Docker

From “It Works on My Machine” to “It Works Everywhere”

The Problem: Development Environment Hell

```
Engineer A:
  "My Node.js app runs perfectly!"
Engineer B:
  "I get errors when I run your code..."
Client A:
  "It works on macOS but not Windows..."
Client B:
  "My MongoDB version is different..."
Server Manager:
  "It fails behind Nginx..."
```

What's Really Happening

Layer	Varies by machine?	Impact
OS/Kernel	Yes	Path, permissions, sockets
Runtimes	Yes	Node 16 vs 20 behavior
Dependencies	Yes	Native modules, ABI, builds
Services	Yes	Mongo/Nginx versions & config

Tiny differences → **non-reproducible** environments.

Docker — What It Is (Precisely)

- Image: Snapshot (recipe) of an environment (OS libs + runtime + your app)
- Container: A **running** instance of that image (isolated process + filesystem)
- Compose: Orchestrates multiple containers (e.g., web + db + nginx)

Same image ⇒ same bits ⇒ same behavior on **any** host with Docker.

Why Docker Solves It

- Pin versions (Node 20, Mongo 7, Nginx stable) inside images
- Bundle dependencies in the image layer (repeatable builds)
- Isolate each service (no host conflicts/ports/libs)
- Development == Operation parity (run the *same* containers everywhere)

Docker for Software Engineers

1. Instant Setup

```
# Without Docker
brew install node mongodb nginx

# With Docker
docker compose up
```

2. Perfect Consistency

- Same Node version
- Same DB config
- Same proxy rules/configurations

3. Easy Cleanup

```
docker compose down
```

4. Fast Onboarding

❌ 2 days setup → ✅ 10 minutes

5. Reliable Deployment

Same environment locally & in production.

6. Scalable

Run multiple containers per server via Nginx load balancing.

7. Cost-Efficient

One VPS runs many apps.

Run The previous Node.js app and NGINX using Docker

Three Steps

1. Make app/index.js Node.js application a container
2. Make Nginx reverse proxy sever a container.
3. Run these containers

```
docker
├── app
│   └── Dockerfile
├── nginx
│   ├── Dockerfile
│   └── nginx.conf
└── docker-compose.yml
```

Make app/index.js a container

We make a Dockerfile for the Node.js app: it is an instruction about how to build the isolated environment.

```
FROM node:18-alpine
# Set working directory
WORKDIR /app
# Copy package files
COPY app/package*.json ./
# Install dependencies
RUN npm ci --only=production
# Copy application code
COPY app/ ./
# Expose port 3000
EXPOSE 3000
# Start the application
CMD ["npm", "start"]
```

docker-compose.yml

docker-compose.yml is used to orchestrate multiple docker containers.

```
  app:
    build:
      context: ..
      dockerfile: docker/app/Dockerfile
    container_name: nodejs_app
    restart: unless-stopped
    environment:
      - NODE_ENV=production
      - PORT=3000
    expose:
      - "3000"
    networks:
      - app-network
```

The context means the base directory, so dockerfile is located in `../docker/app/Dockerfile`.

Make Nginx a container

The Dockerfile for Nginx server is simple because it copies only the nginx.conf and expose two ports.

```
FROM nginx:alpine
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80 443
```

- The host computer uses 8080, but it maps to the cotainer port 80.

```
    ports:
      - "8080:80"  # Host port 8080 → Container port 80
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf:ro
```

The context (base directory) is `./nginx`, so the dockerfile is located in `./nginx/Dockerfile`.

```
  nginx:
    build:
      context: ./nginx
      dockerfile: Dockerfile
    container_name: nginx_proxy
    restart: unless-stopped
    ports:
      - "8080:80"  # Host port 8080 → Container port 80
    depends_on:
      - app
    networks:
      - app-network
```

This is nginx.conf file: it accesses the node.js app at port 3000, and it listens to port 80.

```
# NGINX Configuration for Docker
# Uses container name instead of localhost

upstream nodejs_app {
    # Use Docker service name
    server app:3000;
    keepalive 8;
}

server {
    listen 80;
    server_name localhost;

    location / {
        proxy_pass http://nodejs_app;
        proxy_http_version 1.1;

        # Pass 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;
    }
}
```

Orchestrate two containers

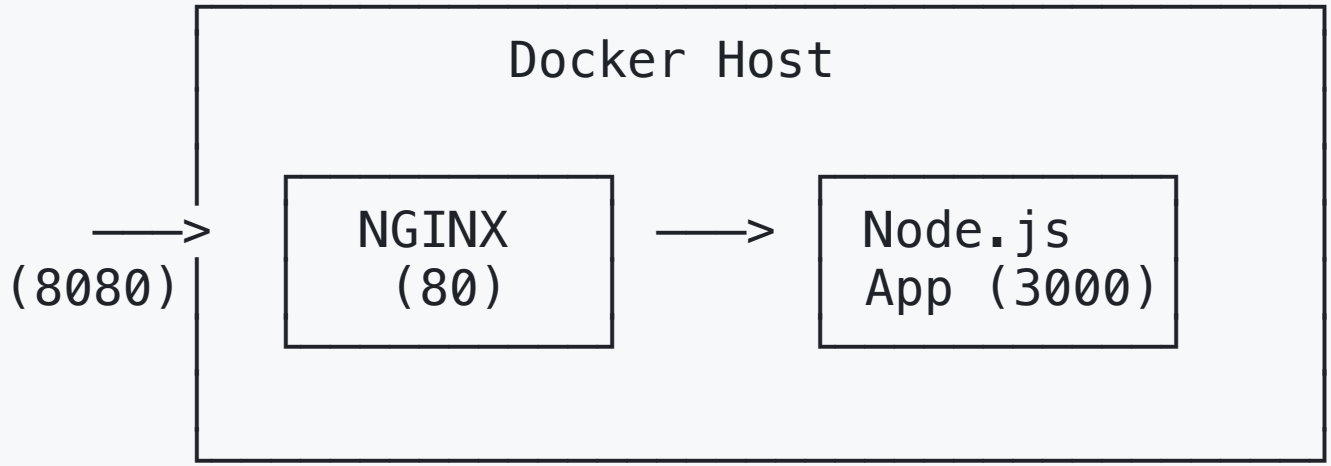
```yaml
version: '3.8'

services:
  # Node.js Application
  app:
    ...

  # NGINX Reverse Proxy
  nginx:
    ...

networks:
  app-network:
    driver: bridge
```

This diagram shows how the ports are mapped from the host (8080) to the Ngnix container (80) and the node.js (3000).



Run the Docker

Make sure Docker Desktop (or command line tools) are installed and running.

Build the docker containers

```
docker-compose build

OR to build from scratch

docker-compose build --no-cache
```

Run & Shutdown docker containers

```
docker> docker-compose up -d
[+] Building 0.0s (0/0)                          docker:desktop-linux
[+] Running 3/3
 ✔ Network docker_app-network  Created                          0.0s
 ✔ Container nodejs_app        Started                          0.0s
 ✔ Container nginx_proxy       Started                          0.0s

docker> docker-compose down
[+] Running 3/2
 ✔ Container nginx_proxy       Removed                          0.1s
 ✔ Container nodejs_app        Removed                          0.6s
 ✔ Network docker_app-network  Removed                          0.0s
```

Then, we can access the Docker server with `http://localhost:8080`.

docker_https

We can build the docker that supports https.

Build certificates

Install and run mkcert to make certificate and key.

```
brew install mkcert nss
mkcert -install
mkcert localhost 127.0.0.1 ::1 myapp.local
```

Then generated files in the certs directory.

```
docker_https/
├── app
│   └── Dockerfile
├── docker-compose.yml
├── nginx
│   ├── certs
│   │   ├── localhost+3-key.pem
│   │   └── localhost+3.pem
│   ├── Dockerfile
│   └── nginx-https.conf
└── run.sh
```

Update Nginx

The docker file should copy the certs directory to container.

```
FROM nginx:alpine
COPY nginx-https.conf /etc/nginx/conf.d/default.conf
COPY certs/ /etc/nginx/certs/
EXPOSE 80 443
```

1. Redirect the 80 port (http) to 432 (https).
2. Make HTTPS server block.

```
upstream nodejs_app {
    server app:3000;
    keepalive 8;
}

# HTTP Server - Redirect to HTTPS
server {
    listen 80;
    server_name localhost;

    # Redirect all HTTP to HTTPS
    return 301 https://$server_name$request_uri;
}

# HTTPS Server
server {
    listen 443 ssl;
    server_name localhost;

    # SSL Certificate and Key
    ssl_certificate     /etc/nginx/certs/localhost+3.pem;
    ssl_certificate_key /etc/nginx/certs/localhost+3-key.pem;

    ...
}
```

Update docker-compose.yml

Add 8443 to 443 port mapping.

```
services:
  ...
  # NGINX Reverse Proxy
  nginx:
    ...
    ports:
      - "8080:80"  # Host port 8080 → Container port 80
      - "8443:443"
...
```

Build and run Docker containers

```
docker-compose build --no-cache
docker-compose up -d
```

Then, we can access the Docker server with `https://localhost:8433`.