

NGINX BASICS

V0.7 – Leif Beaton – l.beaton@f5.com – October 2020

Prereqs:

1. A computer with a working Docker environment. The lab should not impact other containers, but do not interpret this as a guarantee, neither implicit nor explicit.
2. A command interpreter capable of running basic linux commands. If you're on OSX I personally use iTerm2 (<https://www.iterm2.com/downloads.html>), if you're on Windows I personally use cmdr (<https://cmdr.net/>). This is not an endorsement of either tool; simply what I am using and thus what this lab has been tested on.

Begin by cloning (<https://github.com/nginxLeif/nginxbasics.git>) or copying (<https://github.com/nginxLeif/nginxbasics/archive/main.zip>) the repository to a working directory on your Docker enabled computer. If you are copying the repository rather than using git the contents of main.zip must be put into a folder named *nginxbasics*, which is what we define as our root working directory on your docker host.

Inside the working directory there are several files and folders. We will discuss the ones relevant to this workshop as we progress. There are two files required for the workshop to be deployable: *nginx-repo.crt* and *nginx-repo.key*. These are the certificate and key files required to authenticate against the NGINX Plus repository, thus allowing you to deploy NGINX Plus. These files can be acquired by contacting your NIGNX representative, or by requesting a free trial at <https://www.nginx.com/free-trial-request/>. The files should be copied into the working directory under subdirectory *nginxplus/certs*.

Initial Preparations

Once you have set up the working directory and copied in the cert and key, we are almost ready. There are a few steps left:

1. Add a reference to localhost for the domain *wp.nginx.local* in your hosts file. In OSX this is achieved by adding the line below to */etc/hosts* (sudo required) using your favourite text editor, in Windows you add the line to *C:\Windows\System32\drivers\etc\hosts* – and the easiest way to do so is to click the Start button, type NOTEPAD, right click the notepad application and choose “Run as Administrator”. Accept the UAC warning, and click “File” -> “Open...” in Notepad. In the resulting dialog window, navigate to *C:\Windows\System32\drivers\etc* and make sure that the File Type dropdown filter is set to “All Files (*.*)” rather than the default “Text Documents (*.txt)”. You should then be able to double click the “hosts” file to open it for editing. Make sure to save the file after inserting the required line.
 - a. Add the line “127.0.0.1 wp.nginx.local” without the quotes and save.
2. Next, we need to build our environment. Docker is going to do a lot of the heavy lifting here, as it provides a tool called docker-compose. To start the process of building the images,

creating the network, assigning the volumes, and deploying the containers, all we have to do is to ensure we are in our root working directory (*nginxbasics*) in our shell and run the following command:

- a. `docker-compose up -d`
3. This tells docker-compose to look for a file named *docker-compose.yml* in the working directory and use it as a blueprint for building and deploying our environment. This will take a short while.

Once docker-compose is finished and our environment is running (check with “*docker ps*”) we need to make some post-deploy changes. These changes are found in the “*runinterminal.bat*” file in our root working directory. On windows you can run the file as is in a command interpreter that supports Linux commands, in OSX you can either insert “*#!/bin/bash*” without the quotes as the first line of the file, make it executable (*chmod +x filename*) and run it – or, simply run “*bash runinterminal.bat*” without the quotes. You could also go simple and copy the contents of the file and paste it to your terminal. Using any of the methods above, this will prepare the environment, set up our WordPress instances - and break some stuff... 😊

In our environment we have several running containers now, the important ones are:

nginxbasics_nginxplus_1
nginxbasics_wordpress1_1
nginxbasics_wordpress2_1
nginxbasics_wordpress3_1
nginxbasics_wordpress4_1
nginxbasics_wordpress5_1

There is also a mysql database container and a couple of supporting containers we need not concern ourselves with for the purposes of this workshop.

The WordPress instances are accessible directly from the host using the following URLs:

1. <http://wp.nginx.local:8001>
2. <http://wp.nginx.local:8002>
3. <http://wp.nginx.local:8003>
4. <http://wp.nginx.local:8004>
5. <http://wp.nginx.local:8005>

The NGINX Plus container is accessible directly from the host using the following URL:

1. <http://wp.nginx.local>

Primer

NGINX Plus ships with a few basic HTML pages, and some other goodies, and that makes it easy for us to test the web server use case. First, we need to understand the config file structure of NGINX Plus.

NGINX and NGINX Plus are similar to other services in that they use a text-based configuration file written in a particular format. By default, the file is named *nginx.conf* and for NGINX Plus is placed in the */etc/nginx* directory. (For NGINX Open Source, the location depends on the package system used to install NGINX and the operating system. It is typically one of */usr/local/nginx/conf*, */etc/nginx*, or */usr/local/etc/nginx*.)

The configuration file consists of directives and their parameters. Simple (single-line) directives each end with a semicolon. Other directives act as “containers” that group together related directives, enclosing them in curly braces (*{ }*); these are often referred to as blocks.

Simple directives:

```
user          nobody;
error_log     logs/error.log notice;
worker_processes 1;
```

To make the configuration easier to maintain, it is recommended that we split it into a set of feature-specific files stored in the */etc/nginx/conf.d* directory and use the include directive in the main *nginx.conf* file to reference the contents of the feature-specific files:

```
include conf.d/http;
include conf.d/stream;
include conf.d/exchange-enhanced;
```

A few top-level directives, referred to as contexts, group together the directives that apply to different traffic types:

- **events** – General connection processing
- **http** – HTTP traffic
- **mail** – Mail traffic
- **stream** – TCP and UDP traffic

Directives placed outside of these contexts are said to be in the main context.

The following configuration illustrates the use of contexts:

```
user nobody; # a directive in the 'main' context

events {
    # configuration of connection processing
}

http {
    # Configuration specific to HTTP and affecting all virtual servers

    server {
```

```

    # configuration of HTTP virtual server 1
    location /one {
        # configuration for processing URIs starting with '/one'
    }
    location /two {
        # configuration for processing URIs starting with '/two'
    }
}

server {
    # configuration of HTTP virtual server 2
}

stream {
    # Configuration specific to TCP/UDP and affecting all virtual servers
    server {
        # configuration of TCP virtual server 1
    }
}

```

In general, a child context – one contained within another context (its parent) – inherits the settings of directives included at the parent level. Some directives can appear in multiple contexts, in which case you can override the setting inherited from the parent by including the directive in the child context.

Module 1 – Web Server

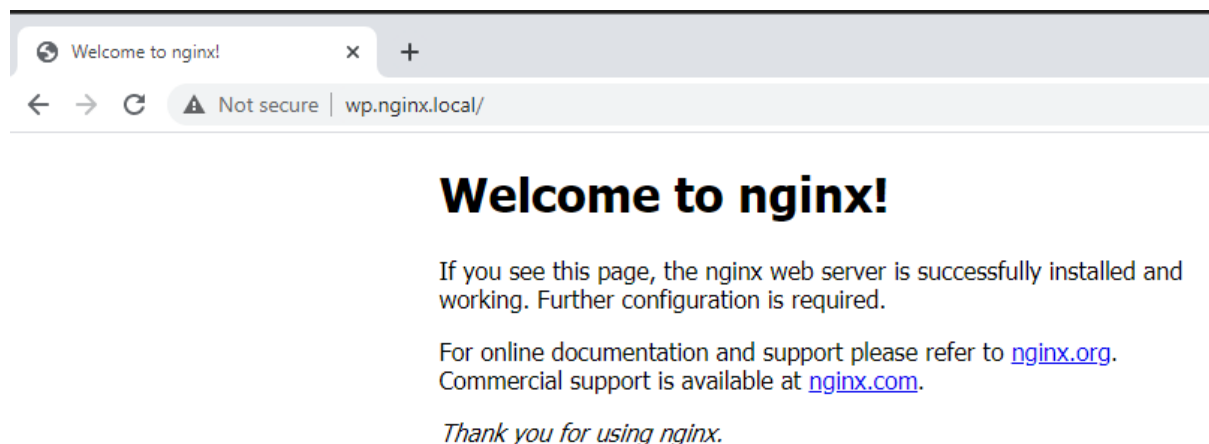
Within the *nginxplus/config* directory, create/edit the **default.conf** file to reflect the following contents:

```
server {  
    listen 80;  
  
    location / {  
        root /usr/share/nginx/html;  
        index index.html;  
    }  
}
```

Make sure to reload the configuration after saving the file (*docker exec nginxbasics_nginxplus_1 nginx -s reload*).

The above config defines a virtual server, telling it to listen to port 80 on any interface (*:80, 0.0.0.0:80), and sets up a location matching block using a prefix match for “/”. When a request comes in on port 80 that is matched by the prefix of “/” we will serve content from disk, starting from the */usr/share/nginx/html* directory. If no specific resource is requested, we will attempt to deliver a resource named *index.html*.

Open your browser and navigate to <http://wp.nginx.local> and verify that you see a result similar to this:



Module 2 – Reverse Proxy

In the root working directory on the docker host, run “*docker inspect nginxbasics_wordpress1_1 | grep IPAddress*” and make note of the IP address of the container. In my case it is 192.168.91.5. Update the below config snippet to reflect your IP address.

Within the *nginxplus/config* directory, edit the *default.conf* file to reflect the following contents:

```
server {  
    listen 80;  
  
    location / {  
        proxy_set_header Host "wp.nginx.local";  
        proxy_pass http://192.168.91.5;  
    }  
}
```

Make sure to reload the configuration after saving the file (*docker exec nginxbasics_nginxplus_1 nginx -s reload*).

In this example, our virtual server is still listening on port 80, and it still has a prefix matching block looking for “/”. Now, however, we are not serving content locally from disk; rather, we are proxying the request to an upstream server, namely 192.168.91.5. We are also modifying a request header, Host, to reflect the expected domain. In our WordPress example this is redundant, but it is important to note this trick – as many upstreams break if they do not receive the expected host header. If we had not included this modified header, the upstream would see a Host request header value of “*http://192.168.91.5*”.

Open your web browser and navigate to <http://wp.nginx.local> and verify that you see a result similar to this:



Module 3 – Load Balancing

In the root working directory on the docker host, run “*docker inspect nginxbasics_wordpress2_1 | grep IPAddress*” and make note of the IP address of the container. In my case it is 192.168.91.8. Update the below config snippet to reflect your IP address.

Within the *nginxplus/config* directory, edit the **default.conf** file to reflect the following contents:

```
upstream wordpress {
    server 192.168.91.5; #wp1
    server 192.168.91.8; #wp2
}

server {
    listen 80;

    location / {
        proxy_set_header Host "wp.nginx.local";
        proxy_pass http://wordpress;
    }
}
```

Make sure to reload the configuration after saving the file (*docker exec nginxbasics_nginxplus_1 nginx -s reload*).

In this example we are defining an upstream group (load balancing pool) and naming it “wordpress”. Within the upstream group we are defining two members, the two servers described by their IP addresses above. Note that we do not have to use IP addresses; resolvable host/domain names are also acceptable. We could also specify non-standard ports on a per-member basis if needed.

Within the location block we are changing the *proxy_pass* directive to no longer proxy to one specific IP; rather, we are proxying to the upstream group named “wordpress”. Since we made no definition of load balancing algorithm, we are now performing round-robin load balancing of inbound requests.

Open your web browser and navigate to <http://wp.nginx.local> and reload the page a few times. Observe that it alternates between “My Awesome Site 1” and “My Awesome Site 2”.

Module 4 – There Be Dragons

In the root working directory on the docker host, run “*docker inspect nginxbasics_wordpress3_1 | grep IPAddress*” and make note of the IP address of the container. In my case it is 192.168.91.4. Update the below config snippet to reflect your IP address.

Within the *nginxplus/config* directory, edit the **default.conf** file to reflect the following contents:

```
upstream wordpress {
    server 192.168.91.5; #wp1
    server 192.168.91.8; #wp2
    server 192.168.91.4; #wp3
}

server {
    listen 80;

    location / {
        proxy_set_header Host "wp.nginx.local";
        proxy_pass http://wordpress;
    }
}
```

Make sure to reload the configuration after saving the file (*docker exec nginxbasics_nginxplus_1 nginx -s reload*).

We have now added a third member to our Upstream Group named “wordpress”. We still have not specified a load balancing algorithm, thus still defaulting to round-robin.

Open your web browser and navigate to <http://wp.nginx.local> and reload the page a good few times.

Jumping Jehosaphat! We seem to have a failure rate of 33%!!!

No need to panic, this is an intentional “bug”. The inquisitive amongst us will soon realize that I have simply thrown an error in the *index.php* file of the third instance. It gives us something real-world to work with. In our lab we are lucky; the website clearly identifies which upstream node it originated from, so troubleshooting becomes a bit easier. In the real world though, additional steps would need to be made to isolate where the error occurs, and even worse, is it just one upstream failing or several? Is it failing constantly or intermittent? If only we had greater visibility into the goings on of the NGINX Plus instance...

Module 5 – Let There Be Light

Within the `nginxplus/config` directory, create/edit the **monitoring.conf** file to reflect the following contents:

```
server {
    listen 8000;

    location /api {
        api write=on;
    }
    location = /dashboard.html {
        root /usr/share/nginx/html;
    }
}
```

Make sure to reload the configuration after saving the file (`docker exec nginxbasics_nginxplus_1 nginx -s reload`).

We now created an additional config file, within which we defined another virtual server – this time, listening on port 8000. This virtual server is set up for monitoring purposes, specifically to expose the NGINX Plus API and the bundled Extended Status Monitoring dashboard. Since the main config file for NGINX Plus (`/etc/nginx/nginx.conf`) defines an import directive for all files ending with `.conf` in the `/etc/nginx/conf.d` directory our newly created file will be included automatically.

Open your web browser and navigate to <http://wp.nginx.local:8000/dashboard.html> and verify that you see a result similar to this:

Connections				SSL
Current	Accepted/s	Active	Idle	
7	0	1	6	

We have a status dashboard! Yet, alas, there does not seem to be too much information in it...

This is because we have yet to tell NGINX Plus what it should be monitoring. Let us fix that next.

Module 6 – Make It Brighter

Within the *nginxplus/config* directory, edit the **default.conf** file to reflect the following contents:

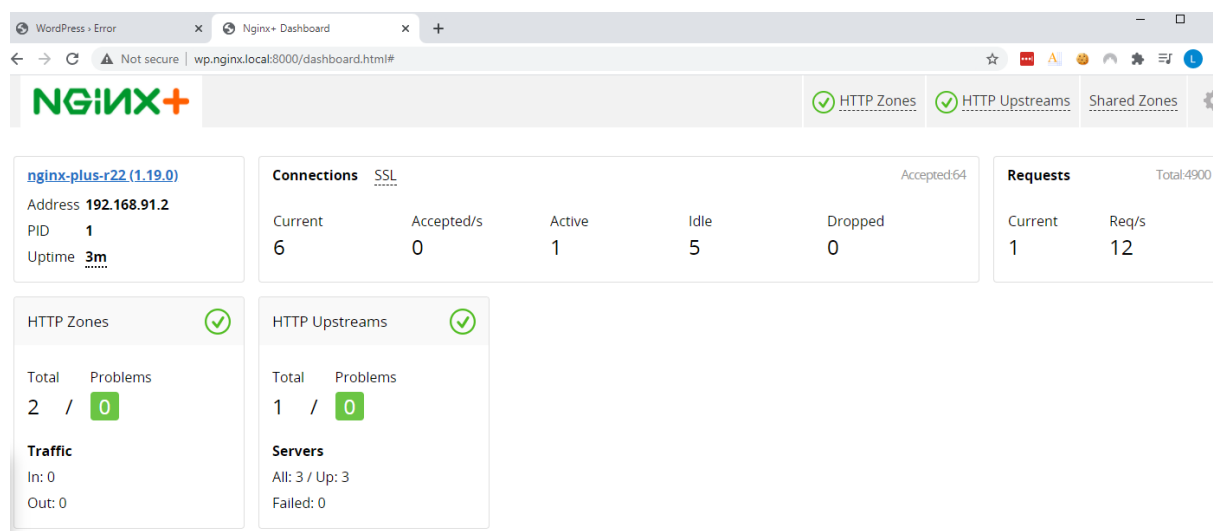
```
upstream wordpress {
    zone    status_wordpress 64k;
    server 192.168.91.5; #wp1
    server 192.168.91.4; #wp2
    server 192.168.91.7; #wp3
}

server {
    status_zone portEighty;
    listen 80;

    location / {
        status_zone portEighty_locationSlash;
        proxy_set_header Host "wp.nginx.local";
        proxy_pass http://wordpress;
    }
}
```

Make sure to reload the configuration after saving the file (*docker exec nginxbasics_nginxplus_1 nginx -s reload*).

What we are doing in this example is to tell the status monitoring what it should keep an eye on for us, and we are also setting up some shared memory zones for data collection. If we return to our web browser and reload the dashboard (it may have auto-reloaded) we should now have much more data at hand:



Not only does the splash screen have more data; we have three additional tabs at the top right to investigate! If we visit our web site (<http://wp.nginx.local>) and reload a number of times, we should be able to pinpoint the culprit, or indeed culprits, in the *HTTP Upstreams* tab. Easy!

Module 7 – A Quick Fix

We now have status monitoring, but the web site is still broken. The obvious solution is to fix the *index.php* file I sabotaged, but in a real-life scenario we might have to remedy before we repair. Luckily, since we have a load balancer in place, we have several ways of easily solving this issue.

One apparent remedy, now that we have identified the misbehaving upstream, is to simply remove the offender from the upstream group and reload the config. This will remedy the issue, sure, but it requires interaction both to remove it and to put it back when it is healthy. Not to mention that we need to detect the failure to begin with before any action takes place. Would it not be better if we could make the load balancer respond to such event on its own?

NGINX Plus has many weapons in its arsenal, and the one best suited to this scenario is its Active Health Check capability. In simple terms, NGINX Plus can check not only if an upstream is healthy, but it can also verify that the upstream is responding in a predictable manner. We can verify health by looking at returned status codes, response headers, or even do regular expression matching on the response body! The Active Health Check engine in NGINX Plus is rich and very configurable. But it can also be super easy.

Within the *nginxplus/config* directory, edit the **default.conf** file to reflect the following contents:

```
upstream wordpress {
    zone    status_wordpress 64k;
    server  192.168.91.5; #wp1
    server  192.168.91.4; #wp2
    server  192.168.91.7; #wp3
}

server {
    status_zone portEighty;
    listen 80;

    location / {
        status_zone portEighty_locationSlash;
        proxy_set_header Host "wp.nginx.local";
        proxy_pass http://wordpress;
        health_check;
    }
}
```

Make sure to reload the configuration after saving the file (*docker exec nginxbasics_nginxplus_1 nginx -s reload*).

If we now return to our web site (<http://wp.nginx.local>) and reload several times, we see that the environment is healthy! Let us check with the Extended Status dashboard on the *HTTP Upstreams* tab to see what is going on:

My Awesome Site 1 – Just another...Nginx+ Dashboard

Not secure | wp.nginx.local:8000/dashboard.html#upstreams

NGINX+

HTTP Zones

HTTP Upstreams

Shared Zones

HTTP Upstreams

Show upstreams list

Failed only

wordpress

Zone: 40 %

Show all

Server	Requests	Responses	Conns	Traffic	Server checks	Health monitors	Response time													
Name	DT	W	Total	Req/s	4xx	5xx	A	L	Sent/s	Rcvd/s	Sent	Rcvd	Fails	Unavail	Checks	Fails	Unhealthy	Last	Headers	Response
192.168.91.5:80	0ms	1	5	0	0	0	0	∞	0	0	2.37 KiB	269 KiB	0	0	29	0	0	passed	67ms	67ms
192.168.91.4:80	0ms	1	4	0	0	0	0	∞	0	0	1.90 KiB	215 KiB	0	0	29	0	0	passed	58ms	58ms
192.168.91.7:80	2m	1	0	0	0	0	0	∞	0	0	0	0	0	0	29	29	1	failed	-	-

We see here that the third instance is in a failed state, and therefore it will not receive traffic. It will remain in a failed state until it starts responding in a predictable manner again, at which time the Active Health Check will allow for it to receive traffic again – all without human intervention.

Read more about the Active Health Check function and its parameters here:

https://nginx.org/en/docs/http/nginx_http_upstream_hc_module.html#health_check

Module 8 – But what now?

Let us start by adding two more members to the upstream group.

Within the *nginxplus/config* directory, edit the **default.conf** file to reflect the following contents:

```
upstream wordpress {
    zone    status_wordpress 64k;
    server 192.168.91.5; #wp1
    server 192.168.91.4; #wp2
    server 192.168.91.7; #wp3
    server 192.168.91.9; #wp4
    server 192.168.91.6; #wp5
}

server {
    status_zone portEighty;
    listen 80;

    location / {
        status_zone portEighty_locationSlash;
        proxy_set_header Host "wp.nginx.local";
        proxy_pass http://wordpress;
        health_check;
    }
}
```

Make sure to reload the configuration after saving the file (*docker exec nginxbasics_nginxplus_1 nginx -s reload*).

Return to our web site (<http://wp.nginx.local>) and reload many times. Did you notice anything out of the ordinary? It seems everything kind of freezes intermittently. Let us head over to the Extended Status Dashboard in the *HTTP Upstreams* tab to investigate:


HTTP Upstreams

Show upstreams list

Failed only


☐

wordpress



Zone:

40 %

Show all 

Server			Requests		Responses		Conns		Traffic				Server checks		Health monitors				Response time		
Name	DT	W	Total	Req/s	...	4xx	5xx	A	L	Sent/s	Rcvd/s	Sent	Rcvd	Fails	Unavail	Checks	Fails	Unhealthy	Last	Headers	Response
192.168.91.5:80	0ms	1	4	0		0	0	0	∞	0	0	1.90 KiB	215 KiB	0	0	58	0	0	passed	42ms	42ms
192.168.91.4:80	0ms	1	4	0		0	0	0	∞	0	0	1.90 KiB	215 KiB	0	0	58	0	0	passed	50ms	50ms
192.168.91.7:80	4m	1	0	0		0	0	0	∞	0	0	0	0	0	0	58	58	1	failed	-	-
192.168.91.9:80	0ms	1	4	0		0	0	0	∞	0	0	1.90 KiB	107 KiB	0	0	29	0	0	passed	5107ms	5107ms
192.168.91.6:80	0ms	1	4	0		0	0	0	∞	0	0	1.90 KiB	215 KiB	0	0	58	0	0	passed	51ms	51ms

Instance 4 is responding much slower than its peers. Can we fix that issue in config, I wonder? It passes health checks because it is responding predictably. It does so excruciatingly slow though. Let us play with load balancing algorithms next.

Module 9 – Choices, choices

Round-robin is a computationally fast algorithm. It is so due to it not requiring knowledge of only one thing to make a load balancing decision: who is next in line? The trade-off, however, is that it is not particularly clever. In our scenario, where we are experiencing a systems degradation of one (or more) upstreams, either permanently as in our case (I broke it), or intermittently – we can see tremendous benefit to the user experience by employing a smarter load balancing algorithm. It so happens that NGINX Plus has the perfect one up its sleeve in the shape of *least_time*.

Within the *nginxplus/config* directory, edit the **default.conf** file to reflect the following contents:

```
upstream wordpress {
    least_time last_byte;
    zone    status_wordpress 64k;
    server 192.168.91.5; #wp1
    server 192.168.91.4; #wp2
    server 192.168.91.7; #wp3
    server 192.168.91.9; #wp4
    server 192.168.91.6; #wp5
}

server {
    status_zone portEighty;
    listen 80;

    location / {
        status_zone portEighty_locationSlash;
        proxy_set_header Host "wp.nginx.local";
        proxy_pass http://wordpress;
        health_check;
    }
}
```

Make sure to reload the configuration after saving the file (*docker exec nginxbasics_nginxplus_1 nginx -s reload*).

Return to our web site (<http://wp.nginx.local>) and reload many times. You will note that there is still the delay on the first hit, but as the load balancer learns the response times of each upstream it will automatically distribute traffic in a sensible fashion.

Module 10 – My API is exposed!

In Modules 5 and 6 we enabled the Extended Status Monitoring dashboard, but for it to function it relies heavily on the NGINX Plus API. We enabled the API in the same modules, but we did nothing to protect it. In a real-world scenario we would be exposed, and our attack vector would be rather broad. Luckily, since the API is defined in a location directive, we have all the safety measures available to any other location available to protect the API as well. Let us tie it down a bit, shall we?

For the sake of simplicity, we will enable basic authentication. First, run the following command to step into the shell of our NGINX Plus container:

```
docker exec -it nginxbasics_nginxplus_1 /bin/sh
```

Within the shell, first install apache2-utils by running this command:

```
apk add apache2-utils
```

Then, let us generate a password file:

```
htpasswd -c /etc/nginx/.htpasswd leif
```

You will be prompted for a password twice. Create one you will remember; no complexity requirements are set. You can verify the created file by running this:

```
cat /etc/nginx/.htpasswd
```

You should receive output similar to this:

```
/ # cat /etc/nginx/.htpasswd  
leif:$apr1$6EbX9hic$mdlrC1X9IfyVtRKxmqdQ/
```

Exit the container by typing *exit* followed by enter. Next, within the *nginxplus/config* directory, create/edit the **monitoring.conf** file to reflect the following contents:

```
server {  
    listen 8000;  
  
    location /api {  
        limit_except GET {  
            auth_basic "NGINX Plus API";  
            auth_basic_user_file /etc/nginx/.htpasswd;  
        }  
        api write=on;  
        allow 192.168.91.0/24;  
        deny all;  
    }  
    location = /dashboard.html {  
        root /usr/share/nginx/html;  
    }  
    location /swagger-ui {  
        root /usr/share/nginx/html;  
    }  
}
```

Make sure to reload the configuration after saving the file (`docker exec nginxbasics_nginxplus_1 nginx -s reload`).

We can verify that our dashboard (<http://wp.nginx.local:8000/dashboard.html>) still works fine. We can also make API requests directly, both via the browser and via curl. Example request:

curl <http://wp.nginx.local:8000/api/6/http/upstreams/wordpress/servers/3>

This gives me a JSON object that contains the settings for member 4 of my upstream group – the slow one.

What if I wanted to manipulate it, or even delete it? I can do that in config, of course, but also:

curl --user "leif" --request PATCH -d '{"down": true}'
<http://wp.nginx.local:8000/api/6/http/upstreams/wordpress/servers/3>

Would you look at that – an authentication prompt! Let us provide our credentials and see what the response is:

```
{"id":3,"server":"192.168.91.9:80","weight":1,"max_conns":0,"max_fails":1,"fail_timeout":"10s","slow_start":"0s","route":"","backup":false,"down":true}
```

It looks like we have marked our slow upstream as down. Let us check the dashboard:

HTTP Upstreams [Show upstreams list](#)

wordpress Zone: **40 %**

Server	Requests		Response			
	Name	DT	W	Total	Req/s	4xx
192.168.91.5:80	0ms	1	5	0	0	0
192.168.91.4:80	0ms	1	6	0	0	0
192.168.91.7:80	23m	1	0	0	0	0
192.168.91.9:80	0ms	1	7	0	0	0
192.168.91.9:80	0ms	1	96	0	0	0

192.168.91.9
192.168.91.9:80
down
Total downtime: 0ms
Last check: passed

Success! We have now altered config in memory, in real-time! Well done!

The eagle eyed observer will have noted that we exposed the swagger-ui for NIGNX Plus as well; you can peruse it at <http://wp.nginx.local:8000/swagger-ui>.

Self Study (optional)

HTTPS/TLS:

<https://docs.nginx.com/nginx/admin-guide/security-controls/terminating-ssl-tcp/>

Caching:

<https://docs.nginx.com/nginx/admin-guide/content-cache/content-caching/>