06. Docker and Containers

We are trying to run a **software system**, not just a single application. This means we will be running software we wrote, software other people wrote (databases/libraries/etc) and we need to have some way to glue all these parts together. Sharing this entire environment is the core of DevOps.

Up until now, we've been putting our systems into a VM, but this can be a bad idea in most cases.

A recent solution to this problem is to use a **container platform**, such as **Docker**.

At a first approximation, a container is a lightweight VM: smaller that a VM and faster to start up and shut down, but something that will still give us an isolated execution environment we can run our systems in.

Important terminology

- **Containers** come from container images, which are similar to a disk image for a VM, they contain a filesystem for the container.
- The container is created from the image, but is distinct from it.
- You can have multiple containers created from the same image, all in different states, with some running and some not...

Common Docker Commands

```
docker build # Creates an image

docker pull # Dowloads an image from a registry (usually DockerHub)

docker run # Runs an image (and creates a container), same as pull + start
docker run -p 8080:80 nginx:latest # Example with port forwarding for ngin
x

docker start <name> # Start a container that was created previously, but is
not running
```

```
docker stop <name> # Stops a running container

docker ps # Shows running containers

docker ps --all # Shows all the containers

docker rm <name> # Deletes a container

docker image ls # Lists all container images

docker image rm <name> # Removes a container image
```

Docker architecture

```
graph TD
  D[Docker CLI] --> E[Docker Engine]
  C[Docker CLI] --> E
```

dockerd (daemon) is the server that runs the docker containers in the background. We never interact with it.The Docker Engine contains the dockerd in Windows and MacOS (which contains a Linux VM for running the dockerd).

Dockerfiles

To create a Docker image, we use a script called a Dockerfile.

Dockerfile command

```
COMMAND args # not case sensitive, but command conventionally wirtten in uppercase

FROM ubuntu:22.04 # Deriving image from existing image (ubuntu - image, version - tag)
```

Docker containers are made of layers, this leads to an important observation: almost all Docker containers are built on top of an existing container (i.e. they modify an existing container).

Docker images are stored on disk as deltas between layers, which is more efficient than storing the layer directly.

Running commands at built time

```
RUN sudo apt install -y git build-essential
# RUN commands get run only when the container image is being built, the
y are used to   install or set up things on the image
```

Environment variables

```
ENV VAR="This is a variable"
# ENV is used to set environment variables in the container itself
```

Dockerfile variables

```
ARG VAR
# ARG variables don't need to be initialised in the Dockerfile, we can just d
eclare them and use the --build-arg switch to specify them while building
```

Copy a file from host to image

```
COPY my_file.html /files
# Move a file from your host filesystem to the container's filesystem
```

Startup command

```
CMD /usr/bin/my_command
# Specify the command to be run when the container is started. Unlike RU
N, this is run  on an active container at runtime, not on an image at build tim
e.
```

Run without a shell

```
CMD ["/usr/bin/my_command"]
# If we put the CMD command into a list, Docker will run the command dire
```

```
ctly, rather   than using the default shell for the image
```

Allow the user to pass arguments

```
ENTRYPOINT ["/usr/sbin/nginx"]
# Specifies the command that always runs when the container starts
```

Give default arguments

```
ENTRYPOINT ["/usr/bin/python"]
CMD ["/example/default.py"]
# If the user doesn't specify arguments, CMD can be used to provide defau
lt arguments
```

Expose a port on the container

```
EXPOSE 8080/tcp
# Allow TCP traffic to use port 8080 on the container
```

Working directories

```
WORKDIR /app
# Create and cd to a working directory in the image
```

Example

```
# Example Dockerfile

FROM nginx:latest

EXPOSE 80

WORKDIR /usr/share/nginx/html
COPY index.html .
```

```
ENTRYPOINT ["/docker-entrypoint.sh"]
CMD ["nginx", "-g", "daemon off"]
```

```
docker buildx build —tag my_container:latest .
docker run -p 8080:80 my_container:latest

# To run an interactive terminal inside the docker container (like VM ssh)
docker exec -it practical_lamarr /usr/bin/bash
```

How does Docker work?

Userspace and kernelspace

```
graph TB
  subgraph " "
     subgraph " "
        Shell
        subgraph " "
           Chrome
           LibreOffice
           VSCode
        end
     end
     Kernel
   end
```

Modern operating systems are composed of two parts:

- the **kernel**, which is the core of the OS and manages hardware.
- the **shell**, which is the user interface.

The kernel runs with special privileges not available to other programmes.

All other programmes run with the same privileges as the shell, this is called the **userspace** (since it is used to run user applications), while the kernel is said to run in the **kernelspace**.

OS-level virtualisation

The modern Linux kernel allows multiple userspaces to run on a single kernel. This is called OS-level virtualisation. There are three things that allow this to work:

- **chroot**: change the root of the filesystem visible to userspace.
- **Namespaces**: isolate resources seen from within a container.
- **cgroups**: limit resources available to a container.

Docker for Windows and MacOS run a Linux VM internally to make this work