la planète.

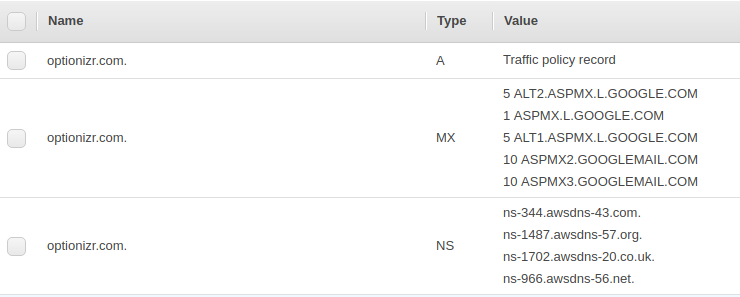


Après une mise à jour DNS certains clients peuvent donc toujours accéder aux anciennes tables de routage pendant quelques temps en fonction des différents caches.

On peut enregistrer différentes informations (records) sur un DNS :

* **A record**: Association du nom avec une IP :  
  + *optionizr.com A 137.74.196.110*
  + *demo.optionizr.com A 137.74.196.109*
  + *\*.optionizr.com A 15.74.196.108*
* **MX record :** Autorisation pour un serveur externe d’envoyer des mails au nom du domaine  
  + *Optionizr.com MX ASPMX.L.GOOGLE.COM* : autorise les serveurs de google à envoyer des mails au nom d’optionizr
* **TXT record:** N’importe quelle information/texte
* **CNAME record:** Redirection  
  + google.optionizr.com CNAME google.com

Chez optionizr nous sommes par exemple enregistrés chez AWS (amazon), voici à quoi ressemble notre table DNS :



Une fois le nom résolu, le client transmet la requête au serveur

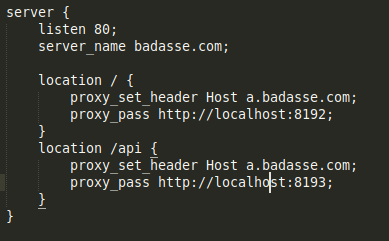
# Serveur

Le serveur un un simple applicatif écoutant sur un ou plusieurs ports d’un ordinateur et permettant de recevoir et d’envoyer des requêtes.  
  
Il se charge de l’encryptage / décryptage des requêtes, compression des réponses, ajout d’informations, vérifications des requêtes, etc...

Les plus connus sont Apache (présent dans LAMP ou WAMP par exemple), Nginx, Tomcat, IIS, … Optionizr utilise le serveur NGINX et c’est donc lui que nous détaillerons plus tard.

Nginx fonctionne via de simple fichier de configuration, on lui indique simplement quel port écouté, quel nom matcher et la règle de redirection vers l'application.  
Nginx charge tous les fichiers présent au path suivant : */etc/nginx/sites-enabled/monfichier.conf*

Les fichiers se présentent sous la forme suivantes :



Des objets serveurs (autant que l’on veut), comprenant les informations suivantes :

* **listen**: port à écouter
* **server\_name**: le nom de domaine à matcher (doit être définis également au niveau de /etc/hosts !)
* **location**: / le path à matcher ( / correspond à tout ce qui ne match pas le reste)
* **proxy\_set\_header** : Ajout d’un header
* **proxy\_pass**: redirection en proxy vers autre adresse - IP

On peut ne pas rediriger systématiquement sur la même adresse en utilisant un upstream, par exemple si on a deux programmes node qui tournent sur le port 81 et sur le port 82 on peut dispatcher les requêtes sur les 2 de cette façon :



On peut tester une config avec la commande suivante, qui nous indique s’il y a des erreurs :

sudo nginx -t

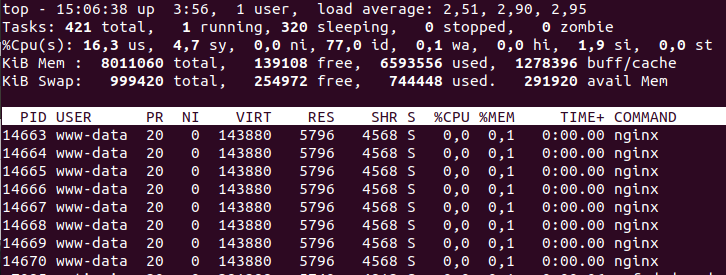


Et recharger les configs après une modification avec la commande suivante :

sudo nginx -s reload



Pour plus d'efficacité et pouvoir répondre à un maximum de requêtes un serveur est en général forké (plusieurs processus du serveurs tournent sur la machine) permettant ainsi de traiter plusieurs requêtes en parallèles. Nginx se forke par exemple en fonction du nombre de core de la machine (4 cores = 4 instances du programme en mémoire).



Un serveur peut être un serveur terminal (renvoyant directement sur l’application), ou être un serveur intermédiaire effectuant simplement une redirection vers un autre serveur. Un load balancer est par exemple un serveur intermédiaire permettant de dispatcher la charge sur plusieurs serveurs (par l'intermédiaire des upstreams).

Si par exemple 2 serveurs pour mon API, je peux dispatcher les requêtes à charge égale sur les deux serveurs api via un load balancer



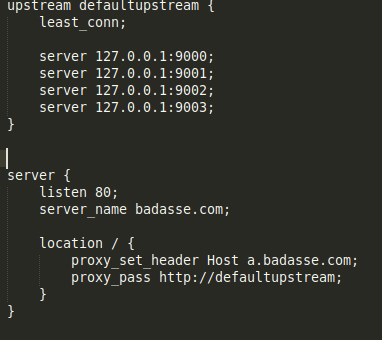
Enfin, une fois le serveur terminal ayant récupéré la requête, il la transmet à l'application. Dans notre cas, notre applicatif est une application node js avec le framework express.

# Applicatif

L'application est la pour traiter la requête et créer une réponse. Dans notre cas une application nodeJS / express.

Express est un framework nodejs permettant de créer un serveurs applicatif. Il permet d’écouter sur un port et de traiter des requêtes, il contient également un serveur nginx et écoute sur un port particulier, en général on utilise 3000 ou 9000, mais on peut le mettre sur n’importe quel port.

En général on fait tourner une instance de l’application par core de la machine afin de maximiser le nombre de requête que l’on peut traiter à la seconde. Dans ce cas la machine contient une configuration nginx permettant de dispatcher les requêtes sur les différents processus.



En général plusieurs serveurs applicatifs coexistent, par exemple pour optionizr :

* Serveur de production 1
* Serveur de production 2
* Serveur de test 1
* Serveur de dev 1
* Serveur d’upload
* ….

# Conclusion

Schéma général du web, dans le contexte optionizr

