# Docker

---

---

## Overview

- A Brief History of Software Delivery
- An Introduction to Docker
- Commands to work with Docker containers
- Creating your own Docker containers
- Sharing your containers with the world

---

## Learning Objectives

- Explain the benefits of images and containers
- Use commands to start, stop and log into containers
- Understand the structure of a Dockerfile and be able to write one
- Demonstrate how Docker Compose can create and run multi-container applications

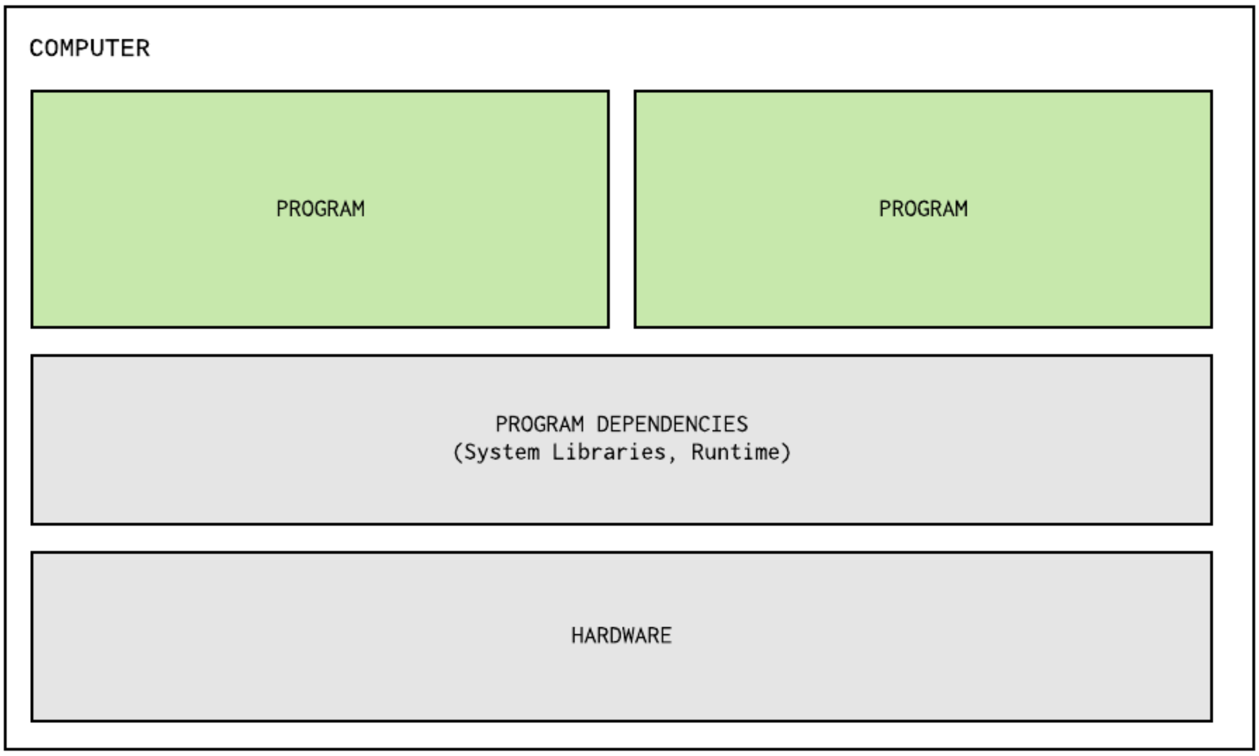
---

## The History of Software Delivery

---

### Multitasking operating systems (1960 - 1990)

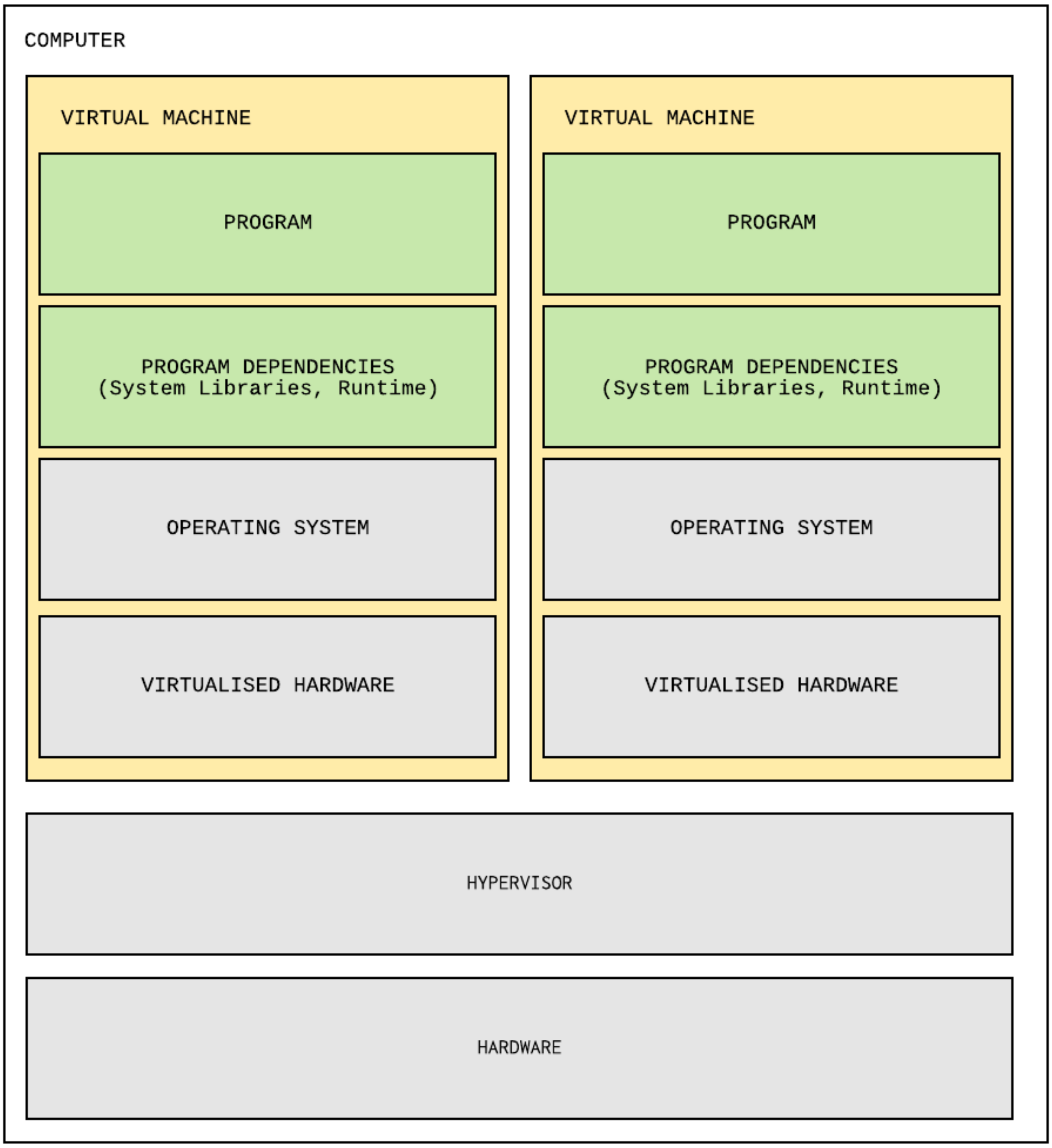
Install many programs to disk and run them on demand



---

## Virtualisation (1990 - 2010s)

Can isolate dependencies in individual VMs (virtual machines)



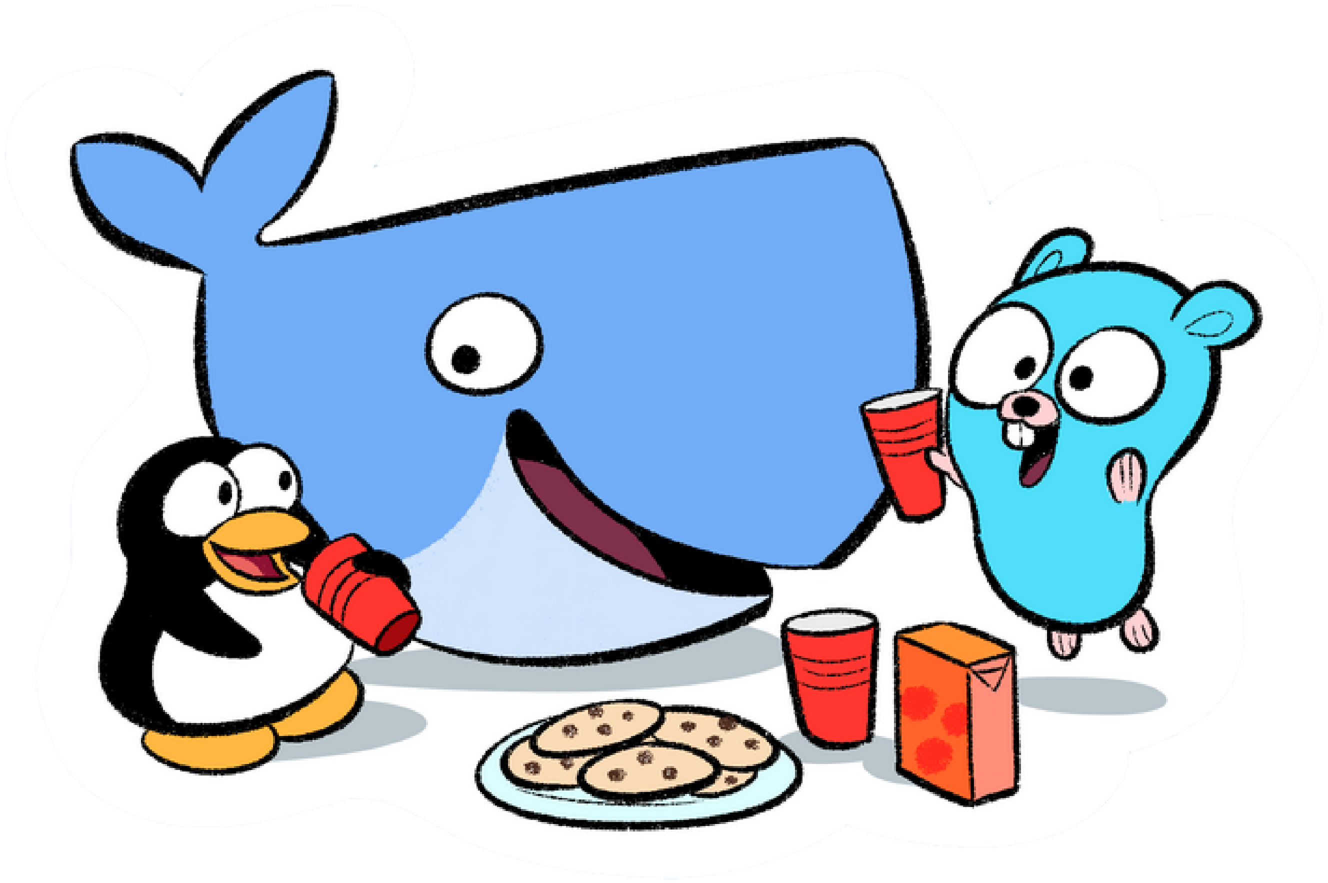
---

## Virtualisation (1990 - 2010s) Pro's and Con's

- Each VM runs its own OS, using a lot of system resources
- Each VM emulates the underlying hardware, using yet more resources
- Still in many cases can save money on computer hardware
- A decent trade off when catering to a large number of users

---

## Enter Docker (2013 - Now)



---

## What is Docker?

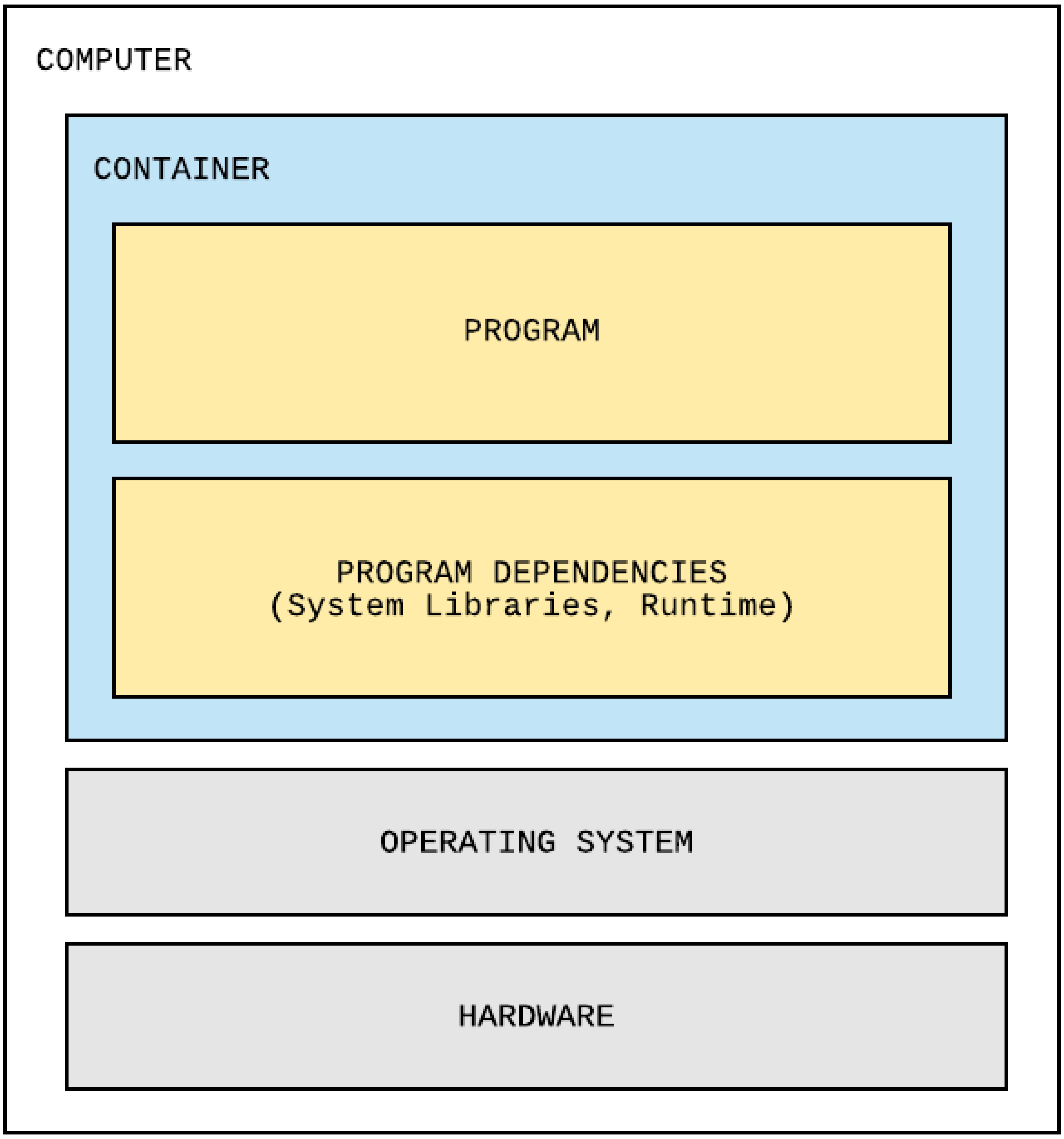
Docker is a tool designed to make it easier to create, deploy, and run applications by using **containers**.

Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.

---

## Containers

Isolate applications and all of their dependencies with none of the overhead of VMs.

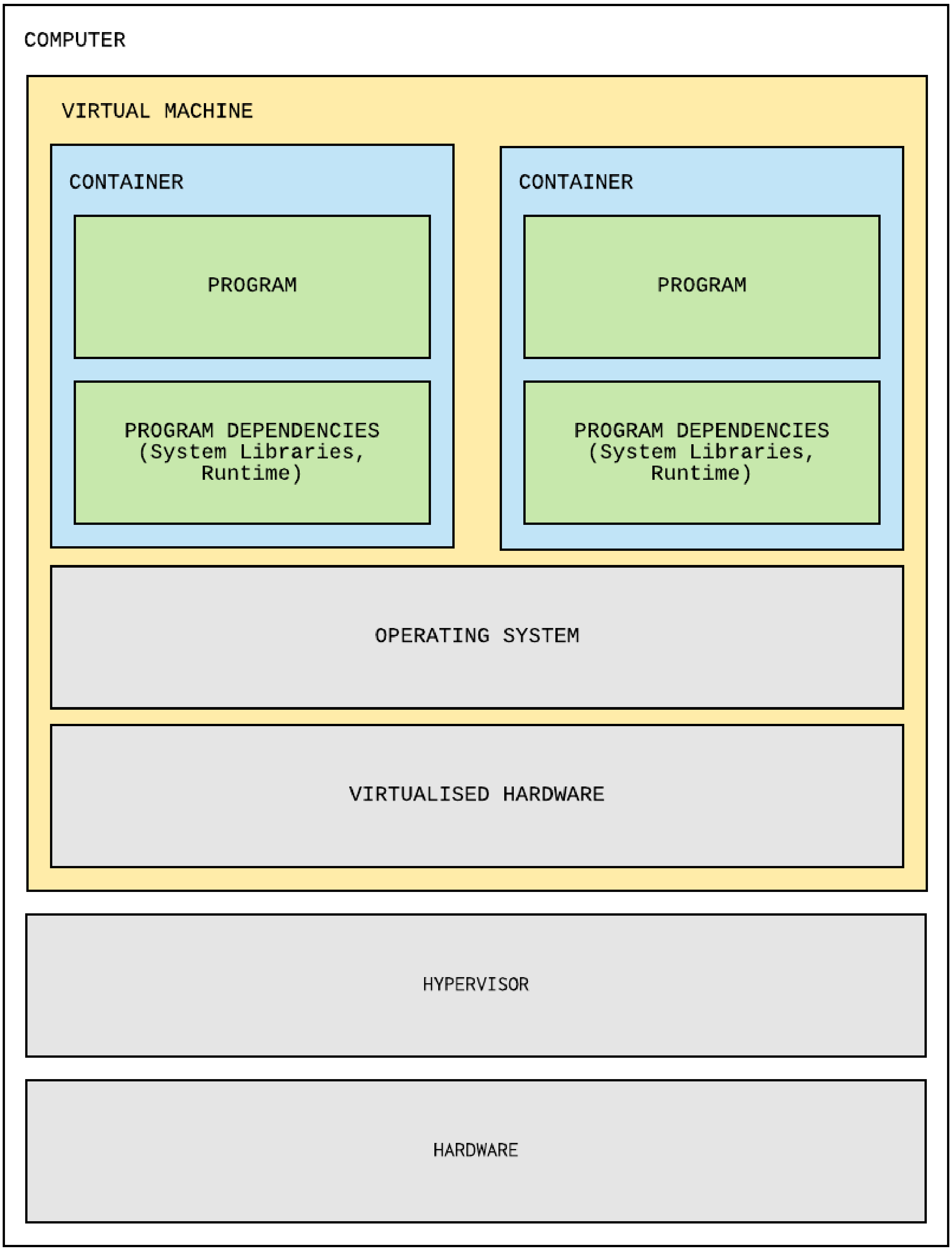


---

## Why containers?

- Provide a self-contained environment to develop and deploy applications
- Isolates application dependencies
- Lightweight in comparison to VMs
- Can easily 'transport' containers and run them on any machine

---



---

---

Quiz Time! 🤓

---

**True or False: A Container has an entire copy of an Operating System (OS) inside it.**

Answer: `False`

---

**Which of the following statements is FALSE?**

1. Containers can improve security if a running application is compromised or hacked
2. Containers often use more memory and CPU than Virtual Machines
3. Containers help prevent "Dependency Hell"

Answer: `2`

---

## Running Docker

Make sure Docker Desktop is running.

You can use your terminal to check if Docker is running:

```
# Get Docker version
$ docker version

# See running containers
$ docker ps
```

---

## Container Images

Container images describe the type of container to run.

An image could contain software such as a web-server or database, or our own applications we have written.

Just like with Python libraries, there thousands of images already available online which we can download and use.

An image is referenced by it's name.

---

## Launching Containers

```
# Example usage
$ docker run [flags] <image> [args]

# Actual usage
$ docker run docker/whalesay cowsay Hello!
```

```
< Hello! >
 --------
```

```
    \
     \
      \
                    ##        .
              ## ## ##       ==
           ## ## ## ##      ===
       /""""""""""""""""___/ ===
  ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
       \______ o          __/
        \    \        __/
          \____\______/
```

---

## Container IDs

Every container has a unique ID we can use to refer to it instead of its name:

```
$ docker ps

CONTAINER ID        IMAGE     ...
31d16fa3edb2        ubuntu    ...
949316a32b9f        mysql     ...
```

We can also see stopped containers with `docker ps -all`

---

## Stopping containers

```
# Stop a container
$ docker stop <container_id>

# Remove a stopped container
$ docker rm <container_id>

# Restart a container
$ docker restart <container_id>
```

---

## Logging into our containers

Just pass in the `-it` flags (`--rm` deletes the container automatically on exit)

```
$ docker run --rm -it debian

root@dba9683049bd:/ # You're now in Debian!
```

---

### exec

Run commands in running containers with the `exec` command

```
# Run a command in an already running container
$ docker exec -it <container_id> <command>

# Launch a bash shell in a container
$ docker exec -it <container_id> bash
```

---

### Running Background Containers

Normally when we run a container, it is `attached` to the current terminal - we see output from it in the terminal.

You can run a container as a background task in `detached` mode.

All output is hidden inside the container, and the container is non-interactive.

```
# Run container as an attached foreground task
$ docker run docker/whalesay cowsay Hello!

# Run container as a detached - no output
$ docker run -d docker/whalesay cowsay Hello!
```

---

### Port Binding

Allows you to connect ports in your containers to your host machine so you can talk to the services running in your containers as though they were running in your machine directly

With port binding, we can run services like web servers and databases in isolated containerised environments and access them as if they were local

```
$ docker run -p host_port:container_port <image>

$ docker run -d -p 8080:80 nginx
```

---

### Running a Database Container

Pass environment variables to your container with the `-e` flag.

Choose which ports your container will use with the `-p` flag.

```
$ docker run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=pass mysql
```

```
# Connect to your containerised DB from your computer
$ docker exec -it <mysql_container_id> mysql -u root -p
```

---

### Exercise: Run a Database

- Start a database instance with the following command:

```
$ docker run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=pass mysql
```

- Confirm the container is running using `docker ps`
- Try and start another instance with the exact same command.
- It won't work! Identify the problem from the error message.
- Change the startup command to fix the problem and allow another database to run at the same time
- Confirm both containers are running using `docker ps`
- Stop both containers

---

### Volume Mounting

Lets us designate a directory in our local filesystem to be shared with a container

```
$ docker run -v <host_dir>:<container_dir> <container_name>

$ docker run -d -v \
  /home/user/projects/html-site:/usr/share/nginx/html nginx
```

---

### Quiz Time! 🤓

---

**True or false: You can create multiple containers from the same image.**

Answer: `True`

---

**What command would you run to get a list of running containers?**

1. `docker exec`
2. `docker inspect`
3. `docker ps`
4. `docker service`

Answer: 3

---

**How would you permanently remove a running container?**

1. `docker rm <c_id>`
2. `docker kill <c_id>`
3. `docker pause <c_id> && docker rm <c_id>`
4. `docker stop <c_id> && docker rm <c_id>`

`<c_id>` refers to a container ID.

Answer: 4

---

## Components of a Docker Image

- The basis of containers
- Consists of the system libraries, system tools and platform settings
- They are built-in layers which represent the commands of a Dockerfile
- When an image is run, it becomes a container

```
# List images
$ docker image ls
```

---

## Dockerfile

- Docker can build images automatically by reading the instructions from a **Dockerfile**
- A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image

---

# Structure of a Dockerfile

`FROM` - what base image we are using

`WORKDIR` - the directory in the image that will contain our files

`COPY` - what files we want copied into the image

`RUN` - any commands we want to run (apt-get install, update etc.)

`EXPOSE` - what ports we want to make available outside our container

`ENTRYPOINT` - what gets executed when starting a container

`CMD` - a command to run when the container is started

Full Dockerfile reference information is here: https://docs.docker.com/engine/reference/builder/

---

### Dockerfile example

```
FROM python:3.9
WORKDIR /application
COPY /app .
RUN pip install -r requirements.txt
CMD python /application/app.py
```

---

### Building a Dockerfile

A Dockerfile can be built into an image that we can then run.

This command will look for a file called `Dockerfile` in the current directory, and build it:

```
$ docker build .
```

To tag the image so we can use it later, use -t

```
$ docker build -t my-application .
```

---

### Docker Pull

- Docker Hub is a registry of Docker images
- We can search for images that might come pre-installed with packages, modules etc.
- By default the pull will chose the image tagged with latest

```
# default
docker pull python

# specific version
docker pull python:3.7.4
```

---

### Docker Push

- Pushes an image to a docker registry
- Creates a repository in your registry
- Tags the image with a version or name

Anyone can pull your docker image from your public docker hub registry

```
# Tag the image
docker tag \
  <name of local image>:<version> \
  <repository name>:<version tag>

# Push the image to your repository
docker push <repository name>:<version tag>
```

---

## Docker Compose

A tool that allows you to create and run Docker applications that use multiple images.

Using Docker Compose is a three-step process:

1. Define your app's environment with a `Dockerfile` so it can be reproduced anywhere.
2. Define the services that make up your app in `docker-compose.yml` so they can be run together in an isolated environment
3. Run `docker-compose up -d` to start docker and run your entire app

---

## Compose Example

See example `docker-compose.yml`

---

## Quiz Time! 🤓

---

**What is a Docker image?**

1. An isolated enviroment to contain a running application and its dependencies.
2. A virtual machine running the host operating system.
3. A template from which a Docker container can be created.
4. An alias for a container.

Answer: 3

---

**What is a Dockerfile?**

1. An isolated enviroment to contain a running application and its dependencies.
2. A text document that contains all the commands needed to assemble a Docker image
3. A virtual machine running the host operating system.
4. Another word for a container

Answer: 2

---

## Exercise: Dockerise your Mini Project!

- Create a Dockerfile to build a container image for your mini project
- Run the Dockerfile and tag the created image as `<yourname>-miniproject`
- Verify the image is created using the `docker image ls` command
- Run your mini project from the image
- Add the dockerfile to source control as part of your Mini Project and commit it to GitHub

---

## Learning Objectives Revisited

- Explain the benefits of images and containers
- Use commands to start, stop and log into containers
- Understand the structure of a Dockerfile and be able to write one
- Demonstrate how Docker Compose can create and run multi-container applications

---

### Terms and Definitions Recap

**Docker**: A set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.

**Dockerfile**: A text document that contains all the commands a user could call on the command line to assemble an image.

**Image**: A serialized copy of the entire state of a computer system stored in some non-volatile form such as a file.

**Docker Image**: A file, comprised of multiple layers, that is used to execute code in a Docker container.

---

### Terms and Definitions Recap

**Virtualisation**: An operating system paradigm in which the kernel allows the existence of multiple isolated user space instances.

**Container**: A standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

**Daemon**: A computer program that runs as a background process, rather than being under the direct control of an interactive user.

---

### Further Reading and Credits

- Docker Documentation
- What even is a container?
- First Steps with Docker (pic)
- Docker (pic)
- opensource.com (citation)
- Docker layers