Load balancing with Nginx

Introduction

In modern web applications, scalability and reliability are very essential in handling growing traffic. **Load balancing** is the perfect way to do this, ensuring high availability and reliability by distributing incoming network traffic across multiple servers. **Nginx**, a reverse proxy server and load balancer, is widely used due to its efficiency and flexibility.

Why use Nginx?

It is an open-source software and very efficient HTTP load balancer, known for its high performance, low resource usage and stability. It supports multiple load balancing methods, being able to handle many connections simultaneously and can be easily configured.

How does Nginx work?

Nginx creates a single entry point for a distributed web application that works on multiple servers.

This tutorial will guide you on how to implement **load balancing with nginx**, using practical examples on a **Health Tracker App (soa-health-tracker)**. The backend is structured in the following microservices:

- auth_service: handles authentication
- metrics_service: tracks user metrics(e.g. steps, water consumption, sleep)

Prerequisites for macOS

- Sudo or root access to your server
- Brew installed
- Multiple backend servers or microservices to distribute traffic
- Docker (for running microservices)
- Docker Compose (for service orchestration)

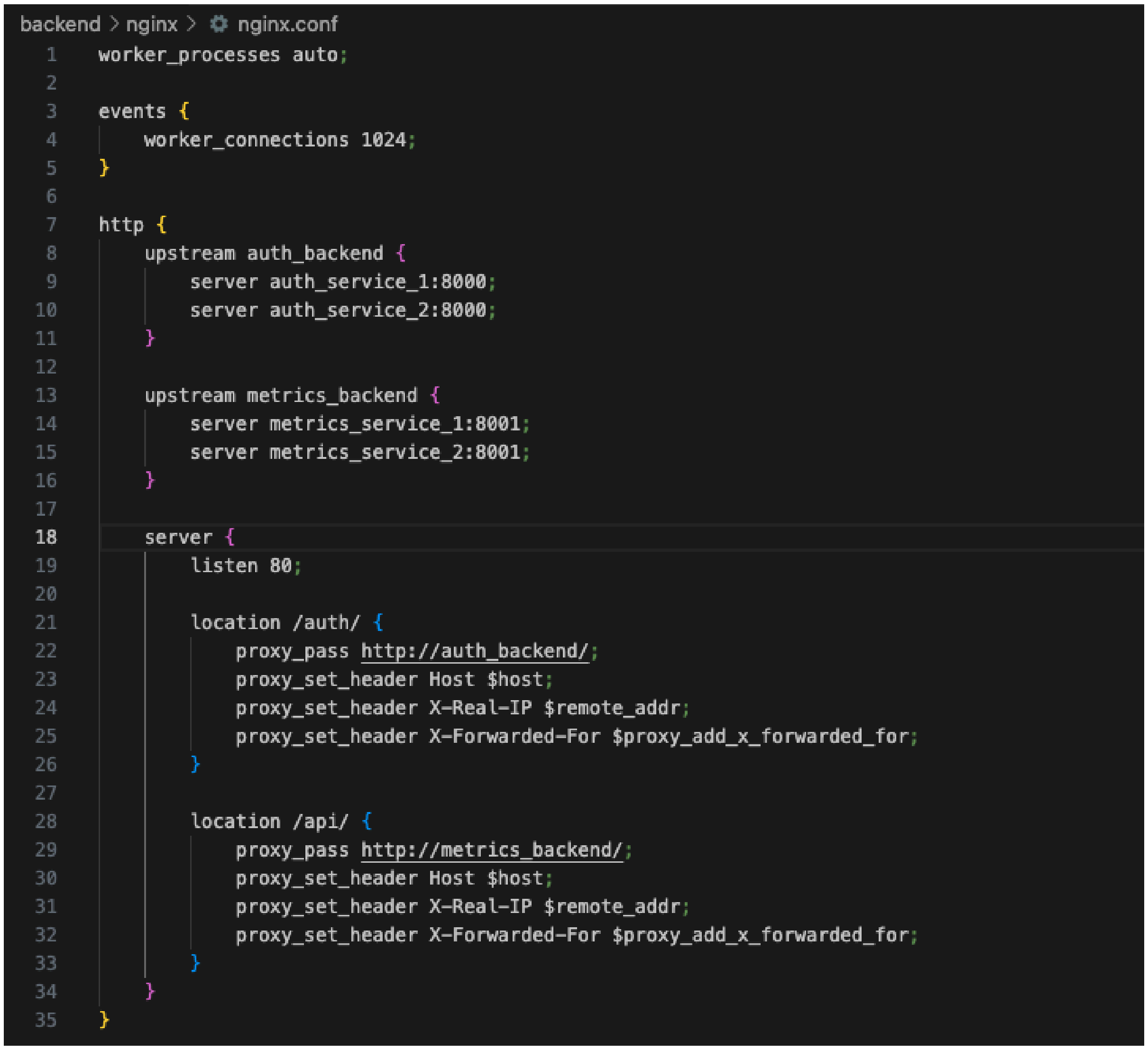
Setting Up Nginx as a Load Balancer

1. Install and start nginx

```
brew install nginx
brew services start nginx
```

2. Create the **Nginx** Configuration File

Inside your project root, create a folder for Nginx and add the following configuration file (`nginx.conf`):



backend > nginx > nginx.conf

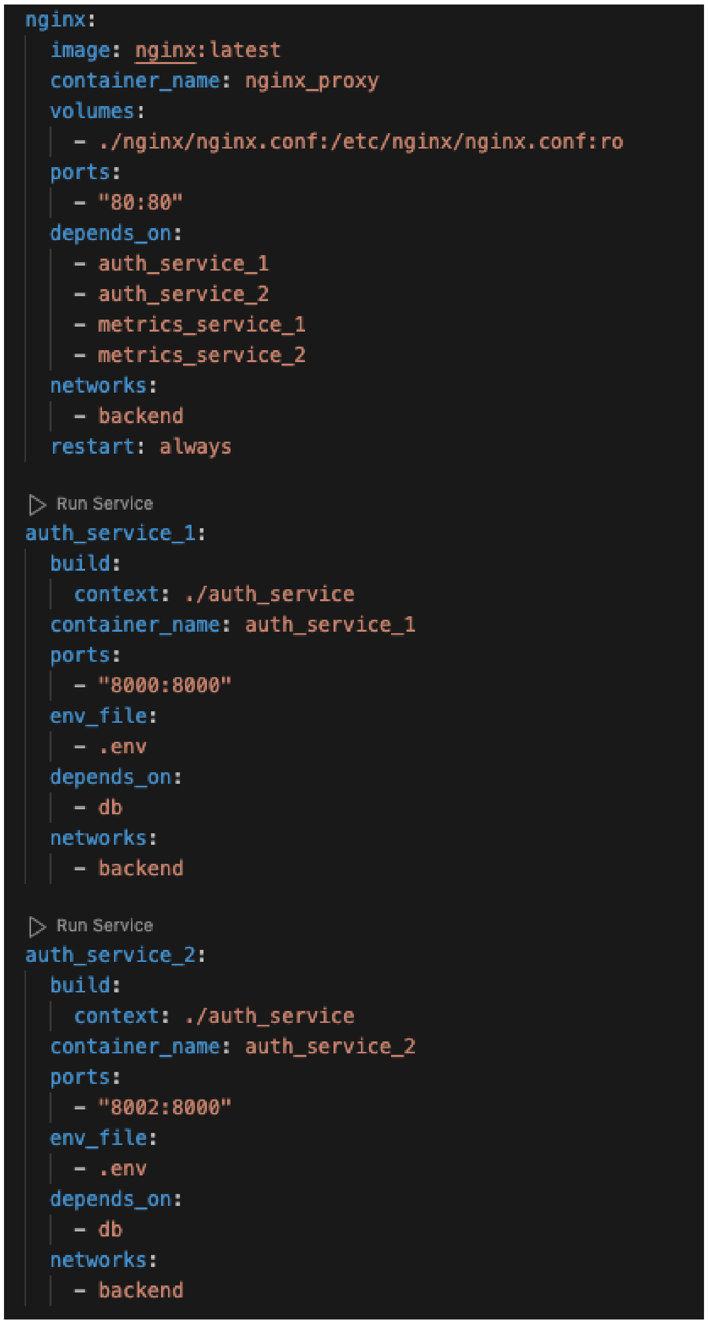
```
worker_processes auto;

events {
    worker_connections 1024;
}

http {
    upstream auth_backend {
        server auth_service_1:8000;
        server auth_service_2:8000;
    }

    upstream metrics_backend {
        server metrics_service_1:8001;
        server metrics_service_2:8001;
    }

    server {
        listen 80;

        location /auth/ {
            proxy_pass http://auth_backend/;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }

        location /api/ {
            proxy_pass http://metrics_backend/;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
}
```

Explanation:

- `upstream auth_backend` defines a pool of servers for the authentication service.
- `upstream metrics_backend` defines a pool for the metrics service.
- The `server` block listens on port **80** and routes requests:
  - `/auth/` Load-balanced across `auth_service_1` and `auth_service_2`
  - `/metrics/` Load-balanced across `metrics_service_1` and `metrics_service_2`

3. Modify docker-compose.yml

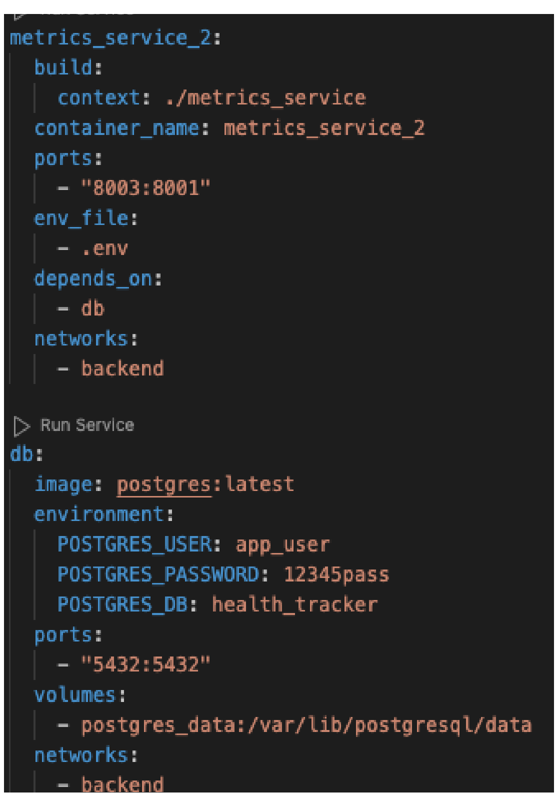
Update your Docker Compose file to include multiple instances of each service and the Nginx container:



```
nginx:
  image: nginx:latest
  container_name: nginx_proxy
  volumes:
    - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
  ports:
    - "80:80"
  depends_on:
    - auth_service_1
    - auth_service_2
    - metrics_service_1
    - metrics_service_2
  networks:
    - backend
  restart: always

Run Service
auth_service_1:
  build:
    context: ./auth_service
  container_name: auth_service_1
  ports:
    - "8000:8000"
  env_file:
    - .env
  depends_on:
    - db
  networks:
    - backend

Run Service
auth_service_2:
  build:
    context: ./auth_service
  container_name: auth_service_2
  ports:
    - "8002:8000"
  env_file:
    - .env
  depends_on:
    - db
  networks:
    - backend
```

Explanation:

- We added **2 instances** of each microservice, running on different ports
- The **nginx container** is included and mounts the `nginx.conf` file



```
metrics_service_2:
  build:
    context: ./metrics_service
  container_name: metrics_service_2
  ports:
    - "8003:8001"
  env_file:
    - .env
  depends_on:
    - db
  networks:
    - backend

Run Service
db:
  image: postgres:latest
  environment:
    POSTGRES_USER: app_user
    POSTGRES_PASSWORD: 12345pass
    POSTGRES_DB: health_tracker
  ports:
    - "5432:5432"
  volumes:
    - postgres_data:/var/lib/postgresql/data
  networks:
    - backend
```

4. Run the Load-Balanced System

Now, start the application using Docker:

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

This will spin up **two instances of each microservice** and launch **Nginx**, which will balance traffic.

5. Testing

Using a web browser, open `http://localhost/auth/` and refresh multiple times. Do the same for `http://localhost/metrics/`. You should see that the responses come from different instances.
If you want to easily monitor the requests, you can log the instance names in the `main.py` file in `auth_service` and `metrics_service.`

6. Conclusion

Setting up Nginx as a load balancer can significantly improve the performance, scalability and reliability of your web apps. Due to its high performance and easy configuration, Nginx is the perfect choice to manage and distribute traffic across your servers.