Introduction to Docker

A quick introduction to Docker

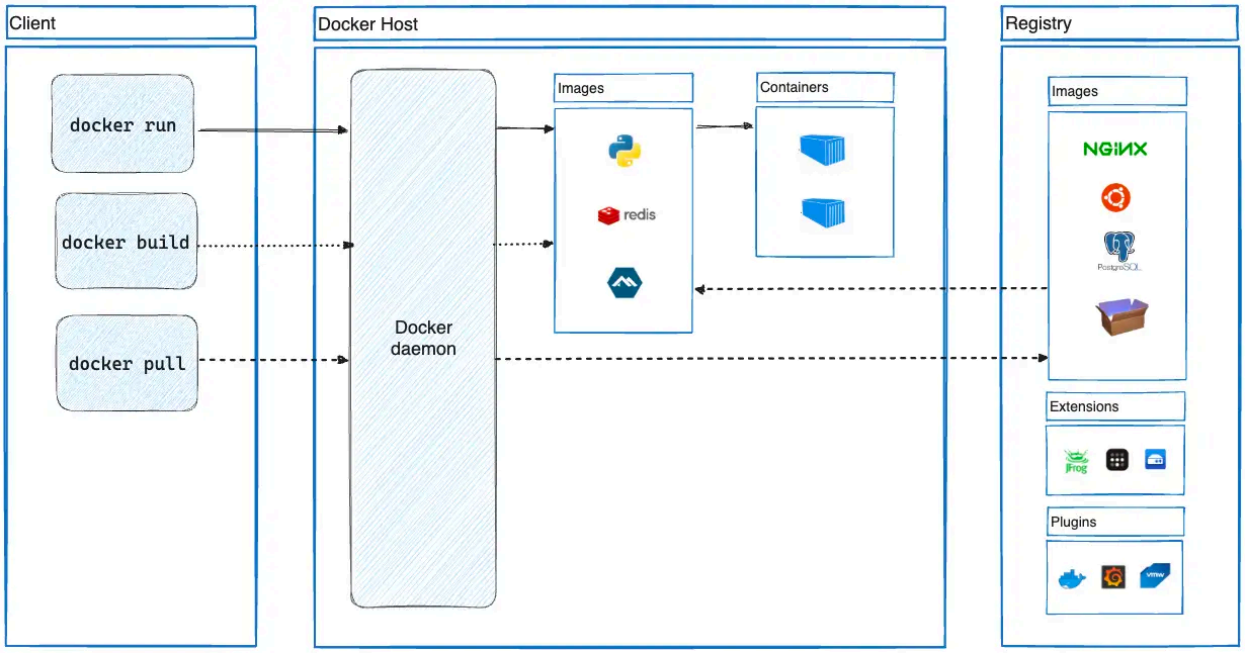
Docker is a set of tools implemented in Go that takes advantage of linux namespaces and cgroups in order to provide a process isolation environment so that applications can be deployed in a uniform way regardless of the host hardware and operating system they run on. All the required libraries and configuration get packaged together in a docker *image* and executed as a docker *container*. A competing and compatible technology is podman. For the purposes of this tutorial it is assumed that you have installed Docker Desktop in your computer.

> You can also use the Rancher Desktop and the docker (moby) container runtime. If you selected the more advanced containerd / nerdctl runtime, you will need to alias the `nerdctl` command to `docker`, or simply use `nerdctl` command instead of docker to follow along.

The docker architecture

The docker architecture comprises of three things:

- The `docker` command line tool (or other REST client),
- the docker daemon (which makes sure that what we request gets executed at the machine where it runs; which may or may not be the same as where the client runs: example when using Docker Desktop, the `docker` client runs on your machine,
- and the docker daemon runs in a VM in your machine). The this component is a registry, which is where the docker artifacts get pushed (stored) for further use and dissemination:



Running our first container

Let's run our first container, based on the smallest linux system available:

```
% docker run -it busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
c2bf9493c1bf: Pull complete
Digest: sha256:6d9ac9237a84afe1516540f40a0fafdc86859b2141954b4d643af7066d598b74
Status: Downloaded newer image for busybox:latest
/ # ls
bin   etc   lib   proc   sys   usr
dev   home   lib64   root   tmp var
/ # exit
```

What happened here? We requested via the `docker` client to run interactively (the `-it`) a busybox container. The system sees that it has no container image locally, and thus pulls it from the docker hub (the most widely used public registry) and then starts the container. Now we can check whether we have any images stored locally:

```
% docker images
REPOSITORY   TAG         IMAGE ID     CREATED      SIZE
busybox      latest3e4fd538a9a0   4 weeks ago   4.04MB
```

We can try and delete an image to free up some space:

```
% docker rmi busybox
Error response from daemon: conflict: unable to remove repository reference
"busybox" (must force) - container d42ca20e839e is using its referenced image
3e4fd538a9a0
```

We can't, as there is a container using that image. Which one?

```
% docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS     PORTS NAMES

% docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED        STATUS                    PORTS
NAMES
d42ca20e839e   busybox   "sh"        6 minutes ago   Exited (0) 6 minutes ago
      interesting_ritchie

% docker rm -f interesting_ritchie
interesting_ritchie

% docker rmi 3e4fd538a9a0
Untagged: busybox:latest
Untagged:
busybox@sha256:6d9ac9237a84afe1516540f40a0fafdc86859b2141954b4d643af7066d598b74
Deleted:
sha256:3e4fd538a9a0b729be05707cf805388be2fb701cfd5d44c6542f1988e8aef6e3
Deleted:
sha256:cd322088bcfba290bcfa064b77165d73708a0a77209146be150b29b7fec4c366
```

What happened here? `docker ps` shows the running containers in our system. We had exited our container, and thus no container is running. But that container, after exiting is still in our system left for further inspection. We can see it with `docker ps -a`. We see that docker produces helpful names for us to refer to the containers, in order to avoid remembering the container hashes. While `interesting_ritchie` is an acceptable hostname, `boring_wozniak` is not. We can remove the stopped container with `docker rm` and we can delete the image afterwards with `docker rmi` (and we can use the hash value instead of the tagged name if we like).

A more complex example

Let's run a simple nginx web server. After all, we run our containers in isolation, but we need a way to interact with them:

```
% docker run -d --rm --name nginx -p 3000:80 nginx
Unable to find image 'nginx:latest' locally
```

```
latest: Pulling from library/nginx
f546e941f15b: Pull complete
2d258780861a: Pull complete
a7d6e9feb830: Pull complete
42e0f9421c7a: Pull complete
14a95f763a2f: Pull complete
164c21b63fde: Pull complete
5b452a5fd809: Pull complete
Digest: sha256:c26ae7472d624ba1fafd296e73cecc4f93f853088e6a9c13c0d52f6ca5865107
Status: Downloaded newer image for nginx:latest
afb8530b625ad863923bd0e6456a4b75f37c7f96f21240265062d81a2e6daa15

% docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED      STATUS       PORTS
      NAMES
afb8530b625a   nginx     "/docker-entrypoint.…"   3 seconds ago   Up 2 seconds
0.0.0.0:3000->80/tcp   nginx
```

Let's see what we asked docker to do for us here: We asked it to run on our behalf a docker image (`nginx`) into a container and run it in the background (`-d`). We also asked it to name it with a specific name (`--name nginx`) and to delete the container from the system once its process is ended (`--rm`). Furthermore, we asked it to expose the (internal to the container) port 80/tcp (which is where nginx listens by default) to port 3000/tcp in our localhost. Thus:

```
% curl 127.0.0.1:3000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
```

```
</body>
</html>
```

Can we run a second container with the same name?

```
% docker run -it --rm --name nginx nginx
docker: Error response from daemon: Conflict. The container name "/nginx" is
already in use by container
"afb8530b625ad863923bd0e6456a4b75f37c7f96f21240265062d81a2e6daa15". You have to
remove (or rename) that container to be able to reuse that name.
See 'docker run --help'.
```

Obviously not. We need to figure out a second name.

```
% docker run -d --rm --name nginx1 -p 3000:80 nginx
496e30d60af9a540128bf6b1f295703c18af2f2e60037a6011a3afa6b787e6ee
docker: Error response from daemon: driver failed programming external
connectivity on endpoint nginx1
(c12b9dfaca18a161b53888099ae2ae9478193f2267be98666b5197e35bb1c4d3): Bind for
0.0.0.0:3000 failed: port is already allocated.
```

As the above example shows, while each container can listen to port 80/tcp (since they run on their own network namespace) they need to be exposed to a different port on localhost (so a `-p 3001:80` would make container nginx1 accessible at `curl 127.0.0.1:3001`)

Accessing the logs of a container

This is fairly easy:

```
% docker logs nginx
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to
perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching
/docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of
/etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in
/etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching
/docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching
/docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2024/02/16 18:54:11 [notice] 1#1: using the "epoll" event method
```

```
2024/02/16 18:54:11 [notice] 1#1: nginx/1.25.4
2024/02/16 18:54:11 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2024/02/16 18:54:11 [notice] 1#1: OS: Linux 6.6.12-linuxkit
2024/02/16 18:54:11 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2024/02/16 18:54:11 [notice] 1#1: start worker processes
2024/02/16 18:54:11 [notice] 1#1: start worker process 29
2024/02/16 18:54:11 [notice] 1#1: start worker process 30
192.168.65.1 - - [16/Feb/2024:18:58:23 +0000] "GET / HTTP/1.1" 200 615 "-"
"curl/8.4.0" "-"
```

For continuous monitoring of the logs we can run `docker logs -f`.

It has to be noted that in order for logs to be accessible this way, we need to write (or configure) our applications to send every logging information to STDOUT. Not via syslog, and definitely not in a logfile. Only STDOUT and STDERR gets collected by `docker logs [-f]`

How about getting into the container to check stuff?

It is possible to enter a container with

```
% docker exec -it nginx bash
root@afb8530b625a:/# cd /usr/share/nginx/html/
root@afb8530b625a:/usr/share/nginx/html# date > index.html
root@afb8530b625a:/usr/share/nginx/html#
exit
```

We asked docker to execute (`docker exec`) inside a container (`nginx`) a command (`bash`) which will take input from the keyboard (`-i`) and output in the terminal (`-t`, with `-i -t` abbreviated to `-it` in Unix fashion). We also changed the `index.html`file which can be verified:

```
% curl 127.0.0.1:3000
Fri Feb 16 19:24:41 UTC 2024
```

However, since containers are *ephemeral*, this change won't stay. If we remove and start again a container, it will respond with the usual nginx homepage.

How about some persistence then?

Docker offers us two ways for persistence. One is to mount a directory, or a file from the host operating system into the container (usually referred to as a bind mount). It also offers us a more sophisticated way via docker volumes (see the `docker volume -h` command). We will deal here only with bind mounts.

Suppose we write an `index.html`file with our own content that we want the nginx container to serve. After the file is prepared, we could run the following command:

```
% docker run -d --name adamo -p 3000:80 -v $(pwd)/index.
html:/usr/share/nginx/html/index.html nginx
1734b903e20ff9436d03b072ba6818e344481390105c367c4e868a4418b3c297

% curl 127.0.0.1:3000
Hello, World!
```

You can `docker exec` into the container and verify that indeed the contents are those of the `index.html` file from the host operating system. You can even change the file from the host operating system and the changes will be served.

Anything more about running containers?

Yes, you can inspect that state of a running, or a stopped container (to figure out why it died for example) using `docker inspect <container name>` and you will get back a huge JSON file with all sorts of useful information about the container.

You can also use `docker cp` to copy files to and from a container. Have a go with this.

All this is nice information, but how can I build my own container?

Docker (the company) has a very good tutorial about this, and we will make use of it here. You are advised to read their tutorial also. To build our own container, we first clone the application

```
git clone https://github.com/docker/docker-nodejs-sample
```

In order to build a container, we need to provide a `Dockerfile` in the directory `docker-nodejs-sample`. Let's see and discuss a minimal working directory

```
# Every container image, starts from a previous "base" container image.
# In Storfund we write mostly in nodejs and the current nodejs version
# that we use is 18.16.1
FROM node:18.16.1

# Set the NPM_TOKEN. If we want it set during the build process, use
# the --build-arg command line option for docker build
ARG NPM_TOKEN

# This is where the container starts, and where files are copied when
# no absolute path is mentioned.
WORKDIR /usr/app
```

```
# Let's all the files from the current directory into the container image
# If we have a .dockerignore file, the files mentioned in it are excluded.
# The first . in the COPY command is the current source code directory.
# The second . in the COPY command is the WORKDIR
COPY . .

# let's set an environment variable that will be set when the container runs.

ENV NODE_ENV production

# Install the npm modules
RUN npm install

# Now specify the command with which the container will start, unless we
# direct it otherwise. Despite what blog posts and other documentation says
# always use the bracket notation. The spaces between the brackets and the
# double quotes matter!
CMD [ "node", "src/index.js" ]
```

See here for [a complete Dockerfile description](). You are now ready to build your first container:

```
% docker build -t yiorgos/docker-nodejs-sample .
[+] Building 46.1s (10/10) FINISHED                    docker:desktop-linux
=> [internal] load build definition from Dockerfile                    0.0s
=> => transferring dockerfile: 1.20kB                                  0.0s
=> [internal] load metadata for docker.io/library/node:18.16.1         3.4s
=> [auth] library/node:pull token for registry-1.docker.io             0.0s
=> [internal] load .dockerignore                                       0.0s
=> => transferring context: 659B                                       0.0s
=> [1/4] FROM docker.io/library/node:18.16.1@sha256:f4698d49371c8a9fa7d  39.5s
=> => resolve docker.io/library/node:18.16.1@sha256:f4698d49371c8a9fa7dd  0.0s
=> => sha256:f4698d49371c8a9fa7dd78b97fb2a532213903066e4 1.21kB / 1.21kB  0.0s
=> => sha256:8e80a3bf661ba38d6d8b92729057dddd71531831f77 7.26kB / 7.26kB  0.0s
=> => sha256:42cbebf8bc116ba1aed7897e2d0566bf49da9d5c 49.57MB / 49.57MB  13.6s
=> => sha256:356172c718acf9930d9465b170864319079e2d2e 63.98MB / 63.98MB  13.6s
=> => sha256:9ffaff1b88cf5635b1b79123c9c0bbf79813da17ec6 2.00kB / 2.00kB  0.0s
=> => sha256:9a0518ec57568a70561f7c04650f9554c88dada97 23.57MB / 23.57MB  7.7s
=> => sha256:dddcd3ceb9980caac379e8f8eeafe1e901f017 202.40MB / 202.40MB  34.1s
=> => extracting sha256:42cbebf8bc116ba1aed7897e2d0566bf49da9d5c70be71b6  1.2s
=> => sha256:abe47058ac42b512c44e901c75cc75241c227a94c1 3.37kB / 3.37kB  14.2s
=> => sha256:52cd8b9f214029f30f0e95d330c83ab4499ebafa 45.60MB / 45.60MB  20.9s
=> => sha256:f0eeafef34cdd8f38ffee08ec08b98645636840210 2.28MB / 2.28MB  15.4s
=> => extracting sha256:9a0518ec57568a70561f7c04650f9554c88dada973f54d88  0.3s
=> => extracting sha256:356172c718acf9930d9465b170864319079e2d2ebac0ddef  1.4s
=> => sha256:55a1a1afffff207f32bb4b7c3efcff310d55215d1302bb3 450B / 450B  15.6s
=> => extracting sha256:dddcd3ceb9980caac379e8f8eeafe1e901f017e5768b8677  3.9s
=> => extracting sha256:abe47058ac42b512c44e901c75cc75241c227a94c18f692e  0.0s
```

```
=> => extracting sha256:52cd8b9f214029f30f0e95d330c83ab4499ebafaac853fdc  1.1s
=> => extracting sha256:f0eeafef34cdd8f38ffee08ec08b9864563684021057b8e6  0.0s
=> => extracting sha256:55a1a1afffff207f32bb4b7c3efcff310d55215d1302bb330  0.0s
=> [internal] load build context                                         0.0s
=> => transferring context: 2.51kB                                       0.0s
=> [2/4] WORKDIR /usr/app                                                0.1s
=> [3/4] COPY . .                                                        0.0s
=> [4/4] RUN npm install                                                 2.9s
=> exporting to image                                                    0.1s
=> => exporting layers                                                   0.1s
=> => writing image sha256:6ebeaaf623312cb7cceff2edaec458dc98bb79be13c02  0.0s
=> => naming to docker.io/yiorgos/docker-nodejs-sample                   0.0s

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout
quickview

% docker images
REPOSITORY                      TAG    IMAGE ID    CREATED           SIZE
yiorgos/docker-nodejs-sample    latest      6ebeaaf62331   25 seconds ago
1.12GB
nginx                           latest760b7cbba31e   2 days ago      192MB
```

Now we are ready to run the container:

`docker run -d -p 3000:3000 yiorgos/docker-nodejs-sample` and you can check how the application behaves by calling it from the browser in `127.0.0.1:3000`. As you can see by inspecting the logs, the container stores data in its ephemeral filesystem. If we need persistence across container restarts, we can achieve it by making use of an environment variable that the application expects and a bind mount:

```
# the file needs to pre-exist in the host machine
% touch lala.db

% docker run -d -p 3000:3000 -v $(pwd)/lala.db:/db/lala.db -e
SQLITE_DB_LOCATION=/db/lala.db yiorgos/docker-nodejs-sample
d813e2aea757a8ba33d801d6b0b6b1ce611e11e6eb3b26ae99c122864cbf6023

% docker logs elegant_burnell
Using sqlite database at /db/lala.db
Listening on port 3000
```

If you call `127.0.0.1:3000` and make some use of the application, kill the container (`docker rm -f`) and then start a new container with the same command, you will see that the data has persisted. This gives you an idea how to work with other containers also

A note about images

Here is an interesting note about images

```
% docker pull node:18.16.1
18.16.1: Pulling from library/node
42cbebf8bc11: Already exists
9a0518ec5756: Already exists
356172c718ac: Already exists
dddcd3ceb998: Already exists
abe47058ac42: Already exists
52cd8b9f2140: Already exists
f0eeafef34cd: Already exists
55a1a1affff2: Already exists
Digest: sha256:f4698d49371c8a9fa7dd78b97fb2a532213903066e47966542b3b1d403449da4
Status: Downloaded newer image for node:18.16.1
docker.io/library/node:18.16.1

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout
quickview node:18.16.1

% docker images
REPOSITORY                     TAG    IMAGE ID    CREATED            SIZE
yiorgos/docker-nodejs-sample   latest      6ebeaaf62331   22 minutes ago
1.12GB
nginx                          latest760b7cbba31e   2 days ago      192MB
node                           18.16.1   8e80a3bf661b   7 months ago      1.09GB
```

Does this mean that 2GB of disk space has been occupied on disk? No. In fact the 1.09GB are shared between `node` and `yiorgos/docker-nodejs-sample` and the extra megabytes are the npm modules and the source code of the application. This is the beauty of the layered filesystem. And this allows for fast deployments by shipping only the necessary changes between container image versions.

Pushing the image to a registry

Assuming you have made an account in the docker hub, you can now push your image and make it available for the world to see. Before pushing, you need to log into the docker hub from the command line by running `docker login` (you do not need to do this every time).

```
docker push yiorgos/docker-nodejs-sample
```

When no registry name is prepended in the image name (and thus in `docker push`) then `index.docker.io` (the docker hub) is implied. The part before the slash implies your username. See this discussion about docker image tagging.

What about docker-compose?

(Or `docker compose` which is slightly different). I have stopped using docker-compose. If you need to use this for running more than two containers in your machine, read here, but it is best that you try something different, like microk8s. Here is though a very simple `docker-compose.yaml` file that can get you started:

```
services:

  bottle:

    image: yiorgos/bottle

    ports:

    - "8000:8080"

    environment:

      REDIS_URL: redis://redis:6379/0

  redis:

    image: redis
```

This describes two services: A service named bottle (after the Python web framework it uses to serve its content) and a redis cache. You can start the services by running `docker compose up -d` and experiment.

Anything to take notice of when on Windows?

When working with Windows WSL and possibly Docker desktop (or podman; much preferred for licensing reasons), the following settings might be useful:

You should use either Ubuntu 22.04 or 24.04 from the Windows App Store.

You need to edit / create the file `C:\Users\yiorgos\.wslconfig` to drive WSL. This assumes you are on a 16G RAM machine with 8 cores.

```
[wsl2]

memory=8GB

processors=4

swap=8GB

#removed#pageReporting=false

localhostforwarding=true

nestedVirtualization=false
```

Within your WSL though you also need to create `/etc/wsl.conf`:

```
[boot]

systemd=true
```

When running a service from within WSL it is quite possible that you want to access it from your browser for example. As such you would call something like `http://localhost:9000`. This works. However, `http://127.0.0.1:9000` might not work. Again, `http://[::1]:9000` will work. This is because WSL forwards localhost to the IPv6 localhost address `::1` instead of the `127.0.0.1` which is the IPv4 one. So when in doubt, call by name.

When on WSL `sudo apt install wslu` - this installs a number of utilities that are helpful for your interaction with Windows. Also `sudo ln -s /usr/bin/wslview /usr/local/bin/xdg-open` - this will allow you to launch a browser window from within WSL when needed.

Sometimes people need to experiment with stuff and do not want to work with the default WSL machine. This is doable. First, create a directory where you will export a pre-existing machine, like the vanilla Ubuntu 22.04 you have already installed. You can see your installed machines by running

```
PS C:\Users\yiorgos> wsl -l -v
```

```
 NAME                      STATE           VERSION
* Ubuntu-22.04              Stopped         2
```

Now create a directory where you will export the machine and then export it. Take care not to export the machine to any of the standard directories, as they might sync with OneDrive and this can cause performance issues.

```
PS C:\Users\yiorgos> mkdir mywsl
PS C:\Users\yiorgos> cd mywsl
PS C:\Users\yiorgos\mywsl> wsl --export Ubuntu-22.04 mywsl -vhd
PS C:\Users\yiorgos\mywsl> wsl --import-in-place mywsl mywsl.vhdx
PS C:\Users\yiorgos\mywsl> wsl -l -v
 NAME                      STATE           VERSION
* Ubuntu-22.04              Stopped         2
 mywsl                     Stopped         2
```

You have now created `mywsl` which is a copy of Ubuntu-22.04

Now start this as root to perform the final configuration

```
wsl -d mywsl -u root
```

You need to edit the file `/etc/wsl.conf` and add the following section

```
[boot]
systemd=true
[user]
default=MY_DEFAULT_USERNAME_AS_IN_UBUNTU_22.04
[network]
hostname=mywsl
```

And you are all set for your experiments!

What is an ENTRYPOINT?

This is best explained with an example using the following Dockerfile

```
FROM busybox
ENTRYPOINT [ "echo", "hello" ]
CMD [ "world" ]
```

Build the image with `docker build -t hello .` and now run the following commands and see what happens

```
docker run hello
docker run hello george
docker run hello george adamopoulos
```

You see now that you can make clever use of the ENTRYPOINT (and the `docker run --entrypoint=...`) to start the container with certain initial conditions and defaults.

Multistage builds

It may be the case from time to time that while we build something in a container, we need only the final artifact and not the whole build system in the final image. This is because (a) the final image will have a considerably smaller size and (b) it is a security feature not to have a development system packaged with the final image when possible. Consider the following two examples of multistage builds

Multistage build for a Golang HTTP server

```
# This is an example for a multistage build
# We build a simple http server using the golang docker image
# but we deploy it using busybox
FROM golang AS server-build
WORKDIR /usr/src
COPY http-servers.go .
# https://golang.org/cmd/cgo/
RUN CGO_ENABLED=0 go build http-servers.go

FROM busybox
WORKDIR /usr/app
COPY --from=server-build /usr/src/http-servers .
EXPOSE 8090
CMD ["./http-servers"]
```

```
# FROM scratch
# COPY --from=server-build /usr/src/http-servers .
# EXPOSE 8090
# CMD ["/http-servers"]
```

Here, we can also see the creation of a container “from scratch” (without any operating system support tooling) since the final artifact is a statically linked binary

Building a base nodejs image

```
FROM public.ecr.aws/amazonlinux/amazonlinux:2023 as build
WORKDIR /usr/src
RUN yum install -y tar gzip
# RUN curl -O https://nodejs.org/dist/v18.16.1/node-v18.16.1-linux-x64.tar.gz
# Instead of running the RUN above, try ADD instead
ADD https://nodejs.org/dist/v18.16.1/node-v18.16.1-linux-x64.tar.gz .
RUN tar zxf node-v18.16.1-linux-x64.tar.gz

FROM public.ecr.aws/amazonlinux/amazonlinux:2023
COPY --from=build /usr/src/node-v18.16.1-linux-x64 /usr/local/node
WORKDIR /usr/app
ENV PATH /usr/local/node/bin:$PATH
```

Here, the base build container contains tools like `gzip` and `tar` which in the final image are not included

And what about distroless containers?

"Distroless" images contain only your application and its runtime dependencies. They do not contain package managers, shells or any other programs you would expect to find in a standard Linux distribution.

Here is how we could build a distroless version of the `docker-nodejs-sample` application:

```
FROM node:20 AS build
WORKDIR /app
COPY . .
RUN npm install

FROM gcr.io/distroless/nodejs20-debian12
WORKDIR /app
COPY --from=build /app /app
EXPOSE 3000
CMD [ "src/index.js" ]
```