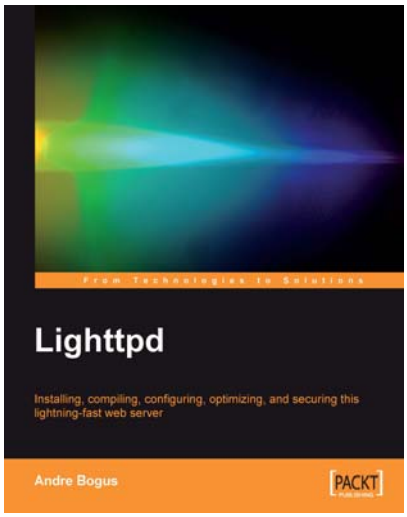
Lighttpd

Andre Bogus



Chapter No. 10
"Migration from Apache"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.10 "Migration from Apache"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Andre Bogus is a musician turned programmer. He has worked in different jobs from voice acting to programming to teaching to managing software projects. At the moment he works as a consultant and implementer for KOGIT GmbH, an Identity Management company based in Germany.

He found Lighttpd while searching for the ideal software for his personal web server and quickly learned the tricks to make it do what he wanted. He enjoys learning new things and telling others about them. When his full schedule allows it, he can be found on the #lighttpd IRC channel.

He wants to thank his wife, Ania, without whose support he would not have been able to finish this book. Also he appreciates his employer for allowing him to write besides his day job. The nice people at PACKT Publishing have also earned his gratitude by helping this book to become what it is.

Lighttpd

This book explains downloading, installing, and configuring the Lighttpd HTTP server, illustrates how to extend it with modules and Lua code, shows a migration path from Apache httpd, gives case studies in setting up a number of popular web applications, and even demonstrates how to extend Lighttpd by writing our own modules.

The name Lighttpd (pronounced "Lighty") is an abbreviation pulling together Light (as in weight) and HTTPD (which is an abbreviation for Hypertext Transport Protocol Daemon, in short web server). Early versions called themselves LightTPD to emphasize the "lightweight" part, but this led to confusion over pronunciation and meaning, so the capitalization was reduced.

What This Book Covers

Chapter 1 gives directions how to obtain Lighttpd. Regardless, if we want to use a binary package or build from source, everything is there. In addition, dependencies, optional packages, and compilation options are examined. After working through this chapter, we should have an installed Lighttpd to work with.

Chapter 2 introduces all elements of the configuration language by example. Usable examples include sending the correct MIME type, setting up multiple domains, rewriting, and redirecting. Also the command line options are explained. For those who are not fluent in regular expressions, the chapter has an excursion. At the end of this chapter, we have our Lighttpd up and running.

Chapter 3 builds on the concepts of the second chapter and discusses the configuration various CGI-like interfaces, three modules for virtual hosting, also introducing the MySQL database, which is used in one of the modules.

Chapter 4 shows how to set up Lighttpd as a download or streaming server, covering optimizations for large downloads as well as guarding our site against denial of service attacks, dealing with proxies, and restricting download speeds for anonymous clients.

Chapter 5 extends our Lighttpd to learn more about our users: Geo-tracking the location from the client IP address, dissecting the page traversal behavior ("clickstream analysis") and other data points. Also responsible access logging practices are outlined.

Chapter 6 adds SSL support to our Lighttpd and walks through the steps to acquire or create the required certificates, whether we obtain a certificate from a public or corporate certificate authority, self-sign a certificate, or become our own certificate authority.

Chapter 7 helps us securing our Lighttpd by authorizing access, limiting traffic by IP to thwart denial-of-service attacks, and measuring our success by rigorously examination of our log files. Setting up log rotate and log parsers is also covered.

Chapter 8 concerns itself with limiting the potential damage a subverted Lighttpd could do to the system. The techniques to achieve this are reducing privileges and putting the whole Lighttpd in a secluded environment. Containing Lighttpd and a CGI backend in different environments is also demonstrated.

Chapter 9 shows a strategy to optimize our Lighttpd from system and configuration settings to the source code itself. The chapter also shows specific optimizations known to yield benefits across most systems.

Chapter 10 takes a pragmatic look on the migration path from Apache httpd. It shows how to port basic configuration, rewrite and redirect rules, how to deal with .htaccess files, and even discusses when not to migrate.

Chapter 11 revisits the CGI interfaces by getting various example applications from Ruby on Rails over WordPress, phpMyAdmin, trac, and AWstats to AjaxTerm up and running with our Lighttpd.

Chapter 12 adds the small and fast scripting language Lua to the mix, which can be used to extend the functionality of Lighttpd by mod_magnet or as a backend language by the Lua/FastCGI interface written by the same author as Lighttpd. Both options are discussed, along with an introduction to the language itself.

Chapter 13 gives a run down of extending Lighttpd by extending existing modules or even writing our own. With these modules, we can change the behavior of Lighttpd from request parsing to sending or altering content. This chapter is aimed at an average C programmer.

Appendix A lists the HTTP status codes that our Lighttpd can return on answering a request, giving directions which chapter or other source might have more information on each request.

Appendix B is the module and configuration index. Each configuration option for every Lighttpd module of the official distribution is explained here in one or two short sentences. Forgotten how a configuration option is written, what type it has or what it means? Look no further.

10
Migration from Apache

The most common web server used today is still Apache, so whilst we wait for Lighttpd world domination, the migration from this server warrants its own chapter. As this is a book on Lighttpd and not on Apache, this chapter assumes some knowledge of the Apache configuration. If anything is unclear, the Apache documentation at `http://apache.org/docs/` will hopefully help.

Now starting from a working Apache installation, what can Lighttpd offer us?

- Improved performance for most cases (as in more hits per second)
- Reduced CPU time and memory usage
- Improved security (refer to Chapter 8 to maximize your return on investment)

Of course, the move to Lighttpd is not a small one, especially if our Apache configuration makes use of its many features. Systems tied into Apache as a module may make the move hard or even impossible without porting the module to a Lighttpd module or moving the functionality into CGI programs, if possible.

We can ease the pain by moving in small steps. The following descriptions assume that we have one Apache instance running on one hardware instance. But we can scale the method by repeating it for every hardware instance.

When not to migrate

Before we start this journey, we need to know that our hardware and operating systems support Lighttpd, that we have root access (or access to someone who has), and that the system has enough space for another Lighttpd installation (yes, I know, Lighttpd should *reduce* space concerns, but I have seen Apache installations munching away entire RAID arrays). Probably, this only makes sense if we plan on moving a big percentage of traffic to Lighttpd. We also might make extensive use of Apache module, which means a complete migration would involve finding or writing suitable substitutes for Lighttpd.

Adding Lighttpd to the Mix

Install Lighttpd on the system that Apache runs on. Refer to Chapter 1 for installation instructions. Find an unused port (refer to a port scanner if needed) to set `server.port` to. For example, if port 4080 is unused on our system, we would look for `server.port` in our Lighttpd configuration and change it to:

```
server.port = 4080
```

If we want to use SSL, we should change all occurrences of the port 443 to another free port, say 4443. We assume our Apache is answering requests on HTTP port 80.

Now let's use this Lighttpd instance as a proxy for our Apache by adding the following configuration:

```
server.modules = (
  #...
  "mod_proxy",
  #...
)

#...

proxy.server = (
  "" => { # proxy everything
    host => "127.0.0.1" # localhost
    port => "80"
  )
)
```

This tells our Lighttpd to proxy all requests to the server that answers on localhost, port 80, which happens to be our Apache server. Now, when we start our Lighttpd and point our browser to `http://localhost:4080/`, we should be able to see the same thing our Apache is returning.

What is a proxy?

A **Proxy** stands in front of another object, simulating the proxied object by relaying all requests to it. A proxy can change requests on the fly, filter requests, and so on. In our case, Lighttpd is the web server to the outside, whilst Apache will still get all requests as usual.

Excursion: mod_proxy

`mod_proxy` is the module that allows Lighttpd to relay requests to another web server. It is not to be confused with `mod_proxy_core` (of Lighttpd 1.5.0), which provides a basis for other interfaces such as CGI. Usually, we want to proxy only a specific subset of requests, for example, we might want to proxy requests for Java server pages to a Tomcat server. This could be done with the following proxy directive:

```
proxy.server = (
  ".jsp" => ( host => "127.0.0.1", port => "8080" )
                              # given our tomcat is on port 8080
)
```

Thus the tomcat server only serves JSPs, which is what it was built to do, whilst our Lighttpd does the rest.

Or we might have another server which we want to include in our Web presence at some given directory:

```
proxy.server = (
  "/somepath" => ( host => "127.0.0.1", port => "8080" )
)
```

Assuming the server is on port 8080, this will do the trick. Now `http://localhost/somepath/index.html` will be the same as http://localhost:8080/index.html.

Reducing Apache Load

Note that as most Lighttpd directives, `proxy.server` can be moved into a selector (refer to Chapter 2), thereby reducing its reach. This way, we can reduce the set of files Apache will have to touch in a phased manner. For example, YouTube™ uses Lighttpd to serve the videos. Usually, we want to make Lighttpd serve static files such as images, CSS, and JavaScript, leaving Apache to serve the dynamically generated pages.

Now, we have two options: we can either filter the extensions we want Apache to handle, or we can filter the addresses we want Lighttpd to serve without asking Apache.

Actually, the first can be done in two ways. Assuming we want to give all addresses ending with `.cgi` and `.php` to Apache, we could either use the matching of `proxy.server`:

```
proxy.server = (
  ".cgi" => ( host = "127.0.0.1", port = "8080" ),
  ".php" => ( host = "127.0.0.1", port = "8080" )
)
```

or match by selector:

```
$HTTP['url'] =~ "(.cgi|.php)$" {
  proxy.server = ( "" => ( host = "127.0.0.1", port = "8080" ) )
}
```

The second way also allows negative filtering and filtering by `regexp`—just use `!~` instead of `=~`.

mod_perl, mod_php, and mod_python

There are no Lighttpd modules to embed scripting languages into Lighttpd (with the exception of `mod_magnet`, which embeds Lua) because this is simply not the Lighttpd way of doing things. Instead, we have the CGI, SCGI, and FastCGI interfaces (refer to Chapter 7) to outsource this work to the respective interpreters. In the next chapter, there will be sample installations and configurations for some popular applications.

Most `mod_perl` scripts are easily converted to FastCGI using `CGI::Fast`. Usually, our `mod_perl` script will look a lot like the following script:

```
use CGI;
my $q = CGI->new;
initialize(); # this might need to be done only once
process_query($q); # this should be done per request
print response($q); # this, too
```

Using the easiest way to convert to FastCGI:

```
use CGI:Fast # instead of CGI
while (my $q = CGI:Fast->new) { # get requests in a while-loop
  initialize();
  process_query($q);
  print response($q);
}
```

If this runs, we may try to put the `initialize()` call outside of the loop to make our script run even faster than under `mod_perl`. However, this is just the basic case. There are `mod_perl` scripts that manipulate the Apache core or use special hooks, so these scripts can get a little more complicated to migrate.

Migrating from `mod_php` to `php-fcgi` is easier—we do not need to change the scripts, just the configuration. This means that we do not get the benefits of an obvious request loop, but we can work around that by setting some global variables only if they are not already set. The security benefit is obvious. Even for Apache, there are some alternatives to `mod_php`, which try to provide more security, often with bad performance implications.

`mod_python` can be a little more complicated, because Apache calls out to the python functions directly, converting form fields to function arguments on the fly. If we are lucky, our python scripts could implement the **WSGI** (**Web Server Gateway Interface**). In this case, we can just use a WSGI-FastCGI wrapper. Looking on the Web, I already found two: one standalone (`http://svn.saddi.com/py-lib/trunk/fcgi.py`), and one, a part of the PEAK project (`http://peak.telecommunity.com/DevCenter/FrontPage`). Otherwise, python usually has excellent support for SCGI.

As with `mod_perl`, there are some internals that have to be moved into the configuration (for example dynamic 404 pages, the directive for this is `server.error-handler-405`, which can also point to a CGI script). However, for basic scripts, we can use SCGI (either from `http://www.mems-exchange.org/software/scgi/` or as a python-only version from `http://www.cherokee-project.com/download/pyscgi/`). We also need to change `import cgi` to `import scgi` and change `CGIHandler` and `CGIServer` to `SCGIHandler` and `SCGIServer`, respectively.

.htaccess

A lot of Lighttpd users converting from Apache ask if Lighttpd has any substitutes for `.htaccess` files, which were made popular by Apache and are now a de-facto Standard used by many web servers. Instead, Lighttpd has its own configuration syntax, so all the old `.htaccess` files won't work with Lighttpd.

There is no perfect solution to this problem, but as the most used feature of `.htaccess` files is authentication, we can at least solve that. Let's have a look at the authentication directive format in Apache and Lighttpd:

- Apache just assumes that the path required for authentication is the path where the `.htaccess` file resides, while Lighttpd needs to add this explicitly.
- The `httpd.conf` adds some more stuff, which is given as default from `httpd.conf`. In the `lighttpd.conf` example, we do not assume such defaults.

Note that the Lighttpd configuration gets a little more complicated if we have multiple backends or user files. In this case, we need to use a selector to limit the reach of our directives. For example, assume that we want digest authentication for `internal.mydomain.com`, but `htpasswd` authentication for some directories in mydomain.com, with a different `htpasswd` file for the messages directory:

```
server.modules = (..., "mod_auth", ...)

auth.backend = "htpasswd"
auth.backend.htpasswd.userfile = "/web/general/.htpasswd"

$HTTP["host"] == "internal.mydomain.com" {
  auth.backend = "htdigest"
  auth.backend.htdigest.userfile = "/web/internal/.htdigest"
  auth.require = (
    "/" => (
      "method" => "digest",
      "realm" => "internal",
      "require" => "valid-user"
    )
  )
}
else
$HTTP["url"] =~ "^/messages" {
  auth.backend.htpasswd.userfile = "/web/messages/.htpasswd"
  auth.require = (
    "/" => (
        "method" => "basic",
        "realm" => "messages",
        "require" => "valid-user"
    )
  )
}

auth.require = ( # This table assigns authentication requirements
                 # to directories or file types.
  "/admin/" => ( # everything below the /admin path
    "method" => "basic",
    "realm" => "admin",
    "require" => "user=andre|user=bob" # allow only bob and me
  ),
  "/download" => (
    "method" => "basic",
    "realm" => "download",
    "require" => "valid-user"
  ),
```

```
    ".private" => ( # all files ending with .private
      "method" => "basic",
      "realm" => "private",
      "require" => "user=andre"
    )
    # ... we could add more directories here.
  )
```

The first selector marks out a region `internal.mydomain.com`, where we then use digest authentication. The next selector catches the message directory everywhere else and includes the use of the `/web/messages/.htpasswd` user file. Finally, we add all the requirements for the other directories.

Note that the following two are identical:

```
$HTTP["url"] =~ "^/messages" {
auth.require = ( "/" => ( ... ) )
}
```

```
auth.require = ( "/messages" => ( ... ) )
```

But the left version is more flexible as it allows defining different user files and backends for each path that matches a certain pattern. Armed with this knowledge, we can write a small script that runs through our web root, finds all `.htaccess` files and emits corresponding Lighttpd configuration (at least for the access directives). In fact we do not even need to do this, because I already did the coding:

```
#!/bin/env python
import os
def toUserList(users):
  return "|".join(["user="+user for user in users.split(" ")])
  def groups(groupFileName, gps):
  groupFile = open(groupFileName)
  groupDict = {}
  for groupLine in groupFile:group, users = groupLine.split(":")
      groupDict[group.strip()] = users.strip()
      return "|".join([toUserList(groupDict[g])
                       for g in gps.split(" ")])
for (root, dirs, files) in os.walk(path):
  if ".htaccess" not in files: continue
  filepath = os.path.join(root, ".htaccess")
  f = open(filepath)
  try:
    realm = root.rsplit(os.path.sep, 1)[1]
  except:
    realm = root
  try:
```

```
        # try some sensible defaults
        r = {"authtype":"Basic", "url":root,
             "required":"nothing","realm":realm,
             "authuserfile":os.path.join(root, ".htpasswd",
             "error":None}
        for line in f:
          try:
            tempdirective, arguments = line.split(" ", 1)
            directive = tempdirective.lower()
              r[directive] = arguments.strip('"')
          except:
             pass
          if r["required"].startswith("user"):
            r["required"] = toUserList(r["required"][5:])
          elif r["required"].startswith("group"):
            r["required"] = groups(r["authgroupfile"], r["required"][6:])
          if r["required"] != "nothing" and r["error"] is None:
             r["backend"] = {"Basic":"htpasswd",
                   "Digest":"htdigest"}[r["authtype"]]
            r["authtype"] = r["authtype"].lower()
          print """$HTTP["url"] =~ "%{url}s" {
      auth.backend = "${backend}s"
      auth.backend.${backend}s.userfile = "${authuserfile}s"
      auth.require = ( "/" => (
        "method" => "${authtype}s",
        "realm" => "${realm}s",
        "require" => "${required}s"
      ) )
    }""" % r;
        finally:
          f.close()
```

The `htaccess2lighttpd.py` script is available at `http://www.packtpub.com/files/code/2103_Code.zip`.

Note the script does have one limitation: Lighttpd does not handle groups. However, it allows specification of a list of users directly, as in `user=andre|user=bob` that we required for admin access. The other way is to have a separate password file for each group. The script, however, takes the first way. This means that we need to re-run the script each time a group membership changes. So this solution would only be temporary—the move to per-group access files can then be made without being hectic.

.htaccess and PHP

Apart from that, some users might put PHP options into the `.htaccess` files. **Pier Alan Joye** maintains a `htscanner` program to ease the transition. It is available at `http://pecl.php.net/package/htscanner`. This program basically moves PHP options from `.htaccess` files into the `php.ini` file.

Rewriting Rules

On the Lighttpd forums, most former Apache administrators ask how they can adapt their rewrite rules to work with Lighttpd. There is no program (yet) to do this, but here are some typical constructs and advice on how to do that in Lighttpd lingua:

Apache	Lighttpd
LoadModule "rewrite_module" RewriteEngine on	server.modules = (..., "mod_rewrite", "mod_redirect", ...)

# A simple rewrite 	
RewriteRule ^from_here(.*)/to_there$1
	# refer to Chapter 2
	url.rewrite = ("^/from_here" => "to_there")

RewriteCond %{HTTP_HOST} me\..*
RewriteRule ^/(.*) /me/$1
	$HTTP["host"] =~ "me\..*" {
	url.rewrite = ("^/" => "/me/"
	}

# Redirecting a single file RewriteRule move.html target.html [R]	url.redirect = ("move.html" => "target.html")
# Solving the trailing slash problem RewriteCond %{REQUEST_FILENAME} -d RewriteRule (.*) $1/	# nothing to do here. Lighttpd does not # have this problem.
# Redirecting failed web pages to xyz.com RewriteCond %{REQUEST_FILENAME} !-f RewriteRule ^(.+) `http://xyz.com/$1`	# use an CGI error page that redirects server-error-handler-404 = "redirect.cgi" # see Chapter 12 on how to do this in lua
# Time-based multiplexing RewriteCond %{TIME_HOUR} > 07 RewriteCond %{TIME_HOUR} < 19 RewriteRule ^foo.html foo.day.html RewriteRule ^foo.html foo.night.html	# either use mod_magnet, see Chapter 12, or solve this outside of Lighttpd, for example by using a cron job to set symbolic links.

Apache	Lighttpd

# Rewrite for google bot 	
RewriteCond %{HTTP_USER_AGENT} \
Google
RewriteRule ^(.+) /bots/$1
	# match for useragent
	$HTTP["useragent"] =~ "Google" {
	url.rewrite = "^/" => "/bots"
	}

# Rewrite by cookie (missing session) 	
RewriteCond %{HTTP_COOKIE} sess [N]
RewriteRule ^(.+) index.php
	# use a negative regexp match
	$HTTP["cookie"] !~ "sess" {
	url.rewrite = ("(.*)" => "index.pho")
	}

# set environment variable based on query 	
RewriteCond %{QUERY_STRING} \
id=([^&]*)
RewriteRule ^(.*)$ /$1 [E=ID:%1]
	server.modules += ("mod_setenv")
	$HTTP["url"] =~ "[?&]id=([^&]*)" {
	setenv.add_request_header = "ID: %1"
	}

# block images by referer 	
RewriteCond %{REFERER} !^$
RewriteCond %{REFERER} !my\.net [NC]
RewriteRule ^images/*.png - [F]
	# deny for non-empty outside referers
	$HTTP["referer"] !~ "^($|.*my\.net) {
	url.access-deny = (".png")
	}

Naturally this table cannot cover all aspects of Apache rewrite rules, but remember that all complex systems have emerged from simple systems. The following chapter will show how to set up some oft-used web applications with Lighttpd.

WebDAV

Apache does WebDAV out of the box, while Lighttpd needs the `mod_webdav` module to support WebDAV, and it still has some rough edges. For example, Window users will find that their `mod_auth` login does not work when they activate WebDAV; this can be compensated by a cookie-based login. Oh, and we need to have webdav support configured at compile time, if we built our Lighttpd from source. The configuration, as always, is pretty straightforward:

```
server.modules += ( "mod_webdav" )

  # activate WebDAV for the server "dav.my.net"
  $HTTP["host"] == "dav.my.net" {
  webdav.activate = "enable"
```

```
    # enable writing for members only (identify by sess cookie)
    $HTTP["cookie"] !~ "sess" {
      $HTTP["url"] =~ "^/members/" {
        webdav.is-readonly = "enable"
      }
    }
  }
```

The important directives here are `webdav.activate` and `webdav.is-readonly`. The former activates WebDAV, if we set it to `enable`. Otherwise, WebDAV is deactivated by default. The latter forbids operations that modify files on the server (PUT and DELETE), and is disabled by default. So unless we enable this option, PUT and DELETES are served.

Summary

There are some obstacles on the way from Apache to Lighttpd. But a planned and careful approach will allow us to keep our server working while we change it. The `.htaccess` scanner script can be a stop gap measure to smoothen the transition for `.htaccess` authentication users. Finally, a heavy use of rewrite rules can make it tricky to move. However, we can translate them one by one into something that will work with Lighttpd, especially when we add Lua to the mix as we will show in the following chapter.

Where to buy this book

You can buy Lighttpd from the Packt Publishing website:
`http://www.packtpub.com/lighttpd/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com