# Hardening a Web Application with NGINX - Part 1

*In these exercises, we'll learn how Apache processes connections, and conduct a simple benchmark to determine the capacity of our WordPress application. We will then attempt a denial-of-service attack against the application.*

*Having seen how vulnerable the server is, we will install NGINX Plus in front of Apache, to manage traffic on its behalf. What effect does this have on the system capacity and its weakness to the denial-of-service attack?*

*We will then see how to configure load-balancing with NGINX Plus, how to use the dashboard to identify performance issues, and how to use Health Checks to detect application failures.*

# The Apache ScoreBoard

*You'll access the Apache Scoreboard; a useful status tool that you'll use when diagnosing issues with Apache.*

The Apache Scoreboard displays the state of each Apache worker process. It is disabled by default, and the `setup.sh` script enabled it (details).

Verify that you can access **http://workshopXX.nginxtraining.com/server-status** (or use the IP address if DNS is not ready). Browse your wordpress site and observe the scoreboard:

**Apache Server Status for workshop01.nginxtraining.com (via 178.62.117.197)**

Server Version: Apache/2.4.29 (Ubuntu)
Server MPM: prefork
Server Built: 2018-06-27T17:05:04

---

Current Time: Tuesday, 27-Nov-2018 13:08:15 UTC
Restart Time: Tuesday, 27-Nov-2018 12:50:17 UTC
Parent Server Config. Generation: 2
Parent Server MPM Generation: 1
Server uptime: 17 minutes 57 seconds
Server load: 0.00 0.02 0.04
Total accesses: 123 - Total Traffic: 2.1 MB
CPU Usage: u1.65 s.26 cu0 cs0 - .177% CPU load
.114 requests/sec - 2063 B/second - 17.6 kB/request
6 requests currently being processed, 3 idle workers

```
KWKKKK___.....................................................
..............................................................
......................
```

Scoreboard Key:
"_" Waiting for Connection, "S" Starting up, "R" Reading Request,
"W" Sending Reply, "K" Keepalive (read), "D" DNS Lookup,
"C" Closing connection, "L" Logging, "G" Gracefully finishing,
"I" Idle cleanup of worker, "." Open slot with no current process

| Srv | PID | Acc | M | CPU | SS | Req | Conn | Child | Slot | Client | Protocol | VHost | Request |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0-1 | 9876 | 9/10/18 | K | 0.07 | 1 | 0 | 104.5 | 0.10 | 0.18 | 92.17.177.209 | http/1.1 | 127.0.1.1:80 | GET /wp-content/themes/twentyseventeen/assets/js/jquery.scrollT |
| 1-1 | 9877 | 10/11/18 | W | 0.49 | 0 | 0 | 188.6 | 0.19 | 0.67 | 92.17.177.209 | http/1.1 | 127.0.1.1:80 | GET /server-status HTTP/1.1 |
| 2-1 | 9880 | 5/6/13 | K | 0.02 | 1 | 0 | 139.9 | 0.14 | 0.23 | 92.17.177.209 | http/1.1 | 127.0.1.1:80 | GET /wp-content/themes/twentyseventeen/assets/js/global.js?ver= |
| 3-1 | 9881 | 8/9/14 | K | 0.06 | 1 | 0 | 99.4 | 0.10 | 0.14 | 92.17.177.209 | http/1.1 | 127.0.1.1:80 | GET /wp-includes/js/wp-embed.min.js?ver=4.9.8 HTTP/1.1 |
| 4-1 | 9882 | 7/8/15 | K | 0.01 | 1 | 0 | 173.8 | 0.17 | 0.26 | 92.17.177.209 | http/1.1 | 127.0.1.1:80 | GET /wp-includes/js/comment-reply.min.js?ver=4.9.8 HTTP/1.1 |
| 5-1 | 9894 | 6/13/15 | K | 0.17 | 1 | 0 | 31.5 | 0.18 | 0.18 | 92.17.177.209 | http/1.1 | 127.0.1.1:80 | GET /wp-includes/css/admin-bar.min.css?ver=4.9.8 HTTP/1.1 |

Note the 'Scoreboard' part of the report. This shows the state of each running Apache worker.

# Run a Basic Benchmark against WordPress

*We'll get a baseline performance measurement for the WordPress home page, and use top and the scorecard to observe the effect of the benchmark.*

The **wrk** benchmark tool is already installed on the WordPress server:

## Run the Benchmark

Run **top** in one terminal on the server, and **wrk** in a second:

```
wrk -c 20 -d 30 http://localhost/
Running 30s test @ http://localhost/
  2 threads and 20 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   606.87ms  110.19ms   1.52s    82.09%
    Req/Sec    16.21      8.84    50.00     81.52%
  896 requests in 30.04s, 60.59MB read
  Socket errors: connect 0, read 0, write 0, timeout 9
Requests/sec:     29.83
Transfer/sec:      2.02MB
```

Monitor the Apache scoreboard at the same time.

Check out this article to interpret top CPU utilization data.

Questions:

- What is limiting the performance of the benchmark?
- What does **top** and the **Apache scoreboard** tell you?
- Can you still browse the WordPress site while the benchmark is running?

**If you have the time:** How does the wrk concurrency (**-c N**) affect measured performance, active processes and Apache scoreboard information?

# Attack your WordPress Instance

*Apache and other concurrency-limited servers are very vulnerable to slow-request ("slow loris") attacks. You'll execute a remote attack against a wordpress server using the slowhttptest tool.*

*If you have a local VM (running on your laptop), you can install the attack tool and mount a remote attack against your own server. Otherwise, team up with another user so you can attack their WordPress server.*

## Install slowhttptest

On your local Linux VM, install the build tools. These are already present on the WorkShop servers:

```
sudo bash
apt-get update
apt-get install build-essential git automake libssl-dev
```

Download and build the **slowhttptest** attack tool:

```
git clone https://github.com/shekyan/slowhttptest.git
cd slowhttptest
./configure
make
```

## Run the attack against a remote WordPress server

Run **slowhttptest**, with a concurrency-level of 100 requests:

```
./src/slowhttptest -H -c 100 -u http://workshopXX.nginxtraining.com/
```

Monitor the **Apache Scoreboard** and use **top** to observe the effect of the workload on the WordPress system (CPU and memory usage). Using a web browser, verify if the WordPress site is accessible.

Now, repeat the test with a higher concurrency level of 500 requests:

```
./src/slowhttptest -H -c 500 -u http://workshopXX.nginxtraining.com/
```

One again, monitor the **Apache Scoreboard**, system usage with **top**, and try to access the WordPress site. What is the effect on the WordPress server?

Recovering from a successful attack:

- If Apache becomes unresponsive: **service apache2 restart**
- If WordPress complains it can't get a database connection: **service mysql restart**

**Questions:**

- How does this attack work, and why is it so effective?
- Can you run the **wrk** benchmark at the same time? How many requests succeed?
- What was the CPU utilization? How could you detect this attack was happening?

What does this tell us about the effect of real-world traffic, and the difficulties of predicting performance based on benchmarks?

**If you have the time:** can you harden Apache against this attack by increasing the number of worker processes?

- Edit **mods-available/mpm_prefork.conf**, increase **MaxRequestWorkers** to 250, add a **ServerLimit** of 250 and restart apache.
- Repeat the **slowhttptest** with a higher concurrency.

Can you maintain a *stable* WordPress server?

# Deploy NGINX Plus on WordPress system

*You'll configure Apache to listen on a high port, and then install NGINX Plus to proxy traffic from port 80 to Apache.*

## Reconfigure Apache

Your Apache server is already configured to listen on ports 8001-8005 (using `setup.sh`). We just need to disable port 80:

Edit **/etc/apache2/ports.conf** and comment out the directive `Listen 80`:

```
#Listen 80
```

Stop the service that is listening on port 80, using the **`a2dissite`** command:

```
a2dissite 000-default
```

Then restart apache: **service apache2 restart**

Verify that you can still access the Apache scoreboard **on port 8001**:
**http://workshopXX.nginxtraining.com:8001/server-status**

Note that although Apache is listening on ports 8001-8005, any attempts to access the WordPress server directly on those ports will result in a redirect to the WordPress Site Address on port 80. This issue will be resolved by using NGINX Plus to listen on port 80, and proxy to WordPress on the high ports.

## Configure NGINX Plus to Proxy to WordPress

NGINX Plus is already installed on the WordPress server, but it's not configured.

The main NGINX configuration is located in **/etc/nginx/nginx.conf**. This file references server definitions in **/etc/nginx/conf.d/*.conf**.

Add **wordpress.conf** to the **/etc/nginx/conf.d** directory:

```
server {
    listen 80;

    location / {

        proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header Accept-Encoding "";

        # Set the Host header to your domain...
        proxy_set_header Host workshopXX.nginxtraining.com;

        proxy_pass http://wp_upstreams;
    }

    status_zone wordpress; # NGINX Plus status monitoring
}

upstream wp_upstreams {
       zone wp_upstreams 64k;

       server localhost:8001;
       keepalive 20;
}
```

Add **status.conf** to the **conf.d** directory. This makes the NGINX Plus dashboard available on port 8081:

```
server {
    listen 8081;

    root /usr/share/nginx/html;

    location = / { return 302 /dashboard.html; }

    location /api {
        # In production, you should protect this with an ACL
        api write=off;
    }
}
```

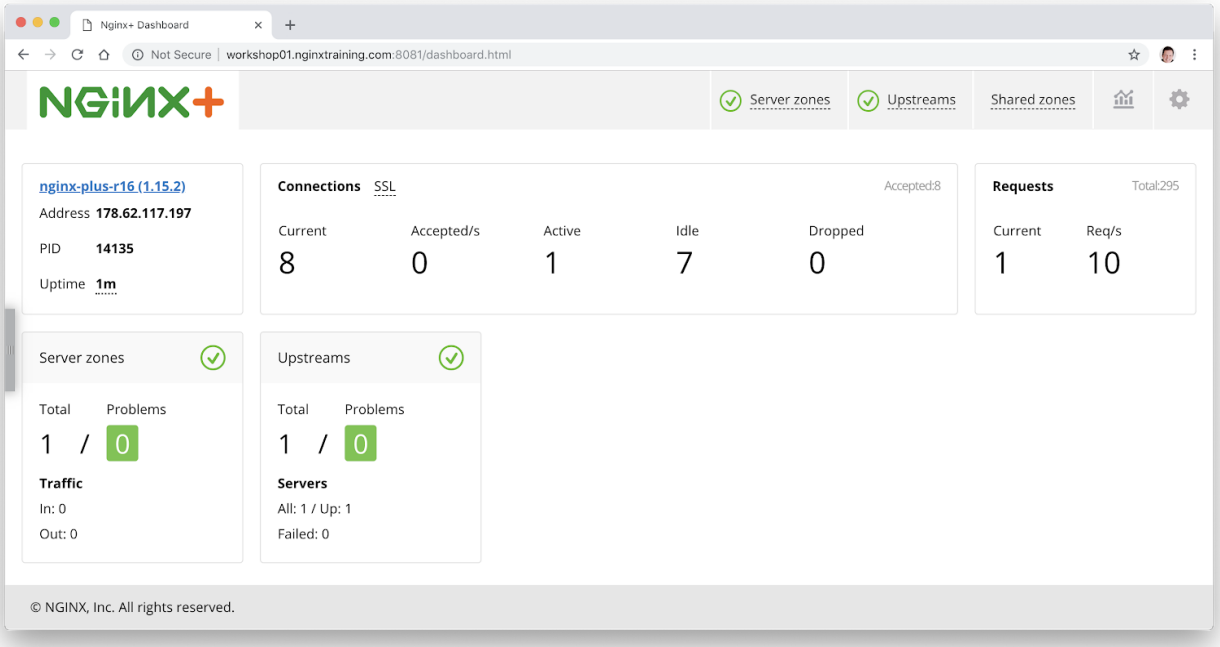
## Test the NGINX Plus proxy

Start nginx: **service nginx start**

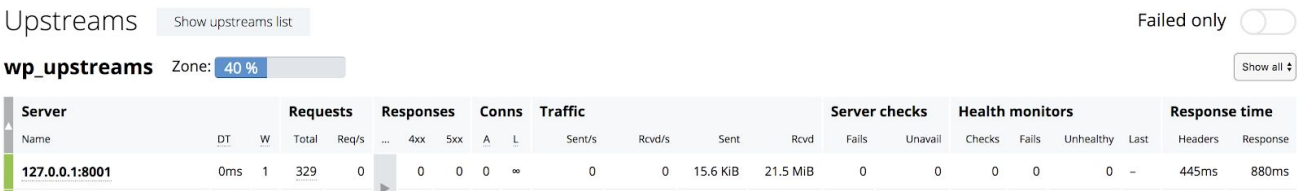
Verify that you can browse the wordpress site: **http://workshopXX.nginxtraining.com/**

Great - you've now installed NGINX Plus and configured it to proxy all traffic to the WordPress application.

Check out the NGINX Plus dashboard: **http://workshopXX.nginxtraining.com:8081/**:




Look at the **Upstreams** tab to see traffic that is sent to WordPress. The easiest way to reset the counters is to reload NGINX: **service nginx reload**



# Re-run the benchmark and slowhttptest attack tool

## Benchmark

Re-run the local wrk benchmark:

```
wrk -c 20 -d 30 http://localhost/
```

**Questions**:

- How does the measured performance and CPU utilization compare to the previous test against Apache?
- How does the Apache Scoreboard compare?
- How can you interpret results from the NGINX Plus dashboard?

## Slowhttptest attack

Re-run the remote slowhttptest attack tool:

```
./src/slowhttptest -H -c 500 -u http://workshopXX.nginxtraining.com/
```

**Questions**:

- How does the measured performance and CPU utilization compare to the previous test against Apache?
- How does the Apache Scoreboard compare?
- What do you see on the NGINX Plus status?

## Both, together

Finally, run the attack tool *and* the wrk benchmark at the same time.  Is the site running effectively?

# Load Balancing with NGINX Plus

*NGINX open source is a highly capable proxy; NGINX Plus adds load-balancing capabilities. You don't have multiple wordpress servers, but we can load-balance across multiple ports on the same Apache server.*

## Configure Apache and NGINX

Apache is already configured to listen on ports 8001 to 8005.

## Configure NGINX

Configure NGINX to load-balance across ports 8001 and 8002.

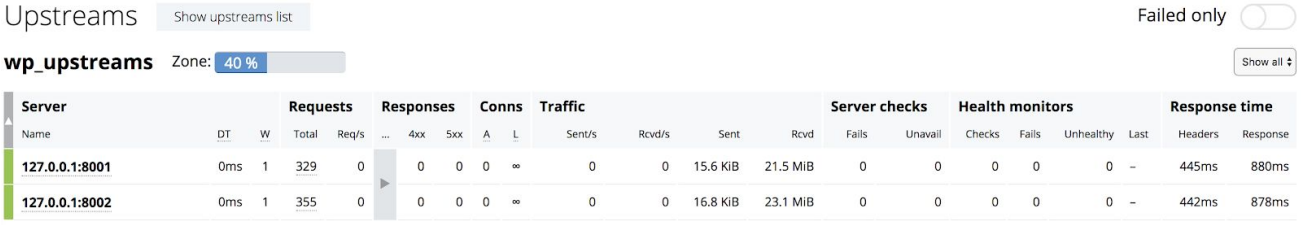
Edit **/etc/nginx/conf.d/wordpress.conf**:

```
upstream wp_upstreams {
       zone wp_upstreams 64k;

       server localhost:8001;
       server localhost:8002;

       keepalive 20;
}
```

Reload nginx: **service nginx reload**

## Test the load balancing

Open the NGINX Plus status page in your browser, and go to the **wp_upstream** upstream group:




Run the local benchmark tool: **wrk -c 20 -d 30 http://localhost/**

Try different levels of concurrency in wrk (**-c 3** and above). Observe the latency measurements from **wrk** and the status page, and the requests/sec figures. How does the concurrency relate to the latency reported by **wrk** and the status page?

Run **wrk** with a concurrency of 5 workers. We'll use that as our reference datapoint:

**wrk -c 5 -d 30 http://localhost/**

## Add a third server

Add the server listening on **port 8003** to the upstream group:

```
upstream wp_upstreams {
       zone wp_upstreams 64k;

       server localhost:8001;
       server localhost:8002;
       server localhost:8003;

       keepalive 20;
}
```

Reload nginx: **service nginx reload**

Rerun the **wrk** benchmark, using concurrency 5. Use the NGINX Plus status page to observe the request distribution across the three upstreams, and the average response time from each. **What is different about the server on port 8003**? **Why does this affect the benchmark results**?

## Tune the load-balancing algorithm

NGINX Plus uses Round Robin load balancing by default. Why is that not effective when the performance of servers varies?

Edit your upstream group to try a couple of alternative load balancing methods, such as least_conn and least_time:

```
upstream wp_upstreams {
       zone wp_upstreams 64k;

       least_conn;

       server localhost:8001;
       server localhost:8002;
       server localhost:8003;

       keepalive 20;
}
```

**If you have time,** experiment with other load-balancing parameters, What other methods can NGINX use to protect vulnerable upstream servers?

### Before you proceed

Remove the server **localhost:8003** from your NGINX configuration, and leave the load-balancing method as **least_conn**.

# Health Checks with NGINX Plus

*During the slowhttptest attack, you may have seen the MySQL server fail. WordPress continues to run, but returns errors. We will investigate this situation further.*

Add the server listening on **port 8004** to your NGINX upstream group:

```
upstream wp_upstreams {
       zone wp_upstreams 64k;

       least_conn;

       server localhost:8001;
       server localhost:8002;
       server localhost:8004;

       keepalive 20;
}
```

Reload nginx: **service nginx reload**

Reload the home page of your blog several times in a web browser. The server on port 8004 has an error; its password for the MySQL database is incorrect. Observe the errors when that server is unable to talk to the database.

Now re-run the **wrk** benchmark (concurrency 5) and monitor the NGINX Plus status page. What do you observe about the requests-per-second sent to each upstream? Why is this happening?

## Implement a Health Check

Edit your NGINX wordpress configuration (**/etc/nginx/conf.d/wordpress.conf**) and add a health check. The default NGINX Plus health check is sufficient:

```
    location / {

        proxy_http_version 1.1; # Always upgrade to HTTP/1.1
        proxy_set_header Connection ""; # Enable keepalives
        proxy_set_header Accept-Encoding ""; # Strip encoding

        proxy_set_header Host workshopXX.nginxtraining.com;

        proxy_pass http://wp_upstreams;

        health_check;
    }
```

Remember to reload NGINX configuration: **service nginx reload**

Check the NGINX Plus status page to see the effects of the health check.  Re-run the browser and **wrk** tests to verify that the health check is effective.

**If you have time:** You can fine-tune the health check to, for example, change the frequency of checks, request a different URI, and perform checks against the status code or response body.

Look at the 'slow_start' parameter for load-balanced servers.  Try correcting the password in the config file for the 8004 server; copy the DB_PASSWORD configuration from **/var/www/html/wp-config.php** to **/var/www/html-8004/wp-config.php**.  If you enable slow_start, do you see the effect if you correct the password on the failed server?  How quickly does it return to full activity once the health check passes?

## Before you proceed

Remove the server **localhost:8004** from your NGINX configuration, and leave the **health_check** enabled.

Add server **localhost:8005** to the configuration.  This server works correctly, so you will finish with 3 working servers in your configuration.

# What have we learnt?

*We have seen how Apache processes connections, the concurrency limitations and the effect of slow connections on the concurrency resources.*

*We deployed NGINX Plus in front of Apache, managing traffic on its behalf.  We saw how this protects Apache and other vulnerable web applications from these slow concurrency attacks.*

*We then configured load-balancing with NGINX Plus, identified and dealt with a slow upstream server, and used a Health Check to route around an application failure.*

*Next, we'll investigate how to improve the performance of the web application using caching, and how to enable TLS/SSL to encrypt and protect traffic.*