10.5 Creating an NGINX Virtual Machine Image on Azure

Problem

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

Solution

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Discussion

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

Also See

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

10.6 Load Balancing Over NGINX Scale Sets on Azure

Problem

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

Solution

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

Discussion

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

10.7 Deploying Through the Azure Marketplace

Problem

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

Solution

Deploy an NGINX Plus VM image through the Azure Marketplace:

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

Discussion

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

10.8 Deploying to Google Compute Engine

Problem

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

Solution

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

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

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

Discussion

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

10.9 Creating a Google Compute Image

Problem

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

Solution

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

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

Discussion

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

Also See

Create, Delete, and Depreciate Private Images

10.10 Creating a Google App Engine Proxy

Problem

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

Solution

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

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

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

Discussion

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

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

CHAPTER 11

Containers/Microservices

11.0 Introduction

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

11.1 DNS SRV Records

Problem

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

Solution

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

```
http {
    resolver 10.0.0.2;

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

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

Discussion

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

11.2 Using the Official NGINX Image

Problem

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

Solution

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

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

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

Discussion

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

Also See

Official NGINX Docker image, NGINX
Docker repo on GitHub

11.3 Creating an NGINX Dockerfile

Problem

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

Solution

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

Dockerfile:

```
FROM centos:7

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

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

EXPOSE 80 443

CMD ["nginx"]
```

The directory structure looks as follows:

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

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

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

Discussion

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

11.4 Building an NGINX Plus Image

Problem

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

Solution

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

```
FROM debian:stretch-slim

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

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

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

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

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

EXPOSE 80
STOPSIGNAL SIGTERM
```

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

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

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

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

Discussion

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

Also See

NGINX blog on Docker images

11.5 Using Environment Variables in NGINX

Problem

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

Solution

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

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

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

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

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

```
FROM centos:7

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

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

EXPOSE 80 443

CMD ["nginx"]
```

Discussion

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

11.6 Kubernetes Ingress Controller

Problem

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

Solution

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

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

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

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

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

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

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

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

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

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

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

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

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

For NGINX Deployment:

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

For NGINX Plus Deployment:

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

For NGINX DaemonSet:

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

For NGINX Plus DaemonSet:

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

Validate that the ingress controller is running:

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

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

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

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

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

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

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

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

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

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

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

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

Then, update the config map:

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

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

Discussion

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

11.7 OpenShift Router

Problem

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

Solution

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

Log in to the OpenShift Cluster as an admin:

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

Select the `default` project:

```
$ oc project default
```

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

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

Delete the Router:

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

Deploy the NGINX Router:

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

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

Validate that your NGINX Router pods are running:

```
$ oc get pods
```

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

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

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

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

Discussion

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

CHAPTER 12

High-Availability Deployment Modes

12.0 Introduction

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

12.1 NGINX HA Mode

Problem

You need a highly available load-balancing solution.

Solution

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

Discussion

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

12.2 Load-Balancing Load Balancers with DNS

Problem

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

Solution

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

Discussion

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

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

12.3 Load Balancing on EC2

Problem

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

Solution

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

Discussion

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

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

12.4 Configuration Synchronization

Problem

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

Solution

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

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

For RHEL or CentOS:

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

For Ubuntu or Debian:

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

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

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

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

Get the IP address of the master node:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Test your configuration:

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

Discussion

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

12.5 State Sharing with Zone Sync

Problem

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

Solution

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

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

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

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

Discussion

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

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

CHAPTER 13

Advanced Activity Monitoring

13.0 Introduction

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

13.1 Enable NGINX Open Source Stub Status

Problem

You need to enable basic monitoring for NGINX.

Solution

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

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

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

Test your configuration by making a request for the status:

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

Discussion

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

13.2 Enabling the NGINX Plus Monitoring Dashboard Provided by NGINX Plus

Problem

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

Solution

Utilize the real-time activity monitoring dashboard:

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

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

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

Discussion

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

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

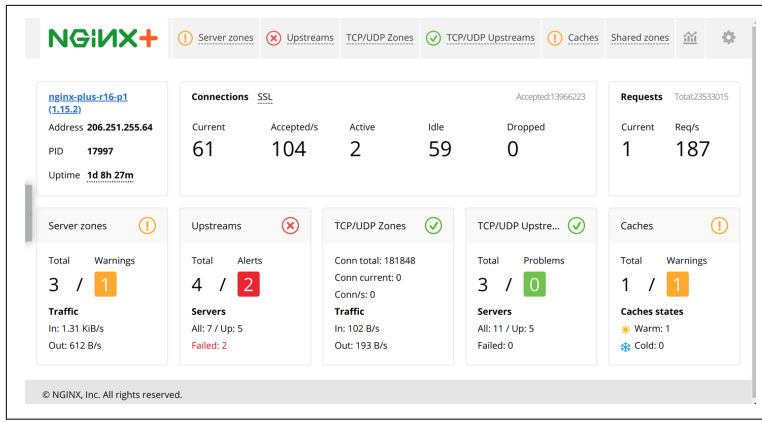

Figure 13-1. The NGINX Plus status dashboard

Also See

NGINX Plus Status Dashboard Demo

13.3 Collecting Metrics Using the NGINX Plus API

Problem

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

Solution

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

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

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

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

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

To limit information returned by the API, use arguments:

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

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

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

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

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

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

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

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

Discussion

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

Also See

NGINX HTTP API Module Documentation
NGINX API Swagger UI

CHAPTER 14

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

14.0 Introduction

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

14.1 Configuring Access Logs

Problem

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

Solution

Configure an access log format:

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

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

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

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

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

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

```
    ...
}
```

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

Discussion

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

14.2 Configuring Error Logs

Problem

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

Solution

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

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

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

Discussion

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

14.3 Forwarding to Syslog

Problem

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

Solution

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

```
error_log syslog:server=10.0.1.42 debug;

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

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

Discussion

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

14.4 Request Tracing

Problem

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

Solution

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

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

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

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

Discussion

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

CHAPTER 15

Performance Tuning

15.0 Introduction

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

15.1 Automating Tests with Load Drivers

Problem

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

Solution

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

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

Discussion

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

15.2 Keeping Connections Open to Clients

Problem

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

Solution

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

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

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

Discussion

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

15.3 Keeping Connections Open Upstream

Problem

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

Solution

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

```
proxy_http_version 1.1;
proxy_set_header Connection "";

upstream backend {
  server 10.0.0.42;
  server 10.0.2.56;

  keepalive 32;
}
```

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

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

Discussion

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

15.4 Buffering Responses

Problem

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

Solution

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

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

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

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

Discussion

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

15.5 Buffering Access Logs

Problem

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

Solution

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

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

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

Discussion

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

15.6 OS Tuning

Problem

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

Solution

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

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

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

Discussion

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

CHAPTER 16

Practical Ops Tips and Conclusion

16.0 Introduction

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

16.1 Using Includes for Clean Configs

Problem

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

Solution

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

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

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

Discussion

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

16.2 Debugging Configs

Problem

You're getting unexpected results from your NGINX server.

Solution

Debug your configuration, and remember these tips:

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

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

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

Discussion

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

Also See

How NGINX Processes Requests
Debugging Admin Guide
Rewrite Log

16.3 Conclusion

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

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

About the Author

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