Instead of handing over a war file to a deployer, we're going to hand over a docker image.And because the Docker image contains a complete environment that needs no further configuration, it's much easier to deploy and to get it up and running.

But the main part of the Linux operating system?Well, any operating system, really, the main parts of the operating system, which handles the low

level services, such as memory management, device drivers and so on, is called the **kernel**.

Graphical user interface, application

Description automatically generated

each container does contain its own distribution, its own Linux distribution.

So I might have one container here running Ubuntu and a stack of applications.

I might have another container running at the same time, which has its own distribution in this case.This is the centos distribution.

And of course, I could go on here.

I have a third container running my SQL, which happens to be bundled with the Red Hat Linux distribution.

It's important to understand that these are not operating systems.

Ubuntu centos and Red Hat Linux are just sets of applications and tools.

When you get hold of, let's say, and Ubuntu installation DVD.

What you have on that DVD, a broadly two things you have on there, a suite of applications that the Ubuntu developers have decided is a **good collection of tools and utilities**, and it also includes a copy of the **Linux kernel**.

Now, when you see the container contains a Ubuntu always saying there is that the container contains **the Ubuntu collection of tools and utilities**.

And there'll be a slightly different set of tools in the centos distribution and in the Red Hat distribution.

But none of these containers contain the Linux kernel.

So that's the reason why running containers is not the same as running virtual machines.

When you spin up a virtual machine, it will have its own kernel.

And for that reason, virtual machines are heavy and you will struggle to run many virtual machines

on a typical development machine.

Containers are much lighter and you can run many more of them.

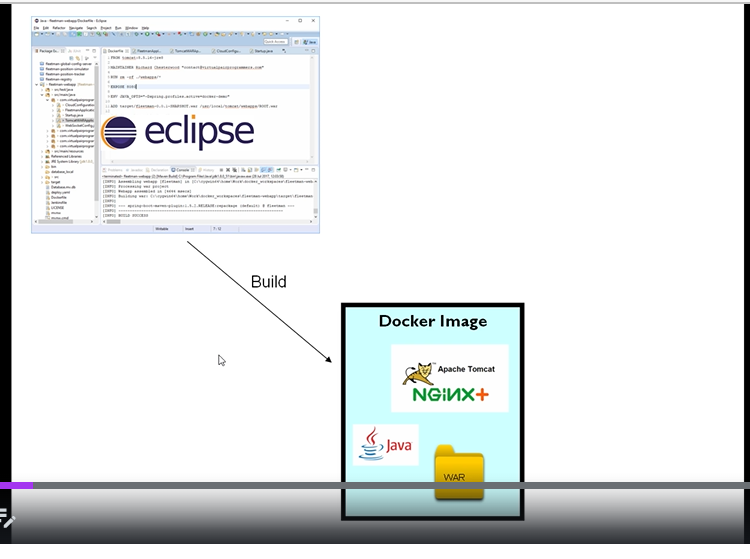
a docker container is actually just a process on your Linux kernel.

So this means that instead of the horrible scenario that I described at the start of this chapter,

if we make it so that instead of building a war file and handing that over for deployment, we build a Docker image instead, then it's easy for the deployer.

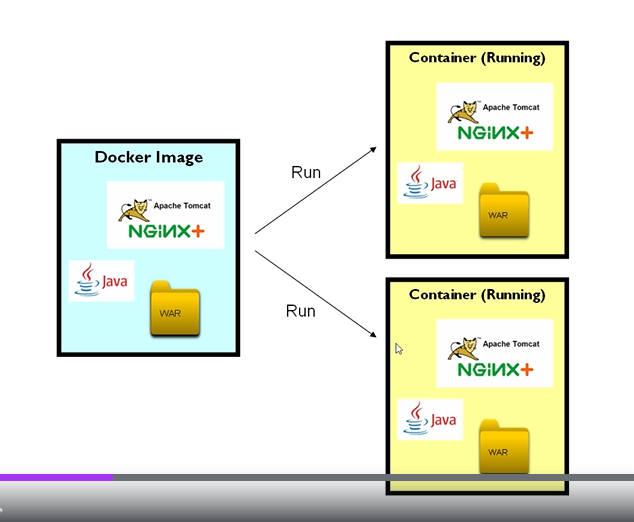
They just run the container on their target hardware, assuming they have Docker installed, of course.And because the container contains all of the necessary configuration and environments, it will just run.





we're going to be build image as part of our regular building process,

either on the command line or possibly through your development environment.



images are intended to be portable.

If I hand over this image to one of my fellow developers on the project.

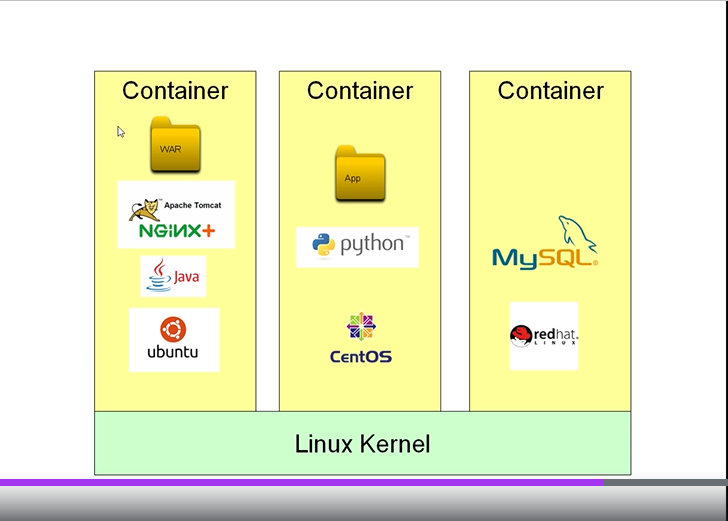
They've got a computer that's configured completely differently, but they can, but they should be

able to run the image in exactly the same way and they would have exactly the same contents inside the

container and there shouldn't be any environmental problems.

 when we're developing, we're going to need to have docker installed to build the image.

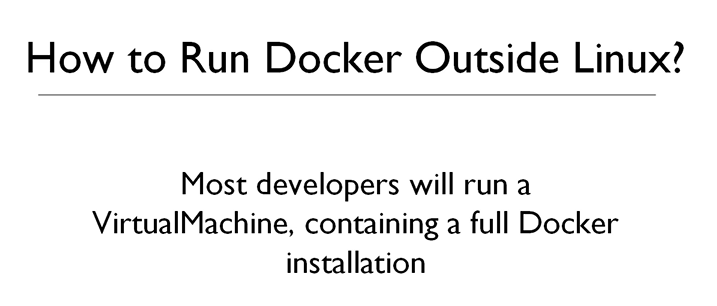
Container is a process running on Linux.

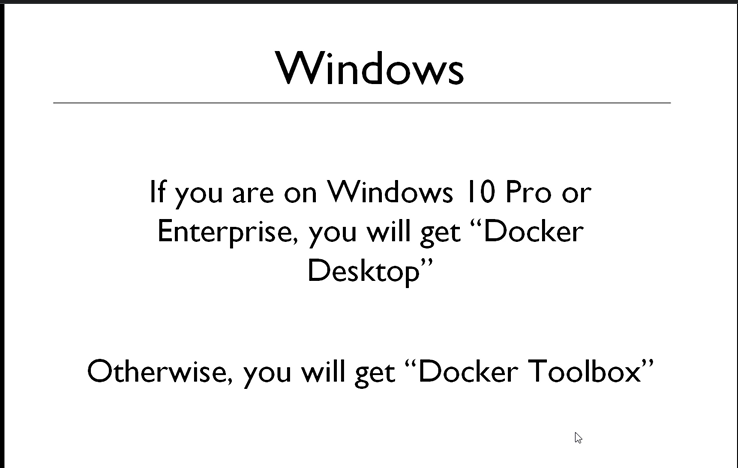


But for those of us who want to develop on Macintosh and Windows, docker containers are not going to run natively. The workaround is that we will have a virtual machine running on our computer and that virtual machine

is going to have a full Linux operating system inside it.

And of course, it will also have docker installed inside and will set things up so that we can access that virtual machine from our regular command line.





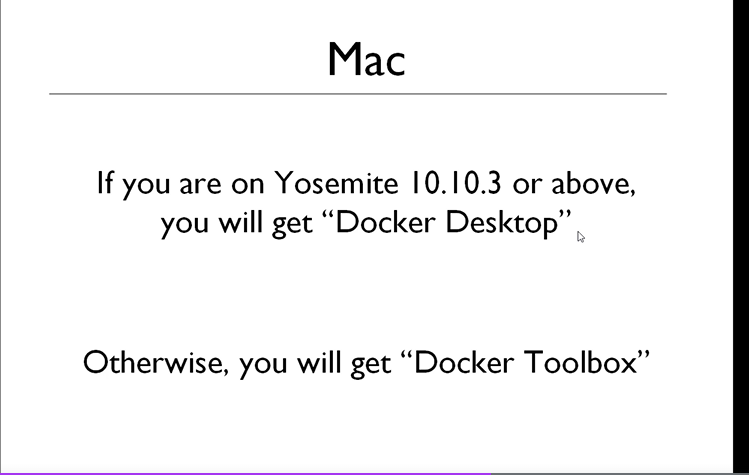
Docker desktop is effectively a virtualised Linux machine. If you are on Windows, then I think you have to enable Hyper V on your computer to make all of this

work.So Hyper V is the built in Windows virtualization supports.

For Mac users, hyperKit is for virtualization support.

Docker Tool Box will automatically download a third party virtual machine implementation. It happens to come from Oracle and it's called Oracle Virtual Box.

This is because virtualization is disabled on windows 10 home edition.



Docker file determines what's going to be in the ultimate running container.

every single container when it started will run a command and your

container will exist and will live and survive for as long as that command is running.

If you're running on Docker Tool Box, you can do `docker-machine IP`.

Now, for me, that's not going to give me any.

Well, it's actually giving an error and that's because I'm using Docker for windows.

But on the docker Tool Box you should see an IP address in there which you'll be able to use in your browser instead of localhost.

So if you do have an IP address here, then copy that in and paste it into your browser: 8080

If you are using docker tool box, you have to use virtual machine ip

If you are using docker desktop, you have to use localhost because in this case you are using your OS virtualization support.

you can't modify any images. You can only extend it by creating your own images based on the original images.

There are a number of images which have been designed by docker themselves,

and it is a full time team of people who are doing this for a living and they create images that they think are going to be globally useful.

And you have a kind of reassurance with the official images that they've been fully tested and that they're safe, they don't contain malware or viruses and so on.

Extending image example:

what we could do once we have that image is we could build on that base image.

We could add on another layer, which would mean creating a new image from that base ubuntu image, which adds on our own layer, perhaps containing Java eight.

Now, notice I've said create a new image because images cannot be modified.

So we would be creating a new image once we've got an image with Ubuntu and Java.

We'd also want our applications to be parts of that image so we would repeat the process.

We create a new image based on the previous image which adds on the application.

So we'd end up with our own image containing Ubuntu and Java and our application, and we could now run that as a container.

If we can have a look at the old fleet man webapp container that we started some time ago, the command

for that was Catalina.sh run.

If you know about Tomcat, then you'll know that that is the command to run Tomcat just as a command,

and it will run forever, basically until you kill it.

So therefore, that's why the Fleet man webapp container once which started it continued running.

But the problem with the Ubuntu container is it's running the command called bash.

Now, you might be familiar with Bash, the shell.

It's one of the shells available on Linux environments.

The problem with Bash is if it's not connected to a terminal, Bash assumes it's just going to be running

a script.

And if there's no script there, it just immediately exits.

**Docker container run -it ubuntu**

The dash i means interactive and the T means connects the container to the terminal

But now I'm on some kind of a Unix command line and I am, in fact, in a bus shell inside that docker

container.

So I can now start doing a UNIX commands, I can do an LS there and the colors are clashing there.

But I hope you can see that there is a Unix Linux style file system right there and you might be wondering,

well, what can we do inside this container and the several answers to that question.

The first answer is really, you can do everything in here that you could do in any other Linux environment.

If I issue the command again after existing

Exit

**Docker container run -it ubuntu**

It will be a new container created from the same image.

**Alpine** is fantastic in that it is a very minimal Linux distribution.

I mean, there's almost nothing bundled with it, just the very basics.

Inside any container(e.g ubuntu container was run in interactive mode and jdk or git is installed) ,the changes we make, are going to be transient when we stop and remove that container.

Those changes are gone.

But actually, you can as I say, you can take a snapshot of a running container.

**Docker container commit**

Creates image from container’s changes.

It is very unusual to create images from snapshots.

The reason why it's not a good idea generally is that the image that we've just created is not repeatable.

And I have no way of looking at that image in four years time and getting any kind of understanding

as to what's inside that image.

I could have put anything in there. So for that reason, we almost always would create images using a Docker file instead

e.g dockerfile

Graphical user interface, text, application

Description automatically generated

The first line is a really important line, and this line is specifying which image we are extending.

It's important to remember that you can't change existing images.

We're taking this image and we are adding to it and creating our own new image.

And even if we haven't pulled that image, it will be automatically downloaded when we try to build

our new image.

So the thing following the Ubuntu, the Kolon, this latest is a tag.

So this is going to work on the latest Ubuntu image.

I could have specified a specific tag here.

And you can get a full list of those tags from Docker Hub.

The second line is actually optional.

It's a courtesy really to future users of this image.

The maintainer line is saying, who wrote this image?

Now, on that base image, we can execute commands and really all I'm doing here is replicating what I did when I did it manually on the bash shell.

that run command in a docker file is saying that docker needs to run the following commands inside the base image. So all I've done here is have copied the APT, get updates and the APT get installed.I don't know if you know, but this double ampersand just allows us to chain multiple commands together on a shell.

And usually the final line in the docket file is the C M D line, there are some alternatives, actually,

but I won't get hung up on that right now.

And the command is what command do we want this container to execute by default when it is run?

And we'll talk a lot more about that over the course of this chapter.

But for now, I'm just going to make it so that it automatically runs a shell when a container is run from this image.

Creating image from dockerfile:

**Docker image build -t jdk-image-form-dockerfile** .

This ending dot(.) is the one mandatory argument to be specified. the argument is what folder do we want the image creation to work from? And what that means is that all of the files in this folder and any sub folders are visible to the Docker file.Dot means current folder.

At every step It's actually starting a new container with that information built in.

And it continues to do this with every step.

It fires up a new container based on the previous image.

It applies this change and then creates a new image from it.

it continues in this way. And each of these steps is actually called a layer in docker.

All these intermediate imgaes are cached. So that next time if there are any changes done e.g change step4 then  it won't have to repeat any steps that it already has an intermediate image for.

Lastly we run container from the image:

**Docker container run -it jdk-image—from-dockerfile**

Add below statement as second last statement in dockerfile.

COPY test-program.jar /usr/local/bin/

there is a command called WORKDIR.

This allows you to specify the working directory

And the benefit of that is that means any future commands, such as the copy command and some of the

commands that we're going to see later will now all work relative to that directory.

So that cleans things up a bit in a complicated docker file for us.

We can just put in now a dot to say that it's going to go in the working directory.

WORKDIR /usr/local/bin

COPY test-program.jar .

**Docker container run -it jdk-image—from-dockerfile**

  when we run when we now run this image, we're not in

the roots anymore, we've been dropped directly into that working directory.

It's very common and best practice for a container to implement one single service.

CMD [“/bin/bash”]

Now, this is the command that you want the container to execute when it starts up.

 it's this command which is going to specify what service is being offered by this container.

FROM UBUNTU:latest

MAINTAINER Hiti Gogia [hitigogia@gmail.com](mailto:hitigogia@gmail.com)

RUN APT-GET update && apt-get install -y openjdk-8-jdk

WORKDIR /usr/local/bin/

COPY test-program.jar .

CMD [“java” “-jar” “test-program.jar”]

When a container is run interactively, it can be stopped using ctrl+ C. however if the container is run non interactively, you need to manually stop it using command, **docker container stop <containerid>**

Docker works is it captures system dot out standard output, in

other words, and it considers that the log of the container.

Use the command **docker container logs -f <containerid>** to see the logs of container which is running in background.

**Docker file reference**

**ADD vs COPY :** It's just that the ADD command has some extra features.

For example, Add Command can work with remotes URLs and it can also do things like unzipping or unpacking archives.

 Docker people seem to recommend that copy is preferred because it's just simpler and

is more obvious what copy is going to do.

**CMD vs ENTRYPOINT :** CMD mentions the default command which can be overridden when the container is run. if you're working with an image that has a CMD, if you wish,

you can change the command that gets executed.

So I could instead of running that Java service, I could do

**Docker container run -it jdk-image-from-dockerfile /bin/bash**

However this isn’t possible using ENTRYPOINT.

MAINTAINER vs LABEL : use the label command which is more flexible and enables setting any metadata you require. E.g maybe set up a key called maintainer and the value being whoever the maintainer is.

e.g LABEL [maintainer=hitigogia@gmail.com](mailto:maintainer=hitigogia@gmail.com)

we can have other metadata with label e.g the version of the dockerfile, the date on which dockerfile was last created.

LABEL creationdate=”19 November 2020”

Some projects will use maintainer some projects.

Will use labels, some projects will do the gold standard, and they will use labels and will use

those official list of labels defined by the open container standard, and some projects will simply

not bother using any metadata at all.

**Docker container run -p 8080:8080 -it fleetman-webapp**

But how did I know that some cat runs on Port 8080? Look for the dockerfile of tomcat image it will have a line like below

EXPOSE 8080.

However this line does not do anything . even if this is skipped image will work. Only good documentation images have this line.

It's just functioning as a type of documentation between us, the builder of the image, and whoever

is going to eventually run this image as a container and using this line, they'll know what to use

in the dash p command.

The left port is completely arbitrary. for example, I, I make it a public port 80, then I'll access Port 80 through the browser and that will be forwarded to Port 880 inside the container.

**Docker container run -p 80:8080 -it fleetman-webapp**

Run in detatched mode like this:

**Docker container run -p 80:8080 -d fleetman-webapp**

Now the container runs in background.

This command just retruns the id of the container.

After that you do “docker container logs 48” where 48 is the id

To go inside the container , run -  
**Docker container exec -it 48 bash**

You will get bash shell inside the container.

Slim/alpine images donot have sophisticated tools installed so you might not have bash available for them.

In those cases, you should use

**Docker container exec -it 48 sh**

Bash is more feature rich.

With bash you will land on some default folder

e.g /usr/local/tomcat#. Where is this folder coming from? It is coming from dockerfile. WORKDIR sets the default directory.

Inside /usr/local/tomcat we have webapps . Inside webapps we have a ROOT folder.

If I look in webapps, this is where we install whatever websites we want Tomcat to be serving up and

the root folder will contain whatever is being displayed by default.

So if I go into the root folder and have a look in there, I reckon the index JSP file is probably the exact file that has been rendering this default Tomcat page

Inside webapps folder we have other folders like manager, host-manager, docs examples

I want to remove all the folders inside the webapps folder. How do I do. Add a RUN command in dockerfile. we can use the run

command, which will allow us to issue any Linux commands that we like that are going to run inside

the container as the image is being built.

FROM tomcat:8.5.47-jdk8-openjdk

MAINTAINER “hiti Gogia”

**RUN -rm -rf /usr /local/tomcat/webapps/\***

#Transfer WAR  
  
EXPOSE 8080

CMD [“Catalina.sh,”,”run”]

After this rebuild the image

Docker image build -t fleetman-webapp .

And then run the container

Docker container run -d -p 80:8080 fleetman-webapp

Docker container exec -it 18 where 18 is containerid

How to copy war file :

COPY ./target/fleetman-0.0.1-SNAPSHOT.war /usr/local/tomcat/webapps/ROOT.war

How to pass java args

ENV JAVA\_OPTS=”-Dspring.profiles.active=docker-demo”

FROM tomcat:8.5.47-jdk8-openjdk

MAINTAINER “hiti Gogia”

**RUN -rm -rf /usr /local/tomcat/webapps/\***

#Transfer WAR  
**COPY ./target/fleetman-0.0.1-SNAPSHOT.war /usr/local/tomcat/webapps/ROOT.war**

**ENV JAVA\_OPTS=”-Dspring.profiles.active=docker-demo”**

EXPOSE 8080

CMD [“Catalina.sh,”,”run”]

So it was definitely useful to build an image featuring a full Tomcat installation that we deployed

a war file too.

But if you're working in Spring Boot, you will know that there isn't really any reason to do that because

Spring boot embeds tomcats and you can run a spring boot application directly as a jar file.

SPRING BOOT APP + JSP = deploy as WAR

SPRING BOOT APP + time lyfe = can be deployed as JAR

FROM openjdk:8u131-jdk-alpine

MAINTAINER Hiti Gogia

EXPOSE 8080

WORKDIR “usr/local/bin”

COPY ./target/fleetman-0.0.1-SNAPSHOT.jar webapp.jar

CMD [“java”,”-Dspring.profiles.active=docker-demo”,”-jar”,”webapp.jar”]

We now have this running as if you like, a native spring boat application with no need to install Tomcats.

all images

have an owner forward slash image name only.

The official images published by docker themselves are allowed to have just a name.

To add the owner name we can use below command

Docker image tag 05c hitigogia/fleetmap-webapp

Here 05c is the imageid

And then push the image using

Docker image push hitigogia/fleetmap-webapp

Instead of using docker image tag command you could have also used below command while building the image

Docker image build -t hitigogia/fleetman-webapp .

**How to add additional service to the container?**

it's best practice to make a container do one thing only . Then it's going to be very difficult for the container itself to know if the service has died.

Now, this is going to be important as we go forward on this course.

We need an easy way of knowing if a container is healthy or not.

Technically it is possible : Now, those answers on Stack Overflow were basically saying you would wrap the two services into a single

one, then, OK, you'd be able to monitor that.

But that's really kind of a technical answer.

Just as a micro service should do one thing only it's exactly the same for the containers on which they

run.

A container should expose just a single service to clients.

And that service is the CMD (command instruction) or ENTRYPOINT

But in our example, we do need both services.

We need our Java webapp and we need MYSQL.

Well, so that's where all of this is leading.

We're simply going to have multiple containers and each container will provide a single service.

Now, obviously, we've got to be able to have these containers talking to each other somehow.

the network connection that you can see between these two containers, it's really important to

point out that this is just a container to Container Network connection.

In fact, the containers are on their own private network.

And what that means is that any computers outside of our Docker system are not going to be able to see

this MYSQL container.

The scheme that Docker networking uses is very clever.

It takes the name of the container and it adds that container name to the DNS service that each container

is using.

And what that means, it's very straightforward, is if we're on any container, then we can network

to another container by simply using the containers name as a domain name.

So we don't need to deal with any IP addresses. And docker uses some really clever techniques such that when we start a brand new container with a brand

new name, then automatically all of the existing containers in the network are propagated with that

new name.

we can give every container one or more

names and the names are really intended to be friendly names that we recognize that container by now.

1. Run my sql container: **docker container run -e MY\_SQL\_ROOT\_PASSWORD=password -d mysql:5(** you can pass environment

variables to a container using the dash, e arguments, and then it's just key value pairs.

1. Exec into the container **docker container exec -it 4f bash**
2. Run fleetman-webapp container : **docker container run -d -p 80:8080 fleetman-webapp**
3. Exec into this container : **docker container exec -it dbe sh** where dbe is container id

Docker network ls: list all n/w

**Bridge** is the default n/w. It’s the n/w that all of our containers have been connecting to.

**Host** is the n/w your host is connected to.i.e connection to you router basically. if you were to connect your containers to this, then it would be visible to the outside world.

there is a **none** network

as well, which would simply mean your container would have no networking at all.

Nothing would be able to connect to it, but crucially, it wouldn't be able to connect outwards either.

So I guess you could use that if you really genuinely don't want your container to be outward facing.

Create your own n/w using below command:

**Docker network create my-network**

**docker container run --network my-network –name database -e MY\_SQL\_ROOT\_PASSWORD=password -d mysql:5**

this name is so important because it's going to become the name by which all of the other containers

are able to find this container over the network.

Similary for webapp container

**docker container run –network my-network –name fleetman-webapp -d -p 80:8080 fleetman-webapp**

**docker container exec -t fleetman-webapp sh**

**/usr/local/bin# ping google.com**

**/usr/local/bin# ping database**

DOCA is automatically managing a domain name server,

and this instance, when we issue the PING command, is consulting the domain name server and it's discovering

that the IP address of the target container. this address beginning one seven two is an IP address from the private range.So this is not a publicly available IP address.

Nobody can get access to this database except for the containers that are connected to this private network.

when you're rebuilding an image, as you know,

any step that has already been done, it will use the cash.

But it is a common mistake to maybe forget to rebuild the jar file, in which case it would have just

run through all the way and every step would have been using cash.

And I just like to remind myself that you want to see at least one of these steps.

And it is in this case, step five of six, where it's not using the cash anymore, is removing an intermediate

container.

It's worth looking out for that just in case you forgot to do a step.

But we do now have a fresh image.

We cannot reuse a name of the container even if it is in stopped state. We should remove it first to reuse the name for a new container.

Docker container rm fleetman-webapp

So it's not enough to stop the container.

We also have to remove the container

Alternative way is use –rm while running the container This is saying that when this container stops, it should also be removed, which is really useful.

**docker container run –network my-network -d -p 80:8080 fleetman-webapp –name fleetman-webapp –rm fleetman-webapp**