

Docker, CI/CD & Deployment

Images, containers, Compose, pipelines, Nginx, AWS basics

1 Images and containers

Definition

A **Docker image** is a read-only, layered package of an app + its runtime and dependencies. A **container** is a **running instance of an image**, isolated but sharing the host kernel (lighter than a VM).

Container	Virtual machine
shares the host kernel	has its own OS kernel
starts in seconds, MBs	starts in minutes, GBs
process-level isolation	hardware-level isolation

Multi-stage Dockerfile for a Node app

```
# 1) build stage
FROM node:22-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci                                # cached unless dependencies change
COPY . .
RUN npm run build

# 2) small runtime image
FROM node:22-alpine
WORKDIR /app
ENV NODE_ENV=production
COPY --from=build /app/package*.json ./
RUN npm ci --omit=dev
COPY --from=build /app/dist ./dist
USER node                                # don't run as root
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

- Layer caching: copy `package.json` and install **before** copying source code.
- Use `.dockerignore` (`node_modules`, `.git`, `.env`).
- Containers are disposable: keep data in **volumes** or managed databases.
- Config via environment variables; secrets via the platform's secret store.

docker-compose.yml: app + Postgres + Redis

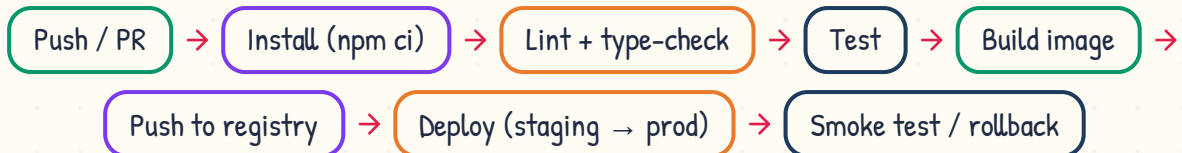
```
services:
  api:
    build: .
    ports: ["3000:3000"]
    environment:
      DATABASE_URL: postgres://app:secret@db:5432/app # service name =
hostname
      REDIS_URL: redis://cache:6379
    depends_on: [db, cache]
  db:
    image: postgres:16
    environment: { POSTGRES_USER: app, POSTGRES_PASSWORD: secret, POSTGRES_DB:
app }
    volumes: [pgdata:/var/lib/postgresql/data]
  cache:
    image: redis:7
volumes:
  pgdata:
```

 `docker compose up -d`, `logs -f api`, `exec api sh`, `down` (add `-v` to delete volumes).

 Containers on the same Compose network reach each other by service name.

3

CI/CD pipelines



GitHub Actions

```
name: ci
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: 22, cache: npm }
      - run: npm ci
      - run: npm run lint && npm test && npm run build
```

- ✏️ **CI** (continuous integration): every change is built and tested automatically. **CD**: every passing change is deployed (or ready to deploy).
- ✏️ Same tools elsewhere: Jenkins (Jenkinsfile stages), GitLab CI, CircleCI.
- ✏️ Deploy strategies: rolling, blue-green (switch traffic between two environments), canary (a small % first).
- ✏️ Keep database migrations backwards-compatible so you can roll back the app.

Nginx reverse proxy in front of Node

```
server {  
    listen 443 ssl;  
    server_name api.example.com;  
    ssl_certificate      /etc/letsencrypt/live/api.example.com/fullchain.pem;  
    ssl_certificate_key  /etc/letsencrypt/live/api.example.com/privkey.pem;  
  
    location / {  
        proxy_pass http://127.0.0.1:3000;  
        proxy_set_header Host $host;  
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
        proxy_set_header Upgrade $http_upgrade;           # WebSockets  
        proxy_set_header Connection "upgrade";  
    }  
}
```

- ✎ DNS: an A record points the domain at the server IP; a CNAME points at another name.
- ✎ Free TLS certificates with Let's Encrypt (certbot) and auto-renewal.
- ✎ Nginx also serves static files, gzip/brotli, caching and load balancing (**upstream**).

Service	Use it for
EC2	virtual servers
S3	file / object storage, static sites
CloudFront	CDN
RDS / Aurora	managed SQL databases
ElastiCache	managed Redis
ECS / EKS / Fargate	run containers
Lambda	serverless functions
Route 53	DNS
IAM	users, roles, least-privilege permissions
CloudWatch	logs, metrics, alarms
SQS / SNS	queues / pub-sub

- Vercel / Netlify for frontends and Next.js; Render, Railway or Fly.io for simple backends.
- Least privilege IAM, private subnets for databases, backups, and alarms on errors and cost.

Revision

Image = package, container = running instance. Multi-stage builds, layer caching, non-root user.
Compose for local multi-service setups. CI on every PR, CD with safe strategies and rollback. Nginx
+ Let's Encrypt in front of the app.

Q. Image vs container?

A. An image is a read-only template; a container is a running instance of it with its own writable layer.

Q. Container vs virtual machine?

A. Containers share the host kernel and start fast with little overhead; VMs run a full OS with stronger isolation.

Q. Why use multi-stage builds?

A. To compile in one stage and copy only the output into a small runtime image — smaller, faster and more secure.

Q. How does Docker layer caching work?

A. Each instruction is a cached layer; if its inputs haven't changed it is reused, so install dependencies before copying source code.

Q. CI vs CD?

A. CI builds and tests every change automatically; CD automatically deploys (or prepares for deployment) every change that passes.

Q. Blue-green vs canary deployment?

A. Blue-green switches all traffic between two identical environments; canary sends a small percentage to the new version first.

Q. What is a reverse proxy?

A. A server (e.g. Nginx) in front of apps that handles TLS, routing, caching, compression and load balancing.

Q. How do containers talk in Docker Compose?

A. They share a network and reach each other by service name as the hostname.

Q. Where should app data live with containers?

A. In volumes or managed services (RDS, S3) — containers are disposable.