Tutorial 2 - Configuring a minimal Apache server

What are we doing?

We are configuring a minimal Apache web server and will occasionally be talking to it with curl, the TRACE method and ab.

Why are we doing this?

A secure server is one that permits only as much as what is really needed. Ideally, you would build a server based on a minimal system by enabling additional features individually. This is also preferable in terms of understanding what's going on, because this is the only way of knowing what is really configured. Starting with a minimal system is also helpful in debugging. If the error is still present in the minimal system, features are added individually and the search for the error goes on. When the error occurs, it can then be narrowed down to the last configuration directive added.

Requirements

- An Apache web server, ideally one created using the file structure shown in Tutorial 1 (Compiling an Apache web server).

Step 1: Creating a minimal configuration

Our web server is stored in `/apache` on the file system. It's default configuration is located in `/apache/conf/httpd.conf`. It is very extensive and rather difficult to understand. It's a problem that even the default configurations in common Linux distributions are contributing to at an ever higher degree. We will be replacing this configuration file with the following greatly simplified configuration.

```
ServerName              localhost
ServerAdmin             root@localhost
ServerRoot              /apache
User                    www-data
Group                   www-data
PidFile                 logs/httpd.pid

ServerTokens            Prod
UseCanonicalName        On
TraceEnable             Off

Timeout                 10
MaxRequestWorkers       100

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

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 None
        AllowOverride None

      </Directory>

</VirtualHost>
```

Step 2: Understanding the configuration

Let's go through this configuration step-by-step.

We are defining *ServerName* as *localhost*, because we are still working in a lab-like setup. In production the fully qualified host name of the service has to be entered here. Or the URL for short.

The server requires an administrator e-mail address, primarily for display on error pages. This is defined in *ServerAdmin*.

The *ServerRoot* directory indicates the main or root directory of the server. It is a symbolic link set as a trick in Tutorial 1. We'll make good use of this, because by reassigning this symbolic link we will be able to test a series of different compiled versions of Apache without having to change anything in the configuration file.

We then assign the user and group to *User* and *Group*. This is a good idea, because we want to prevent the server from running as a root process. In fact, the master or parent process will be running as root, but the actual server or child processes and their threads will be running under the name defined here. The *www-data* user is the name normally used on Debian/Ubuntu systems. Other distributions use different names. Make sure that the user name and associated group you choose are actually present on the system.

The *PidFile* specifies the file that Apache writes its process ID number to. The path selected is the default. It is mentioned here so you don't have to look for this path in the documentation later on.

ServerTokens defines how the server identifies itself. Productive tokens are defined using *Prod*. This means that the server identifies itself only as *Apache* without the version number and loaded modules which is a bit more discreet. Let's not fool ourselves: The version of the server can be easily determined over the internet, but we still don't have to send it along as part of the sender for every communication.

UseCanonicalName tells the server which *host names* and *ports* to use when it has to write a link to itself. The *On* value defines that the *ServerName* is to be used. One alternative would be to use the host header sent by the client, which however we don't want in our setup.

The *TraceEnable* directive prevents certain types of spying attacks on our setup. The HTTP *TRACE* method instructs the web server to return the requests it receives 1:1. This enables us to determine whether a proxy server is interposed and whether it has modified the request. This is no loss in our simple setup, but this information is better kept confidential on a corporate network. So, for the sake of security we are turning *TraceEnable* off by default.

Broadly speaking, *Timeout* indicates the maximum time in seconds that may be used to process a request. In reality it is a bit more complicated, but we don't need to worry about the details at the moment. The default value of 60 seconds is very high. We'll lower it to 10 seconds.

MaxRequestWorkers is the maximum number of threads working in parallel to reply to requests. The default value is once again a bit high. Let's set it to 100. If this value is reached in production, then we have a lot of traffic.

By default, the Apache server listens to any available URL on the internet. However, for our tests we will have it initially listen to only the *IPv4 local host* URL and on default HTTP port 80. It's possible to have multiple *Listen* directives one after the other, but only one is sufficient for our purposes at the moment.

Let’s now load five modules:

- mpm_event_module : “event” process model
- unixd_module : access to Unix user names and groups
- log_config_module : freely defined access log
- authn_core_module : core module for authentication
- authz_core_module : core module for authorization

We already compiled all of the modules supplied by Apache in Tutorial 1. We will now be adding the most important ones to our configuration. *mpm_event_module* and *unixd_module* are needed to operate the server. When compiling in the first tutorial we chose the *event* process model, which we will now be enabling by loading the module. Of interest: In Apache 2.4 such a fundamental setting as the process model of the server can be set in the configuration. We need the *unixd* module to run the server (as described above) under the user name we defined.

log_config_module enables us to freely define the access log, which we will be making use of shortly. And finally, there are the two *authn_core_module* and *authz_core_module* modules. The first part of the name indicates authentication (*authn*) and authorization (*authz*). Core then means that these two modules are the basis for these functions.

In terms of access security, we often hear about *AAA*, short for *authentication*, *authorization* and *access control*. Authentication means checking the identity of the user. Authorization means you define the access permissions for a user that has been authenticated. Finally, access control means the decision as to whether access is granted to an authenticated user with the access permissions defined for him. We lay the foundation for this mechanism by loading these two modules. There are a lot of other modules with the *authn* and *authz* prefixes which require these modules. For the moment we actually only need the authorization module, but by loading the authentication module we are preparing for extensions later on.

We use *ErrorLogFormat* to change the format of the error log file. We will be extending the customary log format a bit by precisely defining the time stamp. `[%{cu}t]` will then produce entries such as `[2015-09-24 06:34:29.199635]`. This means the date written backwards, then the time with a precision of microseconds. Writing the date backwards makes it more easily sortable in the log file; the microseconds provide precise information as to the time of the entry and enable conclusions to be made about how long processing takes in the different modules. This is also the purpose of the next configuration part, `[%-m:%-l]`, that specifies the module doing the logging and the *log level*, i.e. the severity of the error. After this come the IP address of the client (`%-a`), a unique identifier for the request (`%-L`) (a unique ID which can be used in later tutorials in correlation to requests) and the actual message, which we reference using `%M`.

We use *LogFormat* to define a format for the access log file. We give it the name *combined*. This common format includes the client IP address, time stamp, methods, path, HTTP version, HTTP status code, response length, referer and the name of the browser (User-Agent). For the timestamp we are selecting a structure that is quite complicated. The reason for this is the desire to use the same format for timestamps as in the error log and access logs. While easy identification is available in the error log, we have to painstakingly put together the timestamp for the access log format.

By using *debug* we are setting the *LogLevel* for the error log file to the highest level. This is too chatty for production, but it makes perfect sense in a lab-like setting. Apache is not very chatty in general so the volume of data is usually easy enough to handle.

We assign the error log file by adding the path *logs/error.log* to *ErrorLog*. This path is relative to the *ServerRoot* directory.

We now use *LogFormat combined* for our access log file called *logs/access.log*.

The web server delivers files. It searches for them on a disk partition or generates them with help from an installed application. We are still using a simple case here and tell the server to look for the files in *DocumentRoot*. */apache/htdocs* is an absolute path below *ServerRoot*. A relative path could be entered, but it's best to make things clear here! Specifically, *DocumentRoot* means that the URL path */* is being mapped to the */apache/htdocs* operating system path.

Now comes a *directory* block. We use this block to prevent files from being delivered outside the *DocumentRoot* we defined. We forbid any access to the / path using the *Require all denied* directive. This entry refers to the authentication (*all*), makes a statement about authorization (*Require*) and defines access: *denied*, i.e. no access for anyone, at least to the */* directory.

We set the *Options* directive to *SymLinksIfOwnerMatch*. We can use *Options* to define the special features to take into account when sending the / directory. Actually, none at all and that’s why in production we would write Options *None*. But in our case we have set *DocumentRoot* to a symbolic link and it can only be searched for and found if we assign *SymLinksIfOwnerMatch* to the server, also allowing symlinks below /. At least if the permissions are clear. For security reasons, on production systems it is best to not to rely on symlinks when serving files. But convenience still takes precedence on our test system.

AllowOverride tells the server to ignore *.htaccess* files, since we are not planning to use them. These files are mainly of interest to web hosters and shared hosting services. This doesn't really apply to us.

Let's now open up a *VirtualHost*. It corresponds to the *Listen* directive defined above. Together with the *Directory* block we just defined, it defines that by default our web server does not permit any access at all. However, we want to permit access to IP address *127.0.0.1, Port 80*, which is defined in this block.

Specifically, we permit access to our *DocumentRoot*. The final instruction here is *Require all granted*, where unlike the / directory we permit full access. Unlike above, from this path on no provision is made for symlinks or any special capabilities: *Options None*, *AllowOverride None*.

Step 3: Starting the server

Our minimal server has thus been described. It would be possible to define a server that is even more bare bones. It would however not be as comfortable to work with as ours and it wouldn't be any more secure. A certain amount of basic security is however advisable. This is because in the lab we are building a service which should then with specific adjustments be able to be put into a production environment. Wanting to secure a service from top to bottom right before entering a production environment is illusory.

As we did in Tutorial 1 let's start the server in the foreground and not as a daemon:

```
$> cd /apache


$> sudo ./bin/httpd -X
```

Step 4: Talking to the server using curl

Now we can again communicate with the server from a web browser. But working in the shell at first can be more effective, making it easier to understand what is going on.

```
$> curl http://localhost/index.html
```

Returns the following:

```
<html><body><h1>It works!</h1></body></html>
```

We have thus sent an HTTP request and have received a response from our minimally configured server, meeting our expectations.

Step 5: Examining requests and responses

This is what happens during an HTTP request. But what exactly is the server saying to us? To find out, let's start *curl*. This time with the *verbose* option.

```
$> curl --verbose http://localhost/index.html
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET /index.html HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Thu, 24 Sep 2015 09:27:02 GMT
* Server Apache is not blacklisted
< Server: Apache
< Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
< ETag: "2d-432a5e4a73a80"
< Accept-Ranges: bytes
< Content-Length: 45
<
```

```
<html><body><h1>It works!</h1></body></html>
* Connection #0 to host localhost left intact
```

The lines marked with a asterisk (*) describe messages concerning opening and closing the connection. They do not reflect network traffic. The request follows > and the response <.

Specifically, an HTTP request comprises 4 parts:

- Request line and request header
- Request body (optional and missing here for a GET request)
- Response header
- Response body

We don't have to worry about the first parts just yet. It's the *response headers* that are interesting. This is the part used by the web server to describe the response. The actual response, the *response body*, follows after an empty line.

In order, what do the headers mean?

At first comes the *status* line including the *protocol*, the version, followed by the *status code*. *200 OK* is the normal response from a web server. On the next line we see the date and time of the server. The next line begins with an asterisk, *, signifying a line belonging to *curl*. The message is related to *curl's* handling of HTTP pipelining, which we don't have to concern ourselves with. Then comes the *server* line on which our Apache web server identifies itself. This is the shortest possible identification. We have defined it using *ServerTokens* Prod.

The server will then tell us when the file the response is based on was last changed, i.e. the *Unix modified timestamp*. *ETag* and *Accept* ranges don't require our attention for the moment. What's more interesting is *Content-Length*. This specifies how many bytes to expect in the *response body*. 45 bytes in our case.

Incidentally, the order of these headers is characteristic for web servers. *NginX* uses a different order and, for instance, puts the *server header* in front of the date. Apache can still be identified even if the server line is intended to be misleading.

Step 6: Examining the response a bit more closely

During communication it is possible to get a somewhat more detailed view in *curl*. We use the *--trace-ascii* command line parameter to do this:

```
$> curl   http://localhost/index.html --trace-ascii -
== Info: Hostname was NOT found in DNS cache
== Info:   Trying 127.0.0.1...
== Info: Connected to localhost (127.0.0.1) port 80 (#0)
=> Send header, 83 bytes (0x53)
0000: GET /index.html HTTP/1.1
001a: User-Agent: curl/7.35.0
0033: Host: localhost
0044: Accept: */*
0051:
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 37 bytes (0x25)
0000: Date: Thu, 24 Sep 2015 11:46:17 GMT
== Info: Server Apache is not blacklisted
<= Recv header, 16 bytes (0x10)
0000: Server: Apache
<= Recv header, 46 bytes (0x2e)
0000: Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
<= Recv header, 26 bytes (0x1a)
0000: ETag: "2d-432a5e4a73a80"
<= Recv header, 22 bytes (0x16)
0000: Accept-Ranges: bytes
<= Recv header, 20 bytes (0x14)
0000: Content-Length: 45
<= Recv header, 2 bytes (0x2)
0000:
<= Recv data, 45 bytes (0x2d)
0000: <html><body><h1>It works!</h1></body></html>.
<html><body><h1>It works!</h1></body></html>
== Info: Connection #0 to host localhost left intact
```

--trace-ascii requires a file as a parameter in order to make an *ASCII dump* of communication in it. "-" works as a shortcut for *STDOUT*, enabling us to easily see what is being logged.

Compared to *verbose*, *trace-ascii* provides more details about the length of transferred bytes in the *request* and *response* phase. The request headers in the example above are thus 83 bytes. The bytes are then listed for each header in the response and overall for the body in the response: 45 bytes. This may seem like we are splitting hairs. But in fact, it can be crucial when something is missing and it is not quite certain what or where in the sequence it was delivered. Thus, it's worth noting that 2 bytes are added to each header line. These are the CR (carriage returns) and NL (new lines) in the header lines included in the HTTP protocol. This is unlike in the response body, which returns only what is actually in the file. This is obviously only one NL without CR here. On the third to last line (*000: <html ...*) a point comes after the greater than character This is code for the NL character in the response, which like other escape sequences is output in the form of a point.

Step 7: Working with the trace method

The *TraceEnable* directive was described above. We have turned it *off* as a precaution. It can however be very useful in debugging. So, let's give it a try. Let's set the option to on:

```
TraceEnable On
```

We restart the server and make the following curl request:

```
$> curl -v --request TRACE http://localhost/index.html
```

We are thus accessing the known *URL* using the *HTTP TRACE method* (in place of *GET*). We expect the following as the result:

```
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> TRACE /index.html HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Thu, 24 Sep 2015 09:38:01 GMT
* Server Apache is not blacklisted
< Server: Apache
< Transfer-Encoding: chunked
< Content-Type: message/http
<
TRACE /index.html HTTP/1.1
User-Agent: curl/7.35.0
Host: localhost
Accept: */*

* Connection #0 to host localhost left intact
```

In the *body* the server repeats the information about the request sent to it as intended. In fact, the lines are identical here. We are thus able to confirm that nothing has happened to the request in transit. If however we had passed through one or more interposed proxy servers, then there would be additional *header* lines that we would also be able to see as a *client*. At a later point we will become familiar with more powerful tools for debugging. Nevertheless, we don't want to completely ignore the *TRACE* method.

Don't forget to turn *TraceEnable* off again.

Step 8: Using "ab" to test the server

So much for the simple server. But just for fun we can put it to the test. We'll perform a small performance test using *ab*, short for *ApacheBench*. This is a very simple benchmarking program that is always at hand and able to quickly give you initial performance results. I like to run it before and after a configuration change to get an idea about whether anything in terms of performance has changed. *ab* is very powerful and calling it locally does not give you clean results. But you can get an initial impression using this tool.

```
$> ./bin/ab -c 1 -n 1000 http://localhost/index.html
```

We are starting ab using *concurrency 1*. The means that we are executing only one request at a time. In total, we will be executing 1,000 requests from the known *URL*. This is the output from *ab*:

```
$> ./bin/ab -c 1 -n 1000 http://localhost/index.html
This is ApacheBench, Version 2.3 <$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:          /index.html
Document Length:        45 bytes

Concurrency Level:      1
Time taken for tests:   0.676 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      250000 bytes
HTML transferred:       45000 bytes
Requests per second:    1480.14 [#/sec] (mean)
Time per request:       0.676 [ms] (mean)
Time per request:       0.676 [ms] (mean, across all concurrent requests)
Transfer rate:          361.36 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:     0    1   0.2      1       3
Waiting:        0    0   0.1      0       2
Total:          0    1   0.2      1       3

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      1
  98%      1
  99%      1
 100%      3 (longest request)
```

What's of primary interest to us is the number of errors (*Failed requests*) and the number of requests per second (*Requests per second*). A value above one thousand is a good start. Especially considering that we are still working with a single process and not a parallelized daemon (which is also why the *concurrency level* is set to 1).

Step 9: Viewing directives and modules

At the end of this tutorial we are going to be looking at a variety of directives, which an Apache web server started with our configuration file is familiar with, The different loaded modules extend the server's set of commands. The available

configuration parameters are well documented on the Apache Project's website. In fact, in special cases it can however be helpful to get an overview of the directives made available from the loaded modules. You can get the directives by using the command line flag *-L*.

```
$> ./bin/httpd -L
<Directory (core.c)
    Container for directives affecting resources located in the specified directories
    Allowed in *.conf only outside <Directory>, <Files>, <Location>, or <If>
<Location (core.c)
    Container for directives affecting resources accessed through the specified URL paths
    Allowed in *.conf only outside <Directory>, <Files>, <Location>, or <If>
<VirtualHost (core.c)
    Container to map directives to a particular virtual host, takes one or more host addresses
    Allowed in *.conf only outside <Directory>, <Files>, <Location>, or <If>
<Files (core.c)
...
```

The directives follow the order in which they are loaded. A brief description of its function comes after each directive.

Using this list it is now possible to determine whether all of the modules loaded in the configuration, referenced respectively, are actually required. In complicated configurations with a large number of loaded modules it may happen that you are unsure whether all of the modules are actually being used.

You can thus get the modules by reading the configuration file, the output of *httpd -L* summarized for each module and then look in the configuration file to see if any of the directives listed are being used. This nested manner of sending requests demands a find touch, but is one that I can highly recommend. Personally, I have solved it as follows:

```
$> grep LoadModule conf/httpd.conf | awk '{print $2}' | sed -e "s/_module//" | while read M; do \
  echo "Module $M"; R=$(./bin/httpd -L | grep $M | cut -d\  -f1 | tr -d "<" | xargs | tr " " "|"); \
  egrep -q "$R" ./conf/httpd.conf; \
  if [ $? -eq 0 ]; then echo "OK"; else echo "Not used"; fi; echo; \
  done
Module mpm_event
OK

Module unixd
OK

Module log_config
OK

Module authn_core
Not used

Module authz_core
OK
```

The *authn_core* module is thus not being used. This is correct, we described it as such above, since it is being loaded for use in the future. The rest of the modules appear to be needed.

So much for this tutorial. You now have a capable server you can work with. We will continue to extend it in subsequent tutorials.

References

- Apache: http://httpd.apache.org
- Apache directives: http://httpd.apache.org/docs/current/mod/directives.html
- HTTP headers: http://en.wikipedia.org/wiki/List_of_HTTP_header_fields
- RFC 2616 (HTTP protocol): http://www.ietf.org/rfc/rfc2616.txt

License / Copying / Further use