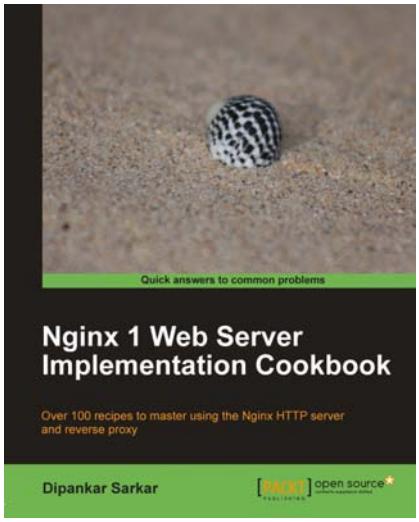
[PACKT] PUBLISHING

Nginx 1 Web Server Implementation Cookbook

Dipankar Sarkar



Chapter No.7
"Nginx as a Reverse Proxy"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "Nginx as a Reverse Proxy"

A synopsis of the book's content

Information on where to buy this book

About the Author

Dipankar Sarkar is a web and mobile entrepreneur. He has a Bachelor's degree in Computer Science and Engineering from the Indian Institute of Technology, Delhi. He is a firm believer in the Open source movement and has participated in the Google Summer of Code, 2005-06 and 2006-07. He has conducted technical workshops for Windows mobile and Python at various technical meet ups. He recently took part in the Startup Leadership Program, Delhi Chapter.

He has worked with Slideshare LLC, one of the world's largest online presentation hosting and sharing service as an early engineering employee. He has since then worked with Mpower Mobile LLC, a mobile payment startup and Clickable LLC, a leading search engine marketing startup. He was a co-founder at Kwippy, which was one of the top micro-blogging sites. He is currently working in the social TV space and has co-founded Jaja.

This is his first technical publication

I would like to thank my patient and long suffering wife, Maitrayee, for putting up with my insane working hours and for being there for me throughout. My mother and my sister, Rickta and Amrita, whose belief and support sustains me. My in laws, Amal and Ruchira Roychoudhury, for opening their homes and hearts to me so generously. Also to my father, the late A. C. Sarkar, who co-authored the first chapter of my life and without whom none of this would have been possible.

I would also like to thank Usha, Hyacintha, Srimoyee, and Kavita from Packt who made this opportunity possible and have been fantastic to work with. I am deeply grateful to the technical reviewers whose insights have been invaluable. Needless to say, errors, if any, are mine.

Nginx 1 Web Server Implementation Cookbook

Nginx is an open source high-performance web server, which has gained quite some popularity recently. Due to its modular architecture and small footprint, it has been the default choice for a lot of smaller Web 2.0 companies to be used as a load-balancing proxy server. It supports most of the existing backend web protocols such as FCGI, WSGI, and SCGI. This book is for you if you want to have in-depth knowledge of the Nginx server.

Nginx 1 Web Server Implementation Cookbook covers the whole range of techniques that would prove useful for you in setting up a very effective web application with the Nginx web server. It has recipes for lesser-known applications of Nginx like a mail proxy server, streaming of video fi les, image resizing on the fl y, and much more.

The first chapter of the book covers the basics that would be useful for anyone who is starting with Nginx. Each recipe is designed to be independent of the others.

The book has recipes based on broad areas such as core, logging, rewrites, security, and others. We look at ways to optimize your Nginx setup, setting up your WordPress blog, blocking bots that post spam on your site, setting up monitoring using munin, and much more.

Nginx 1 Web Server Implementation Cookbook makes your entry into the Nginx world easy with step-by-step recipes for nearly all the tasks necessary to run your own web application.

A practical guide for system administrators and web developers alike to get the best out of the open source Nginx web server.

What This Book Covers

Chapter 1, The Core HTTP Module, deals with the basics of Nginx configuration and implementation. By the end of it you should be able to compile Nginx on your machine, create virtual hosts, set up user tracking, and get PHP to work.

Chapter 2, All About Rewrites: The Rewrite Module, is devoted to the rewrite module; it will teach you the basics and also allow you to configure various commonly available web development frameworks to work correctly with your Nginx setup using the correct rewrite rules.

Chapter 3, Get It All Logged: The Logging Module, aims to teach the basics as well as the advanced configurations that can be done around the Nginx logging module, like log management, backup, rotation, and more. Logging is very crucial as it can help you

identify and track various attributes of your application like performance, user behavior, and much more. It also helps you as a system administrator to identify, both reactively and proactively, potential security issues.

Chapter 4, Slow Them Down: Access and Rate Limiting Module, explains how Nginx provides good protection against cases such as bringing down sites by providing rate limiting and server access based on IP.

Chapter 5, Let's be Secure: Security Modules, looks at how we can use the security modules built-in Nginx to secure your site and user's data.

Chapter 6, Setting Up Applications: FCGI and WSGI Modules, has a practical section devoted to helping programmers and system administrators understand and install their applications using Nginx as the web server. Due to the lack of integrated modules for running PHP and Python, the setting up of such systems can be an issue for non-experienced system administrators.

Chapter 7, Nginx as a Reverse Proxy, deals with the usage of Nginx as a reverse proxy in various common scenarios. We will have a look at how we can set up a rail application; set up load balancing, and also have a look at caching setup using Nginx, which will potentially enhance the performance of your existing site without any codebase changes.

Chapter 8, Improving Performance and SEO Using Nginx, is all about how you can make your site load faster and possibly get more traffic on your site. We will cover the basics of optimizing your Nginx setup and some SEO tricks. These techniques will not only be useful for your SEO, but also for the overall health of your site and applications.

Chapter 9, Using Other Third-party Modules, a look at some inbuilt and third-party modules which allow us to extend and use Nginx with other protocols such as IMAP, POP3, WebDAV, and much more. Due to the flexible and well-defined module API, many module developers have used Nginx for interesting web-based tasks such as XSLT transformations, image resizing, and HTTP publish-subscribe server.

Chapter 10, Some More Third-party Modules, looks at various web situations such as load balancing, server health checks, and more which will be very useful in a production environment. These simple recipes will be highly applicable in enterprise scenarios where you may need to have analytics, external authentication schemes, and many other situations.

7
Nginx as a Reverse Proxy

In this chapter, we will cover:

- Using Nginx as a simple reverse proxy
- Setting up a rails site using Nginx as a reverse proxy
- Setting up correct reverse proxy timeouts
- Setting up caching on the reverse proxy
- Using multiple backends for the reverse proxy
- Serving CGI files using thttpd and Nginx
- Setting up load balancing with reverse proxy
- Splitting requests based on various conditions using split-clients

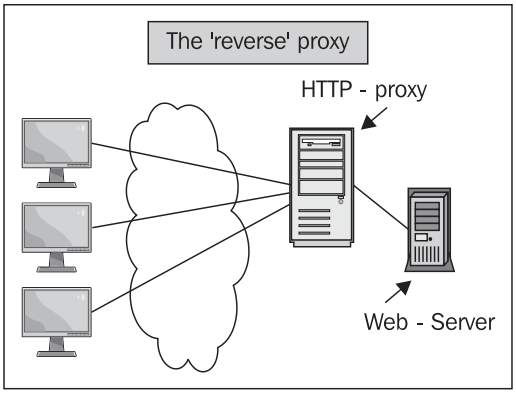
Introduction

Nginx has found most applications acting as a reverse proxy for many sites. A reverse proxy is a type of proxy server that retrieves resources for a client from one or more servers. These resources are returned to the client as though they originated from the proxy server itself.

Due to its event driven architecture and C codebase, it consumes significantly lower CPU power and memory than many other better known solutions out there. This chapter will deal with the usage of Nginx as a reverse proxy in various common scenarios. We will have a look at how we can set up a rail application, set up load balancing, and also look at a caching setup using Nginx, which will potentially enhance the performance of your existing site without any codebase changes.

Using Nginx as a simple reverse proxy

Nginx in its simplest form can be used as a reverse proxy for any site; it acts as an intermediary layer for security, load distribution, caching, and compression purposes. In effect, it can potentially enhance the overall quality of the site for the end user without any change of application source code by distributing the load from incoming requests to multiple backend servers, and also caching static, as well as dynamic content.



How to do it...

You will need to first define `proxy.conf`, which will be later included in the main configuration of the reverse proxy that we are setting up:

```
proxy_redirect          off;
proxy_set_header        Host            $host;
proxy_set_header        X-Real-IP       $remote_addr;
proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
client_max_body_size    10m;
client_body_buffer_size 128k;
proxy_connect_timeout   90;
proxy_send_timeout      90;
proxy_read_timeout      90;s
proxy_buffers           32 4k
```

To use Nginx as a reverse proxy for a site running on a local port of the server, the following configuration will suffice:

```
server {
 listen   80;
 server_name  example1.com;
 access_log  /var/www/example1.com/log/nginx.access.log;
```

```
 error_log  /var/www/example1.com/log/nginx_error.log debug;
location / {
  include proxy.conf;
  proxy_pass         http://127.0.0.1:8080;
 }
}
```

How it works...

In this recipe, Nginx simply acts as a proxy for the defined backend server which is running on the 8080 port of the server, which can be any HTTP web application. Later in this chapter, other advanced recipes will have a look at how one can define more backend servers, and how we can set them up to respond to requests.

Setting up a rails site using Nginx as a reverse proxy

In this recipe, we will set up a working rails site and set up Nginx working on top of the application. This will assume that the reader has some knowledge of rails and thin. There are other ways of running Nginx and rails, as well, like using Passenger Phusion.

How to do it...

This will require you to set up thin first, then to configure thin for your application, and then to configure Nginx.

1. If you already have gems installed then the following command will install thin, otherwise you will need to install it from source:

   ```
   sudo gem install thin
   ```

2. Now you need to generate the thin configuration. This will create a configuration in the `/etc/thin` directory:

```
sudo thin config -C /etc/thin/myapp.yml -c /var/rails/myapp
--servers 5 -e production
```

3. Now you can start the thin service. Depending on your operating system the start up command will vary.
4. Assuming that you have Nginx installed, you will need to add the following to the configuration file:

```
upstream thin_cluster {
    server unix:/tmp/thin.0.sock;
    server unix:/tmp/thin.1.sock;
    server unix:/tmp/thin.2.sock;
    server unix:/tmp/thin.3.sock;
    server unix:/tmp/thin.4.sock;
  }

  server {
    listen       80;
    server_name  www.example1.com;

    root /var/www.example1.com/public;

    location / {
      proxy_set_header  X-Real-IP  $remote_addr;
      proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      proxy_redirect false;

      try_files $uri $uri/index.html $uri.html @thin;
      location @thin {
        include proxy.conf;
        proxy_pass http://thin_cluster;
      }
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
      root   html;
    }
  }
```

How it works...

This is a fairly simple rails stack, where we basically configure and run five upstream thin threads which interact with Nginx through socket connections.

There are a few rewrites that ensure that Nginx serves the static files, and all dynamic requests are processed by the rails backend. It can also be seen how we set proxy headers correctly to ensure that the client IP is forwarded correctly to the rails application. It is important for a lot of applications to be able to access the client IP to show geo-located information, and logging this IP can be useful in identifying if geography is a problem when the site is not working properly for specific clients.

Setting up correct reverse proxy timeouts

In this section we will set up correct reverse proxy timeouts which will affect your user's interaction when your backend application is unable to respond to the client's request.

In such a case, it is advisable to set up some sensible timeout pages so that the user can understand that further refreshing may only aggravate the issues on the web application.

How to do it...

You will first need to set up `proxy.conf` which will later be included in the configuration:

```
proxy_redirect          off;
proxy_set_header        Host            $host;
proxy_set_header        X-Real-IP       $remote_addr;
proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
client_max_body_size    10m;
client_body_buffer_size 128k;
proxy_connect_timeout   90;
proxy_send_timeout      90;
proxy_read_timeout      90;s
proxy_buffers           32 4k
```

Reverse proxy timeouts are some fairly simple flags that we need to set up in the Nginx configuration like in the following example:

```
server {
 listen   80;
 server_name  example1.com;
 access_log  /var/www/example1.com/log/nginx.access.log;
 error_log  /var/www/example1.com/log/nginx_error.log debug;
 #set your default location
```

```
    location / {
     include proxy.conf;
     proxy_read_timeout 120;
     proxy_connect_timeout 120;
     proxy_pass         http://127.0.0.1:8080;
    }
   }
```

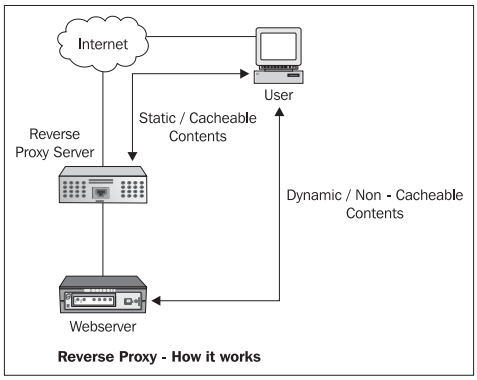
How it works...

In the preceding configuration we have set the following variables, it is fairly clear what these variables achieve in the context of the configurations:

Directive	Use
`proxy_read_timeout`	This directive sets the read timeout for the response of the proxied server. It determines how long Nginx will wait to get the response to a request. The timeout is established not for the entire response, but only between two operations of reading.
`proxy_connect_timeout`	This directive assigns timeout with the transfer of request to the upstream server. Timeout is established not on the entire transfer of request, but only between two write operations. If after this time the upstream server does not take new data, then Nginx shuts down the connection.

Setting up caching on the reverse proxy

In a setup where Nginx acts as the layer between the client and the backend web application, it is clear that caching can be one of the benefits that can be achieved. In this recipe, we will have a look at setting up caching for any site to which Nginx is acting as a reverse proxy. Due to extremely small footprint and modular architecture, Nginx has become quite the Swiss knife of the modern web stack.



Reverse Proxy - How it works

How to do it...

This example configuration shows how we can use caching when utilizing Nginx as a reverse proxy web server:

```
http {
  proxy_cache_path  /var/www/cache levels=1:2 keys_zone=my-cache:8m
max_size=1000m inactive=600m;
  proxy_temp_path /var/www/cache/tmp;
...
server {
 listen   80;
 server_name  example1.com;
 access_log  /var/www/example1.com/log/nginx.access.log;
 error_log  /var/www/example1.com/log/nginx_error.log debug;
 #set your default location
 location / {
      include proxy.conf;
      proxy_pass         http://127.0.0.1:8080/;
      proxy_cache my-cache;
      proxy_cache_valid  200 302  60m;
      proxy_cache_valid  404      1m;
 }
}
}
```

How it works...

This configuration implements a simple cache with 1000MB maximum size, and keeps all HTTP response 200 pages in the cache for 60 minutes and HTTP response 404 pages in cache for 1 minute.

There is an initial directive that creates the cache file on initialization, in further directives we basically configure the location that is going to be cached.

It is possible to actually set up more than one cache path for multiple locations.

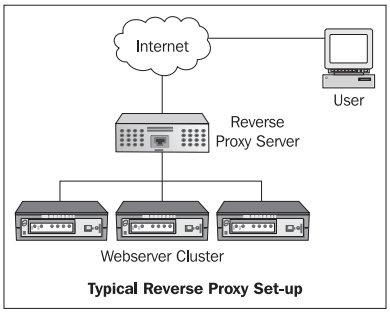
There's more...

This was a relatively small show of what can be achieved with the caching aspect of the proxy module. Here are some more directives that can be really useful in optimizing and making your stack faster and more efficient:

Directive	Use
`proxy_cache_bypass`	The directive specifies the conditions under which the answer will not be taken from the cache. If one string variable is not empty and not equal to "0", the answer is not taken from the cache.
`proxy_cache_min_uses`	This directive determines the number of accesses before a page is cached.
`proxy_cache_use_stale`	This directive tells Nginx when to serve a stale item from the proxy cache. For example, when an Application error HTTP Code 500 occurs.
`proxy_cache_methods`	This directive lets you choose what directives to cache [GET, PUT, and so on].

Using multiple backends for the reverse proxy

As traffic increases, the need to scale the site up becomes a necessity. With a transparent reverse proxy like Nginx in front, most users never even see the scaling affecting their interactions with the site. Usually, for smaller sites one backend process is sufficient to handle the oncoming traffic. As the site popularity increases, the first solution is to increase the number of backend processes and let Nginx multiplex the client requests. This recipe takes a look at how to add new backend processes to Nginx.



Typical Reverse Proxy Set-up

How to do it...

The configuration below adds three upstream servers to which client requests will be sent for processing:

```
upstream backend  {
  server backend1.example1.com weight=5;
  server backend2.example1.com max_fails=3  fail_timeout=30s;
  server backend3.example1.com;
}
server {
 listen   80;
 server_name  example1.com;
 access_log  /var/www/example1.com/log/nginx.access.log;
 error_log  /var/www/example1.com/log/nginx_error.log debug;
 #set your default location
 location / {
  include proxy.conf;
  proxy_pass         http://backend;
 }
}
```

How it works...

In this configuration we set up an upstream, which is nothing but a set of servers with some proxy parameters. For the server `http://backend1.example1.com`, we have set a weight of five, which means that the majority of the requests will be directed to that server. This can be useful in cases where there are some powerful servers and some weaker ones. In the next server `http://backend2.example1.com`, we have set the parameters such that three failed requests over a time period of 30 seconds will result in the server being considered inoperative. The last one is a plain vanilla setup, where one error in a ten second window will make the server inoperative!

This displays the thought put in behind the design of Nginx. It seamlessly handles servers which are problematic and puts them in the set of inoperative servers. All requests to the server are sent in a round robin fashion. We will discuss modules in future recipes that ensure that the requests are sent using other queue mechanisms based on server load and other upstream server performance metrics.

Serving CGI files using thttpd and Nginx

At some point in time in Internet history, most applications were CGI based. Nginx does not serve CGI scripts, so the workaround is to use a really efficient and simple HTTP server called thttpd and to get Nginx to act as a proxy to it.

How to do it...

The best way to go about it is to set up thttpd from source code, apply the IP forwarding patch, and then to use the configuration below:

1. Download thttpd and apply the patch.

```
wget http://www.acme.com/software/thttpd/thttpd-2.25b.tar.gz
tar –xvzf  thttpd-2.25b.tar.gz
```

2. Save the code below in a file called `thttpd.patch`:

```
--- thttpd-2.25b/libhttpd.c   2003-12-25 20:06:05.000000000 +0100
+++ thttpd-2.25b-patched/libhttpd.c   2005-01-09
00:26:04.867255248 +0100
@@ -2207,6 +2207,12 @@
       if ( strcasecmp( cp, "keep-alive" ) == 0 )
           hc->keep_alive = 1;
       }
+       else if ( strncasecmp( buf, "X-Forwarded-For:", 16 ) == 0
)
+      { // Use real IP if available
+      cp = &buf[16];
+      cp += strspn( cp, " \t" );
+      inet_aton( cp, &(hc->client_addr.sa_in.sin_addr) );
+           }
 #ifdef LOG_UNKNOWN_HEADERS
        else if ( strncasecmp( buf, "Accept-Charset:", 15 ) == 0
||
             strncasecmp( buf, "Accept-Language:", 16 ) == 0 ||
```

3. Apply the patch and install thttpd:

```
patch -p 1 -i thttpd.patch
cd thttpd-2.25b
make
sudo make install
```

4. Use the following configuration for `/etc/thttpd.conf`:

```
# BEWARE : No empty lines are allowed!
# This section overrides defaults
# This section _documents_ defaults in effect
# port=80
# nosymlink         # default = !chroot
# novhost
# nocgipat
# nothrottles
# host=0.0.0.0
# charset=iso-8859-1
host=127.0.0.1
port=8000
user=thttpd
logfile=/var/log/thttpd.log
pidfile=/var/run/thttpd.pid
dir=/var/www
cgipat=**.cgi|**.pl
```

5. Set up Nginx as a proxy for the port 8000.

```
server {
 listen   80;
 server_name  example1.com;
 access_log  /var/www/example1.com/log/nginx.access.log;
 error_log  /var/www/example1.com/log/nginx_error.log debug;

location /cgi-bin {
  include proxy.conf;
  proxy_pass         http://127.0.0.1:8000;
 }
}
```

How it works...

The setup above allows you to enjoy the best of CGI and Nginx. You initially set up thttpd, which will run on port 8000 of the server, which will effectively be the core CGI web server and you can run Nginx as the proxy for the user requests.

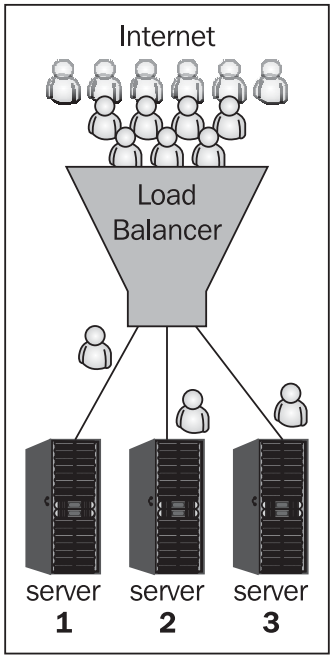
All you need to do is place the perl scripts in the `/var/www` directory and you will be running CGI using Nginx and thttpd.

You can also use the same technique as above to run CGI scripts using other CGI-capable servers like Apache and lightHTTPD as well. You will be required to change the operating ports of those servers to 8000 and the same configuration like above will work.

Setting up load balancing with reverse proxy

In most reverse proxy systems one wants to have some notion of load balancing in the system. In one of the preceding recipes, we have seen how to set up and run multiple upstream servers in a round robin mechanism of sending over the requests.

In this recipe, we will install a load balancing module which will allow us to set up a fair load balancing with the upstream servers.



How to do it...

For this particular recipe we will install a third-party module called "upstream fair module".

1. You will need to go an download the module:

```
wget https://github.com/gnosek/nginx-upstream-fair/tarball/master
```

2. Compile Nginx with the new module:

```
Tar –xvzf nginx-upstream-fair.tgz
Cd nginx
./configure --with-http_ssl_module --add-module=../nginx-upstream-fair/
Make && make install
```

3. You will need to add the following configuration to your `nginx.conf`:

```
upstream backend  {
  server backend1.example1.com;
  server backend2.example1.com;
  server backend3.example1.com;
  fair no_rr;
}

server {
 listen   80;
 server_name  example1.com;
 access_log  /var/www/example1.com/log/nginx.access.log;
 error_log  /var/www/example1.com/log/nginx_error.log debug;

 #set your default location
 location / {
  proxy_pass         http://backend;
 }
}
```

How it works...

This is a fairly straightforward setup once you understand the basics of setting up multiple upstream servers. In this particular "fair" mode, which is `no_rr`, the server will send the request to the first backend whenever it is idle. The goal of this module is to not send requests to already busy backends as it keeps information of how many requests a current backend is already processing. This is a much better model than the default round robin that is implemented in the default upstream directive.

There's more...

You can choose to run this load balancer module in a few other modes, as described below, based on your needs! This is a very simple way of ensuring that none of the backend experiences load unevenly as compared to the rest:

Mode	Meaning
default (that is fair;)	The default mode is a simple WLC-RR (weighted least-connection round-robin) algorithm with a caveat that the weighted part isn't actually too fair under low load.
`no_rr`	This means that whenever the first backend is idle, it's going to get the next request. If it's busy, the request will go to the second backend unless it's busy too, and so on.
`weight_mode=idle no_rr`	This mode redefines the meaning of "idle". It now means "less than weight concurrent requests". So you can easily benchmark your backends and determine that **X** concurrent requests are the maximum for you.
`weight_mode=peak`	This means that Nginx will never send more than weight requests to any single backend. If all backends are full, you will start receiving 502 errors.

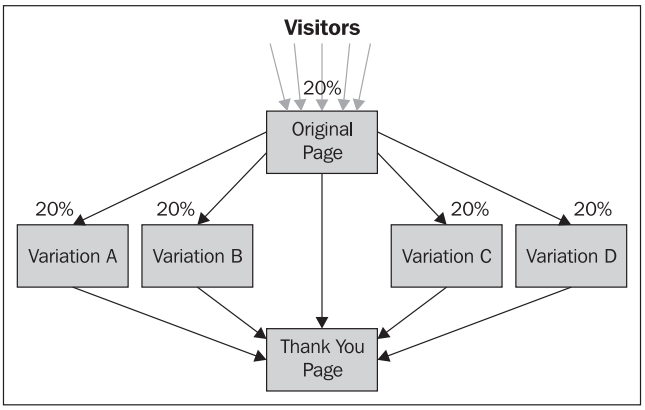
Here is an example of a peak weight mode setup:

```
upstream backend  {
  server backend1.example1.com weight=4;
  server backend2.example1.com weight=3;
  server backend3.example1.com weight=4;
  fair weight_mode=idle no_rr;
}
```

Splitting requests based on various conditions using split-clients

This recipe will take a look at how we can potentially separate client requests based on various conditions that can arise.

We will also understand how we can potentially set up a simple page for A-B testing using this module.



How to do it...

This module is fairly simple to use and comes inbuilt with Nginx. All you need to do is to insert the following configuration in your `nginx.conf`:

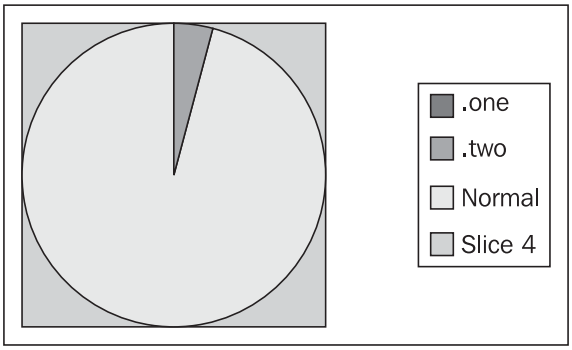
```
http {
    split-clients "${remote-addr}AAA" $variant {
        50.0% .one;
        50,0% .two;
        - "";
    }
...
server {
 listen   80;
 server_name  example1.com;
 access_log  /var/www/example1.com/log/nginx.access.log;
 error_log  /var/www/example1.com/log/nginx_error.log debug;
location / {
  root /var/www/example1.com;
  index index${variant}.html;
 }
}
}
```

How it works...

This particular configuration sets up a system which is based upon the remote client address, assigns the values `.one`, `.two`, or "" to a variable `$variant`. Based upon the variable value, a different page is picked up from the file location.

The following table shows the various probabilities and actions from the above configuration:

Variable value	Probability	Page served
`.one`	50%	`/var/www/example1.com/index.one.html`
`.two`	50%	`/var/www/example1.com/index.two.html`
`""`	0%	`/var/www/example1.com/index.html`



The preceding pie chart clearly displays the split across the two pages. Utilizing this approach, we are able to test out interactions with the page changes that you have made. This forms the basis of usability testing.

Where to buy this book

You can buy Nginx 1 Web Server Implementation Cookbook from the Packt Publishing website: `http://www.packtpub.com/nginx-1-web-server-implementation-cookbook/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-1-web-server-implementation-cookbook/book