CHAPTER 8
HTTP/2

8.0 Introduction

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

8.1 Basic Configuration

Problem

You want to take advantage of HTTP/2.

Solution

Turn on HTTP/2 on your NGINX server:

```
server {
    listen 443 ssl http2 default_server;

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

```
        ...
    }
```

Discussion

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

Also See

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

8.2 gRPC

Problem

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

Solution

Use NGINX to proxy gRPC connections.

```
    server {
        listen 80 http2;

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

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

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

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

```
server {
    listen 443 ssl http2 default_server;

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

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

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

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

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

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

This configuration example uses the `location` directive to route incoming HTTP/2 traffic between two separate gRPC services, as well as a `location` to serve static content. Method calls for the `mypackage.service1` service are directed to the `backend.local` server on port 50051, and calls for mypackage.service2 are directed to port 50052. The location / catches any other HTTP request and serves static content. This demonstrates how NGINX is able to serve gRPC and non-gRPC under the same HTTP/2 endpoint and route accordingly.

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

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

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

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

Discussion

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

8.3 HTTP/2 Server Push

Problem

You need to preemptively push content to the client.

Solution

Use the HTTP/2 server push feature of NGINX.

```
server {
    listen 443 ssl http2 default_server;

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

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

Discussion

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

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

CHAPTER 9

Sophisticated Media Streaming

9.0 Introduction

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

9.1 Serving MP4 and FLV

Problem

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

Solution

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

```
http {
    server {
        ...

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

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

Discussion

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

9.2 Streaming with HLS

Problem

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

Solution

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

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

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

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

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

Discussion

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

9.3 Streaming with HDS

Problem

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

Solution

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

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

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

Discussion

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

9.4 Bandwidth Limits

Problem

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

Solution

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

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

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

Discussion

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

CHAPTER 10

Cloud Deployments

10.0 Introduction

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

10.1 Auto-Provisioning on AWS

Problem

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

Solution

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

Discussion

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

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

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

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

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

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

Packer from HashiCorp

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

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

10.2 Routing to NGINX Nodes Without an AWS ELB

Problem

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

Solution

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

Discussion

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

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

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

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

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

Also See

Amazon Route53 Global Server Load Balancing

10.3 The NLB Sandwich

Problem

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

Solution

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

Discussion

This common pattern is called the *NLB sandwich* (see Figure 10-1), putting NGINX Open Source in an Auto Scaling group behind an NLB and the application Auto Scaling group behind another NLB. The reason for having NLBs between every layer is because the NLB works so well with Auto Scaling groups; they automatically register new nodes and remove those being terminated as well as run health checks and pass traffic to only healthy nodes. It might be necessary to build a second, internal NLB for the NGINX Open Source layer because it allows services within your application to call out to other services through the NGINX Auto Scaling group without leaving the network and re-entering through the public NLB. This puts NGINX in the middle of all network traffic within your application, making it the heart of your application's traffic routing. This pattern used to be called the *elastic load balancer* (ELB) *sandwich*; however, the NLB is preferred when working with NGINX because the NLB is a Layer 4 load balancer, whereas ELBs and ALBs are Layer 7 load balancers. Layer 7 load balancers transform the request via the proxy protocol and are redundent with the use of NGINX. This pattern is needed only for NGINX Open Source because the feature set provided by the NLB is available in NGINX Plus.

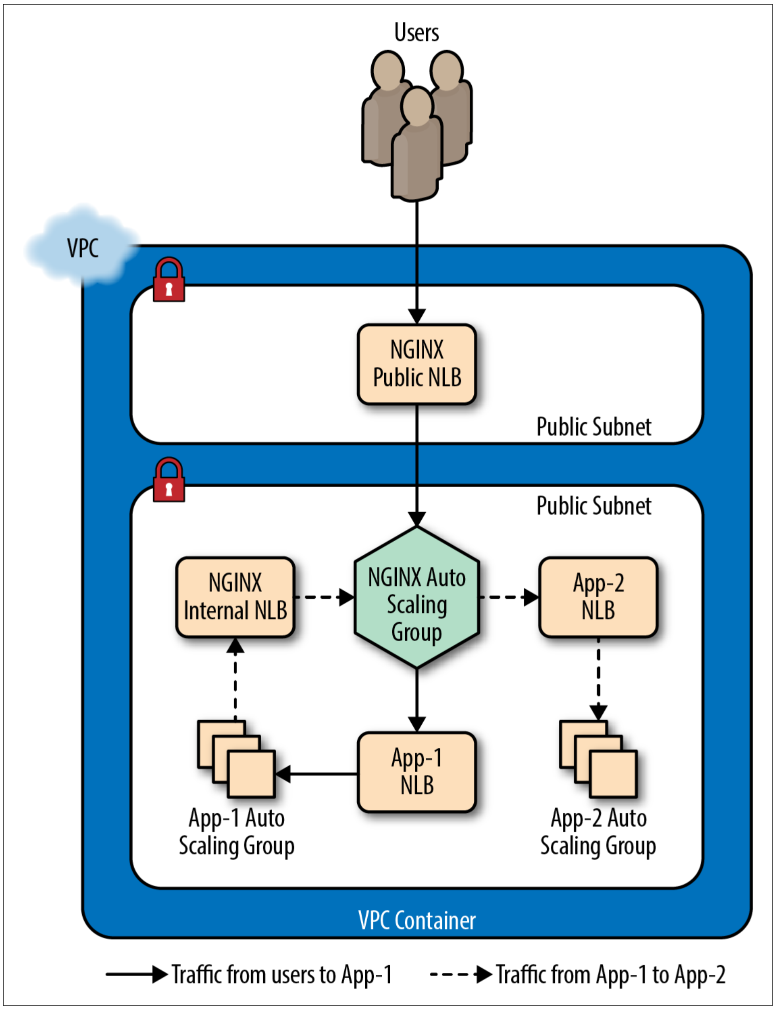

Figure 10-1. This image depicts NGINX in an NLB sandwich pattern with an internal NLB for internal applications to utilize. A user makes a request to App-1, and App-1 makes a request to App-2 through NGINX to fulfill the user's request.

10.4 Deploying from the AWS Marketplace

Problem

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