

Building images

- [Building images](#)
 - [Images and containers](#)
 - [Dockerfile instructions](#)
 - [Choosing the right base image](#)
 - [Copying files into the image](#)
 - [Excluding files](#)
 - [Running commands](#)
 - [Setting environment variables](#)
 - [Exposing ports](#)
 - [Setting the user](#)
 - [Defining entrypoint](#)
 - [Speeding up builds](#)
 - [Removing images](#)
 - [Tagging images](#)
 - [Sharing images](#)
 - [Saving and loading images](#)

Images and containers

- An **Image** includes everything an application needs to run
- A **Container** provides an isolated environment for running the application

Each container gets its base from the image, but each one has its own write layer.

Dockerfile instructions

The first step to dockerize an application is to add a **Dockerfile** to it. This file will contain the instructions for building an image. Instructions can be:

- **FROM** for specifying the base image.
- **WORKDIR** for setting the working directory. Once we set it, all subsequent commands will be run from that directory.
- **COPY** and **ADD** for copying files and directories into the container.
- **RUN** for executing operating system commands.
- **ENV** for setting environment variables.
- **EXPOSE** for exposing the application on a given port
- **USER** for specifying the user that should run the application. This should be a user with limited privileges.
- **CMD** and **ENTRYPOINT** for specifying the command that should be executed when we start a container.

Choosing the right base image

Images can be in any registry. The default that Docker will use is DockerHub, but there are others (like MCR). When not using DockerHub, we need to supply the complete URL for the **FROM** instruction. For each base image (like **node**, **python**, etc) we can have multiple (even hundreds) of different tags that combine versions

of our runtime and OS. Choosing the correct one is important- But doing so requires understanding the specifications of the application, requirements, dependencies, etc. and therefore needs to be done on a case by case basis.

Once we are ready to build the image, we run:

```
docker build -t IMAGE_NAME .
```

To check the images that we have, we use:

```
docker images
```

To start a container using the image that we've built, we use:

```
docker run -it IMAGE_NAME
```

If we want to start the container in a different program than the default (for example, start in `bash`) we run:

```
docker run -it IMAGE_NAME bash
```

Or `shell`:

```
docker run -it IMAGE_NAME sh
```

An example `Dockerfile` at this point would look like:

```
FROM node:14.16.0-alpine3.13
```

Copying files into the image

Once we have selected a base image, we need to copy the application files into the image. We use the `COPY` command to copy files or directories from the current directory (the directory where the `Dockerfile` is located) into the image. The destination can be a directory in the image. If it doesn't exist, Docker will create it.

We can specify multiple files and or directories to copy by simply listing them with spaces. Remember that the list is case-sensitive. We can also use patterns to specify the list of files. If we want to copy everything in the current directory into the image, we use a period (`.`).

The destination can be a directory, in which case we need to end it with a forward-slash (so `/app/`, not `/app`). We can use relative directories if the first set the `WORKDIR` command. If the destination is the `WORKDIR`, then we use a period (`.`).

An example `Dockerfile` at this point would look like:

```
FROM node:14.16.0-alpine3.13
WORKDIR /app
COPY . .
```

If one of our files has a space in its name, then we need to use the array format of the `COPY` command:

```
FROM node:14.16.0-alpine3.13
WORKDIR /app
COPY ["hello world.txt", "."]
```

Alternatively, we can use the `ADD` command. This command has all the same features of the `COPY` command but:

- we can add files from URLs (`http://.../file.json`)
- if we supply a compressed file, `ADD` will automatically decompress it in the image. (`ADD file.zip`)

The best practice is to use `COPY`. When setting the `WORKDIR` the container will start there if we run it in interactive mode.

Excluding files

We don't need to build and ship images with all the application dependencies. We can add files that explain how the environment needs to be built, exclude the dependencies and libraries themselves, and then add the command to re-establish the environment.

To ignore files, we use a `.dockerignore` file. Any file or directory that we include in that file will be excluded at build time.

Running commands

As part of the container build process we can run commands using the `RUN` keyword. For example, we can use this to install dependencies. Docker will download and install all dependencies when building, so that they are available when the image is used in a container.

```
FROM node:14.16.0-alpine3.13
WORKDIR /app
COPY . .
RUN npm install
```

In each invocation of the **RUN** command we can pass a command. We can have as many invocations as needed.

Setting environment variables

To set environment variables we use the **ENV** instruction. To it we need to supply **key=value** pairs. The **=** sign can be omitted, but that is no longer considered best practice.

```
FROM node:14.16.0-alpine3.13
WORKDIR /app
COPY . .
RUN npm install
ENV API_URL=http://api.myapp.com/
```

Exposing ports

To set the port in which the container will be listing on we use the **EXPOSE** command. To it we pass the port number. But this is the port of the container, not the localhost (on our machine).

```
FROM node:14.16.0-alpine3.13
WORKDIR /app
COPY . .
RUN npm install
ENV API_URL=http://api.myapp.com/
EXPOSE 3000
```

Setting the user

By default, Docker will run our application with the **root** user. But this user has too many privileges and can cause security issues in the application. For running the application we need to use a **system** user. This user needs to be created when building the image. And before creating the user, we need to create a group so that we can add the user to that group. We do all of this in one command. The best practice is for this system user and the group to be called **app**.

```
FROM node:14.16.0-alpine3.13
WORKDIR /app
COPY . .
RUN npm install
ENV API_URL=http://api.myapp.com/
EXPOSE 3000
RUN addgroup app && adduser -S -G app app
```

Now we need to use the **USER** command to run the application using this user.

```
FROM node:14.16.0-alpine3.13
WORKDIR /app
COPY . .
RUN npm install
ENV API_URL=http://api.myapp.com/
EXPOSE 3000
RUN addgroup app && adduser -S -G app app
USER app
```

Defining entrypoint

If we try running the application in the container at this point it won't work. The reason is that up until we use `RUN` command, we did everything with the `root` user. The new `app` user does not have permissions to access the `/app` directory in the image.

```
FROM node:14.16.0-alpine3.13
RUN addgroup app && adduser -S -G app app
USER app
WORKDIR /app
COPY . .
RUN npm install
ENV API_URL=http://api.myapp.com/
EXPOSE 3000
```

Now we can rebuild the image. When done we can start the app by running:

```
docker run IMAGE_NAME npm start
```

But we don't want to have to include `npm start` every time we want to start the application in the container. We can work around this by specifying a command at the end of the `Dockerfile`.

```
FROM node:14.16.0-alpine3.13
RUN addgroup app && adduser -S -G app app
USER app
WORKDIR /app
COPY . .
RUN npm install
ENV API_URL=http://api.myapp.com/
EXPOSE 3000
CMD npm start
```

Because the `CMD` instruction is to start the app we should not include multiple calls to it in the `Dockerfile`. If we do, only the last one will take effect.

The difference between **RUN** and **CMD** is that **RUN** is a build-time instruction. It will be executed when we run **docker build**. On the other hand, **CMD** is a run-time instruction. It will be executed when we start the container via **docker run**.

The **CMD** command has two forms. The *shell form* takes a command as we would run it on the local shell. The *execute form* takes an array of strings.

```
# Shell form
CMD npm start

# Execute form
CMD ["npm", "start"]
```

The best practice is to use the execute form, because the shell form will be executed inside a separate shell (**/bin/sh** for Linux, or **cmd** for Windows). Therefore, the cleanup process for the shell form is more expensive.

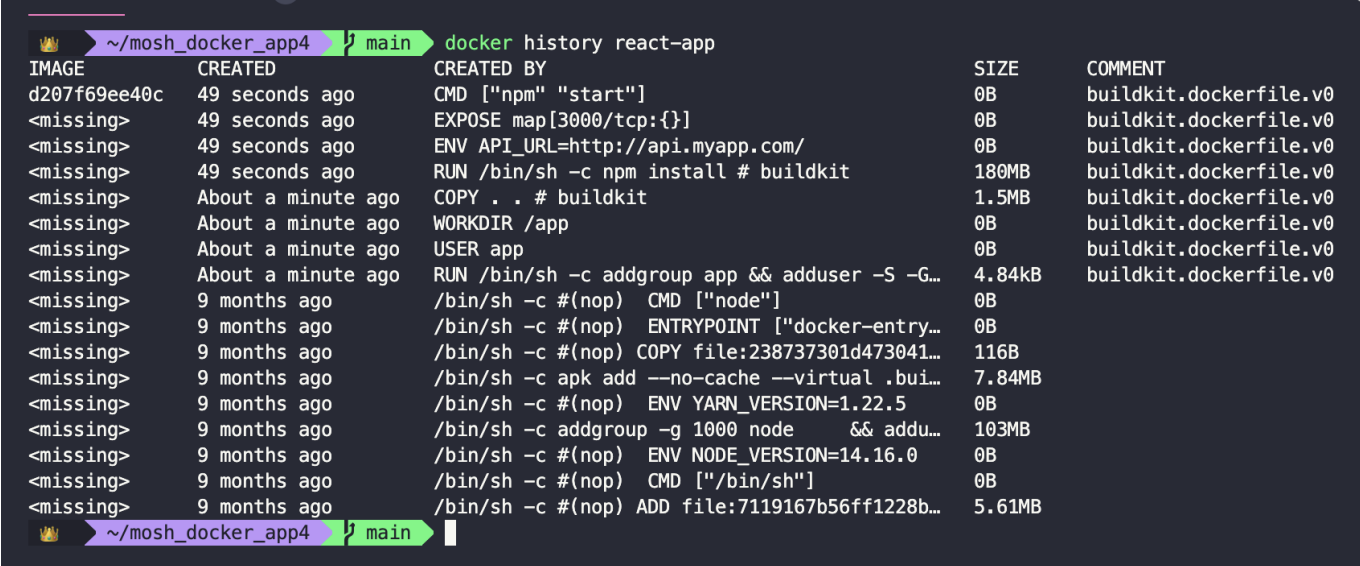
We also have a similar command called **ENTRYPOINT**. This too can be either a shell or execute form. The difference is that the **CMD** command can be over-written when running **docker run** by simply supplying a different command. To override the **ENTRYPOINT** command we would have to use the **--entrypoint** option. Therefore, it's recommended to use the **ENTRYPOINT** to specifying commands that we always want to run when starting the container, and using **CMD** for adhoc commands.

Speeding up builds

An image is a collection of layers. A layer is a small file system that only includes modified files. When Docker builds the image he does so by executing the commands line after line, each one generating a new layer. That layer only includes the files that were modified as a result of that instruction. (Technically, some instructions might generate more than one layer. For example, the base image is generally going to consist of more than one layer.)

We can explore the layers with the following command:

```
docker history IMAGE_NAME
```



The screenshot shows a terminal window with the prompt `~/mosh_docker_app4 main` and the command `docker history react-app`. The output is a table with 5 columns: IMAGE, CREATED, CREATED BY, SIZE, and COMMENT. The table lists the build history of the container, showing layers created by the buildkit. The layers are listed from bottom to top, starting with the base image `d207f69ee40c` and ending with the final image `d207f69ee40c`. The layers include instructions like `EXPOSE map[3000/tcp:{}]`, `ENV API_URL=http://api.myapp.com/`, `RUN /bin/sh -c npm install # buildkit`, `COPY . . # buildkit`, `WORKDIR /app`, `USER app`, and `RUN /bin/sh -c addgroup app && adduser -S -G...`. The sizes of the layers are listed in the `SIZE` column, ranging from 0B to 5.61MB.

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
d207f69ee40c	49 seconds ago	CMD ["npm" "start"]	0B	buildkit.dockerfile.v0
<missing>	49 seconds ago	EXPOSE map[3000/tcp:{}]	0B	buildkit.dockerfile.v0
<missing>	49 seconds ago	ENV API_URL=http://api.myapp.com/	0B	buildkit.dockerfile.v0
<missing>	49 seconds ago	RUN /bin/sh -c npm install # buildkit	180MB	buildkit.dockerfile.v0
<missing>	About a minute ago	COPY . . # buildkit	1.5MB	buildkit.dockerfile.v0
<missing>	About a minute ago	WORKDIR /app	0B	buildkit.dockerfile.v0
<missing>	About a minute ago	USER app	0B	buildkit.dockerfile.v0
<missing>	About a minute ago	RUN /bin/sh -c addgroup app && adduser -S -G...	4.84kB	buildkit.dockerfile.v0
<missing>	9 months ago	/bin/sh -c #(nop) CMD ["node"]	0B	
<missing>	9 months ago	/bin/sh -c #(nop) ENTRYPOINT ["docker-entry...	0B	
<missing>	9 months ago	/bin/sh -c #(nop) COPY file:238737301d473041...	116B	
<missing>	9 months ago	/bin/sh -c apk add --no-cache --virtual .bui...	7.84MB	
<missing>	9 months ago	/bin/sh -c #(nop) ENV YARN_VERSION=1.22.5	0B	
<missing>	9 months ago	/bin/sh -c addgroup -g 1000 node && addu...	103MB	
<missing>	9 months ago	/bin/sh -c #(nop) ENV NODE_VERSION=14.16.0	0B	
<missing>	9 months ago	/bin/sh -c #(nop) CMD ["/bin/sh"]	0B	
<missing>	9 months ago	/bin/sh -c #(nop) ADD file:7119167b56ff1228b...	5.61MB	

In the **CREATED BY** column you can see the instruction that created the layer. And in the **SIZE** column you can see the size of that layer. This list needs to be read from bottom to top.

We can use caching to speed up our builds. When doing so, Docker will first check if the line that it's about to run was modified. If it was not, then it will not re-execute it, but use the cached one. The problem is that once a layer is re-built, all the subsequent layers need to be rebuilt too. Since the **COPY** instruction needs to come before the **RUN** instructions that install dependencies, every time we change the code base of our application the dependencies need to be installed again. This is generally the bottle neck in the build process.

The solution to this problem is to separate the instructions. First we'll copy all the list of dependencies and install them. Then we'll copy the application code.

```
FROM node:14.16.0-alpine3.13
RUN addgroup app && adduser -S -G app app
USER app
WORKDIR /app
COPY package*.json .
RUN npm install
COPY . .
ENV API_URL=http://api.myapp.com/
EXPOSE 3000
CMD npm start
```

```

~/mosh_docker_app4 main ± docker build -t react-app .
[+] Building 54.9s (12/12) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring dockerfile: 243B                                              0.0s
=> [internal] load .dockerignore                                                 0.0s
=> => transferring context: 34B                                                  0.0s
=> [internal] load metadata for docker.io/library/node:14.16.0-alpine3.13      2.5s
=> [auth] library/node:pull token for registry-1.docker.io                     0.0s
=> [internal] load build context                                                0.0s
=> => transferring context: 10.20kB                                             0.0s
=> [1/6] FROM docker.io/library/node:14.16.0-alpine3.13@sha256:2c51dc462a02f15621e7486774d36d048a27225d581374b002b8477a79a59b4b 0.0s
=> CACHED [2/6] RUN addgroup app && adduser -S -G app app                       0.0s
=> CACHED [3/6] WORKDIR /app                                                    0.0s
=> [4/6] COPY package*.json .                                                  0.0s
=> [5/6] RUN npm install                                                        46.6s
=> [6/6] COPY . .                                                              0.1s
=> exporting to image                                                            5.6s
=> => exporting layers                                                            5.6s
=> => writing image sha256:c0ec76fb386a421d256cccbbd1a9badade22b990db5d24f03fb59880fab338b 0.0s
=> => naming to docker.io/library/react-app                                     0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
~/mosh_docker_app4 main ±

```

Here you can see that the first build (after modifying the **Dockerfile**) took almost a minute. Now we can do some changes to the code base and re-build.

```

~/mosh_docker_app4 main ± docker build -t react-app .
[+] Building 1.3s (11/11) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring dockerfile: 37B                                              0.0s
=> [internal] load .dockerignore                                                 0.0s
=> => transferring context: 34B                                                  0.0s
=> [internal] load metadata for docker.io/library/node:14.16.0-alpine3.13      1.1s
=> [1/6] FROM docker.io/library/node:14.16.0-alpine3.13@sha256:2c51dc462a02f15621e7486774d36d048a27225d581374b002b8477a79a59b4b 0.0s
=> [internal] load build context                                                0.0s
=> => transferring context: 13.37kB                                             0.0s
=> CACHED [2/6] RUN addgroup app && adduser -S -G app app                       0.0s
=> CACHED [3/6] WORKDIR /app                                                    0.0s
=> CACHED [4/6] COPY package*.json .                                            0.0s
=> CACHED [5/6] RUN npm install                                                  0.0s
=> [6/6] COPY . .                                                              0.0s
=> exporting to image                                                            0.0s
=> => exporting layers                                                            0.0s
=> => writing image sha256:89802f797aadec27f21b754fdab9fe009ce2a18f7644a236923c4fb4ca5b882c 0.0s
=> => naming to docker.io/library/react-app                                     0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
~/mosh_docker_app4 main ±

```

Now our build only took 1.3 seconds!! If you read the output you can see that while on the first build only layers 1, 2, and 3 were cached, on the second one layers 1 to 5 were cached.

The take away is that the order of the instructions matters!! The more stable instructions should be on the top and the more changing ones on the bottom.



Removing images

Images that have no name and no tag are called **dangling** images. These are essentially layers that have no relationship with tagged images. To get rid of them we use:

```
docker image prune
```

If the dangling images are not removed, that means that we still have containers running that are using the old images. Beware that running `docker ps` might not show these containers, as they might be in the stopped state. Be sure to run `docker ps -a`. To get rid of stopped containers we run:

```
docker container prune
```

After this, `docker images` will still show us the tagged images. To remove one of these images we run:

```
docker image rm IMAGE_NAME_OR_ID
```

We can pass multiple images separated by a space.

Tagging images

Whenever we run `docker build` Docker automatically assigns the image tag as `latest`. This is a meaningless tag. It's just a label. It doesn't necessarily mean that it is in fact the latest build. It might be pointing to an older image and cause problems.

We can tag an image at build time by running:

```
docker build -t REPOSITORY_NAME:TAG_NAME .
```

Which tagging convention you use depends on the situation (semantic versioning, code names, etc.).

```

🔥 ~/mosh_docker_app4 main docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
react-app     1         132c65c38ce8   2 minutes ago  299MB
react-app     latest    132c65c38ce8   2 minutes ago  299MB
<none>        <none>    89802f797aad   15 hours ago   299MB
ubuntu        latest    ba6accedd29    2 months ago   72.8MB
🔥 ~/mosh_docker_app4 main

```

An image can have multiple tags. To check that we just need to look at the image ID. If the ID is the same, it's the same image. We can remove a tag by running:

```
docker image remove REPOSITORY_NAME:TAG_NAME
```

```

🔥 ~/mosh_docker_app4 main docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
react-app     1         132c65c38ce8   2 minutes ago  299MB
react-app     latest    132c65c38ce8   2 minutes ago  299MB
<none>        <none>    89802f797aad   15 hours ago   299MB
ubuntu        latest    ba6accedd29    2 months ago   72.8MB
🔥 ~/mosh_docker_app4 main docker image remove react-app:1
Untagged: react-app:1
🔥 ~/mosh_docker_app4 main docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
react-app     latest    132c65c38ce8   5 minutes ago  299MB
<none>        <none>    89802f797aad   15 hours ago   299MB
ubuntu        latest    ba6accedd29    2 months ago   72.8MB
🔥 ~/mosh_docker_app4 main

```

To tag an image after building it we can use:

```
docker image tag REPOSITORY_NAME:CURRENT_TAG REPOSITORY_NAME:NEW_TAG
```

or

```
docker image tag IMAGE_ID REPOSITORY_NAME:NEW_TAG
```

```

~/mosh_docker_app4 main docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
react-app     latest    132c65c38ce8   11 minutes ago 299MB
<none>        <none>    89802f797aad   16 hours ago   299MB
ubuntu        latest    ba6accce2d29   2 months ago   72.8MB

~/mosh_docker_app4 main docker image tag 132c65c38ce8 react-app:1
~/mosh_docker_app4 main docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
react-app     1         132c65c38ce8   12 minutes ago 299MB
react-app     latest    132c65c38ce8   12 minutes ago 299MB
<none>        <none>    89802f797aad   16 hours ago   299MB
ubuntu        latest    ba6accce2d29   2 months ago   72.8MB

```

The **latest** tag can get out of sync and we should not use it in production. For example, if we now build version 2 we'll see:

```

~/mosh_docker_app4 main docker build -t react-app:2 .
[+] Building 2.5s (12/12) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring dockerfile: 37B                                                0.0s
=> [internal] load .dockerignore                                                  0.0s
=> => transferring context: 34B                                                    0.0s
=> [internal] load metadata for docker.io/library/node:14.16.0-alpine3.13        2.3s
=> [auth] library/node:pull token for registry-1.docker.io                      0.0s
=> [1/6] FROM docker.io/library/node:14.16.0-alpine3.13@sha256:2c51dc462a02f15621e7486774d36d048a27225d581374b002b8477a79a59b4b 0.0s
=> [internal] load build context                                                  0.0s
=> => transferring context: 13.77kB                                                0.0s
=> CACHED [2/6] RUN addgroup app && adduser -S -G app app                        0.0s
=> CACHED [3/6] WORKDIR /app                                                      0.0s
=> CACHED [4/6] COPY package*.json .                                              0.0s
=> CACHED [5/6] RUN npm install                                                    0.0s
=> [6/6] COPY . .                                                                0.0s
=> exporting to image                                                            0.0s
=> => exporting layers                                                            0.0s
=> => writing image sha256:84ba63c80338456e112b88c1bf3a160d2f6828040d1da16f094061d0ac5dec31 0.0s
=> => naming to docker.io/library/react-app:2                                    0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
~/mosh_docker_app4 main docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
react-app     2         84ba63c80338   3 seconds ago   299MB
react-app     1         132c65c38ce8   14 minutes ago 299MB
react-app     latest    132c65c38ce8   14 minutes ago 299MB
<none>        <none>    89802f797aad   16 hours ago   299MB
ubuntu        latest    ba6accce2d29   2 months ago   72.8MB

```

Now the **latest** tag is pointing to version 1, when the latest build is version 2. To change the **latest** tag so that it points to the newest version of the image we run:

```
docker image tag IMAGE_ID REPOSITORY_NAME:latest
```

```

🔥 ~/mosh_docker_app4 main ± docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
react-app 2 84ba63c80338 About a minute ago 299MB
react-app 1 132c65c38ce8 15 minutes ago 299MB
react-app latest 132c65c38ce8 15 minutes ago 299MB
<none> <none> 89802f797aad 16 hours ago 299MB
ubuntu latest ba6accedd29 2 months ago 72.8MB

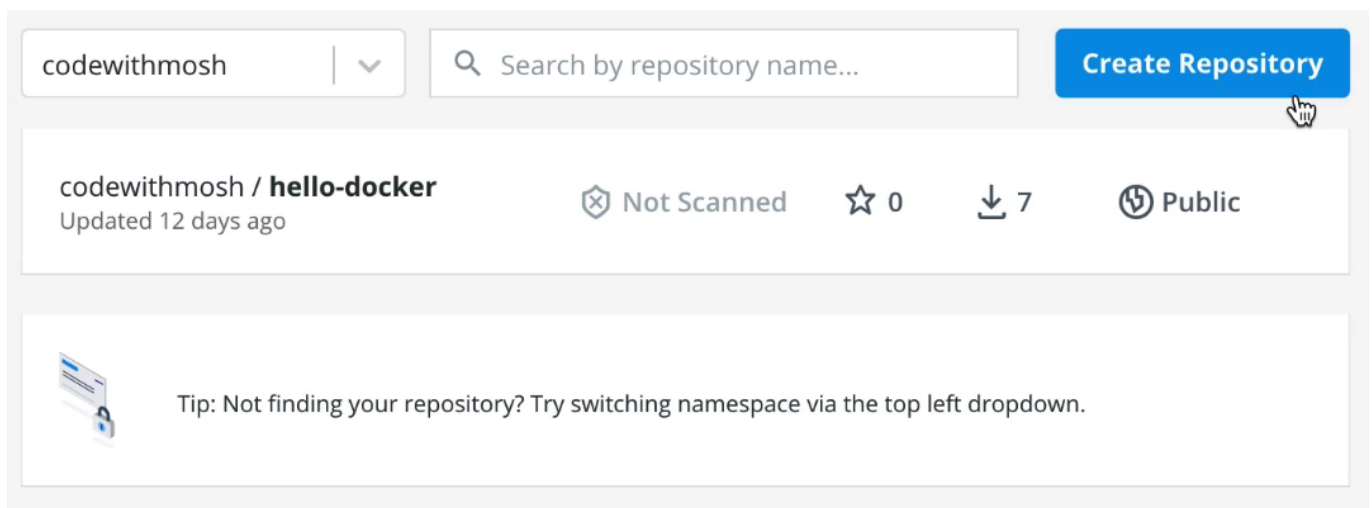
🔥 ~/mosh_docker_app4 main ± docker image tag 84ba63c80338 react-app:latest
🔥 ~/mosh_docker_app4 main ± docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
react-app 2 84ba63c80338 2 minutes ago 299MB
react-app latest 84ba63c80338 2 minutes ago 299MB
react-app 1 132c65c38ce8 16 minutes ago 299MB
<none> <none> 89802f797aad 16 hours ago 299MB
ubuntu latest ba6accedd29 2 months ago 72.8MB

🔥 ~/mosh_docker_app4 main ±

```

Sharing images

To share images on DockerHub, first create a repository (just as you would on GitHub). In this repository you can have multiple images with different tags.



Give it a new and description, and set its visibility.

Create Repository

codewithmosh



react-app

Description



Visibility

Using 0 of 1 private repositories. [Get more](#)

**Public**

Public repositories appear in Docker Hub search results

**Private**

Only you can view private repositories

Next we can connect the DockerHub repository to a GitHub repository. If we do that, every time we new code is included in the GitHub repository, DockerHub will pull the code and build a new image.

Build Settings *(optional)*

Autobuild triggers a new build with every **git push** to your source code repository. [Learn More.](#)

Please re-link a GitHub or Bitbucket account



We've updated how Docker Hub connects to GitHub and Bitbucket. You'll need to re-link a GitHub or Bitbucket account to create new automated builds. [Learn More](#)



Disconnected



Disconnected

To push the image to the repository, we have to give it that name locally too. To do that we just change the tag:

```
docker image tag IMAGE_ID NEW_TAG
```

Only that now `IMAGE_TAG` will be of the form `DOCKERHUB_USERNAME/REPOSITORY_NAME:TAG_NAME`.

In order to push the image, first we need to login. To do so we run:

```
docker login
```

If you are already logged into Docker Desktop, the CLI will use your credentials from there. Otherwise, it will ask you for them.

To push the image to the repo we just run:

```
docker push IMAGE_TAG
```

where once again `IMAGE_TAG` will be of the form `DOCKERHUB_USERNAME/REPOSITORY_NAME:TAG_NAME`. Docker will push the image one layer at a time. Therefore, changes to dependencies might cause this process to be a little bit slower.

Now that the image is on DockerHub, we can pull it from any other machine by running:

```
docker pull IMAGE_TAG
```

Saving and loading images

If you have an image on your computer and want to port it to another machine but without using DockerHub, you can save it in a compress file in your machine, port it, and then load it in the other machine.

To save we run:

```
docker image save -o OUTPUT_FILE IMAGE
```

where `OUTPUT_FILE` is usually a `tar` file. The result will be a `tar` file on your computer. You can unzip it to inspect it. Inside of it there will be folders, each representing a layer. Each layer contains a `json` file and another `tar` file, which itself contains all the files in that layer.

To load the image we use:

```
docker image load -i FILE_NAME
```

where `FILE_NAME` is a `tar` file containing the compressed version of our image.