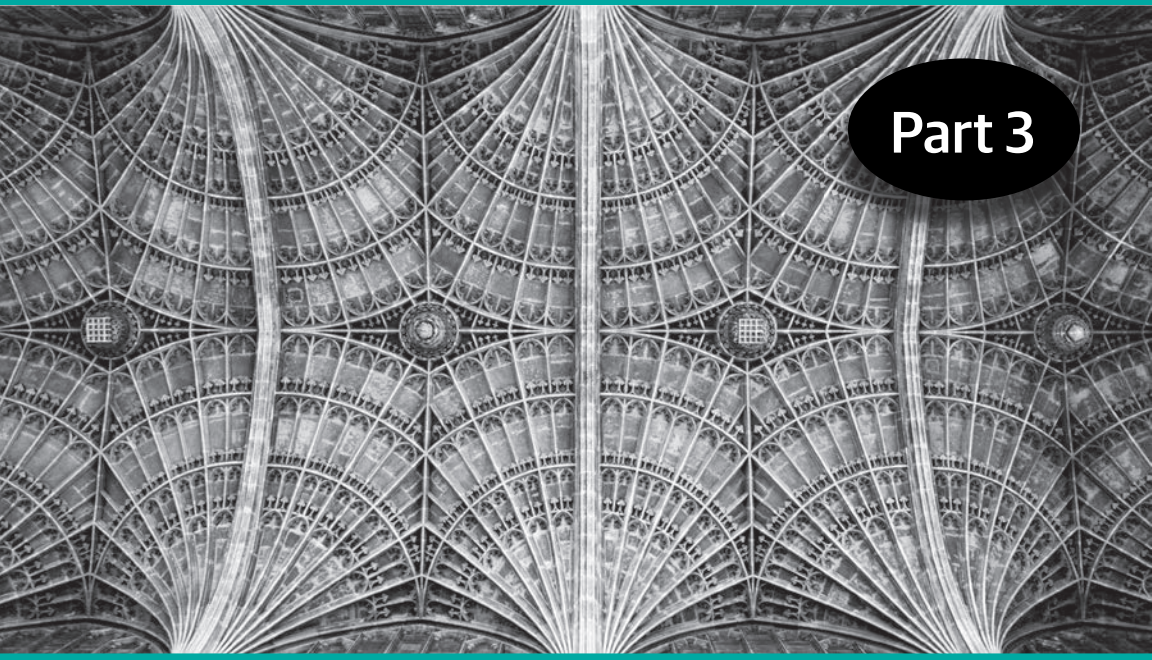
O'REILLY®

Compliments of NGINX

NGINX Cookbook

Advanced Recipes for Operations



Derek DeJonghe

flawless application delivery


Load Balancer


Content Cache


Web Server


Security Controls


Monitoring & Management

FREE TRIAL

LEARN MORE

NGINX+

NGINX Cookbook

Advanced Recipes for Operations

Derek DeJonghe

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

NGINX Cookbook
by Derek DeJonghe

Copyright © 2017 O'Reilly Media Inc. All rights reserved.

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*.

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

March 2017: First Edition

Revision History for the First Edition
2017-03-03: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. NGINX Cookbook, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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

978-1-491-96895-6

[LSI]

Table of Contents

Foreword

I'm honored to be writing the foreword for this third and final part of the NGINX Cookbook series. It's the culmination of a year of collaboration between O'Reilly Media, NGINX, Inc., and author Derek DeJonghe, with the goal of creating a very practical guide to using the open source NGINX software and enterprise-grade NGINX Plus.

We covered basic topics like load balancing and caching in part 1. Part 2 covered the security features in NGINX such as authentication and encryption. This third part focuses on operational issues with NGINX and NGINX Plus, including provisioning, performance tuning, and troubleshooting.

In this part, you'll find practical guidance for provisioning NGINX and NGINX Plus in the big three public clouds: Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure, including how to auto provision within AWS. If you're planning to use Docker, that's covered as well.

Most systems are, by default, configured not for performance but for compatibility. It's then up to you to tune for performance, according to your unique needs. In this ebook, you'll find detailed instructions on tuning NGINX and NGINX Plus for maximum performance, while still maintaining compatibility.

When I'm having trouble with a deployment, the first thing I look at are log files, a great source of debugging information. Both NGINX and NGINX Plus maintain detailed and highly configurable logs to help you troubleshoot issues, and the NGINX Cookbook, Part 3 covers logging with NGINX and NGINX Plus in great detail.

We hope you have enjoyed the NGINX Cookbook series, and that it has helped make the complex world of application development a little easier to navigate.

— Faisal Memon
Product Marketer, NGINX, Inc.

Introduction

This is the third and final installment of the *NGINX Cookbook*. This book is about NGINX the web server, reverse proxy, load balancer, and HTTP cache. This installment will focus on deployment and operations of NGINX and NGINX Plus, the licensed version of the server. Throughout this installment you will learn about deploying NGINX to Amazon Web Services, Microsoft Azure, and Google Cloud Compute, as well as working with NGINX in Docker containers. This installment will dig into using configuration management to provision NGINX servers with tools such as Puppet, Chef, Ansible, and SaltStack. It will also get into automating with NGINX Plus through the NGINX Plus API for on-the-fly reconfiguration and using Consul for service discovery and configuration templating. We'll use an NGINX module to conduct A/B testing and acceptance during deployments. Other topics covered are using NGINX's GeoIP module to discover the geographical origin of our clients, including it in our logs, and using it in our logic. You'll learn how to format access logs and set log levels of error logging for debugging. Through a deep look at performance, this installment will provide you with practical tips for optimizing your NGINX configuration to serve more requests faster. It will help you install, monitor, and maintain the NGINX application delivery platform.

CHAPTER 1

Deploying on AWS

1.0 Introduction

Amazon Web Services (AWS), in many opinions, has led the cloud infrastructure landscape since the arrival of S3 and EC2 in 2006. AWS provides a plethora of *infrastructure-as-a-service* (IaaS) and *platform-as-a-service* (PaaS) solutions. Infrastructure as a service, such as Amazon EC2 or Elastic Cloud Compute, is a service providing virtual machines in as little as a click or API call. This chapter will cover deploying NGINX into an Amazon Web Service environment, as well as some common patterns.

1.1 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 pre-baked Amazon Machine Image. Create an Amazon Machine Image 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 Amazon Machine Images (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 will be 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, SSH's into the virtual machine, runs any provisioning you specify, and burns an image. You can utilize Packer to run the con-

figuration 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. The `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 and 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.

1.2 Routing to NGINX Nodes Without an 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 Amazon Route53 DNS service to route to multiple active NGINX nodes or configure health checks and failover to 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 achieving 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.

1.3 The ELB Sandwich

Problem

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

Solution

Create an *elastic load balancer* (ELB) or two. Create an Auto Scaling group with a launch configuration that provisions an EC2 instance with NGINX installed. The Auto Scaling group has a configuration to link to the elastic load balancer which will automatically register any instance in the Auto Scaling group to the load balancers configured on first boot. Place your upstream applications behind another elastic load balancer and configure NGINX to proxy to that ELB.

Discussion

This common pattern is called the ELB sandwich (see Figure 1-1), putting NGINX in an Auto Scaling group behind an ELB and the application Auto Scaling group behind another ELB. The reason for having ELBs between every layer is because the ELB works so well with Auto Scaling groups; they automatically register new nodes and remove ones being terminated, as well as run health checks and only pass traffic to healthy nodes. The reason behind building a second ELB for NGINX is because it allows services within your application to call out to other services through the NGINX Auto Scaling group without leaving the network and reentering through the public ELB. This puts NGINX in the middle of all network traffic within your application, making it the heart of your application's traffic routing.

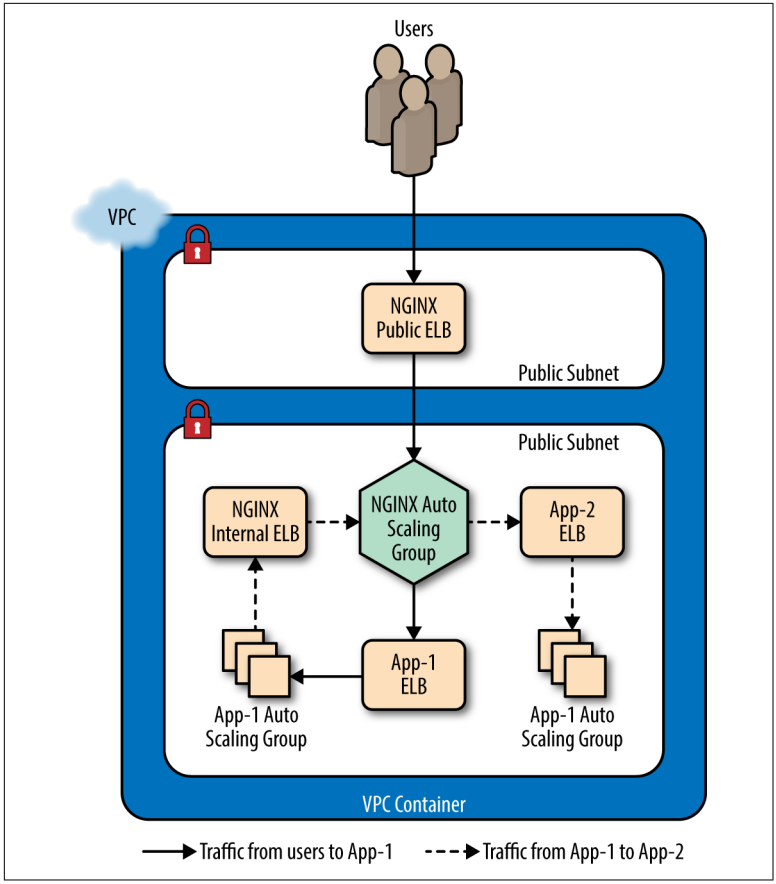


Figure 1-1. This image depicts NGINX in an ELB sandwich pattern with an internal ELB for internal applications to utilize. A user makes a request to App-1, and App-1 makes a request to App-2 through NGINX to fulfill the user's request.

1.4 Deploying from the 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 1-2). Select the Amazon Machine Image (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.

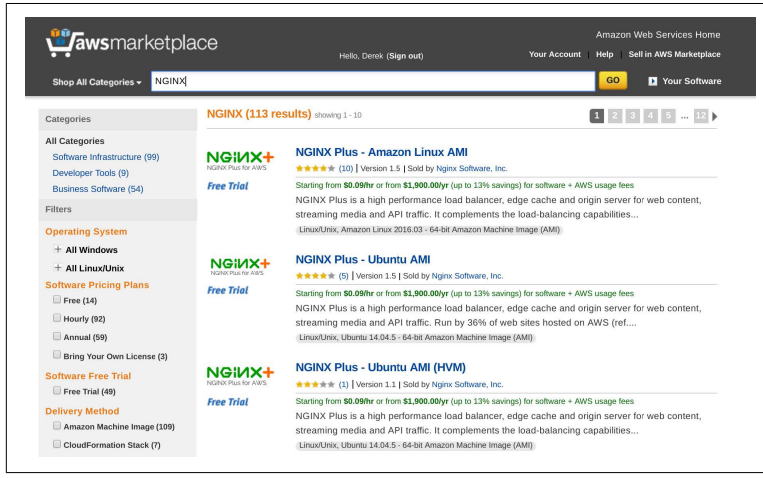


Figure 1-2. This image shows the AWS Marketplace after searching for NGINX.

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.

CHAPTER 2

Deploying on Azure

2.0 Introduction

Azure is a powerful cloud platform offering from Microsoft. Azure enables for cross-platform virtual machine hosting inside of virtual cloud networks. NGINX is an amazing application delivery platform for any OS or application type and works seamlessly in Microsoft Azure. NGINX has provided a pay-per-usage NGINX Plus Marketplace offering, which this chapter will explain how to use, making it easy to get up and running quickly with on-demand licensing in Microsoft Azure.

2.1 Creating an NGINX Virtual Machine Image

Problem

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

Solution

Create a virtual machine 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 virtual machine image. To create a virtual machine image, you must first generalize

the VM. To generalize your virtual machine, 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 virtual machine. 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 virtual machine:

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

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

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

After your virtual machine 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 virtual machine 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 we will need to utilize the ARM tem-

plate created by Azure. Once you have the ID, we 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 of your choosing for the virtual machine name, 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 virtual machine from the custom NGINX or NGINX Plus image you've created. After entering the necessary parameters, Azure will begin to create a new virtual machine 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. Azure creating an ARM template enables you to quickly and reliably deploy this same server time and time again as needed. With the virtual machine image path that can be found in the template, you can use this image to create different sets of infrastructure such as virtual machine 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

2.2 Load Balancing Over NGINX Scale Sets

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 2.3 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 over provisioning resources. In Azure you achieve this with virtual machine scaling sets. 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.

2.3 Deploying Through the Marketplace

Problem

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

Solution

Deploy an NGINX Plus virtual machine 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 decide 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 virtual machine, 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 virtual machine 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 virtual machine 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 virtual machine 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.

CHAPTER 3

Deploying on Google Cloud Compute

3.0 Introduction

Google Cloud Compute is an advanced cloud platform that enables its customers to build diverse, high-performing web applications at will on hardware they provide and manage. Google Cloud Compute offers virtual networking and machines, a tried-and-true platform-as-a-service (PaaS) offering, as well as many other managed service offerings such as Bigtable, BigQuery, and SQL. In this chapter, we will discuss how to deploy NGINX servers to Google Cloud Compute, how to create virtual machine images, and how and why you might want to use NGINX to serve your Google App Engine applications.

3.1 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 virtual machine in Google Compute Engine. Select a name for your virtual machine, zone, machine type, and boot disk.

Configure identity and access management, firewall, and any advanced configuration you'd like. Create the virtual machine.

Once the virtual machine 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 Google Compute Engine startup script, which is an advanced configuration option when creating a virtual machine.

Discussion

Google Compute Engine offers highly configurable virtual machines at a moment's notice. Starting a virtual machine 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.

3.2 Creating a Google Compute Image

Problem

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

Solution

Create a virtual machine as described in the previous section. After installing and configuring NGINX on your virtual machine instance, set the auto-delete state of the boot disk to `false`. To set the auto-delete state of the disk, edit the virtual machine. 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 virtual machine 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 virtual machines 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

3.3 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 would 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 common 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 4

Deploying on Docker

4.0 Introduction

Docker is an open-source project that automates the deployment of Linux applications inside software containers. Docker provides an additional layer of abstraction and automation of operating-system-level virtualization on Linux. Containerized environments have made a huge break into the production world, and I'm excited about it. Docker and other container platforms enable fast, reliable, cross-platform application deployments. In this chapter we'll discuss the NGINX official NGINX Docker image, creating your own Dockerfile to run NGINX, and using environment variables within NGINX, a common Docker practice.

4.1 Running Quickly with the 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, so you'll need to either mount a local configuration directory or create a Dockerfile and `ADD` in your configuration to the image build. We'll mount a volume and get NGINX running in a Docker container locally in two commands:

```
$ docker pull nginx:latest
$ docker run -it -p 80:80 -v $PWD/nginx-conf:/etc/nginx \
                                    nginx:latest
```

The first `docker` command pulls the `nginx:latest` image from Docker Hub. The second `docker` command runs this NGINX image as a Docker container in the foreground, mapping `localhost:80` to port `80` of the NGINX container. It also mounts the local directory *nginx-conf* as a container volume at */etc/nginx*. *nginx-conf* is a local directory that contains the necessary files for NGINX configuration. 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 only two commands! 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.

Also See

Official NGINX Docker image, NGINX
Docker repo on GitHub

4.2 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 logging in 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 choose 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.

4.3 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 these Docker files to build NGINX Plus Docker images. You'll need to download your NGINX Plus repository certificate and keep them in the directory with this Dockerfile named *nginx-repo.crt* and *nginx-repo.key*, respectively. With that, these Dockerfiles 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.

Ubuntu:

```
FROM ubuntu:14.04

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

# Set the debconf frontend to Noninteractive
RUN echo 'debconf debconf/frontend select Noninteractive' \
    | debconf-set-selections

RUN apt-get update && apt-get install -y -q wget \
    apt-transport-https lsb-release ca-certificates

# Download certificate and key from the customer portal
# (https://cs.nginx.com) and copy to the build context
ADD nginx-repo.crt /etc/ssl/nginx/
ADD nginx-repo.key /etc/ssl/nginx/

# Get other files required for installation
RUN wget -q -O - http://nginx.org/keys/nginx_signing.key \
    | apt-key add -
RUN wget -q -O /etc/apt/apt.conf.d/90nginx \
    https://cs.nginx.com/static/files/90nginx
```

```
RUN printf "deb https://plus-pkgs.nginx.com/ubuntu \
 `lsb_release -cs` nginx-plus\n" \
  >/etc/apt/sources.list.d/nginx-plus.list

# Install NGINX Plus
RUN apt-get update && apt-get install -y nginx-plus

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

EXPOSE 80 443

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

CentOS 7:

```
FROM centos:centos7

MAINTAINER NGINX Docker Maintainers "docker-maint@nginx.com"

RUN yum install -y ca-certificates

# Download certificate and key from the customer portal
# (https://cs.nginx.com) and copy to the build context
ADD nginx-repo.crt /etc/ssl/nginx/
ADD nginx-repo.key /etc/ssl/nginx/

# Get other files required for installation
RUN wget -q -O /etc/yum.repos.d/nginx-plus-7.repo \
    https://cs.nginx.com/static/files/nginx-plus-7.repo

# Install NGINX Plus
RUN yum install -y nginx-plus

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

EXPOSE 80 443

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

To build these Dockerfiles into Docker images, 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. These Docker files do not use the Dockerfile property `ADD` to add in configuration; you will need to add in your configuration.

Also See

NGINX blog on Docker images

4.4 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 configuration snippet above 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 Docker file can be used in multiple, diverse environments.

CHAPTER 5

Using Puppet/Chef/ Ansible/SaltStack

5.0 Introduction

Configuration management tools have been an invaluable utility in the age of the cloud. Engineers of large-scale web applications are no longer configuring servers by hand but rather by code, using one of the many configuration management tools available. Configuration management tools enable engineers to write configurations and code one time to produce many servers with the same configuration in a repeatable, testable, and modular fashion. In this chapter we'll discuss 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 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 does have a restart command with the `hasrestart` directive, and we override the `restart` command with an NGINX reload. It 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, it 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 Embedded Ruby (ERB) or Embedded Puppet (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 which 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 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

5.2 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 reloaded 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 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 Embedded Ruby (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 and many public cookbooks, called 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 for NGINX 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

5.3 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 build servers directly to the client or in a masterless model. Ansible bulk SSH's into its servers and runs the configurations. Much like other configuration management tools, there's a large community of public 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

5.4 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*) which will install the `nginx` package and ensure the service is running, enabled at boot, and reloads 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 opposed to restart because the `reload` directive is set to `True`.

Discussion

SaltStack is a powerful configuration management tool which 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

CHAPTER 6

Automation

6.0 Introduction

There are many ways to automate NGINX and NGINX Plus configuration files, such as rerunning your configuration management tools or cron jobs that retemplate configuration files. As dynamic environments increase in popularity and necessity, the need for configuration automation becomes more relevant. In Chapter 5, we made sure that NGINX was reloaded after the configuration file was templated. In this chapter, we'll discuss further on-the-fly reconfiguration of NGINX configuration files with the NGINX Plus API and Consul Template.

6.1 Automating with NGINX Plus

Problem

You need to reconfigure NGINX Plus on the fly to load balance for a dynamic environment.

Solution

Use the NGINX Plus API to reconfigure NGINX Plus upstream pools:

```
$ 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.

Discussion

Covered in great detail in Chapter Eight of the first installment, NGINX Plus offers an API to reconfigure NGINX Plus on the fly. The NGINX Plus API enables adding and removing servers from upstream pools as well as draining connections. You can use this API to automate your NGINX Plus configuration as application servers spawn and release in the environment.

6.2 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 of a Consul Template file that templates an upstream configuration block. This template will loop through nodes in Consul identifying 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"
```

The command demonstrated 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 instructs the daemon to 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 7

A/B Testing with split_clients

7.0 Introduction

NGINX has a module named `split_clients` that allows you to programmatically divide up your users based on a variable key. NGINX splits users by using a lightweight hashing algorithm to hash a given string. Then it mathematically divides them by percentages, mapping predefined values to a variable you can use to change the response of your server. This chapter covers the `split_clients` module.

7.1 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 use of this `split_client` module.

Also See

`split_client` documentation

CHAPTER 8

Locating Users by IP Address Using GeoIP Module

8.0 Introduction

Tracking, analyzing, and utilizing the location of your clients in your application or your metrics can take your understanding of them to the next level. There are many implementations for the location of your clients, and NGINX makes locating them easy through use of the GeoIP module and a couple directives. This module makes it easy to log location, control access, or make decisions based on client locations. It also enables the geography of the client to be looked up initially upon ingestion of the request and passed along to any of the upstream applications so they don't have to do the lookup. This NGINX module is not installed by default and will need to be statically compiled from source, dynamically imported, or included in the NGINX package by installing nginx-full or nginx-extras in Ubuntu, which are both built with this module. On RHEL derivatives such as CentOS, you can install the *nginx-mod-http-geoip* package and dynamically import the module with the `load_module` directive. This chapter will cover importing the GeoIP dynamic module, installing the GeoIP database, the embedded variables available in this module, controlling access, and working with proxies.

8.1 Using the GeoIP Module and Database

Problem

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

Solution

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, we 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

8.2 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 US or Russia.

Discussion

This is a short but simple example of how to only allow access from a couple 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.

8.3 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 the AWS ELB, Google's load balancer, or Azure's load balancer in front of NGINX.

CHAPTER 9

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

9.0 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'll discuss access and error logs, streaming over the Syslog protocol, and tracing requests end to end with request identifiers generated by NGINX.

9.1 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 necessary information you could ever want. It's also possible to structure format to 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 to logs when troubleshooting, debugging, or analyzing your application or market.

9.2 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.

9.3 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.

9.4 Request Tracing

Problem

You need to correlate NGINX logs with application logs to have a 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 10

Performance Tuning

10.0 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 limitation, and repeat until you've reached your desired performance requirements. In this chapter we'll suggest taking measurements when performance tuning by testing with automated tools and measuring results. This chapter will also cover connection tuning for keeping connections open to clients as well as upstream servers, and serving more connections by tuning the operating system.

10.1 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.

10.2 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.

10.3 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 would want to keep connections open to upstream servers to save the amount of time it takes to initiate the connection, and the worker process can 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 `keepalive` 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.

10.4 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.

10.5 Buffering Access Logs

Problem

You need to buffer logs to reduce 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.

10.6 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 systemwide 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 systems 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 11

Practical Ops Tips and Conclusion

11.0 Introduction

This last chapter will cover practical operations tips and is the conclusion to this book. Throughout these three installments, 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.

11.1 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.

11.2 Debugging Configs

Problem

You're getting unexpected results from your NGINX server.

Solution

Debug your configuration, and remember these tips:

1. 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 66.

2. 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;`
3. 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.
4. 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.
5. 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 66.
6. 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

11.3 Conclusion

This book's three installments have focused on high-performance load balancing, security, and deploying and maintaining NGINX and NGINX Plus servers. This book has demonstrated some of the most powerful features of the NGINX application delivery platform. NGINX 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.