Docker Linking and Stateful Containers

HOW WE LINK AND CREATE STATEFUL CONTAINERS

Agenda

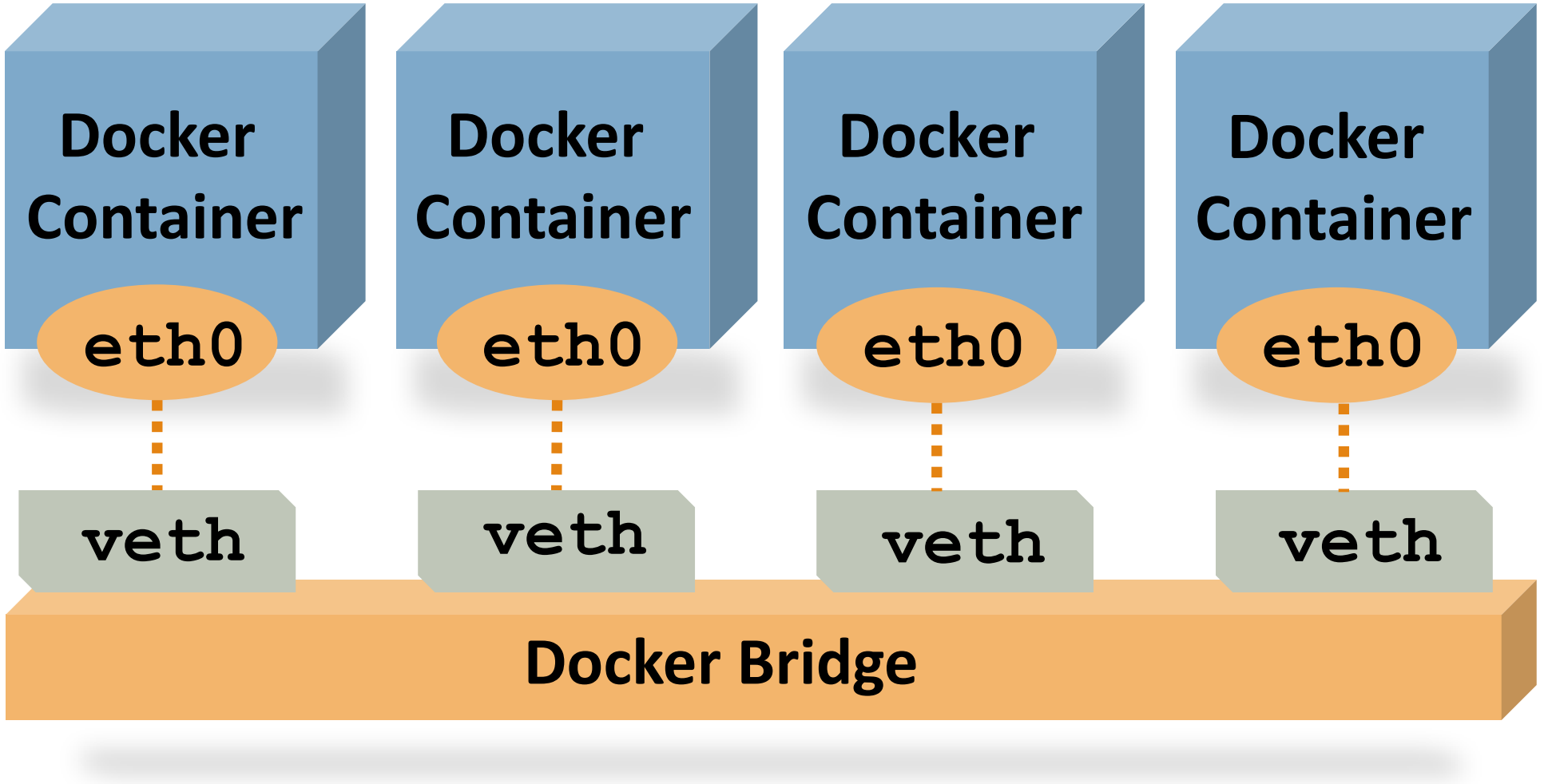
Intro / Prep Environments

Day 1: Docker Deep Dive

Day 2: Docker Advanced Deep Dive

Recap

Quick Look at Docker Networking



Linking Containers

- We want containers to communicate with each other
- Each container has an IP (veth/eth0)
- Containers can expose virtual ports
- Docker bridge networking can link containers without going over host network
- How do we discover IP addresses, etc?

Linking Containers

- Here's Docker's `link` syntax:

```
docker run --link <name or id>:<alias>
```

- Best practice is to name your containers, using (--`name`) links depend on it
- And `alias` is what your containers will see as environment variables

Linking Containers

- Example:

```
docker run --rm --name web2 --link db:db training/webapp env
```

- Creates the following environment variables:

```
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5432_TCP=tcp://172.17.0.5:5432
DB_PORT_5432_TCP_PROTO=tcp
DB_PORT_5432_TCP_PORT=5432
DB_PORT_5432_TCP_ADDR=172.17.0.5
```

- Your applications can then use environment variables to discover the dependent containers/services

Linking Containers with DNS

- Let's pretend to run the following:

```
docker run -t -i --rm --link db:webdb training/webapp /bin/bash
```

- If we check out `/etc/hosts` **inside** the container, we'd find:

```
172.17.0.7 aed84ee21bde
...
172.17.0.5 webdb 6e5cdeb2d300 db
```

- So we can just refer to containers by name: `http://webdb`

Linking Containers Considerations

- **Injecting** environment variables is a very powerful concept
- Can link multiple containers together
- The linking happens between **one** host only

Examples of Linking

- Let's run a database service which can be used by a Java EE Application:

```
docker run --name mysqldb -e MYSQL_USER=mysql -e
MYSQL_PASSWORD=mysql -e MYSQL_DATABASE=sample -e
MYSQL_ROOT_PASSWORD=supersecret -p 3306:3306 -d mysql
```

  - NOTE: We may need to forward the `mysql` port `3306`, if using a VM.

- Now, let's look at what it'd take to link up a Java EE Application:

```
docker run -d --name mywildfly --link mysqldb:db -p 8080:8080
arungupta/wildfly-mysql-javaee7
```

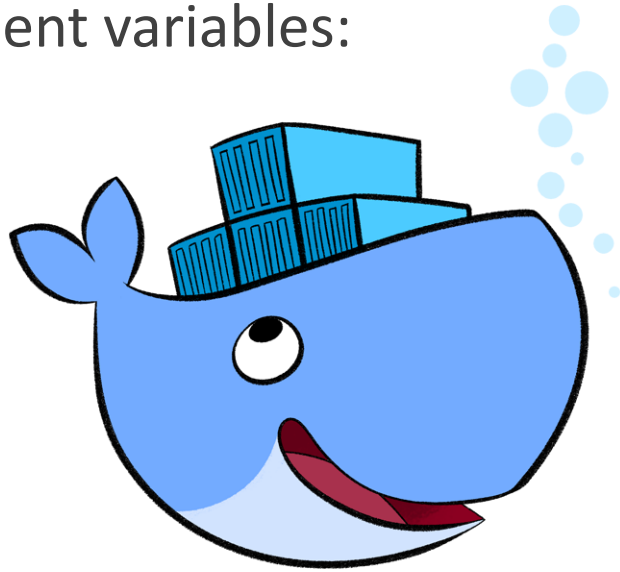
Examples of Linking

- Our app is now using the DB, but let's log into the container and verify the environment variables/DNS are set up:

```
docker exec -it mywildfly bash
```

- Then we'd type the following to list environment variables:

```
# env
```



Examples of Linking

- You can also take a look at the `/etc/hosts` file:

```
# cat /etc/hosts
```

- Output:

```
172.17.0.30     c924917fe4ad
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.28     db ef59a1b98326 mysqldb
172.17.0.30     mywildfly.bridge
172.17.0.18     registry
172.17.0.18     registry.bridge
172.17.0.28     mysqldb
172.17.0.28     mysqldb.bridge
172.17.0.30     mywildfly
```

Docker Containers Die

- Containers are ephemeral
- Nothing is saved from a container if it goes away
- Containers get new IP addresses
- Don’t treat containers like VMs: They are NOT
- But what about stateful applications?

Confronting the Stateful Challenge

- Docker volumes to the **rescue** of all our stateful containers
- Volumes create persistent data outside of the container
- Can be mapped directly to host locations
- Can also be deployed independently of hosts

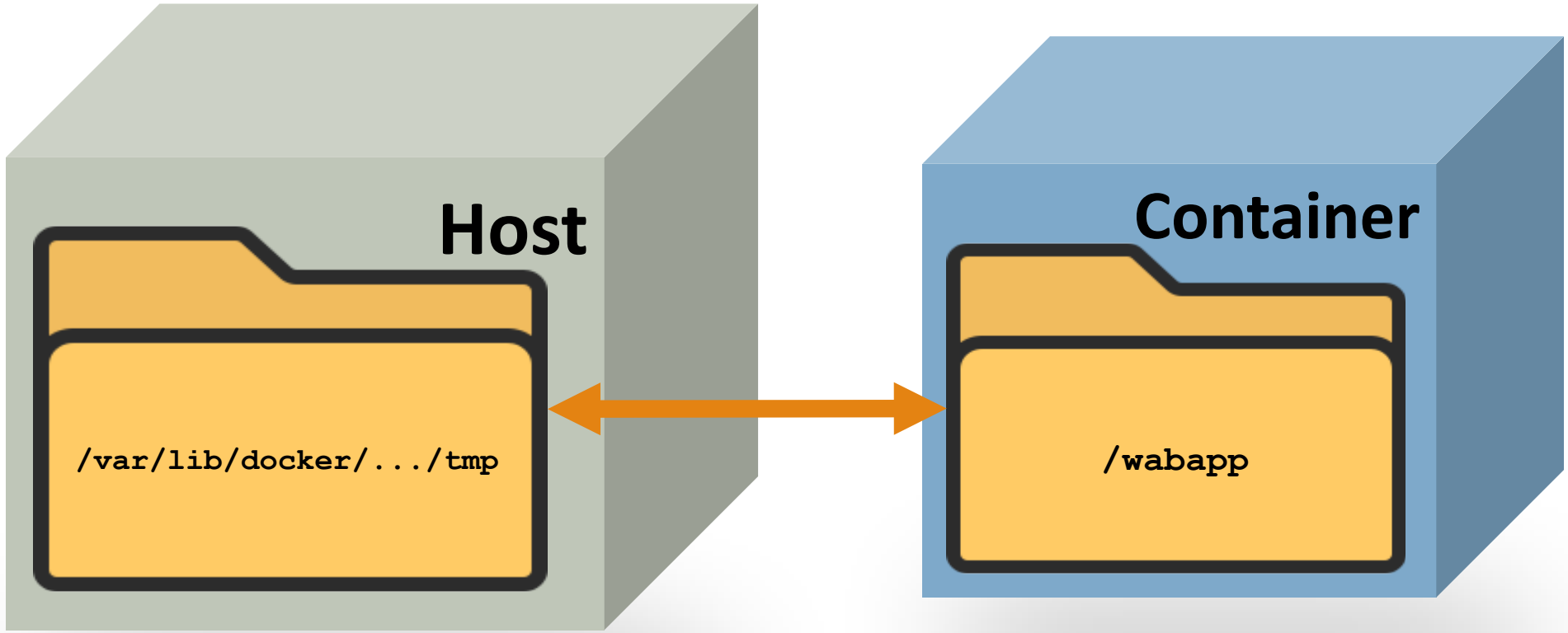
- Example:

```
docker run -d -P --name web -v /webapp training/webapp python app.py
```

Docker Data Volume

- Example:

```
docker run -d -P --name web -v /webapp training/webapp python app.py
```



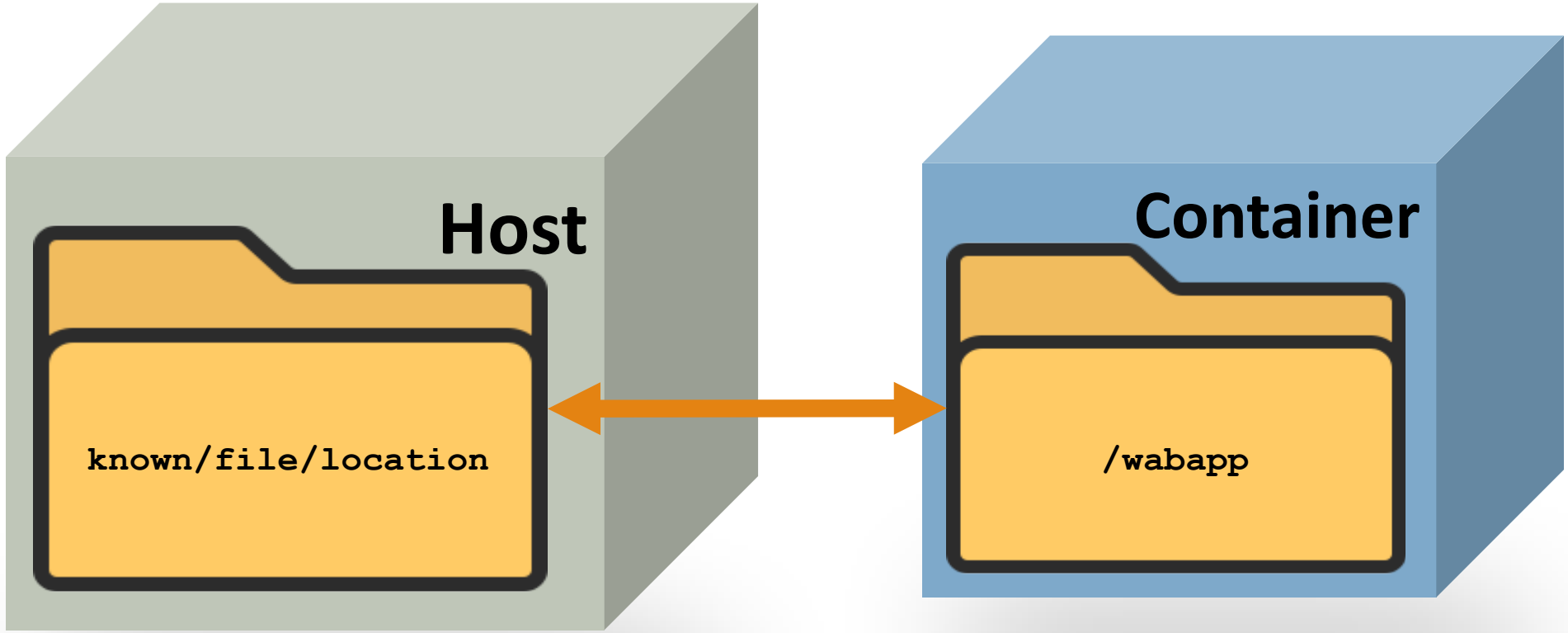
Direct Host Volumes

- We can also map volumes **directly** to host specific locations:
- Useful for known locations on host
- Can use NFS mounts
- Files are visible outside of the container and are persisted
- Can restart new containers up with same location

Docker Host Volumes Direct Location

- Example:

```
docker run -d -P --name web -v /src/webapp:/opt/webapp
training/webapp python app.py
```



Containers as Data Volumes

- Start a container that will manage the volume:

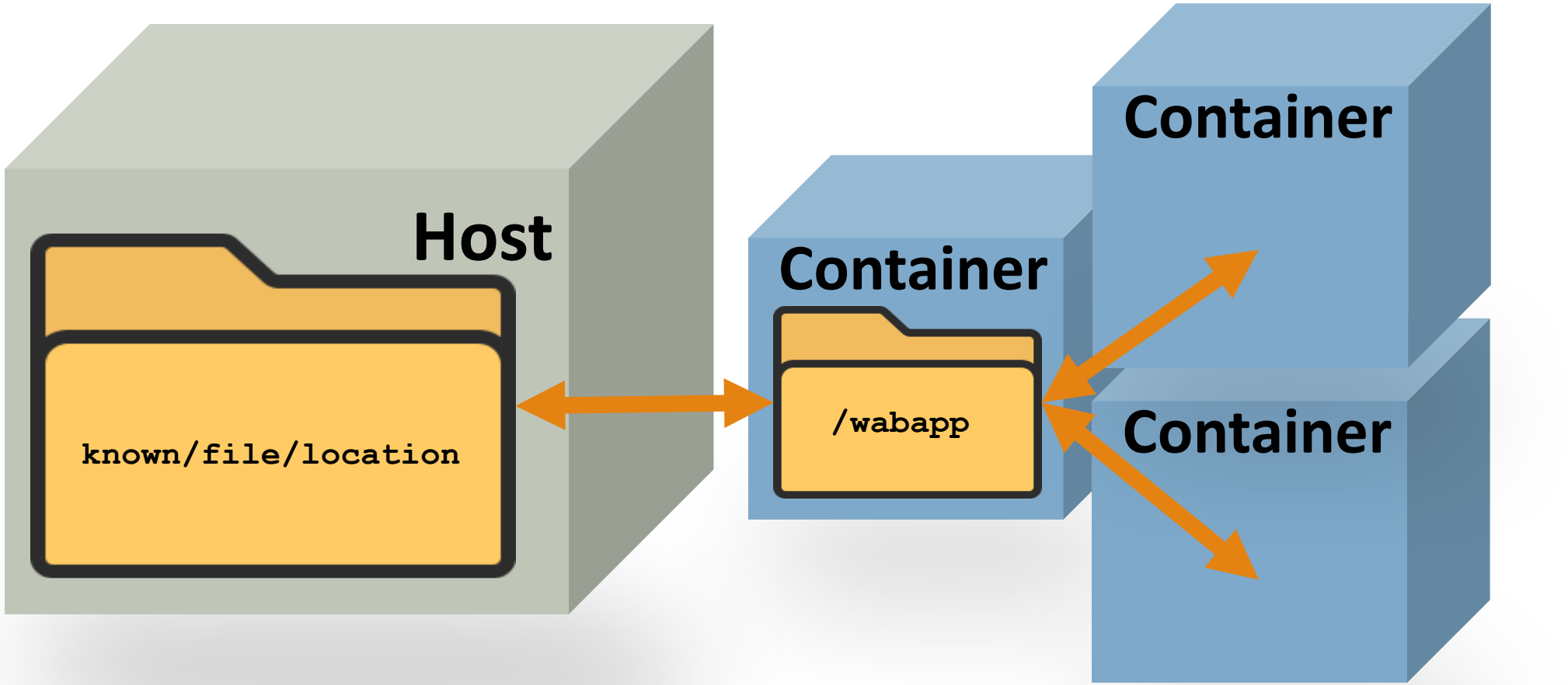
```
docker create -v /dbdata --name dbdata training/postgres
/bin/true
```

- Now other containers can use that container so they're not tied directly to the volumes (mounting them, etc):

```
docker run -d --volumes-from dbdata --name db1 training/postgres
```

Containers as Data Volumes

- Here's a diagram of what that might look like:



Inspect Volumes

- How can we inspect our volumes? Let's first list our current volumes:

```
docker volume ls
```

- Now, pulling from the list that prints out, we can plugin a volume id, like this:

```
docker volume inspect <name-or-id>
```

Jenkins With Volumes

- Let's start by running our Jenkins images from an earlier lesson, like this:

```
docker run -d --name myjenkins -p 8080:8080 jenkins
```

- We can save the changes and jobs that Jenkins creates by adding a host volume:

```
docker run -d --name myjenkins -p 8080:8080 \
 -v /your/home:/var/jenkins_home jenkins
```

- Now when you run Jenkins, you can stop, destroy, and re-run Jenkins and your build jobs should be there.

Final Notes

- Docker runs on a single host.
- Not only can `/var/lib/docker` be managed, it needs to be!
- Use only what you need in your images, avoiding image bloat
- Don’t get into a habit of running as `root`.
- Be careful with Docker Hub and use trusted images whenever possible

Final Notes

- Container security - Containers do not "contain" their contents.
- Always use image tags, whenever humanly possible.
- Use sanity scripts to boot your process from within container.
- One task per container is considered The Docker Way.

Lab

End of Chapter