10 best practices to containerize Node.js web applications with Docker

01 Use officially supported and deterministic image tags

- Avoid `FROM node`
- Avoid `FROM node:lts`
- Avoid `FROM node:14-alpine`

Avoid Alpine which isn't officially supported. Avoid other image tags which have a high software footprint. Prefer a slimmer, up-to-date and LTS version:

- `FROM node:16.17.0-bullseye-slim`

02 Install only production dependencies

Avoid pulling devDependencies and non-deterministic package install like the ones below:

- Avoid `RUN npm install`
- Avoid `RUN yarn install`
- Avoid `RUN npm ci`

Instead, ensure you are installing only production dependencies in a reproducible way:

- `RUN npm ci --only=production`

03 Optimize Node.js apps for production

Some Node.js libraries and frameworks will only enable production-related optimization if they detect that the NODE_ENV env var set to production:

- `ENV NODE_ENV production`

04 Don't run Node.js apps as root

Docker defaults to running the process in the container as the root user, which is a precarious security practice. Use a low privileged user and proper filesystem permissions:

- `USER node`
- `COPY --chown=node:node . /usr/src/app`

05 Properly handle events to safely terminate a Node.js application

Docker creates processes as PID 1, and they must inherently handle process signals to function properly. This is why you should avoid any of these variations:

- `CMD "npm" "start"`
- `CMD ["yarn", "start"]`
- `CMD "node" "server.js"`
- `CMD "start-app.sh"`

Instead, use a lightweight init system, such as **dumb-init**, to properly spawn the Node.js runtime process with signals support:

- `CMD ["dumb-init", "node", "server.js"]`

06 Gracefully tear down Node.js apps

Avoid an abrupt termination of a running Node.js application that halts live connections. Instead, use a process signal event handler:

```
async function closeGracefully(signal) {
   await fastify.close()
   process.kill(process.pid, signal);
}
process.on('SIGINT', closeGracefully)
```

07 Find and fix security vulnerabilities in your Node.js Docker image

Docker base images may include security vulnerabilities in the software toolchain they bundle, including the Node.js runtime itself. Scan and fix security vulnerabilities with the free Snyk Container tool which also provides base image recommendations:

- `npm install -g snyk`
- `snyk auth`
- `snyk container test node:16.17.0-bullseye-slim --file=Dockerfile`

08 Use multi-stage builds

Avoid having one big build stage when attempting to clean up sensitive data from it or dangling dependencies. Instead, use multi-stage Docker image builds and separate concerns between the build flow and the creation of a production base image.

09 Use .dockerignore

Use **.dockerignore** to ensure:

- local artifacts of node_modules/ aren't copied into the container image.
- sensitive files, such as **.npmrc**, **.env** or others, aren't leaked into the container image.
- a small Docker base image without redundant and unnecessary files.

10 Mount secrets into the Docker image

Secrets are a tricky thing to manage. Avoid the following security pitfalls:

- passing secrets via build arguments in non multi-stage builds
- putting secrets inside the Dockerfile

Instead, use the built-in secrets mounting. To mount a **.npmrc** file for package install:

- In the Dockerfile: `RUN --mount=type=secret,id=npmrc, target=/usr/src/app/.npmrc npm ci --only=production`
- Then build the image with: `docker build . --build-arg NPM_TOKEN=1234 --secret id=npmrc,src=.npmrc`

Authors

@liran_tal

@goldbergyoni

snyk