VAASAN AMMATTIKORKEAKOULU
VASA YRKESHÖGSKOLA
UNIVERSITY OF APPLIED SCIENCES

Information Technology Degree Program

Database and Open Interfaces

Northwind Database with GraphQL Backend

Project duration

14/02/2022 – 27/02/2022

Siyuan Xu

Team: C
Class: IT-2020
Date: 27/02/2022

Abstract

Most of the modern apps connect to a backend database system. For certain reasons (security, complexity, frontend developer skill requirement), the frontend (client) mostly do not connect to the database straight. Web APIs are therefore as middle ware to connect frontend and backend.
In this project GraphQL API were created for database as backend solution. The backend system was built on an Linux(Ubuntu Server) machine on the cloud(UpCloud). Postgre was used as database system. PGAdmin was used as web GUI based administration tool for Postgres database system. Postgraphile was used as GraphQL API server. Security was done by self-signed SSL and JWT(Json Web Token). The whole system was configured to run in docker. A lightweight hypervisor. Finally, OpenAPI was used to document the APIs.

Introductions

UpCloud

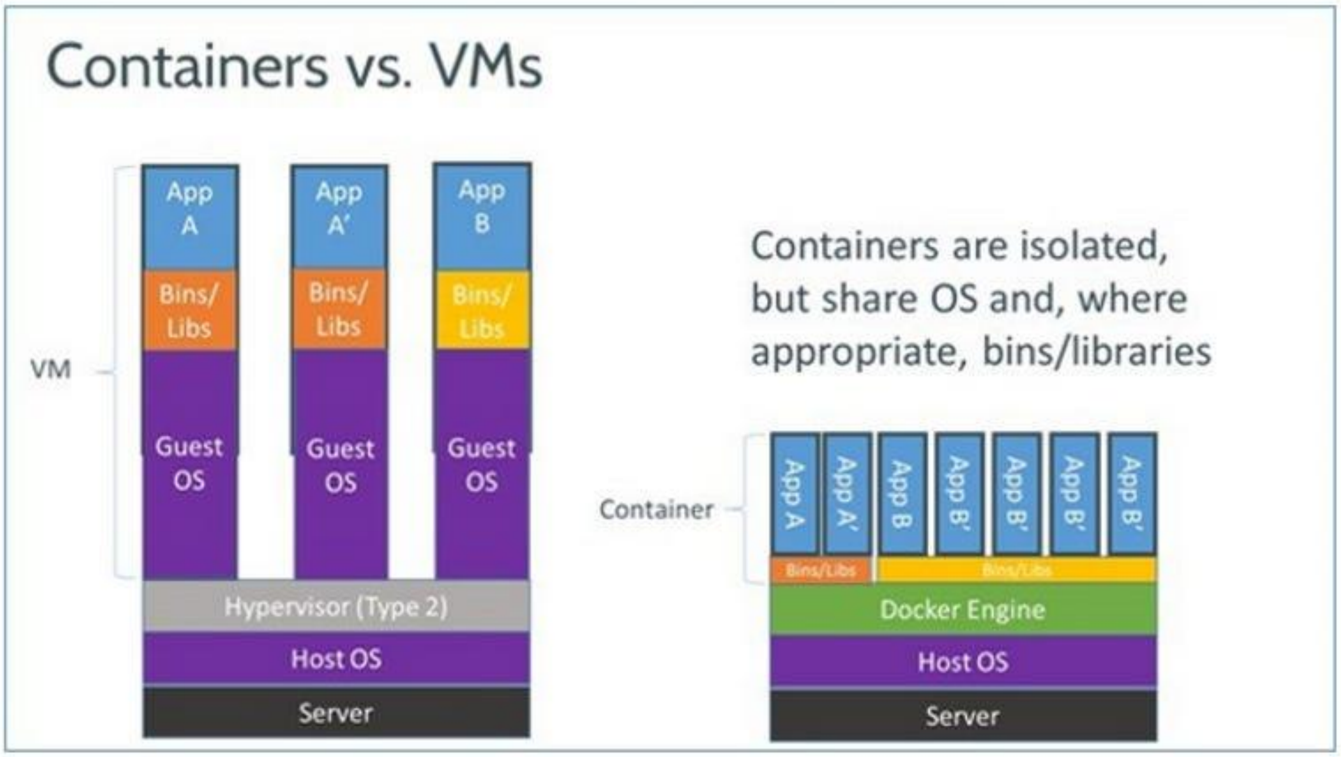
UpCloud is as Finnish cloud platform. It offers cloud computer at affordable price.

GNU/Linux

GNU/Linux (hope this will make Richard Stallman smile :D) is an open source operating system. There are free (as in freedom and free beer) distributions. Ubuntu Server 20.04 LTS was used.

Docker

Docker is a kind of hypervisor software. It differs from Virtual Machines that all docker containers share the same OS (the host OS). Therefore, it is much more efficient than VMs. The drawback is you cannot choose a different kernel nor OS to run the containers.



PostgreSQL

PostgreSQL is free and open-source relational database management system. It was developed by University of California, Berkeley.

PGAdmin

PGadmin is an open-source web based administration tool for PostgreSQL.

GraphQL

GraphQL is an opensource data query and manipulation language for APIs. It was developed first by Facebook in 2012. Then it was open sourced in 2015. The name GraphQL implicates that it is based on Graph Theory. The client could request the exact needed data from GraphQL API by GraphQL queries.

Postgraphile

Postgraphile is a software automatically builds GraphQL APIs from a PostgreSQL database.

NGINX

NGINX is pronounced “engine X”, is a web server software could also be used as a reverse proxy, load balancer, mail proxy and HTTP cache.

OpenSSL

OpenSSL is an open source software library for applications that secure communications over computer networks. It is an open source implementation of SSL and TLS protocols.

JWT

JSON Web Token is a JSON Object which is used to securely transfer information over the web. It can be used for an authentication system and can also be used for information exchange.

OpenAPI/Swagger

The OpenAPI Specification, aka Swagger Specification, is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services.

GraphDoc

GraphDoc is a free and open source static page generator for documenting GraphQL Schema.

Procedure

Ubuntu Server

Initialize the Server

Ubuntu Server 20.04 LTS was setup on UpCloud. The public IP address for the cloud host was 5.22.220.255. Putty with SSH
protocol was used to connect to the remote host. It was done by the professor Timo Kankaanpää. The root user and password were used in this project.

Update the server

```
sudo apt update
sudo apt dist-upgrade
```

Docker

Install Prerequisite software/libraries:

```
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common
wget
```

Add apt-key for docker PPA

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Add Docker PPA

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable"
```

```
sudo apt update
```

Install Docker CE and Docker-Compose (Not used in this project, but useful)

```
sudo apt install docker-ce docker-compose
```

Test for Docker

```
docker run hello-world //run testing container hello-world
docker ps -a           //list all containers in docker
```

The information below should be shown:

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
2ae79125cfa7 hello-world "/hello" 10 seconds ago Exited (0) 9 seconds ago
agitated_benz
```

```
docker container rm 2ae79125cfa7  //CONTAINER_ID_FOR_HELLO_WORLD
```

Docker is ready now!

PostgreSQL

Prerequisite

All containers could reach outside freely. However, to connect to the container is done by docker virtual network bridge. By default, it was running at IP address 172.17.0.1. PostgreSQL database server by default run at port 5432. Therefore, we need to map the ports.

Create Docker bridge network

```
docker network create --driver bridge postgres-network
```

Install PostgreSQL Docker Container

```
docker run --name postgres-with-network --network=postgres-network -v
/root/docker/data:/var/lib/postgresql/data -e
POSTGRES_PASSWORD=PlacePostgresAdminPasswordHere -p 5432:5432 -d postgres
```

`docker run`	**creates a container from a given image and starts the container using a given command**.
`--name`	Container name
`--network=`	Connect a container to a network
`-v`	Bind mount a volume. host-src:container-dest
`-e`	Set ENVIRONMENT VARIABLE in the container
`-p`	Publish a container's port or a range of ports to the host hostPort:containerPort
`-d`	Start a container in detached mode

```
docker ps -a
```

The information below should be shown:

```
CONTAINER ID   IMAGE    COMMAND                  CREATED          STATUS          PORTS                    NAMES
74ba0bb1877e   postgres   "docker-entrypoint.s…"   45 seconds ago   Up 45 seconds   0.0.0.0:5432->5432/tcp,
:::5432->5432/tcp    postgres-with-network
```

Download NorthWind Database Query

```
wget https://github.com/pthom/northwind_psql/blob/master/northwind.sql
```

Add NorthWind Database to PostgreSQL in Docker

```
psql -h 172.17.0.1 -U postgres -f northwind.sql
```

Check NorthWind Database

```
psql -h 172.17.0.1 -U postgres
```

PostgrSQL commands

\l	List all databases
\c	Connect to a database
\dt	Display all tales
\du	Display all users

PGADMIN

Install PGAdmin

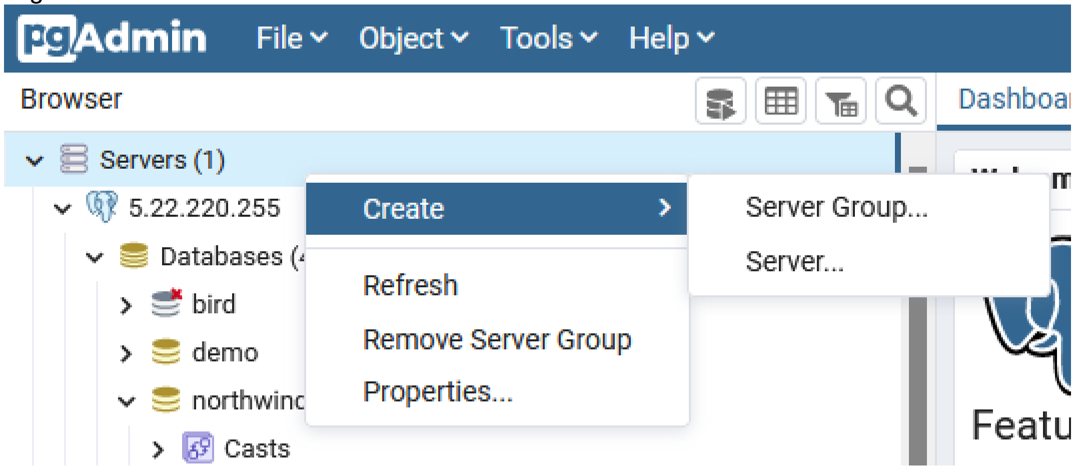
```
docker run --name pgadmin --network=postgres-network -p 80:80 -e
PGADMIN_DEFAULT_EMAIL=e2101066@edu.vamk.fi -e PGADMIN_DEFAULT_PASSWORD=admin123 -d
dpage/pgadmin4

docker ps -a //check if it's running correctly
```

The information below should be shown:

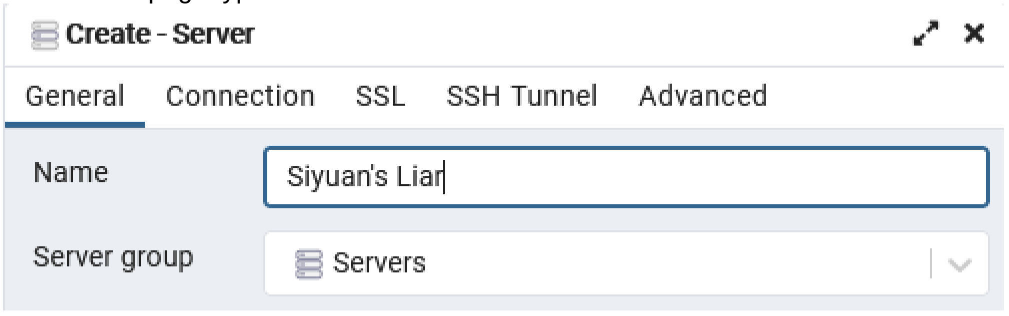
```
CONTAINER ID   IMAGE                    COMMAND                  CREATED          STATUS
PORTS                                          NAMES
ff944268ee38   dpage/pgadmin4           "/entrypoint.sh"         51 minutes ago   Up 51
minutes   0.0.0.0:80->80/tcp, :::80->80/tcp, 443/tcp    pgadmin
74ba0bb1877e   postgres                 "docker-entrypoint.s…"   57 minutes ago   Up 57
minutes   0.0.0.0:5432->5432/tcp, :::5432->5432/tcp     postgres-with-network
```

Configure PGAdmin

1. Go to server's public IP address via a browser: 5.22.220.255
2. Login with email address and password configured from ealier
3. Right click Servers -> Create -> Server…

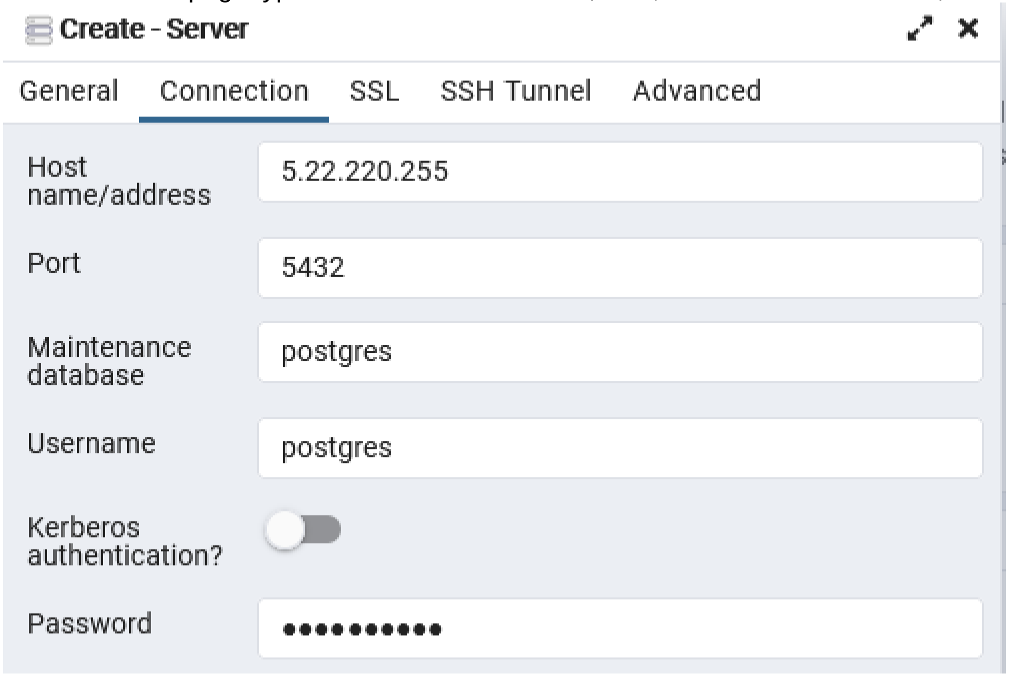


4. On *General* page type a name for the server



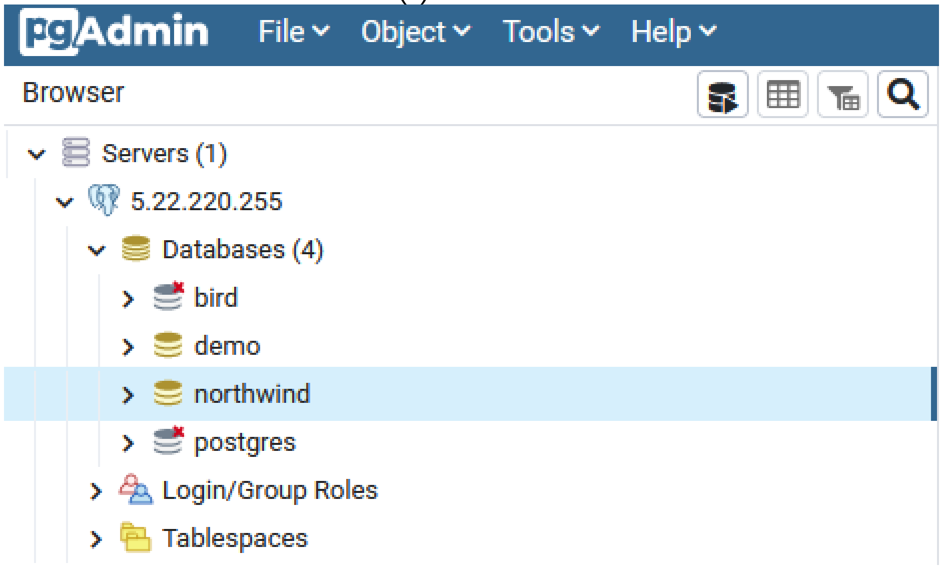
Name: Siyuan's Liar
Server group: Servers

5. On *Connection* page type the *Host name/address*, *Port*, *Maintenance database*, *Username* and *Password*



Host name/address: 5.22.220.255
Port: 5432
Maintenance database: postgres
Username: postgres
Kerberos authentication?
Password: ●●●●●●●●●●●

Now we should see our database(s)



PostgreSQL Security

PostgreSQL offers robust security features such as: role(user) based privilege control, database level security, table level security, column level and row level security.

Login To PostgreSQL

```
psql -h 172.17.0.1 -U postgres
```

Add Two tables to NorthWind database: *api_admin* and *api_public*

```
CREATE TABLE public.api_admin
(
    id serial PRIMARY KEY,
    username varchar(50) NOT NULL,
    password varchar(100) NOT NULL,
    email varchar(50) NOT NULL,
    CONSTRAINT api_admin_email_key UNIQUE (email)
)

CREATE TABLE public.api_public
(
    id serial PRIMARY KEY,
    username varchar(50) NOT NULL,
    password varchar(100) NOT NULL,
    email varchar(50) NOT NULL,
    CONSTRAINT api_public_email_key UNIQUE (email)
)
```

Add Two Roles

```
CREATE ROLE NW_ADMIN;  # Full privileges (Create, Select, Update, Delete) on NorthWind database
GRANT ALL PRIVILEGES ON DATABASE northwind TO NW_ADMIN;

CREATE ROLE NW_PUBLIC;  # Select privilege on all original NorthWind tables
                        # No access to api_admin table
                        # Row level access on api_public table, only access to data related to their own user

GRANT SELECT ON DATABASE northwind TO NW_PUBLIC;
REVOKE SELECT ON api_admin FROM NW_PUBLIC;
```

Set Row Level Security

```
ALTER TABLE api_admin ENABLE ROW LEVEL SECURITY;
ALTER TABLE api_public ENABLE ROW LEVEL SECURITY;

CREATE POLYCY plcy_nw_admin ON api_admin TO NW_ADMIN USING (ture);
CREATE POLYCY plcy_nw_public ON api_public TO NW_PUBLIC FOR SELECT USING (username =
current_user);
```

Add API Users

```
INSERT INTO api_admin(username, password, email) VALUES
('admin_all','adminADMIN','admin@admin.com');
CREATE ROLE admin_all;
GRANT NW_ADMIN TO admin_all;

INSERT INTO api_public(username, password, email) VALUES
('public','publicPUBLIC','public@public.com');
CREATE ROLE public;
GRANT NW_PUBLIC TO public;
```

Encrypt Password with PGCRYPTO

There is an extension in PostgreSQL pgcrypto could be used to encrypt plain text into hash code.

```
CREAT EXTENSION pgcrypto;
UPDATE api_admin SET password=crypt(‘adminADMIN’,gen_salt(‘bf’)) WHERE username=’admin_all’;
UPDATE api_public SET password=crypt(‘publicPUBLIC’,gen_salt(‘bf’)) WHERE username=’public’;
```

Add Default No Access Role

```
CREATE ROLE NO_ACCESS_ROLE;
```

Create JWT Type

```
CREATE TYPE public.jwt as (
  role text,
  exp integer,
  user_id integer,
  username text
);
```

Create Authentication Function

```
create function public.authenticate(
  email text,
  password text
) returns public.jwt as $$
declare
  account public.api_admin;
begin
  select a.* into account
    from public.api_admin as a
    where a.email = authenticate.email;

  if account.password = crypt(password, account.password) then
    return (
      account.username,
      extract(epoch from now() + interval '7 days'),
      account.id,
      account.email
    )::public.jwt;
  else
    return null;
  end if;
end;
$$ language plpgsql strict security definer;

create function public.authenticate_public(
  email text,
  password text
) returns public.jwt as $$
declare
  account public.api_public;
begin
  select a.* into account
    from public.api_public as a
    where a.email = authenticate_public.email;

  if account.password = crypt(password, account.password) then
    return (
      account.username,
      extract(epoch from now() + interval '7 days'),
      account.id,
      account.email
    )::public.jwt;
  else
    return null;
  end if;
end;
$$ language plpgsql strict security definer;
```

The following query could test the functions:

```
SELECT authenticate(‘admin@admin.com’,’adminADMIN’);
SELECT authenticate_public(‘public@public.com’,’publicPUBLIC’);
```

They should return the JWT

Postgre SQL Commands ref: https://www.postgresql.org/docs/9.1/sql-commands.html

PostGraphile

Postgraphile runs default at Port 5000. In our case, it run at port 5000 in a docker container. The host port 5002 was mapped for connecting to postgraphile in the container.

Start Postgraphile in docker

```
docker run --name northwind_enhance -p 5002:5000 -d graphile/postgraphile --enhance-graphiql
--allow-explain –dynamic-json --connection
postgres://postgres:1234567890@172.17.0.1:5432/northwind --schema public --watch --jwt-
token-identifier public.jwt --jwt-secret graphql --default-role no_access_role
```

```
Recommanded Postgraphile Options
postgraphile \
  --subscriptions \
  --watch \
  --dynamic-json \
  --no-setof-functions-contain-nulls \
  --no-ignore-rbac \
  --show-error-stack=json \
  --extended-errors hint,detail,errcode \
  --append-plugins @graphile-contrib/pg-simplify-inflector \
  --export-schema-graphql schema.graphql \
  --graphiql "/" \
  --enhance-graphiql \
  --allow-explain \
  --enable-query-batching \
  --legacy-relations omit \
  --connection $DATABASE_URL \
  --schema app_public
```

Postgraphile CLI ref: https://www.graphile.org/postgraphile/usage-cli/

NorthWind GraphQL APIs were created:
GraphQL API Documentation http://5.22.220.255:5002/graphiql
GraphQL API http://5.22.220.255:5002/graphql
PostgreSQL Database host: http://5.22.220.255:5002/graphql

Request for JWT

Go to GraphQL API Documentation at http://5.22.220.255:5002/graphiql
Give the mutation:

```
mutation {
    authenticate(input: {email: "admin@admin.com", password: "adminADMIN"}) {
        jwt
    }
}
```

The response should be:

```
{
  "data": {
    "authenticate": {
      "jwt":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyb2xlIjoiYWRtaW5fYWxsIiwiZXhwIjoxNjQ2MzM5NTUxLCJ1c2
VyX2lkIjoxLCJ1c2VybmFtZSI6ImFkbWluQGFkbWluLmNvbSIsImlhdCI6MTY0NTczNDc1MCwiYXVkIjoicG9zdGdyYX
BoaWxlIiwiaXNzIjoicG9zdGdyYXBoaWxlIn0.QCLmcZUCxDkYOz_J7dFk3Rb726QFRs7bs3dQEroxPlI"
    }
  }
}
```

Using JWT as token

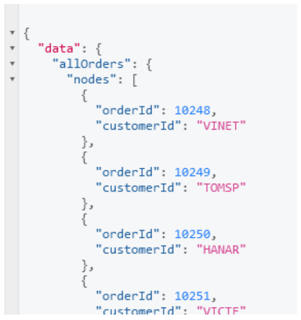
JWT could be used as an authorization key. The first step was to acquire the token. Then next was to send the JWT in the HEADER as authorization key.

1. Click REQUEST HEADERS at bottom of the window
2. Give the token as value next to “Authorization”:“bearer HERE_PUT_THE_JWT”
3. Query the exact data you want, example:

Query

```
query{
  allOrders{
    nodes{
      orderId
      customerId
    }
  }
}
```

Response



```
{
  "data": {
    "allOrders": {
      "nodes": [
        {
          "orderId": 10248,
          "customerId": "VINET"
        },
        {
          "orderId": 10249,
          "customerId": "TOMSP"
        },
        {
          "orderId": 10250,
          "customerId": "HANAR"
        },
        {
          "orderId": 10251,
          "customerId": "VICTE"
```

NGINX with SSL as Reverse Proxy

Install NGINX and OpenSSL

```
sudo apt update
sudo apt install nginx openssl
```

Create Certificate (self-signed)

Self-signed certificate was created in the project due to the cost of official signed certificate. The browsers would show warnings about self-signed SSL certificate.

```
cd /etc/nginx
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/ssl/private/nginx-
selfsigned.key -out /etc/ssl/certs/nginx-selfsigned.crt
```

openssl: This is the basic command line tool for creating and managing OpenSSL certificates, keys, and other files.
req: This subcommand specifies that we want to use X.509 certificate signing request (CSR) management. The “X.509” is a public key infrastructure standard that SSL and TLS adheres to for its key and certificate management. We want to create a new X.509 cert, so we are using this subcommand.
-x509: This further modifies the previous subcommand by telling the utility that we want to make a self-signed certificate instead of generating a certificate signing request, as would normally happen.
-nodes: This tells OpenSSL to skip the option to secure our certificate with a passphrase. We need Nginx to be able to read the file, without user intervention, when the server starts up. A passphrase would prevent this from happening because we would have to enter it after every restart.
-days 365: This option sets the length of time that the certificate will be considered valid. We set it for one year here.

-newkey rsa:2048: This specifies that we want to generate a new certificate and a new key at the same time. We did not create the key that is required to sign the certificate in a previous step, so we need to create it along with the certificate. The rsa:2048 portion tells it to make an RSA key that is 2048 bits long.
-keyout: This line tells OpenSSL where to place the generated private key file that we are creating.
-out: This tells OpenSSL where to place the certificate that we are creating.

Output
```
Country Name (2 letter code) [AU]:FI
State or Province Name (full name) [Some-State]:POHJANMAA
Locality Name (eg, city) []:VAASA
Organization Name (eg, company) [Internet Widgits Pty Ltd]:VAMK OY
Organizational Unit Name (eg, section) []:IT-2020
Common Name (e.g. server FQDN or YOUR name) []:5.22.220.255
Email Address []:e2101066@edu.vamk.fi
```

Configure NGINX as HTTPS Reverse Proxy Server

Remove default Configuration
```
cd /etc/nginx/sites-enabled/
# ls
default
# sudo rm default
```

Create a Configuration Snippet Pointing to SSL Key and Certificate
```
sudo nano /etc/nginx/snippets/self-signed.conf
```

Add following codes:

```
ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;
```

Create a Configuration Snippet with Strong Encryption Settings
```
sudo nano /etc/nginx/snippets/ssl-params.conf
```

ADD following codes:

```
# from https://cipherli.st/
# and https://raymii.org/s/tutorials/Strong_SSL_Security_On_nginx.html

ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
ssl_prefer_server_ciphers on;
ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
ssl_ecdh_curve secp384r1;
ssl_session_cache shared:SSL:10m;
ssl_session_tickets off;
ssl_stapling on;
ssl_stapling_verify on;
resolver 8.8.8.8 8.8.4.4 valid=300s;
resolver_timeout 5s;
# Disable preloading HSTS for now.  You can use the commented out header line that
includes
# the "preload" directive if you understand the implications.
#add_header Strict-Transport-Security "max-age=63072000; includeSubdomains;
preload";
add_header Strict-Transport-Security "max-age=63072000; includeSubdomains";
add_header X-Frame-Options DENY;
add_header X-Content-Type-Options nosniff;

ssl_dhparam /etc/ssl/certs/dhparam.pem;
```

Adding Reverse Proxy Configuration

```
nano /etc/nginx/sites-availabe/reverse_proxy.conf

Add Following Codes:

server {
        listen 8443 ssl http2;
        listen [::]:8443 ssl http2;
        server_name 5.22.220.255;
        return 301 https://$server_name$request_uri;

        access_log /var/log/nginx/reverse-access.log;
        error_log /var/log/nginx/reverse-error.log;

        location / {
        proxy_pass http://5.22.220.255:5002/graphql;
        }

        include snippets/self-signed.conf;
        include snippets/ssl-params.conf;
}
```

Configure the Firewall – UFW

Ubuntu Server 20.04LTS come with default firewall, UFW, Uncomplicated Firewall. However, it was disabled by default.

```
sudo ufw enable
sudo ufw app list
sudo ufw allow ‘Nginx Full’
sudo ufw allow ‘OpenSSH’
sudo ufw allow ‘Nginx Full’
sudo ufw allow 8432
sudo ufw allow 80
sudo ufw allow 8443
sudo ufw allow 5002
```

Enable the Changes in Nginx

```
sudo nginx -t
```

Output

```
nginx: [warn] "ssl_stapling" ignored, issuer certificate not found
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful

sudo systemctl restart nginx
```

Testing by going to https://5.22.220.225:8443. This should be redirect to http://5.22.220.225:5002/graphql.

Documentation

OpenAPI/Swagger

OpenAPI/Swagger is a specification meant for REST APIs. It is a pain in the ass to implement it with GraphQL APIs. One could perhaps manually write all the specs with OpenAPI/Swagger (if I knew GraphQL, Apolo Server, REST, OpenAPI/Swagger better, I might be able to find another solution). Therefore after 10s of hours research, I decided not to manually write all the specal with OpenAPI/Swagger.

GraphDoc (to rescure)

GraphDoc is a free and open source tool for documentaion of GraphQL APIs. It generates the documents in html automatically.

Install npm

```
sudo apt npm
```

Install GraphDoc

```
npm install -g @2fd/graphdoc
```

Generate Documentation for GraphQL API

```
graphdoc -e http://5.22.220.255/graphql -o /var/www/nwGraphDoc/schema
```

Host the Document with Nginx

```
nano /etc/nginx/sites-available/nwGraphDoc.conf
```

Add following codes

```
server{
        listen 5004 http2;
        listen [::]:5004 http2;

        listen 8444 ssl http2;
        listen [::]:8444 ssl http2;
        server_name 5.22.220.255;
        root /var/www/nwGraphDoc/schema;
        index query.doc.html;

        access_log /var/log/nginx/nwGraphDoc-access.log;
        error_log /var/log/nginx/nwGraphDoc-error.log;

        include snippets/self-signed.conf;
        include snippets/ssl-params.conf;
}
```

Test the nginx configuration file

```
nginx -t
```

OUTPUT

```
nginx: [warn] "ssl_stapling" ignored, issuer certificate not found for certificate
"/etc/ssl/certs/nginx-selfsigned.crt"
```

This is ok, because of we had self-signed SSL Certificate. The information below should be shown if configuration file is otherwise ok.

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

Configure Firewall

```
ufw allow 5004
ufw allow 8444
```

Restart Nginx For New Webserver

```
sudo systemctl restart nginx
```

Now our NorthWind GraphQL API Documentation is hosted at:
https://5.22.220.255:8444/

Conclusion

GraphQL API could be implemented with extrem ease. Once learned the procedure, one may setup the database server, API and documentation in one day. With GrpahQL API client could request exact needed data, no more no less. Hence it is very efficient in terms of network. GrpahQL api could be customised. And it is strongly typed. Which make it a versatile and less prone to bugs.
However, GraphQL API is very differnet from REST API. The frontend developers need to learn the syntax of GraphQL queries and a little bit graph theory to better understand the schema of GraphQL APIs.