Nginx Troubleshooting

Nginx is clearly winning the race to be the dominant software powering modern websites. It is fast and open source, supported with passion by a brilliant team. This book will help you maintain your Nginx instances in a healthy and predictable state.

You will learn how to write great configuration files and get insights from Nginx logs. It will provide you with solutions to problems such as missing or broken functionality and also show you how to tackle performance issues with the Nginx server. A special chapter is devoted to the art of prevention, that is, monitoring and alerting services you may use to detect problems before they manifest themselves on a big scale. The book ends with a reference of some error and warning messages Nginx may emit to help you during incident investigations.

Who this book is written for

The book is for technical specialists who already use Nginx to serve web pages for their users. Whether you are an experienced system administrator or a new professional, this book will help you do your job in the most efficient way.

What you will learn from this book

- Detect and investigate problems with your web servers
- Create correct and professional configuration files
- Get an in-depth understanding of the web server software
- Interpret Nginx log files easily and with actionable insights
- Make your websites respond as quickly as possible
- Prevent problems by setting up proper monitoring and alerts

Alex Kapranoff

Investigate and solve problems with Nginx-powered websites using a deep understanding of the underlying principles
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Searching for Problems in Nginx Configuration'
- A synopsis of the book’s content
- More information on Nginx Troubleshooting
Alex Kapranoff is an established IT professional with more than 20 years of experience in roles ranging from a database developer to the CTO of a social search engine start-up to an infrastructure manager of a huge web application with hundreds of millions of users. His background allows him to be more on the developer side of things, and he is happy to share some of that perspective with fellow system administration specialists.
You will learn how to notes problems before your boss calls you about some pages not loading. You will learn how to find those problems using logs and your usual Linux toolbox. You will also learn how to minimize the probability of problems happening again.

Nginx started as a web accelerator reverse proxy inside one of the big Russian web companies of the early 2000s. The main web server software was Apache 1.3, and it started to show architectural problems serving thousands of relatively slow clients using the old process-based model. Smart web engineers were already building two-tier systems of light frontends based on the mod_proxy Apache module or even used the squid caching proxy in the reverse proxy mode.

The early predecessor of Nginx was named mod_accel, and it was also implemented as an Apache module. The mod_accel module gained some popularity among the administrators of some of the busiest websites, but it is nothing compared with what Nginx later enjoyed. Both of them are built on the idea that the additional level of proxying on the server side of a busy website is a good thing, providing both the extra flexibility and separating the job of serving slow clients from the actual response generation.

Nginx took the idea of mod_proxy module to the extreme by being a self-sufficient separate HTTP server with a goal to solve the so-called C10K problem, that is, serving 10,000 concurrent connections. The numbers do not look impressive at all in 2016, but they did in 2007 when Nginx first claimed a significant share of 1% of the Web according to Netcraft.

Since that time, the share grew manifold while Nginx steadily gained new functionality and remained the ideal open source success story project with a single, talented developer devoting his genius to producing free quality software, which the whole Web could benefit from.
In 2011, a commercial enterprise named Nginx, Inc. was founded, which allowed even more freedom for the developers (now a team). The firm provides both support services and a special subscription-based extended version of the software named Nginx Plus. We will mention some of the Nginx Plus features in the sixth chapter.

In 2016, Nginx is a great tool many businesses are built upon. However, it is still just a tool which requires a master to show its full potential. If you want to understand what is going on in your web server, to be able to write correct Nginx configuration files and read Nginx logs, and if you want your web server to be very fast, you will have to become that master.

What this book covers

Chapter 1, Searching for Problems in Nginx Configuration, briefly describes the configuration language of Nginx and presents some of the corner cases and several techniques to search for problems.

Chapter 2, Searching for Problems in Log Files, describes the logging subsystem, log syntax, and what to look for when you troubleshoot. Nginx provides thorough logs of everything it does.

Chapter 3, Troubleshooting Functionality, the central chapter in this book, contains a list of steps you will make while investigating a problem. You will find the types of problems people generally encounter with their Nginx-powered web servers.

Chapter 4, Optimizing Website Performance, is dedicated to all things about performance. Starting with thorough explanation of the basic principles behind the Nginx event-driven processing model, it also touches on caching and even gives some advice on possible upstream optimization.

Chapter 5, Troubleshooting Rare Specific Problems, is devoted to studies of several real cases that you may face, from some of the simplest and easiest cases to fix to more problematic cases. The cases described might not be the most frequent, but they still provide valuable insight on the internals of the software and the methods of troubleshooting.

Chapter 6, Monitoring Nginx, is devoted to the abundance of tools available today that you may use for monitoring. No system is complete without good processes for the detection of emerging problems.
Chapter 7, Going Forward with Nginx, the short final chapter, provides a selection of directions you may choose for your further development as a specialist. The whole industry is very dynamic, and you should never settle.

Appendix, Rare Nginx Error Messages, provides a reference of interesting and not very common error messages that you might encounter in your log files.
Nginx is a complex piece of software that will help you implement your own part of the World Wide Web—one of the killer applications of the Internet as a whole. Although seeming to be simple, the Web and the underlying HTTP have a lot of intricate details that may require special attention. Nginx gives you the power to pay that attention to those details by means of the extensive configuration language. Following the grand UNIX tradition of human-readable and writable text configuration files, Nginx expects a certain level of understanding and zeal from you so that it can serve in the best way possible. It also means that there is freedom and huge potential for mistakes.

The main goal of this chapter is to lead you through the way Nginx is configured and show you some of the areas that are prone to errors.

You will find further:

- Configuration syntax with description and examples
- Description of all files in the default configuration bundled with Nginx
- Some mistakes you could make with examples from the default configuration and techniques to avoid them
Introducing basic configuration syntax, directives, and testing

Igor Sysoev, the principal author of Nginx, said, on several occasions, that he designed the Nginx configuration language in such a way that writing the configuration should not feel like programming, or actual coding. For a long time, he himself worked as a professional system administrator for several relatively big websites in Russia. He understood perfectly that the goal of a website administrator is not to end up with beautiful, elegant configurations or to have at one's disposal every imaginable function for all possible situations no matter how rare they are. The goal is to be able to declaratively describe the business requirements, to formulate which behavior is needed without delving into how that could be achieved in software. One interesting example of quite the opposite idea in language design is the Lighttpd configuration language, but that's out of the scope of this book.

This is what we have now—a simple declarative language inspired by Apache's one but without all the XML-like tags. Open the default nginx.conf file to see what Nginx configuration looks like. Some distributions contain their own modifications to the default file. We will use the one from the original tarball available at http://nginx.org/download/nginx-1.9.12.tar.gz. What follows is a quick syntax introduction using parts of that file as examples. You might find it too obvious but bear with us; even the most experienced reader will do good to refresh his or her memory.

Let us look at the very beginning of the file. Lines starting with # are comments, and they are ignored. Commenting out is a very common technique to make Nginx ignore a part of configuration. The topmost line in default Nginx configuration file (as of version 1.9.12) is actually commented out:

```plaintext
...  
#user  nobody;  
...  
```

One easy way to comment out a block of lines in vim is highlighting them visually with Shift-V and then issuing the :s/^/#/ ex command. In Emacs, just select a region and then press M-;.

Nonempty noncommented lines in Nginx configuration are of the two following types.
Chapter 1

Simple directives
Simple directives consist of a command word followed by a number of parameters and a semicolon. For example (see at the top of the default nginx.conf file):

```plaintext
... 
  worker_processes  1;
... 
```

Nothing to worry about here. People having too much experience with modern scripting languages, such as Python and Ruby, tend to forget the semicolon; we advise you to make sure that you add it.

The parameters mentioned here can be either constant values (numbers or strings, which does not matter, they are all parsed in the same way at this level) or they may contain variables. Variables in Nginx are the $dollar_prefixed identifiers that are replaced with some actual value at runtime. For example, there are variables containing data from an HTTP request, and you can modify website behavior depending on their values or just log them.

A very good example of variables in the default nginx.conf file is this:

```plaintext
... 
  #log_format  main  '$remote_addr - $remote_user [$time_local]
                 "$request"
                 '$status $body_bytes_sent "$http_referer"
                 "$http_user_agent" "$http_x_forwarded_for"';
... 
```

This directive creates a log format by constructing a template for each line of the log. It uses a number of variables available during the request/response cycle.

Multiline directives
Multiline directives are simple directives with a BUT. Instead of a semicolon in the end, there is a block enclosed in braces `{ ... }`. And here instead is meant literally. You don't put semicolons after closing braces. Those of you with just enough experience with more traditional C-like syntax programming languages will find this very natural.

Here is an example of the very first multiline directive in the default Nginx 1.9.12 nginx.conf file:

```plaintext
  events { 
    worker_connections  1024; 
  }
```
Now, this is an `events` directive, which does not have any parameters, and it contains a block instead of a semicolon. Because of these blocks, multiline directives are also named "block directives". Blocks contain various kinds of content, but one of the most important and interesting blocks is the one containing other directives — both simple and multiline.

In the previous example, the block of the `events` directive contains a simple `worker_connections` directive.

Multiline directives that allow other directives inside their blocks are named "contexts". They introduce new context for the enclosed, inner part of the configuration.

Most of the multiline directives are actually contexts — from the most popular, such as `server` or `location`, to the most obscure, such as `limit_except`. An example of a multiline directive that is not a context is `types`, which introduces the relation between file extensions and the so-called Multipurpose Internet Mail Extensions (MIME) types. We will look at `types` later in this chapter.

Contexts are very important. They are scopes and topics of the directives that are inside. If a command is not included in any multiline directive block, then it is considered part of the special context named "main" with the widest scope. Directives in this context affect the whole Nginx application. Other contexts are all either inside "main" or even deeper below, and the commands that are contained within those contexts have narrower scopes and affect only parts of the whole.

**Include directive**

We will not describe actual directives here except for one of them. It is the `include` directive, a special dear to the hearts of all sysadmins who scale their work to many websites, servers, or just URLs. It is a very simple block-level "package management tool" if we are allowed to use more programming terminology. This simple directive has one parameter, that is, a filename or a wild card (UNIX glob-style) matching a number of files. During processing, this directive is replaced by the contents of the files it refers to. A quick example (from the default `nginx.conf` file):

```
  ...  
  include fastcgi_params;  
  ...  
```

We won't offend you by spending more time on explaining `include`. What we need to add is that included files have to be fully correct syntactically. You cannot have half of a command in one file and then include the rest from another.
So, this is it, the whole syntax is described. Let us show you a fictional piece of configuration that demonstrates everything but does not actually work because it contains nonexistent directives (or maybe those are from some future version of Nginx):

```nginx
... simple_command 4 "two";
# another_simple_command 0;

special_context {
    some_special_command /new/path;
    multilineDirective param {
        1 2 3 5 8 13;
    }
    include common_parameters;
}
... 
```

Testing Nginx configuration

There is a very handy tool in the Nginx kit, a syntax checker for the configuration files. It is built into the main Nginx executable application and invoked by using the `-t` command-line switch as follows:

```bash
% nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
...
```

The command `nginx -t` tries to check your configuration quite thoroughly. For example, it will check all the included files and try to access all the auxiliary files like logs or pids to warn you about their nonexistence or insufficient permissions. You will become a better Nginx administrator if you acquire a habit of frequently running `nginx -t`.

The default configuration directory layout

We will now run through the entire configuration that you get bundled with Nginx by default. Some of it is a good example from which you will start writing your own. Some of it is just a sign of Nginx age. Again, we use the original tarball for the 1.9.12 version that is available on the official Nginx website.
This is a list of files inside the conf folder of the Nginx source archive:

```bash
% ls
fastcgi.conf koi-utf mime.types scgi_params win-utf
fastcgi_params koi-win nginx.conf uwsgi_params
```

The nginx.conf is the main file, the one everything starts with. All other files are either included from nginx.conf or not used at all. Actually, nginx.conf is the only configuration file that is required by Nginx code (and you can override even that by using -c command-line switch). We will discuss its content a little bit later.

A pair of fastcgi.conf and fastcgi_params files contains almost the same list of simple commands configuring the Nginx FastCGI client. FastCGI, being an interface to run web applications behind Nginx, is not turned on by default. These two files are provided as examples (one of them is even included with the include command from a commented section of the nginx.conf file).

Three files with enigmatic names koi-utf, koi-win, and win-utf are character maps to convert between different ways to encode Cyrillic characters in electronic documents. And Cyrillic is, of course, the script used for Russian and several other languages. In the old days of the first Internet hosts in Russia, there was a dispute on which way to encode Russian letters in documents. You can read about different Cyrillic charsets/encodings at [http://czyborra.com/charsets/cyrillic.html](http://czyborra.com/charsets/cyrillic.html). Several of them got popular, and web servers had to include functionality of converting documents on the fly in the case that a client browser requested a different encoding from what was used by the server. There was also a whole fork of Apache Web Server that had this functionality built in. Nginx had to do the same to stand a chance against Apache. And now, more than 10 years later, we still have these re-encoding files that are deeply obsolete as the global World Wide Web continues to move towards UTF-8 as the one universal encoding for all human languages. You won't ever use these koi-utf, koi-win, and win-utf files unless you support a very old website for Russian-speaking visitors.

The file named mime.types is used by default. You can see that it is included from the main nginx.conf, and you better leave it that way. "MIME types" is a registry of different types of information in files.
They have their origin in some of the email standards (hence, the MIME name) but are used everywhere, including the Web. Let's look inside `mime.types`:

```plaintext
... types {
text/html html htm shtml;
text/css css;
text/xml xml;
image/gif gif;
... }
```

Because it is included from `nginx.conf`, it should have a proper Nginx configuration language syntax. That's right, it contains a single multiline directive `types`, which is not a context (as described in the previous section). Its block is a list of pairs, each being a mapping from one MIME type to a list of file extensions. This mapping is used to mark static files served by Nginx as having a particular MIME (or content) type. According to the quoted segment, the files `common.css` and `new.css` will get the type `text/css`, whereas `index.shtml` will be `text/html`, and so on and so forth; it is really easy.

**A quick example of modifying the MIME types registry**

Sometimes, you will add things to this registry. Let's try to do this now and demonstrate an introduction of a simple mistake and the workflow to find and fix it.

Your website will host calendars for your colleagues. A calendar is a file in the iCalendar format generated by a third-party application and saved to a file with `.ics` extension. There is nothing about `.ics` in the default `mime.types`, and because of this, your Nginx instance will serve these files with the default `application/octet-stream` MIME type, which basically means "it is a bunch of octets (bytes) and I don't have the faintest idea of what they mean". Suppose that the new calendar application your colleagues use require proper iCalendar-typed HTTP responses. This means that you have to add this `text/calendar` type into your `mime.types` file.

You open `mime.types` in your editor and add this line to the very end (not in the middle, not to the start, but the end is important for the sake of this experiment) of the file:

```plaintext
... text/calendar ics ...
```
Searching for Problems in Nginx Configuration

You then run `nginx -t` because you are a good Nginx administrator:

```
...  
nginx: [emerg] unexpected end of file, expecting ";" or "}" in  
/etc/nginx/mime.types:91  
nginx: configuration file /etc/nginx/nginx.conf test failed
...
Bam. Nginx is smart enough to tell you what you need to fix; this line does not look  
like either a simple or a multiline directive. Let's add the semicolon:

```
...  
text/calendar ics;
...

```
...  
nginx: [emerg] unknown directive "text/calendar" in  
/etc/nginx/mime.types:90  
nginx: configuration file /etc/nginx/nginx.conf test failed
...

Now this is more obscure. What you should do here is understand that this line is  
not a separate standalone directive. It is a part of the big `types multiline (the rare,  
non-context one)` directive; therefore, it should be moved into the block.

Change the tail of the `mime.types` from this:

```
}

text/calendar ics;

```

The preceding code should look as follows:

```

text/calendar ics;
}

```

It is done by swapping the last two meaningful lines:

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok  
nginx: configuration file /etc/nginx/nginx.conf test is successful

```

Congratulations, you just enabled a new business process for your company  
involving mobile workforce.
Two last default configuration files are `scgi_params` and `uwsgi_params`. Those two are the counterparts for the `fastcgi_params`, setting up two alternative methods of running web application on your web servers (SCGI and UWSGI, respectively, as you guessed). You will use them if and when your application developers will bring you applications written with these interfaces in mind.

**Default nginx.conf**

Now, let's dig deeper into the main configuration file `nginx.conf`. In its default form that you see inside the tarball, it is rather empty and useless. At the same time, it is always what you use as a starting point when writing your own configuration, and it can also be used as a demonstration of some common troubles that people inflict on themselves. Going over each directive is not needed, so only those that are good to demonstrate a technique or a common place of errors will be included in this section:

```plaintext
... #user nobody;
...
```

This directive specifies the name of the UNIX user that Nginx processes will run as. Commenting out pieces of configuration is a common documentation technique. It shows the default values and removing the comment character is safe. Nginx will complain if you try to run it as a nonexistent user. As a general rule, you should either trust your package vendor and not change the default or use an account with the least permissions possible.

```plaintext
... #error_log  logs/error.log;
#error_log  logs/error.log  notice;
#error_log  logs/error.log  info;
#pid        logs/nginx.pid;
...
```

These lines specify some default filenames. The three `error_log` directives are an example of yet another technique: providing several variants as comments so that you can uncomment the one you prefer. These three differ by the level of detail that goes into the error log. There is a whole chapter about logs as those are definitely the first and foremost debugging and troubleshooting tool available for any Nginx administrator.

The `pid` directive allows you to change the filename where pid of the main Nginx process will be stored. You rarely have to change this.
Searching for Problems in Nginx Configuration

Note that these directives use relative paths in these examples, but this is not required. They could also use absolute paths (starting with `/`). The `error_log` directive provides two other ways of logging besides simple files, which you will see later.

```conf
... 
  events { 
    worker_connections  1024; 
  } 
... 
```

This is our first context and a confusing one. `events` is not used to narrow the scope of directives inside it. Most of those directives cannot be used in any other context except `events`. This is used as a logical grouping mechanism for many parameters that configure the way Nginx responds to activity on the network. These are very general words, but they fit the purpose. Think of `events` as a fancy historical way of marking a group of parameters that are close to one another.

The `worker_connections` directive specifies the maximum number of all network connections each worker process will have. It may be a source of strange mistakes. You should remember that this limit includes both the client connections between Nginx and your user's browsers and the server connections that Nginx will have to open for your backend web application code (unless you only serve static files).

**The `http` directive**

```conf
... 
  http { 
    include       mime.types; 
    default_type  application/octet-stream; 
  } 
... 
```

Here we go, `http` marks the start of a huge context that usually spans several files (via nested includes) and groups all the configuration parameters that concern the web part of Nginx. You might feel that this sounds a lot like `events`, but it is actually a very valid context requiring a separate directive because Nginx can work not only as an HTTP server but also serve some other protocols, for example, IMAP and POP3. It is an infrequent use case, to put it mildly, and we won't spend our time on it, but it shows a very legitimate reason to have a special scope for all HTTP options.
You probably know what the first two directives inside `http` do. Never change the default MIME type. Many web clients use this particular type as an indication of a file that needs to be saved on the client computer as an opaque blob of data, and it is a good idea for all the unknown files.

```conf
... #log_format main '$remote_addr - $remote_user [$time_local] "$request" ' #                  '$status $body_bytes_sent "$http_referer" ' #                  '"$http_user_agent" "$http_x_forwarded_for"'; #access_log logs/access.log main;
...
```

These two directives specify logging of all requests, both successful and unsuccessful, for the reason of tracing and statistics. The first directive creates a log format and the second initiates logging to a specific file according to the mentioned format. It is a very powerful mechanism that gets special attention later in this book. Then we have the following code:

```conf
... sendfile on; #tcp_nopush on;
    #keepalive_timeout 0;
    keepalive_timeout 65;
    #gzip on;
...
```

The first and the second of these turn on certain networking features of the HTTP support. `sendfile` is a syscall that allows copying of bytes from a file to a socket by the OS kernel itself, sometimes using "zero copy" semantics. It is always safe to leave it on unless you have very little memory—there were reports that sometimes `sendfile` boxes may work unreliably on servers with little memory. `tcp_nopush` is an option that makes sense only in the presence of `sendfile on`. It allows you to optimize a number of network packets that a `sendfile` file gets sent in. `keepalive` is a feature of modern HTTP—the browser (on any other client) may choose not to close a connection to a server right away but to keep it open just in case there will be a need to talk to the same server again very soon. For many modern web pages, consisting of hundreds of objects, this could help a lot, especially on HTTPS, where the cost of opening a new connection is higher. `keepalive` timeout is a period in seconds that Nginx will not drop open connections to clients. Tweaking the default value of 75 will rarely affect performance. You can try if you know something special about your clients, but usually people either leave the default timeout or turn the `keepalive` off altogether by setting the timeout to zero.
There are a (big) number of compression algorithms much better than the LZW of the traditional gzip, but gzip is most widely available among servers and clients on the web, providing good enough compression for texts with very little cost. gzip on will turn on automatic compression of data on the fly between Nginx and its clients, that is, those which announce support for gzipped server responses, of course. There are still browsers in the wild that do not support gzip properly. See the description of the gzip_disable directive in the Nginx documentation at http://nginx.org/en/docs/http/ngx_http_gzip_module.html#gzip_disable. It might be a source of problems, but only if you have some really odd users either with weird special-case client software or from the past.

Now we have another multiline context directive inside http. It is a famous server directive that configures a single web server object with a hostname and a TCP port to listen on. Those two are the top-most directives inside this server. The first, listen has a much more complex syntax than just a port number, and we will not describe it here. The second one has a simple syntax, but some complex rules of matching that are also better described in the online documentation. It will be sufficient to say that these two provide a way of choosing the right server to process an incoming HTTP request. The most useful is the server_name in its simplest form; it just contains a hostname in the form of DNS domain and it will be matched against the name that browser sent in the Host: header which, in turn, is just the host name part of the URL.

This is a way to indicate the charset (encoding) of the documents you serve to the browsers. It is set by default to the special value off and not the good old koi8-r from RFC1489. Nowadays, your best bet is specifying utf8 here or just leaving it off. If you specify a charset that does not correspond to the actual charset of your documents, you will get troubles.
Here is an interesting example of using a directive inside a narrowing context. Remember that we already discussed `access_log` one level higher, inside the `http` directive. This one will turn on logging of requests to this particular server only. It is a good habit to include the name of the server in the name of its access log. So, replace `host` with something very similar to your `server_name`.

... location / {
    root html;
    index index.html index.htm;
}
...

Again, we see a multiline directive introducing a context for a number of URLs on this particular server. `location /` will match all the requests unless there is a more specific location on the same level. The rules to choose the correct location block to process an incoming request are quite complex, but simple cases could be described with simple words.

The `index` directive specifies the way to process URLs that correspond to a local folder. In this case, Nginx seeks the first existing file from the list in this directive. Serving either an `index.html` or `index.htm` for such URLs is a very old convention; you shouldn't break it unless you know what you are doing.

By the way, `index.htm` without the last `l` is an artifact of the old Microsoft filesystems that allowed three or less characters in the filename extension. Nginx never worked on Microsoft systems with such limitations, but files ending in `htm` instead of `html` still linger around.

... #error_page 404 /404.html;

# redirect server error pages to the static page /50x.html
#
error_page  500 502 503 504 /50x.html;
location = /50x.html {
    root html;
}
...

Searching for Problems in Nginx Configuration

These directives set up the way errors are reported to the user. You, as the webmaster, will most certainly rely on your logs but just in case something happened, your users should not be left in the dark. The `error_page` directive installs a handler for an HTTP error based on the famous HTTP status codes. The first example (commented) tells Nginx that in case it encounters a 404 (not found) error, it should not report it to the user as a real 404 error but instead initiate the subrequest to the `/404.html` location, render the results, and present them in the response to the original user request.

By the way, one of the most embarrassing mistakes you could make with Apache web server is to provide a 404 handler that raises another 404 error. Remember these?

Nginx will not show this type of detail to users, but they will still see some very ugly error messages:

The `location = /50x.html` looks suspiciously similar to the one we discussed earlier. The only important difference is the `=` character that means "exact match". The whole matching algorithm is a complete topic in itself, but here you should definitely remember that `=` means "process requests for this and only this URL, do not treat it as a prefix that could match longer URLs".

```bash
... # proxy the PHP scripts to Apache listening on 127.0.0.1:80
```
This is a big commented chunk of options all about the same – processing PHP scripts using two different strategies. Nginx, as you know, does not try to be everything, and it especially tries to never be an application server. The first location directive sets up proxying to another local PHP server, probably Apache with mod_php.

Pay attention to the ~ character in location. It turns on regular expressions engine for the matching of the URLs, hence the escaped . and the $ in the end. Nginx regular expressions use the common syntax originating from the first grep and ed programs written in the late 1960s. They are implemented with the PCRE library. See the PCRE documentation for the comprehensive description of the language at http://www.pcre.org/original/doc/html/pcrepattern.html.

The second block talks to a FastCGI server running locally on the 9000 port instead of HTTP proxying. It is a bit more modern way of running PHP, but it also requires a lot of parameters (see included file) as compared with the very simple and humble HTTP.

# deny access to .htaccess files, if Apache's document root
# concurs with Nginx's one
The last part of the server block under discussion introduces **Access Control Lists (ACLs)**, in a location with a regular expression. The note in the comment is a curious one. There is a tradition of "bolting" Nginx onto an existing Apache installation so that Nginx would serve all the static files itself while proxying more complex, dynamic URLs to the downstream Apache. This kind of setup is definitely not recommended, but you have probably seen or even inherited one. Nginx itself does not support the local .htaccess files but has to protect those files left from Apache because they could contain sensitive information.

And the final server multiline directive is an example of a secure server serving HTTPS:

```nginx
# HTTPS server
#
#server {
#   listen    443 ssl;
#   server_name localhost;
#
#   ssl_certificate      cert.pem;
#   ssl_certificate_key  cert.key;
#
#   ssl_session_cache    shared:SSL:1m;
#   ssl_session_timeout  5m;
#
#   ssl_ciphers  HIGH:!aNULL:!MD5;
#   ssl_prefer_server_ciphers  on;
#
#   location / {
#     root   html;
#     index  index.html index.htm;
#   }
#}
```

Besides a bunch of simple `ssl_` directives in the middle, the important thing to note is `listen 443 ssl`, which enables HTTPS (basically, HTTPS is HTTP over SSL on the TCP port 443). We talk about HTTPS in Chapter 3, *Troubleshooting Functionality* of this book.
Common mistakes in configuration

Reading default configuration files may turn out interesting and educating, but more useful thing is, of course, looking at examples of configuration that is actually used in production. We will now look at some common mistakes that happen during the configuration of Nginx.

If you don't see something that has happened to you and you need help immediately, by all means skip and browse the rest of the book. There are a lot more examples throughout the chapters grouped by the nature of the problem or the effects it has.

Semicolons and newlines

One common feature of truly great software is forgiving. Nginx will understand and autocorrect some syntax violations when the result is unambiguous. For example, if your hands insist on enclosing values in quotes—you can actually do this.

This is completely legal and works okay:

```plaintext
...
  user "nobody" 'www-data';
  worker_processes '2';
...
```

On the other hand, here is a case when Nginx will not allow you to leave a stray, unneeded semicolon although it does not introduce any ambiguity:

```plaintext
...
  events {
    worker_connections 768;
    # multi_accept on;
  }
...
```

% sudo nginx -t

nginx: [emerg] unexpected ";" in /etc/nginx/nginx.conf:13
nginx: configuration file /etc/nginx/nginx.conf test failed

The author once had a configuration file saved in the older Mac format, that is, with <CR> as the newline separator. This is a format used on pre-OS X Apple operating systems. Text editors and pagers work around this rare curiosity, and you will have a hard time noticing anything unusual. Nginx could not split the file into lines at all:

% sudo nginx -t
Searching for Problems in Nginx Configuration

nginx: [emerg] unexpected end of file, expecting "}" in
/etc/nginx/nginx.conf:1
nginx: configuration file /etc/nginx/nginx.conf test failed

The way to fix it is to convert newlines from <CR> to <LF> or to <CR><LF>. The easiest method, using tr from the Unix/Linux command line, looks like this:

% tr '\r' '\n' < /etc/nginx/nginx.conf > /tmp/nginx.conf

(After this, check it manually and replace the old file with mv.)

File permissions

Have you noticed that we run nginx -t with sudo? Let us try without sudo and see what happens:

```
    kantara:~[0]# nginx -t
    nginx: [alert] could not open error log file: open() "'/var/log/nginx/error.log" failed (13: Permission denied)
    2015/01/31 17:11:34 [warn] 9104#0: the "user" directive makes sense only if the master process runs with super-user privileges, ignored in /etc/nginx/nginx.conf:1
    nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
    2015/01/31 17:11:34 [warn] 9104#0: open() "/run/nginx.pid" failed (13: Permission denied)
    nginx: configuration file /etc/nginx/nginx.conf test failed
    kantara:~[1]#
```

It is actually quite interesting. Nginx reported that the syntax of the file is okay, but then it decided to dig deeper and check not only the syntax but also the availability of some features mentioned in the configuration. First, it complained about not being able to change the effective user under whose permissions all the worker processes should run. Do you remember the user directive? It also tried to open both the main server-wide error log and the pid file that is rewritten on each restart of Nginx. Both of them are not available for writing from the main working account (and they should not be, of course). That is why sudo is needed when running nginx -t.
Variables

Here is another example of a simple syntax error that might bite you once or twice in your career. Do you remember variables that we discussed several pages before? Nginx uses $syntax that is very familiar to everyone with the UNIX shell, awk, Perl, or PHP programming experience. Still, it is very easy to miss the dollar character and Nginx will not notice that because a variable will just turn into a simple string constant.

When you set up your Nginx as a proxy for another web server (such configuration is traditionally named "reverse accelerator", but less and less often so in recent times), you will quickly find that all client connections to your backend server come from the same IP address, the address of your Nginx host. The reason is obvious, but once you have some backend logic depending on getting the actual client address, you will need to work around this limitation of proxying. A common practice is to include an additional HTTP request header on all requests from Nginx to the backend. Here is how you do that:

...  
proxy_set_header X-Real-IP $remote_addr;
...  

The application will have to check for this header, and only in its absence use the actual client IP address from the socket. Now, imagine losing that dollar sign in the beginning of $remote_addr. Suddenly, your Nginx will add a very strange header of X-Real-IP: remote_addr to all requests. Nginx -t won't be able to detect this. Your backend application might blow up in case there is a strict IP address parser or, and this is ironically worse, it might skip the unparsable IP address of remote_addr and default to the actual address of your Nginx never ever reporting this to any logs. You will end up with a working configuration that silently loses valuable information! Depending on luck, this could be in production for months before someone notices that some fresh "rate-limiting by IP" feature of the application starts to affect all users at once!

Ah, the horrors!

Regular expressions

Let us get to something less destructive. Many Nginx directives make use of regular expressions. You should be familiar with them. If not, we would recommend stopping your work as soon as possible and leaving for a bookstore. Regular expressions are considered by many IT practitioners to be the single most important technology or skill for everyday use after fast typing.
Searching for Problems in Nginx Configuration

Most often, you will see regexps in location multiline directive. Besides this, they are very useful (and sometimes unavoidable) in URL rewriting and hostname matching. Regular expressions are a mini-language that uses several characters as metacharacters to construct patterns from strings. Those patterns cover sets of strings (very often infinite sets); the process of checking whether a particular string is contained in the set corresponding to a pattern is named matching. This is a simple regexp from the default nginx.conf file:

```bash
...#location ~ \.php$ {
    #    proxy_pass   http://127.0.0.1;
#}
... 
```

The tilde after the location command word means that what follows is a regular expression to match against incoming URLs. `\.php$` covers an infinite set of all strings in the universe that have these exact four characters in the very end: `.php`. The backslash before the dot cancels the metavalue of the dot, which is "any character". The dollar sign is a metacharacter that matches the very end of a string.

How many ways are there to make a mistake in that expression? A lot. A very big number. Will `nginx -t` point out those errors? Most probably, no, unless you happen to make the whole directive somehow invalid and due to the very expressive nature of the mini-language, almost all character combinations are valid. Let's try some:

```bash
...#location ~ \.php$ {
    #    proxy_pass   http://127.0.0.1;
#}
... 
```

Did you notice? Right, no dollar, again as in the variable example shown previously. This is perfectly valid. It will also pass most tests because this regexp covers an even larger infinite set of all strings that have `.php` as a substring, not necessarily in the end. What could possibly go wrong? Well, first, you could get a request for the URL that looks like "/mirrors/www.phpworld.example.com/index.html" and blow up. And second, matching by comparing the last 4 characters is logically much simpler than searching the whole buffer for the substring. This could have performance effects, however, small.

Let's skip the backslash instead:

```bash
...#location ~ .php$ {
... 
```
Evil. This will also pass the tests but again, the set of matching strings grew. Now the dot before the `php` is not literal. It is a metadot meaning any character. You have to be lucky to get something like `/download/version-for-php`, but once you get this, the location will match. You have a time bomb.

Now, let's drop the tilde:

```nginx
... location \.php$ {
...
```

Do you like our game by the way? You should already predict what will happen and how to fix it, that is, if you do like it and are starting to think like an Nginx instance.

The missing tilde will turn this location directive into its simplest form—no regular expressions whatsoever. The string `\.php$` is interpreted as a prefix to search for in all incoming URLs, together with the backslash and the dollar. Will this location block ever process a single request? We don't know. One important thing here is that `nginx -t` still does not have anything to say about this directive. It is still valid syntactically.

**Summary**

In this chapter, you refreshed your knowledge of how exactly Nginx is configured. We showed how the language looks and what some common pitfalls are when writing it. Some of you learned a thing or two about the mysterious files that Nginx authors included in the default `conf` folder; some will never ever miss a semicolon again. Run `nginx -t` often, but never blindly rely on it saying everything is okay.

The next chapter is devoted to reading and configuring logging mechanisms inside Nginx.
Where to buy this book

You can buy Nginx Troubleshooting from the Packt Publishing website.

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

Click here for ordering and shipping details.