

Cloud Engineering Principles and Practices

Build Your Own Docker Image and Push It to Docker Hub

A step-by-step tutorial and lab

This lab continues from **Your First Docker Containers**. You have already run containers from existing images. In this lab you will create a simple web application, describe it in a Dockerfile, build your own image, run and test it, then publish the image to Docker Hub so that it can be pulled and run on another machine.

By the end of the lab you should be able to explain what a **Dockerfile** is, build an image from a build context, apply repository names and tags, authenticate to Docker Hub, push an image to a registry, and verify that the published image can be pulled and run.

You will then build a small **Python** application, as in the lecture, and use it to inspect image layers and metadata, observe the build cache, keep the build context clean with `.dockerignore`, run as a non-root user, and reduce image size with a multi-stage build.

Part 0 Getting ready

1. Start Docker Desktop and wait until the Docker engine reports that it is running.
2. Open PowerShell/Windows Terminal on Windows, or Terminal on macOS/Linux.
3. Make sure you have a Docker Hub account. You will need your Docker Hub **username** in Parts 5 and 6.
4. Check that Docker is working:

```
docker --version
```

5. Check that you can list local images:

```
docker image ls
```

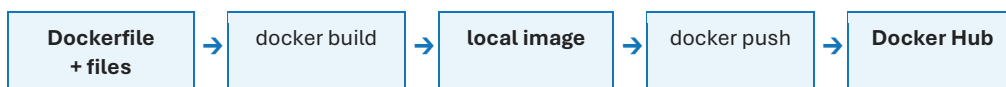
Before you publish anything

- Use a **public test repository** only if you are comfortable with its contents being visible to others.
- Do not put passwords, API keys, access tokens, private certificates or other secrets in the Dockerfile or application files.
- Use a simple project name. In the examples below, replace `YOUR_DOCKERHUB_USERNAME` with your own Docker Hub username.

Part 1 From a Dockerfile to Docker Hub

1.1 Concept

Until now you have used images that someone else built, such as `nginx` and `alpine`. A **Dockerfile** is a text file containing the instructions Docker uses to build a new image. The files sent to Docker during the build are called the **build context**.



The build produces an image on your computer. You can run containers from that image immediately. To share the image, give it a Docker Hub-compatible name and push it to the registry.

A Docker Hub image name normally has this form:

```
USERNAME/REPOSITORY:TAG
```

Part	Example	Meaning
USERNAME	alexstudent	Your Docker Hub namespace
REPOSITORY	student-web	The repository that holds the image
TAG	v1	A version label for the image

For example, if the Docker Hub username is [alexstudent](#), the complete image name could be [alexstudent/student-web:v1](#).

Exercise 1: identify the image name

1. Write down your Docker Hub username.
2. Choose a repository name such as [student-web](#).
3. Write the complete image name you intend to publish: [USERNAME/student-web:v1](#).
4. Keep this name. You will use exactly the same spelling later when you tag and push the image.

Part 2 Create a small web application

2.1 Create the project folder

Create a new folder for the lab and move into it. The folder name is not important, but the examples use [student-docker-site](#).

```
mkdir student-docker-site
cd student-docker-site
```

Inside this folder create a file called [index.html](#). You can use Notepad, VS Code, nano or any other text editor. Put the following content in it, replacing the sample name with your own name or initials.

```
<!doctype html>
<html>
<head><title>My Docker Site</title></head>
<body>
  <h1>Hello from my own Docker image</h1>
  <p>Created by Student Name</p>
</body>
</html>
```

2.2 Create the Dockerfile

In the same folder create a file named exactly [Dockerfile](#) with **no file extension**. Add these instructions:

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
EXPOSE 80
```

Instruction	What it does
FROM nginx:alpine	Starts with the small nginx image as the base image.
COPY ...	Copies your web page into the location served by nginx.
EXPOSE 80	Documents that the application inside the image listens on port 80.

Two things this Dockerfile does not show

EXPOSE only documents the port. It does not make the container reachable from your computer. Publishing happens at run time with `-p HOST:CONTAINER`.

There is no CMD line. Your image inherits the default command from the `nginx:alpine` base image. You will check this in Exercise 3. In Part 7 you will write your own `CMD`.

Important: Dockerfile, not Dockerfile.txt

Windows may hide file extensions. If the file is actually called `Dockerfile.txt`, Docker will not find it when you use the normal build command. In File Explorer, enable **View > Show > File name extensions** if you are unsure.

Exercise 2: check the build context

1. Confirm that `index.html` and `Dockerfile` are in the **same folder**.
2. Display the folder contents with `dir` on Windows or `ls` on macOS/Linux.
3. Open the Dockerfile and confirm that the first line is exactly `FROM nginx:alpine`.
4. Do not continue until both files are present.

Part 3 Build your image

3.1 Build command

Run the build command **from the project folder**. The final dot is important: it tells Docker to use the current folder as the build context.

```
docker build -t student-web:v1 .
```

`-t student-web:v1` gives the new image a local name and tag. During the first build Docker may download the `nginx:alpine` base image. Later builds can reuse cached layers and may finish much faster.

When the build finishes, list your images:

```
docker image ls
```

You should see an image named `student-web` with the tag `v1`.

3.2 Read the build output

The build output is useful. It shows Docker processing the instructions in the Dockerfile. If a step fails, the last successful step and the error immediately after it usually tell you where to look.

Exercise 3: build and inspect

1. Build `student-web:v1`.
 2. List your images and locate the image name and tag, the image ID and the size. In recent Docker versions the name and tag are shown together in the `IMAGE` column, and the size appears as `DISK USAGE` and `CONTENT SIZE`.
 3. Run `docker image history student-web:v1` and look at the layers created by the base image and your Dockerfile.
 4. Edit the heading in `index.html`, rebuild with the same command, and notice which build steps are cached and which step is repeated.
 5. Run `docker image inspect student-web:v1 --format "{{.Config.Cmd}} {{.Config.ExposedPorts}}"` and confirm that the command and port were inherited from the base image.
- Note: this Dockerfile has only one step of its own, so the cache effect is small here. Part 7 shows it much more clearly.

Part 4 Run and test your image

Run a container from your own image. Use host port 8080 and container port 80:

```
docker container run -d --name my-student-site -p 8080:80 student-web:v1
```

Open <http://localhost:8080> in a browser. You should see the page you placed in [index.html](#).

Check that the container is running:

```
docker container ls
```

If the page does not load, inspect the logs:

```
docker container logs my-student-site
```

Exercise 4: prove that your image works

1. Start the container and open the web page in a browser.
2. Confirm that the page contains your own heading or name/initials.
3. Run `docker container ls` and identify the image name and port mapping.
4. Stop and remove the test container before moving to Part 5: `docker container rm -f my-student-site`.

Part 5 Name and push the image to Docker Hub

5.1 Tag the local image

Docker Hub uses your username as the namespace. Tag the working local image with the complete Docker Hub name. Replace `YOUR_DOCKERHUB_USERNAME` before running the command.

```
docker tag student-web:v1 YOUR_DOCKERHUB_USERNAME/student-web:v1
```

List the images again:

```
docker image ls
```

You will normally see two names pointing to the same image ID: `student-web:v1` and `YOUR_DOCKERHUB_USERNAME/student-web:v1`. Tagging does **not** duplicate the image data; it adds another name for the same image.

5.2 Sign in to Docker Hub

Sign in using Docker Desktop or the Docker CLI. With the CLI:

```
docker login
```

Follow the authentication instructions shown by Docker. Depending on your setup, Docker may use browser/device authentication or ask for credentials. **Never put your password or access token inside the Dockerfile or in a command that you plan to submit as evidence.**

Recommended: create a **personal access token** (select your avatar in Docker Hub, then Account settings > Personal access tokens > Generate new token) and use it instead of your account password, for example `docker login -u YOUR_DOCKERHUB_USERNAME`, then paste the token when asked for a password.

5.3 Push the image

Push the tagged image to Docker Hub:

```
docker push YOUR_DOCKERHUB_USERNAME/student-web:v1
```

Docker uploads the layers that the registry does not already have. When the push finishes, open Docker Hub in a browser and check your repositories. You should see [student-web](#) with the tag [v1](#).

If the repository does not exist yet, [docker push](#) creates it automatically using your account's **default repository privacy** (My Hub > Settings > Default privacy). Check that setting, or create the repository first (My Hub > Repositories > Create repository) and choose Public or Private, as shown in the lecture.

5.4 Add the latest tag

Tags are labels. You can give the same image another tag called [latest](#):

```
docker tag student-web:v1 YOUR_DOCKERHUB_USERNAME/student-web:latest
docker push YOUR_DOCKERHUB_USERNAME/student-web:latest
```

Now the Docker Hub repository should contain both [v1](#) and [latest](#). In a real project, versioned tags help you identify exactly which image was deployed.

Exercise 5: publish your image

1. Tag your image using your real Docker Hub username.
2. Run [docker login](#) and complete the authentication process.
3. Push the [v1](#) tag.
4. Create and push the [latest](#) tag.
5. Open Docker Hub and confirm that the repository and both tags are visible.
6. Copy the full Docker Hub image name. A classmate should be able to use that name without receiving your project files.

Part 6 Verify the published image

6.1 Pull the image by its Docker Hub name

The purpose of a registry is to let another Docker host obtain your image. Ask a classmate to pull your image, or test from another machine if one is available:

```
docker pull YOUR_DOCKERHUB_USERNAME/student-web:v1
```

Then run it on a free host port, for example 8085:

```
docker container run -d --name pulled-site -p 8085:80 YOUR_DOCKERHUB_USERNAME/student-web:v1
```

Open <http://localhost:8085>. The same page should appear even though the machine only received the image from Docker Hub.

What this demonstrates

The source folder and Dockerfile were needed to **build** the image, but they are not needed to **run** a container once the image has been published. The registry stores and distributes the image. A Docker host can pull the image and create containers from it.

Mac with Apple silicon? Check the image platform

Images built on an Apple silicon Mac are `linux/arm64` by default. A classmate on a Windows or Intel PC may then see `no matching manifest for linux/amd64` when pulling.

Fix: build and push a multi-platform image with `docker buildx build --platform linux/amd64,linux/arm64 -t YOUR_DOCKERHUB_USERNAME/student-web:v1 --push`.

If Docker reports that multi-platform builds are not supported, run `docker buildx create --use` once and try again.

Exercise 6: verification

1. Share only your Docker Hub image name with a classmate.
2. Ask them to pull `:v1` and run it on an unused host port.
3. Confirm that your personalised page appears.
4. On your own machine, first delete the local copies with `docker image rm YOUR_DOCKERHUB_USERNAME/student-web:v1 student-web:v1`, then run `docker pull YOUR_DOCKERHUB_USERNAME/student-web:v1`. Docker must now download the layers from Docker Hub, which proves the registry copy works.
5. Remove any test containers when finished.

Part 7 A Python app: layers, cache and multi-stage builds

So far your Dockerfile has used only `FROM`, `COPY` and `EXPOSE`. In this part you build the Python example from the lecture and use it to practise `WORKDIR`, `RUN`, `USER` and `CMD`, the build cache, `.dockerignore` and multi-stage builds.

7.1 Create the project files

Create a new folder next to your first project and move into it:

```
mkdir student-python-app
cd student-python-app
```

Create `app.py` (replace the sample name with your own name or initials):

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "<h1>Hello from my Python Docker image</h1><p>Created by Student Name</p>"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

Create `requirements.txt`. Pinning the version keeps builds repeatable:

```
flask==3.1.0
```

Create `.dockerignore`. Files listed here are **not sent** in the build context, so they can never be copied into the image:

```
__pycache__/
*.pyc
.venv/
.git/
.env
```

Finally, create a file called `.env` containing a pretend secret. You will use it to prove that `.dockerignore` works:

```
SECRET=do-not-ship-this
```

7.2 Write the Dockerfile

Create [Dockerfile](#). It is the lecture example with a non-root user added:

```
FROM python:3.13-slim
WORKDIR /usr/local/app

# Install dependencies first (better cache)
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the source
COPY ..

# Run as a non-root user
RUN useradd --create-home appuser
USER appuser

EXPOSE 8000
CMD ["python", "app.py"]
```

Instruction	What it does
WORKDIR /usr/local/app	Sets the folder used by the following instructions and by the running container.
COPY requirements.txt ./	Copies only the dependency list first, so the next step can be cached.
RUN pip install ...	Runs a command at build time and saves the result as a new layer.
COPY ..	Copies the rest of the build context, except anything in .dockerignore .
USER appuser	Runs the application as a normal user instead of root, which limits damage if the app is compromised.
CMD [...]	Sets the default command that runs when the container starts .

7.3 Build, run and look inside

```
docker build -t student-py:v1 .
docker run --rm -d --name py-test -p 8000:8000 student-py:v1
```

Open <http://localhost:8000>. [--rm](#) means the container is removed automatically when it stops. Now open a shell inside the running container:

```
docker exec -it py-test sh
```

Inside the container, run [whoami](#) (it should print [appuser](#)) and [ls -a](#) (there should be no [.env](#) file). Type [exit](#) to leave, then stop the container with [docker stop py-test](#).

7.4 Inspect layers and metadata

```
docker image history student-py:v1
docker image inspect student-py:v1 --format "{{.Config.User}} {{.Config.WorkingDir}} {{.Config.Cmd}}"
```

[docker image history](#) shows which instruction created each layer and how large it is. The [pip install](#) layer should be one of the largest layers that your Dockerfile adds.

Exercise 7: observe the build cache

1. Change the heading in `app.py` and rebuild: `docker build -t student-py:v2 .`
2. In the build output, find the `pip install` step. It should be marked **CACHED**, while `COPY ..` and the steps after it run again. Tip: add `--progress=plain` to see every step.
3. Copy `Dockerfile` to `Dockerfile.bad` and change the order so that `COPY ..` comes before `RUN pip install`.
4. Build it with `docker build -f Dockerfile.bad -t student-py:bad .`, edit `app.py` again, and rebuild with the same command.
5. Notice that `pip install` now runs again on every code change. Write one sentence explaining why, using the words **layer** and **cache**.

7.5 Separate building from running with a multi-stage build

A multi-stage build uses one stage to **build** things and a second, clean stage to **run** them. Only what you copy with `COPY --from` reaches the final image. Create `Dockerfile.multi`:

```
# Stage 1: install dependencies using the full Python image
FROM python:3.13 AS builder
WORKDIR /build
COPY requirements.txt .
RUN python -m venv /opt/venv && /opt/venv/bin/pip install --no-cache-dir -r requirements.txt

# Stage 2: small runtime image
FROM python:3.13-slim
WORKDIR /usr/local/app
COPY --from=builder /opt/venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
COPY . .
RUN useradd --create-home appuser
USER appuser
EXPOSE 8000
CMD ["python", "app.py"]
```

Build it, then download the builder base image so that it appears in your image list for comparison. BuildKit does not list base images used during a build, so the `docker pull` is needed.

```
docker build -f Dockerfile.multi -t student-py:multi .
docker pull python:3.13
docker image ls
```

What to expect

The builder base image `python:3.13` is well over 1 GB, while `student-py:multi` is about the same size as `student-py:v1`, because both are built on the slim image.

Flask is pure Python and needs no compiler, so here the multi-stage build mainly keeps build tools out of the runtime image. The size saving becomes large when a build needs compilers, header files or a full SDK, for example packages with C extensions, or Java and Go applications.

Exercise 8: optimise and publish

1. Run `student-py:multi` on port 8000 and confirm the page still works.
2. Use `docker image ls` to compare the sizes of `python:3.13` (the builder stage), `student-py:v1` and `student-py:multi`. Explain why the final image is so much smaller than the builder image even though `pip` ran in the large image.
3. Tag the image as `YOUR_DOCKERHUB_USERNAME/student-py:v1` and push it to Docker Hub.
4. Remove the local copy, pull it back from Docker Hub and run it again.

Part 8 Final independent exercise

Complete this exercise **without copying the commands from a classmate**. Look back through the handout when you need a reminder.

Exercise 9: build and publish your own image

Create a new image that meets all of the following requirements:

- Use `nginx:alpine` as the base image.
- Create your own `index.html` containing a page title, your name or initials, and at least one sentence describing the module.
- Build the image with a local version tag such as `v1`.
- Run it locally and prove that the page is being served from your container.
- Publish the image to a repository in **your own Docker Hub namespace**.
- Push both a versioned tag (`v1`) and the `latest` tag.
- Verify the published image by pulling/running it from its Docker Hub name.
- Add a `.dockerignore` file (as in Part 7) and explain in one sentence what it keeps out of the build context.
- Keep the repository free of passwords, tokens, private keys and other secrets.

Evidence to retain

Unless your lecturer specifies a different submission format, retain the following evidence so that you can demonstrate the complete workflow:

- The contents of your `Dockerfile`.
- The successful `docker build` output or the resulting entry in `docker image ls`.
- Your web page running in a browser from the local container.
- The Docker Hub repository page showing your repository and tags.
- The full image name in the form `USERNAME/REPOSITORY:TAG`.
- Evidence that the published image can be pulled and run.
- The output of `docker image history` for one of your images.
- Build output from Exercise 7 showing a **CACHED** step, and your sentence explaining the cache.
- The `docker image ls` sizes you compared in Exercise 8.

Quick knowledge check

1. What is the difference between a Dockerfile and a Docker image?
2. What does the final dot mean in `docker build -t student-web:v1 .`?
3. Why must a Docker Hub image name include your username/namespace?
4. Does `docker tag` create a second copy of all the image data?
5. What is the difference between `docker push` and `docker pull`?
6. Why should secrets never be copied into a Docker image?
7. If `USERNAME/student-web:v1` and `USERNAME/student-web:latest` have the same image ID, what does that tell you?
8. What is an image layer, and which Dockerfile instructions create the largest ones?
9. Why does copying `requirements.txt` before the rest of the source code make rebuilds faster?
10. What is the difference between `.dockerignore` and deleting a file with `RUN rm` later in the Dockerfile?
11. In a multi-stage build, which parts of the builder stage end up in the final image?

Troubleshooting

Problem

What to try

Problem	What to try
Cannot connect to the Docker daemon or error during connect	Docker Desktop is not running. Start it and wait until the engine reports that it is running.
failed to read dockerfile or no such file	You are in the wrong folder, or the file is called <code>Dockerfile.txt</code> . Check with <code>dir</code> or <code>ls</code> .
port is already allocated	Another container uses that host port. Use a different host port, or find and remove the old container with <code>docker container ls</code> .
push access denied or requested access to the resource is denied on push	The username in the tag does not match the account you logged in with, or you are not logged in. Run <code>docker login</code> and re-tag the image.
no matching manifest for linux/amd64	The image was built for a different CPU architecture. See the Apple silicon box in Part 6.
The page does not change after a rebuild	The old container is still running. Remove it and start a new container from the new tag.
Python app: page not reachable on port 8000	The app must listen on <code>0.0.0.0</code> , not <code>127.0.0.1</code> , and the container must be started with <code>-p 8000:8000</code> .

Clean up

When you have collected your evidence, remove the test containers and images, and sign out, especially on a shared lab computer:

```
docker container rm -f my-student-site pulled-site py-test
docker image rm student-web:v1 student-py:v1 student-py:v2 student-py:bad student-py:multi
docker image rm YOUR_DOCKERHUB_USERNAME/student-web:v1 YOUR_DOCKERHUB_USERNAME/student-web:latest
YOUR_DOCKERHUB_USERNAME/student-py:v1
docker logout
```

`docker image rm` reports `No such image` for any name you have already removed. That is expected and safe to ignore. The images stay on Docker Hub. Only your local copies are deleted.

Reference

Dockerfile reference: <https://docs.docker.com/reference/dockerfile/>

Understanding image layers: <https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/>

Using the build cache: <https://docs.docker.com/get-started/docker-concepts/building-images/using-the-build-cache/>

Multi-stage builds: <https://docs.docker.com/get-started/docker-concepts/building-images/multi-stage-builds/>

Build, tag and publish an image: <https://docs.docker.com/get-started/docker-concepts/building-images/build-tag-and-publish-an-image/>

Docker Hub, pushing images to a repository: <https://docs.docker.com/docker-hub/repos/manage/hub-images/push/>

Dockerfile best practices: <https://docs.docker.com/build/building/best-practices/>