Apprendre à optimiser les performances de Nginx

Par Quentin Busuttil

Date de publication : 24 juillet 2016

Dernière mise à jour : 3 décembre 2017

Nginx de par sa robustesse, sa structure minimaliste et son fonctionnement événementiel asynchrone est un serveur web plébiscité pour ses performances. Mais ce n'est pas parce qu'on a une Porsche qu'il ne faut pas tenter de la rendre encore plus puissante !

Pour des sites à fort trafic, un serveur bien optimisé signifie deux choses : des requêtes servies plus vite et un besoin en machines (scaling horizontal) inférieur. Alors, pourquoi s'en priver ?

Pour réagir au contenu de ce tutoriel, un espace de dialogue vous est proposé sur le forum. **Commentez**

En complément sur Developpez.com

- Apprendre à optimiser la gestion des ressources système avec ulimit

I - Les workers....3
II - Les buffers....3
II-A - Proxy buffers....4
III - Les timeouts....5
IV - La compression....6
V - Le cache statique....7
VI - L'open file cache....7
VII - La spéciale TLS....7
VIII - Optimisations TCP....8
IX - La thread pool....10
X - Note de la rédaction de Developpez.com....10

I - Les workers

La première chose à explorer est le fichier de configuration général de Nginx : /etc/nginx/nginx.conf. Nous allons ici définir le nombre de workers ainsi que le nombre de connexions pour chacun d'entre eux.

Concentrons-nous d'abord sur worker_processes. Cette directive spécifie le nombre total de workers à créer au démarrage de Nginx. La valeur optimale est d'en avoir un par *cpu core*. Si vous avez un VPS - lesquels ont souvent un vCPU avec un seul vCore - , il arrive souvent que la valeur par défaut soit supérieure au nombre total de cores. Ce n'est pas très grave, néanmoins, les processus supplémentaires vont un peu se tourner les pouces… pas bien utile. Pour définir cette valeur, il nous suffit donc de déterminer le nombre total de cores :

```
# grep -c ^processor /proc/cpuinfo
12
```

Nous avons notre réponse, ici : 12.

Passons maintenant à la directive worker_connections. Elle spécifie combien de connexions simultanées chaque worker est en mesure d'établir. Étant donné qu'une connexion nécessite un *file descriptor* au minimum, une bonne base est d'établir ce nombre en fonction des limites de notre système. Dans le cas de Nginx en reverse proxy, il faut un *file descriptor* pour la connexion client et un autre vers le serveur proxifié, soit deux par connexion.

```
# ulimit -n
1024
```

Par défaut, la limite du nombre de fichiers ouverts est souvent de 1024. Selon la configuration du système, cette limite peut être très largement augmentée. Voyez mon **article sur ulimit** afin de modifier ces configurations.

Vous lirez peut-être dans certains articles qu'il faut définir la directive worker_rlimit_nofile. Cette dernière a justement pour but de configurer le *ulimit*. Il faut impérativement que le serveur démarre en root avant de passer à un utilisateur non privilégié, sans quoi il ne pourra augmenter cette limite. Par ailleurs, s'il y a d'autres logiciels qui utilisent le même utilisateur - nginx sous Debian - il est bon de leur en laisser quelques-uns. Vous devrez dans ce cas avoir une valeur pour worker_connections légèrement inférieure à worker_rlimit_nofile.

Pour ma part, sauf cas particulier, je trouve plus fiable de paramétrer le ulimit en utilisant la configuration du système et de se contenter d'attribuer la même valeur au *ulimit* et à worker_connections (si Nginx est seul à tourner avec ce user).

En faisant worker_processes * worker_connections, on peut déjà se faire une idée du nombre de connexions simultanées que Nginx va pouvoir encaisser. Bien entendu, d'autres paramètres entrent en ligne de compte.

Par ailleurs, il ne s'agit pas non plus de s'obstiner à atteindre le plus grand nombre possible de connexions. Mieux vaut en effet servir 1000 connexions simultanées en 1 s et passer aux 1000 suivantes, que d'en servir 2000 en 4 s.

En dernier lieu, on permet aux workers d'accepter plusieurs nouvelles connexions de manière simultanée en activant **multi_accept**. Cela peut être d'une grande utilité lors de pics de trafic.

```
multi_accept on;
```

II - Les buffers

Les buffers permettent à Nginx de travailler en RAM plutôt que sur le disque. Je ne vais pas vous faire un dessin, vous savez bien que les accès disque sont infiniment plus lents que le travail en RAM. On va donc configurer les buffers pour que notre serveur puisse travailler en mémoire autant que faire se peut.

Il y a quatre types de buffers :

client_body_buffer_size

Le buffer qui récupère les données clients (typiquement les données POST) ;

client_header_buffer_size

Celui-ci s'occupe également des données du client, mais concerne l'en-tête. Généralement, 1 k suffit ici amplement (la valeur par défaut). Qui plus est, dans le cas où cette limite est dépassée, alors c'est la directive large_client_header_buffers qui s'applique ;

client_max_body_size

La taille maximum des requêtes envoyées par le client. Si vous autorisez des uploads de fichiers, il s'agit d'y penser ici. Si cette limite est dépassée, Nginx retourne une erreur 413 ;

large_client_header_buffers

Taille et nombre maximum que peuvent atteindre les buffers pour les en-têtes. Au-delà, une erreur est retournée.

Les réglages des buffers sont plus délicats et subjectifs que ceux des workers. Cela dépend énormément de votre application. client_body_buffer_size définit la quantité maximale des données GET ou POST qui seront stockées en RAM. La question à se poser est donc la suivante : quelle proportion de vos requêtes sont supérieures à une taille donnée ? Ces requêtes nécessitent-elles d'être traitées très rapidement ? Par exemple, pour l'upload de gros fichiers, il est tout à fait tolérable d'écrire les fichiers sur le disque. En effet, le temps supplémentaire liée à l'écriture sur le disque est dérisoire par rapport au temps d'upload.

En revanche, il faut bien avoir à l'esprit que plus cette limite est élevée, plus votre serveur prête le flanc aux attaques DoS, puisqu'il allouera potentiellement plus de mémoire aux requêtes et arrivera donc potentiellement plus rapidement à saturation en RAM.

La logique prévalente pour les buffers est souvent d'essayer et d'aviser ! Dans une application permettant l'upload de fichiers, je laisserai client_body_buffer_size à sa valeur par défaut de 16k (sur processeurs 64 bits), idem pour client_header_buffer_size qui est de 1k.

client_max_body_size aura ici une valeur élevée puisque je veux autoriser l'upload de fichiers (disons 20m, là encore tout dépend du type de fichiers !). Enfin, je serai plus conservateur que les défauts (4 8k) pour large_client_header_buffers, et mettrai 2 3k, car mon application possède des headers relativement modestes. En résumé :

```
1. client_body_buffer_size 16K;
2. client_header_buffer_size 1k;
3. client_max_body_size 20m;
4. large_client_header_buffers 2 3k;
```

J'insiste de nouveau sur ce point, tout dépend de votre application. Si votre applicatif soumet de gros articles en POST, il faudra peut-être revoir à la hausse les 16k de client_body_buffer_size. Il n'y a ici pas de règle générale.

II-A - Proxy buffers

Il existe des buffers spécifiquement dédiés aux proxies. Dans le cas où les buffers sont désactivés, Nginx commence l'envoi des données au client aussitôt qu'il les reçoit du backend serveur. Si le client est rapide, tout est pour le mieux. Cependant, si le client est moins véloce, ce fonctionnement oblige à conserver une connexion ouverte entre Nginx - le serveur de proxy - et le backend serveur ; ce qui peut s'avérer dommageable.

L'activation des buffers permet donc à Nginx de d'abord récupérer l'ensemble des données de la requête depuis le backend serveur, de libérer ce dernier, puis de servir les données au client.

proxy_buffering

Contrôle l'activation du buffer pour le proxy (activé par défaut).

proxy_buffer_size

Définit la taille du buffer pour les en-têtes de la réponse. 8k par défaut sur systèmes 64 bits. On peut ici laisser la valeur par défaut, car les en-têtes dépassent rarement 8k.

proxy_buffers

Détermine la taille et le nombre des buffers pour le corps de la réponse. Une fois n'est pas coutume, la valeur dépendra grandement de votre application. La valeur par défaut (toujours pour les systèmes 64 bits) est de 8 buffers de 8k. Il s'agit de paramètres s'appliquant **par requête**. Ainsi, le réglage par défaut permettra de stocker dans les buffers des réponses jusqu'à 64kb. À vous de voir si votre application retourne des résultats plus importants (sachant qu'ensuite les fichiers sont **écrits sur le disque**).

```
proxy_buffering on;
proxy_buffer_size 1k;
proxy_buffers 12 4k;
```

III - Les timeouts

Question existentielle s'il en est, les timeouts peuvent avoir un impact important à la fois sur la vitesse ressentie par les utilisateurs, mais aussi sur la charge du serveur. Les timeouts se divisent également en plusieurs directives :

client_body_timeout

Ce timeout s'applique au body. Il définit le temps maximum entre deux opérations d'écriture (pas le temps total de transfert donc). Admettons que je veuille transférer de gros fichiers (plusieurs centaines de MB), je pourrais fixer ce timeout à 30 s (défaut 60 s). Si le client n'envoie aucune donnée dans ce laps de temps, le serveur émet une erreur 408.

client_header_timeout

Même logique que la directive précédente, mais le timeout s'applique bien ici à la totalité de la transaction. Néanmoins, les en-têtes étant beaucoup plus légers, je me contenterai pour ma part d'établir le timeout à 10 s (défaut 60 s) pour les headers.

keepalive_timeout

Cette directive permet à la fois de spécifier le *keepalive timeout*, mais également le header *Keep-Alive: timeout=durée*.
Avec des serveurs tels qu'Apache où le serveur conserve un thread par connexion ouverte, de telles connexions impliquent une consommation de mémoire. Cependant, avec des serveurs événementiels comme Nginx, ce coût est relativement faible et il n'est donc pas très impactant en consommation ressources.
Une fois n'est pas coutume, la valeur idéale dépendra de la typologie de votre application. L'effet du keepalive se fait surtout ressentir lorsque de nombreux éléments sont à télécharger puisqu'on gagne *nbr_ressources * temps_connexion_tcp*. C'est évidemment moins flagrant pour une requête unique. Si votre application nécessite de nombreuses requêtes à intervalles réguliers (*polling ajax* par exemple), il pourra être judicieux de conserver un keepalive long. Au contraire, dans le cas d'un site plus statique, comme un blog, si aucune

autre connexion n'est envisagée après le chargement de la page et de ses ressources, on pourra avoir un keepalive assez court.
L'intérêt du keepalive est d'autant plus grand si vous êtes en HTTPS. En effet, à chaque ouverture de connexion, il vous faudra renégocier un **Three-way handshake TLS**, ce qui prend du temps et demande de la puissance au serveur.
Le keepalive est à 75 s par défaut, il n'y a vraiment pas de règle en la matière. On peut estimer que 30 s est une valeur standard, mais on pourra même l'abaisser à 10 s dans notre exemple de blog. HTML5 boilerplate le définit même à 300 dans sa configuration Nginx ! Comme il n'est pas très risqué de laisser la valeur élevée, on peut donc aisément le mettre à plusieurs minutes et ne l'abaisser qu'en cas de problème de performances face à un fort trafic.

send_timeout

C'est en quelque sorte le pendant inverse des body et header timeouts. Ici, il s'agit de définir le temps après lequel le serveur coupe la connexion si le client ne reçoit plus la réponse. Par défaut, 60 s.

keepalive_requests

Cette directive détermine le nombre de requêtes au bout duquel la connexion sera fermée. La valeur par défaut est à 100, ce qui est assez confortable et il n'est souvent pas nécessaire de modifier cette valeur. Néanmoins, si votre application nécessite le chargement de très nombreuses ressources (> 100), il peut être intéressant d'augmenter cette valeur pour qu'elle soit légèrement supérieure au nombre de ressources à charger.

Comme expliqué, étant donné le faible impact de performance sur Nginx, je préfère être généreux au départ et abaisser les timeouts au besoin :

```
client_body_timeout 30;
client_header_timeout 10;
keepalive_timeout 30;
send_timeout 60;
keepalive_requests 100;
```

IV - La compression

Il ne fait aucun doute que la compression permet d'accélérer les transferts sur le réseau. Il est aussi un fait que compresser les ressources demande un peu de puissance processeur. Comme il existe plusieurs niveaux de compression, il s'agit de trouver le juste milieu.

```
gzip on;
gzip_comp_level 5;
gzip_min_length 1000;
gzip_proxied any;
gzip_types
        application/atom+xml
        application/javascript
        application/json
        application/ld+json
         application/manifest+json
         application/rss+xml
         application/vnd.geo+json
         application/vnd.ms-fontobject
         application/x-font-ttf
         application/x-web-app-manifest+json
         application/xhtml+xml
         application/xml
         font/opentype
         image/bmp
         image/svg+xml
         image/x-icon
         text/cache-manifest
```

```
23.         text/css
24.         text/plain
25.         text/vcard
26.         text/vnd.rim.location.xloc
27.         text/vtt
28.         text/x-component
29.         text/x-cross-domain-policy;
```

Voici en général mes réglages. Le niveau de compression 5 est le juste milieu **selon moi** - on pourra toujours le baisser si on a trop de charge CPU. On ne compresse pas les fichiers de taille inférieure à 1 ko (on n'y gagnera pas grand-chose), on compresse sans distinction les éléments qui proviennent d'un proxy et on active la compression pour les fichiers textes, le js, les css, le json, le xml et d'autres un peu moins courants. Vous ne trouverez ici pas les images jpeg et png, car ce sont déjà des fichiers compressés.

Sachez également que la directive **gzip_buffers** peut s'avérer intéressante. Elle définit le nombre et la taille des buffers alloués à la compression. Par défaut, sur les architectures 64 bits, jusqu'à 16 buffers de 8 k sont autorisés.

Tout va ici dépendre de la taille de vos fichiers. Il s'agit, comme **expliqué dans ce post** d'arbitrer entre le nombre de buffers (gérer de nombreux buffers consomme un peu de CPU) ou attribuer plus d'espace aux buffers (utilise plus d'espace mémoire). Si vous n'avez pas assez d'espace dans les buffers pour contenir l'ensemble de la réponse, Nginx attendra qu'une partie de la réponse soit envoyée au client et utilisera ensuite l'espace libéré.

Enfin, sachez aussi qu'il est possible, avec le module **gzip static**, de précompresser des fichiers pour que Nginx puisse les servir sans avoir à les compresser à la volée.

V - Le cache statique

Grand classique du web, permettre aux navigateurs et proxies intermédiaires de placer certaines ressources en cache, c'est éviter au serveur d'avoir à les renvoyer plus tard. Selon la fréquence à laquelle changent vos fichiers, on peut attribuer une période de validité plus ou moins longue au cache. Disons deux mois pour l'exemple :

```
location ~* .(jpg|jpeg|png|gif|ico|css|js)$ {
  expires 60d;
}
```

Nous cachons donc pour deux mois tous les fichiers d'extensions suivantes : jpg, jpeg, png, gif, ico, css, js.

VI - L'open file cache

Ce type de cache permet de conserver les métadonnées en mémoire, et donc de limiter l'I/O. Voici un exemple de configuration :

```
1. open_file_cache max=2000 inactive=5m;
2. open_file_cache_valid 2m;
3. open_file_cache_min_uses 2;
4. open_file_cache_errors on;
```

Cette configuration indique au serveur de mettre en cache 2000 *open file descriptors* et de les fermer si aucune requête les concernant n'est demandée au bout de cinq minutes. La validité des informations en cache est revérifiée après deux minutes et un fichier doit être requêté un minimum de deux fois afin d'être valide pour le cache. Enfin, les fichiers d'erreurs sont ici également valables pour le cache.

VII - La spéciale TLS

Bien que Nginx nomme toutes les variables SSL, vous savez très bien que le protocole est définitivement à bannir au profit de TLS. Petite parenthèse, si la sécurité vous intéresse, sachez qu'au-delà du protocole, c'est aussi la suite de

chiffrement qui assure la sécurité. Vous pouvez tester la sécurité de votre site avec ce **scanner SSL/TLS** et générer vos configurations serveur avec le **générateur de Mozilla**.

Bref, revenons-en à nos moutons. Nous l'avons dit, le three-way handshake prend du temps et est coûteux en ressources. Puisque le client et le serveur se sont déjà entendus, il est possible de dire à Nginx de mettre en cache cette session. Ainsi, lorsqu'il voudra de nouveau établir une connexion, le client n'aura plus qu'à se rappeler aux bons souvenirs de Nginx. Le three-way handshake ne compte plus que deux allers-retours entre client et serveur et Nginx s'épargne quelques calculs cryptographiques. Bien que datant de 2011, ce **post** vous éclairera sur la théorie derrière cette histoire de cache.

Trois paramètres sont à considérer :

ssl_session_cache

Il permet de définir si on active le cache on non, de spécifier le type de cache (`builtin` ou `shared`) ainsi que la taille de ce dernier. Il est à `none` par défaut. La doc Nginx établit que 1 MB permet de stocker 4000 sessions. À vous de juger combien de sessions vous estimez devoir faire face et la durée de rétention que vous leur accordez. N'ayez crainte, au pire des cas, Nginx invalidera les sessions prématurément, il n'y aura pas d'effusion de sang ;

ssl_session_timeout

Précise la durée au bout de laquelle la session est invalidée ;

ssl_buffer_size

Permet de déterminer pour l'envoi des données. **Gros débat** sur la question de la taille de ce dernier, une fois de plus, tout dépend du type de contenu que vous servez.

Et bien entendu quelques valeurs d'exemples :

```
ssl_session_cache shared:SSL:10m;
ssl_session_timeout 24h;
ssl_buffer_size 1400;
```

En plus des caches, il y a une dernière optimisation qu'il est possible de réaliser, elle a pour doux nom l'**OSCP stapling**. Lorsqu'un serveur fournit un certificat, le client en vérifie la validité en interrogeant l'autorité émettrice du certificat. Évidemment, cela demande une requête supplémentaire au client. On peut éviter cela en demandant au serveur de joindre directement l'autorité émettrice et d'ainsi « agrafer » (d'où le *stampling*) la réponse signée et horodatée de l'autorité afin de prouver la validité du certificat.

```
1. ssl_stapling on;
2. ssl_stapling_verify on;
3. resolver 8.8.8.8 8.8.4.4 216.146.35.35 216.146.36.36 valid=60s;
4. resolver_timeout 2s;
```

Il est possible de préciser un resolver, en cas d'absence de la directive `resolver`, le serveur DNS par défaut sera interrogé. C'est d'ailleurs souvent avantageux, car dans le cas d'un serveur, il se trouve en général dans le même datacenter.

VIII - Optimisations TCP

On arrive là dans le domaine de l'optimisation de pointe ! Il est possible de spécifier à Nginx la manière dont quelques options de TCP doivent être gérées. De manière synthétique, ces réglages vont nous permettre de :

- diminuer la latence de 200 ms avant l'envoi des données sur le réseau ;

- d'optimiser la copie des données depuis le *file descriptors* vers le buffer du kernel (OK, c'est compliqué !) ;
- et de n'attribuer un thread Nginx qu'au dernier moment de la connexion (économie de ressources).

Pour comprendre tout cela dans les détails, vous pouvez vous référer à ce **très bon article** qui plonge littéralement dans les spécificités du noyau.

```
sendfile on;
tcp_nodelay on;
tcp_nopush on;
```

Dans le cas de très nombreuses connexions entre le proxy et le serveur backend, on peut faire face à un problème. Le *TCP Maximum Segment Lifetime* définit la durée de vie maximale d'une connexion TCP et impacte directement le TIME_WAIT - temps à attendre avant de fermer une connexion TCP, égale à *2*MSL*.

Là où le bât peut blesser, c'est qu'un grand nombre de connexions en TIME_WAIT peut empêcher d'en ouvrir de nouvelles si on arrive à épuisement du nombre de connexions ouvertes entre deux IP - défini par le couple port/ip, soit le nombre de ports définis dans /proc/sys/net/ipv4/ip_local_port_range, en général ± 30 000. Dans cette configuration, on ne pourra effectuer plus de *30 000/(2*MSL)* requêtes par secondes, soit *300 000/(2*30)*, soit 500 req/sec.

Plusieurs solutions sont possibles :

- activer keepalive pour maintenir les connexions ouvertes ;
- utiliser net.ipv4.tcp_tw_reuse afin de recycler plus rapidement les connexions en TIME_WAIT ;
- diminuer le tcp_fin_timeout.

Le plus efficace et le moins risqué selon moi est d'utiliser keepalive. Cependant, tous les serveurs ne le prennent pas en charge (Haproxy par exemple). Il vous reste donc au choix de diminuer le tcp_fin_timeout ou d'utiliser net.ipv4.tcp_tw_reuse (ou les deux). Je vous invite à lire **cet article de qualité**, lequel traite du problème de connexions en TIME_WAIT et des remèdes potentiels (dont tcp_fin_timeout).

Vous pourrez modifier ces paramètres respectivement dans /proc/sys/net/ipv4/tcp_fin_timeout et /proc/sys/net/ipv4/tcp_tw_reuse.

En dernier lieu, nous allons nous pencher sur quelques optimisations qui s'effectuent au niveau du block server, elles se placeront donc en général dans un VHOST.

Nous l'avons déjà vu plus haut, une connexion TCP s'effectue en trois temps, cela porte le doux nom de **Three-way handshake**. En gros, cela demande trois échanges de trames TCP/IP entre le client et le serveur. Le premier envoie un paquet SYN (demande de connexion [*synchronize*]), le second répond SYN/ACK (en gros : 5/5, Roger ! [synchronize acknowledge]) et enfin, le client confirme que tout est OK avec un ACK.

À partir de là, la connexion est ouverte, et théoriquement, un socket est créé et le processus en écoute sur ce socket est réveillé : attribution de ressources Nginx dans le cas présent. Pour autant, tant qu'aucune vraie requête n'est effectuée, Nginx n'a rien à faire. L'option **TCP_DEFER_ACCEPT** du kernel permet donc de ne réveiller le processus que lors de l'envoi effectif de données. Cette option se traduit dans nginx par deferred.

Dans un autre registre, ce n'est pas tout à fait lié à TCP, mais j'en parle ici quand même, activer le HTTP/2 peut avoir un impact significatif sur les performances. Pour plus de détails sur le protocole, je vous laisse jeter un œil sur **l'article Wikipedia**.

Les deux optimisations dont nous venons de parler se définissent dans le bloc server et au niveau de la directive listen.

```
1. server {
2.     # on active deferred
3.     # pour ipv6 sur le port 80 (http)
4.     listen [::]:80 default_server deferred;
5.
```

```
6.      # pour ipv4 sur le port 80 (http)
7.      listen 80 default_server deferred;
8.
9.      # pour ipv6 sur le port 443 (https)
10.      listen [::]:443 ssl http2 deferred;
11.
12.      # pour ipv4 sur le port 443 (https)
13.      listen 443 ssl http2 deferred;
14.      return 444;
15. }
```

Quelques précisions supplémentaires sur ce morceau de config. Vous noterez que l'on active HTTP/2 seulement pour le TLS. Il est possible de l'activer en HTTP non sécurisé, mais les navigateurs ne le supportent pas.

Vous remarquez aussi peut-être le default_server. Cela indique à Nginx d'utiliser ce VHOST si l'en-tête Host n'est pas passé avec la requête. Dans le cas d'un accès direct via l'IP par exemple. Et en dernier lieu, return 444 signifie que dans ces cas-là, la connexion sera réinitialisée puisque si plusieurs sites sont hébergés sur cette même ip:port, en l'absence de Host, il n'est pas possible de savoir lequel on doit servir.

Il y a de nombreuses optimisations potentielles au niveau de la pile TCP/IP, non spécifiques à Nginx, nombre d'entre elles sont détaillées dans cette **très bonne ressource** dont j'encourage la lecture.

IX - La thread pool

Il arrive que certaines opérations bloquantes soient lentes (comme la lecture de fichiers sur le disque). Les requêtes de fichiers de taille importante qui ne tiennent pas en mémoire par exemple, bloqueront un thread de Nginx jusqu'à ce que le disque retourne le fichier en question. Admettons-le, c'est assez dommage. Heureusement, Nginx a une solution pour ça !

Il s'agit de placer cette requête dans une « file d'attente » et de traiter d'autres requêtes en attendant que le disque ait fini son opération de lecture. Ainsi, on ne bloque pas plusieurs requêtes en attendant une I/O pour l'une d'entre elles, on place cette dernière en attente et on utilise le thread ainsi libéré pour servir d'autres requêtes.

Un article sur **le blog de Nginx** détaille le fonctionnement de la thread pool et présente un benchmark où les performances en charge sont multipliées par 9 ! Il faut pour cela utiliser l'option **aio** :

```
1. location / {
2.   root /var/www;
3.   aio threads;
4. }
```

N'hésitez pas à me faire part de vos remarques et compléments d'infos en commentaires. Avez-vous été confronté à des problèmes ? Avez-vous pu augmenter les performances ? Faites-nous part de vos expériences !

X - Note de la rédaction de Developpez.com

Nous tenons à remercier **Quentin Busuttil** qui nous a aimablement autorisés à publier son tutoriel : **Optimiser les performances de NGINX**. Nous remercions également **Winjerome** pour la mise au gabarit et **Claude Leloup** pour la relecture orthographique.