

Server web Nginx

Daniele Iamartino



POUL

have a lot of fun...

Cos'è nginx? Di cosa parleremo?

- È un server web pensato per avere le performance migliori
- Nasce nel 2002, progettato appositamente per servire le richieste dirette al sito www.rambler.ru (500 milioni di visite al giorno nel 2008), dove il più noto Apache falliva.
- Oggi è usato da Facebook, Google, Wordpress, Hulu, SourceForge e altri..
- Utilizzato dal 9% dei siti con almeno 1 milione di visitatori
- Rilasciato sotto licenza BSD like
- Si pronuncia “Engine X”
- Scritto in C
- Vedremo come configurarlo per il nostro server web

Le features principali

- Programma in sé leggerissimo in memoria e CPU. Possiamo compilarlo escludendone molte parti per renderlo ancora più leggero.
- Idea principale: un processo per core della CPU è sufficiente a gestire migliaia di richieste. Generare un sacco di processi/thread è inutile (vedi Apache...).
- Utilizzo di socket asincroni per gestire le richieste HTTP.
- File di configurazione più lunghi ma più “logici”, semplici da capire e dettagliati. Possibilità infinite per configurazioni molto particolari (limitazioni di accesso a cartelle specifiche, controlli sui limiti delle richieste, limitazioni di banda, regexp ovunque,...)

Le features principali

- Scritto pensando anche alla sicurezza.
- Non supporta CGI ma solo FastCGI per vari motivi... (feature?)
- **Tanti** interessanti moduli già integrati e altri aggiuntivi abilitabili.
- Load balancing semplicissimo da implementare.
- Dal file di configurazione possiamo modificare **tantissime** impostazioni anche a basso livello (numero di file aperti per ogni processo, chiamata di sistema da usare per gestire i socket asincroni, affinità ai core della CPU, engine di cifratura hardware, ...)

Feature dei moduli

- Tra quelli di default:
 - Compressione gzip, cookie UserID, Access (range IP), Auth Basic, Rewrite, proxy (reverse), FastCGI, Limit requests, identificazione User Agents, load balancing, ...
- Tra quelli aggiuntivi abilitabili:
 - SSL, IPv6, Image filter, RealIP, GeoIP, FLV support, Body addition, ...

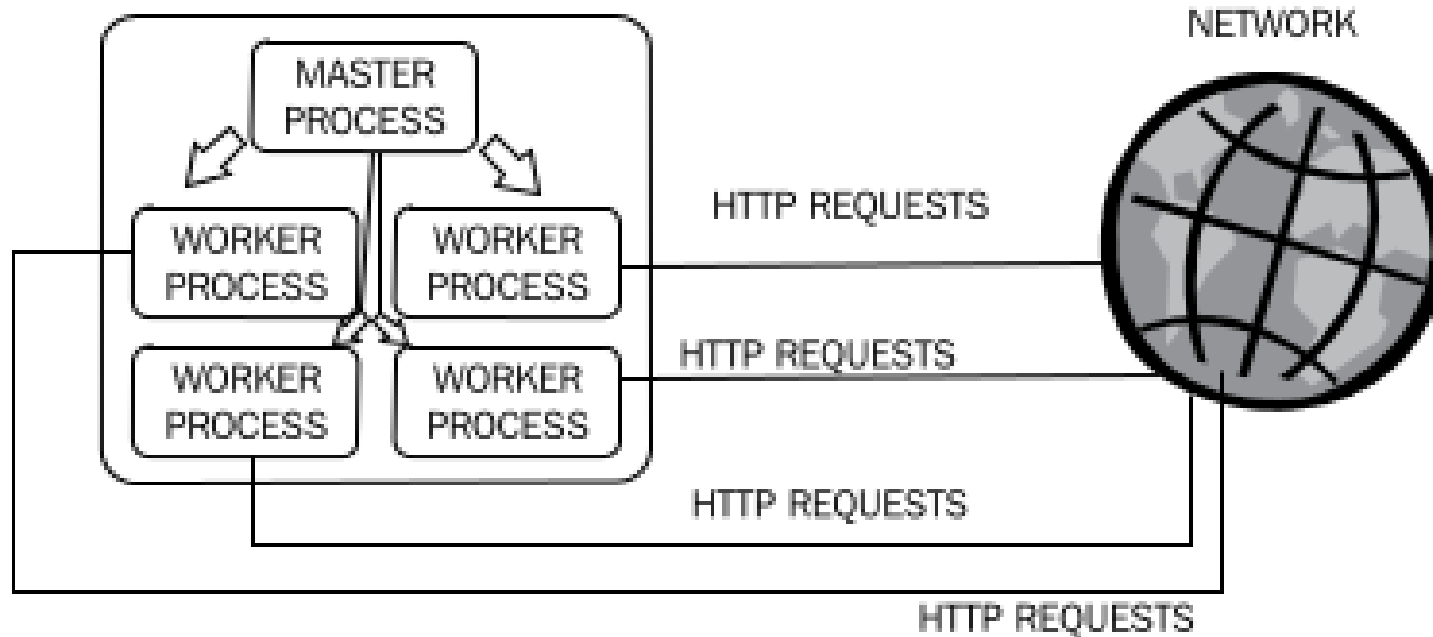
Dove trovarlo

- Sorgenti a www.nginx.org
- Già pacchettizzato in tutte le maggiori distribuzioni
- In atto di compilazione bisogna decidere quali moduli abilitare, assicuratevi che il vostro pacchetto abbia già compilati tutti i moduli che vi servono (**nginx -V**)
- In debian il pacchetto ha tutti i moduli abilitati..

Struttura principale

- È un demone, quindi lo controlliamo come al solito via `/etc/init.d/nginx` o `/etc/rc.d/nginx` . Possiamo anche lanciare il comando “nginx” passando alcuni parametri per gestirlo.
 - `nginx -s start`
`nginx -s stop`
...
- Nginx suddiviso in due processi:
 - **Master process** (lanciato da root, apre il socket in ascolto e legge i file di configurazione)
 - Una serie di **Workers processes** (servono le richieste HTTP, avviati con un altro utente, ad esempio `www-data`)

Struttura principale



La configurazione

- L'intera configurazione si trova in un **unico** file, che però può richiamarne altri (ad esempio uno per sito da ospitare è una buona idea).
- A seconda della distribuzione il file di configurazione `nginx.conf` può trovarsi in percorsi diversi..
- In Debian c'è `/etc/nginx/nginx.conf` già “pronto” che richiama altri file in `/etc/nginx/sites-enabled` (classico come per Apache)
- Quello che dobbiamo fare è:
 - Modifichiamo la configurazione
 - **nginx -t** (testiamo la correttezza sintattica)
 - (Ri)avviamo il demone (`/etc/init.d/nginx restart`)

La configurazione

- All'interno del file di configurazione ci sono una serie di **blocchi** dove sono dichiarate delle **direttive**. Ogni direttiva ha uno o più blocchi dove può stare.
- La sintassi dei file di configurazione è in stile programmatico:
 - # = Commento
 - Ogni riga termina con ;
 - Ogni blocco di impostazioni è compreso tra { }
 - Possiamo richiamare in qualunque punto delle variabili, precedute da \$ (ex: \$user_agent)
 - Richiamiamo altri file di configurazione con “include”
 - È possibile inserire delle condizioni if(\$variabile){...

```
user www-data www-data;
worker_processes 2;
```

**Impostazioni
generali**

```
error_log /var/log/nginx/error.log;
pid /var/run/nginx.pid;
```

```
events {
    worker_connections 128;
}
```

**Impostazioni "tecniche" per
gli workers**

```
http {
    include /etc/nginx/mime.types;
    log_not_found off;

    gzip on;
    gzip_disable "MSIE [1-6]\.(?!.*SV1)";

    server {
        listen 80;
        server_name website.com www.website.com;
        access_log /var/log/nginx/antani.access.log;
        root /var/www;
        location / {
            index index.html;
        }
    }
}
```

Dichiarazione di un sito

Blocco HTTP

Struttura

- Rivediamo passo passo l'esempio:
- “**user www-data www-data**”: specifichiamo utente e gruppo che avranno i processi workers
- “**worker_processes**”: decidiamo quanti processi worker avviare (sono processi fissi, no respawn). È inutile metterne più di uno per core della CPU.
- “**error_log**”: decidiamo dove salvare il file per gli errori
- “**pid**”: ...
- Il blocco “events”: può essere presente solo una volta e serve a dichiarare le impostazioni di rete utilizzate dagli workers

Struttura

- All'interno di events abbiamo “**worker_connections**”: qui decidiamo quante connessioni può gestire ciascun worker.
- Il numero di connessioni da gestire dipende molto dall'hardware che abbiamo a disposizione.
- Qualche esempio “pratico”:
 - CPU dual core, 2GB di ram, 2 workers, 128 connessioni per worker \approx 1 richiesta/sec.
 - CPU quad core, 4GB di ram, 2 workers, 1024 connessioni per worker \approx 50 richieste/sec.
- È consigliabile limitare anche il numero di files aperti per ogni worker (`worker_rlimit_nofile`).
- In generale è conveniente fare dei benchmark (utilizzando ad esempio “`httperf`”) e modificare questi parametri.

Struttura

- Notiamo che i vari blocchi hanno anche una struttura gerarchica
- Il blocco **HTTP** ci permette di dichiarare un “server web” (quindi al suo interno metteremo impostazioni generali del server web)
- All'interno di un blocco **HTTP** possiamo dichiarare un blocco **server**, che corrisponde ad un singolo sito ospitato sul server.
Ogni blocco server è “indipendente”. Stessa idea dei VirtualHost di Apache
- All'interno dei blocchi **server** possiamo dichiarare blocchi **location** dove possiamo specificare impostazioni/azioni da fare solo per quello specifico percorso

Direttive principali del blocco server

- “**listen**” scegliamo la porta e l'indirizzo su cui restare in ascolto (aggiungendo inoltre default scegliamo il server di default in caso di problemi e con “ssl” abilitiamo SSL sul server)
 - `listen 192.168.0.1:80;`
- “**server_name**”: diciamo al blocco server alle richieste per quali siti rispondere. È possibile usare anche regexp!
 - `server_name www.website.com www2.website.com;`
 - `server_name *.website.com;`
 - `server_name ~^\.example\.com$`
- “**root**”: scegliamo la cartella del sito da cui prendere i files.
- “**access_log**” insieme ad “**error_log**” può essere posizionato in vari blocchi. Servono per registrare i log degli accessi alle pagine e i log di errore.

Funzionalità del blocco location

- All'interno di un blocco server possiamo filtrare vari percorsi o file e effettuare operazioni diverse. È possibile utilizzare regex per filtrare percorsi e files!
 - `location = /abc {`
 `[...]`
 `}`
 - `location /abc {` Effettua il match anche su /abcdefghijklm...
 `[...]`
 `}`
 - `location ~ ^/abc$ {`
 `[...]`
 `}`
 - `location *~ ^/abc$ {`
 `[...]`
 `}`

Esempi

- Bloccare l'accesso a tutti i files che iniziano con .ht (per retrocompatibilità con i .htaccess e .htpasswd usati con apache ad esempio...
 - ```
location ~ /\.ht {
 deny all;
}
```
- Potremmo però decidere di lasciare la visione solo all'amministratore che è ad 192.168.0.1...
  - ```
location ~ /admin/ {  
    allow 192.168.0.1;  
    deny all;  
}
```

Esempi

- Potremmo voler proteggere una o più pagine da una password:
 - ```
location /admin/ {
 deny all;
 auth_basic "Authentication required"
 auth_basic_user_file config/htpasswd;
}
```
- Limitiamo la banda per ciascuna connessione a files in una certa cartella..
  - ```
location /downloads/ {  
    limit_rate 500k;  
}
```
-

Altri esempi utili

- Il sito del linuxday?

- ```
location / {
 rewrite ^ http://www.fsugitalia.org/.. permanent;
}
```

- Rewrite più complicata:

- ```
location /documenti/ {  
    rewrite ^/documenti/(.*)$ /doc/$1;  
}
```

- Limitazione sugli accessi

- ```
limit_zone zone_name $binary_remote_addr 10m;
```
- ```
limit_conn zone_name connection_limit;
```
- ```
limit_conn myzone 5;
```

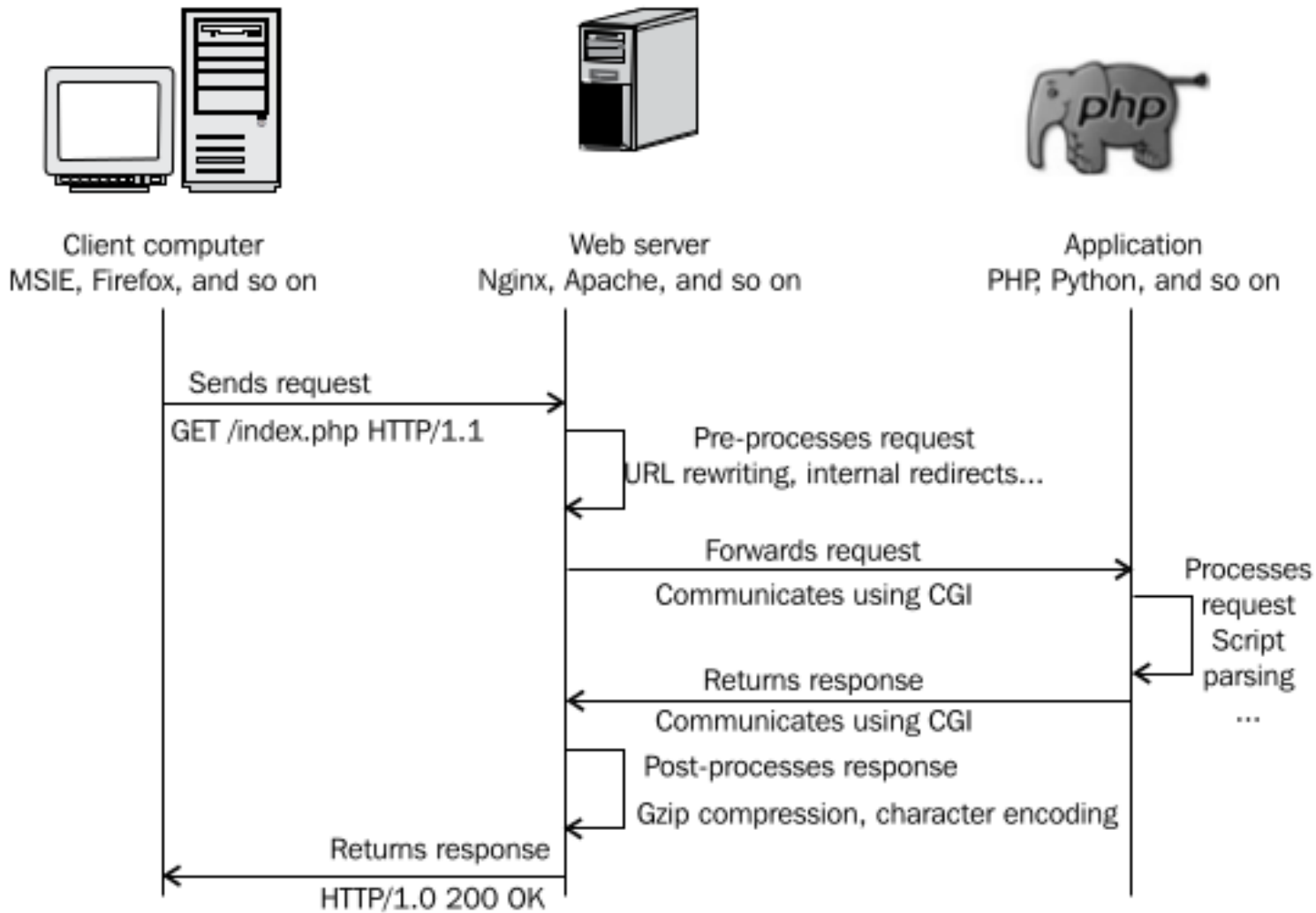
# Altre direttive

- `error_page 404 /not_found.html;`
- `error_page 500 501 502 /server_error.html;`
- **Modulo addition:**
  - `add_after_body /var/www/tail.html`
- **Modulo image filter:**
  - `location ~* \.(png|jpg|gif)$ {  
 image_filter resize 200 100;  
}`
- **Modulo GeoIP (MaxMind):**
  - `geoip_country country.dat;  
geoip_city city.dat;`
  - `if($geoip_country_code=IT) {  
 [...]  
}`

# CGI

- CGI o Common Gateway Interface, è un meccanismo ideato per realizzare applicazioni lato server in qualunque linguaggio di programmazione (C, PHP, Python, Perl, ...) per servire alcune richieste
- Il meccanismo di solito è:
  - Il server riceve una richiesta per servire una certa pagina (relativa all'applicazione)
  - Il server lancia un processo per eseguire il programma e gli passa come parametri da linea di comando le varie variabili passate dalla GET/POST
  - Il programma risponde alla richiesta e il server web inoltra la risposta al client

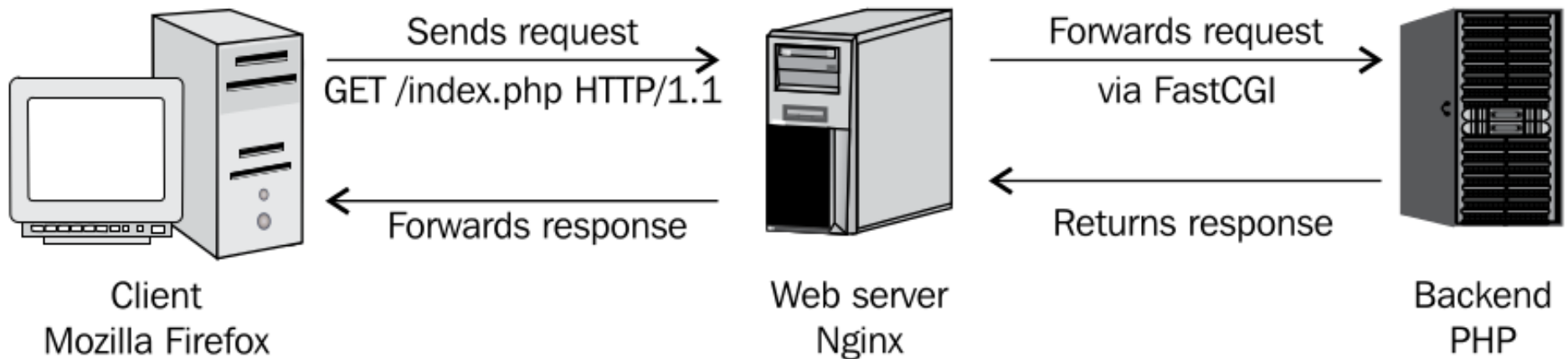
# CGI



# FastCGI

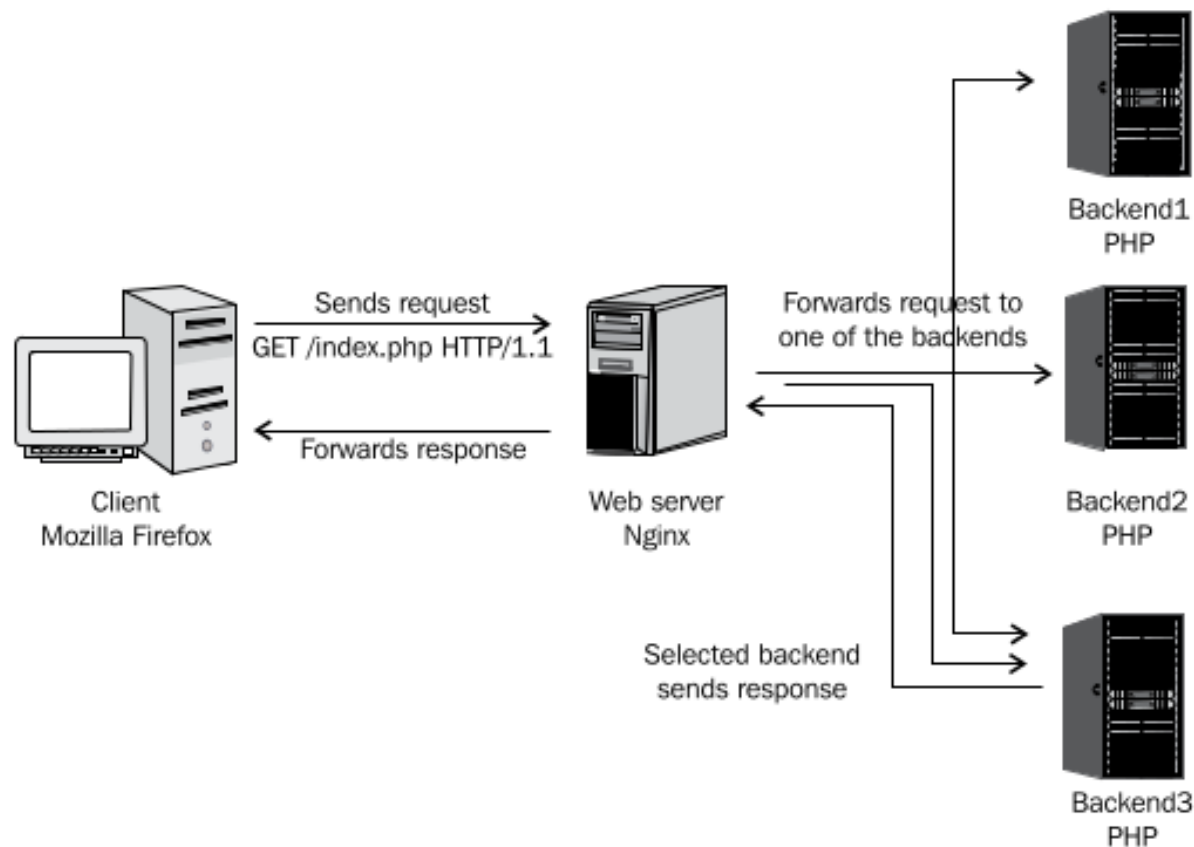
- L'idea di FastCGI è evitare di lanciare un processo per ogni richiesta.
- Avviamo un piccolo server socket per scambiare le richieste tra server web e applicazione. Si occupa poi l'applicazione di generare eventualmente altri processi..
- Quindi:
  - Il server web riceve la richiesta e “chiama” l'applicazione su un socket passandogli i dati della richiesta
  - L'applicazione elabora il risultato e lo restituisce sul socket.

# FastCGI



# CGI vs. FastCGI

- Ok, ma che vantaggio c'è? FastCGI si presta per essere molto più scalabile (l'applicazione/programma che deve gestire la richiesta può essere anche su un server separato). Non è roba da poco!

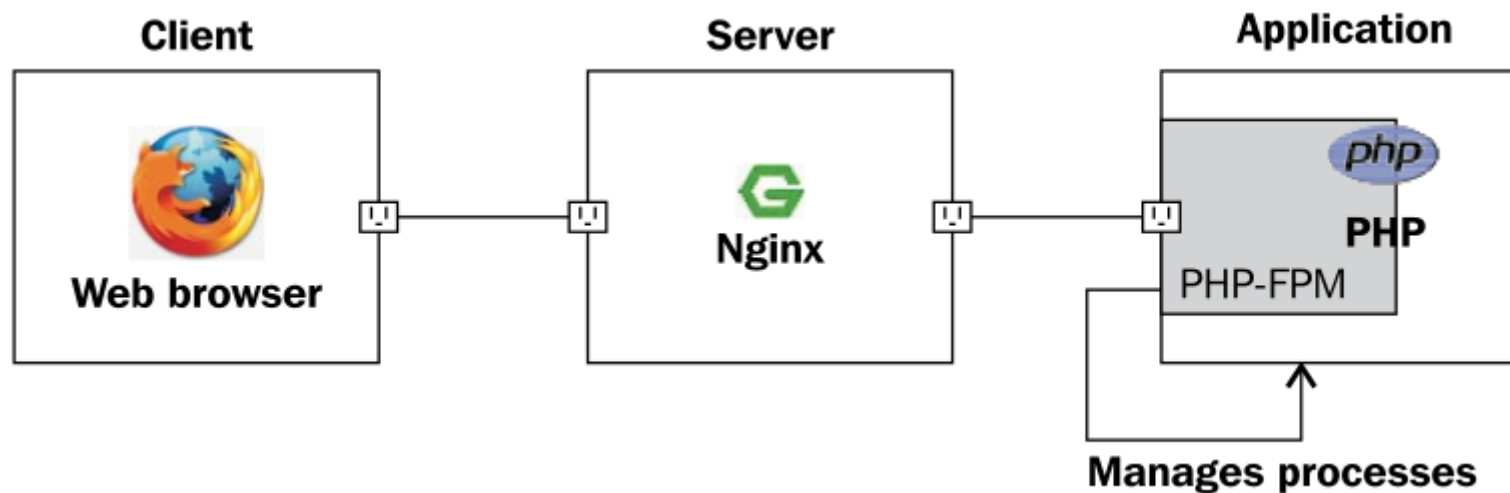


# Perché stiamo parlando di ciò?

- Nginx, per motivi di performance e altro, **non supporta CGI** classici ma solo FastCGI.
- Per nostra fortuna però esistono implementazioni di PHP (e non solo) che supportano FastCGI. Una di queste è **PHP-FPM**.
- Tra l'altro possiamo fare caching sulle richieste FastCGI

# PHP-FPM

- È il metodo consigliato per utilizzare PHP su nginx. (E sembra anche quello più efficiente)
- Si tratta di un servizio che apre un socket (TCP o Unix socket) che riceve le richieste da nginx per servire alcune pagine php generando un gruppo di processi.



# PHP-FPM

- PHP-FPM: **PHP** FastCGI **P**rocess **M**anager
- In Debian non è tra i pacchetti dei repository principali. Possiamo comunque prenderlo dai repository di dotdeb.org
- In alternativa è sempre possibile compilare php5 applicando la patch per abilitare php-pfm
- Dobbiamo fare attenzione a configurarlo di modo da non generare troppi processi (file di configurazione `/etc/php5/fpm/pool.d/www.conf` )
-

# Configurazione lato nginx

```
server {
 server_name *.website.com;
 listen 80;
 root /var/www;
 index index.php;

 location ~* \.php$ {
 fastcgi_pass 127.0.0.1:9000;
 fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;

 fastcgi_param PATH_INFO $fastcgi_script_name;
 include fastcgi_params;
 }
}
```

# Vogliamo scalare? (modulo upstream incluso)

- Niente di più facile: cambiamo la riga

```
fastcgi_pass 127.0.0.1:9000;
```

- Con qualcosa tipo:

```
fastcgi_pass php_backend;
```

- Poi dichiariamo un blocco:

```
upstream php_backend {
 server 192.168.0.101:9000;
 server 192.168.0.102:9000;
 server 192.168.0.103:9000;
}
```

- Potremmo avere qualche problema con cookie di sessione passando da un server di backend all'altro, allora aggiungiamo l'opzione "ip\_hash"
- Lo stesso metodo funziona per scalare nginx.

# Configurazione PHP-FPM

```
[www]
```

```
Listen = 127.0.0.1:9000
```

```
listen.backlog = 10
```

```
listen.allowed_clients = 127.0.0.1
```

```
user = www-data
```

```
group = www-data
```

```
pm = static
```

```
pm.max_children = 5
```

```
pm.max_requests = 10
```

```
chdir = /
```

```
php_admin_value[memory_limit] = 35M
```

# fastcgi\_params

```
fastcgi_param QUERY_STRING $query_string;
fastcgi_param REQUEST_METHOD $request_method;
fastcgi_param CONTENT_TYPE $content_type;
fastcgi_param CONTENT_LENGTH $content_length;

fastcgi_param SCRIPT_NAME $fastcgi_script_name;
fastcgi_param REQUEST_URI $request_uri;
fastcgi_param DOCUMENT_URI $document_uri;
fastcgi_param DOCUMENT_ROOT $document_root;
fastcgi_param SERVER_PROTOCOL $server_protocol;

fastcgi_param GATEWAY_INTERFACE CGI/1.1;
fastcgi_param SERVER_SOFTWARE nginx/$nginx_version;

fastcgi_param REMOTE_ADDR $remote_addr;
fastcgi_param REMOTE_PORT $remote_port;
fastcgi_param SERVER_ADDR $server_addr;
fastcgi_param SERVER_PORT $server_port;
fastcgi_param SERVER_NAME $server_name;

PHP only, required if PHP was built with --enable-force-cgi-redirect
fastcgi_param REDIRECT_STATUS 200;
```

# ...e per altra roba

- Non ho approfondito ma in Django è sufficiente installare “**python-flup**” per abilitare il supporto a FastCGI e lanciare il servizio.
- Se vogliamo utilizzare comunque dei CGI possiamo farlo mettendo un altro server web che li supporta (ad esempio lighttpd) e usare nginx come reverse proxy.. (sembra assurdo ma ha buone performance)
- Lato nginx resta tutto come già visto per PHP-FPM

# Altri spunti

- Varnish come reverse proxy per il caching

# Riferimenti

- Libro “Nginx HTTP Server” di Clément Nedelcu
- <http://wiki.nginx.org/>
- Google?

