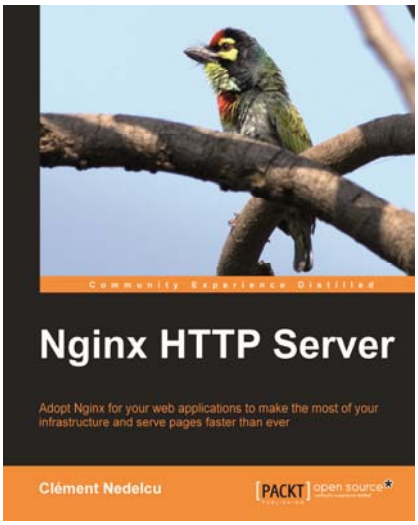
[PACKT] open source
community experience distilled
PUBLISHING

Nginx HTTP Server

Clément Nedelcu



Chapter No.3
"Basic Nginx Configuration"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Basic Nginx Configuration"

A synopsis of the book's content

Information on where to buy this book

About the Author

Clément Nedelcu was born and raised in France, and studied in U.K., French, and Chinese universities. He is now a computer science teacher at Jiangsu University of Science and Technology in Zhenjiang, a southwestern city of China. He also works as technology consultant in France, specialized in web and Microsoft .NET development as well as Linux server administration. Since 2005, he has been administering a major network of websites in his spare time. This eventually led him to discover Nginx: it made such a difference that he started his own blog about it. One thing leading to another…

The author's blog can be visited at `http://cnedelcu.net` and contains articles about Nginx and other web development topics.

I would like to express my gratitude to my girlfriend, my family and my friends who have been very supportive all along the writing stage. This book is dedicated to Martin Fjordvald for originally directing me to Nginx when my servers were about to kick the bucket. Special thanks to Maxim Dounin, Jérémie Bertrand, Shaun James, Zhang Yichun, Brendan, and all the folks on the #Nginx IRC channel on Freenode.

Nginx HTTP Server

It is a well-known fact that the market of web servers has a long-established leader: Apache. According to recent surveys, as of October 2009 over 45 percent of the World Wide Web is served by this fifteen years old open source application. However, for the past few months the same reports reveal the rise of a new competitor: Nginx, a lightweight HTTP server originating from Russia— pronounced "engine X". There have been many interrogations surrounding the pronounced newborn. Why has the blogosphere become so effervescent about it? What is the reason causing so many server administrators to switch to Nginx since the beginning of year 2009? Is this apparently tiny piece of software mature enough to run my high-traffic website?

To begin with, Nginx is not as young as one might think. Originally started in 2002, the project was first carried out by a standalone developer, Igor Sysoev, for the needs of an extremely high-traffic Russian website, namely Rambler, which received as of September 2008 over 500 million HTTP requests per day. The application is now used to serve some of the most popular websites on the Web such as WordPress, Hulu, SourceForge, and many more. Nginx has proven to be a very efficient, lightweight yet powerful web server. Along the chapters of this book, you will discover the many features of Nginx and progressively understand why so many administrators have decided to place their trust in this new HTTP server, often at the expense of Apache.

There are many aspects in which Nginx is more efficient than its competitors. First and foremost, speed. Making use of asynchronous sockets, Nginx does not spawn as many times as it receives requests. One process per core suffices to handle thousands of connections, allowing for a much lighter CPU load and memory consumption. Secondly, ease of use—configuration fi les are much simpler to read and tweak than with other web server solutions such as Apache. A couple of lines are enough to set up a complete virtual host configuration. Last but not least, modularity. Not only is Nginx a completely open source project released under a BSD-like license, but it also comes with a powerful plug-in system—referred to as "modules". A large variety of modules are included with the original distribution archive, and many third-party ones can be downloaded online. All in all, Nginx combines speed, efficiency, and power, providing you the perfect ingredients for a successful web server; it appears to be the best Apache alternative as of today. Although Nginx is available for Windows since version 0.7.52, it is common knowledge that Linux distributions are preferred for hosting production sites. During the various processes described in this book, we will thus assume that you are hosting your website on a Linux operating system such as Debian, Fedora, CentOS, Mandriva, or other well-known distributions.

What This Book Covers

Chapter 1, Preparing your Work Environment provides a basic approach of the Linux command-line environment that we will be using throughout this book.

Chapter 2, Downloading and Installing Nginx guides you through the setup process, by downloading and installing Nginx as well as its prerequisites.

Chapter 3, Basic Nginx Configuration helps you discover the fundamentals of Nginx configuration and set up the Core module.

Chapter 4, HTTP Configuration details the HTTP Core module which contains most of the major configuration sections and directives.

Chapter 5, Module Configuration helps you discover the many first-party modules of Nginx among which are the Rewrite and the SSI modules.

Chapter 6, PHP and Python with Nginx explains how to set up PHP and other third-party applications (if you are interested in serving dynamic websites) to work together with Nginx via FastCGI.

Chapter 7, Apache and Nginx Together teaches you to set up Nginx as reverse proxy server working together with Apache.

Chapter 8, From Apache to Nginx provides a detailed guide to switching from Apache to Nginx.

Appendix A, Directive Index lists and describes all configuration directives, sorted alphabetically. Module directives are also described in their respective chapters too.

Appendix B, Module reference lists available modules.

Appendix C, Troubleshooting discusses the most common issues that administrators face when they configure Nginx

3
Basic Nginx Configuration

In this chapter, we will begin to establish an appropriate configuration for the web server. For this purpose, we first need to approach the topic of syntax in use in the configuration files. Then we need to understand the various directives that will let you optimize your server for different traffic patterns and hardware setups. Finally, create some test pages to make sure that everything has been done correctly and that the configuration is valid. We will only approach the basic configuration directives here; the next chapters will detail more advanced topics such as HTTP module configuration and usage, creating virtual hosts, and more.

This chapter covers:

- Presentation of the configuration syntax
- Basic configuration directives
- Establishing an appropriate configuration for your profile
- Serving a test website
- Testing and maintaining your server

Configuration file syntax

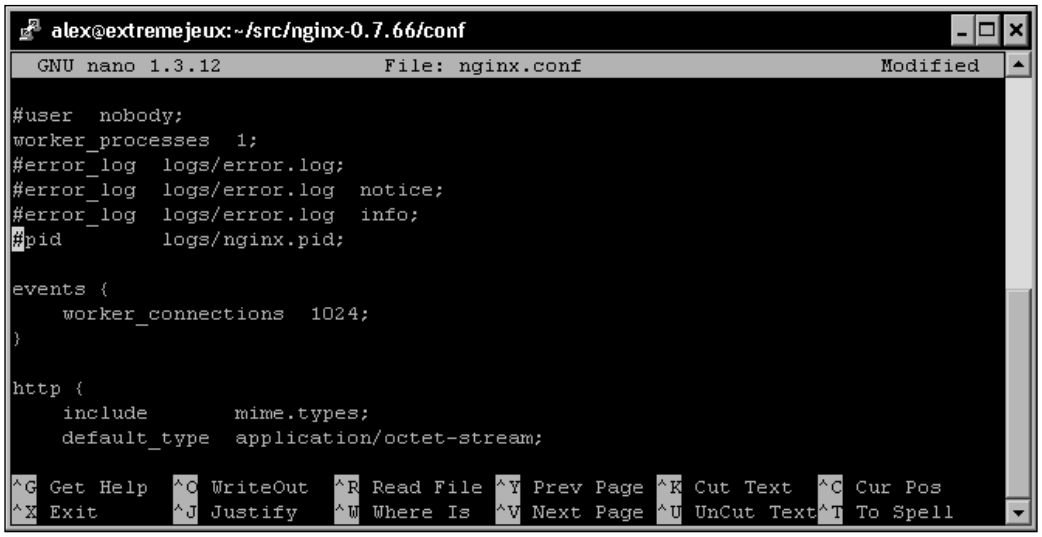
A configuration file is generally a text file that is edited by the administrator and parsed by a program. By specifying a set of values, you define the behavior of the program. In Linux-based operating systems, a large share of applications rely on vast, complex configuration files, which often turn out to be a nightmare to manage. Apache, PHP, MySQL, Qmail, and Bind—all these names bring up bad memories. The fact is that all these applications use their own configuration file with different syntaxes and styles. PHP works with a Windows-style `.ini` file, *sendmail* uses the *M4 macro-processor* to compile configuration files, *Zabbix* pulls its configuration from a MySQL database, and so on. There is, unfortunately, no well-established standard. The same applies to Nginx—you will be required to study a new syntax with its own particularities, its own vocabulary.

On the other hand (and this is one of its advantages), configuring Nginx turns out to be rather simple—at least in comparison to Apache or other mainstream web servers. There are only a few mechanisms that need to be mastered—directives, blocks, and the overall logical structure. Most of the actual configuration process will consist of writing values for directives.

Configuration Directives

The Nginx configuration file can be described as a list of directives organized in a logical structure. The entire behavior of the application is defined by the values that you give to those directives.

By default, Nginx makes use of one main configuration file. The path of this file was defined in the steps described in *Chapter 2*, *Downloading and Installing Nginx* under the *Build configuration* section. If you did not edit the configuration file path and prefix options, it should be located at `/usr/local/nginx/conf/nginx.conf`. Now let's take a quick peek at the first few lines of this initial setup.



A closer look at the first two lines:

```
#user nobody;
worker_processes 1;
```

As you can probably make out from the # character, the first line is a **comment**. In other words, a piece of text that is not interpreted and has no value whatsoever; its sole purpose is to be read by whoever opens the file. You may use the # character at the beginning of a line or following a directive.

The second line is an actual statement—a **directive**. The first bit (`worker_processes`) represents a setting key to which you append one or more values. In this case, the value is `1`, indicating that Nginx should function with a single worker process (more information about this particular directive is given in further sections).

Directives always end with a semicolon (';').

Each directive has a special meaning and defines a particular feature of the application. It may also have a particular syntax. For example, the `worker_process` directive only accepts one numeric value, whereas the `user` directive lets you specify up to two character strings—one for the *user account* (the Nginx worker processes should run as) and a second one for the *user group*.

Nginx works in a modular way, and as such, each module comes with a specific set of directives. The most fundamental directives are part of the Nginx *Core module* and will be detailed in this chapter. As for other directives brought in by other modules, they will be explored in later chapters.

Organization and inclusions

In the preceding screenshot, you may have noticed a particular directive—**include**.

```
include mime.types;
```

As the name suggests, this directive will perform an inclusion of the specified file. In other words, the contents of the file will be inserted at this exact location. Here is a practical example that will help you understand.

`nginx.conf`:

```
user nginx nginx;
worker_processes 4;
include other_settings.conf;
```

`other_settings.conf`:

```
error_log logs/error.log;
pid logs/nginx.pid;
```

Final result, as interpreted by Nginx, is as follows:

```
user nginx nginx;
worker_processes 4;
error_log logs/error.log;
pid logs/nginx.pid;
```

Inclusions are processed recursively. In this case, you have the possibility to use the `include` directive again in the `other_settings.conf` file in order to include yet another file.

In the initial configuration setup, there are two files at use—`nginx.conf` and `mime.types`. However, in the case of a more advanced configuration, there may be five or more files, as described in the table below.

Standard name	Description
`nginx.conf`	Base configuration of the application
`mime.types`	A list of file extensions and their associated MIME types
`fastcgi.conf`	FastCGI-related configuration
`proxy.conf`	Proxy-related configuration
`sites.conf`	Configuration of the websites served by Nginx, also known as virtual hosts. It's recommended to create separate files for each domain.

These filenames were defined conventionally; nothing actually prevents you from regrouping your FastCGI and proxy settings into a common file named `proxy_and_fastcgi_config.conf`.

Note that the `include` directive supports *filename globbing*, in other words, filenames with the `*` wildcard, where `*` may match zero, one, or more consecutive characters:

```
include sites/*.conf;
```

This will include all files with a name that ends with `.conf` in the `sites` folder. This mechanism allows you to create a separate file for each of your websites and include them all at once.

Be careful when including a file—if the specified file does not exist, the configuration checks will fail and Nginx will not start:

```
[alex@example sbin]# ./nginx -t
[emerg]: open() "/usr/local/nginx/conf/dummyfile.conf" failed (2: No
such file or directory) in /usr/local/nginx/conf/nginx.conf:48
```

The previous statement is not true for inclusions with wildcards. Moreover, if you insert `include dummy*.conf` in your configuration and test it (whether there is any file matching this pattern on your system or not), here is what should happen:

```
[alex@example sbin]# ./nginx -t
the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
configuration file /usr/local/nginx/conf/nginx.conf test is successful
```

Directive blocks

Directives are brought in by modules—if you activate a new module, a specific set of directives becomes available. Modules may also enable **directive blocks**, which allow for a logical construction of the configuration.

```
events {
    worker_connections 1024;
}
```

The `events` block that you can find in the default configuration file is brought in by the *Events module*. The directives that the module enables can only be used within that block—in the `preceding` example, `worker_connections` will only make sense in the context of the `events` block. There is one important exception though—some directives may be placed at the root of the configuration file because they have a global effect on the server. The root of the configuration file is also known as the **main** block.

This chapter will detail blocks and directives available in the Core modules—modules that are necessary for the smooth functioning of the server. Optional modules (whether they are enabled by default or not) are discussed in later chapters.

Note that in some cases, blocks can be nested into each other, following a specific logic:

```
http {
    server {
        listen 80;
        server_name example.com;
        access_log /var/log/nginx/example.com.log;
        location ^~ /admin/ {
            index index.php;
        }
    }
}
```

This example shows how to configure Nginx to serve a website, as you can tell from the `http` block (as opposed to, say, *imap*, if you want to make use of the mail server proxy features).

Within the `http` block, you may declare one or more server blocks. A `server` block allows you to configure a virtual host. The `server` block, in this example, contains some configuration that applies to all requests with a `Host` HTTP header exactly matching `example.com`.

Within this `server` block, you may insert one or more `location` blocks. These allow you to enable settings only when the requested URI matches the specified path. More information is provided in *Chapter 4*, *HTTP Configuration* the *Location Block* section.

Last but not least, configuration is inherited within children blocks. The `access_log` directive (defined at the `server` block level in this example) specifies that all HTTP requests for this server should be logged into a text file. This is still true within the `location` child block, although you have the possibility to disable it by reusing the `access_log` directive:

```
[…]
    location ^~ /admin/ {
        index index.php;
        access_log off;
    }
[…]
```

In this case, logging will be enabled everywhere on the website, except for the `/admin/` location path. The value set for the `access_log` directive at the `server` block level is overridden by the one at the `location` block level.

Advanced language rules

There are a number of important observations regarding the Nginx configuration file syntax. These will help you understand certain syntax rules that may seem confusing if you have never worked with Nginx before.

Directives accept specific syntaxes

You may indeed stumble upon complex syntaxes that can be confusing at first sight.

```
rewrite ^/(.*)\.(png|jpg|gif)$ /image.php? file=$1&format=$2 last;
```

Syntaxes are directive-specific. While the listen directive may only accept a port number to open a listening socket, the location block or the rewrite directive support complex expressions in order to match particular patterns. Syntaxes will be explained along with directives in their respective chapters.

Later on, we will approach a module (*the Rewrite module*) that allows for a much more advanced logical structure through the `if`, `set`, `break`, and `return` directives and the use of variables. With all these new elements, configuration files will begin to look like programming scripts. Anyhow, the more modules we discover, the richer the syntax becomes.

Diminutives in directive values

Finally, you may use the following diminutives for specifying a file size in the context of a directive value:

- k or K: Kilobytes
- m or M: Megabytes

As a result, the following two syntaxes are correct and equal:

```
client_max_body_size 2M;
client_max_body_size 2048k;
```

Additionally, when specifying a time value, you may use the following shortcuts:

- ms: Milliseconds
- s: Seconds
- m: Minutes
- h: Hours
- d: Days
- w: Weeks
- M: Months (30 days)
- y: Years (365 days)

This becomes especially useful in the case of directives accepting a period of time as a value:

```
client_body_timeout 3m;
client_body_timeout 180s;
client_body_timeout 180;
```

Note that the default time unit is seconds; the last two lines above thus result in an identical behavior.

Variables

Modules also provide variables that can be used in the definition of directive values. For example, the Nginx HTTP Core module defines the `$nginx_version` variable. When setting the `log_format` directive, you may include all kinds of variables in the format string:

```
[…]
location ^~ /admin/ {
    access_log logs/main.log;
    log_format main '$pid - $nginx_version - $remote_addr';
}
[…]
```

Note that some directives do not allow you to use variables:

```
error_log logs/error-$nginx_version.log;
```

This is a valid configuration directive. However, it simply generates a file named `error-$nginx_version.log`, without parsing the variable.

String values

Character strings that you use as directive values can be written in three forms. First, you may enter the value without quotes:

```
root /home/example.com/www;
```

However, if you want to use a particular character, such as a blank space (" "), a semicolon (`;`), or curly brace (`{` and `}`), you will need to enclose the value in single or double quotes:

```
root '/home/example.com/my web pages';
```

Nginx makes no difference whether you use single or double quotes.

Base module directives

In this section, we will take a closer look at the base modules. We are particularly interested in answering two questions—what are base modules and what directives are made available.

What are base modules?

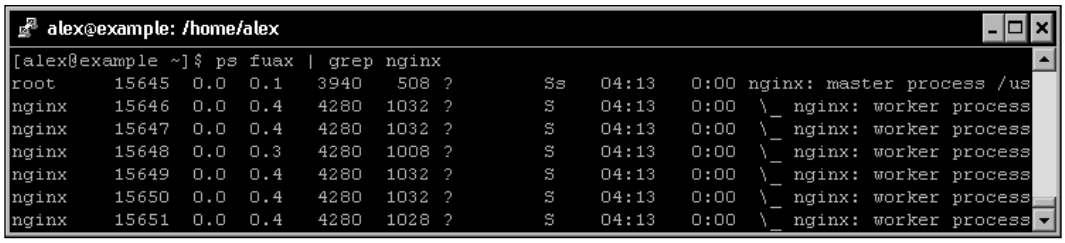
The base modules offer directives that allow you to define parameters of the basic functionality of Nginx. They cannot be disabled at compile time; as a result, the directives and blocks they offer are always available. Three base modules are distinguished:

- **Core module**: Essential features and directives such as process management and security
- **Events module**: It lets you configure the inner mechanisms of the networking capabilities
- **Configuration module**: Enables the inclusion mechanism

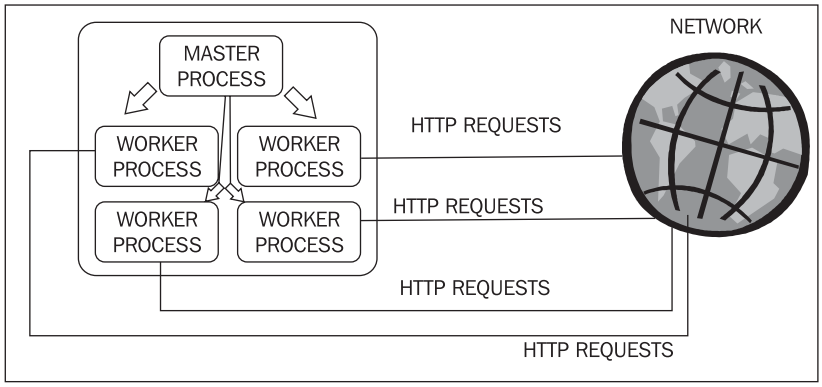
These modules offer a large range of directives; we will be detailing them individually with their syntaxes and default values.

Nginx process architecture

Before we start detailing the basic configuration directives, it's necessary to understand the process architecture, that is, how Nginx works behind the scenes. Although the application comes as a simple binary file, (apparently lightweight background process) the way it functions at runtime is rather intricate.



At the very moment of starting Nginx, one unique process exists in memory—the **Master Process**. It is launched with the current user and group permissions—usually root/root if the service is launched at boot time by an `init` script. The master process itself does not process any client request; instead, it spawns processes that do—the **Worker Process**es, which are affected to a customizable user and group. From the configuration file, you are able to define the amount of worker processes, the maximum connections per worker process, and more.



Core module directives

Below is the list of directives made available by the Core module. Most of these directives must be placed at the root of the configuration file and can only be used once. However, some of them are valid in multiple contexts. If that is the case, the list of valid contexts is mentioned below the directive `name.root` of the configuration file and can only be used once.

Name and context	Syntax and description
`daemon`	Accepted values: `on` or `off` Syntax: `daemon on;` Default value: `on` Enables or disables daemon mode. If you disable it, the program will not be started in the background; it will stay in the foreground when launched from the shell.

Name and context	Syntax and description
`debug_points`	Accepted values: `stop` or `abort` Syntax: `debug_points stop;` Default value: None. Activates debug points in Nginx. Use `stop` to interrupt the application when a debug point comes about in order to attach a debugger. Use `abort` to abort the debug point and create a core dump file. To disable this option, simply do not use the directive.

`env`	Syntax:
	`env MY_VARIABLE;`
	`env MY_VARIABLE=my_value;`
	Lets you (re)define environment variables.

`error_log`
Context: `main`, `http`, `server`, and `location`
	Syntax:
	`error_log /file/path level;`
	Default value: `logs/error.log error`.
	Where level is one of the following values: `debug`, `info`, `notice`, `warn`, `error`, and `crit` (from most to least detailed: debug provides frequent log entries, `crit` only reports critical errors).
	Enables error logging at different levels: Application, HTTP server, virtual host, and virtual host directory.
	By redirecting the log output to `/dev/null`, you can disable error logging. Use the following directive at the root of the configuration file:
	`error_log /dev/null crit;`

`lock_file`	Syntax: File path `lock_file logs/nginx.lock;` Default value: Defined at compile time Use a lock file for mutual exclusion. Disabled by default, unless you enabled it at compile time.
`log_not_found` Context: `main`, `http`, `server`, and `location`	Accepted values: `on` or `off` `log_not_found on;` Default value: `on` Enables or disables logging of **404 not found** HTTP errors. If your logs get filled with 404 errors due to missing `favicon.ico` or `robots.txt` files, you might want to turn this `off`.

Name and context	Syntax and description
`master_process`	Accepted values: `on` or `off` `master_process on;` Default value: `on` If enabled, Nginx will start multiple processes: A main process (the master process) and worker processes. If disabled, Nginx works with a unique process. This directive should be used for testing purposes only as it disables the master process—clients thus cannot connect to your server.
`pid`	Syntax: File path `pid logs/nginx.pid;` Default value: Defined at compile time. Path of the pid file for the Nginx daemon. The default value can be configured at compile time.
`ssl_engine`	Syntax: Character string `ssl_engine enginename;` Default value: None Where `enginename` is the name of an available hardware SSL accelerator on your system. To check for available hardware SSL accelerators, run this command from the shell: **`openssl engine -t`**
`thread_stack_size`	Syntax: Numeric (size) `thread_stack_size 1m;` Default value: None Defines the size of thread stack; please refer to the `worker_threads` directive below
`timer_resolution`	Syntax: Numeric (time) `timer_resolution 100ms;` Default value: None Controls the interval between system calls to `gettimeofday()` to synchronize the internal clock. If this value is not specified, the clock is refreshed after each kernel event notification.

Name and context	Syntax and description

`user`	Syntax:
	`user username groupname;`
	`user username;`
	Default value: Defined at compile time. If still undefined, the user and group of the Nginx master process are used.
	Lets you define the user account and optionally the user group used for starting the Nginx worker processes.

`worker_threads`	Syntax: Numeric `worker_threads 8;` Default value: None Defines the amount of threads per worker process. Warning! Threads are disabled by default. The author stated that "the code is currently broken".

`worker_cpu_affinity`	Syntax:
	`worker_cpu_affinity 1000 0100 0010 0001;`
	`worker_cpu_affinity 10 10 01 01;`
	`worker_cpu_affinity;`
	Default value: None
	This directive works in conjunction with `worker_processes`. It lets you affect worker processes to CPU cores.
	There are as many series of digit blocks as worker processes; there are as many digits in a block as your CPU has cores.
	If you configure Nginx to use three worker processes, there are three blocks of digits. For a dual-core CPU, each block has two digits.
	`worker_cpu_affinity 01 01 10;`
	The first block (01) indicates that the first worker process should be affected to the second core.
	The second block (01) indicates that the second worker process should be affected to the second core.
	The third block (10) indicates that the third worker process should be affected to the first core.
	Note that affinity is only recommended for multi-core CPUs, not for processors with hyper-treading or similar technologies.

Name and context	Syntax and description
`worker_priority`	Syntax: Numeric `worker_priority 0;` Default value: `0` Defines the priority of the worker processes, from `-20` (highest) to `19` (lowest). The default value is `0`. Note that kernel processes run at priority level `-5`, so it's not recommended that you set the priority to `-5` or less.
`worker_processes`	Syntax: Numeric `worker_processes 4;` Default value: `1` Defines the amount of worker processes. Nginx offers to separate the treatment of requests into multiple processes. The default value is 1, but it's recommended to increase this value if your CPU has more than one core. Besides, if a process gets blocked due to slow I/O operations, incoming requests can be delegated to the other worker processes.
`worker_rlimit_core`	Syntax: Numeric (size) `worker_rlimit_core 100m;` Default value: None Defines the size of core files per worker process.
`worker_rlimit_nofile`	Syntax: Numeric `worker_rlimit_nofile 10000;` Default value: None Defines the amount of files a worker process may use simultaneously.
`worker_rlimit_sigpending`	Syntax: Numeric `worker_rlimit_sigpending 10000;` Default value: None Defines the amount of signals that can be queued per user (user ID of the calling process). If the queue is full, signals are ignored past this limit.
`working_directory`	Syntax: Directory path `working_directory /usr/local/nginx/;` Default value: The prefix switch defined at compile time. Working directory used for worker processes; only used to define the location of core files. The worker process user account (`user` directive) must have write permissions on this folder in order to be able to write core files.

Events module

The Events module comes with directives that allow you to configure network mechanisms. Some of the parameters have an important impact on the application's performance.

All of the directives listed below must be placed in the `events` block, which is located at the root of the configuration file:

```
user nginx nginx;
master_process on;
worker_processes 4;
events {
  worker_connections 1024;
  use epoll;
}
[...]
```

These directives cannot be placed elsewhere (if you do so, the configuration test will fail).

Directive name	Syntax and description
`accept_mutex`	Accepted values: `on` or `off` `accept_mutex on;` Default value: `on` Enables or disables the use of an accept mutex (mutual exclusion) to open listening sockets.
`accept_mutex_delay`	`Syntax`: Numeric (time) `accept_mutex_delay 500ms;` Default value: `500 milliseconds` Defines the amount of time a worker process should wait before trying to acquire the resource again. This value is not used if the `accept_mutex` directive is set to off.
`connections`	Replaced by `worker_connections`. This directive is now deprecated.

Directive name	Syntax and description

`debug_connection`	Syntax: IP address or CIDR block.
	`debug_connection 172.63.155.21;`
	`debug_connection 172.63.155.0/24;`
	Default value: None.
	Writes detailed logs for clients matching this IP address or address block. The debug information is stored in the file specified with the `error_log` directive, enabled with the debug level.
	Note: Nginx must be compiled with the `--debug` switch in order to enable this feature.

`multi_accept`	Syntax: `on` or `off` `multi_accept off;` Default value: `off` Defines whether or not Nginx should accept all incoming connections from the listening queue at once.
`use`	Accepted values: `/dev/poll`, `epoll`, `eventport`, `kqueue`, `rtsig`, or `select` `use kqueue;` Default value: Defined at compile time Selects the event model among the available ones (the ones that you enabled at compile time), though Nginx automatically selects the most appropriate one. The supported models are: • `select`: The default and standard module, it is used if the OS does not support a more efficient one (it's the only available method under Windows) • `poll`: It is automatically preferred over `select`, but not available on all systems • `kqueue`: An efficient method for FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0, and MacOS X operating systems • `epoll`: An efficient method for Linux 2.6+ based operating systems • `rtsig`: Real time signals, available as of Linux 2.2.19, but unsuited for high-traffic profiles as default system settings only allow 1,024 queued signals • `/dev/poll`: An efficient method for Solaris 7 11/99+, HP/UX 11.22+, IRIX 6.5.15+, and Tru64 UNIX 5.1A+ operating systems • `eventport`: An efficient method for Solaris 10, though a security patch is required

Directive name	Syntax and description
`worker_connections`	Syntax: Numeric `worker_connections 1024;` Default value: None Defines the amount of connections that a worker process may treat simultaneously.

Configuration module

The Nginx Configuration module is a simple module enabling file inclusions with the include directive, as previously described in the *Organization and inclusions* section. The directive can be inserted anywhere in the configuration file and accepts a single parameter—the file's path.

```
include /file/path.conf;
include sites/*.conf;
```

Note that if you do not specify an absolute path, the file path is relative to the configuration directory. By default, `include sites/example.conf` will include the following file:

```
/usr/local/nginx/conf/sites/example.conf.
```

A configuration for your profile

Following this long list of directives from the base modules, we can begin to envision a first configuration adapted to your profile in terms of targeted traffic and, more importantly, to your hardware. In this section, we will first take a closer look at the default configuration file to understand the implications of each setting.

Understanding the default configuration

There is a reason why Nginx stands apart from other web servers—it's extremely lightweight, optimized, and to put it simply, fast. As such, the default configuration is efficient, and in many cases, you will not need to apply radical changes to the initial setup.

We will study the default configuration by opening up the main configuration file `nginx.conf`, although you will find this file to be almost empty. The reason lies in the fact that when a directive does not appear in the configuration file, the default value is employed. We will thus consider the default values here as well as the directives found in the original setup.

```
user root root;
worker_processes 1;
worker_priority 0;
error_log logs/error.log error;
log_not_found on;
events {
  accept_mutex on;
  accept_mutex_delay 500ms;
  multi_accept off;
  worker_connections 1024;
}
```

While this configuration may work out of the box, there are some issues you need to address right away.

Necessary adjustments

We will review some of the configuration directives that need immediate changing and the possible values you may set:

- `user root root;`

  This directive specifies that the worker processes will be started as root. It is dangerous for security as it grants full permissions over the filesystem. You need to create a new user account on your system and make use of it here. Refer to *Chapter 1*, *Preparing your Work Environment*, the *User and group management* section for more information on creating users and groups. Recommended value (granted that you created an `nginx` user account and group on the system beforehand): `user nginx nginx;`

- `worker_processes 1;`

  With this setting, only one worker process will be started, which implies that all requests will be processed by a unique execution flow (the current version of Nginx is not multi-threaded, by choice). This also implies that the execution is delegated to only one core of your CPU. It is highly recommended to increase this value; you should have at least one process per CPU core. Recommended value (granted your server is powered by a quad-core CPU): `worker_processes 4;`

- `worker_priority 0;`

  By default, the worker processes are started with a regular priority. If your system performs other tasks simultaneously, you might want to grant a higher priority to the Nginx worker processes. In this case, you should decrease the value—the smaller the value, the higher the priority. Values range from `-20` (highest priority) to `19` (lowest priority). There is no recommended value here as it totally depends on your situation. However, you should not set it under `-5` as it is the default priority for kernel processes.

- `log_not_found on;`

  This directive specifies whether Nginx should log `404 errors` or not. While these errors may, of course, provide useful information about missing resources, most of them are generated by web browsers trying to reach the *favicon* (the conventional `/favicon.ico` of a website) or robots trying to access the indexing instructions (`robots.txt`). It is recommended that you disable `log_not_found` in the case of conventional files that may clutter your log files. However, do not disable this at the server level. Note that this directive is part of the HTTP Core module. Refer to the next chapter for more information.

- `worker_connections 1024;`

  This setting, combined with the amount of worker processes, allows you to define the total quantity of connections accepted by the server simultaneously. If you enable four worker processes, each accepting 1,024 connections, your server will treat a total of 4,096 simultaneous connections. You need to adjust this setting to match your hardware: the more RAM and CPU power your server relies on, the more connections you can accept concurrently.

Adapting to your hardware

We will now establish three different setups—a standard one to be used by a regular website with decent hardware, a low-traffic setup intended to optimize performance on modest hardware, and finally an adequate setup for production servers in high-traffic situations.

It's always difficult to classify computer power. Firstly, because each situation has its own resources. If you work in a large company, talking about a *powerful computer* will not have the same meaning as in the case of standalone website administrators who need to resort to third-party web hosting providers. Secondly, because computers get more powerful every year: faster CPUs, cheaper RAM, and the rise of new technologies (SSDs). Consequently, the specifications given below are here for reference and need to be adjusted to your own situation and to your era. The recommended values for the directives are directly based on the specifications—one worker process per CPU core, maximum connections depending on the RAM, and so on.

Low-traffic setup	**Standard setup**	**High-traffic setup**
CPU: Dual-core	CPU: Quad-core	CPU: 8-core
RAM: 2 GB	RAM: 4 GB	RAM: 12 GB
Requests: ~ 1/s	Requests: ~ 50/s	Requests: ~1000/s
Recommended values		

`worker_processes 2;`
`worker_rlimit_nofile 1024;`
`worker_priority -5;`
`worker_cpu_affinity 01 10;`
`events {`
`  multi_accept on;`
`  worker_connections 128;`
`}`
	`worker_processes 4;`
	`worker_rlimit_nofile 8192;`
	`worker_priority 0;`
	`worker_cpu_affinity 0001 0010 0100 1000;`
	`events {`
	`  multi_accept off;`
	`  worker_connections 1024;`
	`}`
		`worker_processes 8;`
		`worker_priority 0;events {`
		`  multi_accept off;`
		`  worker_connections 8192;`
		`}`

There are two adjustments that have a critical effect on the performance, namely, the amount of worker processes and the connection limit. The first one, if set improperly, may clutter particular cores of your CPU and leave other ones unused or underused. Make sure the `worker_processes` match the quantity of cores in your CPU.

The second one, if set too low, could result in connections being refused; if set too high, could overflow the RAM and cause a system-wide crash. Unfortunately, there is no simple equation to calculate the value of the `worker_connections` directive; you will need to base it on expected traffic estimations.

Testing your server

The base configuration of your server is now established. In the following chapters, we will advance to the `http modules` and how to create virtual hosts. But for now, let's make sure that our setup is correct and suitable for production.

Creating a test server

In order to perform simple tests, such as connecting to the server with a web browser, we need to set up a website for Nginx to serve. A test page comes with the default package in the `html` folder (`/usr/local/nginx/html/index.html`) and the original `nginx.conf` is configured to serve this page. Here is the section that we are interested in for now:

```
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    server {
        listen       80;
        server_name  localhost;
        location / {
            root   html;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
}
```

As you can already tell, this segment configures Nginx to serve a website:

- By opening a listening socket on port `80`
- Accessible at the address: `http://localhost/`
- The index page is `index.html`

For more details about these directives, please refer to *Chapter 4*, *HTTP Configuration* and go to the *HTTP module configuration* section. Anyhow, fire up your favorite web browser and visit `http://localhost/`:

You should be greeted with a welcome message; if you aren't, then check the configuration again and make sure you reloaded Nginx in order to apply the changes.

Performance tests

Having configured the basic functioning and the architecture of your Nginx setup, you may already want to proceed with running some tests. The methodology here is experimental—run the tests, edit the configuration, reload the server, run the tests again, edit the configuration again, and so on. Ideally, you should avoid running the testing tool on the same computer that is used to run Nginx as it may cause the results to be biased.

One could question the pertinence of running performance tests at this stage. On one hand, virtual hosts and modules are not fully configured yet and your website might use FastCGI applications (PHP, Python, and so on). On the other hand, we are testing the raw performance of the server without additional components, for example, to make sure that it fully makes use of all CPU cores. Besides, it's always better to come up with a polished configuration before the server is put into production.

We have retained three tools to evaluate the server performance here. All three applications were specifically designed for load tests on web servers and have different approaches due to their origin:

- `httperf`: A relatively well-known open source utility developed by HP, for Linux operating systems only
- `Autobench`: Perl wrapper for `httperf` improving the testing mechanisms and generating detailed reports
- `OpenWebLoad`: Smaller scale open source load testing application; supports both Windows and Linux platforms

The principle behind each of these tools is to generate a massive amount of HTTP requests in order to clutter the server and study the results.

Httperf

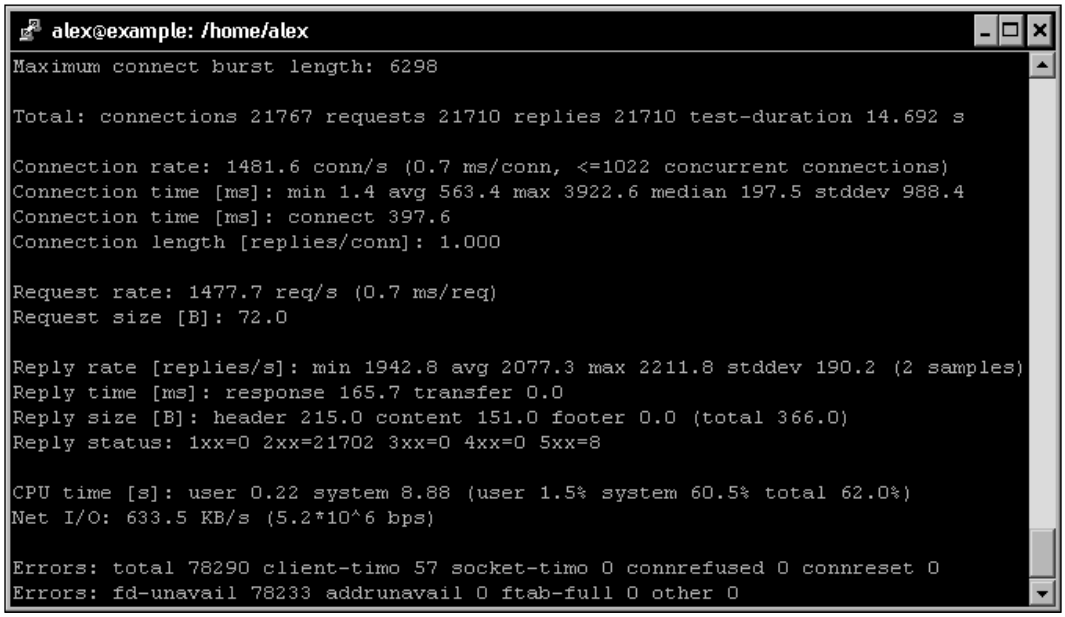
Httperf is a simple command-line tool that can be downloaded from its official website: `http://www.hpl.hp.com/research/linux/httperf/`. The source comes as a `tar.gz` archive and needs to be compiled using the standard method: `./configure`, `make` and `make install`. Once installed, you may execute the following command:

```
[alex@example ~]$ httperf --server 192.168.1.10 --port 80 --uri /
index.html --rate 300 --num-conn 30000 --num-call 1 --timeout 5
```

Replace the values in the command above with your own:

- `--server`: The website hostname you wish to test
- `--uri`: The path of the file that will be downloaded
- `--rate`: How many requests should be sent every second
- `--num-conn`: The total amount of connections
- `--num-call`: How many requests should be sent per connection
- `--timeout`: Quantity of seconds elapsed before a request is considered lost

In this example, `httperf` will download `http://192.168.1.10/index.html` repeatedly, 300 times per second, resulting in a total of 30,000 requests.



The results indicate the response times and the amount of successful requests. If the success ratio is 100 percent or the response time near 0 ms, increase the request rate and run the test again until the server shows signs of weakness. Once the results begin to look a little less perfect, tweak the appropriate configuration directives and run the test again.

Autobench

Autobench is a Perl script that makes use of `httperf` more efficiently—it runs continuous tests and automatically increases request rates until your server gets saturated. One of the interesting features of Autobench is that it generates a `.tsv` report that you can open with various applications to generate graphs. You may download the source code from the author's personal website: `http://www.xenoclast.org/autobench/`. Once again, extract the files from the archive, run `make` then `make install`.

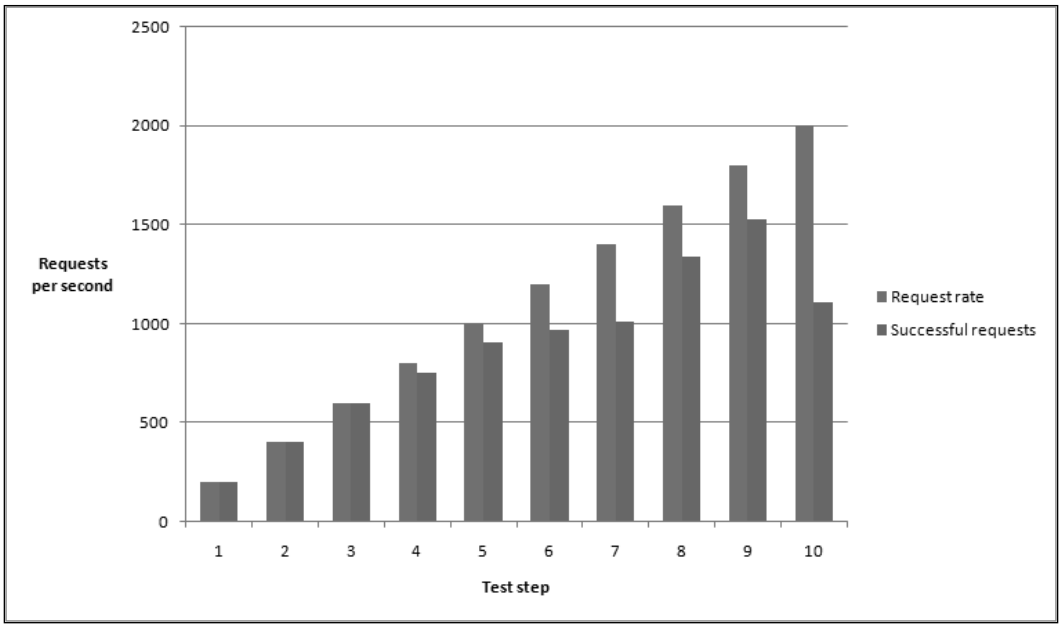
Although it supports testing of multiple hosts at once, we will only be using the single host test for more simplicity. The command we will execute resembles the `httperf` one:

```
[alex@example ~]$ autobench --single_host --host1 192.168.1.10 --uri1 /
index.html --quiet --low_rate 20 --high_rate 200 --rate_step 20 --num_
call 10 --num_conn 5000 --timeout 5 --file results.tsv
```

The switches can be configured as follows:

- `--host1`: The website host name you wish to test.
- `--uri1`: The path of the file that will be downloaded.
- `--quiet`: Does not display httperf information on the screen.
- `--low_rate`: Connections per second at the beginning of the test.
- `--high_rate`: Connections per second at the end of the test.
- `--rate_step`: The number of connections to increase the rate by after each test.
- `--num_call`: How many requests should be sent per connection.
- `--num_conn`: Total amount of connections.
- `--timeout`: The number of seconds elapsed before a request is considered lost.
- `--file`: Export results as specified (.tsv file).

Once the test terminates, you end up with a `.tsv` file that you can import in applications such as Microsoft Excel. Here is a graph generated from results on a test server (note that the report file contains up to 10 series of statistics):



As you can tell from the graph, this test server supports up to 600 requests per second without a loss. Past this limit, some connections get dropped as Nginx cannot handle the load. It stills gets up to over 1,500 successful requests per second at step 9.

Warning: These tests were carried out on a virtual machine and do not reflect the actual capabilities of Nginx running on a production server.

OpenWebLoad

OpenWebLoad is a free open source application. It is available for both Linux and Windows platforms and was developed in the early 2000s, back in the days of *Web 1.0*. A different approach is offered here—instead of throwing loads of requests at the server and seeing how many are handled correctly, it will simply send as many requests as possible using a variable amount of connections and report to you every second.

You may download it from its official website: `http://openwebload.sourceforge.net`. Extract the source from the `.tar.gz` archive, run `./configure, make` and `make install`.

Its usage is simpler than the previous two utilities:

```
[alex@example ~]$ openload example.com/index.html 10
```

The first argument is the URL of the website you want to test. The second one is the amount of connections that should be opened.

A new result line is produced every second. Requests are sent continuously until you press the *Enter* key, following which, a result summary is displayed. Here is how to decipher the output:

- **Tps** (transactions per second): A transaction corresponds to a completed request (back and forth)
- **MaTps**: Average Tps over the last 20 seconds
- **Resp Time**: Average response time for the elapsed second
- **Err** (error rate): Errors occur when the server returns a response that is not the expected `HTTP 200 OK`
- **Count**: Total transaction count

You can fiddle with the amount of simultaneous connections and see how your server performs in order to establish a balanced configuration for your setup. Three tests were run here with a different amount of connections. The results speak for themselves:

	Test 1	Test 2	Test 3
Simultaneous connections	1	20	1000
Transactions per second (Tps)	67.54	205.87	185.07
Average response time	14 ms	91 ms	596 ms

Too few connections result in a low Tps rate; however, the response times are optimal. Too many connections produce a relatively high Tps, but the response times are critically high. You thus need to find a happy medium.

Upgrading Nginx gracefully

There are many situations where you need to replace the Nginx binary, for example, when you compile a new version and wish to put it in production or simply after having enabled new modules and rebuilt the application. What most administrators would do in this situation is stop the server, copy the new binary over the old one, and start Nginx again. While this is not considered to be a problem for most websites, there may be some cases where uptime is critical and connection losses should be avoided at all costs. Fortunately, Nginx embeds a mechanism allowing you to switch binaries with uninterrupted uptime—zero percent request loss is guaranteed if you follow these steps carefully:

1. Replace the old Nginx binary (by default, `/usr/local/nginx/sbin/nginx`) with the new one.
2. Find the pid of the Nginx master process, for example, with `ps x | grep nginx | grep master` or by looking at the value found in the pid file.
3. Send a `USR2` (12) signal to the master process—`kill -USR2 ***`, replacing `***` with the pid found in step 2. This will initiate the upgrade by renaming the old `.pid` file and running the new binary.
4. Send a `WINCH` (28) signal to the old master process—`kill -WINCH ***`, replacing `***` with the pid found in step 2. This will engage a graceful shutdown of the old worker processes.
5. Make sure that all the old worker processes are terminated, and then send a `QUIT` signal to the old master process—`kill -QUIT ***`, replacing `***` with the pid found in step 2.

Congratulations! You have successfully upgraded Nginx and have not lost a single connection.

Summary

This chapter provided a first approach of the configuration architecture by studying the syntax and the core module directives that have an impact on the overall server performance. We then went through a series of adjustments in order to fit your own profile, followed by performance tests that have probably led you to fine-tune some more.

This is just the beginning though—practically everything that we will be doing from now on is to establish configuration sections. The next chapter will detail more advanced directives by further exploring the module system and the exciting possibilities that are offered to you.

Where to buy this book

You can buy Oracle Nginx HTTP Serverfrom the Packt Publishing website: `https://www.packtpub.com/nginx-http-server-for-web-applications/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

For More Information:
www.PacktPub.com/nginx-http-server-for-web-applications/book