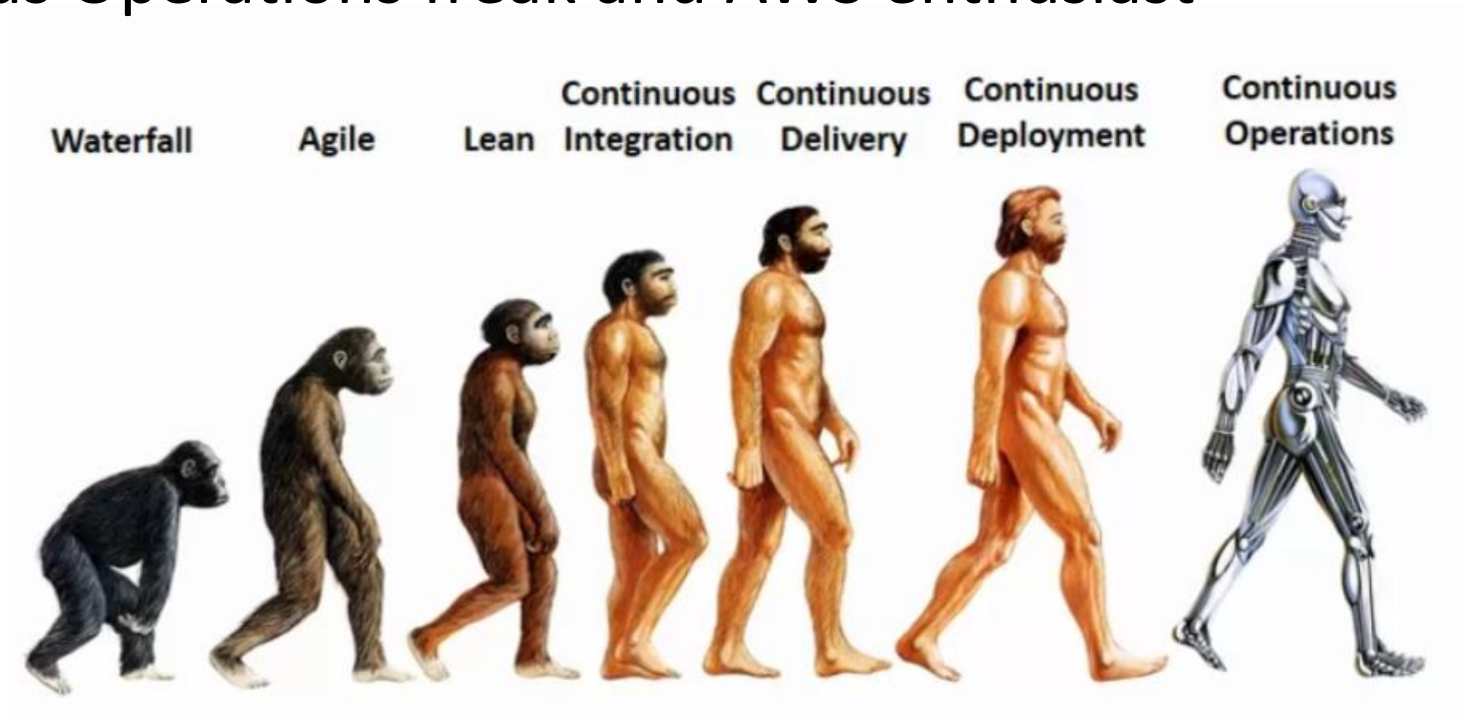
Docker Fundamentials

The open platform to build, ship and run any application anywhere

About me

- 14+ Engineering experience (Cisco Systems, VMware)
- Continuous Operations freak and AWS enthusiast



Session Logistics

- 3 hours (exercising time included)
- “Introduction to Docker” course required
- You will need:
  - Linux machine
    - Dedicated
    - VM
    - Cloud provided
  - Enthusiasm
- Registration at hub.docker.com

Recap from “Introductions to Docker”

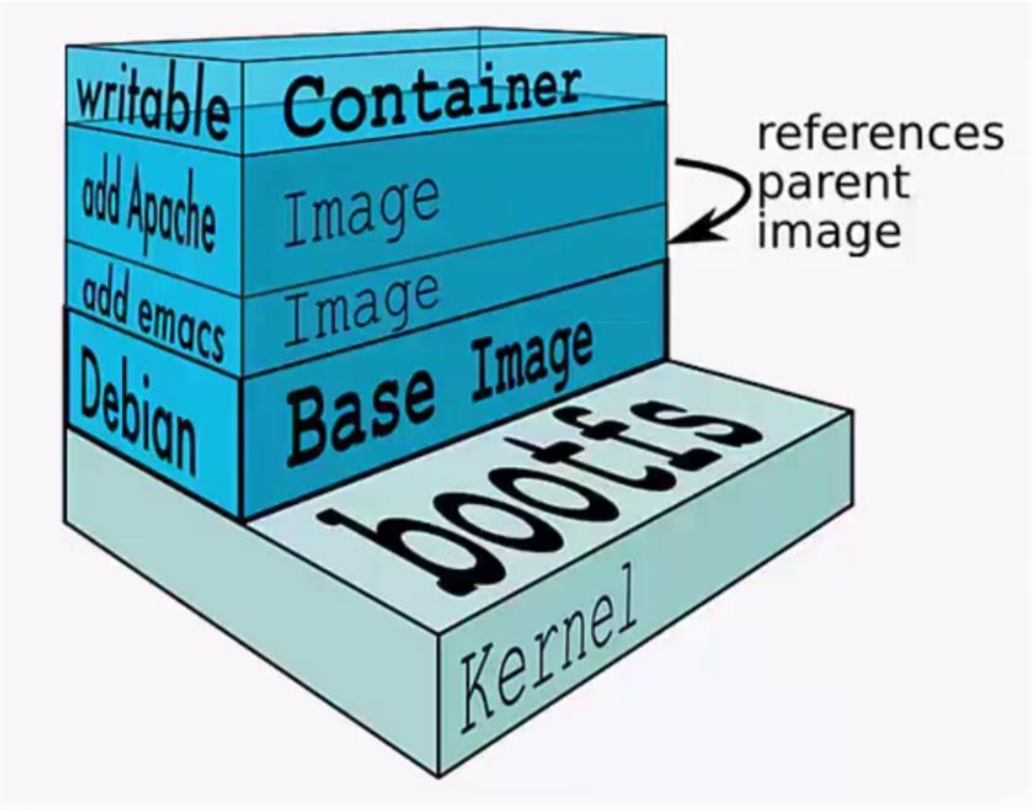
- Intro to Docker
- Benefits of Container based virtualization
- Docker concepts and terms
- Simple container examples
- Commands
    - docker run
    - docker ps
    - docker images

Agenda

- Building Images
- Dockerfile
- Managing Images and Containers
- Pushing Images on Docker Hub
- Docker Volumes
- Basic Container networking
- Configuring our first app

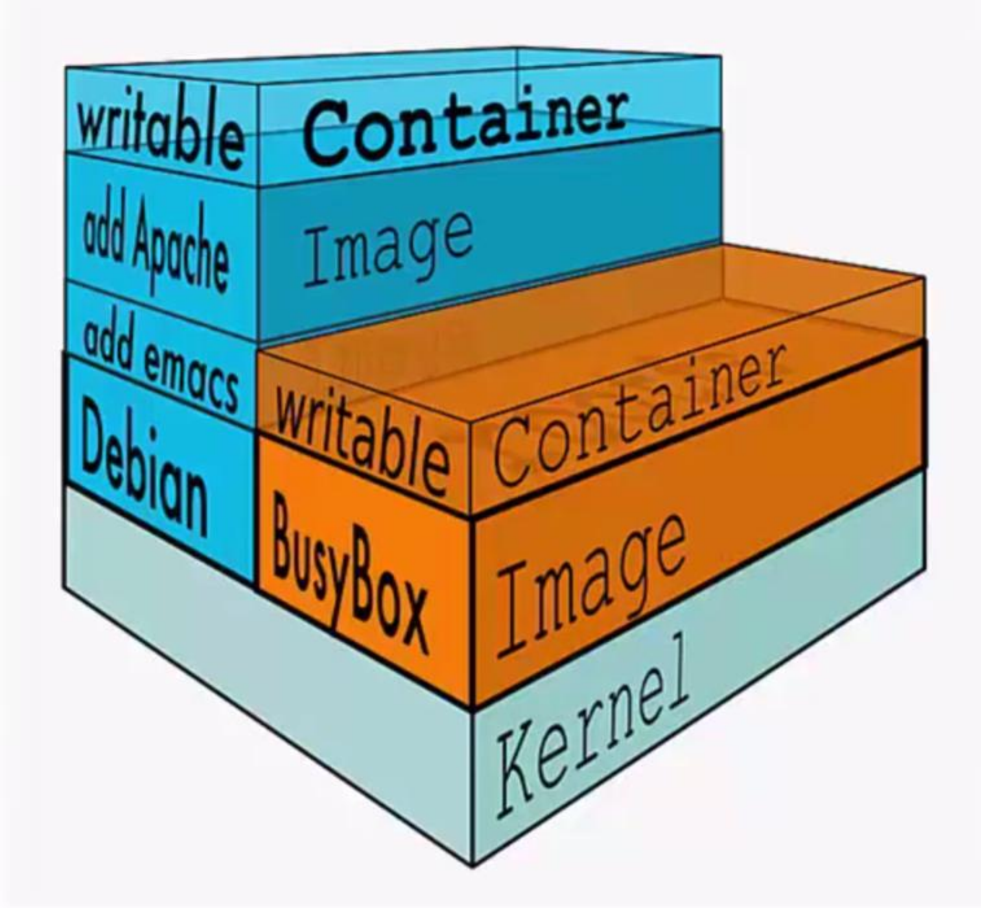
Image Layers

- Images are comprised of multiple layers
- A layer is also just another image
- Every image contains a base layer
- Docker uses a copy on write system
- Layers are read only

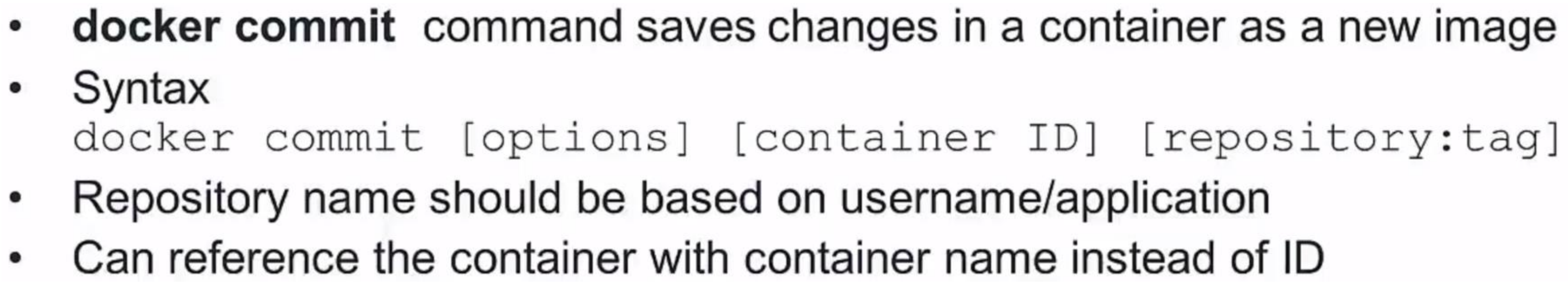


The Container Writable Layers

- Docker creates a top writable layer for containers
- Parent images are read only
- All changes are made at the writeable layer



Docker Commit



- **docker commit** command saves changes in a container as a new image
- Syntax

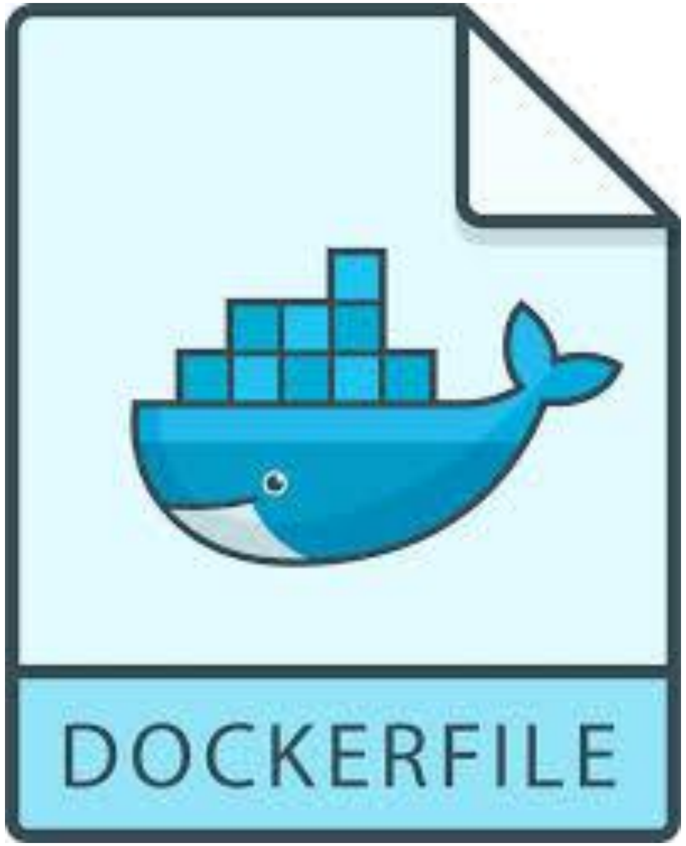
  ```
  docker commit [options] [container ID] [repository:tag]
  ```
- Repository name should be based on username/application
- Can reference the container with container name instead of ID

EXERCISE

Build New Image

1. Create a container from an Ubuntu image and run a bash terminal
   ```
   docker run -i -t ubuntu:14.04 /bin/bash
   ```
2. Inside the container, install curl
   ```
   apt-get install curl
   ```
3. Exit the container terminal
4. Run `docker ps -a` and take note of your container ID
5. Save the container as a new image. For the repository name use <your name>/curl. Tag the image as 1.0
   ```
   docker commit <container ID> <yourname>/curl:1.0
   ```
6. Run `docker images` and verify that you can see your new image

Dockerfile



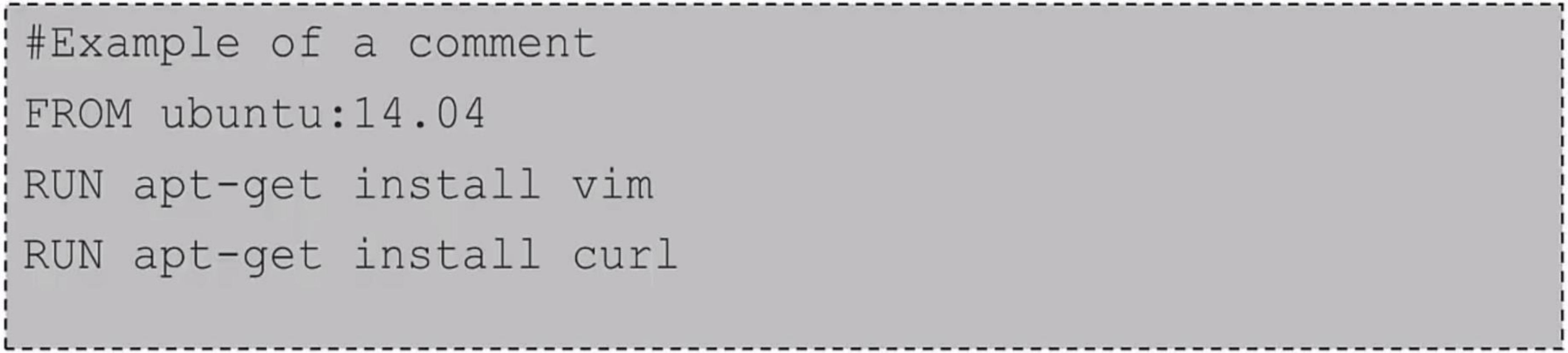
Intro to Dockerfile

*A **Dockerfile** is a configuration file that contains instructions for building a Docker image*

- Provides a more effective way to build images compared to using `docker commit`
- Easily fits into your continuous integration and deployment process

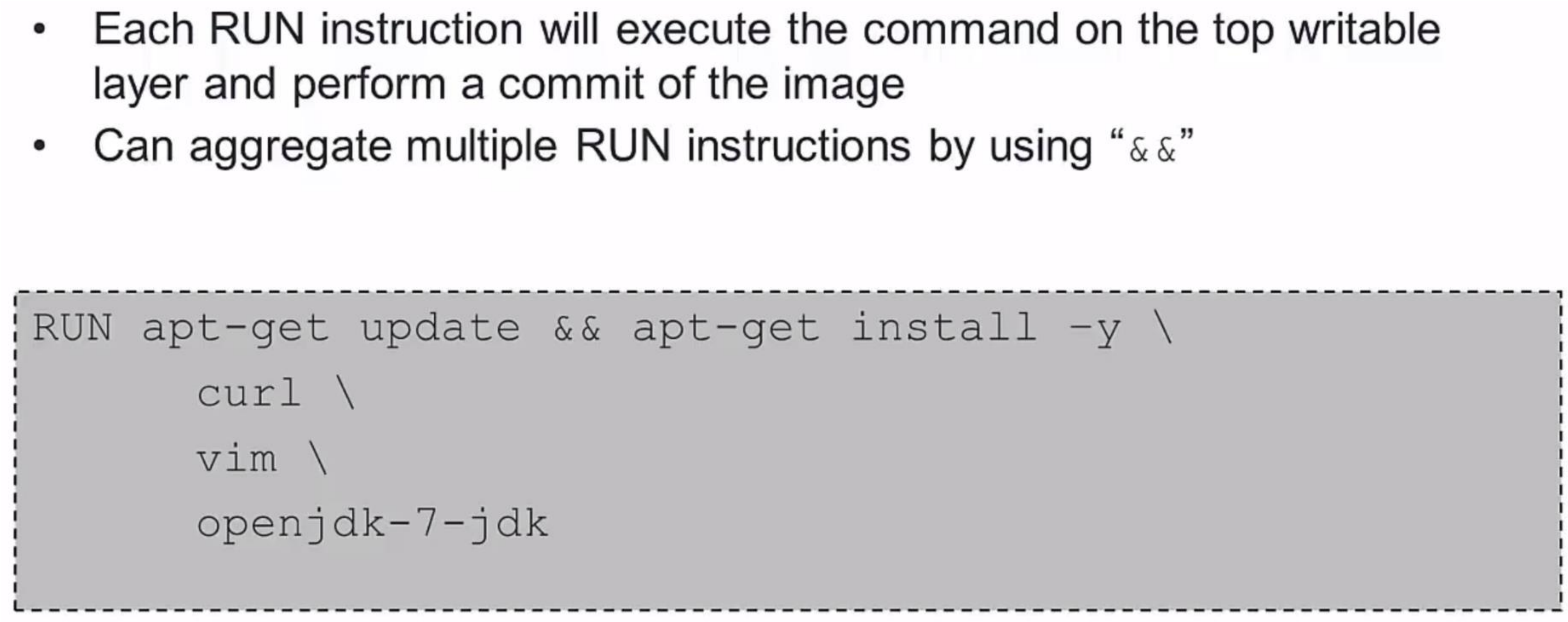
Dockerfile Instructions

- Instructions specify what to do when build the image
- FROM instruction specified what the base image should be
- RUN instruction specified a command to execute



```
#Example of a comment
FROM ubuntu:14.04
RUN apt-get install vim
RUN apt-get install curl
```

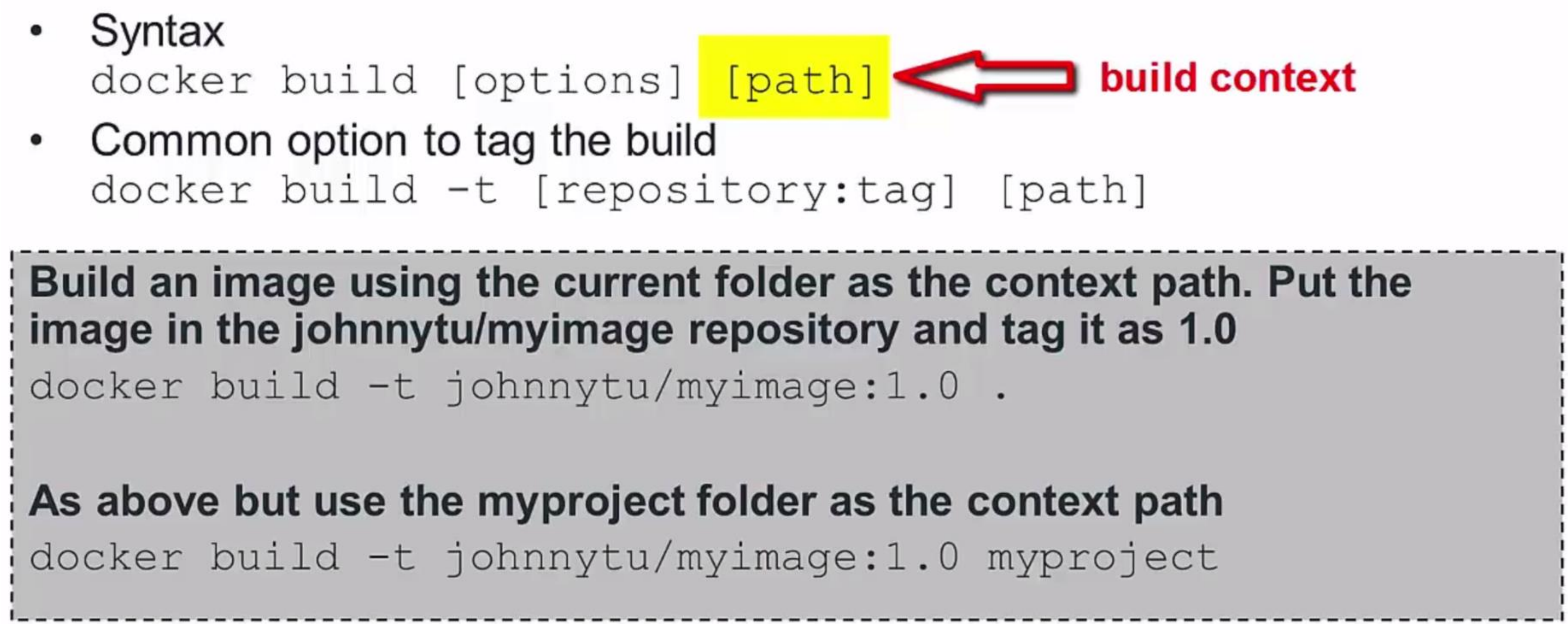
Run Instructions



- Each RUN instruction will execute the command on the top writable layer and perform a commit of the image
- Can aggregate multiple RUN instructions by using “&&”

```
RUN apt-get update && apt-get install -y \
     curl \
     vim \
     openjdk-7-jdk
```

Docker Build



- Syntax

```
docker build [options] [path]
```

build context

- Common option to tag the build

```
docker build -t [repository:tag] [path]
```

Build an image using the current folder as the context path. Put the image in the johnnytu/myimage repository and tag it as 1.0

```
docker build -t johnnytu/myimage:1.0 .
```

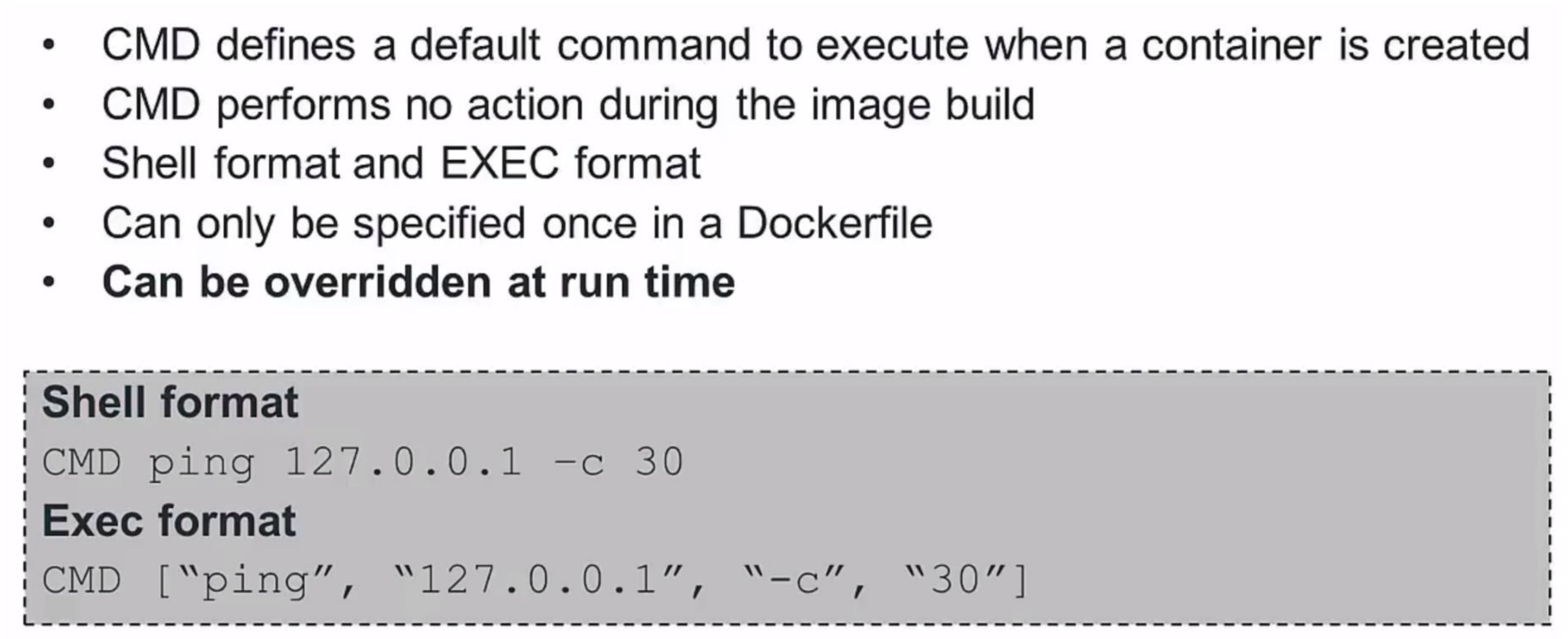
As above but use the myproject folder as the context path

```
docker build -t johnnytu/myimage:1.0 myproject
```

Build from Dockerfile

- Build your own image and verify the **curl** is inside the container

CMD Instructions



- CMD defines a default command to execute when a container is created
- CMD performs no action during the image build
- Shell format and EXEC format
- Can only be specified once in a Dockerfile
- **Can be overridden at run time**

Shell format

```
CMD ping 127.0.0.1 -c 30
```

Exec format

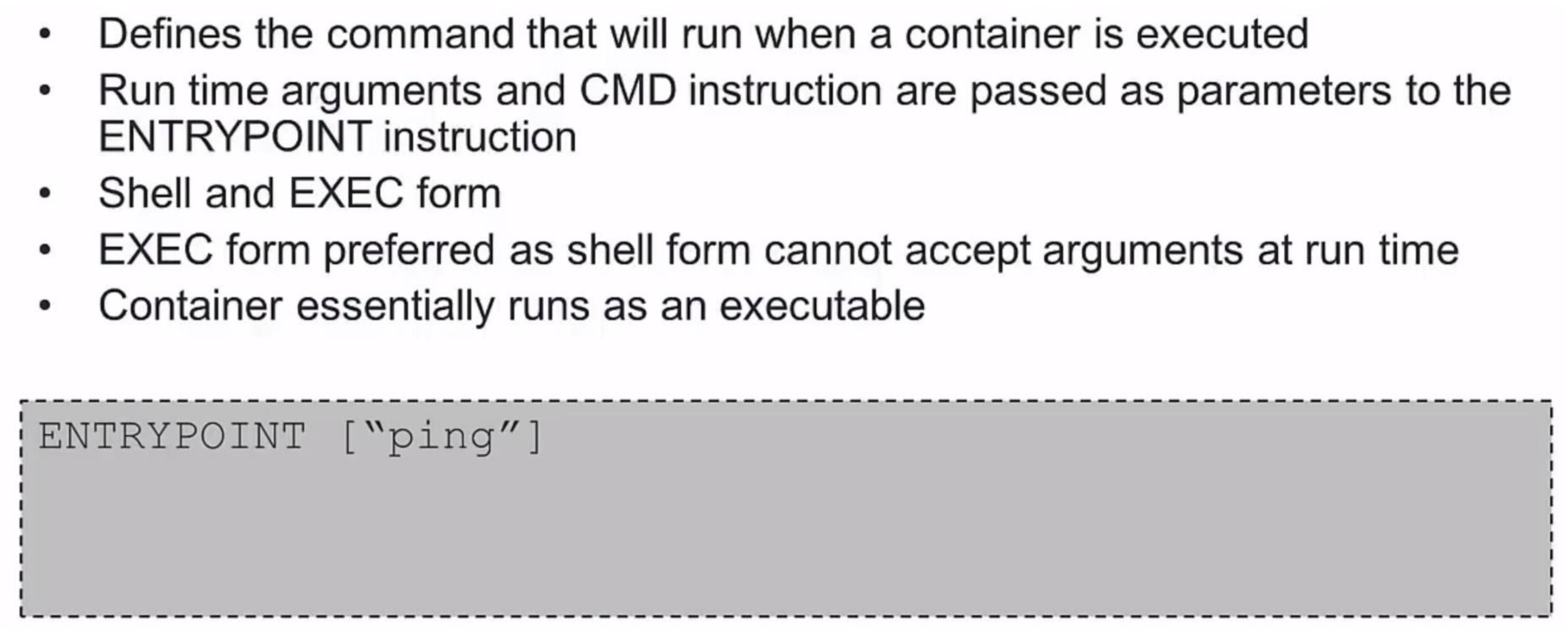
```
CMD [“ping”, “127.0.0.1”, “-c”, “30”]
```

EXERCISE

Try CMD

1. Go into the test folder and open your Dockerfile from the previous exercise
2. Add the following line to the end
   ```
   CMD [“ping”, “127.0.0.1”, “-c”, “30”]
   ```
3. Build the image
   ```
   docker build -t <yourname>/testimage:1.1 .
   ```
4. Execute a container from the image and observe the output
   ```
   docker run <yourname>/testimage:1.1
   ```
5. Execute another container from the image and specify the echo command
   ```
   docker run <yourname>/testimage:1.1 echo “hello world”
   ```
6. Observe how the container argument overrides the CMD instruction

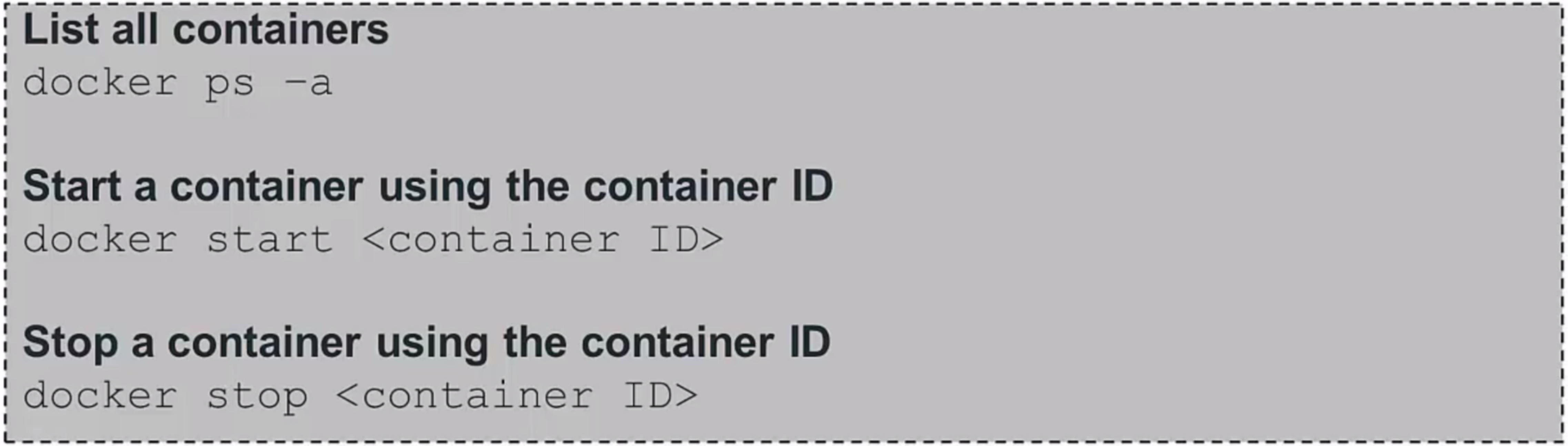
ENTRYPOINT Instruction



- Defines the command that will run when a container is executed
- Run time arguments and CMD instruction are passed as parameters to the ENTRYPOINT instruction
- Shell and EXEC form
- EXEC form preferred as shell form cannot accept arguments at run time
- Container essentially runs as an executable

```
ENTRYPOINT [“ping”]
```

Start and Stop Containers



List all containers

```
docker ps -a
```

Start a container using the container ID

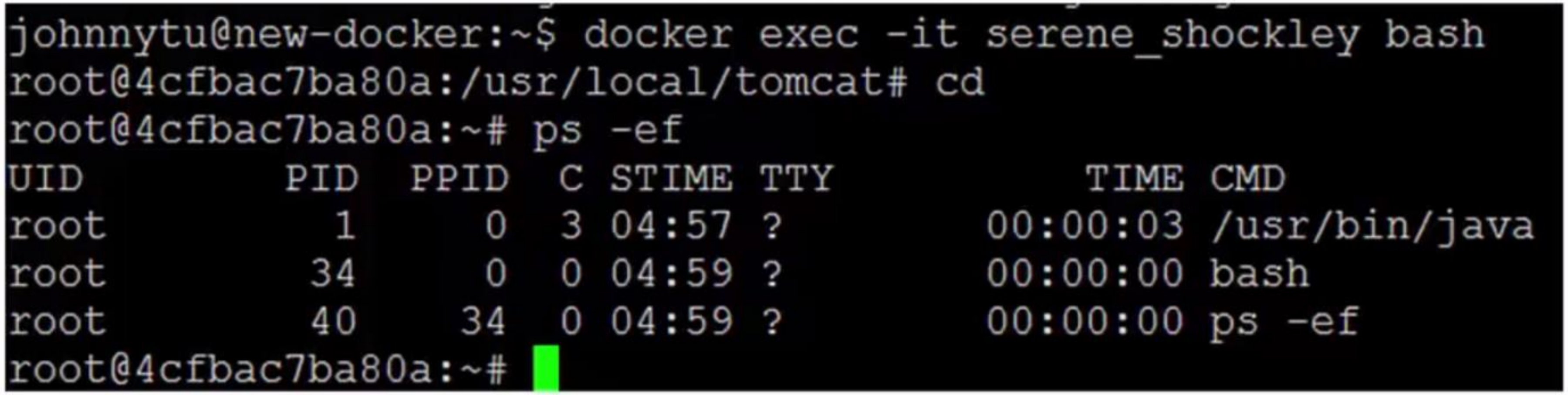
```
docker start <container ID>
```

Stop a container using the container ID

```
docker stop <container ID>
```

Getting terminal access

- Use **`docker exec`** command to start another process within a container
- Execute /bin/bash to get a bash shell
- `docker exec -i -t [container ID] /bin/bash`
- Exiting from the terminal will not terminate the container



```
johnnytu@new-docker:~$ docker exec -it serene_shockley bash
root@4cfbac7ba80a:/usr/local/tomcat# cd
root@4cfbac7ba80a:~# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  3 04:57 ?        00:00:03 /usr/bin/java
root        34     0  0 04:59 ?        00:00:00 bash
root        40    34  0 04:59 ?        00:00:00 ps -ef
root@4cfbac7ba80a:~#
```

Clean-up

- Containers

```
docker rm
docker rm $(docker ps -a –q)
```

- Images

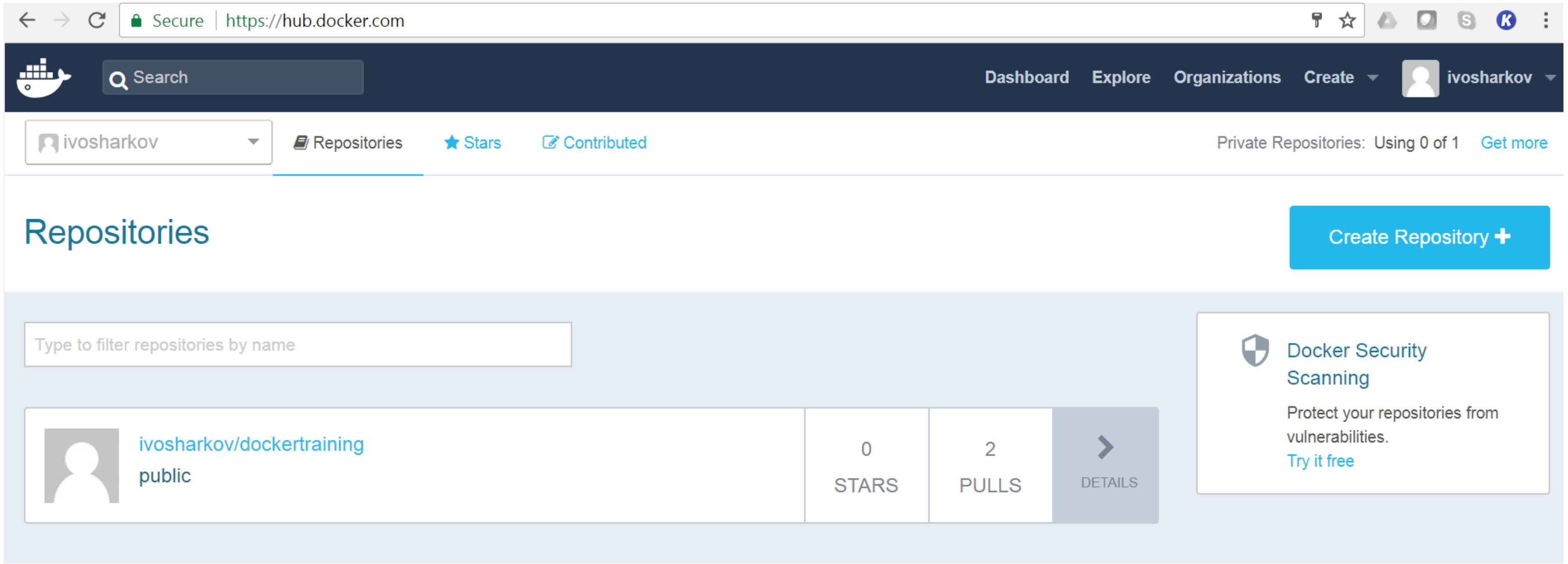
```
docker rmi
docker rmi $(docker images -q --filter "dangling=true")
```

- Volumes

```
docker volumes rm
docker volume rm $(docker volume ls -qf dangling=true)
```

Docker Hub Repositories

- Users can create their own repositories on Docker Hub
- Public vs Private
- Push local images to a repository



Private Repositories: Using 0 of 1

Repositories

ivosharkov/dockertraining
public

0 STARS

2 PULLS

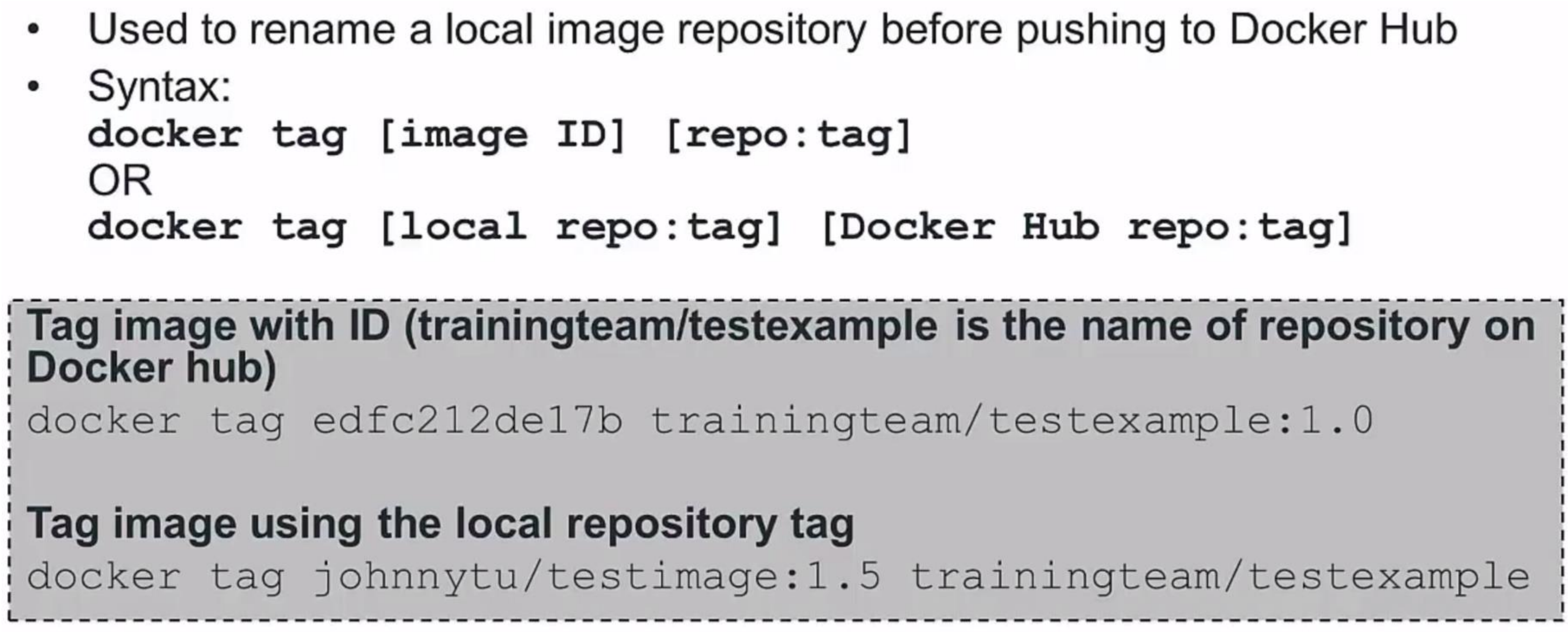
Docker Security Scanning

Protect your repositories from vulnerabilities.

Pushing Images to Docker Hub

- Use **docker push** command
- Syntax: **docker push repo:tag**
- Local repo **MUST** have the same name and tag as the Docker Hub repo

Tagging Images



- Used to rename a local image repository before pushing to Docker Hub
- Syntax:
  `docker tag [image ID] [repo:tag]`
  OR
  `docker tag [local repo:tag] [Docker Hub repo:tag]`

Tag image with ID (trainingteam/testexample is the name of repository on Docker hub)

```
docker tag edfc212de17b trainingteam/testexample:1.0
```

Tag image using the local repository tag

```
docker tag johnnytu/testimage:1.5 trainingteam/testexample
```

EXERCISE

Push to Docker Hub

1. Login to your Docker Hub account
2. Create a new public repository called “testexample”
3. Tag your local image to give it the same repo name as the repository you created on Docker Hub

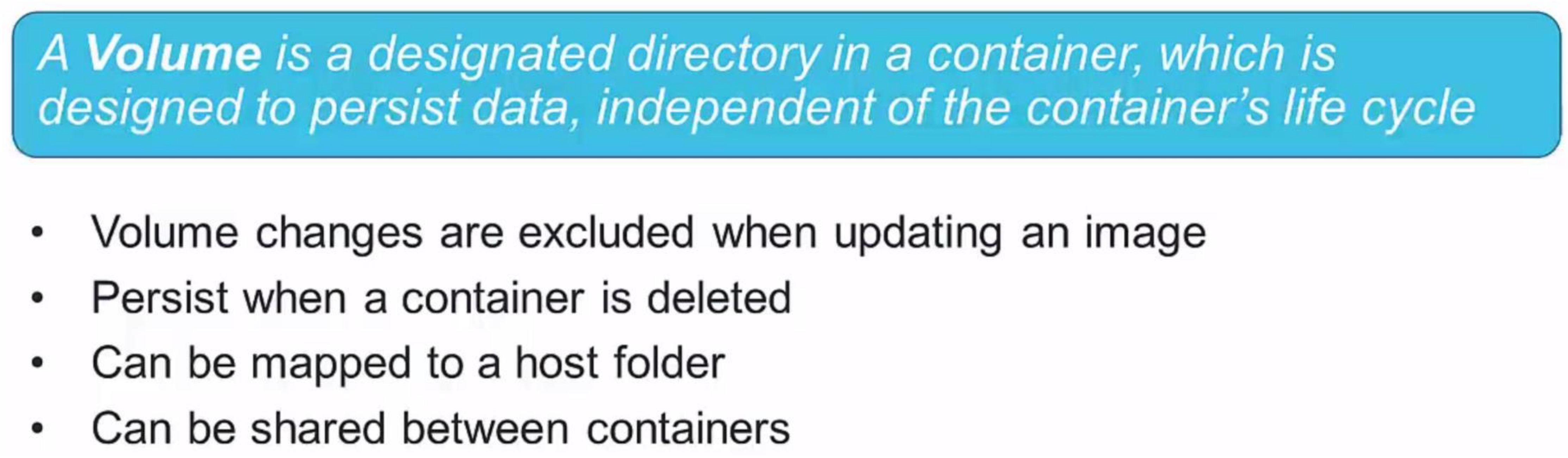
   ```
   docker tag <yourname>/testimage:1.1
   <yourname>/testexample:1.1
   ```

4. Push the new image to Docker Hub

   ```
   docker push <yourname>/testexample:1.1
   ```

5. Go to your Docker Hub repository and check for the tag

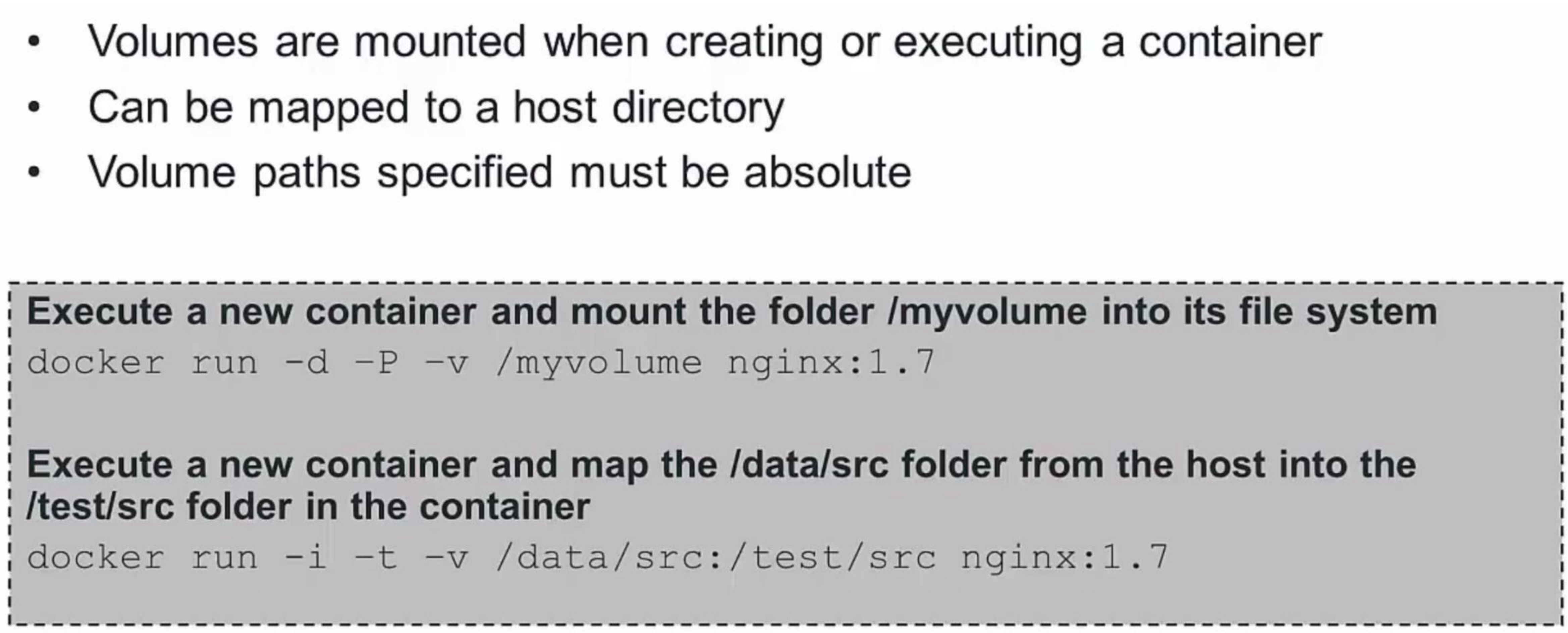
Volumes



*A **Volume** is a designated directory in a container, which is designed to persist data, independent of the container's life cycle*

- Volume changes are excluded when updating an image
- Persist when a container is deleted
- Can be mapped to a host folder
- Can be shared between containers

Mount a Volume



- Volumes are mounted when creating or executing a container
- Can be mapped to a host directory
- Volume paths specified must be absolute

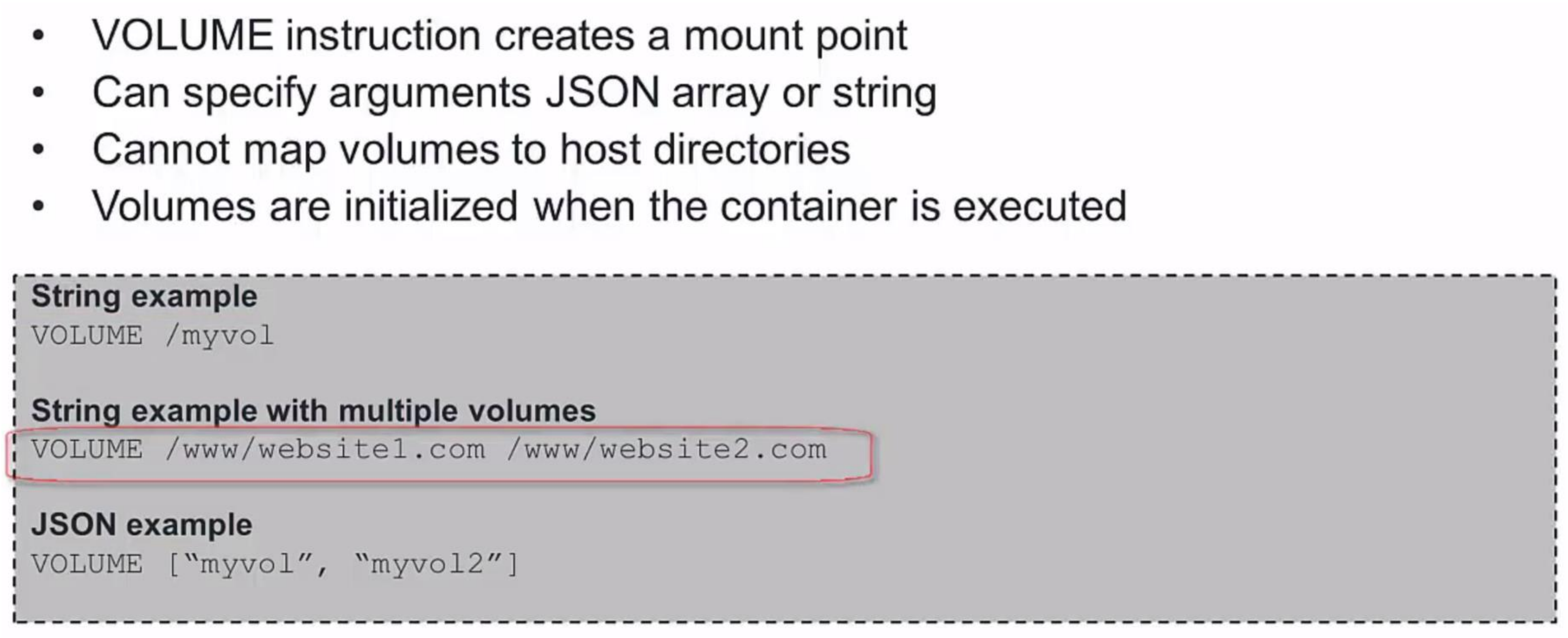
Execute a new container and mount the folder /myvolume into its file system

```
docker run -d -P -v /myvolume nginx:1.7
```

Execute a new container and map the /data/src folder from the host into the /test/src folder in the container

```
docker run -i -t -v /data/src:/test/src nginx:1.7
```

Volumes in Dockerfile



- VOLUME instruction creates a mount point
- Can specify arguments JSON array or string
- Cannot map volumes to host directories
- Volumes are initialized when the container is executed

String example

```
VOLUME /myvol
```

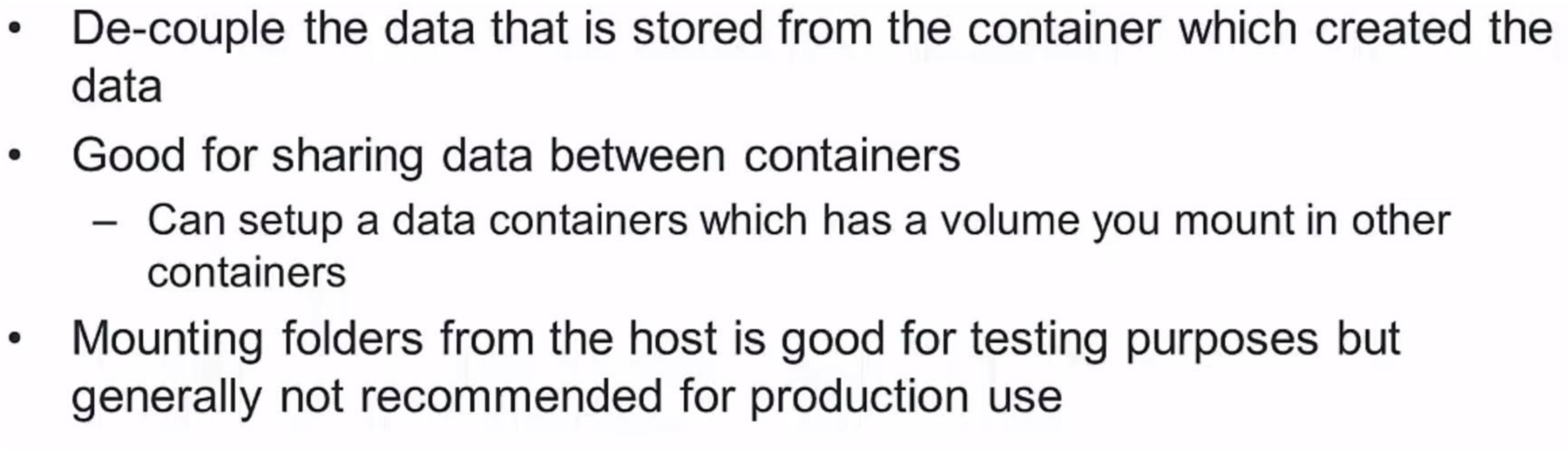
String example with multiple volumes

```
VOLUME /www/website1.com /www/website2.com
```

JSON example

```
VOLUME [“myvol”, “myvol2”]
```

Uses of volumes



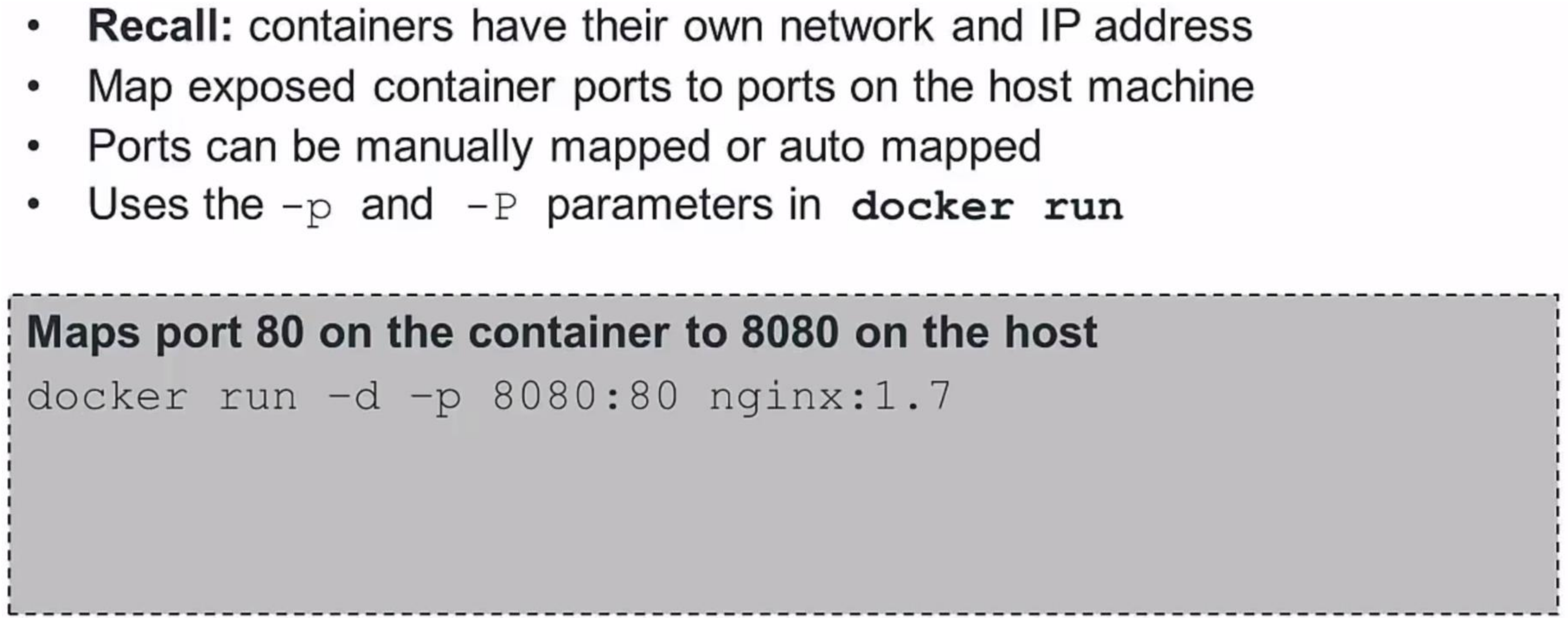
- De-couple the data that is stored from the container which created the data
- Good for sharing data between containers
  - Can setup a data containers which has a volume you mount in other containers
- Mounting folders from the host is good for testing purposes but generally not recommended for production use

EXERCISE

Create and test a Volume

1. Execute a new container and initialise a volume at /www/website.
   Run a bash terminal as your container process
   ```
   docker run -i -t -v /www/website ubuntu:14.04 bash
   ```
2. Inside the container, verify that you can get to /www/website
3. Create a file inside the `/www/website` folder
4. Exit the container
5. Commit the updated container as a new image called test and tag it as 1.0
   ```
   docker commit <container ID> test:1.0
   ```
6. Execute a new container with your test image and go into it’s bash shell
   ```
   docker run -i -t test:1.0 bash
   ```
7. Verify that the `/www/website` folder exists and that there are no files inside

Mapping ports

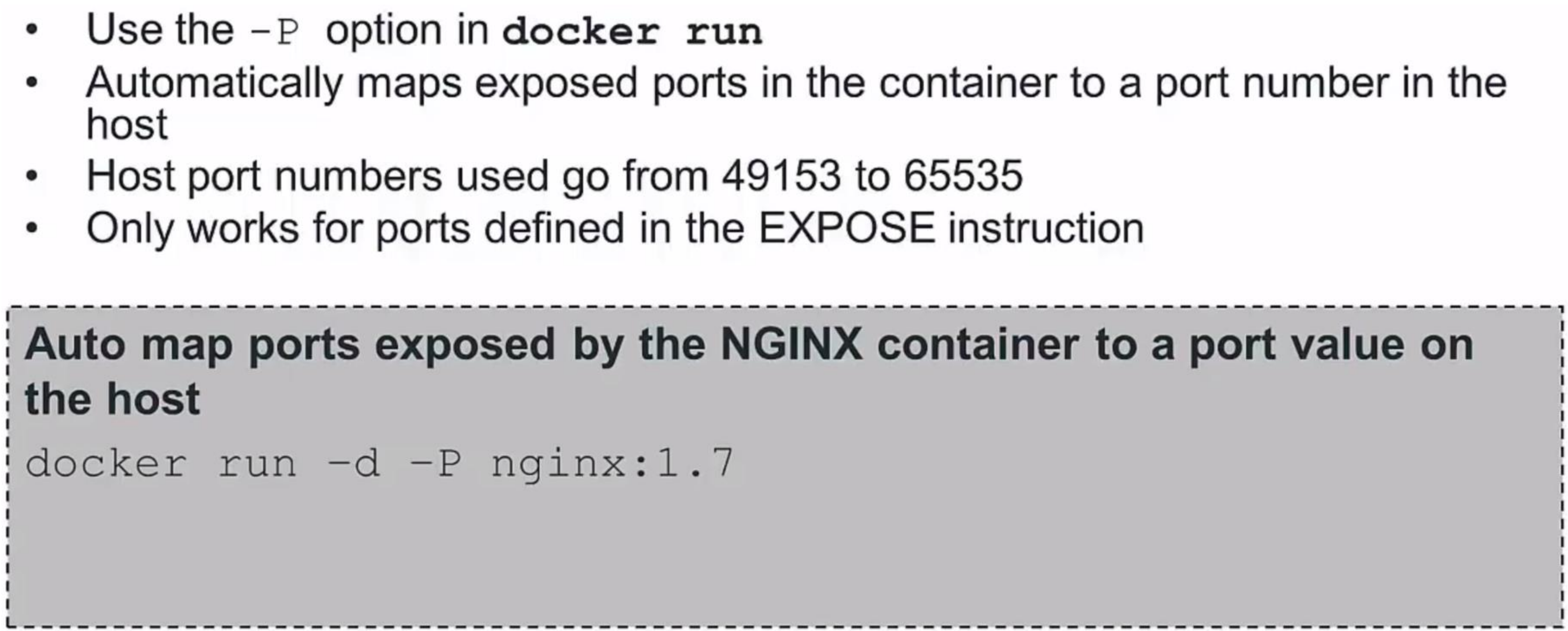


- **Recall:** containers have their own network and IP address
- Map exposed container ports to ports on the host machine
- Ports can be manually mapped or auto mapped
- Uses the `-p` and `-P` parameters in **`docker run`**

Maps port 80 on the container to 8080 on the host

```
docker run -d -p 8080:80 nginx:1.7
```

Automapping ports

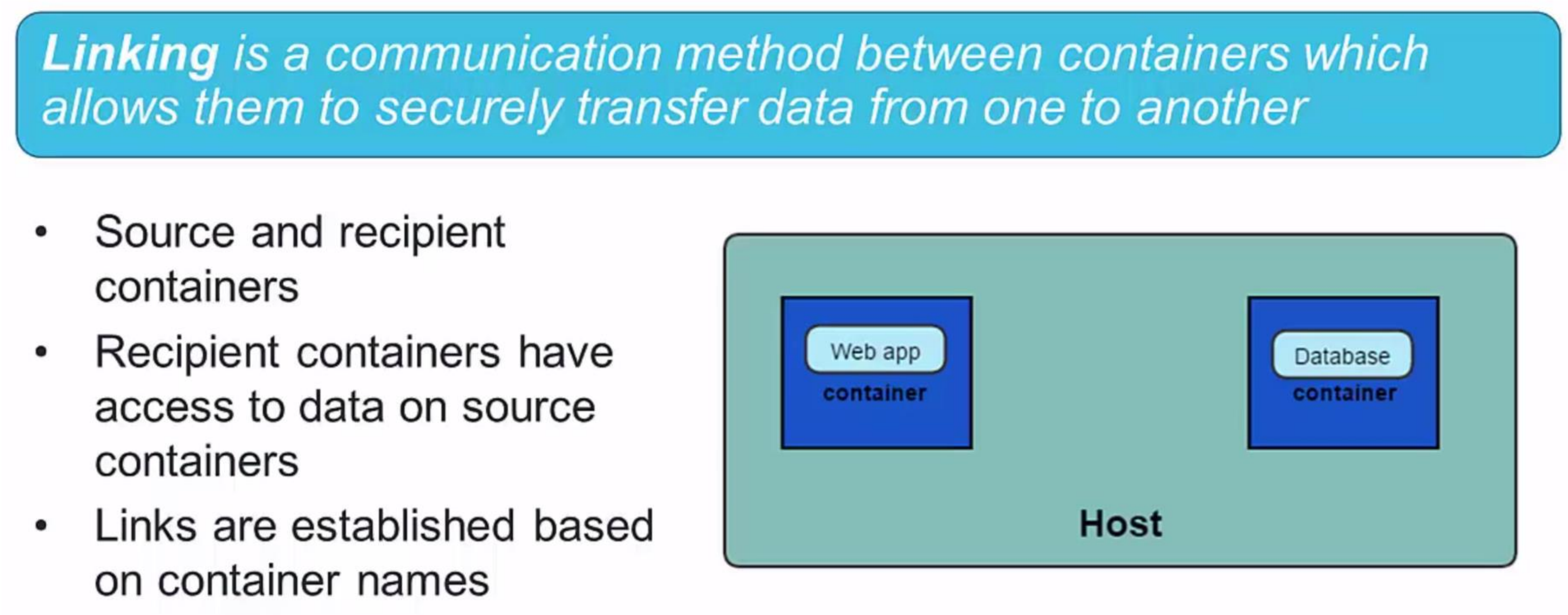


- Use the `-P` option in **`docker run`**
- Automatically maps exposed ports in the container to a port number in the host
- Host port numbers used go from 49153 to 65535
- Only works for ports defined in the EXPOSE instruction

Auto map ports exposed by the NGINX container to a port value on the host

```
docker run -d -P nginx:1.7
```

Linking Containers



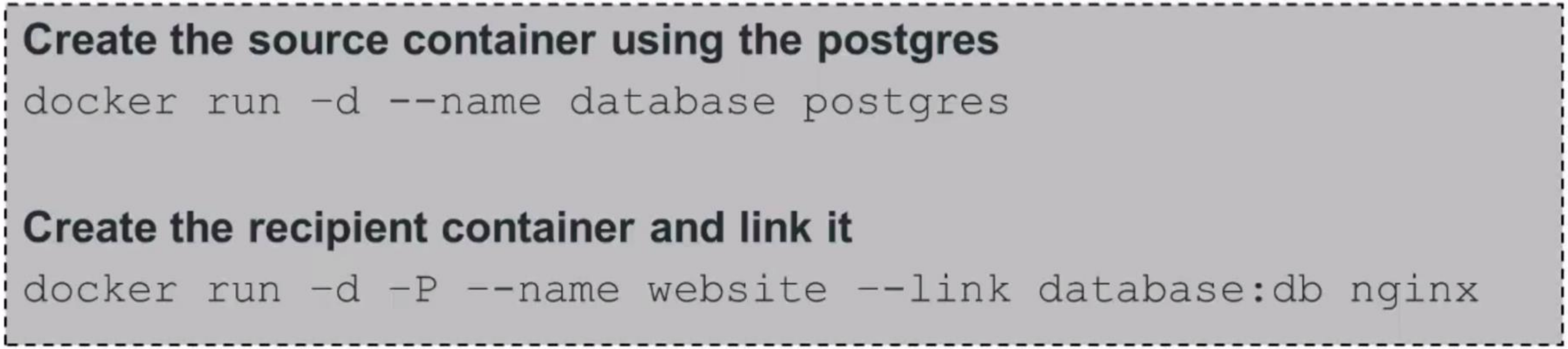
***Linking** is a communication method between containers which allows them to securely transfer data from one to another*

- Source and recipient containers
- Recipient containers have access to data on source containers
- Links are established based on container names

Create a Link

- Create the source container first
- Create the recipient container and use the **--link** option

- Best practice – give your container meaningful names



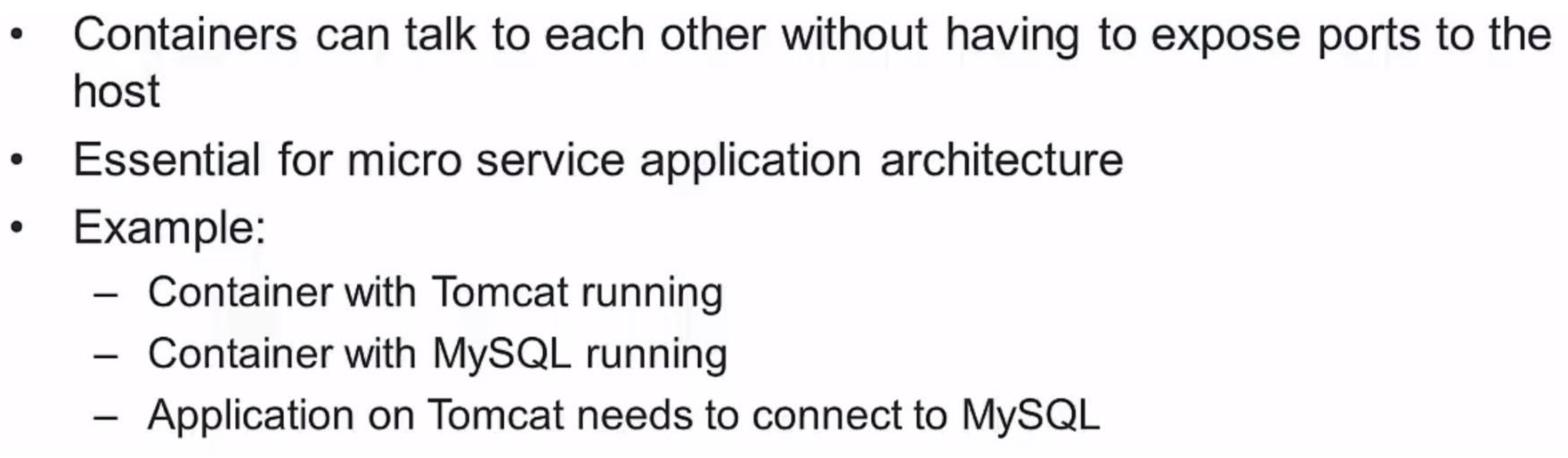
Create the source container using the postgres

```
docker run -d --name database postgres
```

Create the recipient container and link it

```
docker run -d -P --name website --link database:db nginx
```

Linking Use cases



- Containers can talk to each other without having to expose ports to the host
- Essential for micro service application architecture
- Example:
  - Container with Tomcat running
  - Container with MySQL running
  - Application on Tomcat needs to connect to MySQL

EXERCISE

Link two Containers

1. Run a container in detached mode using the `postgres` image. Name the container “dbms”

   ```
   docker run -d --name dbms postgres
   ```

2. Run another container using the Ubuntu image and link it with the “dbms” container. Use the alias “db”, run the bash terminal as the main process

   ```
   docker run -it --name website --link dbms:db
   ubuntu:14.04 bash
   ```

3. In the “website” container terminal, open the `/etc/hosts` file

4. What can you observe?

Make hands dirty

- Start Nginx –p 555:80 and Jenkins –p 8000:8080 images with persistent volumes on the file system
- Ensure you are able to login in localhost:8000 and the data will endurance docker rm –f jenkins
- Modify Nginx configuration so any request to http://localhost:555/jenkins should be redirected to our Jenkins service

Questions

References

- Dockerfile best practices

  https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

- CMD vs Entry

  https://www.ctl.io/developers/blog/post/dockerfile-entrypoint-vs-cmd/

- Books for free

  http://gen.lib.rus.ec/

- Google ☺