2019 UPDATE

# NGINX Cookbook

## *Advanced Recipes for High Performance Load Balancing*

*Derek DeJonghe*

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

**NGINX Cookbook**
by Derek DeJonghe

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Development Editor:** Virginia Wilson
**Acquisitions Editor:** Brian Anderson
**Production Editor:** Justin Billing
**Copyeditor:** Octal Publishing, LLC
**Proofreader:** Chris Edwards
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

March 2017: First Edition

**Revision History for the First Edition**

2017-05-26: First Release
2018-11-21: Second Release

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O’Reilly and NGINX. See our *statement of editorial independence*.

978-1-491-96893-2

[LSI]

# Table of Contents

# Foreword

Welcome to the updated edition of the *NGINX Cookbook*. It has been nearly two years since O'Reilly published the original *NGINX Cookbook*. A lot has changed since then, but one thing hasn't: every day more and more of the world's websites choose to run on NGINX. Today there are 300 million, nearly double the number when the first cookbook was released.

There are a lot of reasons NGINX use is still growing 14 years after its initial release. It's a Swiss Army knife: NGINX can be a web server, load balancer, content cache, and API gateway. But perhaps more importantly, it's reliable.

The *NGINX Cookbook* shows you how to get the most out of NGINX Open Source and NGINX Plus. You will find over 150 pages of easy-to-follow recipes covering everything from how to properly install NGINX, to how to configure all the major features, to debugging and troubleshooting.

This updated version also covers new open source features like gRPC support, HTTP/2 server push, and the Random with Two Choices load-balancing algorithm for clustered environments as well as new NGINX Plus features like support for state sharing, a new NGINX Plus API, and a key-value store. Almost everything you need to know about NGINX is covered in these pages.

We hope you enjoy the *NGINX Cookbook* and that it contributes to your success in creating and deploying the applications we all rely on.

*— Faisal Memon*
*Product Marketing Manager, NGINX, Inc.*

# Preface

The *NGINX Cookbook* aims to provide easy-to-follow examples to real-world problems in application delivery. Throughout this book you will explore the many features of NGINX and how to use them. This guide is fairly comprehensive, and touches most of the main capabilites of NGINX.

Throughout this book, there will be references to both the free and open source NGINX software, as well as the commercial product from NGINX, Inc., NGINX Plus. Features and directives that are only available as part of the paid subscription to NGINX Plus will be denoted as such. Because NGINX Plus is an application delivery contoller and provides many advanced features, it's important to highlight these features to gain a full view of the possibilities of the platform.

The book will begin by explaining the installation process of NGINX and NGINX Plus, as well as some basic getting started steps for readers new to NGINX. From there, the sections will progress to load balancing in all forms, accompanied by chapters about traffic management, caching, and automation. The authentication and security controls chapters cover a lot of ground but are important as NGINX is often the first point of entry for web traffic to your application, and the first line of application layer defense. There are a number of chapters that cover cutting edge topics such as HTTP/2, media streaming, cloud and container environments, wrapping up with more traditional operational topics such as monitoring, debugging, performance, and operational tips.

I personally use NGINX as a multitool, and believe this book will enable you to do the same. It's software that I believe in and enjoy working with. I'm happy to share this knowledge with you, and hope that as you read through this book you relate the recipes to your real world scenarios and employ these solutions.

CHAPTER 1

# Basics

## 1.0 Introduction

To get started with NGINX Open Source or NGINX Plus, you first need to install it on a system and learn some basics. In this chapter you will learn how to install NGINX, where the main configuration files are, and commands for administration. You will also learn how to verify your installation and make requests to the default server.

## 1.1 Installing on Debian/Ubuntu

### Problem

You need to install NGINX Open Source on a Debian or Ubuntu machine.

### Solution

Create a file named */etc/apt/sources.list.d/nginx.list* that contains the following contents:

```
deb http://nginx.org/packages/mainline/OS/ CODENAME nginx
deb-src http://nginx.org/packages/mainline/OS/ CODENAME nginx
```

Alter the file, replacing `OS` at the end of the URL with `ubuntu` or `debian`, depending on your distribution. Replace `CODENAME` with the code name for your distrobution; `jessie` or `stretch` for Debian, or trusty, xenial, artful, or bionic for ubuntu. Then, run the following commands:

```
wget http://nginx.org/keys/nginx_signing.key
apt-key add nginx_signing.key
apt-get update
apt-get install -y nginx
/etc/init.d/nginx start
```

### Discussion

The file you just created instructs the apt package management system to utilize the Official NGINX package repository. The commands that follow download the NGINX GPG package signing key and import it into apt. Providing apt the signing key enables the apt system to validate packages from the repository. The apt-get update command instructs the apt system to refresh its package listings from its known repositories. After the package list is refreshed, you can install NGINX Open Source from the Official NGINX repository. After you install it, the final command starts NGINX.

## 1.2 Installing on RedHat/CentOS

### Problem

You need to install NGINX Open Source on RedHat or CentOS.

### Solution

Create a file named */etc/yum.repos.d/nginx.repo* that contains the following contents:

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/mainline/OS/OSRELEASE/$basearch/
gpgcheck=0
enabled=1
```

Alter the file, replacing OS at the end of the URL with rhel or centos, depending on your distribution. Replace OSRELEASE with 6 or 7 for version 6.x or 7.x, respectively. Then, run the following commands:

```
yum -y install nginx
systemctl enable nginx
systemctl start nginx
firewall-cmd --permanent --zone=public --add-port=80/tcp
firewall-cmd --reload
```

## Discussion

The file you just created for this solution instructs the yum package management system to utilize the Official NGINX Open Source package repository. The commands that follow install NGINX Open Source from the Official repository, instruct systemd to enable NGINX at boot time, and tell it to start it now. The firewall commands open port `80` for the TCP protocol, which is the default port for HTTP. The last command reloads the firewall to commit the changes.

# 1.3 Installing NGINX Plus

## Problem

You need to install NGINX Plus.

## Solution

Visit http://cs.nginx.com/repo_setup. From the drop-down menu, select the OS you're installing and then follow the instructions. The instructions are similar to the installation of the open source solutions; however, you need to install a certificate in order to authenticate to the NGINX Plus repository.

## Discussion

NGINX keeps this repository installation guide up to date with instructions on installing the NGINX Plus. Depending on your OS and version, these instructions vary slightly, but there is one commonality. You must log in to the NGINX portal to download a certificate and key to provide to your system that are used to authenticate to the NGINX Plus repository.

# 1.4 Verifying Your Installation

## Problem

You want to validate the NGINX installation and check the version.

## Solution

You can verify that NGINX is installed and check its version by using the following command:

```
$ nginx -v
nginx version: nginx/1.15.3
```

As this example shows, the response displays the version.

You can confirm that NGINX is running by using the following command:

```
$ ps -ef | grep nginx
root      1738     1  0 19:54 ?  00:00:00 nginx: master process
nginx     1739  1738  0 19:54 ?  00:00:00 nginx: worker process
```

The `ps` command lists running processes. By piping it to `grep`, you can search for specific words in the output. This example uses `grep` to search for `nginx`. The result shows two running processes, a `master` and `worker`. If NGINX is running, you will always see a master and one or more worker processes. For instructions on starting NGINX, refer to the next section. To see how to start NGINX as a daemon, use the init.d or systemd methodologies.

To verify that NGINX is returning requests correctly, use your browser to make a request to your machine or use `curl`:

```
$ curl localhost
```

You will see the NGINX Welcome default HTML site.

## Discussion

The `nginx` command allows you to interact with the NGINX binary to check the version, list installed modules, test configurations, and send signals to the master process. NGINX must be running in order for it to serve requests. The `ps` command is a surefire way to determine whether NGINX is running either as a daemon or in the foreground. The default configuration provided by default with

NGINX runs a static site HTTP server on port `80`. You can test this default site by making an HTTP request to the machine at `localhost` as well as the host's IP and hostname.

# 1.5 Key Files, Commands, and Directories

## Problem

You need to understand the important NGINX directories and commands.

## Solution

### NGINX files and directories

*/etc/nginx/*
: The */etc/nginx/* directory is the default configuration root for the NGINX server. Within this directory you will find configuration files that instruct NGINX on how to behave.

*/etc/nginx/nginx.conf*
: The */etc/nginx/nginx.conf* file is the default configuration entry point used by the NGINX service. This configuration file sets up global settings for things like worker process, tuning, logging, loading dynamic modules, and references to other NGINX configuration files. In a default configuration, the */etc/nginx/nginx.conf* file includes the top-level `http` block, which includes all configuration files in the directory described next.

*/etc/nginx/conf.d/*
: The */etc/nginx/conf.d/* directory contains the default HTTP server configuration file. Files in this directory ending in *.conf* are included in the top-level `http` block from within the */etc/nginx/nginx.conf* file. It's best practice to utilize `include` statements and organize your configuration in this way to keep your configuration files concise. In some package repositories, this folder is named *sites-enabled*, and configuration files are linked from a folder named *site-available*; this convention is deprecated.

*/var/log/nginx/*
: The */var/log/nginx/* directory is the default log location for NGINX. Within this directory you will find an *access.log* file and an *error.log* file. The access log contains an entry for each request NGINX serves. The error log file contains error events and debug information if the debug module is enabled.

### NGINX commands

`nginx -h`
: Shows the NGINX help menu.

`nginx -v`
: Shows the NGINX version.

`nginx -V`
: Shows the NGINX version, build information, and configuration arguments, which shows the modules built in to the NGINX binary.

`nginx -t`
: Tests the NGINX configuration.

`nginx -T`
: Tests the NGINX configuration and prints the validated configuration to the screen. This command is useful when seeking support.

`nginx -s signal`
: The `-s` flag sends a signal to the NGINX master process. You can send signals such as `stop`, `quit`, `reload`, and `reopen`. The `stop` signal discontinues the NGINX process immediately. The `quit` signal stops the NGINX process after it finishes processing inflight requests. The `reload` signal reloads the configuration. The `reopen` signal instructs NGINX to reopen log files.

## Discussion

With an understanding of these key files, directories, and commands, you're in a good position to start working with NGINX. With this knowledge, you can alter the default configuration files and test your changes by using the `nginx -t` command. If your test is successful, you also know how to instruct NGINX to reload its configuration using the `nginx -s reload` command.

# 1.6 Serving Static Content

## Problem

You need to serve static content with NGINX.

## Solution

Overwrite the default HTTP server configuration located in */etc/nginx/conf.d/default.conf* with the following NGINX configuration example:

```
server {
    listen 80 default_server;
    server_name www.example.com;

    location / {
        root /usr/share/nginx/html;
        # alias /usr/share/nginx/html;
        index index.html index.htm;
    }
}
```

## Discussion

This configuration serves static files over HTTP on port `80` from the directory */usr/share/nginx/html/*. The first line in this configuration defines a new `server` block. This defines a new context for NGINX to listen for. Line two instructs NGINX to listen on port `80`, and the `default_server` parameter instructs NGINX to use this server as the default context for port `80`. The `server_name` directive defines the hostname or names of which requests should be directed to this server. If the configuration had not defined this context as the `default_server`, NGINX would direct requests to this server only if the HTTP host header matched the value provided to the `server_name` directive.

The `location` block defines a configuration based on the path in the URL. The path, or portion of the URL after the domain, is referred to as the URI. NGINX will best match the URI requested to a `loca` tion block. The example uses / to match all requests. The root directive shows NGINX where to look for static files when serving content for the given context. The URI of the request is appended to the root directive's value when looking for the requested file. If we had provided a URI prefix to the location directive, this would be included in the appended path, unless we used the alias directory rather than root. Lastly, the index directive provides NGINX with a default file, or list of files to check, in the event that no further path is provided in the URI.

# 1.7 Graceful Reload

## Problem

You need to reload your configuration without dropping packets.

## Solution

Use the reload method of NGINX to achieve a graceful reload of the configuration without stopping the server:

```
$ nginx -s reload
```

This example reloads the NGINX system using the NGINX binary to send a signal to the master process.

## Discussion

Reloading the NGINX configuration without stopping the server provides the ability to change configurations on the fly without dropping any packets. In a high-uptime, dynamic environment, you will need to change your load-balancing configuration at some point. NGINX allows you to do this while keeping the load balancer online. This feature enables countless possibilities, such as rerunning configuration management in a live environment, or building an application- and cluster-aware module to dynamically configure and reload NGINX to meet the needs of the environment.

CHAPTER 2

# High-Performance Load Balancing

## 2.0 Introduction

Today's internet user experience demands performance and uptime. To achieve this, multiple copies of the same system are run, and the load is distributed over them. As the load increases, another copy of the system can be brought online. This architecture technique is called *horizontal scaling*. Software-based infrastructure is increasing in popularity because of its flexibility, opening up a vast world of possibilities. Whether the use case is as small as a set of two for high availability or as large as thousands around the globe, there's a need for a load-balancing solution that is as dynamic as the infrastructure. NGINX fills this need in a number of ways, such as HTTP, TCP, and UDP load balancing, which we cover in this chapter.

When balancing load, it's important that the impact to the client is only a positive one. Many modern web architectures employ stateless application tiers, storing state in shared memory or databases. However, this is not the reality for all. Session state is immensely valuable and vast in interactive applications. This state might be stored locally to the application server for a number of reasons; for example, in applications for which the data being worked is so large that network overhead is too expensive in performance. When state is stored locally to an application server, it is extremely important to the user experience that the subsequent requests continue to be delivered to the same server. Another facet of the situation is that servers should not be released until the session has finished. Working with stateful applications at scale requires an intelligent load balancer. NGINX Plus offers multiple ways to solve this problem by tracking cookies or routing. This chapter covers session persistence as it pertains to load balancing with NGINX and NGINX Plus.

Ensuring that the application NGINX is serving is healthy is also important. For a number of reasons, applications fail. It could be because of network connectivity, server failure, or application failure, to name a few. Proxies and load balancers must be smart enough to detect failure of upstream servers and stop passing traffic to them; otherwise, the client will be waiting, only to be delivered a timeout. A way to mitigate service degradation when a server fails is to have the proxy check the health of the upstream servers. NGINX offers two different types of health checks: passive, available in the open source version; and active, available only in NGINX Plus. Active health checks at regular intervals will make a connection or request to the upstream server and can verify that the response is correct. Passive health checks monitor the connection or responses of the upstream server as clients make the request or connection. You might want to use passive health checks to reduce the load of your upstream servers, and you might want to use active health checks to determine failure of an upstream server before a client is served a failure. The tail end of this chapter examines monitoring the health of the upstream application servers for which you're load balancing.

## 2.1 HTTP Load Balancing

### Problem

You need to distribute load between two or more HTTP servers.

### Solution

Use NGINX's HTTP module to load balance over HTTP servers using the `upstream` block:

```
upstream backend {
    server 10.10.12.45:80      weight=1;
    server app.example.com:80  weight=2;
}
server {
    location / {
        proxy_pass http://backend;
    }
}
```

This configuration balances load across two HTTP servers on port `80`. The `weight` parameter instructs NGINX to pass twice as many connections to the second server, and the `weight` parameter defaults to 1.

## Discussion

The HTTP `upstream` module controls the load balancing for HTTP. This module defines a pool of destinations—any combination of Unix sockets, IP addresses, and DNS records, or a mix. The `upstream` module also defines how any individual request is assigned to any of the upstream servers.

Each upstream destination is defined in the upstream pool by the `server` directive. The `server` directive is provided a Unix socket, IP address, or an FQDN, along with a number of optional parameters. The optional parameters give more control over the routing of requests. These parameters include the weight of the server in the balancing algorithm; whether the server is in standby mode, available, or unavailable; and how to determine if the server is unavailable. NGINX Plus provides a number of other convenient parameters like connection limits to the server, advanced DNS resolution control, and the ability to slowly ramp up connections to a server after it starts.

# 2.2 TCP Load Balancing

## Problem

You need to distribute load between two or more TCP servers.

## Solution

Use NGINX's `stream` module to load balance over TCP servers using the `upstream` block:

```
    stream {
        upstream mysql_read {
            server read1.example.com:3306  weight=5;
            server read2.example.com:3306;
            server 10.10.12.34:3306       backup;
        }

        server {
            listen 3306;
            proxy_pass mysql_read;
        }
    }
```

The server block in this example instructs NGINX to listen on TCP port 3306 and balance load between two MySQL database read replicas, and lists another as a backup that will be passed traffic if the primaries are down. This configuration is not to be added to the conf.d folder as that folder is included within an http block; instead, you should create another folder named stream.conf.d, open the stream block in the nginx.conf file, and include the new folder for stream configurations.

## Discussion

TCP load balancing is defined by the NGINX stream module. The stream module, like the HTTP module, allows you to define upstream pools of servers and configure a listening server. When configuring a server to listen on a given port, you must define the port it's to listen on, or optionally, an address and a port. From there, a destination must be configured, whether it be a direct reverse proxy to another address or an upstream pool of resources.

The upstream for TCP load balancing is much like the upstream for HTTP, in that it defines upstream resources as servers, configured with Unix socket, IP, or fully qualified domain name (FQDN), as well as server weight, max number of connections, DNS resolvers, and connection ramp-up periods; and if the server is active, down, or in backup mode.

NGINX Plus offers even more features for TCP load balancing. These advanced features offered in NGINX Plus can be found throughout this book. Health checks for all load balancing will be covered later in this chapter.

# 2.3 UDP Load Balancing

## Problem

You need to distribute load between two or more UDP servers.

## Solution

Use NGINX's `stream` module to load balance over UDP servers using the `upstream` block defined as `udp`:

```
stream {
    upstream ntp {
        server ntp1.example.com:123  weight=2;
        server ntp2.example.com:123;
    }

    server {
        listen 123 udp;
        proxy_pass ntp;
    }
}
```

This section of configuration balances load between two upstream Network Time Protocol (NTP) servers using the UDP protocol. Specifying UDP load balancing is as simple as using the `udp` parameter on the `listen` directive.

If the service you're load balancing over requires multiple packets to be sent back and forth between client and server, you can specify the `reuseport` parameter. Examples of these types of services are OpenVPN, Voice over Internet Protocol (VoIP), virtual desktop solutions, and Datagram Transport Layer Security (DTLS). The following is an example of using NGINX to handle OpenVPN connections and proxy them to the OpenVPN service running locally:

```
stream {
    server {
        listen 1195 udp reuseport;
        proxy_pass 127.0.0.1:1194;
    }
}
```

## Discussion

You might ask, "Why do I need a load balancer when I can have multiple hosts in a DNS A or SRV record?" The answer is that not only are there alternative balancing algorithms with which we can balance, but we can load balance over the DNS servers themselves. UDP services make up a lot of the services that we depend on in networked systems, such as DNS, NTP, and VoIP. UDP load balancing might be less common to some but just as useful in the world of scale.

You can find UDP load balancing in the `stream` module, just like TCP, and configure it mostly in the same way. The main difference is that the `listen` directive specifies that the open socket is for working with datagrams. When working with datagrams, there are some other directives that might apply where they would not in TCP, such as the `proxy_response` directive, which specifies to NGINX how many expected responses can be sent from the upstream server. By default, this is unlimited until the `proxy_timeout` limit is reached.

The `reuseport` parameter instructs NGINX to create an individual listening socket for each worker process. This allows the kernel to distibute incoming connections between worker processes to handle multiple packets being sent between client and server. The `reuseport` feature works only on Linux kernels 3.9 and higher, DragonFly BSD, and FreeBSD 12 and higher.

# 2.4 Load-Balancing Methods

## Problem

Round-robin load balancing doesn't fit your use case because you have heterogeneous workloads or server pools.

## Solution

Use one of NGINX's load-balancing methods such as least connections, least time, generic hash, IP hash, or random:

```
upstream backend {
    least_conn;
    server backend.example.com;
```

```
        server backend1.example.com;
    }
```

This example sets the load-balancing algorithm for the backend upstream pool to be least connections. All load-balancing algorithms, with the exception of generic hash, random, and least-time, are standalone directives, such as the preceding example. The parameters to these directives are explained in the following discussion.

## Discussion

Not all requests or packets carry equal weight. Given this, round robin, or even the weighted round robin used in previous examples, will not fit the need of all applications or traffic flow. NGINX provides a number of load-balancing algorithms that you can use to fit particular use cases. In addition to being able to choose these load-balancing algorithms or methods, you can also configure them. The following load-balancing methods are available for upstream HTTP, TCP, and UDP pools.

*Round robin*

This is the default load-balancing method, which distributes requests in the order of the list of servers in the upstream pool. You can also take weight into consideration for a weighted round robin, which you can use if the capacity of the upstream servers varies. The higher the integer value for the weight, the more favored the server will be in the round robin. The algorithm behind weight is simply statistical probability of a weighted average.

*Least connections*

This method balances load by proxying the current request to the upstream server with the least number of open connections. Least connections, like round robin, also takes weights into account when deciding to which server to send the connection. The directive name is `least_conn`.

*Least time*

Available only in NGINX Plus, least time is akin to least connections in that it proxies to the upstream server with the least number of current connections but favors the servers with the lowest average response times. This method is one of the most sophisticated load-balancing algorithms and fits the needs of highly performant web applications. This algorithm is a value-add over least connections because a small number of connections does not necessarily mean the quickest response. A parameter of `header` or `last_byte` must be specified for this directive. When `header` is specified, the time to receive the response header is used. When `last_byte` is specified, the time to receive the full response is used. The directive name is `least_time`.

*Generic hash*

The administrator defines a hash with the given text, variables of the request or runtime, or both. NGINX distributes the load among the servers by producing a hash for the current request and placing it against the upstream servers. This method is very useful when you need more control over where requests are sent or for determining which upstream server most likely will have the data cached. Note that when a server is added or removed from the pool, the hashed requests will be redistributed. This algorithm has an optional parameter, `consistent`, to minimize the effect of redistribution. The directive name is `hash`.

*Random*

This method is used to instruct NGINX to select a random server from the group, taking server weights into consideration. The optional `two [method]` parameter directs NGINX to randomly select two servers and then use the provided load-balancing method to balance between those two. By default the `least_conn` method is used if `two` is passed without a method. The directive name for random load balancing is `random`.

*IP hash*

This method works only for HTTP. IP hash uses the client IP address as the hash. Slightly different from using the remote variable in a generic hash, this algorithm uses the first three octets of an IPv4 address or the entire IPv6 address. This method ensures that clients are proxied to the same upstream server as long as that server is available, which is extremely helpful when the session state is of concern and not handled by shared memory of the application. This method also takes the weight parameter into consideration when distributing the hash. The directive name is ip_hash.

# 2.5 Sticky Cookie

## Problem

You need to bind a downstream client to an upstream server using NGINX Plus.

## Solution

Use the sticky cookie directive to instruct NGINX Plus to create and track a cookie:

```
upstream backend {
    server backend1.example.com;
    server backend2.example.com;
    sticky cookie
           affinity
           expires=1h
           domain=.example.com
           httponly
           secure
           path=/;
}
```

This configuration creates and tracks a cookie that ties a downstream client to an upstream server. In this example, the cookie is named affinity, is set for example.com, expires in an hour, cannot be consumed client-side, can be sent only over HTTPS, and is valid for all paths.

## Discussion

Using the cookie parameter on the sticky directive creates a cookie on the first request that contains information about the upstream server. NGINX Plus tracks this cookie, enabling it to continue directing subsequent requests to the same server. The first positional parameter to the cookie parameter is the name of the cookie to be created and tracked. Other parameters offer additional control informing the browser of the appropriate usage, like the expiry time, domain, path, and whether the cookie can be consumed client side or whether it can be passed over unsecure protocols.

## 2.6 Sticky Learn

### Problem

You need to bind a downstream client to an upstream server by using an existing cookie with NGINX Plus.

### Solution

Use the `sticky learn` directive to discover and track cookies that are created by the upstream application:

```
upstream backend {
   server backend1.example.com:8080;
   server backend2.example.com:8081;

   sticky learn
          create=$upstream_cookie_cookiename
          lookup=$cookie_cookiename
          zone=client_sessions:2m;
}
```

This example instructs NGINX to look for and track sessions by looking for a cookie named `COOKIENAME` in response headers, and looking up existing sessions by looking for the same cookie on request headers. This session affinity is stored in a shared memory zone of 2 MB that can track approximately 16,000 sessions. The name of the cookie will always be application specific. Commonly used cookie names, such as `jsessionid` or `phpsessionid`, are typically defaults set within the application or the application server configuration.

### Discussion

When applications create their own session-state cookies, NGINX Plus can discover them in request responses and track them. This type of cookie tracking is performed when the `sticky` directive is provided the `learn` parameter. Shared memory for tracking cookies is specified with the `zone` parameter, with a name and size. NGINX Plus is directed to look for cookies in the response from the upstream server via specification of the `create` parameter, and searches for prior registered server affinity using the `lookup` parameter. The value of these parameters are variables exposed by the HTTP module.

# 2.7 Sticky Routing

## Problem

You need granular control over how your persistent sessions are routed to the upstream server with NGINX Plus.

## Solution

Use the sticky directive with the route parameter to use variables about the request to route:

```
map $cookie_jsessionid $route_cookie {
    ~.+\.(?P<route>\w+)$ $route;
}

map $request_uri $route_uri {
    ~jsessionid=.+\.(?P<route>\w+)$ $route;
}

upstream backend {
    server backend1.example.com route=a;
    server backend2.example.com route=b;

    sticky route $route_cookie $route_uri;
}
```

This example attempts to extract a Java session ID, first from a cookie by mapping the value of the Java session ID cookie to a variable with the first map block, and second by looking into the request URI for a parameter called jsessionid, mapping the value to a variable using the second map block. The sticky directive with the route parameter is passed any number of variables. The first nonzero or nonempty value is used for the route. If a jsessionid cookie is used, the request is routed to backend1; if a URI parameter is used, the request is routed to backend2. Although this example is based on the Java common session ID, the same applies for other session technology like phpsessionid, or any guaranteed unique identifier your application generates for the session ID.

## Discussion

Sometimes, you might want to direct traffic to a particular server with a bit more granular control. The `route` parameter to the `sticky` directive is built to achieve this goal. Sticky route gives you better control, actual tracking, and stickiness, as opposed to the generic hash load-balancing algorithm. The client is first routed to an upstream server based on the route specified, and then subsequent requests will carry the routing information in a cookie or the URI. Sticky route takes a number of positional parameters that are evaluated. The first nonempty variable is used to route to a server. Map blocks can be used to selectively parse variables and save them as other variables to be used in the routing. Essentially, the `sticky route` directive creates a session within the NGINX Plus shared memory zone for tracking any client session identifier you specify to the upstream server, consistently delivering requests with this session identifier to the same upstream server as its original request.

# 2.8 Connection Draining

## Problem

You need to gracefully remove servers for maintenance or other reasons while still serving sessions with NGINX Plus.

## Solution

Use the `drain` parameter through the NGINX Plus API, described in more detail in Chapter 5, to instruct NGINX to stop sending new connections that are not already tracked:

```
$ curl -X POST -d '{"drain":true}' \
  'http://nginx.local/api/3/http/upstreams/backend/servers/0'

{
  "id":0,
  "server":"172.17.0.3:80",
  "weight":1,
  "max_conns":0,
  "max_fails":1,
  "fail_timeout":
  "10s","slow_start":
  "0s",
  "route":"",
  "backup":false,
  "down":false,
  "drain":true
}
```

## Discussion

When session state is stored locally to a server, connections and persistent sessions must be drained before it's removed from the pool. Draining connections is the process of letting sessions to a server expire natively before removing the server from the upstream pool. You can configure draining for a particular server by adding the `drain` parameter to the `server` directive. When the `drain` parameter is set, NGINX Plus stops sending new sessions to this server but allows current sessions to continue being served for the length of their session. You can also toggle this configuration by adding the `drain` parameter to an upstream server directive.

# 2.9 Passive Health Checks

## Problem

You need to passively check the health of upstream servers.

## Solution

Use NGINX health checks with load balancing to ensure that only healthy upstream servers are utilized:

```
upstream backend {
    server backend1.example.com:1234 max_fails=3 fail_timeout=3s;
    server backend2.example.com:1234 max_fails=3 fail_timeout=3s;
}
```

This configuration passively monitors the upstream health, setting the `max_fails` directive to three, and `fail_timeout` to three seconds. These directive parameters work the same way in both stream and HTTP servers.

## Discussion

Passive health checking is available in the Open Source version of NGINX. Passive monitoring watches for failed or timed-out connections as they pass through NGINX as requested by a client. Passive health checks are enabled by default; the parameters mentioned here allow you to tweak their behavior. Monitoring for health is important on all types of load balancing, not only from a user experience standpoint, but also for business continuity. NGINX passively monitors upstream HTTP, TCP, and UDP servers to ensure that they're healthy and performing.

# 2.10 Active Health Checks

## Problem

You need to actively check your upstream servers for health with NGINX Plus.

## Solution

For HTTP, use the `health_check` directive in a location block:

```
http {
    server {
        ...
        location / {
            proxy_pass http://backend;
            health_check interval=2s
                fails=2
                passes=5
                uri=/
                match=welcome;
        }
    }
    # status is 200, content type is "text/html",
    # and body contains "Welcome to nginx!"
    match welcome {
        status 200;
        header Content-Type = text/html;
        body ~ "Welcome to nginx!";
    }
}
```

This health check configuration for HTTP servers checks the health of the upstream servers by making an HTTP request to the URI '/' every two seconds. The upstream servers must pass five consecutive health checks to be considered healthy. They are considered unhealthy if they fail two consecutive checks. The response from the upstream server must match the defined match block, which defines the status code as `200`, the header `Content-Type` value as `'text/`

html', and the string "Welcome to nginx!" in the response body. The HTTP match block has three directives: status, header, and body. All three of these directives have comparison flags, as well.

Stream health checks for TCP/UDP services are very similar:

```
stream {
    ...
    server {
        listen 1234;
        proxy_pass stream_backend;
        health_check interval=10s
            passes=2
            fails=3;
        health_check_timeout 5s;
    }
    ...
}
```

In this example, a TCP server is configured to listen on port 1234, and to proxy to an upstream set of servers, for which it actively checks for health. The stream health_check directive takes all the same parameters as in HTTP with the exception of uri, and the stream version has a parameter to switch the check protocol to udp. In this example, the interval is set to 10 seconds, requires two passes to be considered healthy, and three fails to be considered unhealthy. The active-stream health check is also able to verify the response from the upstream server. The match block for stream servers, however, has just two directives: send and expect. The send directive is raw data to be sent, and expect is an exact response or a regular expression to match.

## Discussion

Active health checks in NGINX Plus continually make requests to the source servers to check their health. These health checks can measure more than just the response code. In NGINX Plus, active HTTP health checks monitor based on a number of acceptance criteria of the response from the upstream server. You can configure active health-check monitoring for how often upstream servers are checked, how many times a server must pass this check to be considered healthy, how many times it can fail before being deemed unhealthy, and what the expected result should be. The match parameter points to a match block that defines the acceptance criteria for the response. The match block also defines the data to send to the upstream server when used in the stream context for TCP/UPD. These features enable NGINX to ensure that upstream servers are healthy at all times.

## 2.11 Slow Start

### Problem

Your application needs to ramp up before taking on full production load.

### Solution

Use the `slow_start` parameter on the `server` directive to gradually increase the number of connections over a specified time as a server is reintroduced to the upstream load-balancing pool:

```
upstream {
    zone backend 64k;

    server server1.example.com slow_start=20s;
    server server2.example.com slow_start=15s;
}
```

The `server` directive configurations will slowly ramp up traffic to the upstream servers after they're reintroduced to the pool. `server1` will slowly ramp up its number of connections over 20 seconds, and `server2` over 15 seconds.

### Discussion

*Slow start* is the concept of slowly ramping up the number of requests proxied to a server over a period of time. Slow start allows the application to warm up by populating caches, initiating database connections without being overwhelmed by connections as soon as it starts. This feature takes effect when a server that has failed health checks begins to pass again and re-enters the load-balancing pool.

# 2.12 TCP Health Checks

## Problem

You need to check your upstream TCP server for health and remove unhealthy servers from the pool.

## Solution

Use the `health_check` directive in the server block for an active health check:

```
stream {
    server {
        listen       3306;
        proxy_pass   read_backend;
        health_check interval=10 passes=2 fails=3;
    }
}
```

The example monitors the upstream servers actively. The upstream server will be considered unhealthy if it fails to respond to three or more TCP connections initiated by NGINX. NGINX performs the check every 10 seconds. The server will only be considered healthy after passing two health checks.

## Discussion

TCP health can be verified by NGINX Plus either passively or actively. Passive health monitoring is done by noting the communication between the client and the upstream server. If the upstream server is timing out or rejecting connections, a passive health check will deem that server unhealthy. Active health checks will initiate their own configurable checks to determine health. Active health checks not only test a connection to the upstream server, but can expect a given response.

CHAPTER 3

# Traffic Management

## 3.0 Introduction

NGINX and NGINX Plus are also classified as web traffic controllers. You can use NGINX to intellengently route traffic and control flow based on many attributes. This chapter covers NGINX's ability to split client requests based on percentages, utilize geographical location of the clients, and control the flow of traffic in the form of rate, connection, and bandwidth limiting. As you read through this chapter, keep in mind that you can mix and match these features to enable countless possibilities.

## 3.1 A/B Testing

### Problem

You need to split clients between two or more versions of a file or application to test acceptance.

### Solution

Use the `split_clients` module to direct a percentage of your clients to a different upstream pool:

```
split_clients "${remote_addr}AAA" $variant {
    20.0%    "backendv2";
    *        "backendv1";
}
```

The split_clients directive hashes the string provided by you as the first parameter and divides that hash by the percentages provided to map the value of a variable provided as the second parameter. The third parameter is an object containing key-value pairs where the key is the percentage weight and the value is the value to be assigned. The key can be either a percentage or an asterisk. The asterisk denotes the rest of the whole after all percentages are taken. The value of the $variant variable will be backendv2 for 20% of client IP addresses and backendv1 for the remaining 80%.

In this example, backendv1 and backendv2 represent upstream server pools and can be used with the proxy_pass directive as such:

```
location / {
    proxy_pass http://$variant
}
```

Using the variable $variant, our traffic will split between two different application server pools.

## Discussion

This type of A/B testing is useful when testing different types of marketing and frontend features for conversion rates on ecommerce sites. It's common for applications to use a type of deployment called canary release. In this type of deployment, traffic is slowly switched over to the new version. Splitting your clients between different versions of your application can be useful when rolling out new versions of code, to limit the blast radius in case of an error. Whatever the reason for splitting clients between two different application sets, NGINX makes this simple through the use of this split_clients module.

## Also See

split_client Documentation

# 3.2 Using the GeoIP Module and Database

## Problem

You need to install the GeoIP database and enable its embedded variables within NGINX to log and specify to your application the location of your clients.

## Solution

The official NGINX Open Source package repository, configured in Chapter 1 when installing NGINX, provides a package named `nginx-module-geoip`. When using the NGINX Plus package repository, this package is named `nginx-plus-module-geoip`. These packages install the dynamic version of the GeoIP module.

RHEL/CentOS NGINX Open Source:

```
# yum install nginx-module-geoip
```

Debian/Ubuntu NGINX Open Source:

```
# apt-get install nginx-module-geoip
```

RHEL/CentOS NGINX Plus:

```
# yum install nginx-plus-module-geoip
```

Debian/Ubuntu NGINX Plus:

```
# apt-get install nginx-plus-module-geoip
```

Download the GeoIP country and city databases and unzip them:

```
# mkdir /etc/nginx/geoip
# cd /etc/nginx/geoip
# wget "http://geolite.maxmind.com/\
download/geoip/database/GeoLiteCountry/GeoIP.dat.gz"
# gunzip GeoIP.dat.gz
# wget "http://geolite.maxmind.com/\
download/geoip/database/GeoLiteCity.dat.gz"
# gunzip GeoLiteCity.dat.gz
```

This set of commands creates a *geoip* directory in the */etc/nginx* directory, moves to this new directory, and downloads and unzips the packages.

With the GeoIP database for countries and cities on the local disk, you can now instruct the NGINX GeoIP module to use them to expose embedded variables based on the client IP address:

```
load_module "/usr/lib64/nginx/modules/ngx_http_geoip_module.so";

http {
    geoip_country /etc/nginx/geoip/GeoIP.dat;
    geoip_city /etc/nginx/geoip/GeoLiteCity.dat;
...
}
```

The `load_module` directive dynamically loads the module from its path on the filesystem. The `load_module` directive is only valid in the main context. The `geoip_country` directive takes a path to the *GeoIP.dat* file containing the database mapping IP addresses to country codes and is valid only in the HTTP context.

## Discussion

The `geoip_country` and `geoip_city` directives expose a number of embedded variables available in this module. The `geoip_country` directive enables variables that allow you to distinguish the country of origin of your client. These variables include `$geoip_country_code`, `$geoip_country_code3`, and `$geoip_country_name`. The country code variable returns the two-letter country code, and the variable with a `3` at the end returns the three-letter country code. The country name variable returns the full name of the country.

The `geoip_city` directive enables quite a few variables. The `geoip_city` directive enables all the same variables as the `geoip_country` directive, just with different names, such as `$geoip_city_country_code`, `$geoip_city_country_code3`, and `$geoip_city_country_name`. Other variables include `$geoip_city`, `$geoip_city_continent_code`, `$geoip_latitude`, `$geoip_longitude`, and `$geoip_postal_code`, all of which are descriptive of the value they return. `$geoip_region` and `$geoip_region_name` describe the region, territory, state, province, federal land, and the like. Region is the two-letter code, where region name is the full name. `$geoip_area_code`, only valid in the US, returns the three-digit telephone area code.

With these variables, you're able to log information about your client. You could optionally pass this information to your application as a header or variable, or use NGINX to route your traffic in particular ways.

## Also See

GeoIP Update

## 3.3 Restricting Access Based on Country

### Problem

You need to restrict access from particular countries for contractual or application requirements.

### Solution

Map the country codes you want to block or allow to a variable:

```
load_module
  "/usr/lib64/nginx/modules/ngx_http_geoip_module.so";

http {
    map $geoip_country_code $country_access {
        "US"    0;
        "RU"    0;
        default 1;
    }
    ...
}
```

This mapping will set a new variable `$country_access` to a `1` or a `0`. If the client IP address originates from the US or Russia, the variable will be set to a `0`. For any other country, the variable will be set to a `1`.

Now, within our `server` block, we'll use an `if` statement to deny access to anyone not originating from the US or Russia:

```
server {
    if ($country_access = '1') {
      return 403;
    }
    ...
}
```

This `if` statement will evaluate `True` if the `$country_access` variable is set to `1`. When `True`, the server will return a 403 unauthorized. Otherwise the server operates as normal. So this `if` block is only there to deny people who are not from the US or Russia.

### Discussion

This is a short but simple example of how to only allow access from a couple of countries. This example can be expounded upon to fit your needs. You can utilize this same practice to allow or block based on any of the embedded variables made available from the GeoIP module.

# 3.4 Finding the Original Client

## Problem

You need to find the original client IP address because there are proxies in front of the NGINX server.

## Solution

Use the geoip_proxy directive to define your proxy IP address range and the geoip_proxy_recursive directive to look for the original IP:

```
load_module "/usr/lib64/nginx/modules/ngx_http_geoip_module.so";

http {
    geoip_country /etc/nginx/geoip/GeoIP.dat;
    geoip_city /etc/nginx/geoip/GeoLiteCity.dat;
    geoip_proxy 10.0.16.0/26;
    geoip_proxy_recursive on;
...
}
```

The geoip_proxy directive defines a CIDR range in which our proxy servers live and instructs NGINX to utilize the X-Forwarded-For header to find the client IP address. The geoip_proxy_recursive directive instructs NGINX to recursively look through the X-Forwarded-For header for the last client IP known.

## Discussion

You may find that if you're using a proxy in front of NGINX, NGINX will pick up the proxy's IP address rather than the client's. For this you can use the geoip_proxy directive to instruct NGINX to use the X-Forwarded-For header when connections are opened from a given range. The geoip_proxy directive takes an address or a CIDR range. When there are multiple proxies passing traffic in front of NGINX, you can use the geoip_proxy_recursive directive to recursively search through X-Forwarded-For addresses to find the originating client. You will want to use something like this when utilizing load balancers such as AWS ELB, Google's load balancer, or Azure's load balancer in front of NGINX.

# 3.5 Limiting Connections

## Problem

You need to limit the number of connections based on a predefined key, such as the client's IP address.

## Solution

Construct a shared memory zone to hold connection metrics, and use the `limit_conn` directive to limit open connections:

```
http {
    limit_conn_zone $binary_remote_addr zone=limitbyaddr:10m;
    limit_conn_status 429;
    ...
    server {
        ...
            limit_conn limitbyaddr 40;
        ...
    }
}
```

This configuration creates a shared memory zone named `limitbyaddr`. The predefined key used is the client's IP address in binary form. The size of the shared memory zone is set to 10 megabytes. The `limit_conn` directive takes two parameters: a `limit_conn_zone` name, and the number of connections allowed. The `limit_conn_status` sets the response when the connections are limited to a status of 429, indicating too many requests. The `limit_conn` and `limit_conn_status` directives are valid in the HTTP, server, and location context.

## Discussion

Limiting the number of connections based on a key can be used to defend against abuse and share your resources fairly across all your clients. It is important to be cautious with your predefined key. Using an IP address, as we are in the previous example, could be dangerous if many users are on the same network that originates from the same IP, such as when behind a *Network Address Transla-*

*tion* (NAT). The entire group of clients will be limited. The `limit_conn_zone` directive is only valid in the HTTP context. You can utilize any number of variables available to NGINX within the HTTP context in order to build a string on which to limit by. Utilizing a variable that can identify the user at the application level, such as a session cookie, may be a cleaner solution depending on the use case. The `limit_conn_status` defaults to 503, service unavailable. You may find it preferable to use a 429, as the service is available, and 500-level responses indicate server error whereas 400-level responses indicate client error.

# 3.6 Limiting Rate

## Problem

You need to limit the rate of requests by a predefined key, such as the client's IP address.

## Solution

Utilize the rate-limiting module to limit the rate of requests:

```
http {
   limit_req_zone $binary_remote_addr
       zone=limitbyaddr:10m rate=1r/s;
   limit_req_status 429;
   ...
   server {
       ...
           limit_req zone=limitbyaddr burst=10 nodelay;
       ...
   }
}
```

This example configuration creates a shared memory zone named `limitbyaddr`. The predefined key used is the client's IP address in binary form. The size of the shared memory zone is set to 10 megabytes. The zone sets the rate with a keyword argument. The `limit_req` directive takes two optional keyword arguments: `zone` and `burst`. `zone` is required to instruct the directive on which shared memory request limit zone to use. When the request rate for a given zone is exceeded, requests are delayed until their maximum burst size is reached, denoted by the `burst` keyword argument. The `burst` keyword argument defaults to zero. `limit_req` also takes a third optional parameter, nodelay. This parameter enables the client to use its burst without delay before being limited. limit_req_status sets the status returned to the client to a particular HTTP status code; the default is 503. limit_req_status and limit_req are valid in the context of HTTP, server, and location. limit_req_zone is only valid in the HTTP context. Rate limiting is cluster-aware in NGINX Plus, new in version R16.

## Discussion

The rate-limiting module is very powerful for protecting against abusive rapid requests while still providing a quality service to everyone. There are many reasons to limit rate of request, one being security. You can deny a brute-force attack by putting a very strict limit on your login page. You can set a sane limit on all requests, thereby disabling the plans of malicious users who might try to deny service to your application or to waste resources. The configuration of the rate-limit module is much like the preceding connection-limiting module described in Recipe 3.5, and much of the same concerns apply. You can specify the rate at which requests are limited in requests per second or requests per minute. When the rate limit is reached, the incident is logged. There's also a directive not in the example, limit_req_log_level, which defaults to error, but can be set to info, notice, or warn. New in NGINX Plus, version R16 rate limiting is now cluster-aware (see Recipe 12.5 for a zone sync example).

# 3.7 Limiting Bandwidth

## Problem

You need to limit download bandwidth per client for your assets.

## Solution

Utilize NGINX's limit_rate and limit_rate_after directives to limit the rate of response to a client:

```
location /download/ {
    limit_rate_after 10m;
    limit_rate 1m;
}
```

The configuration of this location block specifies that for URIs with the prefix *download*, the rate at which the response will be served to the client will be limited after 10 megabytes to a rate of 1 megabyte per second. The bandwidth limit is per connection, so you may want to institute a connection limit as well as a bandwidth limit where applicable.

## Discussion

Limiting the bandwidth for particular connections enables NGINX to share its upload bandwidth across all of the clients in a manner you specify. These two directives do it all: `limit_rate_after` and `limit_rate`. The `limit_rate_after` directive can be set in almost any context: HTTP, server, location, and `if` when the `if` is within a location. The `limit_rate` directive is applicable in the same contexts as `limit_rate_after`; however, it can alternatively be set by setting a variable named `$limit_rate`. The `limit_rate_after` directive specifies that the connection should not be rate limited until after a specified amount of data has been transferred. The `limit_rate` directive specifies the rate limit for a given context in bytes per second by default. However, you can specify `m` for megabytes or `g` for gigabytes. Both directives default to a value of 0. The value 0 means not to limit download rates at all. This module allows you to programmatically change the rate limit of clients.