Tutorial 3 - Setting up an Apache/PHP application server

What are we doing?

We are setting up an Apache/PHP application server using the minimal number of modules necessary.

Why are we doing this?

A bare bones Apache server is suitable for delivering static files. More complex applications rely on dynamic content. This requires an extension of the server. A very fast and secure application server can be implemented using *suEXEC*, an external *FastCGI daemon* and *PHP*. This is by far not the only option and also likely not the most widespread one in a corporate environment. But is a very simple architecture perfectly suited for a test system.

Requirements

- An Apache web server, ideally one created using the file structure shown in Tutorial 1 (Compiling an Apache web server).
- Understanding of the minimal configuration in Tutorial 2 (Configuring a minimal Apache server).

Step 1: Configuring Apache

We'll configure the web server to start off with, aware of the fact that it is not yet capable of running in this configuration. Based on the minimal web server described in Tutorial 2, we'll configure a very simple application server in the *conf/httpd.conf_fastcgid* file.

```
ServerName              localhost
ServerAdmin             root@localhost
ServerRoot              /apache
User                    www-data
Group                   www-data

ServerTokens            Prod
UseCanonicalName        On
TraceEnable             Off

Timeout                 5
MaxRequestWorkers       250

Listen                  127.0.0.1:80

LoadModule              mpm_event_module        modules/mod_mpm_event.so
LoadModule              unixd_module            modules/mod_unixd.so

LoadModule              log_config_module       modules/mod_log_config.so
LoadModule              mime_module             modules/mod_mime.so

LoadModule              authn_core_module       modules/mod_authn_core.so
LoadModule              authz_core_module       modules/mod_authz_core.so

LoadModule              suexec_module           modules/mod_suexec.so
LoadModule              fcgid_module            modules/mod_fcgid.so

ErrorLogFormat          "[%{cu}t] [%-m:%-l] %-a %-L %M"
LogFormat               "%h %l %u [%{%Y-%m-%d %H:%M:%S}t.%{usec_frac}t] \"%r\" %>s %b \
\"%{Referer}i\" \"%{User-Agent}i\"" combined

LogLevel                debug
ErrorLog                logs/error.log
CustomLog               logs/access.log combined

AddHandler              fcgid-script .php

DocumentRoot            /apache/htdocs
```

```
<Directory />

        Require all denied

        Options         SymLinksIfOwnerMatch
        AllowOverride   None

</Directory>

<VirtualHost 127.0.0.1:80>

      <Directory /apache/htdocs>

        Require all granted

        Options          ExecCGI
        AllowOverride    None

        FCGIWrapper      /apache/bin/php-fcgi-starter/php-fcgi-starter.php

      </Directory>

</VirtualHost>
```

I will no longer be discussing the overall configuration, but will instead only be focusing on the differences from Tutorial 2. There are three new modules: Added to the *suEXEC* and *FCGI* modules is the *Mime module*, which enables us to assign the *.php* file suffix to the *FCGI daemon*. It is assigned using the *AddHandler directive*.

The */apache/htdocs* directory now needs the additional *ExecGGI* option. And finally, the *FCGIWrapper*. This is the connection between the web server and the *FCGI daemon* yet to be configured. Once the first request using the *.php* suffix is received by the web server, the server calls a *wrapper script*, starting the *FCGI daemon*, which from this time on handles the *PHP requests*.

FastCGI is a method for executing dynamic program code from a web server. It is a very fast method which for the most part leaves the server untouched and runs the application on a separate daemon. To increase the speed, *FastCGI* provides multiple instances of this daemon, allowing requests to be processed without having to wait. In practice, this is a promising gain in performance and, more importantly, an architecture that saves memory, as will be discussed in more detail below.

Step 2: Compiling Apache with suEXEC support

We now have to compile two missing modules and deploy other components for the *FCGI daemon*. Let's start with the *suEXEC module*.

An Apache web server was compiled in Tutorial 1. However, although the *--enable-mods-shared=all* option was used, *suexec* has not been compiled yet. Hence, the module is so special that it can't be compiled as a default module, although it is present in the source code.

The web server configured in Tutorial 2 runs as user *www-data* or, depending on configuration, as any other dedicated user. We would like to further restrict our dynamic application to have the separate daemon run as an additional, separate user. The *suEXEC* module makes this possible for us. It is not absolutely required. But it adds a bit more security with little extra effort.

Let's enter the directory with the Apache source code and compile the server once more.

```
$> cd /usr/src/apache/httpd-2.4.17
$> ./configure --prefix=/opt/apache-2.4.17 --enable-mods-shared=all \
   --with-apr=/usr/local/apr/bin/apr-1-config \
   --with-apr-util=/usr/local/apr/bin/apu-1-config --enable-mpms-shared="event worker" \
   --enable-nonportable-atomics=yes --enable-suexec --with-suexec-caller=www-data \
   --with-suexec-docroot=/opt/apache-2.4.17/bin && make && sudo make install
```

Besides *configure*, which we are familiar with, three options have been added for handling *suexec*. *Enable-suexec* is self-explanatory, *with-suexec-caller* we tell the conscientious module that only the user *www-data* is to be given permission to access the program behind the module. We are after all telling the module where scripts being called are located. For simplicity's sake let's use the existing *bin directory*. *suexec* is however a bit fussy and we are unable to use the symlink. So it will have to be the fully qualified path.

If successfully configured, the command line above will start the compiler and after being successfully concluded will install the newly compiled server.

By the way, if you make a mistake in the configuration above and want to assign *suexec* different compiler constants, you'll first have to clean out the compiler environment before compiling again. The new options will otherwise be ignored. We can do this via the *make clean* command in front of *configure* or by manually deleting the files *support/suexec*, *support/suexec.lo* and *support/suexec.o*, which is faster, because the whole web server no longer has to be built from scratch.

Step 3: Downloading and compiling the FastCGI module

The *FastCGI module* is managed by Apache. But it is not part of the normal source code for the web server. So, let's download the source code for the additional module and verify the loaded checksum over an encrypted connection.

```
$> cd /usr/src/apache
$> wget http://www.apache.org/dist/httpd/mod_fcgid/mod_fcgid-2.3.9.tar.gz
$> wget https://www.apache.org/dist/httpd/mod_fcgid/mod_fcgid-2.3.9.tar.gz.sha1
$> sha1sum --check mod_fcgid-2.3.9.tar.gz.sha1
```

We again expect an *OK*. When it is correctly returned, it's time for unpacking, compiling and installing.

```
$> tar xvzf mod_fcgid-2.3.9.tar.gz
$> cd mod_fcgid-2.3.9
$> APXS=/apache/bin/apxs ./configure.apxs
$> make
$> sudo make install
```

The *configure* command has a slightly different format here, because the *FCGI module* is a module dependent on Apache. That's why we are using *APXS*, the *Apache Expansion Tool*, part of the server compiled in Tutorial 1. Unfortunately, *make install* will partly destroy some of the ownerships we have set. They will have to be adjusted afterwards.

```
$> sudo chown `whoami` /apache/conf/httpd.conf
```

The Apache user also needs access to a directory for him to create sockets, enabling communication with the FCGI daemon. We'll create the directory and transfer ownership to it directly.

```
$> sudo mkdir /apache/logs/fcgidsock
$> sudo chown www-data:www-data /apache/logs/fcgidsock
```

Step 4: Installing and preconfiguring PHP

Up till now we have been compiling all of the software piece by piece. But for the entire PHP stack a limit has been reached. No one should be prevented from compiling PHP on his own, but we are concentrating on the web server here and for this reason will be using this piece of software from the Linux distribution. In Debian/Ubuntu the package we need is *php5-cgi*, which comes along with *php5-common*.

Properly configuring *PHP* is a broad topic and I recommend consulting the relevant pages, because an improperly configured installation of *PHP* may pose a serious security problem. I don't wish to give you any more information at this point, because it takes us away from our actual topic, which is a simple application server. For operation on the internet, i.e. no longer in a lab-like setup, it is highly recommended to become familiar with the relevant PHP security settings.

Step 5: Creating the CGI user

Above it has already been discussed that we are planning to start a separate daemon to process *PHP requests*. This daemon should be started via *suexec* and run as separate user. We create the user as follows:

```
$> sudo groupadd fcgi-php
$> sudo useradd -s /bin/false -d /apache/htdocs -m -g fcgi-php fcgi-php
```

It can be expected for a warning to appear concerning the presence of the */apache/htdocs* directory. We can ignore it.

Step 6: Creating a PHP wrapper script

It is normal to use a wrapper script for *PHP* and *FCGI* to work together. We already set this up in the Apache configuration above. The web server will invoke only the script, while the script takes care of everything else. We place the script as intended in */apache/bin/php-fcgi-starter/php-fcgi-starter*. The subdirectory isn't needed, because *suexec* requires the directory to be owned by the preconfigured user and we don't want to assign *./bin* completely to the new user. So, let's create the subdirectory:

```
$> cd /apache/bin
$> sudo mkdir php-fcgi-starter
$> sudo chown fcgi-php:fcgi-php php-fcgi-starter
```

We now have to put a starter script in this directory. Since we have already assigned *fcgi-php* to the user, the root user will have to create the script. Or be copied by him to this location. Creating a script in a directory that we no longer own is challenging. We'll solve this with a trick using *cat* and a *subshell*. Here's the trick, followed by the script. (Input in *cat* is ended via CTRL-D).

```
$> sudo sh -c "cat > php-fcgi-starter/php-fcgi-starter"
#!/bin/sh
export PHPRC=/etc/php5/cgi/
export PHP_FCGI_MAX_REQUESTS=5000
export PHP_FCGI_CHILDREN=5
exec /usr/lib/cgi-bin/php
```

What are we defining here? We are telling *PHP* where to find its configuration, setting the maximum number of requests for an *FCGI* to 5,000 (after which it is replaced by a fresh process), defining the number of process children to 5 and executing PHP at the end.

Now don't forget to assign the script to the *FCGI user* and make it executable:

```
$> sudo chown fcgi-php:fcgi-php php-fcgi-starter/php-fcgi-starter
$> sudo chmod +x php-fcgi-starter/php-fcgi-starter
```

Step 7: Creating a PHP test page

To finish things off, we should now create a simple *PHP-based* test page: */apache/htdocs/info.php*.

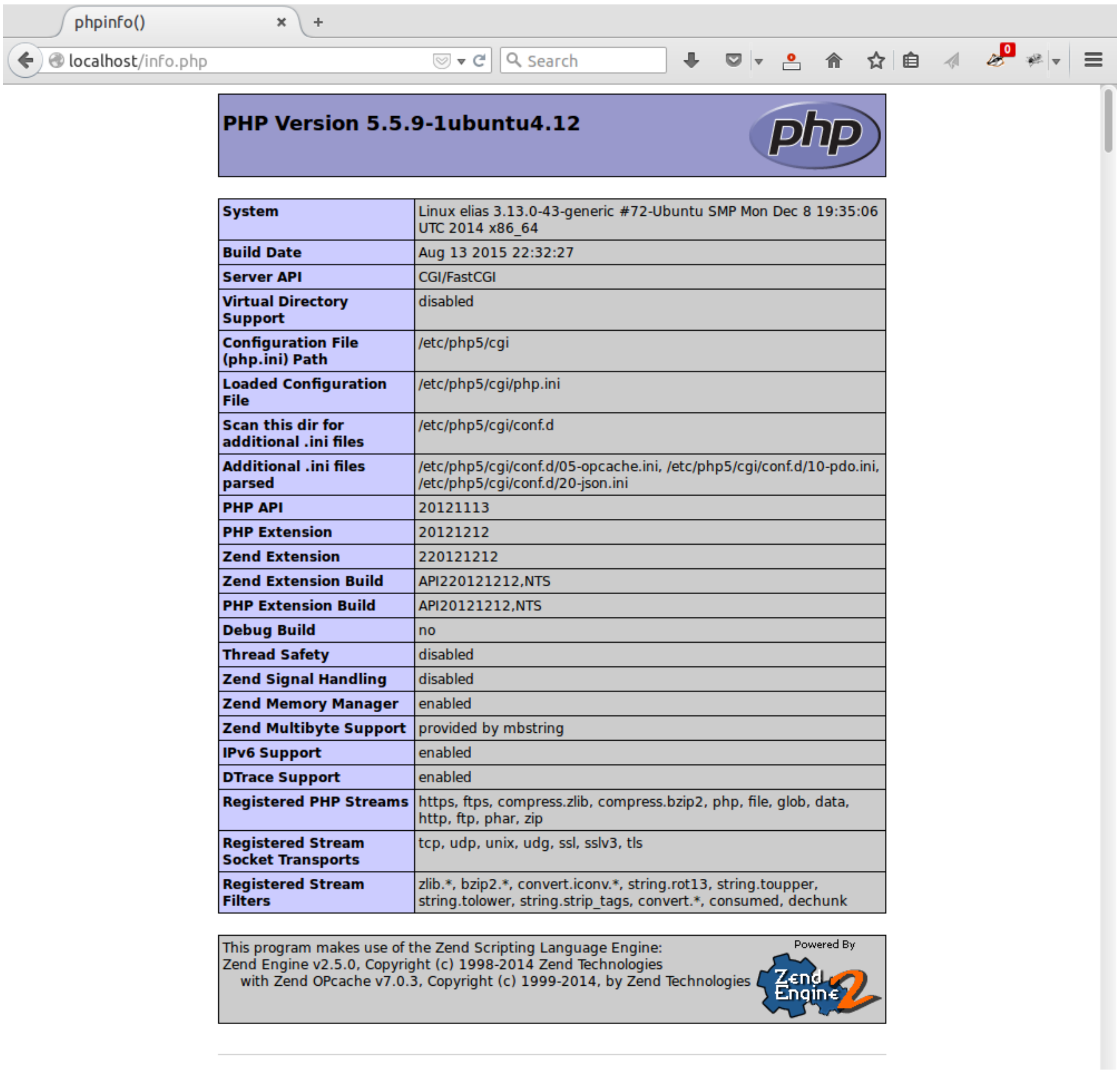
```
<?php
phpinfo();
?>
```

Step 8: Trying it out

That was everything. We can now start our web server and try it out.

```
$> cd /apache
$> sudo ./bin/httpd -X -f conf/httpd.conf_fastcgid
```

Our test script is available at http://localhost/info.php.



PHP Version 5.5.9-1ubuntu4.12

System	Linux elias 3.13.0-43-generic #72-Ubuntu SMP Mon Dec 8 19:35:06 UTC 2014 x86_64
Build Date	Aug 13 2015 22:32:27
Server API	CGI/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/cgi
Loaded Configuration File	/etc/php5/cgi/php.ini
Scan this dir for additional .ini files	/etc/php5/cgi/conf.d
Additional .ini files parsed	/etc/php5/cgi/conf.d/05-opcache.ini, /etc/php5/cgi/conf.d/10-pdo.ini, /etc/php5/cgi/conf.d/20-json.ini
PHP API	20121113
PHP Extension	20121212
Zend Extension	220121212
Zend Extension Build	API220121212,NTS
PHP Extension Build	API20121212,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring
IPv6 Support	enabled
DTrace Support	enabled
Registered PHP Streams	https, ftps, compress.zlib, compress.bzip2, php, file, glob, data, http, ftp, phar, zip
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, sslv3, tls
Registered Stream Filters	zlib.*, bzip2.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk

This program makes use of the Zend Scripting Language Engine:
Zend Engine v2.5.0, Copyright (c) 1998-2014 Zend Technologies
with Zend OPcache v7.0.3, Copyright (c) 1999-2014, by Zend Technologies

Powered By

phpinfo gives you a comprehensive status report in your browser.

If an error message results from starting the server or invoking the URL, the server's *error log* or the separate *suexec log* in *logs/suexec_log* may prove helpful. Errors are typically related to ownership of and permission to access directories and files.

Here again is a summary of the relevant files and their owners:

```
2107985     4 drwxr-xr-x    2 fcgi-php fcgi-php      4096 Jul   2 11:15 bin/php-fcgi-starter/
2107987     4 -rwxr-xr-x    1 fcgi-php fcgi-php       125 Jul   2 11:15 bin/php-fcgi-starter/php-fcgi-starter
2107977    32 -rwsr-xr-x    1 root     root         32146 Jul   2 10:44 bin/suexec
2633547     4 drwxr-xr-x    2 myuser   root          4096 Jul   2 11:16 htdocs/
2633758     4 -rw-r--r--    1 myuser   myuser          20 Jul   2 11:15 htdocs/info.php
2762281     4 drwxr-xr-x    2 www-data www-data      4096 Jul   2 10:46 /apache/logs/fcgidsock/
```

Of note is the *suid bit* on the *suexec binary*.

Step 9 (Bonus): A little performance test

Compared to Apache with integrated *PHP*, the application server built here is very high performance. A little performance test can illustrate this. We start our web server in *daemon mode* and use ApacheBench to put it to the test it with 5 users. The *-l* option is new. It instructs the tool to ignore deviations in the length of the responses. This is because the page is generated dynamically and the content, as well as its length, will of course be a bit different now and then. We again quit the server following the performance test.

```
$> sudo ./bin/httpd -k start -f conf/httpd.conf_fastcgid
$> ./bin/ab -c 5 -n 1000 -l http://localhost/info.php
```

```
...
$> sudo ./bin/httpd -k stop -f conf/httpd.conf_fastcgid
```

In most cases *ab* has the following output:

```
This is ApacheBench, Version 2.3 <$Revision: 1663405 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:        Apache
Server Hostname:        localhost
Server Port:            80

Document Path:          /info.php
Document Length:        Variable

Concurrency Level:      5
Time taken for tests:   2.567 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      66892443 bytes
HTML transferred:       66739443 bytes
Requests per second:    389.63 [#/sec] (mean)
Time per request:       12.833 [ms] (mean)
Time per request:       2.567 [ms] (mean, across all concurrent requests)
Transfer rate:          25452.57 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       1
Processing:     4   13  70.9      7    1147
Waiting:        2   12  70.8      6    1143
Total:          4   13  70.9      7    1147

Percentage of the requests served within a certain time (ms)
  50%      7
  66%      9
  75%     10
  80%     11
  90%     14
  95%     17
  98%     24
  99%     28
 100%   1147 (longest request)
```

That‘s 389 dynamic requests per second. Which is a lot. Especially considering that the result is from a small test computer. On a modern production-size server performance many times this can be achieved.

What’s remarkable isn’t the speed of the system, it’s memory usage. Unlike an application server with integrated *PHP module*, we have separated the *PHP stack* here. This gives us an Apache web server able to use *Event MPM*. In an integrated setup we would have to use the *Prefork MPM*, which works with memory-intensive server processes and not server threads. And each of these processes would then have to load the *PHP module*, regardless of the fact that most requests are normally attributed to static application components such as images, *CSS*, *JavaScripts*, etc. On my test system each *Prefork Apache process*, including *PHP*, brings a *resident size* of 6 MB. An event process with only *4 MB* means a substantial difference. And outside of this, the number of external *FCGI processes* will remain significantly smaller.

References

- Apache: http://httpd.apache.org
- Apache FCGI: http://httpd.apache.org/mod_fcgid
- How2Forge PHP/FCGI: http://www.howtoforge.com/how-to-set-up-apache2-with-mod_fcgid-and-php5-on-ubuntu-10.04

License / Copying / Further use