Intro to NGINX

Nginx is **an open-source web server that functions as a web server, reverse proxy, load balancer, mail proxy, and HTTP cache**. It was created in 2004 by Igor Sysoev and is known for its high performance, scalability, and efficiency. [1, 2, 3]

Key functions and uses

- **Web server:** Serves static content from a defined root directory, handling requests for web pages and files. [4]
- **Reverse proxy:** Sits in front of other servers, forwarding client requests to the appropriate server. This can improve security by hiding the internal servers' IPs and provide features like SSL encryption and decryption. [5, 6, 7, 8, 9, 10]
- **Load balancer:** Distributes incoming network traffic across multiple backend servers to optimize resource utilization and prevent any single server from becoming a bottleneck. [11, 12]
- **Content cache:** Stores copies of static content, like images or HTML files, to serve subsequent requests faster without having to fetch the content from the origin server each time. [9]
- **Mail proxy:** Can act as a mail proxy server for protocols like Post Office Protocol 3 (POP3), Internet Message Access Protocol (IMAP), and Simple Mail Transfer Protocol (SMTP). [1, 13, 14, 15, 16]
- **API gateway:** In commercial versions like NGINX Plus, it can serve as an API gateway, providing features for managing and securing APIs. [17]

Technical architecture

- **Master and worker processes:** Nginx operates with a master process that manages several worker processes. [18]
- **Worker processes:** The worker processes handle the actual client requests, with the number of workers typically set to match the number of CPU cores for optimal performance. [18]
- **Asynchronous, event-driven architecture:** Nginx uses an asynchronous, event-driven approach to handle many connections concurrently with minimal memory usage. [3, 18, 19, 20]

Configuration and usage

- **Configuration file:** Nginx is configured using text files, where you define server blocks, locations, and other directives.
- **Start and reload commands:** You can start Nginx with sudo nginx and reload the configuration file without a full restart by using sudo nginx -s reload. [4]

AI responses may include mistakes.

[1] https://nginx.org/
[2] https://www.f5.com/glossary/nginx
[3] https://www.pal.tech/technology/deploying-applications-with-nginx/
[4] https://www.youtube.com/watch?v=7FpSPSlJj-0
[5] https://gridpane.com/kb/configure-nginx/
[6] https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10/html/deploying_web_servers_and_reverse_proxies/setting-up-and-configuring-nginx
[7] https://www.infoq.com/news/2020/02/nginx-unit-reverse-proxying/
[8] https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/
[9] https://www.youtube.com/watch?v=iInUBOVeBCc
[10] https://www.youtube.com/watch?v=9t9Mp0BGnyI

[11] https://www.youtube.com/watch?v=OEFZUj_RQKc
[12] https://nginx.org/en/docs/http/load_balancing.html
[13] https://en.wikipedia.org/wiki/Nginx
[14] https://www.udemy.com/course/nginxforbeginners/
[15] https://www.bigrock.in/blog/products/hosting/vps-dedicated-servers/nginx-guide
[16] https://www.combell.com/en/technology/nginx
[17] https://docs.nginx.com/
[18] https://github.com/nginx/nginx
[19] https://www.bigrock.in/blog/products/hosting/vps-dedicated-servers/nginx-guide
[20] https://www.combell.com/en/technology/nginx

Reverse-proxy

A **reverse proxy** is a server that sits **in front of your backend servers** and handles requests **on their behalf**. It's the opposite of a forward proxy (which represents the client).

? What a Reverse Proxy Does

When a client (browser, app, etc.) sends a request to your website:

1. The request goes **to the reverse proxy**, not directly to your application servers.
2. The reverse proxy decides **where to forward the request** (maybe server A, B, or C).
3. It receives the response from the backend server.
4. It sends the response back to the client.

? Why Use a Reverse Proxy?

Reverse proxies are very popular because they add powerful features:

1. Load Balancing

Distribute traffic across multiple backend servers to prevent overload.
(e.g., Nginx, HAProxy)

2. Security & Anonymity

Backend servers stay hidden. Clients never know their IP addresses.

3. SSL/TLS Termination

Proxy handles HTTPS decryption so backend servers can run plain HTTP.

4. Caching

Stores static content to improve performance and reduce backend load.

5. Rate Limiting / WAF

Protects backend from attacks by filtering bad requests.

? Real-Life Example

If you use **Nginx** as a reverse proxy in front of a Node.js app:

```
Client → Nginx (reverse proxy) → Node.js server
```

″ Reverse Proxy vs Forward Proxy

Feature	Reverse Proxy	Forward Proxy
Acts on behalf of	Server	Client

Feature	Reverse Proxy	Forward Proxy
Hides	Backend servers	Client identity
Typical use	Load balancing, SSL, security	Browsing anonymously, filtering

---

If you want, I can also show:

- How to configure a reverse proxy using **Nginx**
- Reverse proxy in **Kubernetes (Ingress)**
- Reverse proxy vs API Gateway

Just tell me!

Here's a clean and practical **Nginx reverse proxy setup** for deploying a simple website (e.g., a Node.js app, Python app, or another web server running on port 3000 or 8080).

Ĺ 1. Install Nginx

Ubuntu

```
sudo apt update
sudo apt install nginx -y
```

CentOS / Rocky / Alma

```
sudo yum install nginx -y
sudo systemctl enable --now nginx
```

Ĺ 2. Basic Reverse Proxy Configuration

Assume your backend app runs on:

- **http://localhost:3000**

Create a new Nginx site config:

/etc/nginx/sites-available/mywebsite

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

    location / {
        proxy_pass http://localhost:3000;
```

```
        # Recommended reverse-proxy headers
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Enable the config:

```
sudo ln -s /etc/nginx/sites-available/mywebsite /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx
```

Now Nginx is serving your backend app.

Ĺ 3. Your Architecture Now Looks Like

```
Client → Nginx (port 80/443) → Backend app (localhost:3000)
```

Ĺ Optional: HTTPS Using Certbot (Free Let's Encrypt SSL)

Install Certbot:

```
sudo apt install certbot python3-certbot-nginx -y
```

Get a certificate:

```
sudo certbot --nginx -d example.com -d www.example.com
```

Automatically configures Nginx for HTTPS.

? 4. Reverse Proxy for a Static HTML Website

If instead you have:

- HTML/CSS/JS in `/var/www/mywebsite`

Use this Nginx config:

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

    root /var/www/mywebsite;
    index index.html;
```

```
    location / {
        try_files $uri $uri/ =404;
    }
}
```

└ Common Enhancements (Optional)

Load balancing multiple backend servers

```
upstream backend_servers {
    server 127.0.0.1:3000;
    server 127.0.0.1:3001;
    server 127.0.0.1:3002;
}

server {
    listen 80;
    server_name example.com;

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

Caching static files

```
location ~* \.(jpg|jpeg|png|gif|css|js)$ {
    expires 30d;
    access_log off;
}
```

Nginx as a reverse proxy in Docker

It is to forward traffic to a backend container (Node, Python, Java, etc.).

Ĺ 1. Directory Structure

Create a project folder:

```
myproxy/
 ├── docker-compose.yml
 └── nginx.conf
```

Ĺ 2. nginx.conf (Reverse Proxy Config)

Create **nginx.conf**:

```
events {}

http {
    server {
        listen 80;

        location / {
            proxy_pass http://backend:3000;

            # Important headers
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }
}
```

Here, `backend` is the Docker service name (DNS works automatically).

Ĺ 3. docker-compose.yml

Create **docker-compose.yml**:

```
version: "3.8"
```

```
services:
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - backend

  backend:
    image: node:18
    working_dir: /app
    volumes:
      - ./app:/app
    command: ["node", "server.js"]
```

Ĺ 4. Example Backend App (Node.js)

Create a directory:

```
myproxy/app/server.js
```

Put a simple server:

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.end("Hello from backend via Nginx reverse proxy!");
});

server.listen(3000, () => console.log("Backend running on port 3000"));
```

Ĺ 5. Start Everything

From inside the project folder:

```
docker compose up -d
```

Now open:

```
http://localhost
```

You should see the backend response via Nginx.

? Optional Upgrades

A. Add HTTPS (Let's Encrypt) with Certbot

Use the `linuxserver/letsencrypt` image **or** pair Nginx with Traefik or Caddy.

B. Run Nginx with a Custom Dockerfile

If you need SSL certificates inside container:

```
FROM nginx:latest
COPY nginx.conf /etc/nginx/nginx.conf
```

C. Reverse Proxy to Multiple Backends (Load Balancing)

```
upstream app_servers {
    server backend1:3000;
    server backend2:3000;
}

server {
    listen 80;
    location / {
        proxy_pass http://app_servers;
    }
}
```

D. Reverse Proxy for Dockerized Static Website

```
nginx:
  image: nginx
  ports:
    - "80:80"
  volumes:
    - ./static:/usr/share/nginx/html
```

Nginx as a reverse proxy in Kubernetes via an **Ingress**.

1. **Prerequisites**
    1. Kubernetes cluster (Minikube, Kind, EKS, GKE, AKS, etc.)
    2. `kubectl` configured
    3. **Nginx Ingress Controller** installed

Install Nginx Ingress Controller (if not already):

Using Helm:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
helm install nginx-ingress ingress-nginx/ingress-nginx
```

Or with kubectl (YAML):

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
nginx/controller-v1.14.1/deploy/static/provider/cloud/deploy.yaml
```

Check controller is running:

```
kubectl get pods -n ingress-nginx
```

Ĺ 2. Deploy a Sample Backend Service

Example: a simple Nginx backend (can be Node, Python, etc.)

backend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-backend
  template:
    metadata:
      labels:
        app: my-backend
    spec:
      containers:
      - name: my-backend
        image: nginx:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: my-backend
spec:
  selector:
    app: my-backend
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Apply it:

```
kubectl apply -f backend-deployment.yaml
```

Ĺ 3. Create an Ingress Resource

This tells the **Nginx Ingress Controller** how to route traffic.

my-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: example.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-backend
            port:
              number: 80
```

Apply it:

```
kubectl apply -f my-ingress.yaml
```

Ĺ 4. Access the Service

1. Edit `/etc/hosts` (for local test):

```
<INGRESS_CONTROLLER_IP> example.local
```

Get IP:

```
kubectl get svc -n ingress-nginx
```

2. Open in browser:

```
http://example.local
```

Nginx Ingress will forward traffic to the backend service.

Ĺ 5. Optional Enhancements

A. HTTPS with Cert-Manager

- Install **cert-manager**
- Annotate Ingress for Let's Encrypt certificate

```
metadata:
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
  - hosts:
    - example.local
    secretName: example-tls
```

B. Path-based Routing

```
rules:
- host: example.local
  http:
    paths:
    - path: /api
      pathType: Prefix
      backend:
        service:
          name: api-service
          port:
            number: 80
    - path: /web
      pathType: Prefix
      backend:
        service:
          name: web-service
          port:
            number: 80
```

C. Load Balancing & Sticky Sessions

- Nginx Ingress supports session affinity:

```
nginx.ingress.kubernetes.io/affinity: "cookie"
```

```
nginx.ingress.kubernetes.io/session-cookie-name: "route"
```

---

1. Project Structure

```
k8s-nginx-node/
 ├── backend/
 │    ├── server.js
 │    └── package.json
 ├── k8s/
 │    ├── backend-deployment.yaml
 │    ├── ingress.yaml
 └── README.md
```

---

2. Node.js Backend

backend/package.json

```
{
  "name": "backend",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

backend/server.js

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello from Node.js backend via Nginx Ingress!');
});

app.get('/api', (req, res) => {
  res.json({ message: 'API endpoint reached!' });
});

app.listen(port, () => {
  console.log(`Backend listening on port ${port}`);
});
```

---

3. Dockerfile for Backend

Create **backend/Dockerfile**

```
FROM node:18-alpine

WORKDIR /app
COPY package.json package-lock.json* ./
RUN npm install
COPY . .

EXPOSE 3000
CMD ["npm", "start"]
```

Build and test locally:

```
docker build -t node-backend:1.0 ./backend
docker run -p 3000:3000 node-backend:1.0
```

---

4. Kubernetes Deployment & Service

k8s/backend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: node-backend
  template:
    metadata:
      labels:
        app: node-backend
    spec:
      containers:
      - name: node-backend
        image: node-backend:1.0
        ports:
        - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: node-backend
spec:
  selector:
    app: node-backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

 Note: If using local cluster (Minikube/Kind), you may need to**load Docker images into the cluster** or push to a registry.

---

5. Nginx Ingress Resource

k8s/ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: node-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: node.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: node-backend
            port:
              number: 80
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: node-backend
            port:
              number: 80
```

---

6. Apply Kubernetes Resources

```
kubectl apply -f k8s/backend-deployment.yaml
kubectl apply -f k8s/ingress.yaml
```

Check pods, services, ingress:

```
kubectl get pods
kubectl get svc
kubectl get ingress
```

---

7. Local Testing

If using **Minikube**, get ingress IP:

```
minikube ip
```

Add entry to `/etc/hosts`:

```
<MINIKUBE_IP> node.local
```

Then open:

```
http://node.local/      → Hello page
http://node.local/api   → JSON API endpoint
```

---

8. Optional Enhancements

HTTPS with Cert-Manager

```
metadata:
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
    - hosts:
        - node.local
      secretName: node-tls
```

Path-Based Routing to Multiple Services

```
rules:
- host: node.local
  http:
    paths:
    - path: /api
      pathType: Prefix
      backend:
        service:
          name: api-service
          port:
            number: 80
    - path: /
      pathType: Prefix
      backend:
        service:
          name: web-service
          port:
            number: 80
```

This setup is **production-ready**, scalable, and uses **Nginx Ingress as a reverse proxy** in front of a Node.js backend.

Docker + Kubernetes combined workflow

Perfect! Let's make a complete workflow that builds your Node.js Docker image and deploys it to

Kubernetes automatically using a `Makefile`. This is ideal for local Minikube, Kind, or even cloud clusters.

---

1. Project Structure

```
k8s-nginx-node/
├── backend/
│   ├── server.js
│   ├── package.json
│   └── Dockerfile
├── k8s/
│   ├── backend-deployment.yaml
│   └── ingress.yaml
└── Makefile
```

---

2. Dockerfile (in `backend/`)

```
FROM node:18-alpine

WORKDIR /app
COPY package.json package-lock.json* ./
RUN npm install
COPY . .

EXPOSE 3000
CMD ["npm", "start"]
```

---

3. Kubernetes Deployment & Service (`k8s/backend-deployment.yaml`)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: node-backend
  template:
    metadata:
      labels:
        app: node-backend
    spec:
      containers:
```

```
      - name: node-backend
        image: node-backend:latest
        ports:
        - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: node-backend
spec:
  selector:
    app: node-backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

---

4. Nginx Ingress (`k8s/ingress.yaml`)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: node-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: node.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: node-backend
            port:
              number: 80
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: node-backend
            port:
              number: 80
```

---

5. Makefile (Automatic Build & Deploy)

```
# Makefile for building Node.js Docker image and deploying to Kubernetes

IMAGE_NAME=node-backend
```

```
IMAGE_TAG=latest
K8S_DIR=k8s

# Use Minikube docker daemon (optional)
USE_MINIKUBE ?= true
ifeq ($(USE_MINIKUBE),true)
    DOCKER_ENV := $(shell minikube docker-env --shell bash)
    export $(DOCKER_ENV)
endif

.PHONY: all build push deploy ingress clean

all: build deploy ingress

# Build Docker image
build:
        docker build -t $(IMAGE_NAME):$(IMAGE_TAG) ./backend

# Push image to registry (optional)
push:
        docker push $(IMAGE_NAME):$(IMAGE_TAG)

# Deploy Kubernetes resources
deploy:
        kubectl apply -f $(K8S_DIR)/backend-deployment.yaml

# Apply Ingress
ingress:
        kubectl apply -f $(K8S_DIR)/ingress.yaml

# Delete all resources
clean:
        kubectl delete -f $(K8S_DIR)/ingress.yaml || true
        kubectl delete -f $(K8S_DIR)/backend-deployment.yaml || true
        docker rmi $(IMAGE_NAME):$(IMAGE_TAG) || true
```

---

6. Usage

Step 1: Build & Deploy

```
make
```

This will:

1. Build the Docker image
2. Deploy the backend deployment & service
3. Deploy the Nginx ingress

Step 2: Check Status

```
kubectl get pods
kubectl get svc
```

```
kubectl get ingress
```

Step 3: Local Testing

If using Minikube:

```
minikube ip
```

Add to `/etc/hosts`:

```
<MINIKUBE_IP> node.local
```

Open in browser:

```
http://node.local/
http://node.local/api
```

Step 4: Clean Up

```
make clean
```

---

7. Optional Enhancements

- HTTPS via cert-manager
- Path-based routing to multiple services
- Auto-increment image tags in CI/CD
- Helm chart integration
