Lab 2 Report

# Publishing and exposing ports

## Publishing Ports

* A port is an endpoint for communication
* Publishing a port is a method of mapping an internal port to an external port

A screenshot of a computer

AI-generated content may be incorrect.

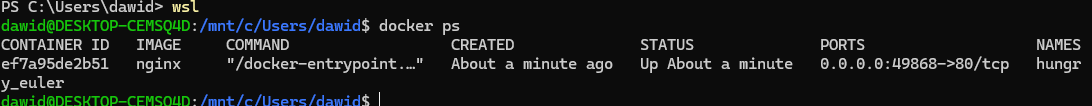
* Above, I publish an 80 nginx port to an 8080 port.

## Publishing to ephemeral ports

* Publishing to ephemeral ports just means you don’t care which external port is chosen during the mapping process

A screenshot of a computer

AI-generated content may be incorrect.



* Above you can see port 80 nginx being assigned to a random external port.

## Publishing all ports

* Container image is a package containing everything needed to run a piece of software.

A screenshot of a computer

AI-generated content may be incorrect.

* Above, I publish all exposed ports to ephemeral ports.

## Try it out

1. Installed docker desktop
2. Started new container called docker/welcome-to-docker

A screen shot of a computer

AI-generated content may be incorrect.

1. Viewed docker dashboard to see if the new container was up.



1. Went to containers website hosted on 8080

A screenshot of a computer

AI-generated content may be incorrect.

## Use Docker Compose

* Docker Compose is a type of tool that you can use to automate certain tasks within docker.
* You can create a compose.yaml files which configures certain values for your containers.

1. Created a compose.yaml file

A screen shot of a computer

AI-generated content may be incorrect.

1. Directed to the directory containing this file
2. Used docker compose up

A computer screen with text

AI-generated content may be incorrect.

1. Opened webpage

A screenshot of a computer

AI-generated content may be incorrect.

# Overriding container defaults

## Overriding the network ports

* If you want to run separate database instances, you can use -p option on docker run

## Setting environment variables

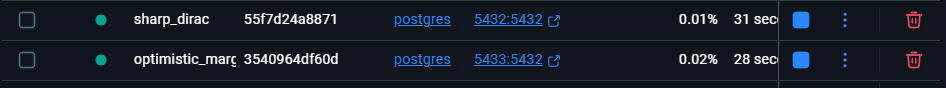
* You can set environment variables using 2 methods
  + docker run -e foo=bar postgres env
  + set env files
    - docker run --env-file .env postgres env

## Restricting the container to consume the resources

* You can control how much memory and CPU a container uses.
* Example:
  + docker run -e POSTGRES\_PASSWORD=secret --memory="512m" --cpus="0.5" postgres
  + POSTGRES\_PASSWORD environmental variables set
  + Memory set to 512m
  + CPU set to 0.5

## Run multiple instances of postgres database

1. Started a container using the PostgreSQL image
2. Started a second PostgreSQL container mapped to a different port
3. Verified both containers are up



Postgres is mapped to 2 different ports on the machine

## Run Postgres container in a controlled network

* By default, containers connect to the bridge network.
* A bridge network is a type of connection in that allows containers on the same host to communicate with each other like above.
* You can create a custom network using –network.

1. Created new custom network
2. Viewed the new network

A black background with white text

AI-generated content may be incorrect.

1. Connected postgres to the new custom network

A screenshot of a computer program

AI-generated content may be incorrect.

Above, you can see what network its connected to.

## Manage the resources

* Essentially the same section as “Restricting the container to consume the resources”.

## Override the default CMD and ENTRYPOINT in Docker Compose

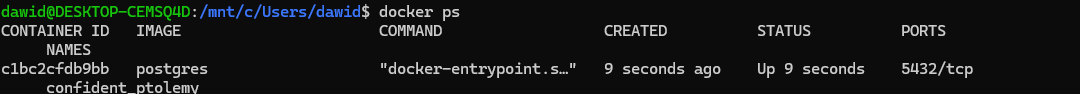
1. Created a compose file
2. Brought up the service
3. Verified the authentication

A screenshot of a computer

AI-generated content may be incorrect.

## Override the default CMD and ENTRYPOINT with docker run

* Here I ran this command:
  + docker run -e POSTGRES\_PASSWORD=secret postgres docker-entrypoint.sh -h localhost -p 5432
  + Below you can see the result of it



* Here's how it breaks down:
  + POSTGRES\_PASSWORD is a set environmental variable
  + docker-entrypoint.sh -h localhost -p 5432 defaults cmd is overwritten
    - Basically, what happens you change the default command this container runs when it starts

# Persisting container data

## Container volumes

* Volumes in docker are areas where your container can access and store data.
* docker run -d -p 80:80 -v log-data:/logs docker/welcome-to-docker
  + This command creates a volume called log-data for the welcome-to-docker container.

## Managing volumes

* There are several ways to manage volumes some of the commands include
* Docker volume ls – lists all volumes
* Docker volume rm <volume-name-or-id> - removes a volume
* docker volume prune – remove all unused volumes

## Try it out – Use Volumes

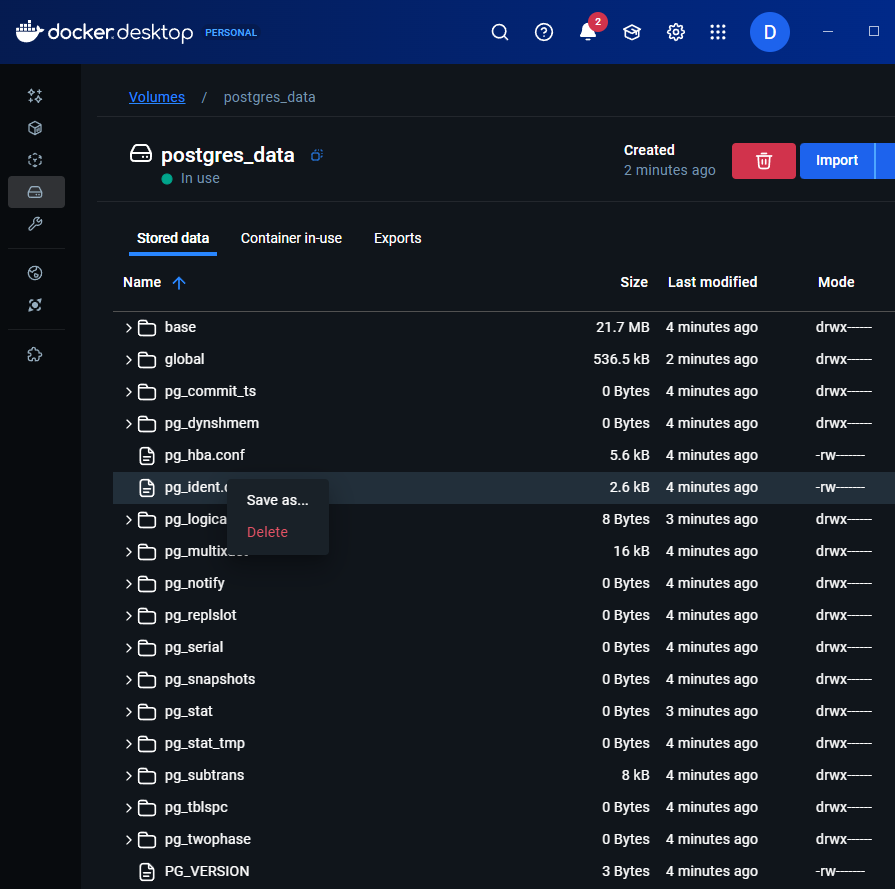
1. I already had preinstalled docker
2. Started a container using postgres image using
   1. docker run --name=db -e POSTGRES\_PASSWORD=secret -d -v postgres\_data:/var/lib/postgresql/data postgres
3. Connected to database using
   1. docker exec -ti db psql -U postgres
4. Created a database table and inserted 2 records.
5. Verified the data was in the database
6. Exited the PostgreSQL shell
7. Removed the database container
8. Started a new container attaching the same volume with persistent data
9. Verified if all the previous data was still there.

A screenshot of a computer program

AI-generated content may be incorrect.

## Try it out – View volume contents

1. Opened Docker Desktop on the volumes view.
2. Selected postgres\_data volume
3. Viewed the data tab
4. Saved files and deleted files



## Try it out – Remove volumes

* Before you remove a volume you need to make sure it’s not attached to any containers, you can do so by using:
  + docker rm -f new-db -> deattaches the container and removes it
* 2 ways to remove volumes:
  + docker volume rm postgres\_data
  + docker volume prune

# Sharing local files with containers

* Containers stored in isolation
* 2 solution:
  + Volumes
  + Bind mounts

## Volume versus bind mounts

* Volume -> data stays even if the container stops running
* Bind mounts -> direct link to a host path

## Sharing files between a host and a container

* -–volume for volume
  + docker run -v /HOST/PATH:/CONTAINER/PATH -it nginx
* --mount for bind mounts
  + docker run --mount type=bind,source=/HOST/PATH,target=/CONTAINER/PATH,readonly nginx
* They both share files or directories with your host

## File permissions for Docker access to host files

* Permission in docker must be expressed.
* Rw = read write access
* Ro = read only

## Try it out – Run a container

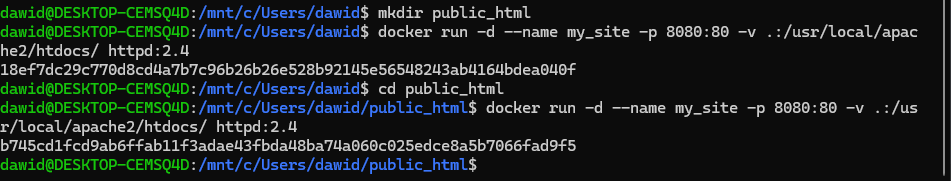
1. Docker Desktop is already pre-installed
2. docker run -d -p 8080:80 --name my\_site httpd:2.4 used to start a container
3. Opened localhost to verify

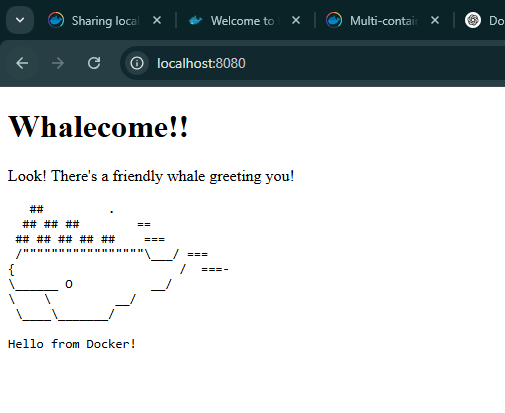
A screen shot of a computer

AI-generated content may be incorrect.

## Try it out – Use a bind mount

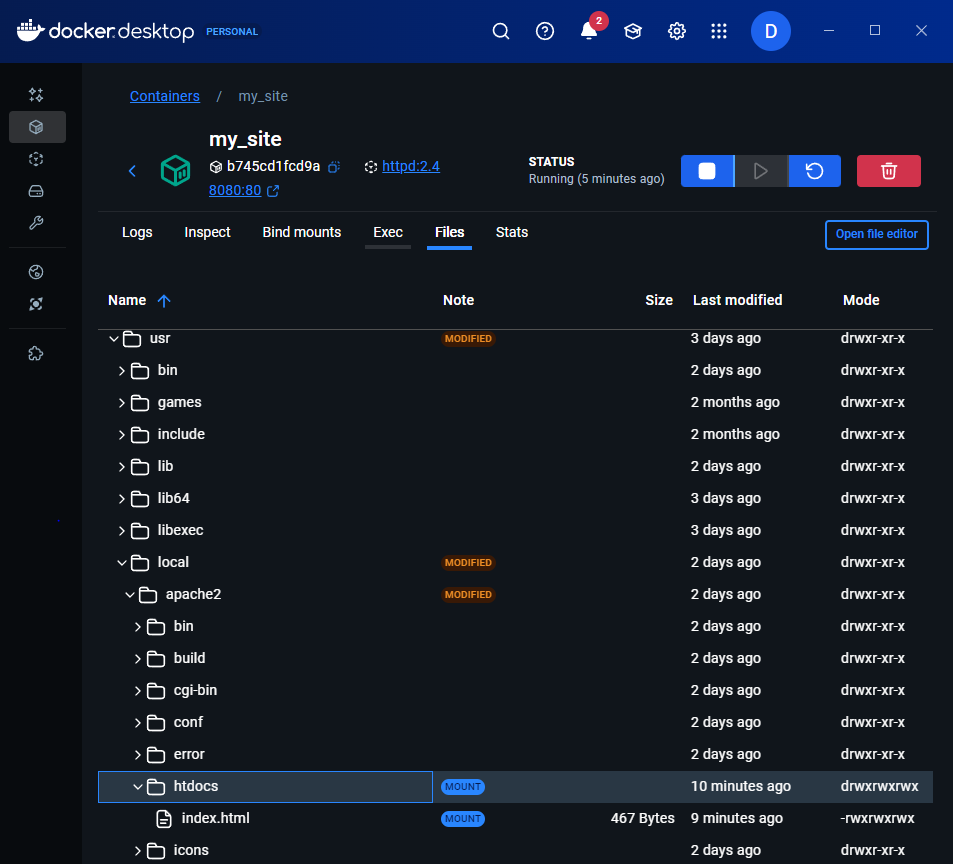
1. Deleted existing container my\_site
2. Created a new directory called public\_html
3. Created an index.html file in that directory
4. Ran the container with a bind mount
5. Verified if the website was up on localhost





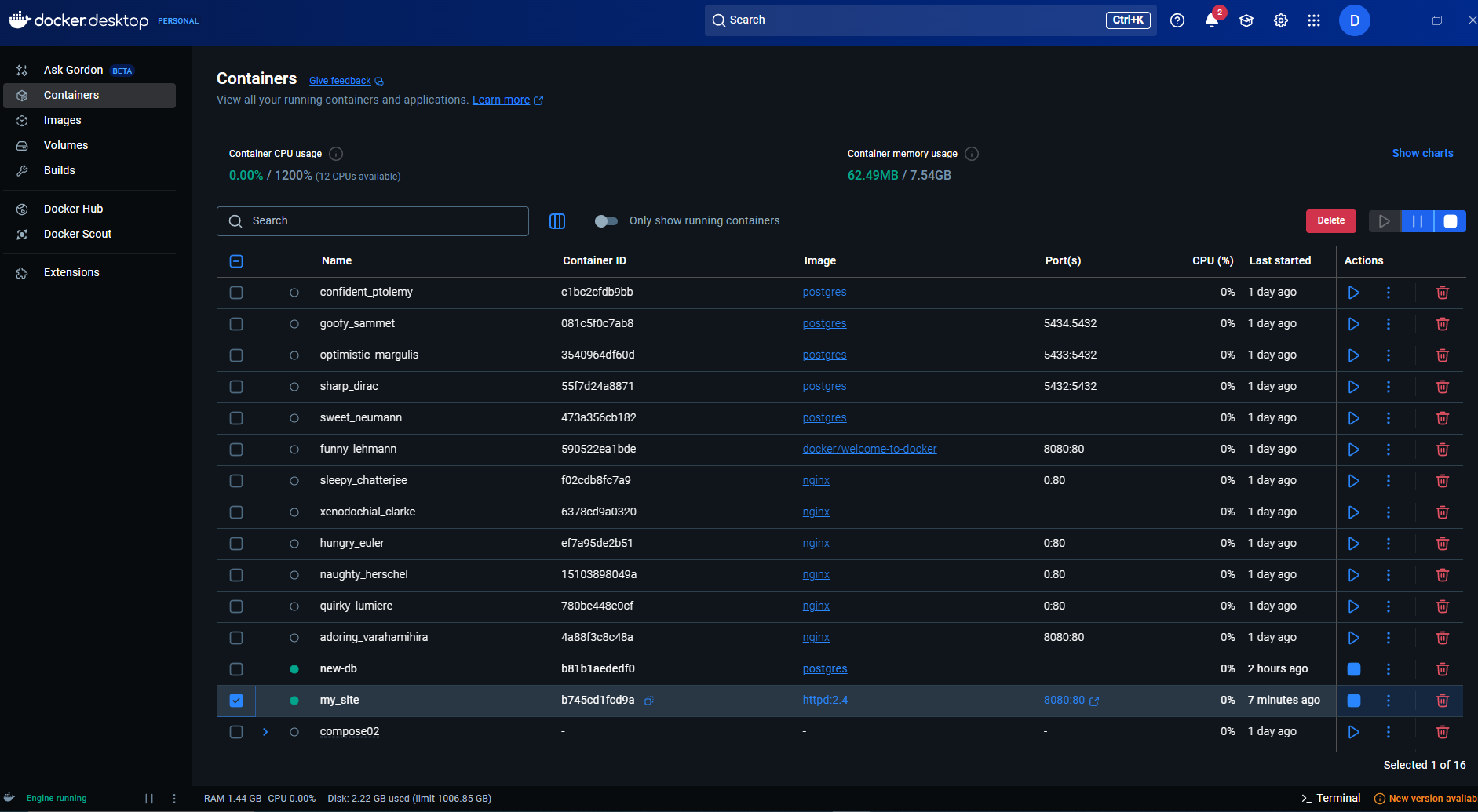
## Try it out – Access the file on the Docker Desktop Dashboard

1. Opened the files tab and selected /usr/local/apache2/htdocs/ directory. Then selected open file editor.
2. Deleted file on host to verify that it gets deleted on the container as well.
3. Recreated the file to see if the change is noticed on the docker desktop.



## Try it out – Stop your container

1. Went to the containers view on the desktop Docker.
2. Locate the container I’d like to stop, which is the my site container
3. Selected delete



# Multi-container applications

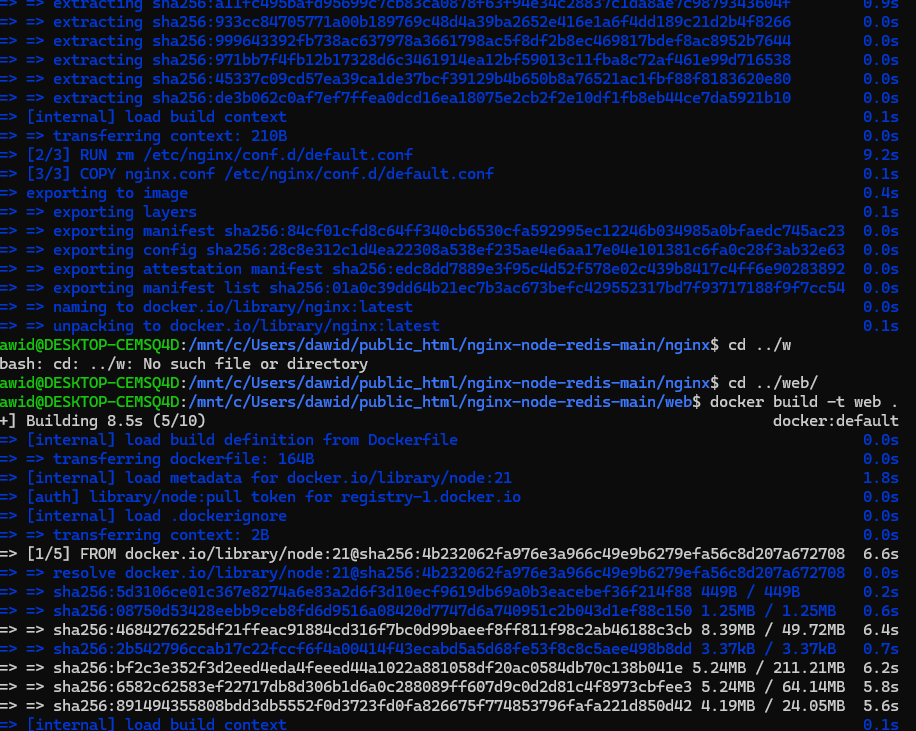
* Multiple containers cooperate to perform as a single unit
* Multiple containers are needed to combine multiple services for things like databases.
* Compose.yml used to run multiple Docker runs at once.
  + This allows you to configure a lot of settings.

## Try it out – Set up

1. Got the repo for this exercise
2. Had preinstalled docker

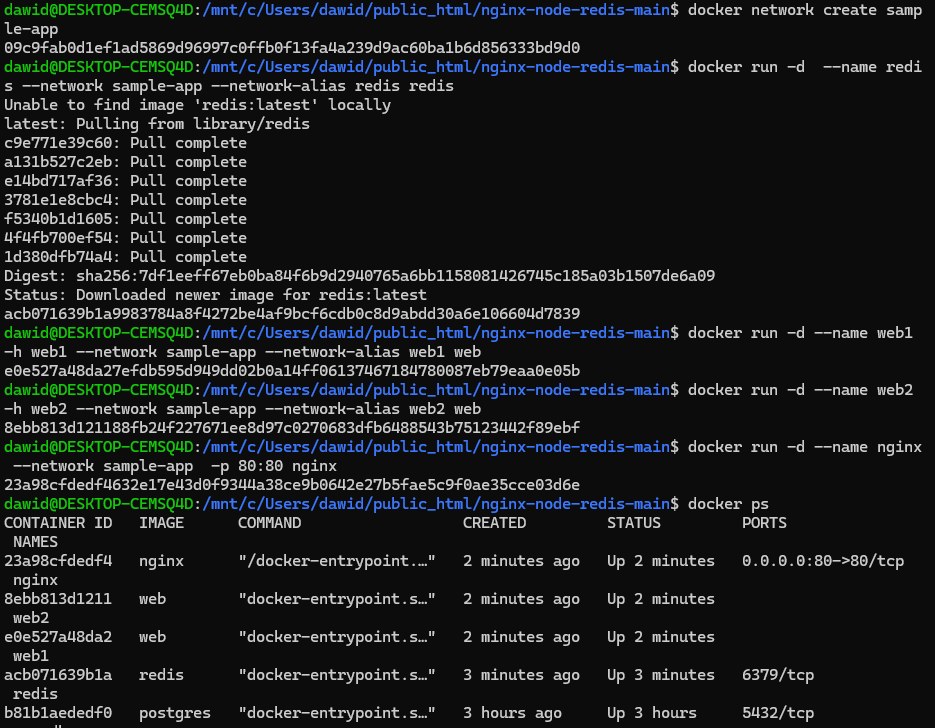
## Try it out – Build the images

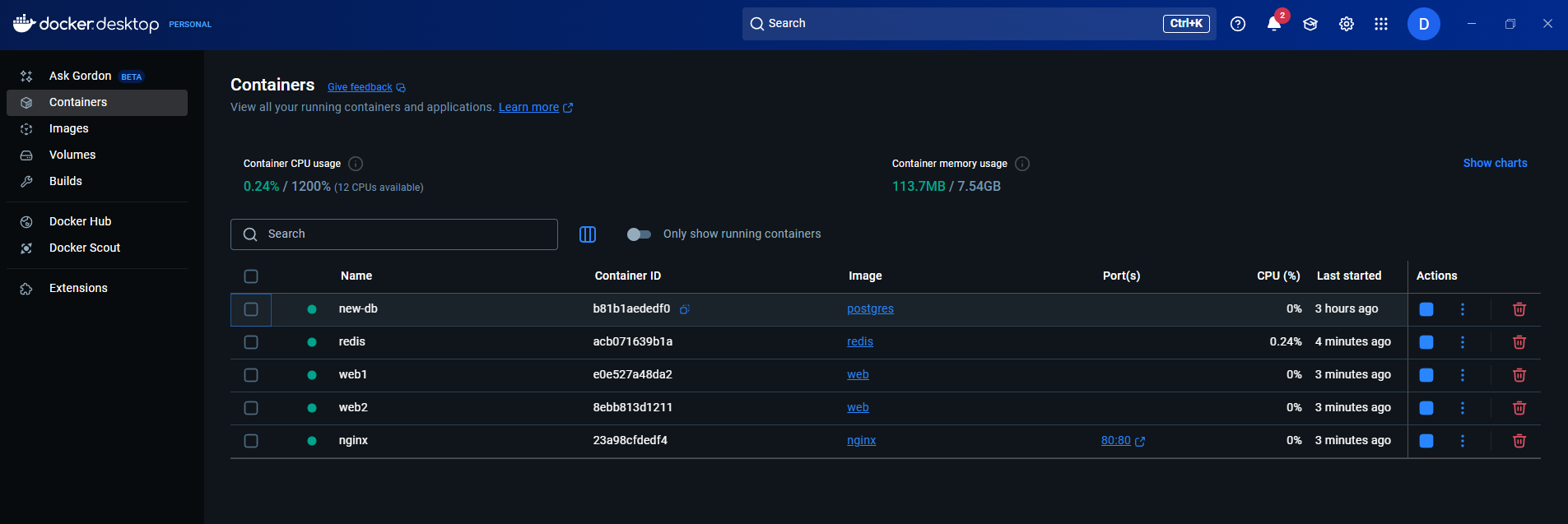
1. Build the images in the nginx directory
2. Built the first web image in the web directory

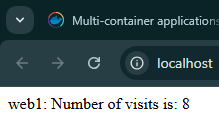


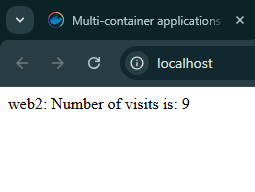
## Try it out – Run the containers

1. Created a network for the multi-container application
2. Started a Redis container, which attached to the previously connected network and created a network alias.
3. Started the first web application
4. Started the second web application
5. Started an nginx container
6. Verified if the containers are up and running with docker ps
7. Had a look at the containers on the Docker desktop dashboard
8. Went to localhost to view the results
9. Removed the containers using the desktop dashboard.









## Simplify the deployment using Docker Compose

1. Started the compose application
2. Checked whether the containers have been created on desktop dashboard
3. Deleted the containers

