Terraform

Alejandro Campos

June, 2024

Contents

1 Introduction

1.1 What is NGINX?

NGINX is an open source web server written in C and can also be used as a reverse proxy and a load balancer. It serves the web content (static or dynamic) listening on an HTTP endpoint and understand how to talk this HTTP protocol.

As a **reverse proxy**, you can make NGINX face the internet getting the requests and then, internally it moves these requests across your back ends appropiately, it also can morph the request, authenticate the request, change the request, etc. It also can check if the back-end is down to not redirect traffic to this back-end. So in summary, reverse can provide:

- **Load Balancing:** balance the requests between n instances of back-end.
- **Back-end Routing:** redirect the traffic to one back-end or another depending on: the path they use, the domain, etc.
- **Cache:** cache requests that are repetitive directly on the NGINX, to avoid ask the back-end for some of the requests. Minimizing the latency and resource consumtion.
- **Firewall:** filter traffic, number of requests, etc. To avoid malicious usage of the back-ends.

1.2 Why do we need NGINX?

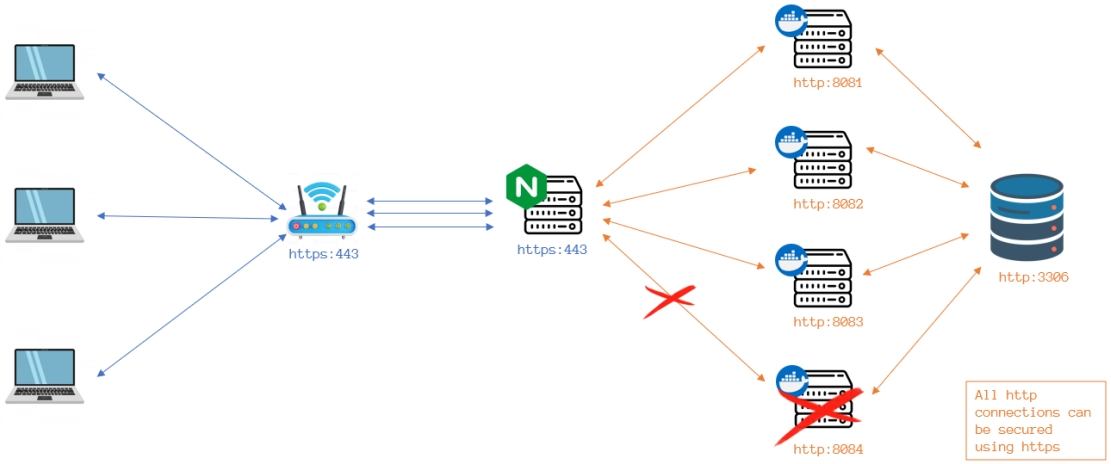
Imagine that we have our web application running without NGINX serving HTTPS. We should have:

- Physical Server or VM running on port 443 the service or docker container with the web content.
- Certificate installed and configured on the service or docker container.
- Router (public IP) with the port 443 opened to the World and forwarded to the same port on physical server.

What would happen if we want to grow up this infrastructure, we will need new servers, open new ports on the router, give the new ports to all the customers... It will be a mess, and if we have HTTPS what would you do? Share the certificate between all the new servers? It is dificult to handle. And what would happen if one of the machine is down? You would need to tell the client: hey, don't go to this port because it is down. It is a lot of work, isn't it?

That's why we introduce an extra layer as **reverse-proxy**, NGINX.

- HTTPS it's only configured from NGINX out, it uses HTTP to communicate withe the back-ends or you can as well enable more security and communicate using SSL but without having the web certificate in all the back-ends. So the web certificate just need to be configured on NGINX.
- NGINX completely hide back-ends, so clients just talk with NGINX, and it's NGINX who decides to which back-end send the requests, how, do it or not do it, etc.
- NGINX will load balance the request (by default using round-robin), and if one back-end is not alive it will not send requests to it.



Note

Round-robin load balancing is a method used to distribute network traffic across multiple servers. It ensures that no single server is overwhelmed with too many requests. Here's how it works:

Request Distribution: Incoming requests from clients are distributed to a pool of servers in a cyclic order. Each server in the pool gets an equal number of requests over time.

Sequential Assignment: The load balancer maintains a list of servers. The first incoming request is sent to the first server, the second request to the second server, and so on. Once the load balancer reaches the end of the list, it cycles back to the beginning.

Cycle Repeat: This process repeats continuously, ensuring that each server receives traffic in a round-robin fashion. For example, if there are three servers (A, B, C), the request distribution would follow the pattern: A, B, C, A, B, C, and so on.

Cons:

- Not Load-Aware: Standard round-robin does not account for the current load on each server, potentially leading to inefficiencies.
- Performance Variations: If servers have different performance capabilities, a plain round-robin approach might not utilize the servers optimally.
- Session Persistence: Does not inherently support session persistence (also known as sticky sessions), where a client's requests need to be handled by the same server.

The clients have no idea that their requests are going to one of the back-end servers, they just see the NGINX. In the same way, the back-end servers don't know the originating client of the requests they are answering, they know NGINX makes the request.

Note

Just to clarify, from now, we are going to talk about:

- **NGINX Front-end:** communication between NGINX and the clients.
- **NGINX Back-end:** communication between NGINX and the servers (back-ends).

You can configure your NGINX to perform at Layer 4, Layer 7 or both at the same time through the configuration NGINX file.

1.3 NGINX Layer 4 vs Layer 7 Proxing

Let's make a refresh on OSI Layers:

1.3.1 Layer 4 - Transport Layer

Transmits data using transmission protocols (like UPD and TCP). So information we have on Layer 4:

- Source IP and source Port (plain text information, never encrypted).
- Destination IP and destination Port (plain text information, never encrypted).
- Packet inspection: SYN (to stablish conneciton), TSL (to retrieve some data).

It is really useful when NGINX doesn't really understand the protocol, like for example a Postgres db protocol or MySQL. NGINX does not know how to terminate it, read it, parse it, turn around and talk to a backend that understand it.

NGINX on Layer 4:

- Block connections from source IPs or source Ports.
- Block connections to destination IPs or destination Ports.
- Raise events based on Layer 4 data retrieved.

1.3.2 Layer 7 - Application Layer

Human-computer interaction layer, where applications can access the network services. All application data use to be encrypted, so the reverse proxy **decrytps** all this data.

NGINX on Layer 7:

- Type of connection: HTTP, HTTPS, gRPC, etc.
- Filter, modify, redirect based on context.
- Know which pages clients are visiting, the headers of the requests, the cookies of the requests, the cookies of the navigator, etc.

It is really useful when NGINX understand the protocol, like for example HTTP or HTTPS, because it enables NGINX to redirect traffic, parse, modify, rewrite or add more headers, cache, etc.

1.4 TLS Termination vs TLS Passthrough

1.4.1 Introduction

TLS (Transport Layer Security) is a way to establish end-to-end encryption between one another, every time you use HTTPS your are using TLS. It uses a symmetric encryption for communication (client and server has the same key).

So the question is mandatory, how can I share the key in plain text? TLS uses **Asymmetric Encription** to exchange the symmetric key.

The reason for not do it all using Asymmetric Encryption is because it is very slow compared to Symmetric Encryption, so it will increase a lot the latency in all the connections, and we don't want that.

1.4.2 Certificates

How can you guarantee as server or client who are you talking to? There is no method of authentication, so anyone in the middle can just stop the key exchange and then reply back and sey: hey I'm the server or the client. And then intercept all the connections, decrypt them, and encrypt them again using its own key, performing a terminaton of TLS. It is commonly named as **man in the middle**.

How can we actually identify that you are talking with the right endpoint you want to talk? Using Certificates, that nobody else except the servers has. So, the server during the handshake of the TLS, will reply back with its own certificate, signed by a Certificate Authority (or certbot XD) demonstrating that he actually is which it says it is.

As well as server, client can be authenticate, and that's when we name the connection as **MTLS**

1.4.3 NGINX doing TLS Termination

NGINX can use HTTPS (TLS) meanwhile the back-ends not (you can choose to secure as well this connection). So NGINX does the same as **man in the middle**, it **terminates** the TLS connection, it **decrypts** all the data and send to the back-ends unencrypted (or encrypted if you enable the feature).

NGINX does this because he is the one which negotiates the key exchange with the clients and servers. So anyone sniffing traffic between the client and the NGINX won't see anything, because it is all encrypted, but if he is in the middle of the NGINX connections with the back-ends, he will see everything. But... If you have your infrastructure on a private network. Who would be sitting on your network?

So for cloud resources it is recommended to encrypt both sides (NGINX front-end connections and NGINX back-end connections). So in that case, NGINX will make the key exchange with the back-ends and encrypt the data again for them.

So for doing all of that, NGINX should have its own certificate with private and public keys configured inside.

1.4.4 NGINX doing TLS Passthrough

As well, you can configure NGINX to not TLS Terminate the connections, to don't act as man in the middle. So NGINX can just pipe everything through itself and send it to the backend with all the properties of load-balancing and that stuff, but without decrypt and encrypt anything. NGINX then will just stream the packets directly to the backends. And then, will be responsible of the back-ends to handle with certificates and keys.

The bad thing on this configuration is that NGINX cannot see anything regarding **Layer 7**, because the encryption is end-to-end. So NGINX cannot take any decition based on this data, just act as Layer 4 proxy, deciding what to do with the traffic just based on IP's and Ports.

1.5 NGINX Internal Architecture