🚀 Architecture & Deployment

2025-2026 v0.1.0 on branch main Rev: c5cdbe8f75ddbb918b91340af69beb502fe32fa4

Docker

Useful commands and tips for working with Docker.

Table of contents

- Common Docker commands
- Common Dockerfile instructions
- Dockerfile tips
  - Using smaller base images
  - Labeling images
  - Environment variables
    - Non-root users
  - Speeding up builds
  - Documenting exposed ports

Common Docker commands

Command	Purpose
`docker run <image>`	Create and start a container from an image.
`docker ps [-a] [-s]`	Lists running containers.
`docker stop <container>`	Gracefully stops a running container.
`docker start <container>`	Restarts a stopped container.
`docker rm <container>`	Removes a container.

Command	Purpose
`docker exec -it <container> bash`	Provides shell access to a running container.
`docker logs <container>`	Displays a container’s logs.
`docker stats`	Shows a live stream of container(s) resource usage statistics.

To see a full list of Docker commands and their options, see the Dockerfile CLI reference.

Common Dockerfile instructions

Instruction	Purpose
`FROM image_name`	Specifies the base image to use for the new image.
`WORKDIR /some/path`	Sets the working directory for the instructions that follow.
`COPY <src> <dest>`	Copies files or directories from the build context to the image.
`RUN <command>`	Executes commands in the shell during image builds.
`EXPOSE <port>`	Port(s) Docker will be listening on at runtime.
`ENV KEY=VALUE`	Sets environment variables.
`USER user`	Set user and group ID.
`CMD <command>`	The default command to execute when the container starts.
`ENTRYPOINT <command>`	Similar as `CMD`, but cannot be overriden.

To see a full list of Dockerfile instructions, see the Dockerfile reference.

Dockerfile tips

The following tips suggest various best practices for writing Dockerfiles.

Using smaller base images

Many popular Docker images these days have an Alpine variant. This means that the image is based on the official `alpine` image on Docker hub, based on the Alpine Linux distribution. Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.

```
FROM node:24-alpine
```

Here we use the `node:24-alpine` tag instead of simply `node:24`.

These variants are highly recommended when final image size being as small as possible is desired.

The main caveat to note is that Alpine Linux uses musl libc instead of glibc and friends, so certain software might run into compilation issues depending on the depth of their libc requirements. However, most software doesn't have an issue with this, so this variant is usually a very safe choice. See this Hacker News comment thread for more discussion of the issues that might arise and some pro/con comparisons of using Alpine-based images.

To minimize image size, it's uncommon for additional related tools (such as Git or Bash) to be included in Alpine-based images. Using this image as a base, add the things you need in your own Dockerfile (see the alpine image description for examples of how to install packages if you are unfamiliar).

Labeling images

Labels are metadata attached to images and containers. They can be used to influence the behavior of some commands, such as `docker ps`. You can add labels from a Dockerfile with the `LABEL` instruction.

A popular convention is to add a `org.opencontainers.image.authors` label to provide an author (and potential maintenance contact e-mail):

```
LABEL org.opencontainers.image.authors="john.doe@example.com"
```

You may see the labels of an image or container with `docker inspect`.

You may also filter containers by label. For example, to see all running containers that have the `foo` label set to the value `bar`, you can use the following command:

```
docker ps -f label=foo=bar
```

Environment variables

The `ENV` instruction allows you to set environment variables. Many applications change their behavior in response to some variables. For example, a Node.js application might run in production mode if the `$NODE_ENV` variable is set to `production`. Additionally, it might listen on the port specified by the `$PORT` variable.

Here's how you could set both variables:

```
ENV NODE_ENV=production \
    PORT=3000
```

Non-root users

All commands run by a Dockerfile (`RUN` and `CMD` instructions) are **run by the `root` user** of the container by default. This is not a good idea as any security flaw in your application may give root access to the entire container to an attacker.

The security impact of this would be mitigated since the container is isolated from the host machine, but it could still be a **severe security issue** depending on your container's configuration.

Therefore, it is good practice to create an **unprivileged user** to run your application even in the container. Here we use Alpine Linux's `addgroup` and `adduser` commands to create a user, and make sure that your application's directory, e.g. `/app`, where you copy the application is owned by that user:

```
RUN addgroup -S app && \
    adduser -S -G app app && \
    mkdir -p /app && \
    chown app:app /app
```

(Note that these commands are specific to Alpine Linux. You would use `groupadd` and `useradd` on Ubuntu, for example, which use different options.)

Finally, we use the `USER` instruction to make sure that all further commands run in this Dockerfile (by `RUN` or `CMD` instructions) are executed as the new user instead of the root user:

```
USER app:app
```

When using the `COPY` command, you can use the `--chown=app:app` flag to copy files and set their ownership in one go.

Speeding up builds

The following pattern is popular to speed up builds of applications that use a package manager (e.g. npm, RubyGems, Composer).

Installing packages is often one of the slowest command to run for small applications, so we want to take advantage of Docker's build cache as much as possible to avoid running it every time. Suppose you did this like in the Dockerfile for a Node.js application:

```
COPY ./ /app/
WORKDIR /app
RUN npm ci
```

Every time you make the slightest change in any of the application's files, the `COPY` instruction's cache, and all further commands' caches will be invalidated, including the cache for the `RUN npm ci` instruction. Therefore, any change will trigger a full installation of all dependencies from scratch.

To improve this behavior, you can split the installation of your application in the container into two parts. The first part is to copy only the package manager's files (in this case `package.json` and `package-lock.json`) into the application's directory, and to run an `npm ci` command like before:

```
COPY package.json package-lock.json /app/
WORKDIR /app
RUN npm ci
```

Now, if a change is made to the `package.json` or `package-lock.json` files, the cache of the `RUN npm ci` instruction will be invalidated like before, and the dependencies will be re-installed, which we want since the change was probably a dependency update. However, changes in any other file of the application will not invalidate the cache for those 3 instructions, so the result of the `RUN npm ci` instruction will remain cached.

The second part of the installation process is to copy the rest of your application into the directory:

```
COPY ./ /app/
```

Now, if any file in your application changes, the cache of further instructions will be invalidated, but since the `RUN npm ci` instruction comes before, it will remain in the cache and be skipped at build time (unless you modify the `package.json` or `package-lock.json` files).

Documenting exposed ports

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime.

```
EXPOSE 3000
```

The `EXPOSE` instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the `-p` option on `docker run` to publish and map one or more ports, or the `-P` option to publish all exposed ports and map them to high-order ports.

↑ Back to top