## RabbitMQ & Docker

**Step 1: Pulling the RabbitMQ Docker Image**[**​**](https://www.svix.com/resources/guides/rabbitmq-docker-setup-guide/#step-1-pulling-the-rabbitmq-docker-image)

**Pull the official RabbitMQ image**:

docker pull rabbitmq:3-management

This command pulls the RabbitMQ image with the management plugin enabled, which provides a web-based UI for managing your RabbitMQ server.

**Step 2: Running a RabbitMQ Container**[**​**](https://www.svix.com/resources/guides/rabbitmq-docker-setup-guide/#step-2-running-a-rabbitmq-container)

1. **Start a RabbitMQ instance:**

docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management

* -d runs the container in detached mode.
* --name assigns a name to the container.
* -p maps the ports from the container to your host machine. Port 5672 is for RabbitMQ server, and 15672 is for the management UI.

1. **Access the RabbitMQ Management Console:**

* Open a web browser and navigate to http://localhost:15672/.
* Log in with the default username guest and password guest.

**Step 3: Configuring RabbitMQ in Docker**[**​**](https://www.svix.com/resources/guides/rabbitmq-docker-setup-guide/#step-3-configuring-rabbitmq-in-docker)

You can mount a custom configuration file from your host to the container.

docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 -v /path/to/your/rabbitmq.conf:/etc/rabbitmq/rabbitmq.conf rabbitmq:3-management

**Step 4: Setting Up a RabbitMQ Cluster in Docker**[**​**](https://www.svix.com/resources/guides/rabbitmq-docker-setup-guide/#step-4-setting-up-a-rabbitmq-cluster-in-docker)

1. Create a Docker network:

docker network create rabbitmq\_cluster

1. Start RabbitMQ instances on the network:

docker run -d --name rabbitmq1 --hostname rabbitmq1 --network rabbitmq\_cluster rabbitmq:3-management  
docker run -d --name rabbitmq2 --hostname rabbitmq2 --network rabbitmq\_cluster rabbitmq:3-management

1. Cluster the nodes:

First, access the shell of one of the containers:

docker exec -it rabbitmq1 bash

Then, cluster the nodes using rabbitmqctl:

rabbitmqctl stop\_app  
rabbitmqctl reset  
rabbitmqctl join\_cluster rabbit@rabbitmq2  
rabbitmqctl start\_app

Exit the container shell and repeat for other nodes if necessary.

1. Verify the cluster status:

Access the shell of one of the nodes as before and run:

rabbitmqctl cluster\_status

**Step 5: Persisting Data**[**​**](https://www.svix.com/resources/guides/rabbitmq-docker-setup-guide/#step-5-persisting-data)

To persist data, you can mount a volume to the RabbitMQ container.

docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 -v /path/to/data:/var/lib/rabbitmq rabbitmq:3-management

**Conclusion**[**​**](https://www.svix.com/resources/guides/rabbitmq-docker-setup-guide/#conclusion)

Running RabbitMQ in Docker simplifies the process of setting up and managing RabbitMQ instances, especially in development and testing environments. By following the steps outlined in this guide, you can quickly get RabbitMQ running in Docker, configure it according to your needs, and even set up a RabbitMQ cluster.

Remember, while Docker is excellent for development and testing, consider the specifics of your production environment before deploying RabbitMQ in Docker in a production setting.

## PostgreSQL & Docker

### How to use this image

#### **start a postgres instance**

$ docker run --name some-postgres -e POSTGRES\_PASSWORD=mysecretpassword -d postgres

The default postgres user and database are created in the entrypoint with initdb.

The postgres database is a default database meant for use by users, utilities and third party applications.

[postgresql.org/docs⁠](https://www.postgresql.org/docs/14/app-initdb.html)

#### **... or via psql**

$ docker run -it --rm --network some-network postgres psql -h some-postgres -U postgres

psql (14.3)

Type "help" for help.

postgres=# SELECT 1;

?column?

----------

1

(1 row)

#### **... via**[**docker-compose⁠**](https://github.com/docker/compose)**or**[**docker stack deploy⁠**](https://docs.docker.com/engine/reference/commandline/stack_deploy/)

Example docker-compose.yml for postgres:

# Use postgres/example user/password credentials

version: '3.9'

services:

db:

image: postgres

restart: always

# set shared memory limit when using docker-compose

shm\_size: 128mb

# or set shared memory limit when deploy via swarm stack

#volumes:

# - type: tmpfs

# target: /dev/shm

# tmpfs:

# size: 134217728 # 128\*2^20 bytes = 128Mb

environment:

POSTGRES\_PASSWORD: example

adminer:

image: adminer

restart: always

ports:

- 8080:8080

Run docker stack deploy -c stack.yml postgres (or docker-compose -f stack.yml up), wait for it to initialize completely, and visit http://swarm-ip:8080, http://localhost:8080, or http://host-ip:8080 (as appropriate).

### How to extend this image

There are many ways to extend the postgres image. Without trying to support every possible use case, here are just a few that we have found useful.

#### **Environment Variables**

The PostgreSQL image uses several environment variables which are easy to miss. The only variable required is POSTGRES\_PASSWORD, the rest are optional.

**Warning**: the Docker specific variables will only have an effect if you start the container with a data directory that is empty; any pre-existing database will be left untouched on container startup.

##### **POSTGRES\_PASSWORD**

This environment variable is required for you to use the PostgreSQL image. It must not be empty or undefined. This environment variable sets the superuser password for PostgreSQL. The default superuser is defined by the POSTGRES\_USER environment variable.

**Note 1:** The PostgreSQL image sets up trust authentication locally so you may notice a password is not required when connecting from localhost (inside the same container). However, a password will be required if connecting from a different host/container.

**Note 2:** This variable defines the superuser password in the PostgreSQL instance, as set by the initdb script during initial container startup. It has no effect on the PGPASSWORD environment variable that may be used by the psql client at runtime, as described at [https://www.postgresql.org/docs/14/libpq-envars.html⁠](https://www.postgresql.org/docs/14/libpq-envars.html). PGPASSWORD, if used, will be specified as a separate environment variable.

##### **POSTGRES\_USER**

This optional environment variable is used in conjunction with POSTGRES\_PASSWORD to set a user and its password. This variable will create the specified user with superuser power and a database with the same name. If it is not specified, then the default user of postgres will be used.

Be aware that if this parameter is specified, PostgreSQL will still show The files belonging to this database system will be owned by user "postgres" during initialization. This refers to the Linux system user (from /etc/passwd in the image) that the postgres daemon runs as, and as such is unrelated to the POSTGRES\_USER option. See the section titled "Arbitrary --user Notes" for more details.

##### **POSTGRES\_DB**

This optional environment variable can be used to define a different name for the default database that is created when the image is first started. If it is not specified, then the value of POSTGRES\_USER will be used.

##### **POSTGRES\_INITDB\_ARGS**

This optional environment variable can be used to send arguments to postgres initdb. The value is a space separated string of arguments as postgres initdb would expect them. This is useful for adding functionality like data page checksums: -e POSTGRES\_INITDB\_ARGS="--data-checksums".

##### **POSTGRES\_INITDB\_WALDIR**

This optional environment variable can be used to define another location for the Postgres transaction log. By default the transaction log is stored in a subdirectory of the main Postgres data folder (PGDATA). Sometimes it can be desireable to store the transaction log in a different directory which may be backed by storage with different performance or reliability characteristics.

**Note:** on PostgreSQL 9.x, this variable is POSTGRES\_INITDB\_XLOGDIR (reflecting [the changed name of the --xlogdir flag to --waldir in PostgreSQL 10+⁠](https://wiki.postgresql.org/wiki/New_in_postgres_10#Renaming_of_.22xlog.22_to_.22wal.22_Globally_.28and_location.2Flsn.29)).

##### **POSTGRES\_HOST\_AUTH\_METHOD**

This optional variable can be used to control the auth-method for host connections for all databases, all users, and all addresses. If unspecified then [scram-sha-256 password authentication⁠](https://www.postgresql.org/docs/14/auth-password.html) is used (in 14+; md5 in older releases). On an uninitialized database, this will populate pg\_hba.conf via this approximate line:

echo "host all all all $POSTGRES\_HOST\_AUTH\_METHOD" >> pg\_hba.conf

See the PostgreSQL documentation on [pg\_hba.conf⁠](https://www.postgresql.org/docs/14/auth-pg-hba-conf.html) for more information about possible values and their meanings.

**Note 1:** It is not recommended to use trust since it allows anyone to connect without a password, even if one is set (like via POSTGRES\_PASSWORD). For more information see the PostgreSQL documentation on [Trust Authentication⁠](https://www.postgresql.org/docs/14/auth-trust.html).

**Note 2:** If you set POSTGRES\_HOST\_AUTH\_METHOD to trust, then POSTGRES\_PASSWORD is not required.

**Note 3:** If you set this to an alternative value (such as scram-sha-256), you might need additional POSTGRES\_INITDB\_ARGS for the database to initialize correctly (such as POSTGRES\_INITDB\_ARGS=--auth-host=scram-sha-256).

##### **PGDATA**

**Important Note:** Mount the data volume at /var/lib/postgresql/data and not at /var/lib/postgresql because mounts at the latter path WILL NOT PERSIST database data when the container is re-created. The Dockerfile that builds the image declares a volume at /var/lib/postgresql/data and if no data volume is mounted at that path then the container runtime will automatically create an [anonymous volume⁠](https://docs.docker.com/engine/storage/#volumes) that is not reused across container re-creations. Data will be written to the anonymous volume rather than your intended data volume and won't persist when the container is deleted and re-created.

This optional variable can be used to define another location - like a subdirectory - for the database files. The default is /var/lib/postgresql/data. If the data volume you're using is a filesystem mountpoint (like with GCE persistent disks), or remote folder that cannot be chowned to the postgres user (like some NFS mounts), or contains folders/files (e.g. lost+found), Postgres initdb requires a subdirectory to be created within the mountpoint to contain the data.

For example:

$ docker run -d \

--name some-postgres \

-e POSTGRES\_PASSWORD=mysecretpassword \

-e PGDATA=/var/lib/postgresql/data/pgdata \

-v /custom/mount:/var/lib/postgresql/data \

postgres

This is an environment variable that is not Docker specific. Because the variable is used by the postgres server binary (see the [PostgreSQL docs⁠](https://www.postgresql.org/docs/14/app-postgres.html#id-1.9.5.14.7)), the entrypoint script takes it into account.

#### **Docker Secrets**

As an alternative to passing sensitive information via environment variables, \_FILE may be appended to some of the previously listed environment variables, causing the initialization script to load the values for those variables from files present in the container. In particular, this can be used to load passwords from Docker secrets stored in /run/secrets/<secret\_name> files. For example:

$ docker run --name some-postgres -e POSTGRES\_PASSWORD\_FILE=/run/secrets/postgres-passwd -d postgres

Currently, this is only supported for POSTGRES\_INITDB\_ARGS, POSTGRES\_PASSWORD, POSTGRES\_USER, and POSTGRES\_DB.

#### **Initialization scripts**

If you would like to do additional initialization in an image derived from this one, add one or more \*.sql, \*.sql.gz, or \*.sh scripts under /docker-entrypoint-initdb.d (creating the directory if necessary). After the entrypoint calls initdb to create the default postgres user and database, it will run any \*.sql files, run any executable \*.sh scripts, and source any non-executable \*.sh scripts found in that directory to do further initialization before starting the service.

**Warning**: scripts in /docker-entrypoint-initdb.d are only run if you start the container with a data directory that is empty; any pre-existing database will be left untouched on container startup. One common problem is that if one of your /docker-entrypoint-initdb.d scripts fails (which will cause the entrypoint script to exit) and your orchestrator restarts the container with the already initialized data directory, it will not continue on with your scripts.

For example, to add an additional user and database, add the following to /docker-entrypoint-initdb.d/init-user-db.sh:

#!/bin/bash

set -e

psql -v ON\_ERROR\_STOP=1 --username "$POSTGRES\_USER" --dbname "$POSTGRES\_DB" <<-EOSQL

CREATE USER docker;

CREATE DATABASE docker;

GRANT ALL PRIVILEGES ON DATABASE docker TO docker;

EOSQL

These initialization files will be executed in sorted name order as defined by the current locale, which defaults to en\_US.utf8. Any \*.sql files will be executed by POSTGRES\_USER, which defaults to the postgres superuser. It is recommended that any psql commands that are run inside of a \*.sh script be executed as POSTGRES\_USER by using the --username "$POSTGRES\_USER" flag. This user will be able to connect without a password due to the presence of trust authentication for Unix socket connections made inside the container.

Additionally, as of [docker-library/postgres#253⁠](https://github.com/docker-library/postgres/pull/253), these initialization scripts are run as the postgres user (or as the "semi-arbitrary user" specified with the --user flag to docker run; see the section titled "Arbitrary --user Notes" for more details). Also, as of [docker-library/postgres#440⁠](https://github.com/docker-library/postgres/pull/440), the temporary daemon started for these initialization scripts listens only on the Unix socket, so any psql usage should drop the hostname portion (see [docker-library/postgres#474 (comment)⁠](https://github.com/docker-library/postgres/issues/474#issuecomment-416914741) for example).

#### **Database Configuration**

There are many ways to set PostgreSQL server configuration. For information on what is available to configure, see the [PostgreSQL docs⁠](https://www.postgresql.org/docs/14/runtime-config.html) for the specific version of PostgreSQL that you are running. Here are a few options for setting configuration:

* Use a custom config file. Create a config file and get it into the container. If you need a starting place for your config file you can use the sample provided by PostgreSQL which is available in the container at /usr/share/postgresql/postgresql.conf.sample (/usr/local/share/postgresql/postgresql.conf.sample in Alpine variants).
  + **Important note:** you must set listen\_addresses = '\*'so that other containers will be able to access postgres.
* $ # get the default config
* $ docker run -i --rm postgres cat /usr/share/postgresql/postgresql.conf.sample > my-postgres.conf
* $ # customize the config
* $ # run postgres with custom config
* $ docker run -d --name some-postgres -v "$PWD/my-postgres.conf":/etc/postgresql/postgresql.conf -e POSTGRES\_PASSWORD=mysecretpassword postgres -c 'config\_file=/etc/postgresql/postgresql.conf'
* Set options directly on the run line. The entrypoint script is made so that any options passed to the docker command will be passed along to the postgres server daemon. From the [PostgreSQL docs⁠](https://www.postgresql.org/docs/14/app-postgres.html#id-1.9.5.14.6.3) we see that any option available in a .conf file can be set via -c.
* $ docker run -d --name some-postgres -e POSTGRES\_PASSWORD=mysecretpassword postgres -c shared\_buffers=256MB -c max\_connections=200

#### **Locale Customization**

You can extend the Debian-based images with a simple Dockerfile to set a different locale. The following example will set the default locale to de\_DE.utf8:

FROM postgres:14.3

RUN localedef -i de\_DE -c -f UTF-8 -A /usr/share/locale/locale.alias de\_DE.UTF-8

ENV LANG de\_DE.utf8

Since database initialization only happens on container startup, this allows us to set the language before it is created.

Also of note, Alpine-based variants starting with Postgres 15 support [ICU locales⁠](https://www.postgresql.org/docs/15/locale.html#id-1.6.11.3.7). Previous Postgres versions based on alpine do not support locales; see ["Character sets and locale" in the musl documentation⁠](https://wiki.musl-libc.org/functional-differences-from-glibc.html#Character-sets-and-locale) for more details.

You can set locales in the Alpine-based images with POSTGRES\_INITDB\_ARGS to set a different locale. The following example will set the default locale for a newly initialized database to de\_DE.utf8:

$ docker run -d -e LANG=de\_DE.utf8 -e POSTGRES\_INITDB\_ARGS="--locale-provider=icu --icu-locale=de-DE" -e POSTGRES\_PASSWORD=mysecretpassword postgres:15-alpine

#### **Additional Extensions**

When using the default (Debian-based) variants, installing additional extensions (such as PostGIS) should be as simple as installing the relevant packages (see [github.com/postgis/docker-postgis⁠](https://github.com/postgis/docker-postgis/blob/81a0b55/14-3.2/Dockerfile) for a concrete example).

When using the Alpine variants, any postgres extension not listed in [postgres-contrib⁠](https://www.postgresql.org/docs/14/contrib.html) will need to be compiled in your own image (again, see [github.com/postgis/docker-postgis⁠](https://github.com/postgis/docker-postgis/blob/81a0b55/14-3.2/alpine/Dockerfile) for a concrete example).

### Arbitrary --user Notes

As of [docker-library/postgres#253⁠](https://github.com/docker-library/postgres/pull/253), this image supports running as a (mostly) arbitrary user via --user on docker run. As of [docker-library/postgres#1018⁠](https://github.com/docker-library/postgres/pull/1018), this is also the case for the Alpine variants.

The main caveat to note is that postgres doesn't care what UID it runs as (as long as the owner of /var/lib/postgresql/data matches), but initdbdoes care (and needs the user to exist in /etc/passwd):

$ docker run -it --rm --user www-data -e POSTGRES\_PASSWORD=mysecretpassword postgres

The files belonging to this database system will be owned by user "www-data".

...

...

\*\*Note:\*\* the description for this image is longer than the Hub length limit of 25000, so has been trimmed. The full description can be found at [https://github.com/docker-library/docs/tree/master/postgres/README.md](https://github.com/docker-library/docs/tree/master/postgres/README.md). See also [docker/hub-feedback#238](https://github.com/docker/hub-feedback/issues/238) and [docker/roadmap#475](https://github.com/docker/roadmap/issues/475).

## ASP.NET Core & Docker

**Run in a Linux container or Windows container**

* To run in a Linux container, right-click the System Tray's Docker client icon and select [switch to Linux containers](https://docs.docker.com/desktop/windows/#switch-between-windows-and-linux-containers).
* To run in a Windows container, right-click the System Tray's Docker client icon and select [switch to Windows containers](https://docs.docker.com/desktop/windows/#switch-between-windows-and-linux-containers).
* Navigate to the Dockerfile folder at *dotnet-docker/samples/aspnetapp*.
* Run the following commands to build and run the sample in Docker:

ConsoleCopy

docker build -t aspnetapp .

docker run -it --rm -p <port>:8080 --name aspnetcore\_sample aspnetapp

The build command arguments:

* + Name the image aspnetapp.
  + Look for the Dockerfile in the current folder (the period at the end).

The run command arguments:

* + Allocate a pseudo-TTY and keep it open even if not attached. (Same effect as --interactive --tty.)
  + Automatically remove the container when it exits.
  + Map <port> on the local machine to port 8080 in the container.
  + Name the container aspnetcore\_sample.
  + Specify the aspnetapp image.
* Go to http://localhost:<port> in a browser to test the app.

**Build and deploy manually**

In some scenarios, you might want to deploy an app to a container by copying its assets that are needed at run time. This section shows how to deploy manually.

* Navigate to the project folder at *dotnet-docker/samples/aspnetapp/aspnetapp*.
* Run the [dotnet publish](https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-publish) command:

.NET CLICopy

dotnet publish -c Release -o published

The command arguments:

* + Build the app in release mode (the default is debug mode).
  + Create the assets in the *published* folder.
* Run the app.
  + Windows:

.NET CLICopy

dotnet published\aspnetapp.dll

* + Linux:

.NET CLICopy

dotnet published/aspnetapp.dll

* Browse to http://localhost:<port> to see the home page.

To use the manually published app within a Docker container, create a new *Dockerfile* and use the docker build . command to build an image.

DockerfileCopy

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime

WORKDIR /app

COPY published/ ./

ENTRYPOINT ["dotnet", "aspnetapp.dll"]

To see the new image use the docker images command.

**The Dockerfile**

Here's the *Dockerfile* used by the docker build command you ran earlier. It uses dotnet publish the same way you did in this section to build and deploy.

DockerfileCopy

# https://hub.docker.com/\_/microsoft-dotnet

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /source

# copy csproj and restore as distinct layers

COPY \*.sln .

COPY aspnetapp/\*.csproj ./aspnetapp/

RUN dotnet restore

# copy everything else and build app

COPY aspnetapp/. ./aspnetapp/

WORKDIR /source/aspnetapp

RUN dotnet publish -c release -o /app --no-restore

# final stage/image

FROM mcr.microsoft.com/dotnet/aspnet:8.0

WORKDIR /app

COPY --from=build /app ./

ENTRYPOINT ["dotnet", "aspnetapp.dll"]

In the preceding *Dockerfile*, the \*.csproj files are copied and restored as distinct *layers*. When the docker build command builds an image, it uses a built-in cache. If the \*.csproj files haven't changed since the docker build command last ran, the dotnet restore command doesn't need to run again. Instead, the built-in cache for the corresponding dotnet restore layer is reused. For more information, see [Best practices for writing Dockerfiles](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#leverage-build-cache).

## Nginx & Docker

#### **Hosting some simple static content**

$ docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx

Alternatively, a simple Dockerfile can be used to generate a new image that includes the necessary content (which is a much cleaner solution than the bind mount above):

FROM nginx

COPY static-html-directory /usr/share/nginx/html

Place this file in the same directory as your directory of content ("static-html-directory"), then run these commands to build and start your container:

$ docker build -t some-content-nginx .

$ docker run --name some-nginx -d some-content-nginx

#### **Exposing external port**

$ docker run --name some-nginx -d -p 8080:80 some-content-nginx

Then you can hit http://localhost:8080 or http://host-ip:8080 in your browser.

#### **Customize configuration**

You can mount your configuration file, or build a new image with it.

If you wish to adapt the default configuration, use something like the following to get it from a running nginx container:

$ docker run --rm --entrypoint=cat nginx /etc/nginx/nginx.conf > /host/path/nginx.conf

And then edit /host/path/nginx.conf in your host file system.

For information on the syntax of the nginx configuration files, see [the official documentation⁠](http://nginx.org/en/docs/) (specifically the [Beginner's Guide⁠](http://nginx.org/en/docs/beginners_guide.html#conf_structure)).

##### **Mount your configuration file**

$ docker run --name my-custom-nginx-container -v /host/path/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx

##### **Build a new image with your configuration file**

FROM nginx

COPY nginx.conf /etc/nginx/nginx.conf

If you add a custom CMD in the Dockerfile, be sure to include -g daemon off; in the CMD in order for nginx to stay in the foreground, so that Docker can track the process properly (otherwise your container will stop immediately after starting)!

Then build the image with docker build -t custom-nginx . and run it as follows:

$ docker run --name my-custom-nginx-container -d custom-nginx

##### **Using environment variables in nginx configuration (new in 1.19)**

Out-of-the-box, nginx doesn't support environment variables inside most configuration blocks. But this image has a function, which will extract environment variables before nginx starts.

Here is an example using docker-compose.yml:

web:

image: nginx

volumes:

- ./templates:/etc/nginx/templates

ports:

- "8080:80"

environment:

- NGINX\_HOST=foobar.com

- NGINX\_PORT=80

By default, this function reads template files in /etc/nginx/templates/\*.template and outputs the result of executing envsubst to /etc/nginx/conf.d.

So if you place templates/default.conf.template file, which contains variable references like this:

listen ${NGINX\_PORT};

outputs to /etc/nginx/conf.d/default.conf like this:

listen 80;

This behavior can be changed via the following environment variables:

* NGINX\_ENVSUBST\_TEMPLATE\_DIR
  + A directory which contains template files (default: /etc/nginx/templates)
  + When this directory doesn't exist, this function will do nothing about template processing.
* NGINX\_ENVSUBST\_TEMPLATE\_SUFFIX
  + A suffix of template files (default: .template)
  + This function only processes the files whose name ends with this suffix.
* NGINX\_ENVSUBST\_OUTPUT\_DIR
  + A directory where the result of executing envsubst is output (default: /etc/nginx/conf.d)
  + The output filename is the template filename with the suffix removed.
    - ex.) /etc/nginx/templates/default.conf.template will be output with the filename /etc/nginx/conf.d/default.conf.
  + This directory must be writable by the user running a container.

#### **Running nginx in read-only mode**

To run nginx in read-only mode, you will need to mount a Docker volume to every location where nginx writes information. The default nginx configuration requires write access to /var/cache/nginx and /var/run. This can be easily accomplished by running nginx as follows:

$ docker run -d -p 80:80 --read-only -v $(pwd)/nginx-cache:/var/cache/nginx -v $(pwd)/nginx-pid:/var/run nginx

If you have a more advanced configuration that requires nginx to write to other locations, simply add more volume mounts to those locations.

#### **Running nginx in debug mode**

Images since version 1.9.8 come with nginx-debug binary that produces verbose output when using higher log levels. It can be used with simple CMD substitution:

$ docker run --name my-nginx -v /host/path/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx nginx-debug -g 'daemon off;'

Similar configuration in docker-compose.yml may look like this:

web:

image: nginx

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf:ro

command: [nginx-debug, '-g', 'daemon off;']

#### **Entrypoint quiet logs**

Since version 1.19.0, a verbose entrypoint was added. It provides information on what's happening during container startup. You can silence this output by setting environment variable NGINX\_ENTRYPOINT\_QUIET\_LOGS:

$ docker run -d -e NGINX\_ENTRYPOINT\_QUIET\_LOGS=1 nginx

#### **User and group id**

Since 1.17.0, both alpine- and debian-based images variants use the same user and group ids to drop the privileges for worker processes:

$ id

uid=101(nginx) gid=101(nginx) groups=101(nginx)

#### **Running nginx as a non-root user**

It is possible to run the image as a less privileged arbitrary UID/GID. This, however, requires modification of nginx configuration to use directories writeable by that specific UID/GID pair:

$ docker run -d -v $PWD/nginx.conf:/etc/nginx/nginx.conf nginx

where nginx.conf in the current directory should have the following directives re-defined:

pid /tmp/nginx.pid;

And in the http context:

**http** {

client\_body\_temp\_path /tmp/client\_temp;

proxy\_temp\_path /tmp/proxy\_temp\_path;

fastcgi\_temp\_path /tmp/fastcgi\_temp;

uwsgi\_temp\_path /tmp/uwsgi\_temp;

scgi\_temp\_path /tmp/scgi\_temp;

...

}

Alternatively, check out the official [Docker NGINX unprivileged image](https://hub.docker.com/r/nginxinc/nginx-unprivileged).

### Image Variants

The nginx images come in many flavors, each designed for a specific use case.

#### **nginx:<version>**

This is the defacto image. If you are unsure about what your needs are, you probably want to use this one. It is designed to be used both as a throw away container (mount your source code and start the container to start your app), as well as the base to build other images off of.

Some of these tags may have names like bookworm in them. These are the suite code names for releases of [Debian⁠](https://wiki.debian.org/DebianReleases) and indicate which release the image is based on. If your image needs to install any additional packages beyond what comes with the image, you'll likely want to specify one of these explicitly to minimize breakage when there are new releases of Debian.

#### **nginx:<version>-perl / nginx:<version>-alpine-perl**

Starting with nginx:1.13.0 / mainline and nginx:1.12.0 / stable, the perl module has been removed from the default images. A separate -perl tag variant is available if you wish to use the perl module.

#### **nginx:<version>-alpine**

This image is based on the popular [Alpine Linux project⁠](https://alpinelinux.org/), available in [the alpine official image](https://hub.docker.com/_/alpine). Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.

This variant is useful when final image size being as small as possible is your primary concern. The main caveat to note is that it does use [musl libc⁠](https://musl.libc.org/) instead of [glibc and friends⁠](https://www.etalabs.net/compare_libcs.html), so software will often run into issues depending on the depth of their libc requirements/assumptions. See [this Hacker News comment thread⁠](https://news.ycombinator.com/item?id=10782897) 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](https://hub.docker.com/_/alpine/) for examples of how to install packages if you are unfamiliar).

#### **nginx:<version>-slim**

This image does not contain the common packages contained in the default tag and only contains the minimal packages needed to run nginx. Unless you are working in an environment where only the nginx image will be deployed and you have space constraints, we highly recommend using the default image of this repository.

### License

View [license information⁠](http://nginx.org/LICENSE) for the software contained in this image.

As with all Docker images, these likely also contain other software which may be under other licenses (such as Bash, etc from the base distribution, along with any direct or indirect dependencies of the primary software being contained).

Some additional license information which was able to be auto-detected might be found in [the repo-info repository's nginx/ directory⁠](https://github.com/docker-library/repo-info/tree/master/repos/nginx).

As for any pre-built image usage, it is the image user's responsibility to ensure that any use of this image complies with any relevant licenses for all software contained within.