NGINX - A Complete Guide

Introduction:

Understanding NGINX is essential for anyone involved in using web technology for personal projects or large scale enterprise applications. This article will explain what Nginx is, what it is used for, and also a hands-on demo which will run a simple Nodejs application, configure nginx, also secure the application using SSL/TLS certificate.

Overview:

- What is NGINX ?
- NGINX Use Cases
- How to Configure NGINX ?

What is NGINX ?

NGINX is an open source web server software which is fast, lightweight and provides high performance used to serve static files. When Nginx was introduced in the initial release it functioned for HTTP web serving. However today it is used as a reverse proxy, load balancing, caching, and health checks.

NGINX Use Cases

1. **NGINX as Web Server**

One of nginx's primary function is to serve static content delivery which includes HTML, CSS and JavaScript. When a user requests a web page, nginx locates and serves the necessary static content. It can also handle multiple requests concurrently, ensuring maximum performance under heavy load.

2. **NGINX as Load Balancer**

As the web grew in popularity, single servers began receiving millions of requests. To handle this load, multiple servers became necessary. But how do we determine which server handles which request? This is where NGINX acts as a load balancer. Positioned at the entry point, it distributes incoming web traffic across multiple servers. This approach ensures that no single server becomes a bottleneck, allowing requests to be served efficiently. The distribution of the load depends on the configured algorithm. For example, it might use simple logic like "least connections" or "round robin."

3. **NGINX as Caching**

Caching is another functionality of a proxy server. Caching is a process of storing data temporarily so that future requests for the same data can be served faster. As Nginx acts as reverse proxy that can cache response from the backend server. It stores these responses in a cached location in the server. When a client makes a request, if the response is available, it will serve immediately. If not, Nginx forwards the request to backend server and caches the new response.

4. **NGINX as Security**

The proxy server acts as a **single entry point** that is publicly available, protecting all other web servers and minimizing their exposure. So one can focus on this one single entry point protecting. Nginx also handle SSL/TLS termination and encryption ensuring secure client and server communication. So when an encrypted data is sent to the proxy, even if attacker tries to intercept, they cannot read the message.

5. **NGINX as Compression**

Consider an example of Netflix or any other streaming platform. When a popular series drops, millions of users rush to access the content at the same time. This can lead to enormous bandwidth consumption so Nginx with its compression capabilities can optimize this scenario by reducing the size and ensure faster load times. It sends response in chunks instead of sending the entire file at once.

How to Configure NGINX ?

As we discusses all the functionalities of NGINX, So how does one can configure NGINX to work as proxy server, caching, SSL etc?

NGINX Config File

Configuring Nginx involves changing the Nginx Configuration files to customize its behavior.

1. Locate the Nginx config file, usually it is located in `/etc/nginx/nginx.conf` location.
2. Use command line text editor to open the configuration file.
3. The nginx configuration file consists of multiple block, directives. Each block defines specific content.

Common Blocks:

- **http:** Configure settings for handling web traffic
- **server:** Configure a virtual host, ports.
- **location:** Where to find the files, or pass requests.

Common Directives:

- **listen**: Specifies the IP address and port that Nginx should listen on
- **server_name**: Defines the Domain name or IP address
- **root**: Sets the root directory where Nginx will look for the files to serve

- **index**: You can customize the index files. By default, Nginx has index.html, index.htm, index.php in this list
- **try_files**: Defines the order in which Nginx should try different URI paths.

Examples:

1. Nginx as web server

```
server{
        listen 80;
        server_name example.com;

        location / {
                root /var/www/example.com;
                index index.html index.htm;
        }
}
```

2. Nginx as Proxy to other web servers

```
server{
        listen 80;
        server_name api.example.com;

        location / {
                proxy_pass http://backend_server_address;
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_f
                proxy_set_header X-Forwarded-Proto $scheme;
        }
}
```

3. Nginx as Redirect http to https

```
server{
        listen 80;
        server_name example.com www.example.com;

        #redirect all http requests to https
        return 301 http://$host$request_uri;
}

server {
        listen 443 ssl;
        server_name example.com www.example.com;

        #SSL Configuration
        ssl_certificate /etc/letsencrypt/live/example.com/fullch
        ssl_certificate_key /etc/letsencrypt/live/example.com/pr

        #Security headers
        add_header Strict-Transport-Security "max-age=31536000;

        location / {
                root /var/www/example.com;
                index index.html index.htm;
        }
```

4. Nginx as Load Balancer

```
http {
        upstream myapp {
                least_conn;
                server srv1.example.com;
                server srv2.example.com;
                server srv3.example.com;
        }

        server {
                listen 80;

                location / {
                        proxy_pass http://myapp1;
                }
        }
}
```

`least_conn` **will forward the request to least busy server.**

5. NGINX as Caching

```
http {
        # ..
        proxy_cache_path /data/nginx/cache keys_zone=mycache:10r
}
```

The time indicates how long the cache should be stored before its refreshed.

NGINX as Kubernetes Ingress Controller

What is Kubernetes Ingress Controller ?

Ingress Controller is a specialized load balancer that watches for the ingress resource and then proceeds the rules which is defined within it. Kubernetes Ingress provides a centralized and efficient way to manage the external access to services running within a kubernetes cluster. Kubernetes Ingress is specifically designed for kubernetes environments, enabling advanced traffic management for containerized applications.

Nginx Load Balancer vs Cloud Load Balancer

Nginx Ingress Controller which acts a Load Balancer is used inside the kubernetes cluster, which means it is not publicly accessible. So the Cloud load Balancer is the entry point from where the request is forwarded to Ingress Controller. The cluster is never exposed publicly and the request always comes from the load balancer which then forwards it to the Ingress Controller within the cluster. Then the routing takes place based on the path or host matching.

NGINX vs Apache

Both were released as a basic web servers, later added more functionalities which we have discussed earlier.

Choose NGINX

- Faster and Lightweight
- High performance
- Static Content

Choose Apache

- Shared Hosting
- Customizable Solutions
- Dynamic Content

Hands-on Demo

Title: Build a simple Node.js Application which serves static files, dockerize it, configure Nginx, test load balancing and finally encrypt connection with https.

> Note: The application used in this article is a similar project I created, which aligns with the concepts shown in the original tutorial.

Steps:

Running Simple Web Application using localhost.

1. Open your favorite editor, create a folder called public and inside the public folder create a file called index.html which has your static content,
2. Paste the below code to your index.html file.

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-s
    <title>Welcome to DevOps Insights by BALA VIGNESH</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 0;
            background-color: #f4f4f4;
            color: #333;
        }
        header {
            background-color: #4CAF50;
            padding: 10px 20px;
            text-align: center;
            color: white;
        }
        nav {
            display: flex;
            justify-content: center;
            padding: 10px;
            background-color: #333;
        }
        nav a {
            color: white;
            padding: 10px;
            text-decoration: none;
        }
        nav a:hover {
            background-color: #ddd;
            color: black;
        }
        .container {
            padding: 20px;
        }
        footer {
            text-align: center;
            padding: 10px;
            background-color: #333;
            color: white;
            position: fixed;
            bottom: 0;
            width: 100%;
        }
    </style>
</head>
<body>
    <header>
        <h1>Welcome to DevOps Insights by BALA VIGNESH</h1>
    </header>
    <nav>
        <a href="index.html">Home</a>
        <a href="about.html">About</a>
    </nav>
    <div class="container">
        <h2>Your Gateway to DevOps Insights</h2>
        <p>At DevOps Insights, I share my journey, knowledge, ar
        <p>Connect with me on <a href="https://www.linkedin.com/
    </div>
    <footer>
        <p>&copy; 2024 DevOps Insights. All rights reserved.</p
    </footer>
</body>
</html>
```

3. Create another file in the same location which is called about.html, and paste the below code.

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-s
    <title>About - DevOps Insights/title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 0;
            background-color: #f4f4f4;
            color: #333;
        }
        header {
            background-color: #4CAF50;
            padding: 10px 20px;
            text-align: center;
            color: white;
        }
        nav {
            display: flex;
            justify-content: center;
            padding: 10px;
            background-color: #333;
        }
        nav a {
            color: white;
            padding: 10px;
            text-decoration: none;
        }
        nav a:hover {
            background-color: #ddd;
            color: black;
        }
        .container {
            padding: 20px;
        }
        footer {
            text-align: center;
            padding: 10px;
            background-color: #333;
            color: white;
            position: fixed;
            bottom: 0;
            width: 100%;
        }
    </style>
</head>
<body>
    <header>
        <h1>About DevOps Insights/h1>
    </header>
    <nav>
        <a href="index.html">Home</a>
        <a href="about.html">About</a>
    </nav>
    <div class="container">
        <h2>About Me</h2>
        <p>I am a DevOps engineer who is passionate about sharin
        <p>Through my blogs and articles, I delve into various a
        <p>Feel free to connect with me on <a href="https://www
    </div>
    <footer>
        <p>&copy; 2024 DevOps Insights. All rights reserved.</p
    </footer>
</body>
</html>
```

4. Create a backend web server using the index,js file. Copy the below code into it.

```
const express = require('express');
const path = require('path');
const app = express();

// Get the application name from the environment variable
const appName = process.env.APP_NAME || 'Node App';

// Serve static files from the "public" directory
app.use(express.static(path.join(__dirname, 'public')));

// Route to serve the index page
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

// Route to serve the about page
app.get('/about', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'about.html'));
});

// Start the server on the specified port
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`${appName} is running on http://localhost:${PORT
});
```

5. Add package.json file in the next step

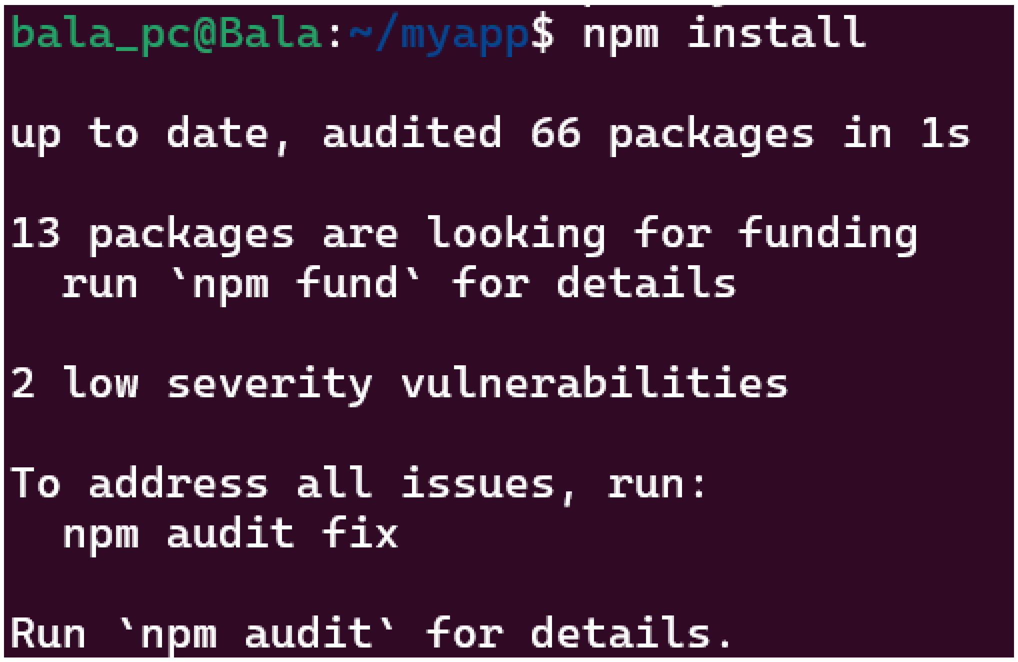
```
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.21.0"
  }
}
```

6. We need to run npm command to install the packages defined in the package.json file. Before that ensure you have installed Node in your system.
7. Install node using the below command (Linux)

```
curl -sL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt install nodejs
```

8. Install the dependencies by running the npm command

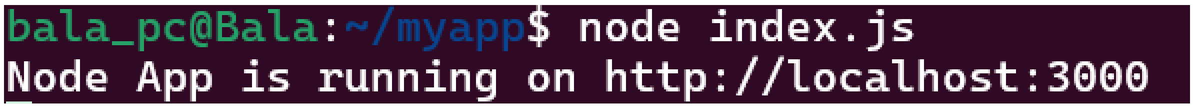
```
npm install
```



```
bala_pc@Bala:~/myapp$ npm install

up to date, audited 66 packages in 1s

13 packages are looking for funding
  run `npm fund` for details

2 low severity vulnerabilities

To address all issues, run:
  npm audit fix

Run `npm audit` for details.
```

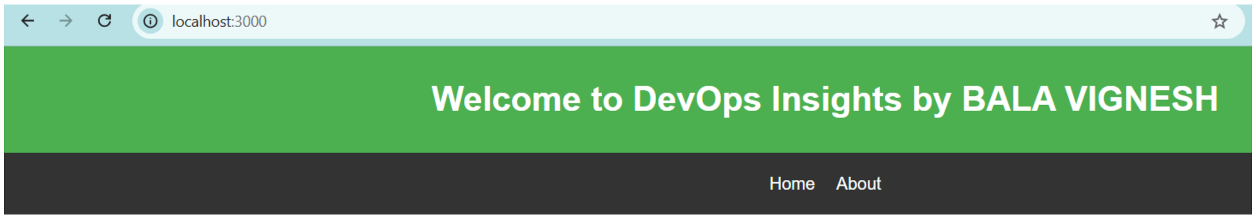
9. Now you can see that the dependencies are installed. Next step is to run the application.
10. Run the application using the below command.

```
node index.js
```



```
bala_pc@Bala:~/myapp$ node index.js
Node App is running on http://localhost:3000
```

11. Go to your browser and access your application using `http://localhost:3000`.



Running Simple Web Application using Docker and NGINX

1. First step is to create a Dockerfile.

```
FROM node:14-alpine

WORKDIR /app

COPY . .

RUN npm install

EXPOSE 3000

CMD ["node", "index.js"]
```

2. Build the Dockerfile using the below command.

```
docker build -t balav8/myapp:1.0 .
```

3. Check the docker image which we have created.

```
docker images | grep myapp
```

4. Run the application using the below command

```
docker run -p 3000:3000 balav8/myapp:1.0
```

5. Check your browser to see the application working.
6. Stop the container, as we need 3 instances of the same application.
7. We will achieve the same using Docker Compose. Docker Compose is a tool which is used to run multiple containers easily.
8. Let us create a docker-compose file.

```
version: '3'
services:
  app1:
    build: .
    environment:
      - APP_NAME=App1
    ports:
      - "3001:3000"

  app2:
    build: .
    environment:
      - APP_NAME=App2
    ports:
      - "3002:3000"

  app3:
    build: .
    environment:
      - APP_NAME=App3
    ports:
      - "3003:3000"
```

Here we are running 3 services, each service corresponds to a container.

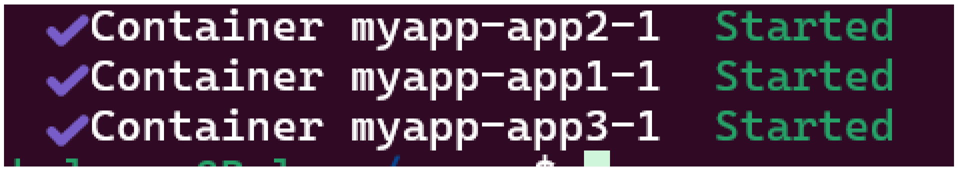
- app1: Name of the service.
- build: . Build the image using the Dockerfile in the current directory.
- environment: Sets the environment variable APP_NAME to App1.
- ports: Maps port on the host to the port on the container.

9. Since we have added environment variable, we need to make a slight change in our index.js file.

```
const express = require('express');
const path = require('path');
const app = express();

// Get the application name from the environment variable
const appName = process.env.APP_NAME || 'DefaultApp';

// Serve static files from the "public" directory
app.use(express.static(path.join(__dirname, 'public')));

// Route to serve the index page
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
  console.log(`Request served by ${appName}`);
});

// Route to serve the about page
app.get('/about', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'about.html'));
  console.log(`Request served by ${appName}`);
});

// Start the server on the specified port
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`${appName} is running on http://localhost:${PORT
});
```

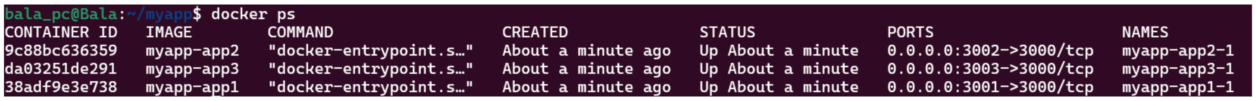
10. Run this compose file using the below command:

```
docker-compose up --build -d
```



```
✓Container myapp-app2-1  Started
✓Container myapp-app1-1  Started
✓Container myapp-app3-1  Started
```

11. Check if the containers are running using docker ps command.



```
bala_pc@Bala:~/myapp$ docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED              STATUS              PORTS                    NAMES
9c88bc636359   myapp-app2    "docker-entrypoint.s…"   About a minute ago   Up About a minute   0.0.0.0:3002->3000/tcp   myapp-app2-1
da03251de291   myapp-app3    "docker-entrypoint.s…"   About a minute ago   Up About a minute   0.0.0.0:3003->3000/tcp   myapp-app3-1
38adf9e3e738   myapp-app1    "docker-entrypoint.s…"   About a minute ago   Up About a minute   0.0.0.0:3001->3000/tcp   myapp-app1-1
```

12. You can individually check the site with all the ports running. You can also check the logs and understand about each containers.

Configure NGINX so that we do not want to access each server separately. We want 1 entry point which forwards the request to one of the backend servers.

1. Install NGINX locally (Linux)

```
sudo apt install nginx
```

2. We need to configure the sites-available file as per our application.

```
sudo nano /etc/nginx/sites-available/nodejs
```

3. Paste the below code into the file.

```
# Main context (global configuration)
    # Upstream block to define the Node.js backend servers
    upstream nodejs_cluster {
        least_conn;  # Distribute requests to the server with tl
        server 127.0.0.1:3001;
        server 127.0.0.1:3002;
        server 127.0.0.1:3003;
    }

    # Server block for HTTP traffic
    server {
        listen 8080;  # Listen on port 80 for HTTP
        server_name localhost;

        # Proxy requests to the Node.js cluster
        location / {
            proxy_pass http://nodejs_cluster;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
```

- **upstream**: A block that defines a group of backend servers for load balancing.

4. Create a symbolic link to sites-enabled. This tells nginx as in which site to serve.

```
sudo ln -s /etc/nginx/sites-available/nodejs /etc/nginx/sites-er
```

5. Remove the default configuration to avoid conflict.

```
sudo rm /etc/nginx/sites-enabled/default
```

6. Test your configuration file.

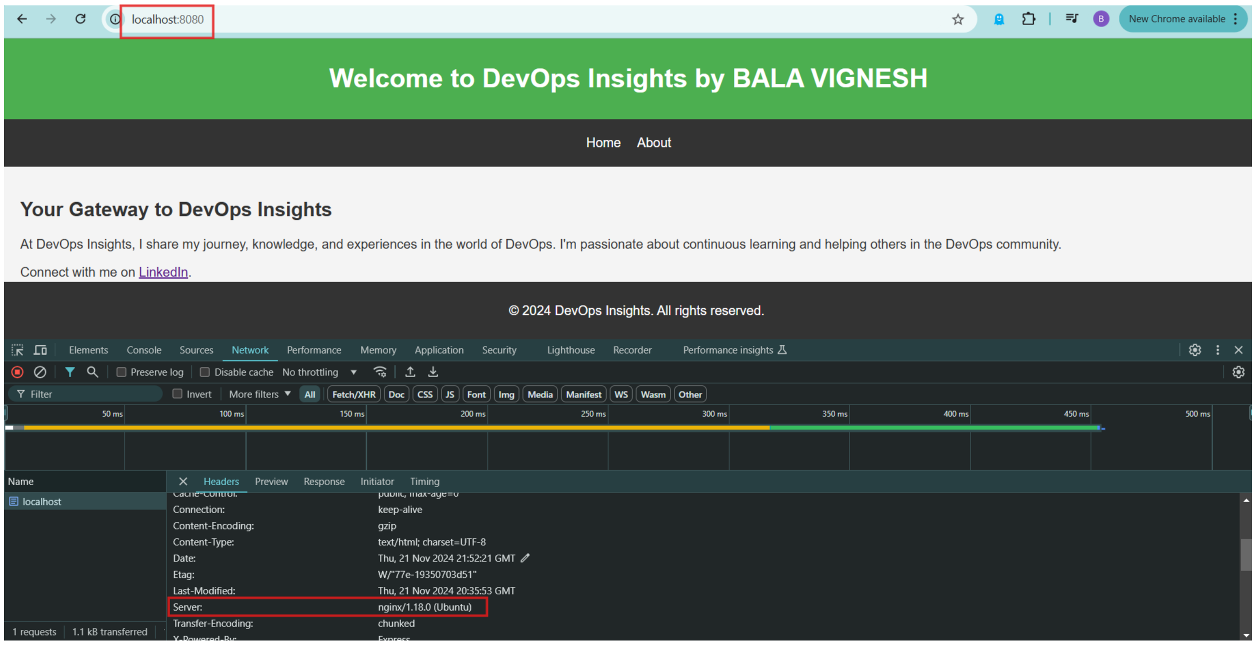
```
sudo nginx -t
```

7. Reload Nginx to apply the changes.

```
sudo systemctl reload nginx
```

8. Go to your browser and access the site with port 8080

```
http://localhost:8080
```



Welcome to DevOps Insights by BALA VIGNESH

Your Gateway to DevOps Insights

At DevOps Insights, I share my journey, knowledge, and experiences in the world of DevOps. I'm passionate about continuous learning and helping others in the DevOps community.

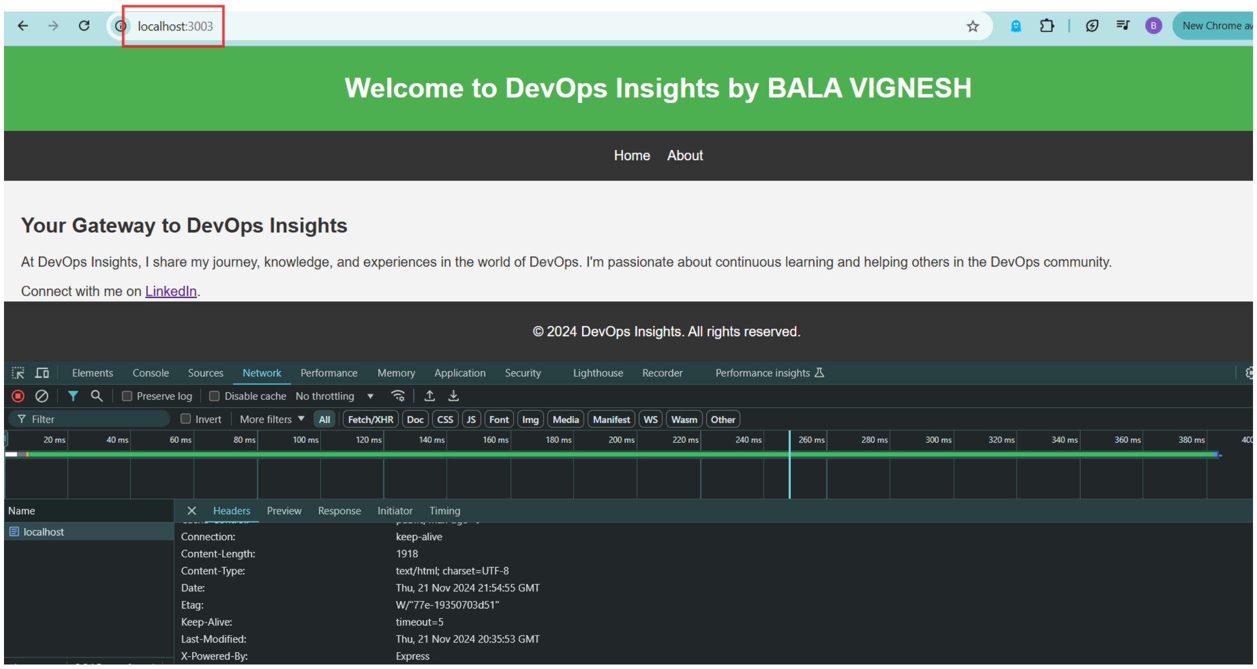
Connect with me on LinkedIn.

© 2024 DevOps Insights. All rights reserved.

```
localhost

Cache-Control:       public, max-age=0
Connection:          keep-alive
Content-Encoding:    gzip
Content-Type:        text/html; charset=UTF-8
Date:                Thu, 21 Nov 2024 21:52:21 GMT
Etag:                W/"77e-19350703d51"
Last-Modified:       Thu, 21 Nov 2024 20:35:53 GMT
Server:              nginx/1.18.0 (Ubuntu)
Transfer-Encoding:   chunked
X-Powered-By:        Express

1 requests | 1.1 kB transferred
```

9. Voila! Its running
10. Let us check to which server is the request getting forwarded to.
11. Open the localhost with port 8080, and check the developer tab. Clear the cache and check the network tab to see the server as Nginx. This means that the request is going through Nginx.

12. If you open the localhost with port 3003, in the network tab you can see there is no server configuration displayed. This means that the request is going directly to the 3rd server.


localhost:3003
Home About
Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights
Preserve log Disable cache No throttling
Filter Invert More filters All Fetch/XHR Doc CSS JS Font Img Media Manifest WS Wasm Other
Name
localhost
Headers Preview Response Initiator Timing

Welcome to DevOps Insights by BALA VIGNESH

Your Gateway to DevOps Insights

At DevOps Insights, I share my journey, knowledge, and experiences in the world of DevOps. I'm passionate about continuous learning and helping others in the DevOps community.

Connect with me on LinkedIn.

© 2024 DevOps Insights. All rights reserved.

```
Connection:          keep-alive
Content-Length:      1918
Content-Type:        text/html; charset=UTF-8
Date:                Thu, 21 Nov 2024 21:54:55 GMT
Etag:                W/"77e-19350703d51"
Keep-Alive:          timeout=5
Last-Modified:       Thu, 21 Nov 2024 20:35:53 GMT
X-Powered-By:        Express
```

13. In real life only the Nginx port will be open rest all the ports where the backend server is running will never be exposed and it won't be accessible. It will only be accessible through Nginx as it is more secure since it will have only one entry point.

Configure HTTPS for a Secure Connection

In today's world security is a paramount. Configuring HTTPS on your web server is a crucial part to protect sensitive data. We will implement a robust certificate management system, with that we can ensure all the communications between the client and the server are encrypted.

There are usually two ways to implement this certificate process.

1. Use Let's Encrypt (A Certificate Authority) using Certbot which issues free certificates. However, you need to have a Domain for this type. This is used for production type cases.

2. Use Self-Signed Certificate. This is not a perfect solution though, it will show a warning that they have a certificate but it is not issued from a Certificate Authority. This is often used for testing and development.

In this article, we will be using the 2nd option as we are using for testing purpose only.

Self-Signed Certificate

1. Create a folder to store the certificate files in your system.

```
mkdir nginx-certs
```

2. Go inside the nginx-certs folder.

```
cd nginx-certs
```

3. Generate the certificate using openssl command. openssl is a open source tool which is used to generate keys, certificates and manage secure connections.

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ngi
```

- openssl req: Initiates the certificate request
- -x509: Output the certificate in this standard format
- -nodes: Not to encrypt the private key with a passphrase
- -days 365: Specifies the validity of this certificate
- -newkey rsa:2048: Creates a 2048 bit RSA key pair
- -keyout nginx-selfsigned.key: Output file for the generated private key
- -out nginx-selfsigned.crt: Output file for the self signed certificate and public key

4. This command will generate few prompts.

- Country Name: IN
- State:
- Locality
- Organization Name:
- Common Name:
- Email:

You can leave some of the fields empty, and press Enter in the end.

4. Type `ls` and you can see both the certificate file and the private key.

Configure NGINX Server with the certificate and the key.

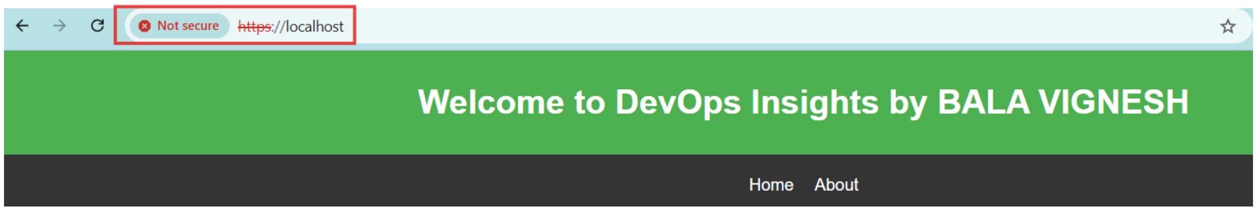
1. Open the nginx.conf file, we need to make some changes in this file.

```
# Main context (this is the global configuration)
    # Upstream block to define the Node.js backend servers
    upstream nodejs_cluster {
        server 127.0.0.1:3001;
        server 127.0.0.1:3002;
        server 127.0.0.1:3003;
    }

    server {
        listen 443 ssl;  # Listen on port 443 for HTTPS
        server_name localhost;

        # SSL certificate settings
        ssl_certificate /Users/bala/nginx-certs/nginx-selfsigne
        ssl_certificate_key /Users/bala/nginx-certs/nginx-selfs

        # Proxying requests to Node.js cluster
        location / {
            proxy_pass http://nodejs_cluster;
            proxy_set_header Host $host;
```

```
                proxy_set_header X-Real-IP $remote_addr;
            }
        }

        # Optional server block for HTTP to HTTPS redirection
        server {
            listen 8080;  # Listen on port 80 for HTTP
            server_name localhost;

            # Redirect all HTTP traffic to HTTPS
            location / {
                return 301 https://$host$request_uri;
            }
        }
    }
```

- We have added the ssl_certificate configuration.
- Added additional server block for port 8080 and redirected, so that someone who is trying to access the application in http gets redirected to https secure connection.

2. Open your browser and try to access the localhost with port 443.



As you can see the site gets loaded with https, but since it is a self signed certificate it shows it is not a valid certificate.

3. You can also try to access the site on localhost:8080 which will redirect to the secured https site.
4. By the way, the default port for http is port 80. You can make the change in the config file and see the changes.

This is how one can use Nginx as a reverse proxy server no matter which tech stack you’re using.

This completes our comprehensive guide on NGINX.

Clean Up:

1. Stop the nginx processes.
2. Stop the docker containers.

Conclusion:

In this article, we have explored what NGINX is and how it can be used and all its features including a Hands-on demo which makes easy to understand the working process.

NGINX is an open source web server software. Due to its simple setup and extensive documentation NGINX is very well suited for beginners. It has many features as seen above such as handling of static files, load balancing, caching, compression and SSL/TLS. These features and its scalability makes NGINX more popular and ideal for modern day web infrastructures.

If you found this post useful, give it a like👍

Repost♻️

Follow @Bala Vignesh for more such posts 💯🚀