O'REILLY®

Compliments of NGINX

NGINX Cookbook

Advanced Recipes for High Performance Load Balancing

Derek DeJonghe

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

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` param-

eter. 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.

CHAPTER 4

Massively Scalable Content Caching

4.0 Introduction

Caching accelerates content serving by storing request responses to be served again in the future. Content caching reduces load to upstream servers, caching the full response rather than running computations and queries again for the same request. Caching increases performance and reduces load, meaning you can serve faster with fewer resources. Scaling and distributing caching servers in strategic locations can have a dramatic effect on user experience. It's optimal to host content close to the consumer for the best performance. You can also cache your content close to your users. This is the pattern of content delivery networks, or CDNs. With NGINX you're able to cache your content wherever you can place an NGINX server, effectively enabling you to create your own CDN. With NGINX caching, you're also able to passively cache and serve cached responses in the event of an upstream failure.

4.1 Caching Zones

Problem

You need to cache content and need to define where the cache is stored.

Solution

Use the `proxy_cache_path` directive to define shared memory cache zones and a location for the content:

```
proxy_cache_path /var/nginx/cache
                 keys_zone=CACHE:60m
                 levels=1:2
                 inactive=3h
                 max_size=20g;
proxy_cache CACHE;
```

The cache definition example creates a directory for cached responses on the filesystem at */var/nginx/cache* and creates a shared memory space named `CACHE` with 60 megabytes of memory. This example sets the directory structure levels, defines the release of cached responses after they have not been requested in 3 hours, and defines a maximum size of the cache of 20 gigabytes. The `proxy_cache` directive informs a particular context to use the cache zone. The `proxy_cache_path` is valid in the HTTP context, and the `proxy_cache` directive is valid in the HTTP, server, and location contexts.

Discussion

To configure caching in NGINX, it's necessary to declare a path and zone to be used. A cache zone in NGINX is created with the directive `proxy_cache_path`. The `proxy_cache_path` designates a location to store the cached information and a shared memory space to store active keys and response metadata. Optional parameters to this directive provide more control over how the cache is maintained and accessed. The `levels` parameter defines how the file structure is created. The value is a colon-separated value that declares the length of subdirectory names, with a maximum of three levels. NGINX caches based on the cache key, which is a hashed value. NGINX then stores the result in the file structure provided, using the cache key as a file path and breaking up directories based on the `levels` value. The `inactive` parameter allows for control over the length of time a cache item will be hosted after its last use. The size of the cache is also configurable with the use of the `max_size` parameter. Other parameters relate to the cache-loading process, which loads the cache keys into the shared memory zone from the files cached on disk.

4.2 Caching Hash Keys

Problem

You need to control how your content is cached and looked up.

Solution

Use the `proxy_cache_key` directive along with variables to define what constitutes a cache hit or miss:

```
proxy_cache_key "$host$request_uri $cookie_user";
```

This cache hash key will instruct NGINX to cache pages based on the host and URI being requested, as well as a cookie that defines the user. With this you can cache dynamic pages without serving content that was generated for a different user.

Discussion

The default `proxy_cache_key`, which will fit most use cases, is `"$scheme$proxy_host$request_uri"`. The variables used include the scheme, HTTP or HTTPS, the `proxy_host`, where the request is being sent, and the request URI. All together, this reflects the URL that NGINX is proxying the request to. You may find that there are many other factors that define a unique request per application, such as request arguments, headers, session identifiers, and so on, to which you'll want to create your own hash key.[1]

Selecting a good hash key is very important and should be thought through with understanding of the application. Selecting a cache key for static content is typically pretty straightforward; using the hostname and URI will suffice. Selecting a cache key for fairly dynamic content like pages for a dashboard application requires more knowledge around how users interact with the application and the degree of variance between user experiences. Due to security concerns you may not want to present cached data from one user to another without fully understanding the context. The `proxy_cache_key` directive configures the string to be hashed for the cache key. The

1 Any combination of text or variables exposed to NGINX can be used to form a cache key. A list of variables is available in NGINX: *http://nginx.org/en/docs/varindex.html*.

`proxy_cache_key` can be set in the context of HTTP, server, and location blocks, providing flexible control on how requests are cached.

4.3 Cache Bypass

Problem

You need the ability to bypass the caching.

Solution

Use the `proxy_cache_bypass` directive with a nonempty or nonzero value. One way to do this is by setting a variable within location blocks that you do not want cached to equal 1:

```
proxy_cache_bypass $http_cache_bypass;
```

The configuration tells NGINX to bypass the cache if the HTTP request header named `cache_bypass` is set to any value that is not `0`.

Discussion

There are a number of scenarios that demand that the request is not cached. For this, NGINX exposes a `proxy_cache_bypass` directive so that when the value is nonempty or nonzero, the request will be sent to an upstream server rather than be pulled from the cache. Different needs and scenarios for bypassing cache will be dictated by your applications use case. Techniques for bypassing cache can be as simple as a using a request or response header, or as intricate as multiple map blocks working together.

For many reasons, you may want to bypass the cache. One important reason is troubleshooting and debugging. Reproducing issues can be hard if you're consistently pulling cached pages or if your cache key is specific to a user identifier. Having the ability to bypass the cache is vital. Options include but are not limited to bypassing the cache when a particular cookie, header, or request argument is set. You can also turn off the cache completely for a given context such as a location block by setting `proxy_cache off;`.

4.4 Cache Performance

Problem

You need to increase performance by caching on the client side.

Solution

Use client-side cache control headers:

```
location ~* \.(css|js)$ {
  expires 1y;
  add_header Cache-Control "public";
}
```

This location block specifies that the client can cache the content of CSS and JavaScript files. The `expires` directive instructs the client that their cached resource will no longer be valid after one year. The `add_header` directive adds the HTTP response header `Cache-Control` to the response, with a value of `public`, which allows any caching server along the way to cache the resource. If we specify private, only the client is allowed to cache the value.

Discussion

Cache performance has many factors, disk speed being high on the list. There are many things within the NGINX configuration you can do to assist with cache performance. One option is to set headers of the response in such a way that the client actually caches the response and does not make the request to NGINX at all, but simply serves it from its own cache.

4.5 Purging

Problem

You need to invalidate an object from the cache.

Solution

Use the purge feature of NGINX Plus, the `proxy_cache_purge` directive, and a nonempty or zero-value variable:

```
map $request_method $purge_method {
    PURGE 1;
    default 0;
}
server {
    ...
    location / {
        ...
        proxy_cache_purge $purge_method;
    }
}
```

In this example, the cache for a particular object will be purged if it's requested with a method of `PURGE`. The following is a `curl` example of purging the cache of a file named `main.js`:

```
$ curl -XPURGE localhost/main.js
```

Discussion

A common way to handle static files is to put a hash of the file in the filename. This ensures that as you roll out new code and content, your CDN recognizes it as a new file because the URI has changed. However, this does not exactly work for dynamic content to which you've set cache keys that don't fit this model. In every caching scenario, you must have a way to purge the cache. NGINX Plus has provided a simple method of purging cached responses. The `proxy_cache_purge` directive, when passed a nonzero or nonempty value, will purge the cached items matching the request. A simple way to set up purging is by mapping the request method for `PURGE`. However, you may want to use this in conjunction with the `geo_ip` module or simple authentication to ensure that not anyone can purge your precious cache items. NGINX has also allowed for the use of *, which will purge cache items that match a common URI prefix. To use wildcards you will need to configure your `proxy_cache_path` directive with the `purger=on` argument.

4.6 Cache Slicing

Problem

You need to increase caching effiency by segmenting the file into fragments.

Solution

Use the NGINX `slice` directive and its embedded variables to divide the cache result into fragments:

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;
server {
    ...
    proxy_cache mycache;
    slice 1m;
    proxy_cache_key $host$uri$is_args$args$slice_range;
    proxy_set_header Range $slice_range;
    proxy_http_version 1.1;
    proxy_cache_valid  200 206 1h;

    location / {
        proxy_pass http://origin:80;
    }
}
```

Discussion

This configuration defines a cache zone and enables it for the server. The `slice` directive is then used to instruct NGINX to slice the response into 1 MB file segments. The cache files are stored according to the `proxy_cache_key` directive. Note the use of the embedded variable named `slice_range`. That same variable is used as a header when making the request to the origin, and that request HTTP version is upgraded to HTTP/1.1 because 1.0 does not support byte-range requests. The cache validity is set for response codes of `200` or `206` for one hour, and then the location and origins are defined.

The Cache Slice module was developed for delivery of HTML5 video, which uses byte-range requests to pseudostream content to the browser. By default, NGINX is able to serve byte-range requests from its cache. If a request for a byte-range is made for uncached content, NGINX requests the entire file from the origin. When you use the Cache Slice module, NGINX requests only the necessary segments from the origin. Range requests that are larger than the slice size, including the entire file, trigger subrequests for each of the required segments, and then those segments are cached. When all of the segments are cached, the response is assembled and sent to the client, enabling NGINX to more efficiently cache and serve content requested in ranges. The Cache Slice module should be used only on large files that do not change. NGINX validates the ETag each time

it receives a segment from the origin. If the ETag on the origin changes, NGINX aborts the transaction because the cache is no longer valid. If the content does change and the file is smaller or your origin can handle load spikes during the cache fill process, it's better to use the Cache Lock module described in the blog listed in the following Also See section.

Also See

Smart and Efficient Byte-Range Caching with NGINX & NGINX Plus

CHAPTER 5

Programmability and Automation

5.0 Introduction

Programmability refers to the ability to interact with something through programming. The API for NGINX Plus provides just that: the ability to interact with the configuation and behavior of NGINX Plus through an HTTP interface. This API provides the ability to reconfigure NGINX Plus by adding or removing upstream servers through HTTP requests. The key-value store feature in NGINX Plus enables another level of dynamic configuration—you can utilize HTTP calls to inject information that NGINX Plus can use to route or control traffic dynamically. This chapter will touch on the NGINX Plus API and the key-value store module exposed by that same API.

Configuration management tools automate the installation and configuration of servers, which is an invaluable utility in the age of the cloud. Engineers of large-scale web applications no longer need to configure servers by hand; instead, they can use one of the many configuration management tools available. With these tools, engineers can write configurations and code one time to produce many servers with the same configuration in a repeatable, testable, and modular fashion. This chapter covers a few of the most popular configuration management tools available and how to use them to install NGINX and template a base configuration. These examples are extremely basic but demonstrate how to get an NGINX server started with each platform.

5.1 NGINX Plus API

Problem

You have a dynamic environment and need to reconfigure NGINX Plus on the fly.

Solution

Configure the NGINX Plus API to enable adding and removing servers through API calls:

```
upstream backend {
    zone http_backend 64k;
}
server {
    # ...
    location /api {
        api [write=on];
        # Directives limiting access to the API
        # See chapter 7
    }

    location = /dashboard.html {
        root   /usr/share/nginx/html;
    }
}
```

This NGINX Plus configuration creates an upstream server with a shared memory zone, enables the API in the `/api` location block, and provides a location for the NGINX Plus dashboard.

You can utilize the API to add servers when they come online:

```
$ curl -X POST -d '{"server":"172.17.0.3"}' \
  'http://nginx.local/api/3/http/upstreams/backend/servers/'

{
  "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
}
```

The `curl` call in this example makes a request to NGINX Plus to add a new server to the backend upstream configuration. The HTTP method is a POST, and a JSON object is passed as the body. The NGINX Plus API is RESTful; therefore, there are parameters in the request URI. The format of the URI is as follows:

```
/api/{version}/http/upstreams/{httpUpstreamName}/servers/
```

You can utilize the NGINX Plus API to list the servers in the upstream pool:

```
$ curl 'http://nginx.local/api/3/http/upstreams/backend/servers/'
[
  {
    "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
  }
]
```

The `curl` call in this example makes a request to NGINX Plus to list all of the servers in the upstream pool named backend. Currently, we have only the one server that we added in the previous `curl` call to the API. The request will return a upstream server object that contains all of the configurable options for a server.

Use the NGINX Plus API to drain connections from an upstream server, preparing it for a graceful removal from the upstream pool. You can find details about connection draining in Recipe 2.8:

```
$ curl -X PATCH -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
}
```

In this `curl`, we specify that the request method is `PATCH`, we pass a JSON body instructing it to drain connections for the server, and specify the server ID by appending it to the URI. We found the ID of the server by listing the servers in the upstream pool in the previous `curl` command.

NGINX Plus will begin to drain the connections. This process can take as long as the length of the sessions of the application. To check in on how many active connections are being served by the server you've begun to drain, use the following call and look for the `active` attribute of the server being drained:

```
$ curl 'http://nginx.local/api/3/http/upstreams/backend'
{
   "zone" : "http_backend",
   "keepalive" : 0,
   "peers" : [
      {
         "backup" : false,
         "id" : 0,
         "unavail" : 0,
         "name" : "172.17.0.3",
         "requests" : 0,
         "received" : 0,
         "state" : "draining",
         "server" : "172.17.0.3:80",
         "active" : 0,
         "weight" : 1,
         "fails" : 0,
         "sent" : 0,
         "responses" : {
            "4xx" : 0,
            "total" : 0,
            "3xx" : 0,
            "5xx" : 0,
            "2xx" : 0,
            "1xx" : 0
         },
         "health_checks" : {
            "checks" : 0,
            "unhealthy" : 0,
            "fails" : 0
         },
         "downtime" : 0
      }
```

```
    ],
    "zombies" : 0
}
```

After all connections have drained, utilize the NGINX Plus API to remove the server from the upstream pool entirely:

```
$ curl -X DELETE \
  'http://nginx.local/api/3/http/upstreams/backend/servers/0'
[]
```

The `curl` command makes a `DELETE` method request to the same URI used to update the servers' state. The `DELETE` method instructs NGINX to remove the server. This API call returns all of the servers and their IDs that are still left in the pool. Because we started with an empty pool, added only one server through the API, drained it, and then removed it, we now have an empty pool again.

Discussion

The NGINX Plus exclusive API enables dynamic application servers to add and remove themselves to the NGINX configuration on the fly. As servers come online, they can register themselves to the pool, and NGINX will start sending it load. When a server needs to be removed, the server can request NGINX Plus to drain its connections, and then remove itself from the upstream pool before it's shut down. This enables the infrastructure, through some automation, to scale in and out without human intervention.

Also See

NGINX Plus API Swagger Documentation

5.2 Key-Value Store

Problem

You need NGINX Plus to make dynamic traffic management decisions based on input from applications.

Solution

Set up the cluster-aware key-value store and API, and then add keys and values:

```
keyval_zone zone=blacklist:1M;
keyval $remote_addr $num_failures zone=blacklist;

server {
    # ...
    location / {
        if ($num_failures) {
            return 403 'Forbidden';
        }
        return 200 'OK';
    }
}
server {
    # ...
    # Directives limiting access to the API
    # See chapter 6
    location /api {
        api write=on;
    }
}
```

This NGINX Plus configuration uses the `keyval_zone` directory to build a key-value store shared memory zone named `blacklist` and sets a memory limit of 1 MB. The `keyval` directive then maps the value of the key matching the first parameter `$remote_addr` to a new variable named `$num_failures` from the zone. This new variable is then used to determine whether NGINX Plus should serve the request or return a 403 Forbidden code.

After starting the NGINX Plus server with this configuration, you can `curl` the local machine and expect to receive a 200 OK response.

```
$ curl 'http://127.0.0.1/'
OK
```

Now add the local machine's IP address to the key-value store with a value of `1`:

```
$ curl -X POST -d '{"127.0.0.1":"1"}' \
  'http://127.0.0.1/api/3/http/keyvals/blacklist'
```

This `curl` command submits an HTTP POST request with a JSON object containing a key-value object to be submitted to the blacklist shared memory zone. The key-value store API URI is formatted as follows:

```
/api/{version}/http/keyvals/{httpKeyvalZoneName}
```

The local machine's IP address is now added to the key-value zone named `blacklist` with a value of 1. In the next request, NGINX

Plus looks up the `$remote_addr` in the key-value zone, finds the entry, and maps the value to the variable `$num_failures`. This variable is then evaluated in the `if` statement. When the variable has a value, the `if` evaluates to True and NGINX Plus returns the 403 Forbidden return code:

```
$ curl 'http://127.0.0.1/'
Forbidden
```

You can update or delete the key by making a `PATCH` method request:

```
$ curl -X PATCH -d '{"127.0.0.1":null}' \
  'http://127.0.0.1/api/3/http/keyvals/blacklist'
```

NGINX Plus deletes the key if the value is `null`, and requests will again return 200 OK.

Discussion

The key-value store, an NGINX Plus exclusive feature, enables applications to inject information into NGINX Plus. In the example provided, the `$remote_addr` variable is used to create a dynamic blacklist. You can populate the key-value store with any key that NGINX Plus might have as a variable—a session cookie, for example—and provide NGINX Plus an external value. In NGINX Plus R16, the key-value store became cluster-aware, meaning that you have to provide your key-value update to only one NGINX Plus server, and all of them will receive the information.

Also See

Dynamic Bandwidth Limits

5.3 Installing with Puppet

Problem

You need to install and configure NGINX with Puppet to manage NGINX configurations as code and conform with the rest of your Puppet configurations.

Solution

Create a module that installs NGINX, manages the files you need, and ensures that NGINX is running:

```
class nginx {
   package {"nginx": ensure => 'installed',}
   service {"nginx":
       ensure => 'true',
       hasrestart => 'true',
       restart => '/etc/init.d/nginx reload',
   }
   file { "nginx.conf":
       path    => '/etc/nginx/nginx.conf',
       require => Package['nginx'],
       notify  => Service['nginx'],
       content => template('nginx/templates/nginx.conf.erb'),
       user=>'root',
       group=>'root',
       mode='0644';
   }
}
```

This module uses the package management utility to ensure the NGINX package is installed. It also ensures NGINX is running and enabled at boot time. The configuration informs Puppet that the service has a restart command with the `hasrestart` directive, and we can override the `restart` command with an NGINX reload. The file resource will manage and template the *nginx.conf* file with the Embedded Ruby (ERB) templating language. The templating of the file will happen after the NGINX package is installed due to the `require` directive. However, the file resource will notify the NGINX service to reload because of the `notify` directive. The templated configuration file is not included. However, it can be simple to install a default NGINX configuration file, or very complex if using ERB or EPP templating language loops and variable substitution.

Discussion

Puppet is a configuration management tool based in the Ruby programming language. Modules are built in a domain-specific language and called via a manifest file that defines the configuration for a given server. Puppet can be run in a master-slave or masterless configuration. With Puppet, the manifest is run on the master and then sent to the slave. This is important because it ensures that the slave is only delivered the configuration meant for it and no extra configurations meant for other servers. There are a lot of extremely advanced public modules available for Puppet. Starting from these modules will help you get a jump-start on your configuration. A

public NGINX module from voxpupuli on GitHub will template out NGINX configurations for you.

Also See

Puppet Documentation
Puppet Package Documentation
Puppet Service Documentation
Puppet File Documentation
Puppet Templating Documentation
Voxpupuli NGINX Module

5.4 Installing with Chef

Problem

You need to install and configure NGINX with Chef to manage NGINX configurations as code and conform with the rest of your Chef configurations.

Solution

Create a cookbook with a recipe to install NGINX and configure configuration files through templating, and ensure NGINX reloads after the configuration is put in place. The following is an example recipe:

```
package 'nginx' do
  action :install
end

service 'nginx' do
  supports :status => true, :restart => true, :reload => true
  action   [ :start, :enable ]
end

template 'nginx.conf' do
  path   "/etc/nginx.conf"
  source "nginx.conf.erb"
  owner  'root'
  group  'root'
  mode   '0644'
  notifies :reload, 'service[nginx]', :delayed
end
```

The `package` block installs NGINX. The `service` block ensures that NGINX is started and enabled at boot, then declares to the rest of Chef what the `nginx` service will support as far as actions. The `template` block templates an ERB file and places it at */etc/nginx.conf* with an owner and group of root. The `template` block also sets the `mode` to `644` and notifies the `nginx` service to `reload`, but waits until the end of the Chef run declared by the `:delayed` statement. The templated configuration file is not included. However, it can be as simple as a default NGINX configuration file or very complex with ERB templating language loops and variable substitution.

Discussion

Chef is a configuration management tool based in Ruby. Chef can be run in a master-slave, or solo configuration, now known as Chef Zero. Chef has a very large community with many public cookbooks called the Supermarket. Public cookbooks from the Supermarket can be installed and maintained via a command-line utility called Berkshelf. Chef is extremely capable, and what we have demonstrated is just a small sample. The public NGINX cookbook in the Supermarket is extremely flexible and provides the options to easily install NGINX from a package manager or from source, and the ability to compile and install many different modules as well as template out the basic configurations.

Also See

Chef documentation
Chef Package
Chef Service
Chef Template
Chef Supermarket for NGINX

5.5 Installing with Ansible

Problem

You need to install and configure NGINX with Ansible to manage NGINX configurations as code and conform with the rest of your Ansible configurations.

Solution

Create an Ansible playbook to install NGINX and manage the *nginx.conf* file. The following is an example task file for the playbook to install NGINX. Ensure it's running and template the configuration file:

```
- name: NGINX | Installing NGINX
  package: name=nginx state=present

- name: NGINX | Starting NGINX
  service:
    name: nginx
    state: started
    enabled: yes

- name: Copy nginx configuration in place.
  template:
    src: nginx.conf.j2
    dest: "/etc/nginx/nginx.conf"
    owner: root
    group: root
    mode: 0644
  notify:
    - reload nginx
```

The `package` block installs NGINX. The `service` block ensures that NGINX is started and enabled at boot. The `template` block templates a *Jinja2* file and places the result at */etc/nginx.conf* with an owner and group of root. The `template` block also sets the `mode` to `644` and notifies the `nginx` service to `reload`. The templated configuration file is not included. However, it can be as simple as a default NGINX configuration file or very complex with Jinja2 templating language loops and variable substitution.

Discussion

Ansible is a widely used and powerful configuration management tool based in Python. The configuration of tasks is in YAML, and you use the Jinja2 templating language for file templating. Ansible offers a master named Ansible Tower on a subscription model. However, it's commonly used from local machines or to build servers directly to the client or in a masterless model. Ansible bulk SSHes into its servers and runs the configurations. Much like other configuration management tools, there's a large community of pub-

lic roles. Ansible calls this the Ansible Galaxy. You can find very sophisticated roles to utilize in your playbooks.

Also See

Ansible Documentation
Ansible Packages
Ansible Service
Ansible Template
Ansible Galaxy

5.6 Installing with SaltStack

Problem

You need to install and configure NGINX with SaltStack to manage NGINX configurations as code and conform with the rest of your SaltStack configurations.

Solution

Install NGINX through the package management module and manage the configuration files you desire. The following is an example state file (*sls*) that will install the `nginx` package and ensure the service is running, enabled at boot, and reload if a change is made to the configuration file:

```
nginx:
  pkg:
    - installed
  service:
    - name: nginx
    - running
    - enable: True
    - reload: True
    - watch:
      - file: /etc/nginx/nginx.conf

/etc/nginx/nginx.conf:
  file:
    - managed
    - source: salt://path/to/nginx.conf
    - user: root
    - group: root
    - template: jinja
    - mode: 644
```

```
    - require:
      - pkg: nginx
```

This is a basic example of installing NGINX via a package management utility and managing the *nginx.conf* file. The NGINX package is installed and the service is running and enabled at boot. With SaltStack you can declare a file managed by Salt as seen in the example and templated by many different templating languages. The templated configuration file is not included. However, it can be as simple as a default NGINX configuration file or very complex with the Jinja2 templating language loops and variable substitution. This configuration also specifies that NGINX must be installed prior to managing the file because of the `require` statement. After the file is in place, NGINX is reloaded because of the `watch` directive on the service and reloads as opposed to restarts because the `reload` directive is set to `True`.

Discussion

SaltStack is a powerful configuration management tool that defines server states in YAML. Modules for SaltStack can be written in Python. Salt exposes the Jinja2 templating language for states as well as for files. However, for files there are many other options, such as Mako, Python itself, and others. Salt works in a master-slave configuration as well as a masterless configuration. Slaves are called minions. The master-slave transport communication, however, differs from others and sets SaltStack apart. With Salt you're able to choose ZeroMQ, TCP, or Reliable Asynchronous Event Transport (RAET) for transmissions to the Salt agent; or you can not use an agent, and the master can SSH instead. Because the transport layer is by default asynchronous, SaltStack is built to be able to deliver its message to a large number of minions with low load to the master server.

Also See

SaltStack
Installed Packages
Managed Files
Templating with Jinja

5.7 Automating Configurations with Consul Templating

Problem

You need to automate your NGINX configuration to respond to changes in your environment through use of Consul.

Solution

Use the `consul-template` daemon and a template file to template out the NGINX configuration file of your choice:

```
upstream backend { {{range service "app.backend"}}
    server {{.Address}};{{end}}
}
```

This example is a Consul Template file that templates an upstream configuration block. This template will loop through nodes in Consul identified as `app.backend`. For every node in Consul, the template will produce a server directive with that node's IP address.

The `consul-template` daemon is run via the command line and can be used to reload NGINX every time the configuration file is templated with a change:

```
# consul-template -consul consul.example.internal -template \
 template:/etc/nginx/conf.d/upstream.conf:"nginx -s reload"
```

This command instructs the `consul-template` daemon to connect to a Consul cluster at `consul.example.internal` and to use a file named *template* in the current working directory to template the file and output the generated contents to */etc/nginx/conf.d/upstream.conf*, then to reload NGINX every time the templated file changes. The `-template` flag takes a string of the template file, the output location, and the command to run after the templating process takes place; these three variables are separated by a colon. If the command being run has spaces, make sure to wrap it in double quotes. The `-consul` flag tells the daemon what Consul cluster to connect to.

Discussion

Consul is a powerful service discovery tool and configuration store. Consul stores information about nodes as well as key-value pairs in

a directory-like structure and allows for restful API interaction. Consul also provides a DNS interface on each client, allowing for domain name lookups of nodes connected to the cluster. A separate project that utilizes Consul clusters is the `consul-template` daemon; this tool templates files in response to changes in Consul nodes, services, or key-value pairs. This makes Consul a very powerful choice for automating NGINX. With `consul-template` you can also instruct the daemon to run a command after a change to the template takes place. With this, we can reload the NGINX configuration and allow your NGINX configuration to come alive along with your environment. With Consul you're able to set up health checks on each client to check the health of the intended service. With this failure detection, you're able to template your NGINX configuration accordingly to only send traffic to healthy hosts.

Also See

Consul Home Page
Introduction to Consul Template
Consul Template GitHub

CHAPTER 6

Authentication

6.0 Introduction

NGINX is able to authenticate clients. Authenticating client requests with NGINX offloads work and provides the ability to stop unauthenticated requests from reaching your application servers. Modules available for NGINX Open Source include basic authentication and authentication subrequests. The NGINX Plus exclusive module for verfying JSON Web Tokens (JWTs) enables integration with third-party authentication providers that use the authentication standard OpenID Connect.

6.1 HTTP Basic Authentication

Problem

You need to secure your application or content via HTTP basic authentication.

Solution

Generate a file in the following format, where the password is encrypted or hashed with one of the allowed formats:

```
# comment
name1:password1
name2:password2:comment
name3:password3
```

The username is the first field, the password the second field, and the delimiter is a colon. There is an optional third field, which you can use to comment on each user. NGINX can understand a few different formats for passwords, one of which is whether the password is encrypted with the C function `crypt()`. This function is exposed to the command line by the `openssl passwd` command. With `openssl` installed, you can create encrypted password strings by using the following command:

```
$ openssl passwd MyPassword1234
```

The output will be a string that NGINX can use in your password file.

Use the `auth_basic` and `auth_basic_user_file` directives within your NGINX configuration to enable basic authentication:

```
location / {
    auth_basic          "Private site";
    auth_basic_user_file conf.d/passwd;
}
```

You can use the `auth_basic` directives in the HTTP, server, or location contexts. The `auth_basic` directive takes a string parameter, which is displayed on the basic authentication pop-up window when an unauthenticated user arrives. The `auth_basic_user_file` specifies a path to the user file.

Discussion

You can generate basic authentication passwords a few ways and in a few different formats with varying degrees of security. The `htpasswd` command from Apache can also generate passwords. Both the `openssl` and `htpasswd` commands can generate passwords with the `apr1` algorithm, which NGINX can also understand. The password can also be in the salted SHA-1 format that Lightweight Directory Access Protocol (LDAP) and Dovecot use. NGINX supports more formats and hashing algorithms; however, many of them are considered insecure because they can easily be defeated by brute-force attacks.

You can use basic authentication to protect the context of the entire NGINX host, specific virtual servers, or even just specific location blocks. Basic authentication won't replace user authentication for web applications, but it can help keep private information secure.

Under the hood, basic authentication is done by the server returning a 401 unauthorized HTTP code with the response header `WWW-Authenticate`. This header will have a value of `Basic realm="`*`your string`*`"`. This response causes the browser to prompt for a username and password. The username and password are concatenated and delimited with a colon, then base64-encoded, and then sent in a request header named `Authorization`. The `Authorization` request header will specify a `Basic` and `user:password` encoded string. The server decodes the header and verifies against the provided `auth_basic_user_file`. Because the username password string is merely base64-encoded, it's recommended to use HTTPS with basic authentication.

6.2 Authentication Subrequests

Problem

You have a third-party authentication system for which you would like requests authenticated.

Solution

Use the `http_auth_request_module` to make a request to the authentication service to verify identity before serving the request:

```
location /private/ {
    auth_request     /auth;
    auth_request_set $auth_status $upstream_status;
}

location = /auth {
    internal;
    proxy_pass              http://auth-server;
    proxy_pass_request_body off;
    proxy_set_header        Content-Length "";
    proxy_set_header        X-Original-URI $request_uri;
}
```

The `auth_request` directive takes a URI parameter that must be a local internal location. The `auth_request_set` directive allows you to set variables from the authentication subrequest.

Discussion

The `http_auth_request_module` enables authentication on every request handled by the NGINX server. The module makes a subrequest before serving the original to determine if the request has access to the resource it's requesting. The entire original request is proxied to this subrequest location. The authentication location acts as a typical proxy to the subrequest and sends the original request, including the original request body and headers. The HTTP status code of the subrequest is what determines whether or not access is granted. If the subrequest returns with an HTTP 200 status code, the authentication is successful and the request is fulfilled. If the subrequest returns HTTP 401 or 403, the same will be returned for the original request.

If your authentication service does not request the request body, you can drop the request body with the `proxy_pass_request_body` directive, as demonstrated. This practice will reduce the request size and time. Because the response body is discarded, the `Content-Length` header must be set to an empty string. If your authentication service needs to know the URI being accessed by the request, you'll want to put that value in a custom header that your authentication service checks and verifies. If there are things you do want to keep from the subrequest to the authentication service, like response headers or other information, you can use the `auth_request_set` directive to make new variables out of response data.

6.3 Validating JWTs

Problem

You need to validate a JWT before the request is handled with NGINX Plus.

Solution

Use NGINX Plus's HTTP JWT authentication module to validate the token signature and embed JWT claims and headers as NGINX variables:

```
location /api/ {
    auth_jwt          "api";
```

```
        auth_jwt_key_file conf/keys.json;
    }
```

This configuration enables validation of JWTs for this location. The `auth_jwt` directive is passed a string, which is used as the authentication realm. The `auth_jwt` takes an optional token parameter of a variable that holds the JWT. By default, the `Authentication` header is used per the JWT standard. The `auth_jwt` directive can also be used to cancel the effects of required JWT authentication from inherited configurations. To turn `off` authentication, pass the keyword to the `auth_jwt` directive with nothing else. To cancel inherited authentication requirements, pass the `off` keyword to the `auth_jwt` directive with nothing else. The `auth_jwt_key_file` takes a single parameter. This parameter is the path to the key file in standard JSON Web Key format.

Discussion

NGINX Plus is able to validate the JSON web signature types of tokens as opposed to the JSON web encryption type, where the entire token is encrypted. NGINX Plus is able to validate signatures that are signed with the HS256, RS256, and ES256 algorithms. Having NGINX Plus validate the token can save the time and resources needed to make a subrequest to an authentication service. NGINX Plus deciphers the JWT header and payload, and captures the standard headers and claims into embedded variables for your use.

Also See

RFC Standard Documentation of JSON Web Signature
RFC Standard Documentation of JSON Web Algorithms
RFC Standard Documentation of JSON Web Token
NGINX Embedded Variables
Detailed NGINX Blog

6.4 Creating JSON Web Keys

Problem

You need a JSON Web Key for NGINX Plus to use.

Solution

NGINX Plus utilizes the JSON Web Key (JWK) format as specified in the RFC standard. This standard allows for an array of key objects within the JWK file.

The following is an example of what the key file may look like:

```
{"keys":
  [
    {
        "kty":"oct",
        "kid":"0001",
        "k":"OctetSequenceKeyValue"
    },
    {
        "kty":"EC",
        "kid":"0002"
        "crv":"P-256",
        "x": "XCoordinateValue",
        "y": "YCoordinateValue",
        "d": "PrivateExponent",
        "use": "sig"
    },
    {
        "kty":"RSA",
        "kid":"0003"
        "n": "Modulus",
        "e": "Exponent",
        "d": "PrivateExponent"
    }
  ]
}
```

The JWK file shown demonstrates the three initial types of keys noted in the RFC standard. The format of these keys is also part of the RFC standard. The `kty` attribute is the key type. This file shows three key types: the Octet Sequence (`oct`), the EllipticCurve (`EC`), and the `RSA` type. The `kid` attribute is the key ID. Other attributes to these keys are specified in the standard for that type of key. Look to the RFC documentation of these standards for more information.

Discussion

There are numerous libraries available in many different languages to generate the JSON Web Key. It's recommended to create a key service that is the central JWK authority to create and rotate your JWKs at a regular interval. For enhanced security, it's recommended

to make your JWKs as secure as your SSL/TLS certifications. Secure your key file with proper user and group permissions. Keeping them in memory on your host is best practice. You can do so by creating an in-memory filesystem like ramfs. Rotating keys on a regular interval is also important; you may opt to create a key service that creates public and private keys and offers them to the application and NGINX via an API.

Also See

RFC standardization documentation of JSON Web Key

6.5 Authenticate Users via Existing OpenID Connect SSO

Problem

You want to offload OpenID Connect authentication validation to NGINX Plus.

Solution

Use the JWT module that comes with NGINX Plus to secure a location or server, and instruct the `auth_jwt` directive to use `$cookie_auth_token` as the token to be validated:

```
location /private/ {
    auth_jwt "Google Oauth" token=$cookie_auth_token;
    auth_jwt_key_file /etc/nginx/google_certs.jwk;
}
```

This configuration directs NGINX Plus to secure the */private/* URI path with JWT validation. Google OAuth 2.0 OpenID Connect uses the cookie `auth_token` rather than the default bearer token. Thus, you must instruct NGINX to look for the token in this cookie rather than in the NGINX Plus default location. The `auth_jwt_key_file` location is set to an arbitrary path, which is a step that we cover in Recipe 6.6.

Discussion

This configuration demonstrates how you can validate a Google OAuth 2.0 OpenID Connect JSON Web Token with NGINX Plus. The NGINX Plus JWT authentication module for HTTP is able to

validate any JSON Web Token that adheres to the RFC for JSON Web Signature specification, instantly enabling any SSO authority that utilizes JSON Web Tokens to be validated at the NGINX Plus layer. The OpenID 1.0 protocol is a layer on top of the OAuth 2.0 authentication protocol that adds identity, enabling the use of JWTs to prove the identity of the user sending the request. With the signature of the token, NGINX Plus can validate that the token has not been modified since it was signed. In this way, Google is using an asynchronous signing method and makes it possible to distribute public JWKs while keeping its private JWK secret.

NGINX Plus can also control the Authorization Code Flow for OpenID Connect 1.0, enabling NGINX Plus as a Relay Party for OpenID Connect. This capability enables integration with most major identity providers, including CA Single Sign On (formerly SiteMinder), ForgeRock OpenAM, Keycloak, Okta, OneLogin, and Ping Identity. For more information and a reference implementation of NGINX Plus as a relaying party for OpenID Connect authentication, check out the NGINX Inc OpenID Connect GitHub Repository.

Also See

Detailed NGINX Blog on OpenID Connect
OpenID Connect

6.6 Obtaining the JSON Web Key from Google

Problem

You need to obtain the JSON Web Key from Google to use when validating OpenID Connect tokens with NGINX Plus.

Solution

Utilize Cron to request a fresh set of keys every hour to ensure your keys are always up-to-date:

```
0 * * * * root wget https://www.googleapis.com/oauth2/v3/ \
  certs-O /etc/nginx/google_certs.jwk
```

This code snippet is a line from a crontab file. Unix-like systems have many options for where crontab files can live. Every user will

have a user-specific crontab, and there's also a number of files and directories in the */etc/* directory.

Discussion

Cron is a common way to run a scheduled task on a Unix-like system. JSON Web Keys should be rotated on a regular basis to ensure the security of the key, and in turn, the security of your system. To ensure that you always have the most up-to-date key from Google, you'll want to check for new JWKs at regular intervals. This Cron solution is one way of doing so.

Also See

Cron

CHAPTER 7

Security Controls

7.0 Introduction

Security is done in layers, and there must be multiple layers to your security model for it to be truly hardened. In this chapter, we go through many different ways to secure your web applications with NGINX and NGINX Plus. You can use many of these security methods in conjunction with one another to help harden security. The following are a number of security sections that explore features of NGINX and NGINX Plus that can assist in strengthening your application. You might notice that this chapter does not touch upon one of the largest security features of NGINX, the ModSecurity 3.0 NGINX module, which turns NGINX into a Web Application Firewall (WAF). To learn more about the WAF capabilities, download the ModSecurity 3.0 and NGINX: Quick Start Guide.

7.1 Access Based on IP Address

Problem

You need to control access based on the IP address of the client.

Solution

Use the HTTP access module to control access to protected resources:

```
location /admin/ {
    deny  10.0.0.1;
```

```
        allow 10.0.0.0/20;
        allow 2001:0db8::/32;
        deny  all;
    }
```

The given location block allows access from any IPv4 address in 10.0.0.0/20 except 10.0.0.1, allows access from IPv6 addresses in the `2001:0db8::/32` subnet, and returns a 403 for requests originating from any other address. The `allow` and `deny` directives are valid within the HTTP, server, and location contexts. Rules are checked in sequence until a match is found for the remote address.

Discussion

Protecting valuable resources and services on the internet must be done in layers. NGINX provides the ability to be one of those layers. The `deny` directive blocks access to a given context, while the `allow` directive can be used to allow subsets of the blocked access. You can use IP addresses, IPv4 or IPv6, CIDR block ranges, the keyword `all`, and a Unix socket. Typically when protecting a resource, one might allow a block of internal IP addresses and deny access from all.

7.2 Allowing Cross-Origin Resource Sharing

Problem

You're serving resources from another domain and need to allow cross-origin resource sharing (CORS) to enable browsers to utilize these resources.

Solution

Alter headers based on the `request` method to enable CORS:

```
map $request_method $cors_method {
  OPTIONS 11;
  GET  1;
  POST 1;
  default 0;
}
server {
  ...
  location / {
    if ($cors_method ~ '1') {
        add_header 'Access-Control-Allow-Methods'
           'GET,POST,OPTIONS';
```

```
            add_header 'Access-Control-Allow-Origin'
               '*.example.com';
            add_header 'Access-Control-Allow-Headers'
                       'DNT,
                        Keep-Alive,
                        User-Agent,
                        X-Requested-With,
                        If-Modified-Since,
                        Cache-Control,
                        Content-Type';
        }
        if ($cors_method = '11') {
            add_header 'Access-Control-Max-Age' 1728000;
            add_header 'Content-Type' 'text/plain; charset=UTF-8';
            add_header 'Content-Length' 0;
            return 204;
        }
      }
    }
```

There's a lot going on in this example, which has been condensed by using a `map` to group the `GET` and `POST` methods together. The `OPTIONS` request method returns a *preflight* request to the client about this server's CORS rules. `OPTIONS`, `GET`, and `POST` methods are allowed under CORS. Setting the `Access-Control-Allow-Origin` header allows for content being served from this server to also be used on pages of origins that match this header. The preflight request can be cached on the client for 1,728,000 seconds, or 20 days.

Discussion

Resources such as JavaScript make CORS when the resource they're requesting is of a domain other than its own. When a request is considered cross origin, the browser is required to obey CORS rules. The browser will not use the resource if it does not have headers that specifically allow its use. To allow our resources to be used by other subdomains, we have to set the CORS headers, which can be done with the `add_header` directive. If the request is a `GET`, `HEAD`, or `POST` with standard content type, and the request does not have special headers, the browser will make the request and only check for origin. Other request methods will cause the browser to make the preflight request to check the terms of the server to which it will obey for that resource. If you do not set these headers appropriately, the browser will give an error when trying to utilize that resource.

7.3 Client-Side Encryption

Problem

You need to encrypt traffic between your NGINX server and the client.

Solution

Utilize one of the SSL modules, such as the `ngx_http_ssl_module` or `ngx_stream_ssl_module` to encrypt traffic:

```
http { # All directives used below are also valid in stream
    server {
        listen 8433 ssl;
        ssl_protocols TLSv1.2 TLSv1.3;
        ssl_ciphers HIGH:!aNULL:!MD5;
        ssl_certificate /etc/nginx/ssl/example.pem;
        ssl_certificate_key /etc/nginx/ssl/example.key;
        ssl_certificate /etc/nginx/ssl/example.ecdsa.crt;
        ssl_certificate_key /etc/nginx/ssl/example.ecdsa.key;
        ssl_session_cache shared:SSL:10m;
        ssl_session_timeout 10m;
    }
}
```

This configuration sets up a server to listen on a port encrypted with SSL, 8443. The server accepts the SSL protocol versions TLSv1.2 and TLSv1.3. Two sets of certificate and key pair locations are disclosed to the server for use. The server is instructed to use the highest strength offered by the client while restricting a few that are insecure. The Elliptic Curve Cryptopgraphy (ECC) ciphers are prioritized as we've provided an ECC certificate key pair. The SSL session cache and timeout allow workers to cache and store session parameters for a given amount of time. There are many other session cache options that can help with performance or security of all types of use cases. You can use session cache options in conjunction with one another. However, specifying one without the default will turn off that default, built-in session cache.

Discussion

Secure transport layers are the most common way of encrypting information in transit. As of this writing, the TLS protocol is preferred over the SSL protocol. That's because versions 1 through 3 of

SSL are now considered insecure. Although the protocol name might be different, TLS still establishes a secure socket layer. NGINX enables your service to protect information between you and your clients, which in turn protects the client and your business. When using a signed certificate, you need to concatenate the certificate with the certificate authority chain. When you concatenate your certificate and the chain, your certificate should be above the chain in the file. If your certificate authority has provided many files in the chain, it can also provide the order in which they are layered. The SSL session cache enhances performance by not having to negotiate for SSL/TLS versions and ciphers.

In testing, ECC certificates were found to be faster than the equivalent-strength RSA certificates. The key size is smaller, which results in the ability to serve more SSL/TLS connections, and with faster handshakes. NGINX allows you to configure multiple certificates and keys, and then serve the optimal certificate for the client browser. This allows you to take advantage of the newer technology but still serve older clients.

Also See

Mozilla Server Side TLS Page
Mozilla SSL Configuration Generator
Test Your SSL Configuration with SSL Labs SSL Test

7.4 Upstream Encryption

Problem

You need to encrypt traffic between NGINX and the upstream service and set specific negotiation rules for compliance regulations or if the upstream is outside of your secured network.

Solution

Use the SSL directives of the HTTP proxy module to specify SSL rules:

```
location / {
    proxy_pass https://upstream.example.com;
    proxy_ssl_verify on;
    proxy_ssl_verify_depth 2;
```

```
        proxy_ssl_protocols TLSv1.2;
    }
```

These proxy directives set specific SSL rules for NGINX to obey. The configured directives ensure that NGINX verifies that the certificate and chain on the upstream service is valid up to two certificates deep. The `proxy_ssl_protocols` directive specifies that NGINX will only use TLS version 1.2. By default, NGINX does not verify upstream certificates and accepts all TLS versions.

Discussion

The configuration directives for the HTTP proxy module are vast, and if you need to encrypt upstream traffic, you should at least turn on verification. You can proxy over HTTPS simply by changing the protocol on the value passed to the `proxy_pass` directive. However, this does not validate the upstream certificate. Other directives, such as `proxy_ssl_certificate` and `proxy_ssl_certificate_key`, allow you to lock down upstream encryption for enhanced security. You can also specify `proxy_ssl_crl` or a certificate revocation list, which lists certificates that are no longer considered valid. These SSL proxy directives help harden your system's communication channels within your own network or across the public internet.

7.5 Securing a Location

Problem

You need to secure a location block using a secret.

Solution

Use the secure link module and the `secure_link_secret` directive to restrict access to resources to users who have a secure link:

```
location /resources {
    secure_link_secret mySecret;
    if ($secure_link = "") { return 403; }

    rewrite ^ /secured/$secure_link;
}

location /secured/ {
    internal;
```

```
        root /var/www;
    }
```

This configuration creates an internal and public-facing location block. The public-facing location block `/resources` will return a 403 Forbidden unless the request URI includes an `md5` hash string that can be verified with the secret provided to the `secure_link_secret` directive. The `$secure_link` variable is an empty string unless the hash in the URI is verified.

Discussion

Securing resources with a secret is a great way to ensure your files are protected. The secret is used in conjunction with the URI. This string is then `md5` hashed, and the hex digest of that `md5` hash is used in the URI. The hash is placed into the link and evaluated by NGINX. NGINX knows the path to the file being requested as it's in the URI after the hash. NGINX also knows your secret as it's provided via the `secure_link_secret` directive. NGINX is able to quickly validate the `md5` hash and store the URI in the `$secure_link` variable. If the hash cannot be validated, the variable is set to an empty string. It's important to note that the argument passed to the `secure_link_secret` must be a static string; it cannot be a variable.

7.6 Generating a Secure Link with a Secret

Problem

You need to generate a secure link from your application using a secret.

Solution

The secure link module in NGINX accepts the hex digest of an `md5` hashed string, where the string is a concatenation of the URI path and the secret. Building on the last section, Recipe 7.5, we will create the secured link that will work with the previous configuration example given that there's a file present at */var/www/secured/index.html*. To generate the hex digest of the `md5` hash, we can use the Unix `openssl` command:

```
$ echo -n 'index.htmlmySecret' | openssl md5 -hex
(stdin)= a53bee08a4bf0bbea978ddf736363a12
```

Here we show the URI that we're protecting, *index.html*, concatenated with our secret, `mySecret`. This string is passed to the `openssl` command to output an `md5` hex digest.

The following is an example of the same hash digest being constructed in Python using the `hashlib` library that is included in the Python Standard Library:

```
import hashlib
hashlib.md5.(b'index.htmlmySecret').hexdigest()
'a53bee08a4bf0bbea978ddf736363a12'
```

Now that we have this hash digest, we can use it in a URL. Our example will be www.example.com making a request for the file */var/www/secured/index.html* through our */resources* location. Our full URL will be the following:

```
www.example.com/resources/a53bee08a4bf0bbea978ddf736363a12/\
index.html
```

Discussion

Generating the digest can be done in many ways, in many languages. Things to remember: the URI path goes before the secret, there are no carriage returns in the string, and use the hex digest of the `md5` hash.

7.7 Securing a Location with an Expire Date

Problem

You need to secure a location with a link that expires at some future time and is specific to a client.

Solution

Utilize the other directives included in the secure link module to set an expire time and use variables in your secure link:

```
location /resources {
    root /var/www;
    secure_link $arg_md5,$arg_expires;
    secure_link_md5 "$secure_link_expires$uri$remote_addr
   mySecret";
    if ($secure_link = "") { return 403; }
    if ($secure_link = "0") { return 410; }
}
```

The `secure_link` directive takes two parameters separated with a comma. The first parameter is the variable that holds the `md5` hash. This example uses an HTTP argument of `md5`. The second parameter is a variable that holds the time in which the link expires in Unix epoch time format. The `secure_link_md5` directive takes a single parameter that declares the format of the string that is used to construct the `md5` hash. Like the other configuration, if the hash does not validate, the `$secure_link` variable is set to an empty string. However, with this usage, if the hash matches but the time has expired, the `$secure_link` variable will be set to `0`.

Discussion

This usage of securing a link is more flexible and looks cleaner than the `secure_link_secret` shown in Recipe 7.5. With these directives, you can use any number of variables that are available to NGINX in the hashed string. Using user-specific variables in the hash string will strengthen your security as users won't be able to trade links to secured resources. It's recommended to use a variable like `$remote_addr` or `$http_x_forwarded_for`, or a session cookie header generated by the application. The arguments to `secure_link` can come from any variable you prefer, and they can be named whatever best fits. The conditions around what the `$secure_link` variable is set to returns known HTTP codes for Forbidden and Gone. The HTTP 410, Gone, works great for expired links as the condition is to be considered permanent.

7.8 Generating an Expiring Link

Problem

You need to generate a link that expires.

Solution

Generate a timestamp for the expire time in the Unix epoch format. On a Unix system, you can test by using the date as demonstrated in the following:

```
$ date -d "2020-12-31 00:00" +%s --utc
1609372800
```

Next, you'll need to concatenate your hash string to match the string configured with the `secure_link_md5` directive. In this case, our string to be used will be `1293771600/resources/index.html127.0.0.1 mySecret`. The `md5` hash is a bit different than just a hex digest. It's an `md5` hash in binary format, base64-encoded, with plus signs (+) translated to hyphens (-), slashes (/) translated to underscores (_), and equal (=) signs removed. The following is an example on a Unix system:

```
$ echo -n '1609372800/resources/index.html127.0.0.1 mySecret' \
  | openssl md5 -binary \
  | openssl base64 \
  | tr +/ -_ \
  | tr -d =
TG6ck3OpAttQ1d7jW3JOcw
```

Now that we have our hash, we can use it as an argument along with the expire date:

```
/resources/index.html?md5=TG6ck3OpAttQ1d7jW3JOcw&expires=1609372
  800'
```

The following is a more practical example in Python utilizing a relative time for the expiration, setting the link to expire one hour from generation. At the time of writing this example works with Python 2.7 and 3.x utilizing the Python Standard Library:

```
from datetime import datetime, timedelta
from base64 import b64encode
import hashlib

# Set environment vars
resource = b'/resources/index.html'
remote_addr = b'127.0.0.1'
host = b'www.example.com'
mysecret = b'mySecret'

# Generate expire timestamp
now = datetime.utcnow()
expire_dt = now + timedelta(hours=1)
expire_epoch = str.encode(expire_dt.strftime('%s'))

# md5 hash the string
uncoded = expire_epoch + resource + remote_addr + mysecret
md5hashed = hashlib.md5(uncoded).digest()

# Base64 encode and transform the string
b64 = b64encode(md5hashed)
unpadded_b64url = b64.replace(b'+', b'-')\
    .replace(b'/', b'_')\
```

```
        .replace(b'=', b'')

    # Format and generate the link
    linkformat = "{}{}?md5={}?expires={}"
    securelink = linkformat.format(
        host.decode(),
        resource.decode(),
        unpadded_b64url.decode(),
        expire_epoch.decode()
    )
    print(securelink)
```

Discussion

With this pattern we're able to generate a secure link in a special format that can be used in URLs. The secret provides security through use of a variable that is never sent to the client. You're able to use as many other variables as you need to in order to secure the location. `md5` hashing and base64 encoding are common, lightweight, and available in nearly every language.

7.9 HTTPS Redirects

Problem

You need to redirect unencrypted requests to HTTPS.

Solution

Use a rewrite to send all HTTP traffic to HTTPS:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    return 301 https://$host$request_uri;
}
```

This configuration listens on port 80 as the default server for both IPv4 and IPv6 and for any hostname. The `return` statement returns a 301 permanent redirect to the HTTPS server at the same host and request URI.

Discussion

It's important to always redirect to HTTPS where appropriate. You may find that you do not need to redirect all requests but only those

with sensitive information being passed between client and server. In that case, you may want to put the `return` statement in particular locations only, such as */login*.

7.10 Redirecting to HTTPS where SSL/TLS Is Terminated Before NGINX

Problem

You need to redirect to HTTPS, however, you've terminated SSL/TLS at a layer before NGINX.

Solution

Use the standard `X-Forwarded-Proto` header to determine if you need to redirect:

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    if ($http_x_forwarded_proto = 'http') {
        return 301 https://$host$request_uri;
    }
}
```

This configuration is very much like HTTPS redirects. However, in this configuration we're only redirecting if the header `X-Forwarded-Proto` is equal to HTTP.

Discussion

It's a common use case that you may terminate SSL/TLS in a layer in front of NGINX. One reason you may do something like this is to save on compute costs. However, you need to make sure that every request is HTTPS, but the layer terminating SSL/TLS does not have the ability to redirect. It can, however, set proxy headers. This configuration works with layers such as the Amazon Web Services Elastic Load Balancer, which will offload SSL/TLS at no additional cost. This is a handy trick to make sure that your HTTP traffic is secured.

7.11 HTTP Strict Transport Security

Problem

You need to instruct browsers to never send requests over HTTP.

Solution

Use the HTTP Strict Transport Security (HSTS) enhancement by setting the `Strict-Transport-Security` header:

```
add_header Strict-Transport-Security max-age=31536000;
```

This configuration sets the `Strict-Transport-Security` header to a max age of a year. This will instruct the browser to always do an internal redirect when HTTP requests are attempted to this domain, so that all requests will be made over HTTPS.

Discussion

For some applications a single HTTP request trapped by a man in the middle attack could be the end of the company. If a form post containing sensitive information is sent over HTTP, the HTTPS redirect from NGINX won't save you; the damage is done. This opt-in security enhancement informs the browser to never make an HTTP request, and therefore the request is never sent unencrypted.

Also See

RFC-6797 HTTP Strict Transport Security
OWASP HSTS Cheat Sheet

7.12 Satisfying Any Number of Security Methods

Problem

You need to provide multiple ways to pass security to a closed site.

Solution

Use the `satisfy` directive to instruct NGINX that you want to satisfy any or all of the security methods used:

```
location / {
    satisfy any;

    allow 192.168.1.0/24;
    deny  all;

    auth_basic           "closed site";
    auth_basic_user_file conf/htpasswd;
}
```

This configuration tells NGINX that the user requesting the `location /` needs to satisfy one of the security methods: either the request needs to originate from the *192.168.1.0/24* CIDR block or be able to supply a username and password that can be found in the *conf/htpasswd* file. The `satisfy` directive takes one of two options: `any` or `all`.

Discussion

The `satisfy` directive is a great way to offer multiple ways to authenticate to your web application. By specifying `any` to the `satisfy` directive, the user must meet one of the security challenges. By specifying `all` to the `satisfy` directive, the user must meet all of the security challenges. This directive can be used in conjunction with the `http_access_module` detailed in Recipe 7.1, the `http_auth_basic_module` detailed in Recipe 6.1, the `http_auth_request_module` detailed in Recipe 6.2, and the `http_auth_jwt_module` detailed in Recipe 6.3. Security is only truly secure if it's done in multiple layers. The `satisfy` directive will help you achieve this for locations and servers that require deep security rules.

7.13 Dynamic DDoS Mitigation

Problem

You need a dynamic Distributed Denial of Service (DDoS) mitigation solution.

Solution

Use NGINX Plus to build a cluster-aware rate limit and automatic blacklist:

```
limit_req_zone   $remote_addr zone=per_ip:1M rate=100r/s sync;
                 # Cluster-aware rate limit
limit_req_status 429;

keyval_zone zone=sinbin:1M timeout=600 sync;
              # Cluster-aware "sin bin" with
              # 10-minute TTL
keyval $remote_addr $in_sinbin zone=sinbin;
              # Populate $in_sinbin with
              # matched client IP addresses

server {
    listen 80;
    location / {
        if ($in_sinbin) {
            set $limit_rate 50; # Restrict bandwidth of bad clients
        }

        limit_req zone=per_ip;
              # Apply the rate limit here
        error_page 429 = @send_to_sinbin;
              # Excessive clients are moved to
              # this location
        proxy_pass http://my_backend;
    }

    location @send_to_sinbin {
        rewrite ^ /api/3/http/keyvals/sinbin break;
              # Set the URI of the
              # "sin bin" key-val
        proxy_method POST;
        proxy_set_body '{"$remote_addr":"1"}';
        proxy_pass http://127.0.0.1:80;
    }

    location /api/ {
        api write=on;
        # directives to control access to the API
    }
}
```

Discussion

This solution uses a synchronized rate limit and a synchronized key-value store to dynamically respond to DDoS attacks and mitigate their effects. The `sync` parameter provided to the `limit_req_zone` and `keyval_zone` directives synchronizes the shared memory zone with other machines in the active-active NGINX Plus cluster. This example identifies clients that send more than 100 requests per sec-

ond, regardless of which NGINX Plus node receives the request. When a client exceeds the rate limit, its IP address is added to a "sin bin" key-value store by making a call to the NGINX Plus API. The sin bin is synchronized across the cluster. Further requests from clients in the sin bin are subject to a very low bandwidth limit, regardless of which NGINX Plus node receives them. Limiting bandwidth is preferable to rejecting requests outright because it does not clearly signal to the client that DDoS mitigation is in effect. After 10 minutes the client is automatically removed from the sin bin.

CHAPTER 8

HTTP/2

8.0 Introduction

HTTP/2 is a major revision to the HTTP protocol. Much of the work done in this version was focused on the transport layer, such as enabling full request and response multiplexing over a single TCP connection. Effiencies were gained through the use of compression on HTTP header fields, and support for request prioritization was added. Another large addition to the protocol was the ability for the server to push messages to the client. This chapter details the basic configuration for enabling HTTP/2 in NGINX as well as configuring gRPC and HTTP/2 server push support.

8.1 Basic Configuration

Problem

You want to take advantage of HTTP/2.

Solution

Turn on HTTP/2 on your NGINX server:

```
server {
    listen 443 ssl http2 default_server;

    ssl_certificate     server.crt;
    ssl_certificate_key server.key;
```

```
    ...
}
```

Discussion

To turn on HTTP/2, you simply need to add the `http2` parameter to the `listen` directive. The catch, however, is that although the protocol does not require the connection to be wrapped in SSL/TLS, some implementations of HTTP/2 clients support only HTTP/2 over an encrypted connection. Another caveat is that the HTTP/2 specification listed a number of TLS 1.2 cipher suites as blacklisted and therefore will fail the handshake. The ciphers NGINX uses by default are not on the blacklist. To test that your setup is correct you can install a plugin for Chrome and Firefox browsers that indicates when a site is using HTTP/2, or on the command line with the `nghttp` utility.

Also See

HTTP/2 RFC Blacklisted Ciphers
Chrome HTTP2 and SPDY Indicator Plugin
Firefox HTTP2 Indicator Add-on

8.2 gRPC

Problem

You need to terminate, inspect, route, or load balance gRPC method calls.

Solution

Use NGINX to proxy gRPC connections.

```
server {
    listen 80 http2;

    location / {
        grpc_pass grpc://backend.local:50051;
    }
}
```

In this configuration NGINX is listening on port `80` for unencrypted HTTP/2 traffic, and proxying that traffic to a machine named `backend.local` on port `50051`. The `grpc_pass` directive instructs

NGINX to treat the commuication as a gRPC call. The `grpc://` in front of our backend server location is not neccessary; however, it does directly indicate that the backend communication is not encrypted.

To utilize TLS encryption between the client and NGINX, and terminate that encryption before passing the calls to the application server, turn on SSL and HTTP/2, as you did in the first section:

```
server {
    listen 443 ssl http2 default_server;

    ssl_certificate     server.crt;
    ssl_certificate_key server.key;
    location / {
        grpc_pass grpc://backend.local:50051;
    }
}
```

This configuration terminates TLS at NGINX and passes the gRPC communication to the application over unencrypted HTTP/2.

To configure NGINX to encrypt the gRPC communication to the application server, providing end-to-end encrypted traffic, simply modify the `grpc_pass` directive to specify `grpcs://` before the server information (note the addition of the `s` denoting secure communication):

```
        grpc_pass grpcs://backend.local:50051;
```

You also can use NGINX to route calls to different backend services based on the gRPC URI, which includes the package, service, and method. To do so, utilize the `location` directive.

```
location /mypackage.service1 {
    grpc_pass grpc://backend.local:50051;
}
location /mypackage.service2 {
    grpc_pass grpc://backend.local:50052;
}
location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
}
```

This configuration example uses the `location` directive to route incoming HTTP/2 traffic between two separate gRPC services, as well as a `location` to serve static content. Method calls for the `mypackage.service1` service are directed to the `backend.local`

server on port `50051`, and calls for `mypackage.service2` are directed to port `50052`. The `location /` catches any other HTTP request and serves static content. This demonstrates how NGINX is able to serve gRPC and non-gRPC under the same HTTP/2 endpoint and route accordingly.

Load balancing gRPC calls is also similar to non-gRPC HTTP traffic:

```
upstream grpcservers {
    server backend1.local:50051;
    server backend2.local:50051;
}
server {
    listen 443 ssl http2 default_server;

    ssl_certificate    server.crt;
    ssl_certificate_key server.key;
    location / {
        grpc_pass grpc://grpcservers;
    }
}
```

The `upstream` block works the exact same way for gRPC as it does for other HTTP traffic. The only difference is that the `upstream` is referenced by `grpc_pass`.

Discussion

NGINX is able to receive, proxy, load balance, route, and terminate encryption for gRPC calls. The gRPC module enables NGINX to set, alter, or drop gRPC call headers, set timeouts for requests, and set upstream SSL/TLS specifications. As gRPC communicates over the HTTP/2 protocol, you can configure NGINX to accept gRPC and non-gRPC web traffic on the same endpoint.

8.3 HTTP/2 Server Push

Problem

You need to preemptively push content to the client.

Solution

Use the HTTP/2 server push feature of NGINX.

```
server {
    listen 443 ssl http2 default_server;

    ssl_certificate     server.crt;
    ssl_certificate_key server.key;
    root /usr/share/nginx/html;

    location = /demo.html {
        http2_push /style.css;
        http2_push /image1.jpg;
    }
}
```

Discussion

To use HTTP/2 server push, your server must be configured for HTTP/2, as is done in Recipe 7.1. From there, you can instruct NGINX to push specific files preemptively with the `http2_push` directive. This directive takes one parameter, the full URI path of the file to push to the client.

NGINX can also automatically push resources to clients if proxied applications include an HTTP response header named `Link`. This header is able to instruct NGINX to preload the resources specified. To enable this feature, add `http2_push_preload on;` to the NGINX configuration.

CHAPTER 9

Sophisticated Media Streaming

9.0 Introduction

This chapter covers streaming media with NGINX in MPEG-4 or Flash Video formats. NGINX is widely used to distribute and stream content to the masses. NGINX supports industry-standard formats and streaming technologies, which will be covered in this chapter. NGINX Plus enables the ability to fragment content on the fly with the HTTP Live Stream module, as well as the ability to deliver HTTP Dynamic Streaming of already fragmented media. NGINX natively allows for bandwidth limits, and NGINX Plus's advanced features offers bitrate limiting, enabling your content to be delivered in the most efficient manner while reserving the servers' resources to reach the most users.

9.1 Serving MP4 and FLV

Problem

You need to stream digital media, originating in MPEG-4 (MP4) or Flash Video (FLV).

Solution

Designate an HTTP location block as *.mp4* or *.flv*. NGINX will stream the media using progressive downloads or HTTP pseudostreaming and support seeking:

```
http {
    server {
        ...

        location /videos/ {
            mp4;
        }
        location ~ \.flv$ {
            flv;
        }
    }
}
```

The example location block tells NGINX that files in the *videos* directory are in MP4 format type and can be streamed with progressive download support. The second location block instructs NGINX that any files ending in *.flv* are in FLV format and can be streamed with HTTP pseudostreaming support.

Discussion

Streaming video or audio files in NGINX is as simple as a single directive. Progressive download enables the client to initiate playback of the media before the file has finished downloading. NGINX supports seeking to an undownloaded portion of the media in both formats.

9.2 Streaming with HLS

Problem

You need to support HTTP Live Streaming (HLS) for H.264/AAC-encoded content packaged in MP4 files.

Solution

Utilize NGINX Plus's HLS module with real-time segmentation, packetization, and multiplexing, with control over fragmentation buffering and more, like forwarding HLS arguments:

```
location /hls/ {
    hls;  # Use the HLS handler to manage requests

    # Serve content from the following location
    alias /var/www/video;
```

```
        # HLS parameters
        hls_fragment            4s;
        hls_buffers         10 10m;
        hls_mp4_buffer_size     1m;
        hls_mp4_max_buffer_size 5m;
    }
```

The location block demonstrated directs NGINX to stream HLS media out of the */var/www/video* directory, fragmenting the media into four-second segments. The number of HLS buffers is set to 10 with a size of 10 megabytes. The initial MP4 buffer size is set to 1 MB with a maximum of 5 MB.

Discussion

The HLS module available in NGINX Plus provides the ability to transmultiplex MP4 media files on the fly. There are many directives that give you control over how your media is fragmented and buffered. The location block must be configured to serve the media as an HLS stream with the HLS handler. The HLS fragmentation is set in number of seconds, instructing NGINX to fragment the media by time length. The amount of buffered data is set with the `hls_buffers` directive specifying the number of buffers and the size. The client is allowed to start playback of the media after a certain amount of buffering has accrued specified by the `hls_mp4_buffer_size`. However, a larger buffer may be necessary as metadata about the video may exceed the initial buffer size. This amount is capped by the `hls_mp4_max_buffer_size`. These buffering variables allow NGINX to optimize the end-user experience; choosing the right values for these directives requires knowing the target audience and your media. For instance, if the bulk of your media is large video files, and your target audience has high bandwidth, you may opt for a larger max buffer size and longer length fragmentation. This will allow for the metadata about the content to be downloaded initially without error and your users to receive larger fragments.

9.3 Streaming with HDS

Problem

You need to support Adobe's HTTP Dynamic Streaming (HDS) that has already been fragmented and separated from the metadata.

Solution

Use NGINX Plus's support for fragmented FLV files (`F4F`) module to offer Adobe Adaptive Streaming to your users:

```
location /video/ {
    alias /var/www/transformed_video;
    f4f;
    f4f_buffer_size 512k;
}
```

The example instructs NGINX Plus to serve previously fragmented media from a location on disk to the client using the NGINX Plus `F4F` module. The buffer size for the index file (*.f4x*) is set to 512 kilobytes.

Discussion

The NGINX Plus `F4F` module enables NGINX to serve previously fragmented media to end users. The configuration of such is as simple as using the `f4f` handler inside of an HTTP location block. The `f4f_buffer_size` directive configures the buffer size for the index file of this type of media.

9.4 Bandwidth Limits

Problem

You need to limit bandwidth to downstream media streaming clients without impacting the viewing experience.

Solution

Utilize NGINX Plus's bitrate-limiting support for MP4 media files:

```
location /video/ {
    mp4;
    mp4_limit_rate_after 15s;
    mp4_limit_rate       1.2;
}
```

This configuration allows the downstream client to download for 15 seconds before applying a bitrate limit. After 15 seconds, the client is allowed to download media at a rate of 120% of the bitrate, which enables the client to always download faster than they play.

Discussion

NGINX Plus's bitrate limiting allows your streaming server to limit bandwidth dynamically based on the media being served, allowing clients to download just as much as they need to ensure a seamless user experience. The MP4 handler described in Recipe 9.1 designates this location block to stream MP4 media formats. The rate-limiting directives, such as `mp4_limit_rate_after`, tell NGINX to only rate-limit traffic after a specified amount of time, in seconds. The other directive involved in MP4 rate limiting is `mp4_limit_rate`, which specifies the bitrate at which clients are allowed to download in relation to the bitrate of the media. A value of 1 provided to the `mp4_limit_rate` directive specifies that NGINX is to limit bandwidth (1-to-1) to the bitrate of the media. Providing a value of more than one to the `mp4_limit_rate` directive will allow users to download faster than they watch so they can buffer the media and watch seamlessly while they download.

CHAPTER 10

Cloud Deployments

10.0 Introduction

The advent of cloud providers has changed the landscape of web application hosting. A process such as provisioning a new machine used to take hours to months; now, you can create one with as little as a click or API call. These cloud providers lease their virtual machines, called *Infrastructure as a Service* (IaaS), or managed software solutions such as databases, through a pay-per-usage model, which means you pay only for what you use. This enables engineers to build up entire environments for testing at a moment's notice and tear them down when they're no longer needed. These cloud providers also enable applications to scale horizontally based on performance need at a moment's notice. This chapter covers basic NGINX and NGINX Plus deployments on a couple of the major cloud provider platforms.

10.1 Auto-Provisioning on AWS

Problem

You need to automate the configuration of NGINX servers on Amazon Web Services for machines to be able to automatically provision themselves.

Solution

Utilize EC2 `UserData` as well as a prebaked Amazon Machine Image. Create an Amazon Machine Image (AMI) with NGINX and any supporting software packages installed. Utilize Amazon EC2 `UserData` to configure any environment-specific configurations at runtime.

Discussion

There are three patterns of thought when provisioning on Amazon Web Services:

Provision at boot
: Start from a common Linux image, then run configuration management or shell scripts at boot time to configure the server. This pattern is slow to start and can be prone to errors.

Fully baked AMIs
: Fully configure the server, then burn an AMI to use. This pattern boots very fast and accurately. However, it's less flexible to the environment around it, and maintaining many images can be complex.

Partially baked AMIs
: It's a mix of both worlds. Partially baked is where software requirements are installed and burned into an AMI, and environment configuration is done at boot time. This pattern is flexible compared to a fully baked pattern, and fast compared to a provision-at-boot solution.

Whether you choose to partially or fully bake your AMIs, you'll want to automate that process. To construct an AMI build pipeline, it's suggested to use a couple of tools:

Configuration management
: Configuration management tools define the state of the server in code, such as what version of NGINX is to be run and what user it's to run as, what DNS resolver to use, and who to proxy upstream to. This configuration management code can be source controlled and versioned like a software project. Some popular configuration management tools are Ansible, Chef, Puppet, and SaltStack, which were described in Chapter 5.

Packer from HashiCorp
: Packer is used to automate running your configuration management on virtually any virtualization or cloud platform and to burn a machine image if the run is successful. Packer basically builds a virtual machine on the platform of your choosing, SSHes into the virtual machine, runs any provisioning you specify, and burns an image. You can utilize Packer to run the configuration management tool and reliably burn a machine image to your specification.

To provision environmental configurations at boot time, you can utilize the Amazon EC2 `UserData` to run commands the first time the instance is booted. If you're using the partially baked method, you can utilize this to configure environment-based items at boot time. Examples of environment-based configurations might be what server names to listen for, resolver to use, domain name to proxy to, or upstream server pool to start with. `UserData` is a base64-encoded string that is downloaded at the first boot and run. `UserData` can be as simple as an environment file accessed by other bootstrapping processes in your AMI, or it can be a script written in any language that exists on the AMI. It's common for `UserData` to be a bash script that specifies variables or downloads variables to pass to configuration management. Configuration management ensures the system is configured correctly, templates configuration files based on environment variables, and reloads services. After `UserData` runs, your NGINX machine should be completely configured, in a very reliable way.

10.2 Routing to NGINX Nodes Without an AWS ELB

Problem

You need to route traffic to multiple active NGINX nodes or create an active-passive failover set to achieve high availability without a load balancer in front of NGINX.

Solution

Use the Amazon Route53 DNS service to route to multiple active NGINX nodes or configure health checks and failover between an active-passive set of NGINX nodes.

Discussion

DNS has balanced load between servers for a long time; moving to the cloud doesn't change that. The Route53 service from Amazon provides a DNS service with many advanced features, all available through an API. All the typical DNS tricks are available, such as multiple IP addresses on a single A record and weighted A records. When running multiple active NGINX nodes, you'll want to use one of these A record features to spread load across all nodes. The round-robin algorithm is used when multiple IP addresses are listed for a single A record. A weighted distribution can be used to distribute load unevenly by defining weights for each server IP address in an A record.

One of the more interesting features of Route53 is its ability to health check. You can configure Route53 to monitor the health of an endpoint by establishing a TCP connection or by making a request with HTTP or HTTPS. The health check is highly configurable with options for the IP, hostname, port, URI path, interval rates, monitoring, and geography. With these health checks, Route53 can take an IP out of rotation if it begins to fail. You could also configure Route53 to failover to a secondary record in case of a failure, which would achieve an active-passive, highly available setup.

Route53 has a geological-based routing feature that will enable you to route your clients to the closest NGINX node to them, for the least latency. When routing by geography, your client is directed to the closest healthy physical location. When running multiple sets of infrastructure in an active-active configuration, you can automatically failover to another geological location through the use of health checks.

When using Route53 DNS to route your traffic to NGINX nodes in an Auto Scaling group, you'll want to automate the creation and removal of DNS records. To automate adding and removing NGINX machines to Route53 as your NGINX nodes scale, you can use Amazon's Auto Scaling Lifecycle Hooks to trigger scripts within the

NGINX box itself or scripts running independently on Amazon Lambda. These scripts would use the Amazon CLI or SDK to interface with the Amazon Route53 API to add or remove the NGINX machine IP and configured health check as it boots or before it is terminated.

Also See

Amazon Route53 Global Server Load Balancing

10.3 The NLB Sandwich

Problem

You need to autoscale your NGINX Open Source layer and distribute load evenly and easily between application servers.

Solution

Create a *network load balancer* (NLB). During creation of the NLB through the console, you are prompted to create a new target group. If you do not do this through the console, you will need to create this resource and attach it to a listener on the NLB. You create an Auto Scaling group with a launch configuration that provisions an EC2 instance with NGINX Open Source installed. The Auto Scaling group has a configuration to link to the target group, which automatically registers any instance in the Auto Scaling group to the target group configured on first boot. The target group is referenced by a listener on the NLB. Place your upstream applications behind another network load balancer and target group and then configure NGINX to proxy to the application NLB.

Discussion

This common pattern is called the *NLB sandwich* (see Figure 10-1), putting NGINX Open Source in an Auto Scaling group behind an NLB and the application Auto Scaling group behind another NLB. The reason for having NLBs between every layer is because the NLB works so well with Auto Scaling groups; they automatically register new nodes and remove those being terminated as well as run health checks and pass traffic to only healthy nodes. It might be necessary to build a second, internal NLB for the NGINX Open Source layer

because it allows services within your application to call out to other services through the NGINX Auto Scaling group without leaving the network and re-entering through the public NLB. This puts NGINX in the middle of all network traffic within your application, making it the heart of your application's traffic routing. This pattern used to be called the *elastic load balancer* (ELB) *sandwich*; however, the NLB is preferred when working with NGINX because the NLB is a Layer 4 load balancer, whereas ELBs and ALBs are Layer 7 load balancers. Layer 7 load balancers transform the request via the proxy protocol and are redundent with the use of NGINX. This pattern is needed only for NGINX Open Source because the feature set provided by the NLB is available in NGINX Plus.

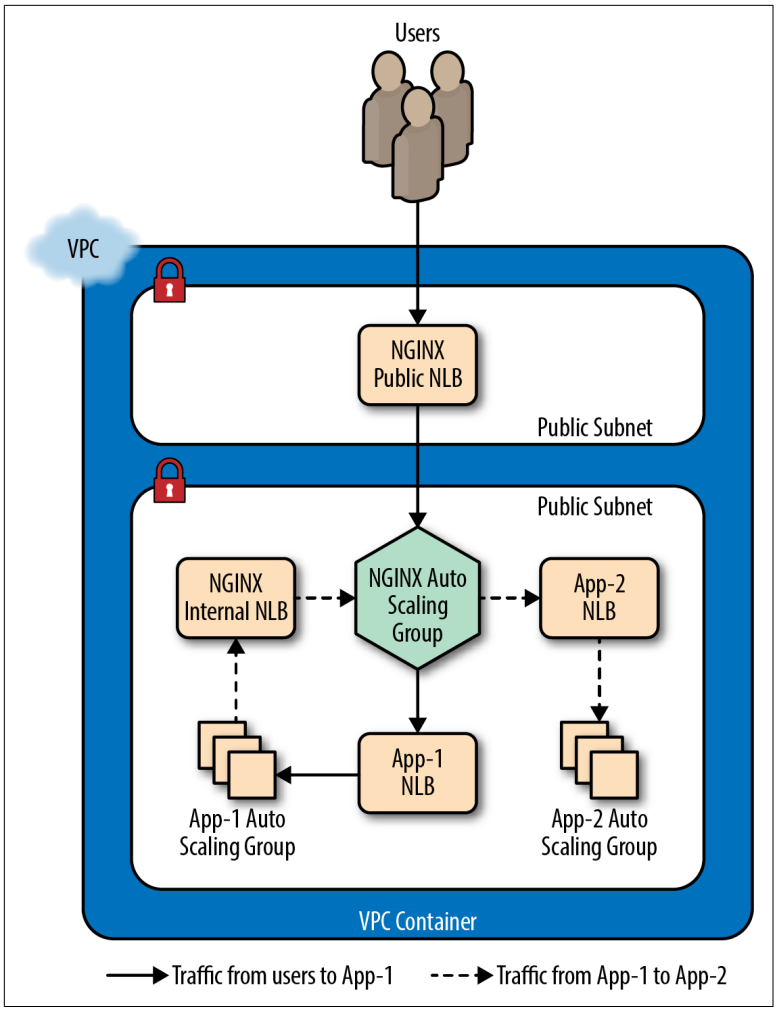

Figure 10-1. This image depicts NGINX in an NLB sandwich pattern with an internal NLB for internal applications to utilize. A user makes a request to App-1, and App-1 makes a request to App-2 through NGINX to fulfill the user's request.

10.4 Deploying from the AWS Marketplace

Problem

You need to run NGINX Plus in AWS with ease with a pay-as-you-go license.

Solution

Deploy through the AWS Marketplace. Visit the AWS Marketplace and search "NGINX Plus" (see Figure 10-2). Select the AMI that is based on the Linux distribution of your choice; review the details, terms, and pricing; then click the Continue link. On the next page you'll be able to accept the terms and deploy NGINX Plus with a single click, or accept the terms and use the AMI.

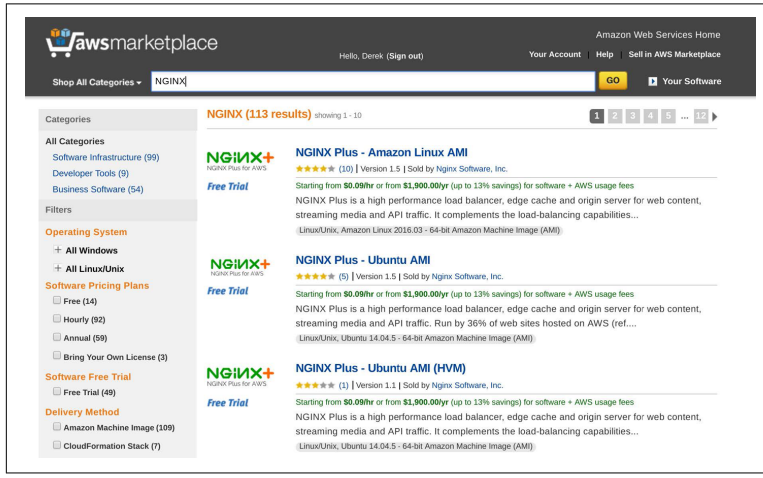

Figure 10-2. Searching for NGINX on the AWS Marketplace

Discussion

The AWS Marketplace solution to deploying NGINX Plus provides ease of use and a pay-as-you-go license. Not only do you have nothing to install, but you also have a license without jumping through hoops like getting a purchase order for a year license. This solution enables you to try NGINX Plus easily without commitment. You can also use the NGINX Plus Marketplace AMI as overflow capacity. It's a common practice to purchase your expected workload worth of licenses and use the Marketplace AMI in an Auto Scaling group as overflow capacity. This strategy ensures you only pay for as much licensing as you use.

10.5 Creating an NGINX Virtual Machine Image on Azure

Problem

You need to create a virtual machine (VM) image of your own NGINX server configured as you see fit to quickly create more servers or use in scale sets.

Solution

Create a VM from a base operating system of your choice. Once the VM is booted, log in and install NGINX or NGINX Plus in your preferred way, either from source or through the package management tool for the distribution you're running. Configure NGINX as desired and create a new VM image. To create a VM image, you must first generalize the VM. To generalize your VM, you need to remove the user that Azure provisioned, connect to it over SSH, and run the following command:

```
$ sudo waagent -deprovision+user -force
```

This command deprovisions the user that Azure provisioned when creating the VM. The `-force` option simply skips a confirmation step. After you've installed NGINX or NGINX Plus and removed the provisioned user, you can exit your session.

Connect your Azure CLI to your Azure account using the Azure login command, then ensure you're using the Azure Resource Manager mode. Now deallocate your VM:

```
$ azure vm deallocate -g <ResourceGroupName> \
                        -n <VirtualMachineName>
```

Once the VM is deallocated, you will be able to generalize it with the `azure vm generalize` command:

```
$ azure vm generalize -g <ResourceGroupName> \
                        -n <VirtualMachineName>
```

After your VM is generalized, you can create an image. The following command will create an image and also generate an Azure Resources Manager (ARM) template for you to use to boot this image:

```
$ azure vm capture <ResourceGroupName> <VirtualMachineName> \
    <ImageNamePrefix> -t <TemplateName>.json
```

The command line will produce output saying that your image has been created, that it's saving an ARM template to the location you specified, and that the request is complete. You can use this ARM template to create another VM from the newly created image. However, to use this template Azure has created, you must first create a new network interface:

```
$ azure network nic create <ResourceGroupName> \
    <NetworkInterfaceName> \
    <Region> \
    --subnet-name <SubnetName> \
    --subnet-vnet-name <VirtualNetworkName>
```

This command output will detail information about the newly created network interface. The first line of the output data will be the network interface ID, which you will need to utilize the ARM template created by Azure. Once you have the ID, you can create a deployment with the ARM template:

```
$ azure group deployment create <ResourceGroupName> \
    <DeploymentName> \
    -f <TemplateName>.json
```

You will be prompted for multiple input variables such as `vmName`, `adminUserName`, `adminPassword`, and `networkInterfaceId`. Enter a name for the VM and the admin username and password. Use the network interface ID harvested from the last command as the input for the `networkInterfaceId` prompt. These variables will be passed as parameters to the ARM template and used to create a new VM from the custom NGINX or NGINX Plus image you've created. After entering the necessary parameters, Azure will begin to create a new VM from your custom image.

Discussion

Creating a custom image in Azure enables you to create copies of your preconfigured NGINX or NGINX Plus server at will. An Azure ARM template enables you to quickly and reliably deploy this same server time and time again as needed. With the VM image path that can be found in the template, you can create different sets of infrastructure such as VM scaling sets or other VMs with different configurations.

Also See

Installing Azure Cross-platform CLI
Azure Cross-platform CLI Login
Capturing Linux Virtual Machine Images

10.6 Load Balancing Over NGINX Scale Sets on Azure

Problem

You need to scale NGINX nodes behind an Azure load balancer to achieve high availability and dynamic resource usage.

Solution

Create an Azure load balancer that is either public facing or internal. Deploy the NGINX virtual machine image created in the prior section or the NGINX Plus image from the Marketplace described in Recipe 10.7 into an Azure virtual machine scale set (VMSS). Once your load balancer and VMSS are deployed, configure a backend pool on the load balancer to the VMSS. Set up load-balancing rules for the ports and protocols you'd like to accept traffic on, and direct them to the backend pool.

Discussion

It's common to scale NGINX to achieve high availability or to handle peak loads without overprovisioning resources. In Azure you achieve this with VMSS. Using the Azure load balancer provides ease of management for adding and removing NGINX nodes to the pool of resources when scaling. With Azure load balancers, you're able to check the health of your backend pools and only pass traffic to healthy nodes. You can run internal Azure load balancers in front of NGINX where you want to enable access only over an internal network. You may use NGINX to proxy to an internal load balancer fronting an application inside of a VMSS, using the load balancer for the ease of registering and deregistering from the pool.

10.7 Deploying Through the Azure Marketplace

Problem

You need to run NGINX Plus in Azure with ease and a pay-as-you-go license.

Solution

Deploy an NGINX Plus VM image through the Azure Marketplace:

1. From the Azure dashboard, select the New icon, and use the search bar to search for "NGINX." Search results will appear.
2. From the list, select the NGINX Plus Virtual Machine Image published by NGINX, Inc.
3. When prompted to choose your deployment model, select the Resource Manager option, and click the Create button.
4. You will then be prompted to fill out a form to specify the name of your VM, the disk type, the default username and password or SSH key-pair public key, which subscription to bill under, the resource group you'd like to use, and the location.
5. Once this form is filled out, you can click OK. Your form will be validated.
6. When prompted, select a VM size, and click the Select button.
7. On the next panel, you have the option to select optional configurations, which will be the default based on your resource group choice made previously. After altering these options and accepting them, click OK.
8. On the next screen, review the summary. You have the option of downloading this configuration as an ARM template so that you can create these resources again more quickly via a JSON template.
9. Once you've reviewed and downloaded your template, you can click OK to move to the purchasing screen. This screen will notify you of the costs you're about to incur from this VM usage. Click Purchase and your NGINX Plus box will begin to boot.

Discussion

Azure and NGINX have made it easy to create an NGINX Plus VM in Azure through just a few configuration forms. The Azure Marketplace is a great way to get NGINX Plus on demand with a pay-as-you-go license. With this model, you can try out the features of NGINX Plus or use it for on-demand overflow capacity of your already licensed NGINX Plus servers.

10.8 Deploying to Google Compute Engine

Problem

You need to create an NGINX server in Google Compute Engine to load balance or proxy for the rest of your resources in Google Compute or App Engine.

Solution

Start a new VM in Google Compute Engine. Select a name for your VM, zone, machine type, and boot disk. Configure identity and access management, firewall, and any advanced configuration you'd like. Create the VM.

Once the VM has been created, log in via SSH or through the Google Cloud Shell. Install NGINX or NGINX Plus through the package manager for the given OS type. Configure NGINX as you see fit and reload.

Alternatively, you can install and configure NGINX through the Google Compute Engine startup script, which is an advanced configuration option when creating a VM.

Discussion

Google Compute Engine offers highly configurable VMs at a moment's notice. Starting a VM takes little effort and enables a world of possibilities. Google Compute Engine offers networking and compute in a virtualized cloud environment. With a Google Compute instance, you have the full capabilities of an NGINX server wherever and whenever you need it.

10.9 Creating a Google Compute Image

Problem

You need to create a Google Compute Image to quickly instantiate a VM or create an instance template for an instance group.

Solution

Create a VM as described in Recipe 10.8. After installing and configuring NGINX on your VM instance, set the auto-delete state of the boot disk to `false`. To set the auto-delete state of the disk, edit the VM. On the Edit page under the disk configuration is a checkbox labeled "Delete boot disk when instance is deleted." Deselect this checkbox and save the VM configuration. Once the auto-delete state of the instance is set to `false`, delete the instance. When prompted, do not select the checkbox that offers to delete the boot disk. By performing these tasks, you will be left with an unattached boot disk with NGINX installed.

After your instance is deleted and you have an unattached boot disk, you can create a Google Compute Image. From the Image section of the Google Compute Engine console, select Create Image. You will be prompted for an image name, family, description, encryption type, and the source. The source type you need to use is disk; and for the source disk, select the unattached NGINX boot disk. Select Create and Google Compute Cloud will create an image from your disk.

Discussion

You can utilize Google Cloud Images to create VMs with a boot disk identical to the server you've just created. The value in creating images is being able to ensure that every instance of this image is identical. When installing packages at boot time in a dynamic environment, unless using version locking with private repositories, you run the risk of package version and updates not being validated before being run in a production environment. With machine images, you can validate that every package running on this machine is exactly as you tested, strengthening the reliability of your service offering.

Also See

Create, Delete, and Depreciate Private Images

10.10 Creating a Google App Engine Proxy

Problem

You need to create a proxy for Google App Engine to context switch between applications or serve HTTPS under a custom domain.

Solution

Utilize NGINX in Google Compute Cloud. Create a virtual machine in Google Compute Engine, or create a virtual machine image with NGINX installed and create an instance template with this image as your boot disk. If you've created an instance template, follow up by creating an instance group that utilizes that template.

Configure NGINX to proxy to your Google App Engine endpoint. Make sure to proxy to HTTPS because Google App Engine is public, and you'll want to ensure you do not terminate HTTPS at your NGINX instance and allow information to travel between NGINX and Google App Engine unsecured. Because App Engine provides just a single DNS endpoint, you'll be using the `proxy_pass` directive rather than upstream blocks in the open source version of NGINX. When proxying to Google App Engine, make sure to set the endpoint as a variable in NGINX, then use that variable in the `proxy_pass` directive to ensure NGINX does DNS resolution on every request. For NGINX to do any DNS resolution, you'll need to also utilize the `resolver` directive and point to your favorite DNS resolver. Google makes the IP address `8.8.8.8` available for public use. If you're using NGINX Plus, you'll be able to use the `resolve` flag on the `server` directive within the upstream block, keepalive connections, and other benefits of the upstream module when proxying to Google App Engine.

You may choose to store your NGINX configuration files in Google Storage, then use the startup script for your instance to pull down the configuration at boot time. This will allow you to change your configuration without having to burn a new image. However, it will add to the startup time of your NGINX server.

Discussion

You want to run NGINX in front of Google App Engine if you're using your own domain and want to make your application available via HTTPS. At this time, Google App Engine does not allow you to upload your own SSL certificates. Therefore, if you'd like to serve your app under a domain other than appspot.com with encryption, you'll need to create a proxy with NGINX to listen at your custom domain. NGINX will encrypt communication between itself and your clients, as well as between itself and Google App Engine.

Another reason you may want to run NGINX in front of Google App Engine is to host many App Engine apps under the same domain and use NGINX to do URI-based context switching. Microservices are a popular architecture, and it's common for a proxy like NGINX to conduct the traffic routing. Google App Engine makes it easy to deploy applications, and in conjunction with NGINX, you have a full-fledged application delivery platform.

CHAPTER 11

Containers/Microservices

11.0 Introduction

Containers offer a layer of abstraction at the application layer, shifting the installation of packages and dependencies from the deploy to the build process. This is important because engineers are now shipping units of code that run and deploy in a uniform way regardless of the environment. Promoting containers as runnable units reduces the risk of dependency and configuration snafus between environments. Given this, there has been a large drive for organizations to deploy their applications on container platforms. When running applications on a container platform, it's common to containerize as much of the stack as possible, including your proxy or load balancer. NGINX and NGINX Plus containerize and ship with ease. They also include many features that make delivering containerized applications fluid. This chapter focuses on building NGINX and NGINX Plus container images, features that make working in a containerized environment easier, and deploying your image on Kubernetes and OpenShift.

11.1 DNS SRV Records

Problem

You'd like to use your existing DNS SRV record implementation as the source for upstream servers with NGINX Plus.

Solution

Specify the service directive with a value of `http` on an upstream server to instruct NGINX to utilize the SRV record as a load-balancing pool:

```
http {
    resolver 10.0.0.2;

    upstream backend {
        zone backends 64k;
        server api.example.internal service=http resolve;
    }
}
```

This feature is an NGINX Plus exclusive. The configuration instructs NGINX Plus to resolve DNS from a DNS server at 10.0.0.2 and set up an upstream server pool with a single `server` directive. This `server` directive specified with the `resolve` parameter is instructed to periodically re-resolve the domain name. The `service=http` parameter and value tells NGINX that this is an SRV record containing a list of IPs and ports and to load balance over them as if they were configured with the `server` directive.

Discussion

Dynamic infrastructure is becoming ever more popular with the demand and adoption of cloud-based infrastructure. Autoscaling environments scale horizontally, increasing and decreasing the number of servers in the pool to match the demand of the load. Scaling horizontally demands a load balancer that can add and remove resources from the pool. With an SRV record, you offload the responsibility of keeping the list of servers to DNS. This type of configuration is extremely enticing for containerized environments because you may have containers running applications on variable port numbers, possibly at the same IP address. It's important to note that UDP DNS record payload is limited to about 512 bytes.

11.2 Using the Official NGINX Image

Problem

You need to get up and running quickly with the NGINX image from Docker Hub.

Solution

Use the NGINX image from Docker Hub. This image contains a default configuration. You'll need to either mount a local configuration directory or create a Dockerfile and `ADD` in your configuration to the image build to alter the configuration. Here, we mount a volume where NGINX's default configuration serves static content to demonstrate its capabilities by using a single command:

```
$ docker run --name my-nginx -p 80:80 \
    -v /path/to/content:/usr/share/nginx/html:ro -d nginx
```

The `docker` command pulls the `nginx:latest` image from Docker Hub if it's not found locally. The command then runs this NGINX image as a Docker container, mapping `localhost:80` to port `80` of the NGINX container. It also mounts the local directory */path/to/content/* as a container volume at */usr/share/nginx/html/* as read only. The default NGINX configuration will serve this directory as static content. When specifying mapping from your local machine to a container, the local machine port or directory comes first, and the container port or directory comes second.

Discussion

NGINX has made an official Docker image available via Docker Hub. This official Docker image makes it easy to get up and going very quickly in Docker with your favorite application delivery platform, NGINX. In this section, we were able to get NGINX up and running in a container with a single command! The official NGINX Docker image mainline that we used in this example is built off of the Debian Jessie Docker image. However, you can choose official images built off of Alpine Linux. The Dockerfile and source for these official images are available on GitHub. You can extend the official image by building your own Dockerfile and specifying the official image in the `FROM` command.

Also See

Official NGINX Docker image, NGINX
Docker repo on GitHub

11.3 Creating an NGINX Dockerfile

Problem

You need to create an NGINX Dockerfile in order to create a Docker image.

Solution

Start `FROM` your favorite distribution's Docker image. Use the `RUN` command to install NGINX. Use the `ADD` command to add your NGINX configuration files. Use the `EXPOSE` command to instruct Docker to expose given ports or do this manually when you run the image as a container. Use `CMD` to start NGINX when the image is instantiated as a container. You'll need to run NGINX in the foreground. To do this, you'll need to start NGINX with `-g "daemon off;"` or add `daemon off;` to your configuration. This example will use the latter with `daemon off;` in the configuration file within the main context. You will also want to alter your NGINX configuration to log to */dev/stdout* for access logs and */dev/stderr* for error logs; doing so will put your logs into the hands of the Docker daemon, which will make them available to you more easily based on the log driver you've chosen to use with Docker:

Dockerfile:

```
FROM centos:7

# Install epel repo to get nginx and install nginx
RUN yum -y install epel-release && \
    yum -y install nginx

# add local configuration files into the image
ADD /nginx-conf /etc/nginx

EXPOSE 80 443

CMD ["nginx"]
```

The directory structure looks as follows:

```
.
├── Dockerfile
└── nginx-conf
    ├── conf.d
    │   └── default.conf
    ├── fastcgi.conf
```

```
├── fastcgi_params
├── koi-utf
├── koi-win
├── mime.types
├── nginx.conf
├── scgi_params
├── uwsgi_params
└── win-utf
```

I chose to host the entire NGINX configuration within this Docker directory for ease of access to all of the configurations with only one line in the Dockerfile to add all my NGINX configurations.

Discussion

You will find it useful to create your own Dockerfile when you require full control over the packages installed and updates. It's common to keep your own repository of images so that you know your base image is reliable and tested by your team before running it in production.

11.4 Building an NGINX Plus Image

Problem

You need to build an NGINX Plus Docker image to run NGINX Plus in a containerized environment.

Solution

Use this Dockerfile to build an NGINX Plus Docker image. You'll need to download your NGINX Plus repository certificates and keep them in the directory with this Dockerfile named *nginx-repo.crt* and *nginx-repo.key*, respectively. With that, this Dockerfile will do the rest of the work installing NGINX Plus for your use and linking NGINX access and error logs to the Docker log collector.

```
FROM debian:stretch-slim

LABEL maintainer="NGINX <docker-maint@nginx.com>"

# Download certificate and key from the customer portal
# (https://cs.nginx.com) and copy to the build context

COPY nginx-repo.crt /etc/ssl/nginx/
COPY nginx-repo.key /etc/ssl/nginx/
```

```
# Install NGINX Plus
RUN set -x \
  && APT_PKG="Acquire::https::plus-pkgs.nginx.com::" \
  && REPO_URL="https://plus-pkgs.nginx.com/debian" \
  && apt-get update && apt-get upgrade -y \
  && apt-get install \
    --no-install-recommends --no-install-suggests\
    -y apt-transport-https ca-certificates gnupg1 \
  && \
  NGINX_GPGKEY=573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62;\
  found=''; \
  for server in \
    ha.pool.sks-keyservers.net \
    hkp://keyserver.ubuntu.com:80 \
    hkp://p80.pool.sks-keyservers.net:80 \
    pgp.mit.edu \
  ; do \
   echo "Fetching GPG key $NGINX_GPGKEY from $server"; \
   apt-key adv --keyserver "$server" --keyserver-options \
      timeout=10 --recv-keys "$NGINX_GPGKEY" \
    && found=yes \
    && break;\
  done;\
  test -z "$found" && echo >&2 \
    "error: failed to fetch GPG key $NGINX_GPGKEY" && exit 1; \
  echo "${APT_PKG}Verify-Peer "true";"\
    >> /etc/apt/apt.conf.d/90nginx \
  && echo \
    "${APT_PKG}Verify-Host "true";">>\
    /etc/apt/apt.conf.d/90nginx \
  && echo "${APT_PKG}SslCert     \
    "/etc/ssl/nginx/nginx-repo.crt";" >> \
    /etc/apt/apt.conf.d/90nginx \
  && echo "${APT_PKG}SslKey      \
    "/etc/ssl/nginx/nginx-repo.key";" >> \
      /etc/apt/apt.conf.d/90nginx \
  && printf \
    "deb ${REPO_URL} stretch nginx-plus" \
    > /etc/apt/sources.list.d/nginx-plus.list \
  && apt-get update && apt-get install -y nginx-plus \
  && apt-get remove --purge --auto-remove -y gnupg1 \
  && rm -rf /var/lib/apt/lists/*

# Forward request logs to Docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
  && ln -sf /dev/stderr /var/log/nginx/error.log

EXPOSE 80
STOPSIGNAL SIGTERM
```

```
CMD ["nginx", "-g", "daemon off;"]
```

To build this Dockerfile into a Docker image, run the following in the directory that contains the Dockerfile and your NGINX Plus repository certificate and key:

```
$ docker build --no-cache -t nginxplus .
```

This `docker build` command uses the flag `--no-cache` to ensure that whenever you build this, the NGINX Plus packages are pulled fresh from the NGINX Plus repository for updates. If it's acceptable to use the same version on NGINX Plus as the prior build, you can omit the `--no-cache` flag. In this example, the new Docker image is tagged `nginxplus`.

Discussion

By creating your own Docker image for NGINX Plus, you can configure your NGINX Plus container however you see fit and drop it into any Docker environment. This opens up all of the power and advanced features of NGINX Plus to your containerized environment. This Dockerfile does not use the Dockerfile property `ADD` to add in your configuration; you will need to add in your configuration manually.

Also See

NGINX blog on Docker images

11.5 Using Environment Variables in NGINX

Problem

You need to use environment variables inside your NGINX configuration in order to use the same container image for different environments.

Solution

Use the `ngx_http_perl_module` to set variables in NGINX from your environment:

```
daemon off;
env APP_DNS;
```

```
include /usr/share/nginx/modules/*.conf;
...
http {
  perl_set $upstream_app 'sub { return $ENV{"APP_DNS"}; }';
  server {
    ...
    location / {
      proxy_pass https://$upstream_app;
    }
  }
}
```

To use `perl_set` you must have the `ngx_http_perl_module` installed; you can do so by loading the module dynamically or statically if building from source. NGINX by default wipes environment variables from its environment; you need to declare any variables you do not want removed with the `env` directive. The `perl_set` directive takes two parameters: the variable name you'd like to set and a perl string that renders the result.

The following is a Dockerfile that loads the `ngx_http_perl_module` dynamically, installing this module from the package management utility. When installing modules from the package utility for CentOS, they're placed in the */usr/lib64/nginx/modules/* directory, and configuration files that dynamically load these modules are placed in the */usr/share/nginx/modules/* directory. This is why in the preceding configuration snippet we include all configuration files at that path:

```
FROM centos:7

# Install epel repo to get nginx and install nginx
RUN yum -y install epel-release && \
    yum -y install nginx nginx-mod-http-perl

# add local configuration files into the image
ADD /nginx-conf /etc/nginx

EXPOSE 80 443

CMD ["nginx"]
```

Discussion

A typical practice when using Docker is to utilize environment variables to change the way the container operates. You can use environment variables in your NGINX configuration so that your NGINX Dockerfile can be used in multiple, diverse environments.

11.6 Kubernetes Ingress Controller

Problem

You are deploying your application on Kubernetes and need an ingress controller.

Solution

Ensure that you have access to the ingress controller image. For NGINX, you can use the *nginx/nginx-ingress* image from DockerHub. For NGINX Plus, you will need to build your own image and host it in your private Docker registry. You can find instructions on building and pushing your own NGINX Plus Kubernetes Ingress Controller on NGINX Inc's GitHub.

Visit the Kubernetes Ingress Controller Deployments folder in the kubernetes-ingress repository on GitHub. The commands that follow will be run from within this directory of a local copy of the repository.

Create a namespace and a service account for the ingress controller; both are named `nginx-ingress`:

```
$ kubectl apply -f common/ns-and-sa.yaml
```

Create a secret with a TLS certificate and key for the ingress controller:

```
$ kubectl apply -f common/default-server-secret.yaml
```

This certificate and key are self-signed and created by NGINX Inc. for testing and example purposes. It's recommended to use your own because this key is publicly available.

Optionally, you can create a config map for customizing NGINX configuration (the config map provided is blank; however, you can read more about customization and annotation here):

```
$ kubectl apply -f common/nginx-config.yaml
```

If Role-Based Access Control (RBAC) is enabled in your cluster, create a cluster role and bind it to the service account. You must be a cluster administrator to perform this step:

```
$ kubectl apply -f rbac/rbac.yaml
```

Now deploy the ingress controller. Two example deployments are made available in this repository: a Deployment and a DaemonSet. Use a Deployment if you plan to dynamically change the number of ingress controller replicas. Use a DaemonSet to deploy an ingress controller on every node or a subset of nodes.

If you plan to use the NGINX Plus Deployment manifests, you must alter the YAML file and specify your own registry and image.

For NGINX Deployment:

```
$ kubectl apply -f deployment/nginx-ingress.yaml
```

For NGINX Plus Deployment:

```
$ kubectl apply -f deployment/nginx-plus-ingress.yaml
```

For NGINX DaemonSet:

```
$ kubectl apply -f daemon-set/nginx-ingress.yaml
```

For NGINX Plus DaemonSet:

```
$ kubectl apply -f daemon-set/nginx-plus-ingress.yaml
```

Validate that the ingress controller is running:

```
$ kubectl get pods --namespace=nginx-ingress
```

If you created a DaemonSet, port `80` and `443` of the ingress controller are mapped to the same ports on the node where the container is running. To access the ingress controller, use those ports and the IP address of any of the nodes on which the ingress controller is running. If you deployed a Deployment, continue with the next steps.

For the Deployment methods, there are two options for accessing the ingress controller pods. You can instruct Kubernetes to randomly assign a node port that maps to the ingress controller pod. This is a service with the type `NodePort`. The other option is to create a service with the type `LoadBalancer`. When creating a service of type `LoadBalancer`, Kubernetes builds a load balancer for the given cloud platform, such as Amazon Web Services, Microsoft Azure, and Google Cloud Compute.

To create a service of type `NodePort`, use the following:

```
$ kubectl create -f service/nodeport.yaml
```

To statically configure the port that is opened for the pod, alter the YAML and add the attribute `nodePort: {port}` to the configuration of each port being opened.

To create a service of type `LoadBalancer` for Google Cloud Compute or Azure, use this code:

```
$ kubectl create -f service/loadbalancer.yaml
```

To create a service of type `LoadBalancer` for Amazon Web Services:

```
$ kubectl create -f service/loadbalancer-aws-elb.yaml
```

On AWS, Kubernetes creates a classic ELB in TCP mode with the PROXY protocol enabled. You must configure NGINX to use the PROXY protocol. To do so, you can add the following to the config map mentioned previously in reference to the file *common/nginx-config.yaml*.

```
proxy-protocol: "True"
real-ip-header: "proxy_protocol"
set-real-ip-from: "0.0.0.0/0"
```

Then, update the config map:

```
$ kubectl apply -f common/nginx-config.yaml
```

You can now address the pod by its `NodePort` or by making a request to the load balancer created on its behalf.

Discussion

As of this writing, Kubernetes is the leading platform in container orchestration and management. The ingress controller is the edge pod that routes traffic to the rest of your application. NGINX fits this role perfectly and makes it simple to configure with its annotations. The NGINX-Ingress project offers an NGINX Open Source ingress controller out of the box from a DockerHub image, and NGINX Plus through a few steps to add your repository certificate and key. Enabling your Kubernetes cluster with an NGINX ingress controller provides all the same features of NGINX but with the added features of Kubernetes networking and DNS to route traffic.

11.7 OpenShift Router

Problem

You are deploying your application on OpenShift and would like to use NGINX as a router.

Solution

Build the Router image and upload it to your private registry. You can find the source files for the image in the Origin Repository. It's important to push your Router image to the private registry before deleting the default Router because it will render the registry unavailable.

Log in to the OpenShift Cluster as an admin:

```
$ oc login -u system:admin
```

Select the `default` project:

```
$ oc project default
```

Back up the default Router config, in case you need to recreate it:

```
$ oc get -o yaml service/router dc/router \
    clusterrolebinding/router-router-role \
    serviceaccount/router > default-router-backup.yaml
```

Delete the Router:

```
$ oc delete -f default-router-backup.yaml
```

Deploy the NGINX Router:

```
$ oc adm router router --images={image} --type='' \
    --selector='node-role.kubernetes.io/infra=true'
```

In this example, the `{image}` must point to the NGINX Router image in your registry. The selector parameter specifies a label selector for nodes where the Router will be deployed: `node-role.kubernetes.io/infra=true`. Use a selector that makes sense for your environment.

Validate that your NGINX Router pods are running:

```
$ oc get pods
```

You should see a Router pod with the name `router-1-{string}`.

By default, the NGINX stub status page is available via port `1936` of the node where the Router is running (you can change this port by using the `STATS_PORT` env variable). To access the page outside of the node, you need to add an entry to the IPtables rules for that node:

```
$ sudo iptables -I OS_FIREWALL_ALLOW -p tcp -s {ip range} \
    -m tcp --dport 1936 -j ACCEPT
```

Open your browser to *http://{node-ip}:1936/stub_status* to access the stub status page.

Discussion

The OpenShift Router is the entry point for external requests bound for applications running on OpenShift. The Router's job is to receive incoming requests and direct them to the appropriate application pod. The load-balancing and routing abilities of NGINX make it a great choice for use as an OpenShift Router. Switching out the default OpenShift Router for an NGINX Router enables all of the features and power of NGINX as the ingress of your OpenStack application.

CHAPTER 12

High-Availability Deployment Modes

12.0 Introduction

Fault-tolerant architecture separates systems into identical, independent stacks. Load balancers like NGINX are employed to distribute load, ensuring that what's provisioned is utilized. The core concepts of high availability are load balancing over multiple active nodes or an active-passive failover. Highly available applications have no single points of failure; every component must use one of these concepts, including the load balancers themselves. For us, that means NGINX. NGINX is designed to work in either configuration: multiple active or active-passive failover. This chapter details techniques on how to run multiple NGINX servers to ensure high availability in your load-balancing tier.

12.1 NGINX HA Mode

Problem

You need a highly available load-balancing solution.

Solution

Use NGINX Plus's highly available (HA) mode with keepalived by installing the `nginx-ha-keepalived` package from the NGINX Plus repository.

Discussion

The `nginx-ha-keepalived` package is based on keepalived and manages a virtual IP address exposed to the client. Another process is run on the NGINX server that ensures that NGINX Plus and the keepalived process are running. Keepalived is a process that utilizes the Virtual Router Redundancy Protocol (VRRP), sending small messages often referred to as heartbeats, to the backup server. If the backup server does not receive the heartbeat for three consecutive periods, the backup server initiates the failover, moving the virtual IP address to itself and becoming the master. The failover capabilities of `nginx-ha-keepalived` can be configured to identify custom failure situations.

12.2 Load-Balancing Load Balancers with DNS

Problem

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

Solution

Use DNS to round robin across NGINX servers by adding multiple IP addresses to a DNS A record.

Discussion

When running multiple load balancers, you can distribute load via DNS. The A record allows for multiple IP addresses to be listed under a single, fully qualified domain name. DNS will automatically round robin across all the IPs listed. DNS also offers weighted round robin with weighted records, which works in the same way as weighted round robin in NGINX as described in Chapter 1. These techniques work great. However, a pitfall can be removing the record when an NGINX server encounters a failure. There are DNS

providers—Amazon Route53 for one, and Dyn DNS for another—that offer health checks and failover with their DNS offering, which alleviates these issues. If you are using DNS to load balance over NGINX, when an NGINX server is marked for removal, it's best to follow the same protocols that NGINX does when removing an upstream server. First, stop sending new connections to it by removing its IP from the DNS record, then allow connections to drain before stopping or shutting down the service.

12.3 Load Balancing on EC2

Problem

You're using NGINX on AWS, and the NGINX Plus HA does not support Amazon IPs.

Solution

Put NGINX behind an AWS NLB by configuring an Auto Scaling group of NGINX servers and linking the Auto Scaling group to a target group and then attach the target group to the NLB. Alternatively, you can place NGINX servers into the target group manually by using the AWS console, command-line interface, or API.

Discussion

The HA solution from NGINX Plus based on keepalived will not work on AWS because it does not support the floating virtual IP address, since EC2 IP addresses work in a different way. This does not mean that NGINX can't be HA in the AWS cloud; in fact, the opposite is true. The AWS NLB is a product offering from Amazon that will natively load balance over multiple, physically separated data centers called *availability zones*, provide active health checks, and a DNS CNAME endpoint. A common solution for HA NGINX on AWS is to put an NGINX layer behind the NLB. NGINX servers can be automatically added to and removed from the target group as needed. The NLB is not a replacement for NGINX; there are many things NGINX offers that the NLB does not, such as multiple load-balancing methods, rate limiting, caching, and Layer 7 routing. The AWS ALB does perform Layer 7 load balancing based on the URI path and host heade, but it does not by itself offer features NGINX

does, such as WAF caching, bandwidth limiting, HTTP/2 server push, and more. In the event that the NLB does not fit your need, there are many other options. One option is the DNS solution, Route53. The DNS product from AWS offers health checks and DNS failover.

12.4 Configuration Synchronization

Problem

You're running a HA NGINX Plus tier and need to synchronize configuration across servers.

Solution

Use the NGINX Plus exclusive configuration synchronization feature. To configure this feature, follow these steps:

Install the `nginx-sync` package from the NGINX Plus package repository.

For RHEL or CentOS:

```
$ sudo yum install nginx-sync
```

For Ubuntu or Debian:

```
$ sudo apt-get install nginx-sync
```

Grant the master machine SSH access as root to the peer machines.

Generate an SSH authentication key pair for root and retrieve the public key:

```
$ sudo ssh-keygen -t rsa -b 2048
$ sudo cat /root/.ssh/id_rsa.pub
ssh-rsa AAAAB3Nz4rFgt...vgaD root@node1
```

Get the IP address of the master node:

```
$ ip addr
1: lo:  mtu 65536 qdisc noqueue state UNKNOWN group default
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
      valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
      valid_lft forever preferred_lft forever
2: eth0:  mtu 1500 qdisc pfifo_fast state UP group default qlen \
     1000
   link/ether 52:54:00:34:6c:35 brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.1.2/24 brd 192.168.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:fe34:6c35/64 scope link
       valid_lft forever preferred_lft forever
```

The `ip addr` command will dump information about interfaces on the machine. Disregard the loopback interface, which is normally the first. Look for the IP address following `inet` for the primary interface. In this example, the IP address is `192.168.1.2`.

Distribute the public key to the root user's `authorized_keys` file on each peer node, and specify to authorize only from the master's IP address:

```
$ sudo echo 'from="192.168.1.2" ssh-rsa AAAAB3Nz4rFgt...vgaD \
    root@node1' >> /root/.ssh/authorized_keys
```

Add the following line to */etc/ssh/sshd_config* and reload `sshd` on all nodes:

```
$ sudo echo 'PermitRootLogin without-password' >> \
    /etc/ssh/sshd_config
$ sudo service sshd reload
```

Verify that the root user on the master node can `ssh` to each of the peer nodes without a password:

```
$ sudo ssh root@node2.example.com
```

Create the configuration file */etc/nginx-sync.conf* on the master machine with the following configuration:

```
NODES="node2.example.com node3.example.com node4.example.com"
CONFPATHS="/etc/nginx/nginx.conf /etc/nginx/conf.d"
EXCLUDE="default.conf"
```

This example configuration demonstrates the three common configuration parameters for this feature: `NODES`, `CONFIGPATHS`, and `EXCLUDE`. The `NODES` parameter is set to a string of hostnames or IP addresses separated by spaces; these are the peer nodes to which the master will push its configuration changes. The `CONFIGPATHS` parameter denotes which files or directories should be synchronized. Lastly, you can use the `EXCLUDE` parameter to exclude configuration files from synchronization. In our example, the master pushes configuration changes of the main NGINX configuration file and includes the directory */etc/nginx/nginx.conf* and */etc/nginx/conf.d* to peer nodes named `node2.example.comnode3.example.com` and `node4.example.com`. If the synchronization process finds a file

named *default.conf*, it will not be pushed to the peers, because it's configured as an `EXCLUDE`.

There are advanced configuration parameters to configure the location of the NGINX binary, RSYNC binary, SSH binary, diff binary, lockfile location, and backup directory. There is also a parameter that utilizes `sed` to template given files. For more information about the advanced parameters, see Configuration Sharing.

Test your configuration:

```
$ nginx-sync.sh -h # display usage info
$ nginx-sync.sh -c node2.example.com # compare config to node2
$ nginx-sync.sh -C # compare master config to all peers
$ nginx-sync.sh # sync the config & reload NGINX on peers
```

Discussion

This NGINX Plus exclusive feature enables you to manage multiple NGINX Plus servers in a highly available configuration by updating only the master node and synchronizing the configuration to all other peer nodes. By automating the synchronization of configuration, you limit the risk of mistakes when transferring configurations. The `nginx-sync.sh` application provides some safeguards to prevent sending bad configurations to the peers. They include testing the configuration on the master, creating backups of the configuration on the peers, and validating the configuration on the peer before reloading. Although it's preferable to synchronize your configuration by using configuration management tools or Docker, the NGINX Plus configuration synchronization feature is valuable if you have yet to make the big leap to managing environments in this way.

12.5 State Sharing with Zone Sync

Problem

You need NGINX Plus to synchronize its shared memory zones across a fleet of highly available servers.

Solution

Use the `sync` parameter when configuring an NGINX Plus shared memory zone:

```
stream {
    resolver 10.0.0.2 valid=20s;

    server {
        listen 9000;
        zone_sync;
        zone_sync_server nginx-cluster.example.com:9000 resolve;
        # ... Security measures
    }
}
http {
    upstream my_backend {
        zone my_backend 64k;
        server backends.example.com resolve;
        sticky learn zone=sessions:1m
                     create=$upstream_cookie_session
                     lookup=$cookie_session
                     sync;
    }

    server {
        listen 80;
        location / {
            proxy_pass http://my_backend;
        }
    }
}
```

Discussion

The `zone sync` module is an NGINX Plus exclusive feature that enables NGINX Plus to truly cluster. As shown in the configuration, you must set up a `stream` server configured as the `zone_sync`. In the example, this is the server listening on port `9000`. NGINX Plus communicates with the rest of the servers defined by the `zone_sync_server` directive. You can set this directive to a domain name that resolves to multiple IP addresses for dynamic clusters, or statically define a series of `zone_sync_server` directives. You should restrict access to the zone sync server; there are specific SSL/TLS directives for this module for machine authentication. The benefit of configuring NGINX Plus into a cluster is that you can synchronize shared memory zones for rate limiting, sticky learn sessions, and the key-value store. The example provided shows the `sync` parameter tacked on to the end of a `sticky learn` directive. In this example, a user is bound to an upstream server based on a cookie named `session`. Without the zone sync module if a user makes a request to a

different NGINX Plus server, he could lose his session. With the zone sync module, all of the NGINX Plus servers are aware of the session and to which upstream server it's bound.

CHAPTER 13

Advanced Activity Monitoring

13.0 Introduction

To ensure that your application is running at optimal performance and precision, you need insight into the monitoring metrics about its activity. NGINX Plus offers an advanced monitoring dashboard and a JSON feed to provide in-depth monitoring about all requests that come through the heart of your application. The NGINX Plus activity monitoring provides insight into requests, upstream server pools, caching, health, and more. This chapter details the power and possibilities of the NGINX Plus dashboard, the NGINX Plus API, and the Open Source stub status module.

13.1 Enable NGINX Open Source Stub Status

Problem

You need to enable basic monitoring for NGINX.

Solution

Enable the `stub_status` module in a location block within a NGINX HTTP server:

```
location /stub_status {
    stub_status;
    allow 127.0.0.1;
    deny all;
```

```
        # Set IP restrictions as appropriate
    }
```

Test your configuration by making a request for the status:

```
$ curl localhost/stub_status
Active connections: 1
server accepts handled requests
 1 1 1
Reading: 0 Writing: 1 Waiting: 0
```

Discussion

The `stub_status` module enables some basic monitoring of the Open Source NGINX server. The information that is returned provides insight into the number of active connections as well as the total connections accepted, connections handled, and requests served. The current number of connections being read, written, or in a waiting state is also shown. The information provided is global and is not specific to the parent `server` where the `stub_status` directive is defined. This means that you can host the status on a protected `server`. This module provides active connection counts as embedded variables for use in logs and elsewhere. These variables are `$connections_active`, `$connections_reading`, `$connections_writing`, and `$connections_waiting`.

13.2 Enabling the NGINX Plus Monitoring Dashboard Provided by NGINX Plus

Problem

You require in-depth metrics about the traffic flowing through your NGINX Plus server.

Solution

Utilize the real-time activity monitoring dashboard:

```
server {
    # ...
    location /api {
        api [write=on];
        # Directives limiting access to the API
        # See chapter 7
    }
```

```
        location = /dashboard.html {
            root   /usr/share/nginx/html;
        }
    }
```

The NGINX Plus configuration serves the NGINX Plus status monitoring dashboard. This configuration sets up an HTTP server to serve the API and the status dashboard. The dashboard is served as static content out of the */usr/share/nginx/html* directory. The dashboard makes requests to the API at */api/* in order to retrieve and display the status in real time.

Discussion

NGINX Plus provides an advanced status monitoring dashboard. This status dashboard provides a detailed status of the NGINX system, such as number of active connections, uptime, upstream server pool information, and more. For a glimpse of the console, see Figure 13-1.

The landing page of the status dashboard provides an overview of the entire system. Clicking into the Server Zones tab lists details about all HTTP servers configured in the NGINX configuration, detailing the number of responses from 1XX to 5XX and an overall total as well as requests per second and the current traffic throughput. The Upstream tab details upstream server status, as if it were in a failed state, how many requests it has served, and a total of how many responses have been served by status code, as well as other statistics such as how many health checks it has passed or failed. The TCP/UDP Zones tab details the amount of traffic flowing through the TCP or UDP streams and the number of connections. The TCP/UDP Upstream tab shows information about how much each of the upstream servers in the TCP/UDP upstream pools is serving, as well as health check pass and fail details and response times. The Caches tab displays information about the amount of space utilized for cache; the amount of traffic served, written, and bypassed; as well as the hit ratio. The NGINX status dashboard is invaluable in monitoring the heart of your applications and traffic flow.

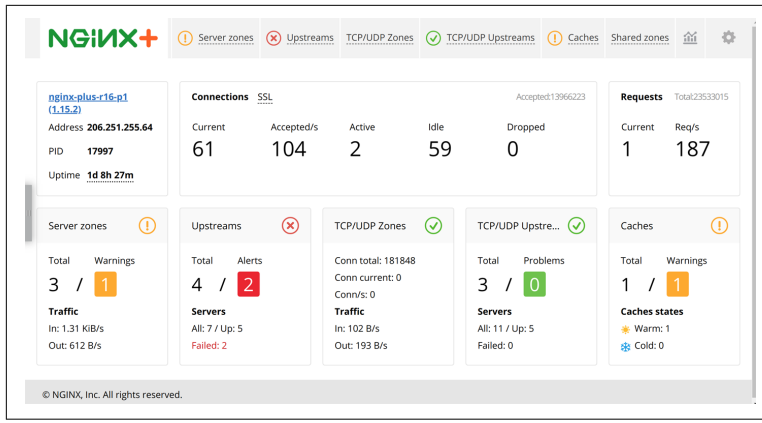

Figure 13-1. The NGINX Plus status dashboard

Also See

NGINX Plus Status Dashboard Demo

13.3 Collecting Metrics Using the NGINX Plus API

Problem

You need API access to the detail metrics provided by the NGINX Plus status dashboard.

Solution

Utilize the RESTful API to collect metrics. The examples pipe the output through `json_pp` to make them easier to read:

```
$ curl "demo.nginx.com/api/3/" | json_pp
[
   "nginx",
   "processes",
   "connections",
   "ssl",
   "slabs",
   "http",
   "stream"
]
```

The `curl` call requests the top level of the API, which displays other portions of the API.

To get information about the NGINX Plus server, use the `/api/{version}/nginx` URI:

```
$ curl "demo.nginx.com/api/3/nginx" | json_pp
{
   "version" : "1.15.2",
   "ppid" : 79909,
   "build" : "nginx-plus-r16",
   "pid" : 77242,
   "address" : "206.251.255.64",
   "timestamp" : "2018-09-29T23:12:20.525Z",
   "load_timestamp" : "2018-09-29T10:00:00.404Z",
   "generation" : 2
}
```

To limit information returned by the API, use arguments:

```
$ curl "demo.nginx.com/api/3/nginx?fields=version,build" \
  | json_pp
{
   "build" : "nginx-plus-r16",
   "version" : "1.15.2"
}
```

You can request connection statistics from the `/api/{version}/connections` URI:

```
$ curl "demo.nginx.com/api/3/connections" | json_pp
{
   "active" : 3,
   "idle" : 34,
   "dropped" : 0,
   "accepted" : 33614951
}
```

You can collect request statistics from the `/api/{version}/http/requests` URI:

```
$ curl "demo.nginx.com/api/3/http/requests" | json_pp
{
   "total" : 52107833,
   "current" : 2
}
```

You can retrieve statistics about a particular server zone using the `/api/{version}/http/server_zones/{httpServerZoneName}` URI:

```
$ curl "demo.nginx.com/api/3/http/server_zones/hg.nginx.org" \
  | json_pp
{
   "responses" : {
      "1xx" : 0,
      "5xx" : 0,
      "3xx" : 938,
      "4xx" : 341,
      "total" : 25245,
      "2xx" : 23966
   },
   "requests" : 25252,
   "discarded" : 7,
   "received" : 5758103,
   "processing" : 0,
   "sent" : 359428196
}
```

The API can return any bit of data you can see on the dashboard. It has depth and follows a logical pattern. You can find links to resources at the end of this recipe.

Discussion

The NGINX Plus API can return statistics about many parts of the NGINX Plus server. You can gather information about the NGINX Plus server, its processes, connections, and slabs. You can also find information about `http` and `stream` servers running within NGINX, including servers, upstreams, upstream servers, and key-value stores, as well as information and statistics about HTTP cache zones. This provides you or third-party metric aggregators with an in-depth view of how your NGINX Plus server is performing.

Also See

NGINX HTTP API Module Documentation
NGINX API Swagger UI

CHAPTER 14

Debugging and Troubleshooting with Access Logs, Error Logs, and Request Tracing

14.0 Introduction

Logging is the basis of understanding your application. With NGINX you have great control over logging information meaningful to you and your application. NGINX allows you to divide access logs into different files and formats for different contexts and to change the log level of error logging to get a deeper understanding of what's happening. The capability of streaming logs to a centralized server comes innately to NGINX through its Syslog logging capabilities. In this chapter, we discuss access and error logs, streaming over the Syslog protocol, and tracing requests end to end with request identifiers generated by NGINX.

14.1 Configuring Access Logs

Problem

You need to configure access log formats to add embedded variables to your request logs.

Solution

Configure an access log format:

```
http {
    log_format  geoproxy
                '[$time_local] $remote_addr '
                '$realip_remote_addr $remote_user '
                '$request_method $server_protocol '
                '$scheme $server_name $uri $status '
                '$request_time $body_bytes_sent '
                '$geoip_city_country_code3 $geoip_region '
                '"$geoip_city" $http_x_forwarded_for '
                '$upstream_status $upstream_response_time '
                '"$http_referer" "$http_user_agent"';
    ...
}
```

This log format configuration is named `geoproxy` and uses a number of embedded variables to demonstrate the power of NGINX logging. This configuration shows the local time on the server when the request was made, the IP address that opened the connection, and the IP of the client as NGINX understands it per `geoip_proxy` or `realip_header` instructions. `$remote_user` shows the username of the user authenticated by basic authentication, followed by the request method and protocol, as well as the scheme, such as HTTP or HTTPS. The `server name` match is logged as well as the request URI and the return status code. Statistics logged include the processing time in milliseconds and the size of the body sent to the client. Information about the country, region, and city are logged. The HTTP header `X-Forwarded-For` is included to show if the request is being forwarded by another proxy. The `upstream` module enables some embedded variables that we've used that show the status returned from the upstream server and how long the upstream request takes to return. Lastly we've logged some information about where the client was referred from and what browser the client is using. The `log_format` directive is only valid within the HTTP context.

This log configuration renders a log entry that looks like the following:

```
[25/Nov/2016:16:20:42 +0000] 10.0.1.16 192.168.0.122 Derek
GET HTTP/1.1 http www.example.com / 200 0.001 370 USA MI
"Ann Arbor" - 200 0.001 "-" "curl/7.47.0"
```

To use this log format, use the `access_log` directive, providing a logfile path and the format name `geoproxy` as parameters:

```
server {
    access_log  /var/log/nginx/access.log  geoproxy;
```

```
    ...
}
```

The `access_log` directive takes a logfile path and the format name as parameters. This directive is valid in many contexts and in each context can have a different log path and or log format.

Discussion

The log module in NGINX allows you to configure log formats for many different scenarios to log to numerous logfiles as you see fit. You may find it useful to configure a different log format for each context, where you use different modules and employ those modules' embedded variables, or a single, catchall format that provides all the information you could ever want. It's also possible to format the log in JSON or XML. These logs will aid you in understanding your traffic patterns, client usage, who your clients are, and where they're coming from. Access logs can also aid you in finding lag in responses and issues with upstream servers or particular URIs. Access logs can be used to parse and play back traffic patterns in test environments to mimic real user interaction. There's limitless possibility for logs when troubleshooting, debugging, or analyzing your application or market.

14.2 Configuring Error Logs

Problem

You need to configure error logging to better understand issues with your NGINX server.

Solution

Use the `error_log` directive to define the log path and the log level:

```
error_log /var/log/nginx/error.log warn;
```

The `error_log` directive requires a path; however, the log level is optional. This directive is valid in every context except for `if` statements. The log levels available are `debug`, `info`, `notice`, `warn`, `error`, `crit`, `alert`, or `emerg`. The order in which these log levels were introduced is also the order of severity from least to most. The `debug` log level is only available if NGINX is configured with the `--with-debug` flag.

Discussion

The error log is the first place to look when configuration files are not working correctly. The log is also a great place to find errors produced by application servers like FastCGI. You can use the error log to debug connections down to the worker, memory allocation, client IP, and server. The error log cannot be formatted. However, it follows a specific format of date, followed by the level, then the message.

14.3 Forwarding to Syslog

Problem

You need to forward your logs to a Syslog listener to aggregate logs to a centralized service.

Solution

Use the `access_log` and `error_log` directives to send your logs to a Syslog listener:

```
error_log syslog:server=10.0.1.42 debug;

access_log syslog:server=10.0.1.42,tag=nginx,severity=info
  geoproxy;
```

The `syslog` parameter for the `error_log` and `access_log` directives is followed by a colon and a number of options. These options include the required server flag that denotes the IP, DNS name, or Unix socket to connect to, as well as optional flags such as `facility`, `severity`, `tag`, and `nohostname`. The `server` option takes a port number, along with IP addresses or DNS names. However, it defaults to UDP 514. The `facility` option refers to the facility of the log message defined as one of the 23 defined in the RFC standard for Syslog; the default value is `local7`. The `tag` option tags the message with a value. This value defaults to `nginx`. `severity` defaults to `info` and denotes the severity of the message being sent. The `nohostname` flag disables adding the hostname field into the Syslog message header and does not take a value.

Discussion

Syslog is a standard protocol for sending log messages and collecting those logs on a single server or collection of servers. Sending logs to a centralized location helps in debugging when you've got multiple instances of the same service running on multiple hosts. This is called aggregating logs. Aggregating logs allows you to view logs together in one place without having to jump from server to server and mentally stitch together logfiles by timestamp. A common log aggregation stack is ElasticSearch, Logstash, and Kibana, also known as the ELK Stack. NGINX makes streaming these logs to your Syslog listener easy with the `access_log` and `error_log` directives.

14.4 Request Tracing

Problem

You need to correlate NGINX logs with application logs to have an end-to-end understanding of a request.

Solution

Use the `request` identifying variable and pass it to your application to log as well:

```
log_format trace '$remote_addr - $remote_user [$time_local] '
                 '"$request" $status $body_bytes_sent '
                 '"$http_referer" "$http_user_agent" '
                 '"$http_x_forwarded_for" $request_id';
upstream backend {
    server 10.0.0.42;
}
server {
    listen 80;
    add_header X-Request-ID $request_id; # Return to client
    location / {
        proxy_pass http://backend;
        proxy_set_header X-Request-ID $request_id; #Pass to app
        access_log /var/log/nginx/access_trace.log trace;
    }
}
```

In this example configuration, a `log_format` named `trace` is set up, and the variable `$request_id` is used in the log. This `$request_id` variable is also passed to the upstream application by use of the

`proxy_set_header` directive to add the request ID to a header when making the upstream request. The request ID is also passed back to the client through use of the `add_header` directive setting the request ID in a response header.

Discussion

Made available in NGINX Plus R10 and NGINX version 1.11.0, the `$request_id` provides a randomly generated string of 32 hexadecimal characters that can be used to uniquely identify requests. By passing this identifier to the client as well as to the application, you can correlate your logs with the requests you make. From the front-end client, you will receive this unique string as a response header and can use it to search your logs for the entries that correspond. You will need to instruct your application to capture and log this header in its application logs to create a true end-to-end relationship between the logs. With this advancement, NGINX makes it possible to trace requests through your application stack.

CHAPTER 15

Performance Tuning

15.0 Introduction

Tuning NGINX will make an artist of you. Performance tuning of any type of server or application is always dependent on a number of variable items, such as, but not limited to, the environment, use case, requirements, and physical components involved. It's common to practice bottleneck-driven tuning, meaning to test until you've hit a bottleneck, determine the bottleneck, tune for limitations, and repeat until you've reached your desired performance requirements. In this chapter, we suggest taking measurements when performance tuning by testing with automated tools and measuring results. This chapter also covers connection tuning for keeping connections open to clients as well as upstream servers, and serving more connections by tuning the operating system.

15.1 Automating Tests with Load Drivers

Problem

You need to automate your tests with a load driver to gain consistency and repeatability in your testing.

Solution

Use an HTTP load-testing tool such as Apache JMeter, Locust, Gatling, or whatever your team has standardized on. Create a configuration for your load-testing tool that runs a comprehensive test

on your web application. Run your test against your service. Review the metrics collected from the run to establish a baseline. Slowly ramp up the emulated user concurrency to mimic typical production usage and identify points of improvement. Tune NGINX and repeat this process until you achieve your desired results.

Discussion

Using an automated testing tool to define your test gives you a consistent test to build metrics off of when tuning NGINX. You must be able to repeat your test and measure performance gains or losses to conduct science. Running a test before making any tweaks to the NGINX configuration to establish a baseline gives you a basis to work from so that you can measure if your configuration change has improved performance or not. Measuring for each change made will help you identify where your performance enhancements come from.

15.2 Keeping Connections Open to Clients

Problem

You need to increase the number of requests allowed to be made over a single connection from clients and the amount of time idle connections are allowed to persist.

Solution

Use the `keepalive_requests` and `keepalive_timeout` directives to alter the number of requests that can be made over a single connection and the time idle connections can stay open:

```
http {
    keepalive_requests 320;
    keepalive_timeout 300s;
    ...
}
```

The `keepalive_requests` directive defaults to 100, and the `keepalive_timeout` directive defaults to 75 seconds.

Discussion

Typically the default number of requests over a single connection will fulfill client needs because browsers these days are allowed to open multiple connections to a single server per fully qualified domain name. The number of parallel open connections to a domain is still limited typically to a number less than 10, so in this regard, many requests over a single connection will happen. A trick commonly employed by content delivery networks is to create multiple domain names pointed to the content server and alternate which domain name is used within the code to enable the browser to open more connections. You might find these connection optimizations helpful if your frontend application continually polls your backend application for updates, as an open connection that allows a larger number of requests and stays open longer will limit the number of connections that need to be made.

15.3 Keeping Connections Open Upstream

Problem

You need to keep connections open to upstream servers for reuse to enhance your performance.

Solution

Use the `keepalive` directive in the upstream context to keep connections open to upstream servers for reuse:

```
proxy_http_version 1.1;
proxy_set_header Connection "";

upstream backend {
  server 10.0.0.42;
  server 10.0.2.56;

  keepalive 32;
}
```

The `keepalive` directive in the `upstream` context activates a cache of connections that stay open for each NGINX worker. The directive denotes the maximum number of idle connections to keep open per worker. The proxy modules directives used above the upstream block are necessary for the `keepalive` directive to function properly

for upstream server connections. The `proxy_http_version` directive instructs the proxy module to use HTTP version 1.1, which allows for multiple requests to be made over a single connection while it's open. The `proxy_set_header` directive instructs the proxy module to strip the default header of `close`, allowing the connection to stay open.

Discussion

You want to keep connections open to upstream servers to save the amount of time it takes to initiate the connection, allowing the worker process to instead move directly to making a request over an idle connection. It's important to note that the number of open connections can exceed the number of connections specified in the `keep alive` directive as open connections and idle connections are not the same. The number of `keepalive` connections should be kept small enough to allow for other incoming connections to your upstream server. This small NGINX tuning trick can save some cycles and enhance your performance.

15.4 Buffering Responses

Problem

You need to buffer responses between upstream servers and clients in memory to avoid writing responses to temporary files.

Solution

Tune proxy buffer settings to allow NGINX the memory to buffer response bodies:

```
server {
    proxy_buffering on;
    proxy_buffer_size 8k;
    proxy_buffers 8 32k;
    proxy_busy_buffer_size 64k;
    ...
}
```

The `proxy_buffering` directive is either `on` or `off`; by default it's `on`. The `proxy_buffer_size` denotes the size of a buffer used for reading the first part of the response from the proxied server and defaults to either `4k` or `8k`, depending on the platform. The

`proxy_buffers` directive takes two parameters: the number of buffers and the size of the buffers. By default, the `proxy_buffers` directive is set to a number of `8` buffers of size either `4k` or `8k`, depending on the platform. The `proxy_busy_buffer_size` directive limits the size of buffers that can be busy, sending a response to the client while the response is not fully read. The busy buffer size defaults to double the size of a proxy buffer or the buffer size.

Discussion

Proxy buffers can greatly enhance your proxy performance, depending on the typical size of your response bodies. Tuning these settings can have adverse effects and should be done by observing the average body size returned, and thoroughly and repeatedly testing. Extremely large buffers set when they're not necessary can eat up the memory of your NGINX box. You can set these settings for specific locations that are known to return large response bodies for optimal performance.

15.5 Buffering Access Logs

Problem

You need to buffer logs to reduce the opportunity of blocks to the NGINX worker process when the system is under load.

Solution

Set the buffer size and flush time of your access logs:

```
http {
    access_log /var/log/nginx/access.log main buffer=32k
        flush=1m;
}
```

The `buffer` parameter of the `access_log` directive denotes the size of a memory buffer that can be filled with log data before being written to disk. The `flush` parameter of the `access_log` directive sets the longest amount of time a log can remain in a buffer before being written to disk.

Discussion

Buffering log data into memory may be a small step toward optimization. However, for heavily requested sites and applications, this can make a meaningful adjustment to the usage of the disk and CPU. When using the `buffer` parameter to the `access_log` directive, logs will be written out to disk if the next log entry does not fit into the buffer. If using the `flush` parameter in conjunction with the `buffer` parameter, logs will be written to disk when the data in the buffer is older than the time specified. When buffering logs in this way, when tailing the log, you may see delays up to the amount of time specified by the `flush` parameter.

15.6 OS Tuning

Problem

You need to tune your operating system to accept more connections to handle spike loads or highly trafficked sites.

Solution

Check the kernel setting for *net.core.somaxconn*, which is the maximum number of connections that can be queued by the kernel for NGINX to process. If you set this number over 512, you'll need to set the `backlog` parameter of the `listen` directive in your NGINX configuration to match. A sign that you should look into this kernel setting is if your kernel log explicitly says to do so. NGINX handles connections very quickly, and for most use cases, you will not need to alter this setting.

Raising the number of open file descriptors is a more common need. In Linux, a file handle is opened for every connection; and therefore NGINX may open two if you're using it as a proxy or load balancer because of the open connection upstream. To serve a large number of connections, you may need to increase the file descriptor limit system-wide with the kernel option `sys.fs.file_max`, or for the system user NGINX is running as in the */etc/security/limits.conf* file. When doing so you'll also want to bump the number of `worker_connections` and `worker_rlimit_nofile`. Both of these configurations are directives in the NGINX configuration.

Enable more ephemeral ports. When NGINX acts as a reverse proxy or load balancer, every connection upstream opens a temporary port for return traffic. Depending on your system configuration, the server may not have the maximum number of ephemeral ports open. To check, review the setting for the kernel setting `net.ipv4.ip_local_port_range`. The setting is a lower- and upper-bound range of ports. It's typically OK to set this kernel setting from 1024 to 65535. 1024 is where the registered TCP ports stop, and 65535 is where dynamic or ephemeral ports stop. Keep in mind that your lower bound should be higher than the highest open listening service port.

Discussion

Tuning the operating system is one of the first places you look when you start tuning for a high number of connections. There are many optimizations you can make to your kernel for your particular use case. However, kernel tuning should not be done on a whim, and changes should be measured for their performance to ensure the changes are helping. As stated before, you'll know when it's time to start tuning your kernel from messages logged in the kernel log or when NGINX explicitly logs a message in its error log.

CHAPTER 16

Practical Ops Tips and Conclusion

16.0 Introduction

This last chapter will cover practical operations tips and is the conclusion to this book. Throughout this book, we've discussed many ideas and concepts pertinent to operations engineers. However, I thought a few more might be helpful to round things out. In this chapter I'll cover making sure your configuration files are clean and concise, as well as debugging configuration files.

16.1 Using Includes for Clean Configs

Problem

You need to clean up bulky configuration files to keep your configurations logically grouped into modular configuration sets.

Solution

Use the `include` directive to reference configuration files, directories, or masks:

```
http {
    include config.d/compression.conf;
    include sites-enabled/*.conf
}
```

The `include` directive takes a single parameter of either a path to a file or a mask that matches many files. This directive is valid in any context.

Discussion

By using `include` statements you can keep your NGINX configuration clean and concise. You'll be able to logically group your configurations to avoid configuration files that go on for hundreds of lines. You can create modular configuration files that can be included in multiple places throughout your configuration to avoid duplication of configurations. Take the example *fastcgi_param* configuration file provided in most package management installs of NGINX. If you manage multiple FastCGI virtual servers on a single NGINX box, you can include this configuration file for any location or context where you require these parameters for FastCGI without having to duplicate this configuration. Another example is SSL configurations. If you're running multiple servers that require similar SSL configurations, you can simply write this configuration once and include it wherever needed. By logically grouping your configurations together, you can rest assured that your configurations are neat and organized. Changing a set of configuration files can be done by editing a single file rather than changing multiple sets of configuration blocks in multiple locations within a massive configuration file. Grouping your configurations into files and using `include` statements is good practice for your sanity and the sanity of your colleagues.

16.2 Debugging Configs

Problem

You're getting unexpected results from your NGINX server.

Solution

Debug your configuration, and remember these tips:

- NGINX processes requests looking for the most specific matched rule. This makes stepping through configurations by hand a bit harder, but it's the most efficient way for NGINX to

work. There's more about how NGINX processes requests in the documentation link in the section "Also See" on page 160.

- You can turn on debug logging. For debug logging you'll need to ensure that your NGINX package is configured with the `--with-debug` flag. Most of the common packages have it; but if you've built your own or are running a minimal package, you may want to at least double-check. Once you've ensured you have debug, you can set the `error_log` directive's log level to `debug`: `error_log /var/log/nginx/error.log debug;`.
- You can enable debugging for particular connections. The `debug_connection` directive is valid inside the `events` context and takes an IP or CIDR range as a parameter. The directive can be declared more than once to add multiple IP addresses or CIDR ranges to be debugged. This may be helpful to debug an issue in production without degrading performance by debugging all connections.
- You can debug for only particular virtual servers. Because the `error_log` directive is valid in the main, HTTP, mail, stream, server, and location contexts, you can set the `debug` log level in only the contexts you need it.
- You can enable core dumps and obtain backtraces from them. Core dumps can be enabled through the operating system or through the NGINX configuration file. You can read more about this from the admin guide in the section "Also See" on page 160.
- You're able to log what's happening in rewrite statements with the `rewrite_log` directive on: `rewrite_log on;`.

Discussion

The NGINX platform is vast, and the configuration enables you to do many amazing things. However, with the power to do amazing things, there's also the power to shoot your own foot. When debugging, make sure you know how to trace your request through your configuration; and if you have problems, add the debug log level to help. The debug log is quite verbose but very helpful in finding out what NGINX is doing with your request and where in your configuration you've gone wrong.

Also See

How NGINX Processes Requests
Debugging Admin Guide
Rewrite Log

16.3 Conclusion

This book has focused on high-performance load balancing, security, and deploying and maintaining NGINX and NGINX Plus servers. The book has demonstrated some of the most powerful features of the NGINX application delivery platform. NGINX Inc. continues to develop amazing features and stay ahead of the curve.

This book has demonstrated many short recipes that enable you to better understand some of the directives and modules that make NGINX the heart of the modern web. The NGINX sever is not just a web server, nor just a reverse proxy, but an entire application delivery platform, fully capable of authentication and coming alive with the environments that it's employed in. May you now know that.

About the Author

Derek DeJonghe has had a lifelong passion for technology. His background and experience in web development, system administration, and networking give him a well-rounded understanding of modern web architecture. Derek leads a team of site reliability engineers and produces self-healing, auto-scaling infrastructure for numerous applications. He specializes in Linux cloud environments. While designing, building, and maintaining highly available applications for clients, he consults for larger organizations as they embark on their journey to the cloud. Derek and his team are on the forefront of a technology tidal wave and are engineering cloud best practices every day. With a proven track record for resilient cloud architecture, Derek helps RightBrain Networks be one of the strongest cloud consulting agencies and managed service providers in partnership with AWS today.