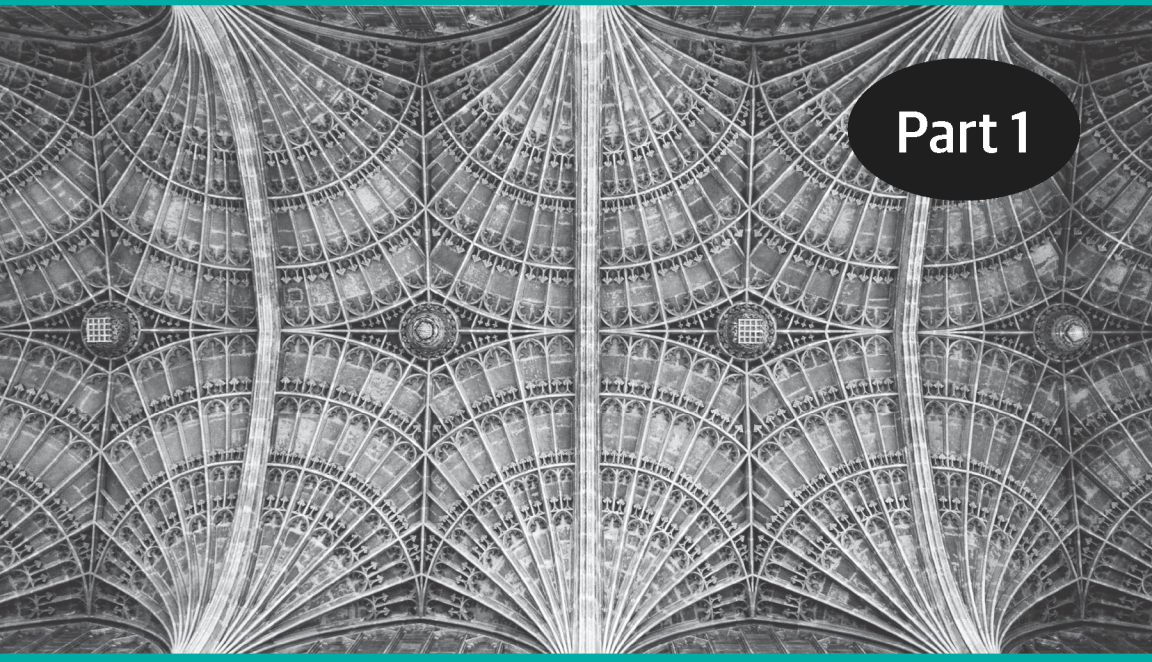
O'REILLY®

Compliments of NGINX

NGINX Cookbook

Advanced Recipes for High Performance Load Balancing



Derek DeJonghe

NGINX Cookbook

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Anderson and Virginia Wilson
Production Editor: Shiny Kalapurakkel
Copyeditor: Amanda Kersey
Proofreader: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Panzer

August 2016: First Edition

Revision History for the First Edition
2016-08-31: First Release

978-1-491-96893-2

[LSI]

Table of Contents

Foreword

NGINX has experienced a spectacular rise in usage since its initial open source release over a decade ago. It's now used by more than half of the world's top 10,000 websites, and more than 165 million websites overall.

How did NGINX come to be used so widely? It's one of the fastest, lightest weight, and most versatile tools available. You can use it as high-performance web server to deliver static content, as a load balancer to scale out applications, as a caching server to build your own CDN, and much, much more.

NGINX Plus, our commercial offering for enterprise applications, builds on the open source NGINX software with extended capabilities including advanced load balancing, application monitoring and active health checks, a fully featured web application firewall (WAF), Single Sign-On (SSO) support, and other critical enterprise features.

The *NGINX Cookbook* shows you how to get the most out of the open source NGINX and NGINX Plus software. This first set of recipes provides a set of easy-to-follow how-tos that cover three of the most important uses of NGINX: load balancing, content caching, and high availability (HA) deployments.

Two more installments of recipes will be available for free in the coming months. We hope you enjoy this first part, and the two upcoming downloads, and that the *NGINX Cookbook* contributes to your success in deploying and scaling your applications with NGINX and NGINX Plus.

— Faisal Memon,
Product Marketer, NGINX, Inc.

Introduction

This is the first of three installments of *NGINX Cookbook*. This book is about NGINX the web server, reverse proxy, load balancer, and HTTP cache. This installment will focus mostly on the load balancing aspect and the advanced features around load balancing, as well as some information around HTTP caching. This book will touch on NGINX Plus, the licensed version of NGINX which provides many advanced features, such as a real-time monitoring dashboard and JSON feed, the ability to add servers to a pool of application servers with an API call, and active health checks with an expected response. The following chapters have been written for an audience that has some understanding of NGINX, modern web architectures such as n-tier or microservice designs, and common web protocols such as TCP, UDP, and HTTP. I wrote this book because I believe in NGINX as the strongest web server, proxy, and load balancer we have. I also believe in NGINX's vision as a company. When I heard Owen Garrett, head of products at NGINX, Inc. explain that the core of the NGINX system would continue to be developed and open source, I knew NGINX, Inc. was good for all of us, leading the World Wide Web with one of the most powerful software technologies to serve a vast number of use cases.

Throughout this report, 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. Most readers in this audience will be users and advocates for the free and open source solution; this report's focus is on just that, free and open source NGINX at its core. However, this first installment provides an opportunity to view some of the advanced features available in the paid solution, NGINX Plus.

CHAPTER 1

High-Performance Load Balancing

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 load increases, another copy of the system can be brought online. The architecture technique is called *horizontal scaling*. Software-based infrastructure is increasing in popularity because of its flexibility, opening up a vast world of possibility. Whether the use case is as small as a set of two for high availability or as large as thousands world wide, 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, the last of which is discussed in Chapter 9.

This chapter discusses load-balancing configurations for HTTP and TCP in NGINX. In this chapter, you will learn about the NGINX load-balancing algorithms, such as round robin, least connection, least time, IP hash, and generic hash. They will aid you in distributing load in ways more useful to your application. When balancing load, you also want to control the amount of load being served to the application server, which is covered in "Connection Limiting" on page 6.

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, either a list 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 reso-

lution control, and the ability to slowly ramp up connections to a server after it starts.

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.

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 interface 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 FQDN; as well as server weight, max num-

ber 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 through out this installment. Features available in NGINX Plus, such as connection limiting, can be found later in this chapter. Health checks for all load balancing will be covered in Chapter 2. Dynamic reconfiguration for upstream pools, a feature available in NGINX Plus, is covered in Chapter 8.

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, or IP hash:

```
upstream backend {
    least_conn;
    server backend.example.com;
    server backend1.example.com;
}
```

This sets the load-balancing algorithm for the backend upstream pool to be least connections. All load-balancing algorithms, with the exception of generic hash, will be standalone directives like the preceding example. Generic hash takes a single parameter, which can be a concatenation of variables, to build the hash from.

Discussion

Not all requests or packets carry an equal weight. Given this, round robin, or even the weighted round robin used in examples prior, will not fit the need of all applications or traffic flow. NGINX provides a number of load-balancing algorithms that can be used to fit particular use cases. These load-balancing algorithms or methods can not only be chosen but also configured. The following load-balancing methods are available for upstream HTTP, TCP, and UDP pools:

Round robin
: The default load-balancing method which distributes requests in order of the list of servers in the upstream pool. Weight can be taken into consideration for a weighted round robin, which could be used 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. Round robin is the default load-balancing algorithm and is used if no other algorithm is specified.

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

Least time
: Available only in NGINX Plus, 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 out there and fits the need of highly performant web applications. 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 amongst 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 determining what upstream server most likely will have the data cached. Redistribution is to be noted, when a server is added or removed from the pool, the hashed requests will be redistributed. NGINX Plus has an optional parameter, `consistent`, to minimize the effect of redistribution. The directive name is `hash`.

IP hash
: Only supported for HTTP, is the last of the bunch but not the least. 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 get proxied to the same upstream server as long as that server is available, 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`.

Connection Limiting

Problem

You have too much load overwhelming your upstream servers.

Solution

Use NGINX Plus's `max_conns` parameter to limit connections to upstream servers:

```
upstream backend {
    zone backends 64k;
    queue 750 timeout=30s;

    server webserver1.example.com max_conns=250;
    server webserver2.example.com max_conns=150;
}
```

The connection-limiting feature is currently only available in NGINX Plus. This NGINX Plus configuration sets an integer on each upstream server that specifies the max number of connections to be handled at any given time. If the max number of connections has been reached on each server, the request can be placed into the queue for further processing, provided the optional `queue` directive is specified. The optional `queue` directive sets the maximum number of requests that can be simultaneously in the queue. A *shared memory zone* is created by use of the `zone` directive. The shared memory zone allows NGINX Plus worker processes to share information about how many connections are handled by each server and how many requests are queued.

Discussion

In dealing with distribution of load, one concern is overload. Overloading a server will cause it to queue connections in a listen queue. If the load balancer has no regard for the upstream server, it can load the server's listen queue beyond repair. The ideal approach is for the load balancer to be aware of the connection limitations of the server and queue the connections itself so that it can send the connection to the next available server with understanding of load as a whole. Depending on the upstream server to process its own queue will lead to poor user experience as connections start to timeout. NGINX Plus provides a solution by allowing connections to queue at the load balancer and by making informed decisions on where it sends the next request or session.

The `max_conns` parameter on the `server` directive within the `upstream` block provides NGINX Plus with a limit of how many connections each upstream server can handle. This parameter is configurable in order to match the capacity of a given server. When the number of current connections to a server meets the value of the `max_conns` parameter specified, NGINX Plus will stop sending new requests or sessions to that server until those connections are released.

Optionally, in NGINX Plus, if all upstream servers are at their `max_conns` limit, NGINX Plus can start to queue new connections until resources are freed to handle those connections. Specifying a queue is optional. When queuing, we must take into consideration a reasonable queue length. Much like in everyday life, users and applications would much rather be asked to come back after a short period of time than wait in a long line and still not be served. The `queue` directive in an `upstream` block specifies the max length of the queue. The timeout parameter of the `queue` directive specifies how long any given request should wait in queue before giving up, which defaults to 60 seconds.

CHAPTER 2

Intelligent Session Persistence

Introduction

While HTTP may be a stateless protocol, if the context it's to convey were stateless, the Internet would be a much less interesting place. 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 may be stored locally for a number of reasons, for example, in applications where 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 are continued to be delivered to the same server. Another portion of the problem 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.

NGINX Plus's `sticky` directive alleviates difficulties of server affinity at the traffic controller, allowing the application to focus on its core. NGINX tracks session persistence in three ways: by creating and tracking its own cookie, detecting when applications prescribe cookies, or routing based on runtime variables.

Sticky Cookie

Problem

You need to bind a downstream client to a upstream server.

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. The cookie in this example is named `affinity`, is set for example.com, persists an hour, cannot be consumed client-side, can only be sent over HTTPS, and is valid for all paths.

Discussion

Using the `cookie` parameter on the `sticky` directive will create a cookie on first request containing 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 expire time, domain, path, and whether the cookie can be consumed client-side or if it can be passed over unsecure protocols.

Sticky Learn

Problem

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

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;
}
```

The 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 megabytes 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 told to look for cookies in the response from the upstream server with specification of the `create` parameter, and searches for prior registered server affinity by the `lookup` parameter. The value of these parameters are variables exposed by the `HTTP` module.

Sticky Routing

Problem

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

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;
}
```

The 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 then 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 not-empty 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`. While 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 may 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 a 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 not-empty variable is used to route to a server. Map blocks can be used to selectively parse variables and save them as another variable 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.

Connection Draining

Problem

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

Solution

Use the `drain` parameter through the NGINX Plus API, described in more detail in Chapter 8, to instruct NGINX to stop sending new connections that are not already tracked:

```
$ curl 'http://localhost/upstream_conf\
?upstream=backend&id=1&drain=1'
```

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 that server expire natively before removing the server from the upstream pool. Draining can be configured for a particular server by adding the

`drain` parameter to the `server` directive. When the `drain` parameter is set, NGINX Plus will stop sending new sessions to this server but will allow current sessions to continue being served for the length of their session.

CHAPTER 3

Application-Aware Health Checks

Introduction

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; as well as active, available only in NGINX Plus. Active health checks on a regular interval will make a connection or request to the upstream server and have the ability to 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 may want to use passive health checks to reduce the load of your upstream servers, and you may want to use active health checks to determine failure of a upstream server before a client is served a failure.

What to Check

Problem

You need to check your application for health but don't know what to check.

Solution

Use a simple but direct indication of the application health. For example, a handler that simply returns a HTTP 200 response tells the load balancer that the application process is running.

Discussion

It's important to check the core of the service you're load balancing for. A single comprehensive health check that ensures all of the systems are available can be problematic. Health checks should check that the application directly behind the load balancer is available over the network and that the application itself is running. With application-aware health checks, you want to pick a endpoint that simply ensures that the processes on that machine are running. It may be tempting to make sure that the database connection strings are correct or that the application can contact its resources. However, this can cause a cascading effect if any particular service fails.

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.

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.

HTTP Health Checks

Problem

You need to actively check your upstream HTTP servers for health.

Solution

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 a HTTP request to the URI '/' every two seconds. The upstream servers must pass five consecutive health checks to be considered healthy and will be considered unhealthy if they fail just a single request. The response from the upstream server must match the defined match block, which defines the status code as 200, the header `Content-Type` value to `'text/html'`, and the string `"Welcome to nginx!"` in the response body.

Discussion

HTTP health checks in NGINX Plus 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. Active health check monitoring can be configured for how often upstream servers are checked, the URI to check, how many times it 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 has three directives: status, header, and body. All three of these directives have comparison flags as well.

CHAPTER 4

High-Availability Deployment Modes

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 will detail techniques on how to run multiple NGINX servers to ensure high availability in your load-balancing tier.

NGINX HA Mode

Problem

You need a highly available load-balancing solution.

Solution

Use NGINX Plus's HA mode with keepalived by installing the nginx-ha-keepalived package from the NGINX Plus repository.

Discussion

The NGINX Plus repository includes a package called nginx-ha-keepalived. This package, based on keepalived, 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.

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 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 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.

Load Balancing on EC2

Problem

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

Solution

Put NGINX behind an elastic load balancer by configuring an auto-scaling group of NGINX servers and linking the auto-scaling group to the elastic load balancer. Alternatively, you can place NGINX servers into the elastic load balancer manually through the Amazon Web Services console, command-line interface, or API.

Discussion

The HA solution from NGINX Plus based on keepalived will not work on Amazon Web Services because it does not support the floating virtual IP address, as EC2 IP addresses work in a different way. This does not mean that NGINX can't be HA in the AWS cloud; in fact, it's the opposite. The AWS elastic load balancer 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 provide a DNS CNAME endpoint. A common solution for HA NGINX on AWS is to put an NGINX layer behind the ELB. NGINX servers can be automatically added to and removed from the ELB pool as needed. The ELB is not a replacement for NGINX; there are many things NGINX offers that the ELB does not, such as multiple load-balancing methods, context switching, and UDP load balancing. In the event that the ELB will 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. Amazon also has a white paper about high-availability NGINX Plus, with use of Corosync and Pacemaker, that will cluster the NGINX servers and use an elastic IP to float between boxes for automatic failover.[1]

1 Amazon also has a white paper about NGINX Plus failover on AWS: *http://bit.ly/2aWAqW8*.

CHAPTER 5

Massively Scalable Content Caching

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 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 a 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.

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;
```

The cache definition example creates a directory for cached responses on the file system 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.

Discussion

To configure caching in NGINX, it's necessary to a 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 of 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 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 use of the `max_size` parameter. Other parameters are in relation to the cache loading process, which loads the cache keys into the shared memory zone from the files cached on disk.

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 which was generated for a different user.

Discussion

The default proxy_cache_key is `"$scheme$proxy_host $request_uri"`. This default will fit most use cases. 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. The `proxy_cache_key` directive configures the string to be hashed for the cache key. The `proxy_cache_key` can be set in the context of HTTP, server, and

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*.

location blocks, providing flexible control on how requests are cached.

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:

```
location ~ /admin/ {
    set $cachebypass 1;
}
proxy_cache_bypass $cachebypass;
```

The configuration tells NGINX to bypass the cache if the URI starts with `/admin/`.

Discussion

There are many scenarios that demand that the request is not cached. For this, NGINX exposes a `proxy_cache_bypass` directive that when the value is nonempty or nonzero, the request will be sent to an upstream server rather than be pulled from cache. Interesting techniques and solutions for cache bypass are derived from the need of the client and application. These can be as simple as a request variable or as intricate as a number of map blocks.

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. One way to do this would be to map a request header with an arbitrary name like `cache_bypass`, and map that to a variable provided to the `proxy_cache_bypass` directive.

Cache Performance

Problem

You need to increase performance of your cache.

Solution

Use client-side cache control headers and other caching directives like `proxy_store`:

```
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 to do with many variables, 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. NGINX also has the cache notion called a *store*. This concept is designed for serving large files that do not change. A store will help serve large files faster as the files do not expire, which is an extremely ideal scenario when designing your own CDN with NGINX as edge servers.

Purging

Problem

You need to invalidate an object from the cache.

Solution

Use NGINX Plus's purge feature, 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;
    }
}
```

Discussion

Common concept for 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 this 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 a 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.

CHAPTER 6

Sophisticated Media Streaming

Introduction

This section 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 feature offers bitrate limiting, enabling your content to be delivered in the most efficient manner while reserving the servers' resources to reach the most users.

Serving MP4 and FLV

Problem

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

Solution

Designate a 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 of MP4 format type and can be streamed with progressive download support. The second location block instructs NGINX that any files ending in *.flv* are of Flash Video 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.

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 the number of HLS buffer is set to 10 with a size of 10 megabytes. The initial MP4 buffer size is set to one megabyte with a maximum of five megabytes.

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 a 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.

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 server 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 server previously fragmented media to end users. The configuration of such is as simple as using the f4f handler inside of a HTTP location block. The `f4f_buffer_size` directive configures the buffer size for the index file of this type of media.

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 a previous section designates this location block to stream MP4 media formats. The rate-limiting directives such as `mp4_limit_rate_after` tells 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 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 7

Advanced Activity Monitoring

Introduction

To ensure 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 will detail the power and possibilities of the NGINX Plus dashboard and JSON feed.

NGINX Traffic Monitoring

Problem

You require in-depth metrics about the traffic flowing through your system.

Solution

Utilize NGINX Plus's real-time activity monitoring dashboard:

```
server {
    listen 8080;
    root /usr/share/nginx/html;

    # Redirect requests for / to /status.html
    location = / {
        return 301 /status.html;
    }

    location = /status.html { }

    # Everything beginning with /status
    # (except for /status.html) is
    # processed by the status handler
    location /status {
        status;
    }
}
```

The NGINX Plus configuration serves the NGINX Plus status monitoring dashboard. This configuration sets up a HTTP server to listen on port 8080, serve content out of the */usr/share/nginx/html* directory, and redirect / requests to `/status.html`. All other `/status` requests will be served by the `/status` location which serves the NGINX Plus Status API.

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 7-1.

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 in if it's 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 stats 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.

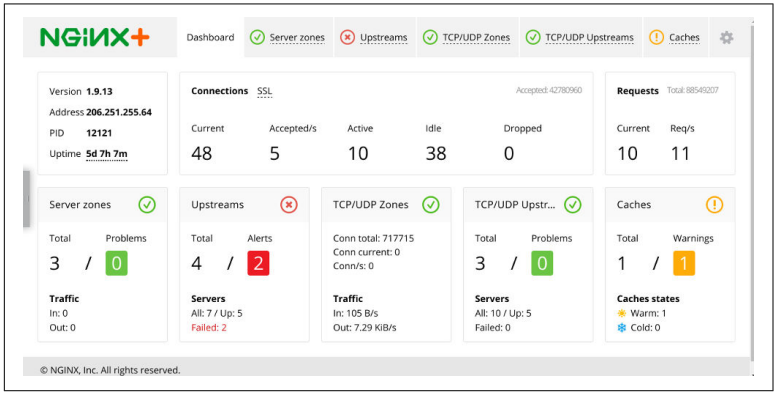


Figure 7-1. This is the NGINX Plus Status Dashboard

The JSON Feed

Problem

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

Solution

Utilize the JSON feed provided by NGINX Plus's status API:

```
$ curl "demo.nginx.com/status/upstreams\
  /demo-backend/peers/0/responses"
{
  "1xx":0,
  "2xx":199237,
  "3xx":7404,
  "4xx":104415,
  "5xx":19574,
  "total":330630
}
```

The *curl* call requests a JSON feed from the NGINX Plus Status API for information about an upstream HTTP server pool, and in particular about the first server in the pool's responses.

Discussion

The NGINX Plus status API is vast, and requesting just the status will return a JSON object with all the information that can be found on the status dashboard in whole. The JSON feed API allows you to drill down to particular information you may want to monitor or use in custom logic to make application or infrastructure decisions. The API is intuitive and RESTful, and you're able to make requests for objects within the overall status JSON feed to limit the data returned. This JSON feed enables you to feed the monitoring data into any other number of systems you may be utilizing for monitoring, such as Graphite, Datadog, and Splunk.

CHAPTER 8

DevOps on the Fly Reconfiguration

Introduction

The term DevOps has being tossed and spun around more than your favorite pizza crust. To the people actually doing the work, the term has nearly lost meaning; the origin of this term comes from a culture of developers and operations folk working together in an Agile workflow to enhance quality and productivity and share responsibility. If you ask a recruiter, it's a job title; ask someone in marketing, it's a hit-generating Swiss army knife. In this context, we mean DevOps to be developing software and tools to solve operational tasks in the ever-evolving dynamic technology landscape. In this chapter, we'll discuss the NGINX Plus API that allows you to dynamically reconfigure the NGINX Plus load balancer, as well as other tools and patterns to allow your load balancer to evolve with the rest of your environment, such as the seamless reload and NGINX Plus's ability to utilize DNS SRV records.

The NGINX API

Problem

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

Solution

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

```
location /upstream_conf {
    upstream_conf;
    allow 10.0.0.0/8; # permit access from private network
    deny all;         # deny access from everywhere else
}

...
upstream backend {
    zone backend 64k;
    state /var/lib/nginx/state/backend.state;
    ...
}
```

The NGINX Plus configuration enables the upstream configuration API and only allows access from a private network. The configuration of the `upstream` block defines a shared memory zone named backend of 64 kilobytes. The `state` directive tells NGINX to persist these changes through a restart by saving them to the file system.

Utilize the API to add servers when they come online:

```
$ curl 'http://nginx.local/upstream_conf?\
  add=&upstream=backend&server=10.0.0.42:8080'
```

The *curl* call demonstrated makes a request to NGINX Plus and requests a new server be added to the backend upstream configuration.

Utilize the NGINX Plus API to list the servers in the upstream pool:

```
$ curl 'http://nginx.local/upstream_conf?upstream=backend'
server 10.0.0.42:8080; # id=0
```

The *curl* call demonstrated makes a request to NGINX Plus to list all of the servers in the upstream pool named backend. Currently we only have the one server that we added in the previous *curl* call to the API. The list request will show the IP address, port, and ID of each server in the pool.

Use the NGINX Plus API to drain connections from an upstream server, preparing it for a graceful removal from the upstream pool. Details about connection draining can be found in Chapter 2, "Connection Draining" on page 13:

```
$ curl 'http://nginx.local/upstream_conf?\
  upstream=backend&id=0&drain=1'
server 10.0.0.42:8080; # id=0 draining
```

In this *curl*, we specify arguments for the upstream pool, backend, the ID of the server we wish to drain, 0, and set the `drain` argument to equal 1. 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 begin to drain, you can use the NGINX Plus JSON feed that was detailed in Chapter 7, "The JSON Feed" on page 39.

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

```
$ curl 'http://nginx.local/upstream_conf?\
  upstream=backend&id=0&remove=1'
```

The *curl* command passes arguments to the NGINX Plus API to remove server 0 from the upstream pool named backend. This API call will return all of the servers and their IDs that are still left in the pool. As 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

This upstream 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 begin to start sending it load. When a server needs to be removed, the server can request NGINX Plus to drain its connections, then remove itself from the upstream pool before it's shut down. This enables the infrastructure to, through some automation, scale in and out without human intervention.

Seamless Reload

Problem

You need to reload you configuration without dropping packets.

Solution

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

```
service nginx reload
```

The command-line example reloads the NGINX system using the NGINX init script generally located in the */etc/init.d/* directory.

Discussion

Reloading the NGINX configuration without stopping the server provides the ability to change configuration 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 the needs of the environment.

SRV Records

Problem

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

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;
    }
}
```

The configuration instructs NGINX 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 a 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.

CHAPTER 9

UDP Load Balancing

Introduction

User Datagram Protocol (UDP) is used in many contexts, such as DNS, NTP, and Voice over IP. NGINX can load balance over upstream servers with all the load-balancing algorithms provided to the other protocols. In this chapter, we'll cover the UDP load balancing in NGINX.

Stream Context

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 dns {
        server ns1.example.com:53  weight=2;
        server ns2.example.com:53;
    }

    server {
        listen 53 udp;
        proxy_pass dns;
    }
}
```

This section of configuration balances load between two upstream DNS servers using the UDP protocol. Specifying UDP load balancing is as simple as using the `udp` parameter on the `listen` directive.

Discussion

One might ask, "Why do you need a load balancer when you can have multiple hosts in a DNS A or SRV record?" The answer is that not only are there alternative balancing algorithms we can balance with, 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 Voice over IP. UDP load balancing may be less common to some but just as useful in the world of scale.

UDP load balancing will be found in the `stream` module, just like TCP, and configured 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 may apply where they would not in TCP, such as the `proxy_response` directive that tells NGINX how many expected responses may be sent from the upstream server, by default being unlimited until the `proxy_timeout` limit is reached.

Load-Balancing Algorithms

Problem

You need to distribute load of a UDP service with control over the destination or for best performance.

Solution

Utilize the different load-balancing algorithms, like IP hash or least conn, described in Chapter 1:

```
upstream dns {
    least_conn;
    server ns1.example.com:53;
    server ns2.example.com:53;
}
```

The configuration load balances over two DNS name servers and directs the request to the name server with the least number of current connections.

Discussion

All of the load-balancing algorithms that were described in "Load-Balancing Algorithms" on page 49 are available in UDP load balancing as well. These algorithms, such as least connections, least time, generic hash, or IP hash, are useful tools to provide the best experience to the consumer of the service or application.

Health Checks

Problem

You need to check the health of upstream UDP servers.

Solution

Use NGINX health checks with UDP load balancing to ensure only healthy upstream servers are sent datagrams:

```
upstream dns {
    server ns1.example.com:53 max_fails=3 fail_timeout=3s;
    server ns2.example.com:53 max_fails=3 fail_timeout=3s;
}
```

This configuration passively monitors the upstream health, setting the `max_fails` directive to 3, and `fail_timeout` to 3 seconds.

Discussion

Health checking is important on all types of load balancing not only from a user experience standpoint but also for business continuity. NGINX can actively and passively monitor upstream UDP servers to ensure they're healthy and performing. Passive monitoring watches for failed or timed-out connections as they pass through NGINX. Active health checks attempt to make a connection to the specified port, and can optionally expect a response.

CHAPTER 10

Cloud-Agnostic Architecture

Introduction

One thing many companies request when moving to the cloud is to be cloud agnostic. Being cloud agnostic in their architectures enables them to pick up and move to another cloud or instantiate the application in a location that one cloud provider may have that another does not. Cloud-agnostic architecture also reduces risk of vendor lock-in and enables an insurance fallback for your application. It's very common for disaster-recovery plans to use an entirely separate cloud, as failure can sometimes be systematic and affect a cloud as a whole. For cloud-agnostic architecture, all of your technology choices must be able to be run in all of those environments. In this chapter, we'll talk about why NGINX is the right technology choice when architecting a solution that will fit in any cloud.

The Anywhere Load Balancer

Problem

You need a load-balancer solution that can be deployed in any datacenter, cloud environment, or even local hosts.

Solution

Load balance with NGINX. NGINX is software that can be deployed anywhere. NGINX runs on Unix; and on multiple flavors of Linux such as Cent OS and Debian, BSD variants, Solaris, OS X, Windows, and others. NGINX can be built from source on Unix and Linux derivatives as well as installed through package managers such as yum, aptitude, and zypper. On Windows, it can be installed by downloading a ZIP archive and running the *.exe* file.

Discussion

The fact that NGINX is a software load balancer rather than strictly hardware allows it to be deployed on almost any infrastructure.[1] Cross-cloud environments and hybrid cloud architectures are on the rise, applications are distributed between different clouds for high availability, and vendor-agnostic architecture limits risk of production outages and reduces network latency between the end user and the application. In these scenarios, the application being hosted typically doesn't change and neither should your load-balancing solution. NGINX can be run in all of these environments with all of the power of its configuration.[2]

The Importance of Versatility

Problem

You need versatility in your architecture and the ability to build in an iterative manner.

Solution

Use NGINX as your load balancer or traffic router. NGINX provides versatility on the platform it runs on or its configuration. If you're architecting a solution, and you're not sure where it's going to live or

1 NGINX provides a page to download its software: *http://nginx.org/en/download.html*.

2 Linux packages and repositories can be found at *http://nginx.org/en/linux_packages.html*.

need the flexibility to be able to move it to another provider, NGINX will fit this need. If you're working in an iterative workflow, and new services or configurations are continually changing during the development cycle, NGINX is a prime resource, as its configuration can change; and with a reload of the service, the new configuration is online without concern of stopping the service. An example might be planning to build out a data center, and then for cost and flexibility, switching gears into a cloud environment. Another example might be refactoring an existing monolithic application and slowly decoupling the application into microservices, deploying service by service as the smaller applications become ready for production.

Discussion

Agile workflows have changed how development work is done. The idea of an Agile workflow is an iterative approach where it's OK if requirements or scope change. Infrastructure architecture can also follow an Agile workflow: you may start out aiming to go into a particular cloud provider and then have to switch to another partway through the project, or want to deploy to multiple cloud providers. NGINX being able to run anywhere makes it an extremely versatile tool. The importance of versatility is that with the inevitable onset of cloud, things are always changing. In the ever-evolving landscape of software, NGINX is able to efficiently serve your application needs as it grows with your features and user base.

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.