

Docker Essentials

Course Agenda

Day 1

- ❑ Introduction to Class
- ❑ Docker Containers
- ❑ Application Server in Docker Container
- ❑ Containerizing Applications
- ❑ Docker Networking and Stateful Containers

Day 2

- ❑ Docker Hub and Creating a Registry
- ❑ Dockerfile and Compose
- ❑ Docker on the Cloud
- ❑ A Taste of Docker Swarm

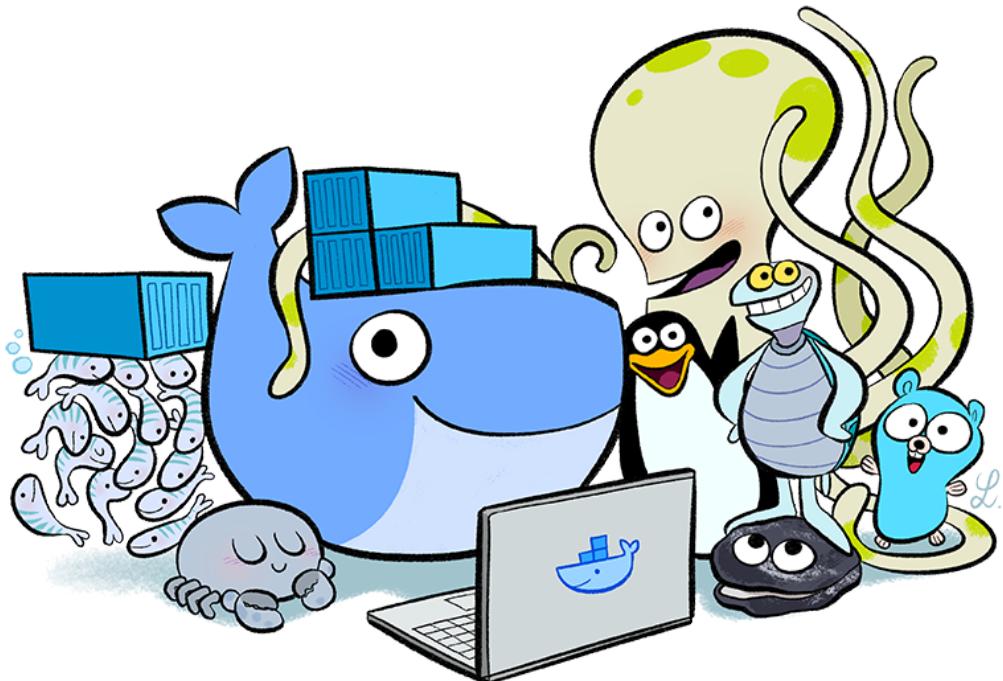


Docker Containers

DISCOVERING DOCKER CONTAINERS

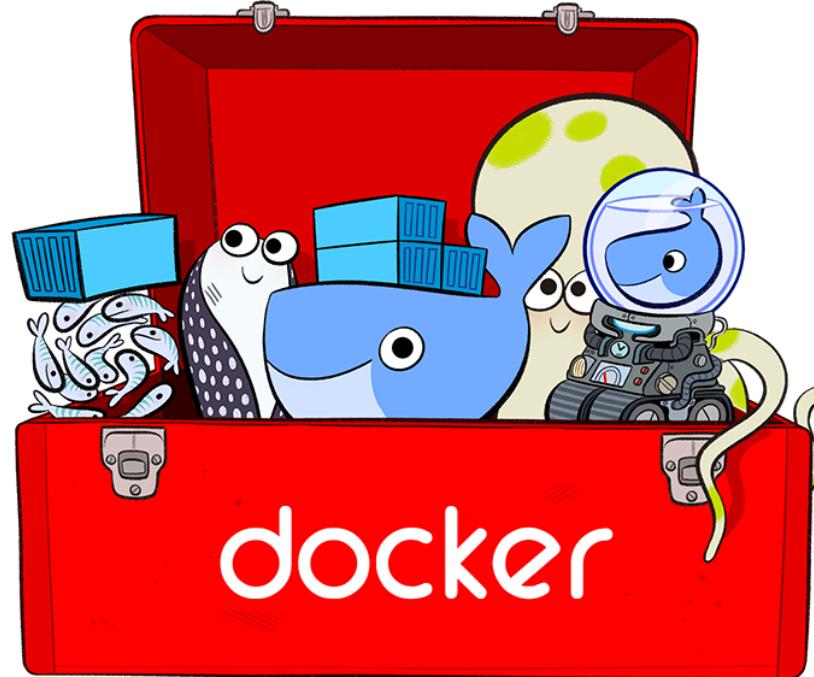
Getting to Know Each Other

- Welcome!
- A little about myself...
- Tell me about yourself...



Course Logistics

- Lecture Time
- Lab Time
- Lab Environment
- Short Breaks
- Time to eat!
- Bathroom Breaks



End of Chapter

Docker Containers

DISCOVERING DOCKER CONTAINERS

Installing Docker Platform

- ❑ Docker is available in two editions:

Community Edition (CE)

- ❑ Docker Community Edition (CE) is ideal for developers and small teams looking to get started
- ❑ Docker CE has two update channels, **stable** and **edge**:
 - ❑ **Stable** gives you reliable updates every quarter
 - ❑ **Edge** gives you new features every month

Enterprise Edition (EE)

- ❑ Docker Enterprise Edition (EE) is designed for enterprise grade development
- ❑ IT teams who build, ship, and run business critical applications in production at scale

What's Included?

- ❑ Docker for Mac and Docker for Windows
 - ❑ Compose - included with Docker install (since Docker 1.12+)
 - ❑ Swarm - included with Docker install (since Docker 1.12+)
 - ❑ Machine - included with Docker install (since Docker 1.12+)
- ❑ Docker Linux Distributions
 - ❑ Compose - installed separately
 - ❑ Swarm - included with Docker install (since Docker 1.12+)
 - ❑ Machine - installed separately

About DockerHub

- ❑ Docker Hub is a cloud-based registry service
- ❑ Allowing you to link to code repositories, build your images and test them, storing manually pushed images
- ❑ Also, link to Docker Cloud so you can deploy images to your hosts
- ❑ It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline

Docker Hub Major Features

Image Repositories: Find and pull images from community and official libraries, and manage, push to, and pull from private image libraries to which you have access

Automated Builds: Automatically create new images when you make changes to a source code repository

Webhooks: A feature of Automated Builds, Webhooks let you trigger actions after a successful push to a repository

Organizations: Create work groups to manage access to image repositories

GitHub and Bitbucket Integration: Add the Hub and your Docker Images to your current workflows.

Creating a Docker ID

- ❑ Logging into Docker Hub and Docker Cloud require a Docker ID
- ❑ Your Docker ID becomes your user namespace for hosted Docker services, and becomes your username on the Docker Forums; register like this:
 1. Go to the Docker Cloud sign up page.
 2. Enter a username that is also your Docker ID.
 3. Your Docker ID must be between 4 and 30 characters long, and can only contain numbers and lowercase letters.
 4. Enter a unique, valid email address.
 5. Enter a password between 6 and 128 characters long.
 6. Click **Sign up**.
 7. Docker sends a verification email to the address you provided.
 8. Click the link in the email to verify your address.

More on Docker Hub

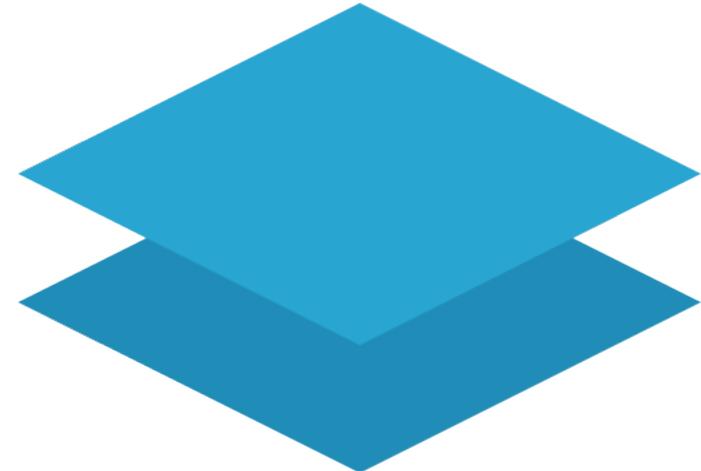
- ❑ You can find public repositories and images from Docker Hub in two ways:
 1. You can “Search” from the Docker Hub website
 2. Use the Docker CLI to run the `docker search` command
- ❑ For example if you were looking for an `ubuntu` image, you might run the following command line search:

```
$ docker search ubuntu
```

- ❑ You can configure Docker Hub repositories in two ways:
 1. **Repositories**, which allow you to push images from a local Docker daemon to Docker Hub
 2. **Automated Builds**, which link to a source code repository and trigger an image rebuild process on Docker Hub when changes are detected in the source code

Working with Docker Images

- ❑ Docker images are the **basis** of containers
- ❑ An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime
- ❑ An image typically contains a union of layered filesystems stacked on top of each other
- ❑ An image does not have state and it never changes
- ❑ But, how do we work with them?



Working with Docker Images

- We can see the available image commands, by using the following command:

```
$ docker image
```

- So, of the most common commands though are listed below:

docker image build

Build an image from a Dockerfile

docker image history

Show the history of an image

docker image inspect

Display detailed information on one or more images

docker image ls

List images

docker image pull

Pull an image or a repository from a registry

docker image push

Push an image or a repository to a registry

docker image rm

Remove one or more images



Ways to Create an Image

- ❑ Dockerfile is the most common way to create a base image, and there's two ways to do that:
 1. Utilizing a **parent image** referenced by the FROM directive in the Dockerfile
 2. A **base image**, either has no FROM line in its Dockerfile, or has FROM scratch
- ❑ (We will learn much more about Dockerfiles in the coming lectures!)
- ❑ Also, images can be saved using `image save` command to a `.tar` file:

```
$ docker image save helloworld > helloworld.tar
```
- ❑ These tar files can then be imported using `load` command:

```
$ docker image load -i helloworld.tar
```

Basic Commands

- See local repository of images

```
$ docker images
```

- Check to see if containers are running

```
$ docker ps
```

- Pull a container down

```
$ docker pull <reponame/imagename:tag>
```

- Check system-wide information

```
$ docker info
```

- Or course, if you want to see all the available Docker commands:

```
$ docker
```

Basic Commands

- ❑ How do we “get in” our container?
 - ❑ By spinning off an image using ‘run’:

```
$ docker run <image-name, or image-id>
```

- ❑ Container is running, but where is it?
 - ❑ Enter active containers with ‘exec -it’:

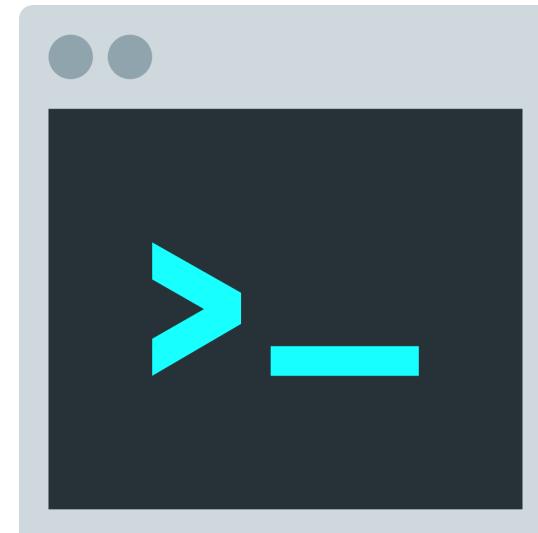
```
$ docker exec -it <image-name, or image-id>
```

Basic Commands

- ❑ You know you are “in” when you see something, like this:

```
root@<container-name, or container-id>
```

- ❑ Now that we are “in” the container, we can use all our trusty Linux commands like, ll, cd, vi, and apt-get...
- ❑ Look around and update your system using apt-get



Basic Commands

- ❑ When you first run an image you can:
 - ❑ Add ports, name the container, set hostname, select a network and more
 - ❑ We can also use ‘-it’ to “get in” to our container immediately
 - ❑ An example of how that might look:

```
$ docker run -it --network networkname -p 8000 \
    --name name-container \
    --hostname name-host \
    reponame/image-name:tag
```

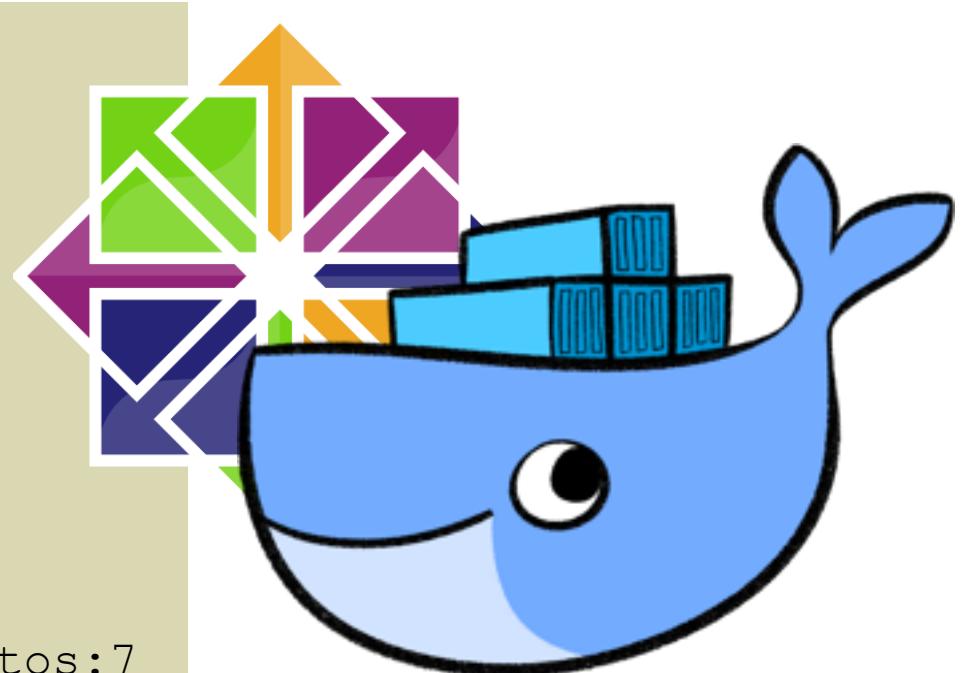
Pulling Our First Container

- ❑ Let's pull a Centos 7 image from DockerHub, like this:

```
docker pull centos:7
```

- ❑ Output:

```
Pulling from library/centos
fa5be2806d4c: Pull complete
0cd86ce0a197: Pull complete
e9407f1d4b65: Pull complete
c9853740aa05: Pull complete
e9fa5d3a0d0e: Pull complete
Digest: sha256:xxxxxxxxxxxx...
Status: Downloaded newer image for centos:7
```



List Docker Images

- Now, let's list all our locally installed images, like:

```
docker images
```

- Output:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
centos	7	e9fa5d3a0d0e	2 days ago	172.3 MB

List Docker Images

- We can pass the images command (-a) option to see all images, past and present:

```
docker images -a
```

- Output:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
centos	7	e9fa5d3a0d0e	2 days ago	172.3 MB
<none>	<none>	c9853740aa05	2 days ago	172.3 MB
<none>	<none>	e9407f1d4b65	2 days ago	172.3 MB
<none>	<none>	0cd86ce0a197	2 days ago	172.3 MB
<none>	<none>	fa5be2806d4c	5 weeks ago	0 B

Running Our First Docker Container

- Here's how we can then run a Linux command inside a Docker container:

```
docker run -rm centos:7 echo "hello world"
```

- Output:

```
hello world
```

- Woah, what happened? It just printed "hello, world"? So what?

Running Our First Docker Container

- ❑ Let's run a Linux shell **inside** a Docker container, from the inside:

```
docker run --it --rm centos:7 bash
```

- ❑ Output:

```
[root@d7dfcc490cbe /]# _
```

- ❑ It's just that easy!

Inside Our Container

- Now from inside the Docker container itself, we'll issue commands, like this:

```
# ll /etc/*-release
```

- Output:

```
-rw-r--r-- 1 root root 38 Mar 31 2015 /etc/centos-release
-rw-r--r-- 1 root root 393 Mar 31 2015 /etc/os-release
lrwxrwxrwx 1 root root 14 Aug 14 21:00 /etc/redhat-release ->
centos-release
lrwxrwxrwx 1 root root 14 Aug 14 21:00 /etc/system-release ->
centos-release
```

Inside Our Container

- We can now play around with Linux commands within the container:

```
# hostname -f  
  
# cat /etc/hosts  
  
# ps aux  
  
# yum -y install vim  
  
# ip a
```

- If it looks like Linux and it works like Linux...

Let's Make a Mess

- ❑ Now, lets pretend we do some destructive stuff, on purpose:

```
# rm -fr /usr/sbin
```

- ❑ And now say we delete this, too:

```
# rm -fr /usr/bin
```

- ❑ Now, if we tried to issue any commands, like ls, ps, or cd we'd get:

```
bash: /usr/bin/ls: No such file or directory
```

- ❑ Whoops... cannot ls, cd or do anything useful anymore. What'd we'd do?

How Easy Is Starting Over

- ❑ In the last slide we pretended to destroy all our hard work.
- ❑ Now, how easy is it to start over?
- ❑ Just exit the container and fire up a new one:

```
docker run --it --rm centos:7 bash
```

- ❑ And everything is back! Wow... right?

Hands-on Exercise(s)

End of Chapter

Application Server in a Docker Container

MAKING CONTAINERS APPLICATION SERVERS

What is an Application Server?

- ❑ In this lesson, you'll be using a Tomcat, but what is a application server?

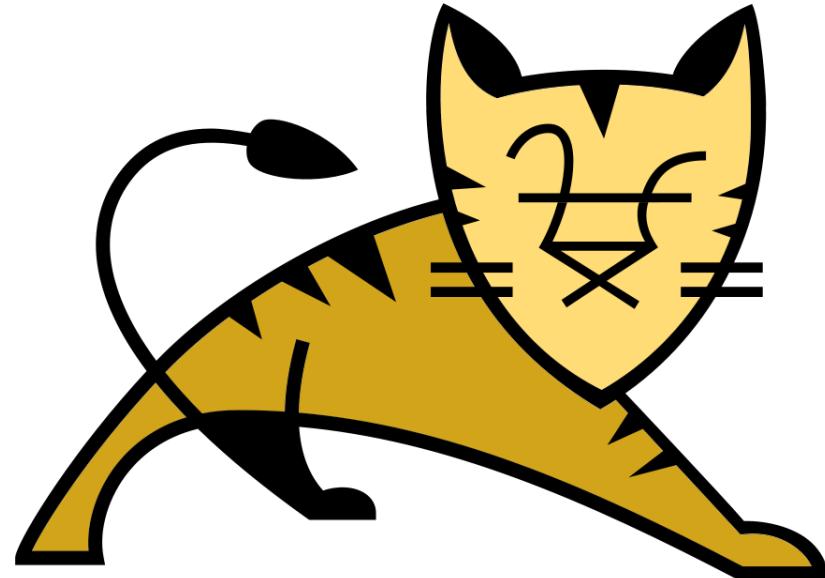
Wikipedia says:

“An application server is a software framework that provides both facilities to create web applications and a server environment to run them.”

- ❑ Docker provides Tomcat images on Docker Hub to do this with
- ❑ We'll look at port-mapping, container parameters, and more!

Understanding Tomcat

- ❑ Apache Tomcat (or simply Tomcat) is an open source web server and servlet container developed by the Apache Software Foundation (ASF).
- ❑ Tomcat implements the Java Servlet and the JavaServer Pages (JSP) specifications from Oracle.
- ❑ It provides a "pure Java" HTTP web server environment for Java code to run in.



Understanding Tomcat

- ❑ In the simplest configuration Tomcat runs in a single operating system process
- ❑ The process runs a Java virtual machine (JVM)
- ❑ Every single HTTP request from a browser to Tomcat is processed in the Tomcat process in a separate thread

Deploy Apache Tomcat in Docker

- ❑ Now let's run a JVM based application like Apache Tomcat:

```
docker run --rm -p 8888:8080 tomcat:8.0
```

- ❑ Output:

```
16-Oct-2016 18:30:51.541 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory /usr/local/tomcat/webapps/manager has finished in 28 ms

16-Oct-2016 18:30:51.542 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory /usr/local/tomcat/webapps/examples

16-Oct-2016 18:30:52.108 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory /usr/local/tomcat/webapps/examples has finished in 566 ms

16-Oct-2016 18:30:52.117 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory /usr/local/tomcat/webapps/ROOT

16-Oct-2016 18:30:52.161 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory /usr/local/tomcat/webapps/ROOT has finished in 45 ms

16-Oct-2016 18:30:52.176 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]

16-Oct-2016 18:30:52.206 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["ajp-nio-8009"]

16-Oct-2016 18:30:52.208 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in 1589 ms
```

Deploy Apache Tomcat

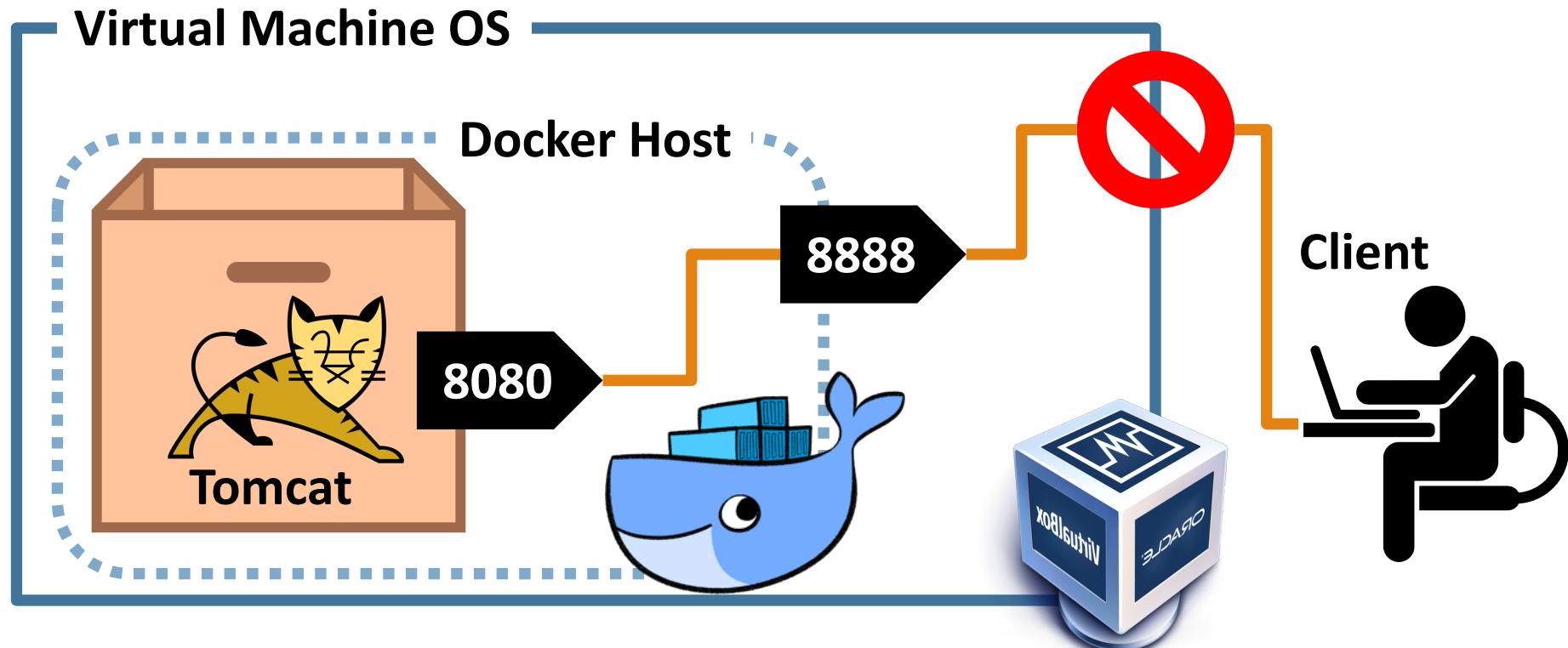
- ❑ Let's explore that command for a quick sec:

```
docker run --rm -p 8888:8080 tomcat:8.0
```

- ❑ The option (`--rm`) tells Docker we want to remove the container when it's done running.
- ❑ We are exposing ports `8888:8080` with `(-p)`, which tells Docker we want to map the container's port `8080` to the host port of `8888`
- ❑ If we were to connect to `http://localhost:8888` we should be able to reach our tomct server.

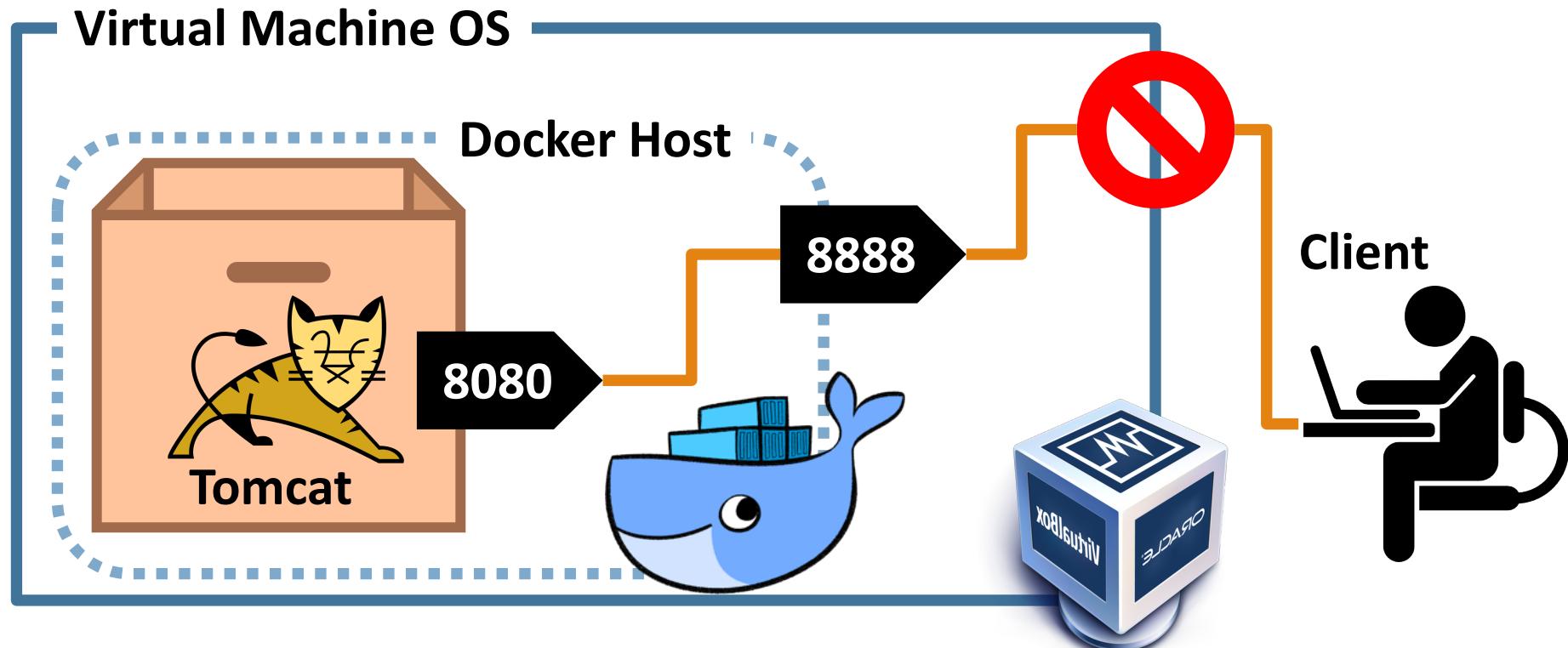
Deploy Apache Tomcat VM

- ❑ If we are operating Docker from inside a VM image, there is some additional work to be done. Here's a look at the problem:



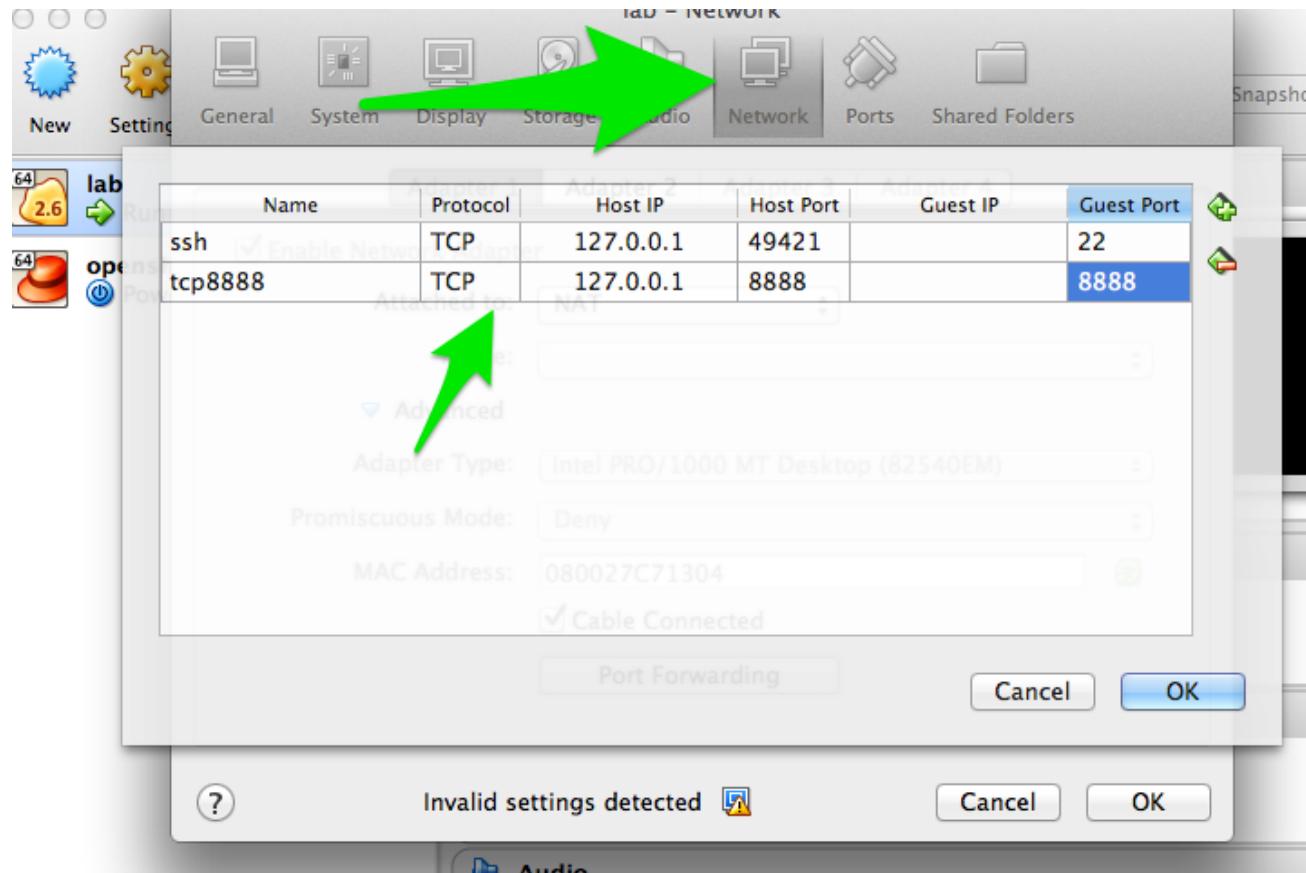
Deploy Apache Tomcat VM

- ❑ Our Docker Host has been mapped properly, but we cannot reach it from our host (Windows/MacOSX) because the VM does not expose those ports.



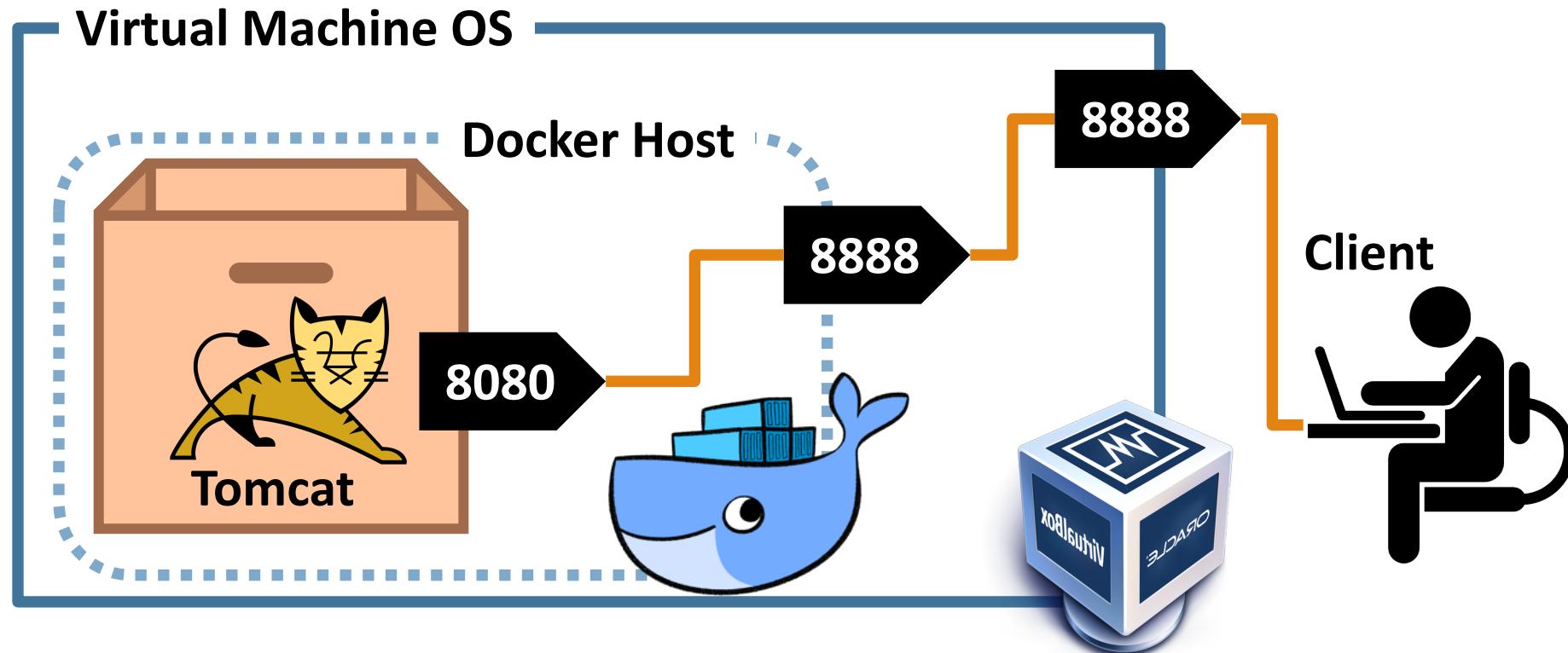
Map Ports for Tomcat VM

- ☐ Enable port forwarding between the VM Host (windows/Mac) and the VM Guest (Docker host):



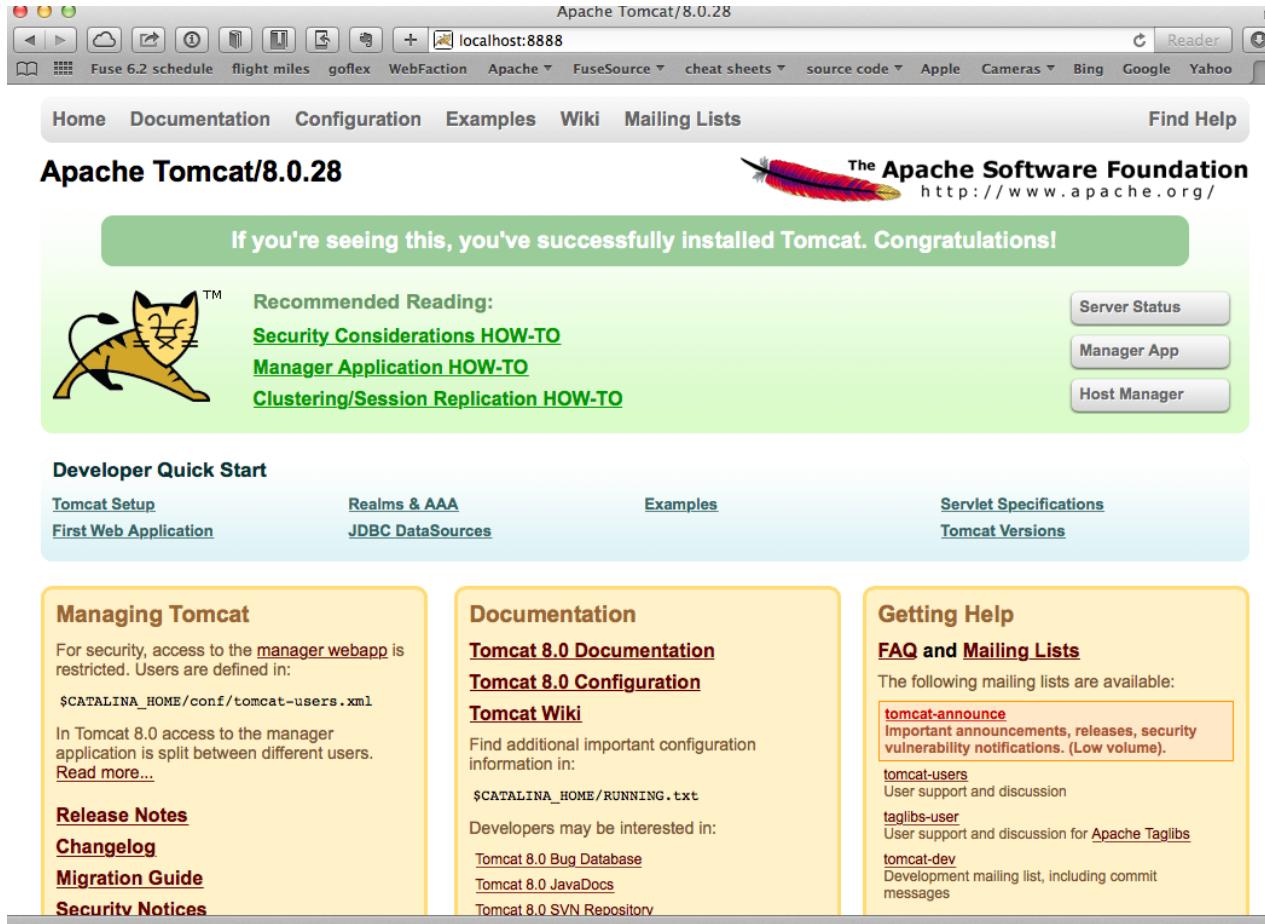
Map Ports for Tomcat

The outcome is represented in the diagram below:



Deploy Apache Tomcat

- Now, when we navigate to `http://localhost:8888` in our browser, we see:



Deploy Apache Tomcat

- We now have a running container that has tomcat in it.
- Let's explore the tomcat container really quick.
- Let's fire up a new shell window and observe Tomcat's processes, like this:

```
docker ps
```

- Output:

CONTAINER ID	IMAGE	COMMAND
xxxxxxxxxxxxxx	tomcat:8.0	...

Explore Inside Tomcat Container

- ❑ Let's log into the container to explore:

```
docker exec -it <container_id> bash
```

- ❑ We should now be at the bash prompt for the tomcat container.
- ❑ We can navigate around like we would normally.

Deploy Apache Tomcat

- ❑ We can exit out of the Tomcat container, like this:

```
# exit
```

- ❑ If we then switch back to the other window where Tomcat is running, we can exit by issuing the CTR+C command.
- ❑ We then have no containers running and we can verify that, with:

```
docker ps -a
```

- ❑ NOTE: Because of the (--rm) option, the container removed itself when finished.

Common Key Container Parameters

- Here are some other useful Docker `run` flags:

--name	Gives containers a unique name
-d	Runs containers in daemon mode (in the background)
--dns	Gives containers a different name server from the host
-it	Interactive with tty (caution using this with -d)
-e	Passes in environment variables to the container
--expose	Exposes ports from the Docker container
-P	Exposes all published ports on containers
-p	Map a specific port from the container to the host (host:container)

Deploy Apache Tomcat as a Daemon

- ❑ Let's use some of those previous run command-line flags and start tomcat in the background:

```
docker run -d --name="tomcat8" -p 8888:8080 tomcat:8.0
```

- ❑ NOTE: We also gave this container a name, so we can refer to it by name instead of container id

Print Apache Tomcat Logs

- With Tomcat running as a daemon (-d) in the background, we can print it's logs, like this:

```
docker logs tomcat8
```

- Output:

```
16-Oct-2016 19:19:20.441 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory
Deployment of web application directory /usr/local/tomcat/webapps/examples has finished in 526 ms

16-Oct-2016 19:19:20.447 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory
Deploying web application directory /usr/local/tomcat/webapps/ROOT

16-Oct-2016 19:19:20.507 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory
Deployment of web application directory /usr/local/tomcat/webapps/ROOT has finished in 60 ms

16-Oct-2016 19:19:20.515 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]

16-Oct-2016 19:19:20.527 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["ajp-nio-8009"]

16-Oct-2016 19:19:20.547 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in 1497 ms
```

Print Apache Tomcat Processes

- ❑ Let's use a couple of interesting Docker commands with our tomcat8 container:

```
docker top tomcat8
```

- ❑ [Output] Instead of the normal Linux top container, it just displays the processes running in the container:

PID	USER	COMMAND
5301	root	/usr/bin/java -Djava.util.logging.config.file=/usr/local/tomcat/conf/logging.properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Djava.endorsed.dirs=/usr/local/tomcat/endorsed -classpath /usr/local/tomcat/bin/bootstrap.jar:/usr/local/tomcat/bin/tomcat-juli.jar -Dcatalina.base=/usr/local/tomcat -Dcatalina.home=/usr/local/tomcat -Djava.io.tmpdir=/usr/local/tomcat/temp org.apache.catalina.startup.Bootstrap start

Inspect Apache Tomcat Daemon

- ❑ Let's say we want to "inspect" tomcat8, we can do that like this:

```
docker inspect tomcat8
```

- ❑ We can also use a --format template to pick out specific info from that output :

```
docker inspect --format='{{.Config.Env}}' tomcat8
```

- ❑ Output:

```
root@server(~) $ docker inspect --format='{{.Config.Env}}' tomcat8
[PATH=/usr/local/tomcat/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
LANG=C.UTF-8 JAVA_VERSION=7u79 JAVA_DEBIAN_VERSION=7u79-2.5.6-1~deb8u1
CATALINA_HOME=/usr/local/tomcat TOMCAT_MAJOR=8 TOMCAT_VERSION=8.0.28
TOMCAT_TGZ_URL=https://www.apache.org/dist/tomcat/tomcat-8/v8.0.28/bin/apache-tomcat-
8.0.28.tar.gz]
```

Lifecycle of a Container

- Many things can be done with containers, but how long does it last?
- Here's a diagram of a container's lifecycle:

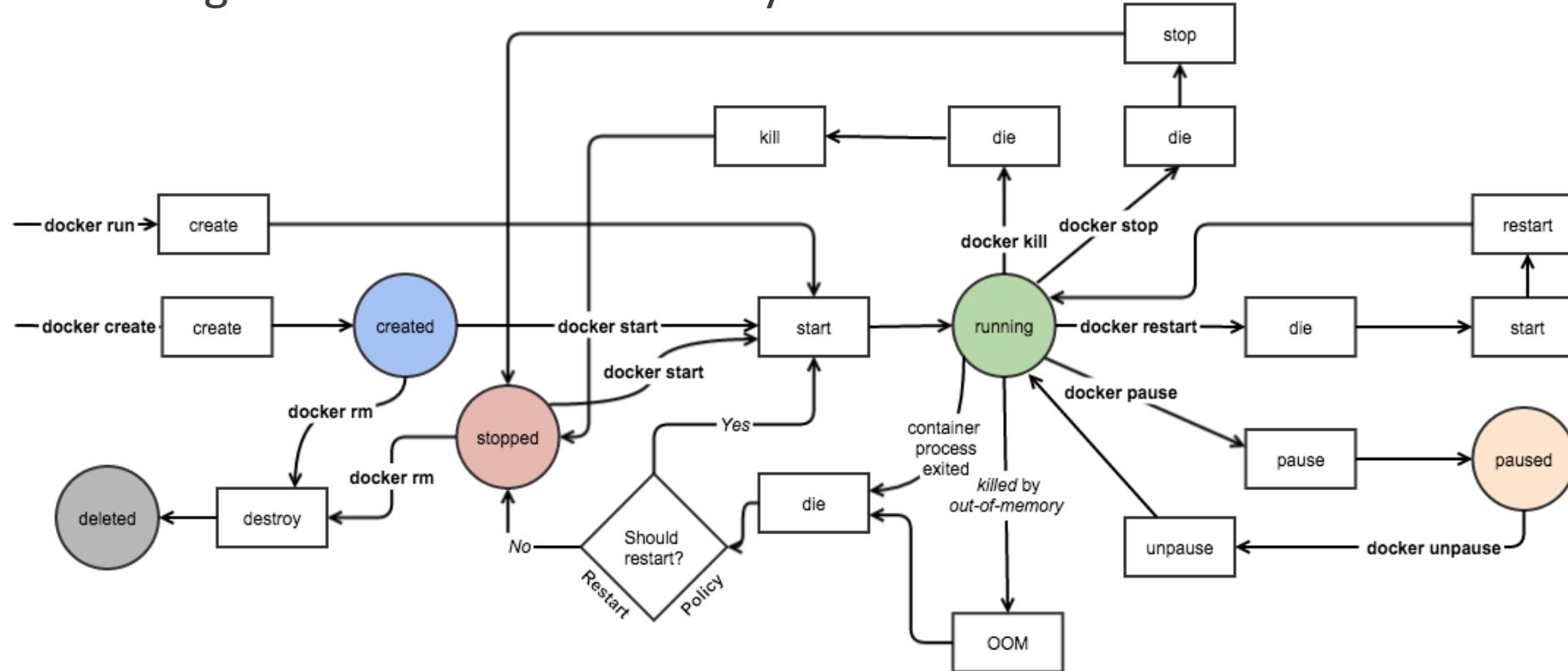


Image Source and Credits: http://www.slideshare.net/w_akram/docker-introduction-59904719

Lifecycle of a Container

Create Container

- ❑ Create a container to run it later on with required image:

```
$ docker create --name <container-name> <image-name>
```

Run Container

- ❑ Run the docker container with the required image and specified command/process; ‘-d’ flag is used for running the container in background:

```
$ docker run -it -d --name <container-name> <image-name> bash
```

Pause Container

- ❑ Used to pause the processes running inside the container.

```
$ docker pause <container-id/name>
```

Lifecycle of a Container

Stop Container

- ❑ To stop the container and processes running inside the container:

```
$ docker stop <container-id/name>
```

- ❑ To stop all the running Docker containers:

```
$ docker stop $(docker ps -a -q)
```

Start Container

- ❑ Start the container, if present in stopped state:

```
$ docker start <container-id/name>
```

Lifecycle of a Container

Restart Container

- ❑ It is used to restart the container as well as processes running inside the container:

```
$ docker restart <container-id/name>
```

Kill Container

- ❑ We can kill the running container:

```
$ docker kill <container-id/name>
```

Stop and Remove Container

- ❑ We know how to stop it, but how do we completely destroy it?

Destroy Container

- ❑ Its preferred to destroy container, only if present in stopped state instead of forcefully destroying the running container:

```
$ docker rm <container-id/name>
```

- ❑ NOTE: To destroy the parent image of a container, just add an “i”, like (`rmi`)
- ❑ To remove all the stopped Docker containers

```
$ docker rm $(docker ps -q -f status=exited)
```

Hands-on Exercise(s)

End of Chapter

Containerizing Java EE Application

DOCKER AND JAVA WORK GREAT TOGETHER

Understanding Structure of a Modern Java EE App

Understanding Java EE App Server

- ❑ Wikipedia says: “An **application server** is a software framework that provides both facilities to create web applications and a server environment to run them.”
- ❑ Java Platform, Enterprise Edition or Java EE (was J2EE) defines the core set of API and features of Java Application Servers
- ❑ Java application servers behave like an extended virtual machine for running applications
- ❑ It transparently handles connections to the database on one side, and, often, connections to the Web client on the other
- ❑ Open source Java application servers that support Java EE include: JOnAS from Object Web, **WildFly** (formerly JBoss AS) from JBoss, TomEE from Apache, and more.

Introduction to Dockerfile

- ❑ Docker build images by reading 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
- ❑ `docker image build` command uses this file and executes all the commands in succession to create an image
- ❑ `build` command is also passed a context that is used during image creation
- ❑ This context can be a path on your local filesystem or a URL to a Git repository

Introduction to Dockerfile

- The common commands are listed below:

Command	Purpose	Example
FROM	First non-comment instruction in <i>Dockerfile</i>	FROM ubuntu
COPY	Copies multiple source files from the context to the file system of the container at the specified path	COPY .bash_profile /home
ENV	Sets the environment variable	ENV HOSTNAME=test
RUN	Executes a command	RUN apt-get update
CMD	Defaults for an executing container	CMD ["/bin/echo", "hello world"]
EXPOSE	Informs the network ports that the container will listen on	EXPOSE 8093

Create Our First Dockerfile Image

- ❑ Create a new directory `hellodocker`
- ❑ In that directory, create a new text file `Dockerfile`, using the following:

```
FROM ubuntu:latest  
  
CMD ["/bin/echo", "hello world"]
```

- ❑ Build the image using the command:

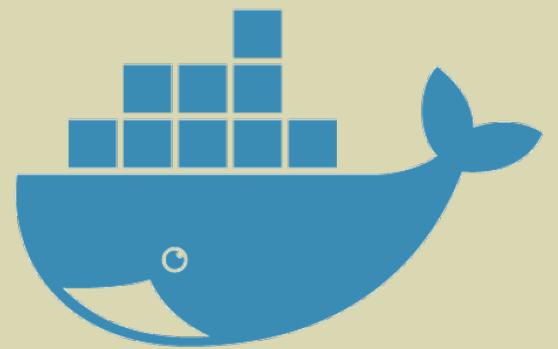
```
$ docker image build . -t helloworld
```

- ❑ `(.)` in this command is the context for the command `docker image build`
- ❑ `-t` adds a tag to the image
- ❑ The following output is shown...

Next,
Slide 

Create Our First Dockerfile Image

```
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM ubuntu:latest
latest: Pulling from library/ubuntu
9fb6c798fa41: Pull complete
3b61feb4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
Digest: sha256:31371c117d65387be2640b8254464102c36c4e23d2abe1f6f4667e47716483f1
Status: Downloaded newer image for ubuntu:latest
--> 2d696327ab2e
Step 2/2 : CMD /bin/echo hello world
--> Running in 9356a508590c
--> e61f88f3a0f7
Removing intermediate container 9356a508590c
Successfully built e61f88f3a0f7
Successfully tagged helloworld:latest
```



Create Our First Dockerfile Image

- ☐ List the images available using `docker image ls`:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
helloworld	latest	e61f88f3a0f7	3 minutes ago	122MB
ubuntu	latest	2d696327ab2e	4 days ago	122MB

- ☐ Now, run the container our Dockerfile created by using the `run` command:

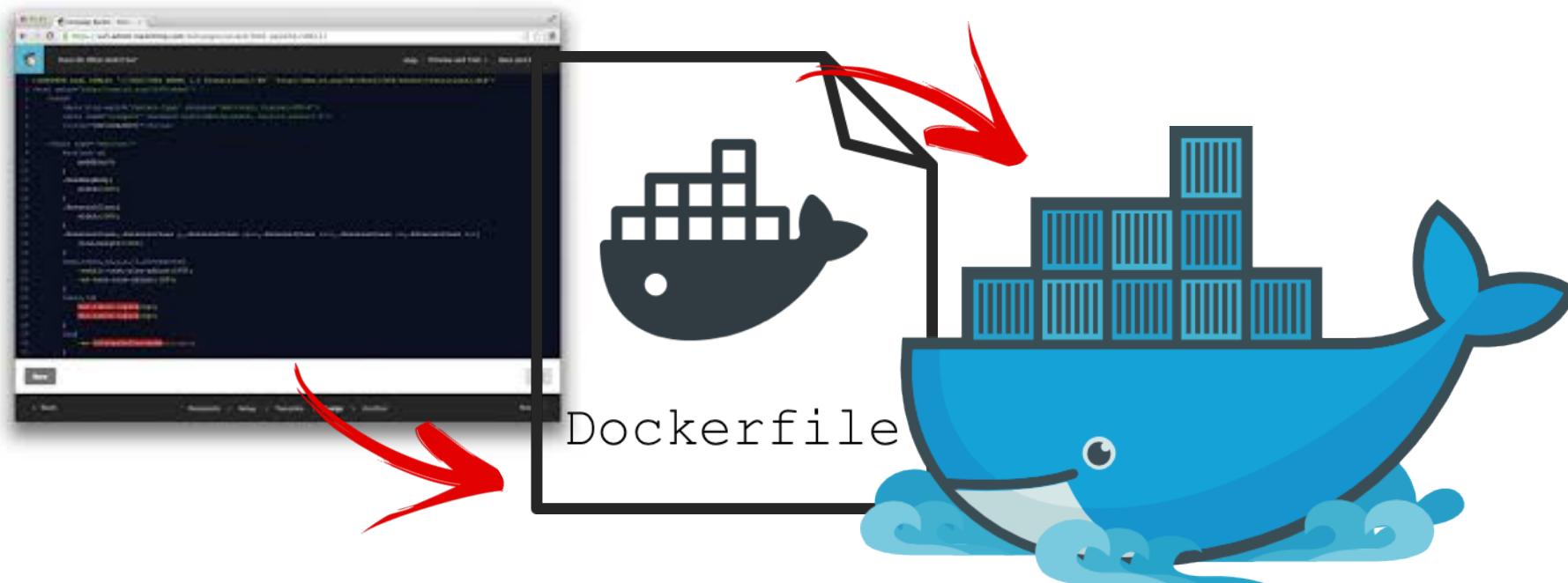
```
$ docker container run helloworld
```

- ☐ Output confirms that it works!

hello world

Understanding Dockerization

- ❑ Dockerizing an application is the process of converting an application to run within a Docker container
- ❑ While Dockerizing most applications is straight-forward, there are a few problems that need to be worked around each time



Stay Mindful of the Cons

- ❑ Two common problems that occur during Dockerization are:
 - ❑ Making an application use environment variables when it relies on configuration files
 - ❑ Sending application logs to STDOUT/STDERR when it defaults to files in the container's file system
- ❑ Remember both of these as we make our way through this course!

Create a Simple Java Application

□ Create a new Java project:

```
$ mvn archetype:generate -DgroupId=org.examples.java \  
-DartifactId=helloworld -DinteractiveMode=false
```

□ Build the project:

```
$ cd helloworld mvn package
```

□ Run the Java class:

```
$ java -cp target/helloworld-1.0-SNAPSHOT.jar org.examples.java.App
```

□ This shows the output:

Hello World!

□ Now, let's package this application as a Docker image!

Java Docker image

- Run the OpenJDK container in an interactive manner:

```
$ docker container run -it openjdk
```

- This will open a terminal in the container; check the version of Java:

```
$ root@8d0af9da5258:/# java -version
```

- Output:

```
openjdk version "1.8.0_141"  
OpenJDK Runtime Environment (build 1.8.0_141-8u141-b15-1~deb9u1-b15)  
OpenJDK 64-Bit Server VM (build 25.141-b15, mixed mode)
```

- NOTE: A different JDK version may be shown.
- Exit out of the container by typing exit in the container shell.

Package and Run Java App as Docker Image

- ❑ Create a new Dockerfile in helloworld directory and use the following content:

```
FROM openjdk:latest  
  
COPY target/helloworld-1.0-SNAPSHOT.jar /usr/src/helloworld-1.0-SNAPSHOT.jar  
  
CMD java -cp /usr/src/helloworld-1.0-SNAPSHOT.jar org.examples.java.App
```

- ❑ Build the image:

```
$ docker image build -t hello-java:latest .
```

- ❑ Output:

Hello World!

- ❑ This shows the exactly same output that was printed when the Java class was invoked using Java CLI.

Don't
Forget
Me!

Difference Between CMD and ENTRYPOINT

- ❑ **CMD** will work for most of the cases
- ❑ Default entry point for a container is /bin/sh, the default shells
- ❑ Running a container as docker container run -it ubuntu uses that command and starts the default shell
- ❑ The output is shown as:

```
$ docker container run -it ubuntu
root@88976ddee107:/#
```

Difference Between CMD and ENTRYPOINT

- ❑ **ENTRYPOINT** allows to override the entry point to some other command, and even customize it
- ❑ For example, a container can be started as:

```
$ docker container run -it --entrypoint=/bin/cat \
ubuntu /etc/passwd root:x:0:0:root:/root:/bin/bash \
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin \
bin:x:2:2:bin:/bin:/usr/sbin/nologin \
sys:x:3:3:sys:/dev:/usr/sbin/nologin . . .
```

root@88976ddee107:/#

- ❑ This command overrides the entry point to the container to `/bin/cat`
- ❑ The argument(s) passed to the CLI are used by the entry point.

Difference Between ADD and COPY

- ❑ **COPY** will work for most of the cases
- ❑ **ADD** has all capabilities of **COPY** and has the following additional features:
 1. Allows tar file auto-extraction in the image

- ❑ For example:

```
ADD app.tar.gz /opt/var/myapp
```

- 2. Allows files to be downloaded from a remote URL

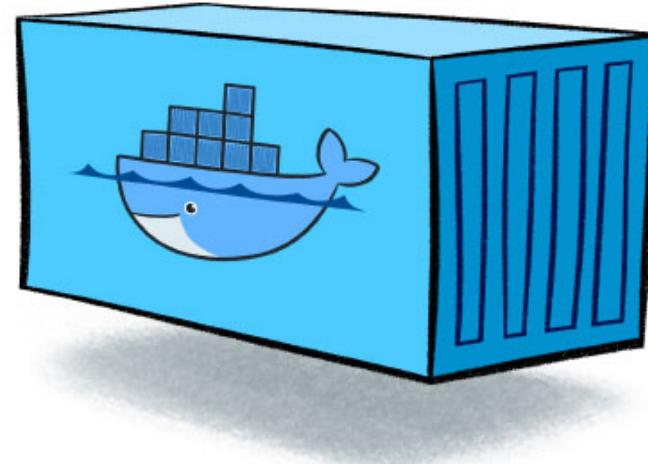
Import and Export Images

- ❑ Docker images can be saved using `image save` command to a `.tar` file:

```
docker image save helloworld > helloworld.tar
```

- ❑ These tar files can then be imported using `load` command:

```
docker image load -i helloworld.tar
```



Starting Our Application Server - Interactively

- Run WildFly container in an interactive mode, like this:

```
$ docker container run -it jboss/wildfly
```

- This will show the **output** as:

```
=====
JBoss Bootstrap Environment

JBOSS_HOME: /opt/jboss/wildfly

JAVA: /usr/lib/jvm/java/bin/java
. .
00:26:27,455 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0060: Http management
interface listening on http://127.0.0.1:9990/management
00:26:27,456 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0051: Admin console
listening on http://127.0.0.1:9990
00:26:27,457 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: WildFly Full
10.1.0.Final
```

Starting Our Application Server - Interactively

- ❑ By default, Docker runs in the foreground!
- ❑ `-i` allows to interact with the STDIN and `-t` attach a TTY to the process
- ❑ Switches can be combined together and used as `-it`, as we've seen already
- ❑ Hit `Ctrl+C` to stop the container



Starting Our Application Server - Detached Mode

- ☐ Restart the container in detached mode:

```
$ docker container run -d jboss/wildfly
```

```
254418caddb1e260e8489f872f51af4422bc4801d17746967d9777f565714600
```

- ☐ `-d`, instead of `-it`, runs the container in detached mode
- ☐ The output is the **unique id** assigned to the container
- ☐ This “container ID” can be used in lots of commands!

Unique
ID

Starting Our Application Server - Detached Mode

- ❑ Logs of the container can be seen using the following command:

```
$ docker container logs <CONTAINER_ID>
```

Unique ID

- ❑ Remember, status of a container can be checked using the docker container ls command, which gives us:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
254418caddb1	jboss/wildfly	"/opt/jboss/wildfl..."	2 minutes ago	Up 2 minutes	8080/tcp	gifted_habit

- ❑ Normally, there would be many listed and our **ID** is a way we could find it!
- ❑ Also remember docker container ls -a shows **all** the containers

Starting Our Application Server - Default Ports

- ❑ To accept incoming connections requires providing port options when invoking `docker run`
- ❑ The container, we just started, can't be accessed by our browser
- ❑ We need to stop it again and restart with different options

```
$ docker container stop `docker container ps | grep wildfly | awk '{print $1}'`
```

- ❑ Restart the container as:

```
$ docker container run -d -P --name wildfly jboss/wildfly
```

- ❑ `-P` map any exposed ports inside the image to a random port on Docker host

Starting Our Application Server - Default Ports

- ❑ In addition, `--name` option is used to give this container a name
- ❑ This name can then later be used to get more details about the container or stop it
- ❑ Of course, this can be verified using `docker container ls` command:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
89fbfbceeb56	jboss/wildfly	/opt/jboss/wildfl..."	9 seconds ago	Up 8 seconds	0.0.0.0:32768->8080/tcp	wildfly

- ❑ The port mapping is shown in the PORTS column

Starting Our Application Server - Default Ports

- ❑ Now, we can access our server's welcome page, via `localhost`!
- ❑ We will access WildFly server at `http://localhost:32768`



Deploy a WAR File

- ❑ Now that your application server is running, lets deploy a WAR file to it!
- ❑ Create a new directory hellojavaee
- ❑ Create a new text file and name it Dockerfile
- ❑ Use the following contents:

```
FROM jboss/wildfly:latest
RUN curl -L https://github.com/javaee-samples/javaee7-simple-
sample/releases/download/v1.10/javaee7-simple-sample-1.10.war -o
/opt/jboss/wildfly/standalone/deployments/javaee-simple-sample.war
```

- ❑ Now, what do you think comes next?

Deploy a WAR File

- Create an image:

```
$ docker image build -t javaee-sample .
```

Don't
Forget
Me!

- Start the container:

```
$ docker container run -d -p 8080:8080 --name wildfly javaee-sample
```

- Access the endpoint:

```
$ curl http://localhost:8080/javaee-simple-sample/resources/persons
```

- See the output:

Reference: **cs_05-output.bash**

Stopping and Removing Application

□ Stop a specific container by id or name:

```
$ docker container stop <CONTAINER ID>
```

```
$ docker container stop <NAME>
```

□ Stop all running containers:

```
$ docker container stop $(docker container ps -q)
```

□ Stop only the exited containers:

```
$ docker container ps -a -f "exited=-1"
```

Stopping and Removing Application

□ Remove a specific container by id or name:

```
$ docker container rm <CONTAINER ID>
```

```
$ docker container rm <NAME>
```

□ Remove containers meeting a regular expression:

```
$ docker container ps -a | grep wildfly |  
awk '{print $1}' | xargs docker container rm
```

□ Remove all containers, without any criteria:

```
$ docker container rm $(docker container ps -aq)
```

Hands-on Exercise(s)

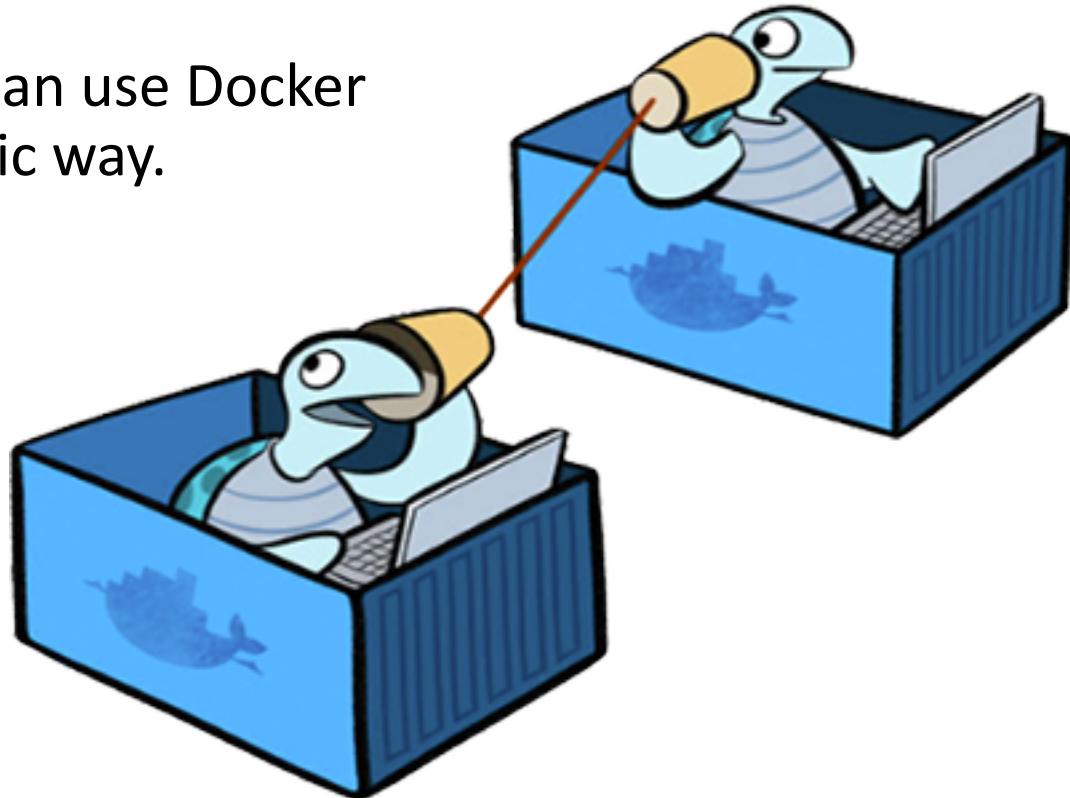
End of Chapter

Docker Linking and Stateful Containers

HOW WE LINK AND CREATE STATEFUL CONTAINERS

Understanding Networking Overview

- ❑ Docker containers and services can connect together or non-Docker workloads
- ❑ Containers and services do not need to be aware they are deployed on Docker, or whether their peers are also Docker workloads or not
- ❑ No matter the OS, or a mix of, you can use Docker to manage them in a platform-agnostic way.



Docker Network Drivers

- ❑ Docker's networking subsystem is pluggable, using drivers
- ❑ Several drivers exist by default, and provide core networking functionality:

bridge:

The default network driver. If you don't specify a driver, this is the type of network you are creating. **Bridge networks are usually used when your applications run in standalone containers that need to communicate.**

host:

For standalone containers, remove network isolation between the container and the Docker host, and use the host's networking directly. **host** is only available for swarm services on Docker 17.06 and higher.

Docker Network Drivers

overlay:

Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers.

macvlan:

Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.

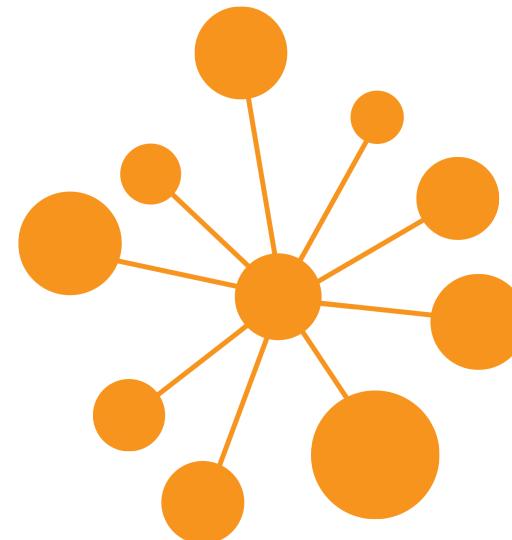
Docker Network Drivers

none:

For this container, disable all networking. Usually used in conjunction with a custom network driver. `none` is not available for swarm services.

Network Plugins:

You can install and use third-party network plugins with Docker. These plugins are available from *Docker Store* or from third-party vendors.



Network Driver Summary

- ❑ **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
- ❑ **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- ❑ **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- ❑ **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- ❑ **Third-party network plugins** allow you to integrate Docker with specialized network stacks.

Docker EE Networking Features

- ❑ The following two features are **only** possible when using *Docker EE* and managing your Docker services using *Universal Control Plane* (UCP):
 1. The *HTTP routing mesh* allows you to share the same network IP address and port among multiple services. UCP routes the traffic to the appropriate service using the combination of hostname and port, as requested from the client.
 2. *Session stickiness* allows you to specify information in the HTTP header which UCP uses to route subsequent requests to the same service task, for applications which require stateful sessions.

Container Networking Specific

- ❑ Type of network a container uses, whether it is a [bridge](#), an [overlay](#), a [macvlan network](#), or a custom network plugin, is transparent from within the container.
- ❑ From the container's point of view, it has:
 - A network interface with an IP address
 - A gateway
 - A routing table
 - DNS services
 - and other networking details (assuming the container is not using the none network driver)
- ❑ This topic is about networking concerns from the point of view of the container

Published Ports

- ❑ By default, when you create a container, it does not publish any of its ports to the outside world
- ❑ To make a port available to services outside of Docker, or to Docker containers which are **not** connected to the container's network, use --publish (-p) flag
- ❑ This creates a firewall rule which maps a container port to a port on the Docker host, like we saw in Docker Registry
- ❑ Let's breakdown some examples of this in the next slide...

Next,
Slide!



Published Ports

- Here are some examples:

Flag Value	Description
<code>-p 8080:80</code>	Map TCP port 80 in the container to port 8080 on the Docker host.
<code>-p 8080:80/udp</code>	Map UDP port 80 in the container to port 8080 on the Docker host.
<code>-p 8080:80/tcp -p 8080:80/udp</code>	Map TCP port 80 in the container to TCP port 8080 on the Docker host, and map UDP port 80 in the container to UDP port 8080 on the Docker host.

Additional Ways to Find Port Mapping Details

- ❑ The exact mapped port can also be found using `docker port` command:

```
$ docker container port <CONTAINER_ID> or <NAME>
```

- ❑ This shows the output as:

```
8080/tcp -> 0.0.0.0:8080
```

- ❑ Port mapping can also be found using `docker inspect` command:

```
docker container inspect --format='{{(index (index .NetworkSettings.Ports "8080/tcp") 0).HostPort}}' <CONTAINER_ID>
```

IP Address and Hostname

- ❑ By default, the container is assigned an IP address for **every** Docker network it connects to
- ❑ The IP address is assigned from the **pool** assigned to the network
- ❑ Each network also has a default subnet mask and gateway
- ❑ When the container starts, it can only be connected to a single network, using `--network`
- ❑ **However**, you can connect a running container to multiple networks using `docker network connect`



IP Address and Hostname

- ❑ When starting containers using `--network`, you can specify the IP address using the `--ip` or `--ip6` flags
- ❑ When you connect an existing container to a different network using `docker network connect`, you can use `--ip` or `--ip6` flags to specify the container's IP address on the **additional** network
- ❑ In the same way, a container's hostname defaults to be the container's **name** in Docker
- ❑ You can override the hostname using `--hostname`
- ❑ When connecting to an existing network using `docker network connect`, you can use the `--alias` flag to specify an additional network alias for the

DNS Services

- ❑ By default, a container inherits the DNS settings of the Docker daemon, including the /etc/hosts and /etc/resolv.conf
- ❑ You can override these settings on a per-container basis
- ❑ Let's look at a few example options...

Next,
Slide!



DNS Services

Flag Value	Description
--dns	The IP address of a DNS server. To specify multiple DNS servers, use multiple --dns flags. If the container cannot reach any of the IP addresses you specify, Google's public DNS server 8.8.8.8 is added, so that your container can resolve internet domains.
--dns-search	A DNS search domain to search non-fully-qualified hostnames. To specify multiple DNS search prefixes, use multiple --dns-search flags.
--dns-opt	A key-value pair representing a DNS option and its value. See your operating system's documentation for resolv.conf for valid options.
--hostname	The hostname a container uses for itself. Defaults to the container's name if not specified.

User-defined Bridge

- ❑ When you create a new container, you can specify one or more network flags
- ❑ This example connects a Nginx container to the `my-net` network
- ❑ Also, it publishes port 80 in the container to port 8080 on the Docker host, so external clients can access that port
- ❑ Any other container connected to the `my-net` network has access to **all** ports on the `my-nginx` container, and vice versa:

```
$ docker create --name my-nginx \
--network my-net \
--publish 8080:80 \
nginx:latest
```

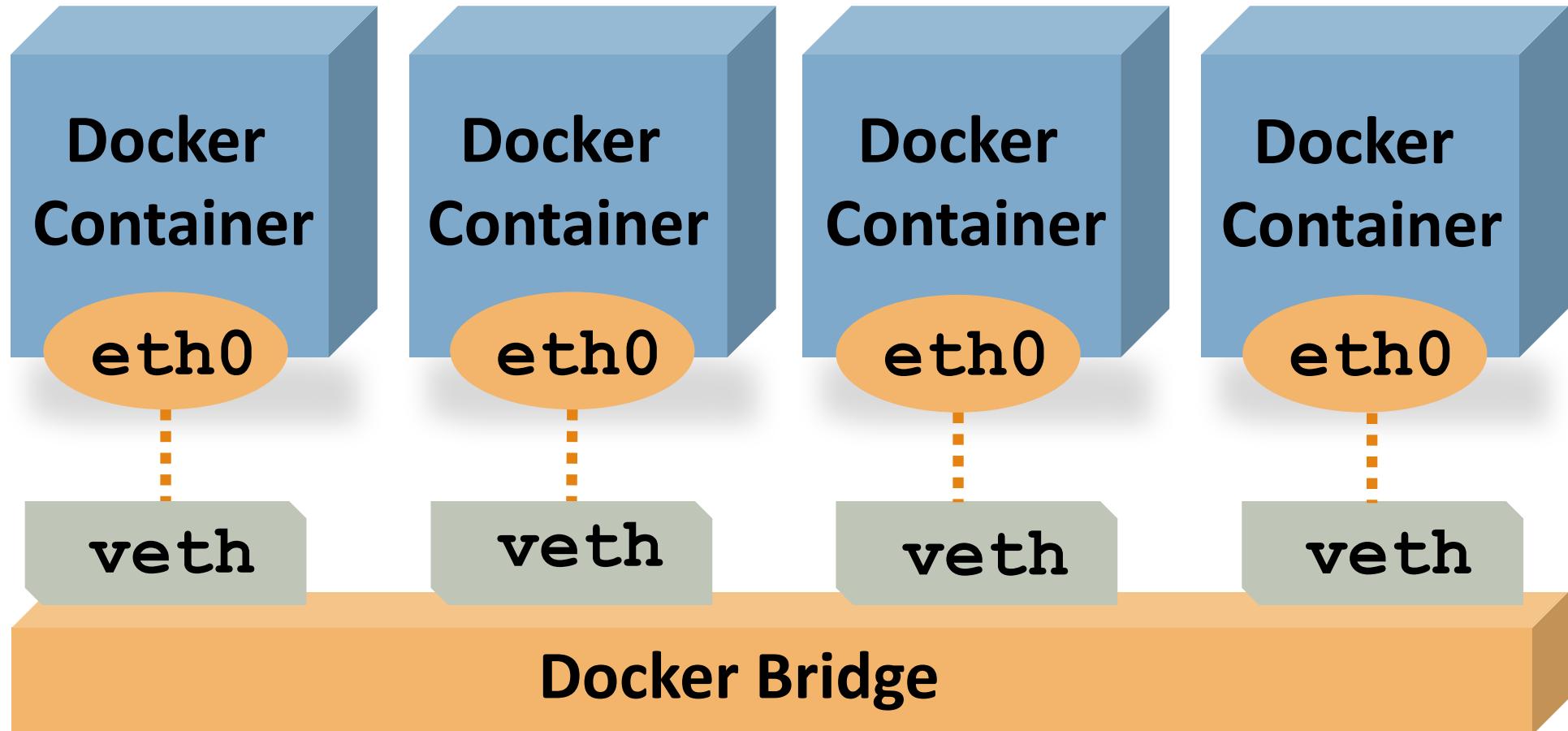
User-defined Bridge

- ❑ To connect a **running** container to an existing user-defined bridge, use the docker network connect command
- ❑ The following command connects an already-running my-nginx container to an already-existing my-net network:

```
$ docker network connect my-net my-nginx
```



Quick Look at Docker Networking



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?



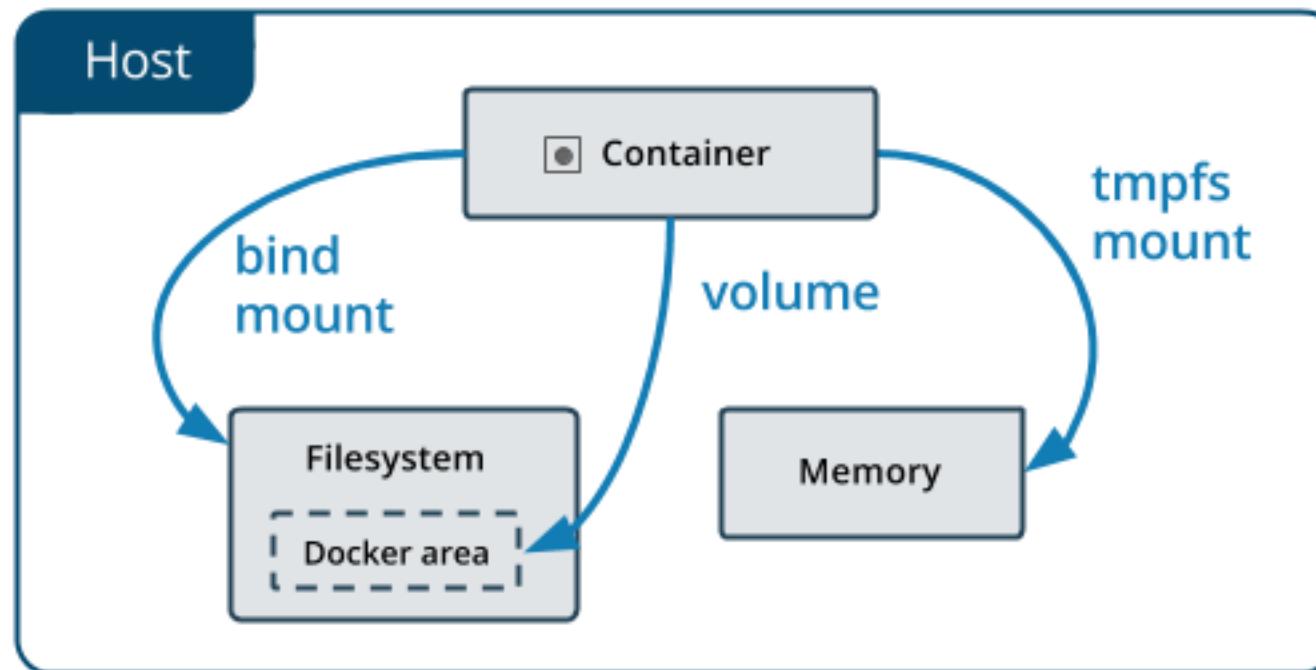
Managing Data in Docker

- ❑ It is possible to store data within the writable layer of a container, but there are some downsides:
 - The data doesn't persist when that container is no longer running
 - A container's writable layer is tightly coupled to the host machine where the container is running
 - Writing into a container's writable layer requires a storage driver to manage the filesystem.

- ❑ Docker offers three different ways to mount data into a container from the Docker host:
 1. volumes
 2. bind mounts
 3. tmpfs volumes

Choosing the Right Mount Type

- ❑ No matter which type of mount you choose the data looks the same
- ❑ To visualize the difference, think about where the data lives on the Docker host

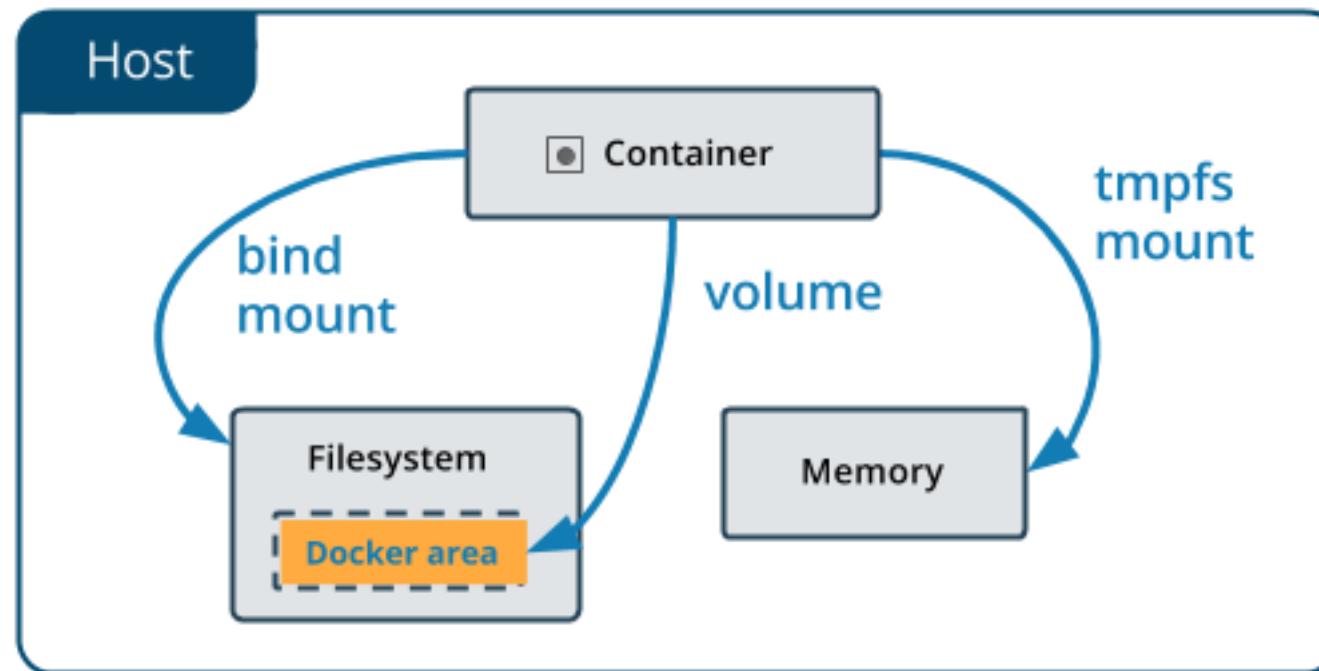


Concept of Volumes

- ❑ Volumes are the preferred mechanism for persisting data, generally
- ❑ Bind mounts are dependent on the directory structure of the host
- ❑ Volumes have several advantages over bind mounts:
 - Volumes are easier to back up or migrate
 - You can manage volumes using Docker CLI commands or the Docker API
 - Volumes work on both Linux and Windows containers
 - Volumes can be more safely shared among multiple containers
 - Volume drivers allow you to store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality
 - A new volume's contents can be pre-populated by a container

Concept of Volumes

- ❑ Volumes are also a better choice than persisting data in a container's writable layer, because using a volume does not increase the size of containers
- ❑ And, the volume's contents exist outside the **lifecycle** of a given container!



Concept of Volumes

- ❑ If your container generates non-persistent state data, consider a tmpfs mount
 - ❑ Avoid storing the data anywhere permanently
 - ❑ Increase the container's performance by avoiding writing into the container's writable layer
- ❑ Volumes use rprivate bind propagation, and bind propagation is **not** configurable for volumes

Confronting the Stateful Challenge

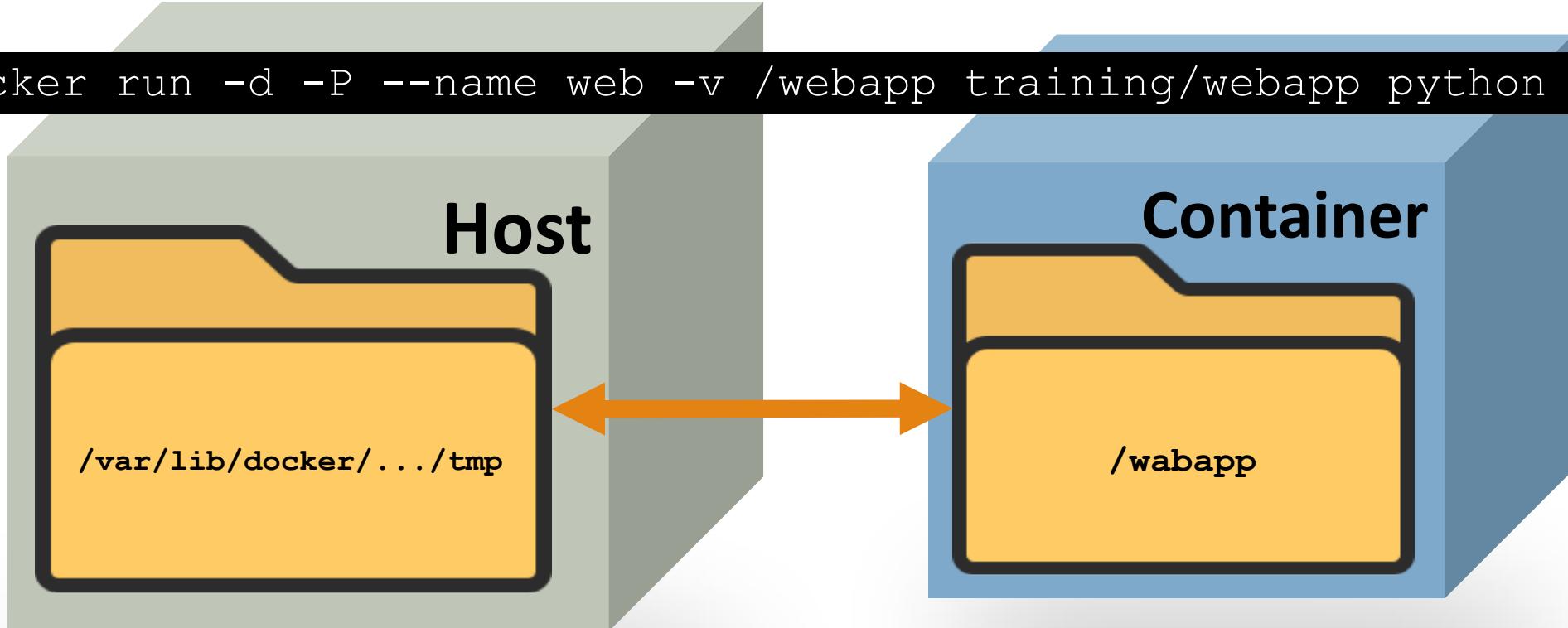
- ❑ Docker volumes to the **rescue** of all our stateful containers
 - ❑ Volumes create persistent data outside of the container
 - ❑ Can be mapped directly to host locations
 - ❑ Can also be deployed independently of hosts
-
- ❑ Example:

```
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
```



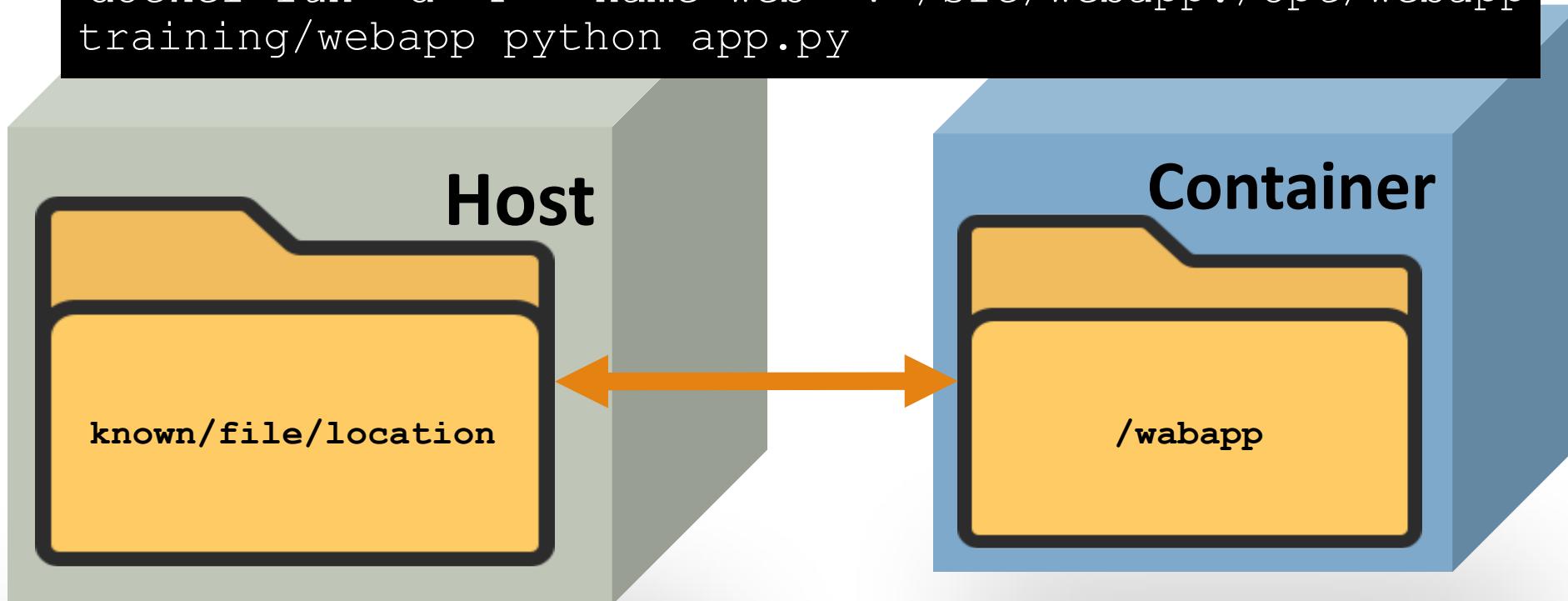
Direct Host Volumes

- ❑ 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:

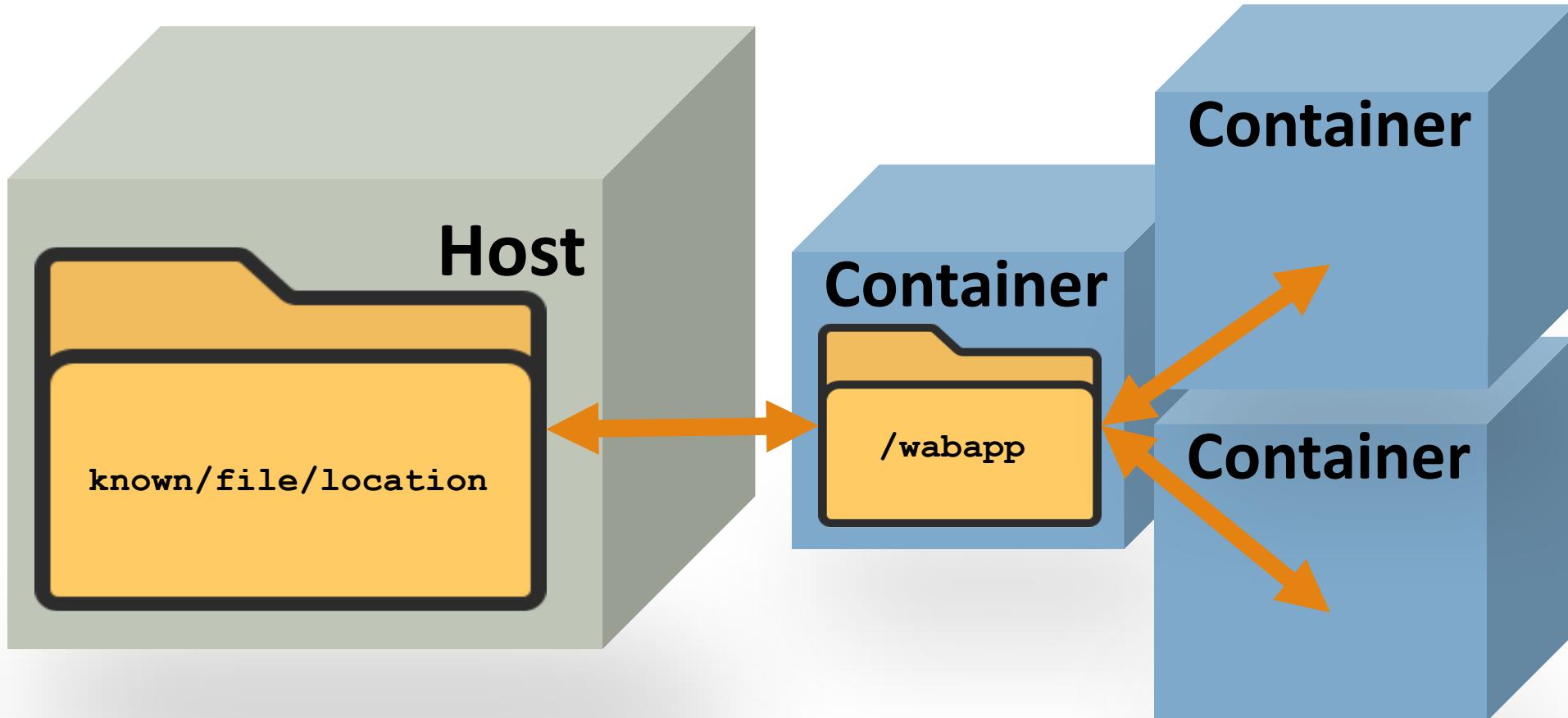
```
docker create -v /dbdata --name dbdata training/postgres  
/bin/true
```

- ❑ 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 dbl 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>
```

Choose the `-v` or `--mount` Flag

- ❑ Originally, the `-v` or `--volume` flag was used for standalone containers and the `--mount` flag was used for swarm services
- ❑ **However**, starting with Docker 17.06, you can also use `--mount` with standalone containers
- ❑ In general, `--mount` is more explicit and verbose
- ❑ The biggest difference is that the `-v` syntax combines all the options together in one field, while the `--mount` syntax separates them

Differences Between `-v` and `--mount` Behavior

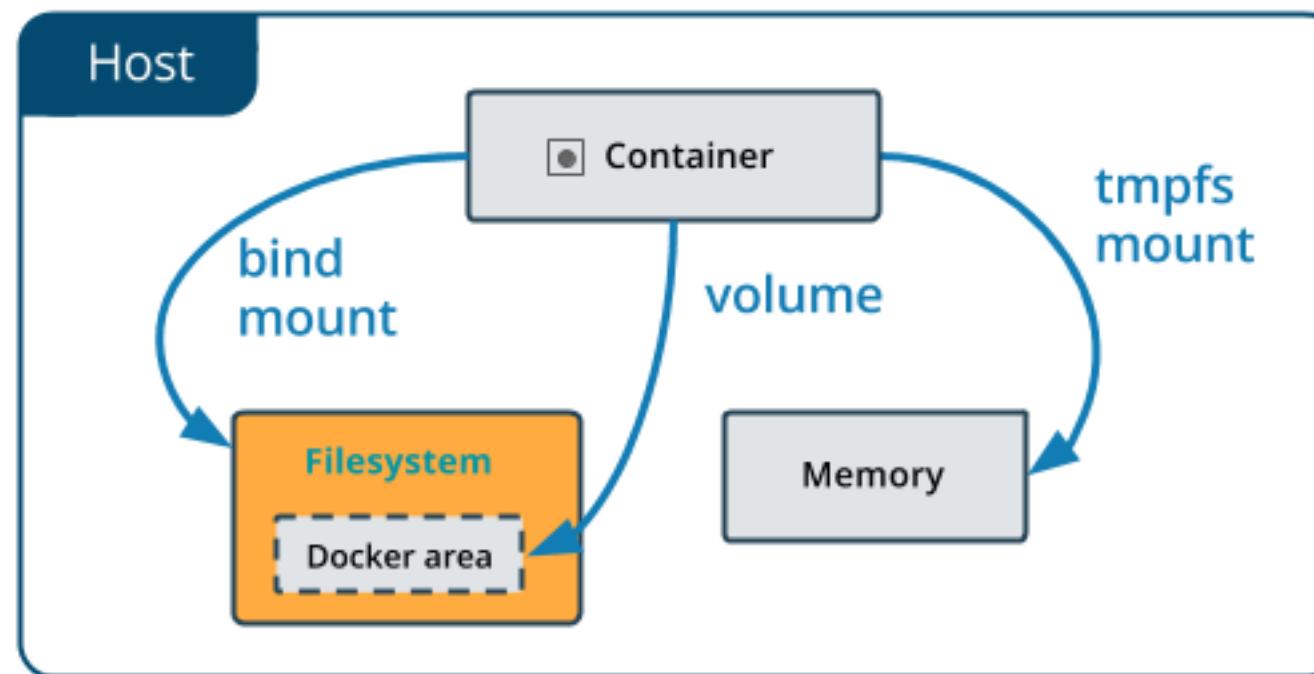
- ❑ Because volume flags have been a part of Docker for a long time, their behavior can't be changed
- ❑ This means that there's only **one** behavior that's different between the two
 - ❑ If you use `-v` or `--volume` to bind-mount a file or directory that does not yet exist on the Docker host, `-v` creates the endpoint for you
 - ❑ If you use `--mount` to bind-mount a file or directory that does not exist on the host, Docker does **not** automatically create it for you

Bind Mounts

- ❑ Bind mounts have been around since the early days of Docker
- ❑ Bind mounts have limited functionality compared to volumes
- ❑ When you use a bind mount, a file or directory on the *host machine* is mounted into a container
- ❑ The file or directory is referenced by its full or relative path on the host machine
- ❑ By contrast, with volumes, a new directory is created within Docker's storage directory on the host machine

Bind Mounts

- ❑ The file or directory does not need to exist on the Docker host already!
- ❑ Bind mounts are very performant, but they rely on the host machine's filesystem having a specific directory structure available
- ❑ If developing new Docker applications, consider using named volumes instead



Start a Container Using a Bind Mount

- ❑ Consider a case where...
 - ❑ You have a directory source and that when you build the source code, the artifacts are saved into another directory source/target/
 - ❑ You want the artifacts to be available to the container at /app/, and you want the container to get access to a new build each time you build the source on your development host
- ❑ Use the following command to bind-mount the target/ directory into your container at /app/

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,target=/app \
nginx:latest
```

Start a Container Using a Bind Mount

- ❑ Use `docker inspect devtest` to verify that the bind mount was created correctly
- ❑ Look for the Mounts section:

```
"Mounts": [  
    {  
        "Type": "bind",  
        "Source": "/tmp/source/target",  
        "Destination": "/app",  
        "Mode": "",  
        "RW": true,  
        "Propagation": "rprivate"  
    },  
]
```

About Storage Drivers

- ❑ To use storage drivers effectively, it's important to know how Docker builds and stores images
- ❑ Use this information to make informed choices about the best way to persist data from your applications and avoid performance problems
- ❑ Storage drivers allow you to persist data in the writable **layer** of your container
- ❑ This is the **least** efficient way to persist data!

Images and Layers

- ❑ A Docker image is built up from a series of layers
- ❑ Each layer represents an instruction in the image's Dockerfile.
- ❑ Each layer except the very last one is read-only
- ❑ Consider the following Dockerfile:

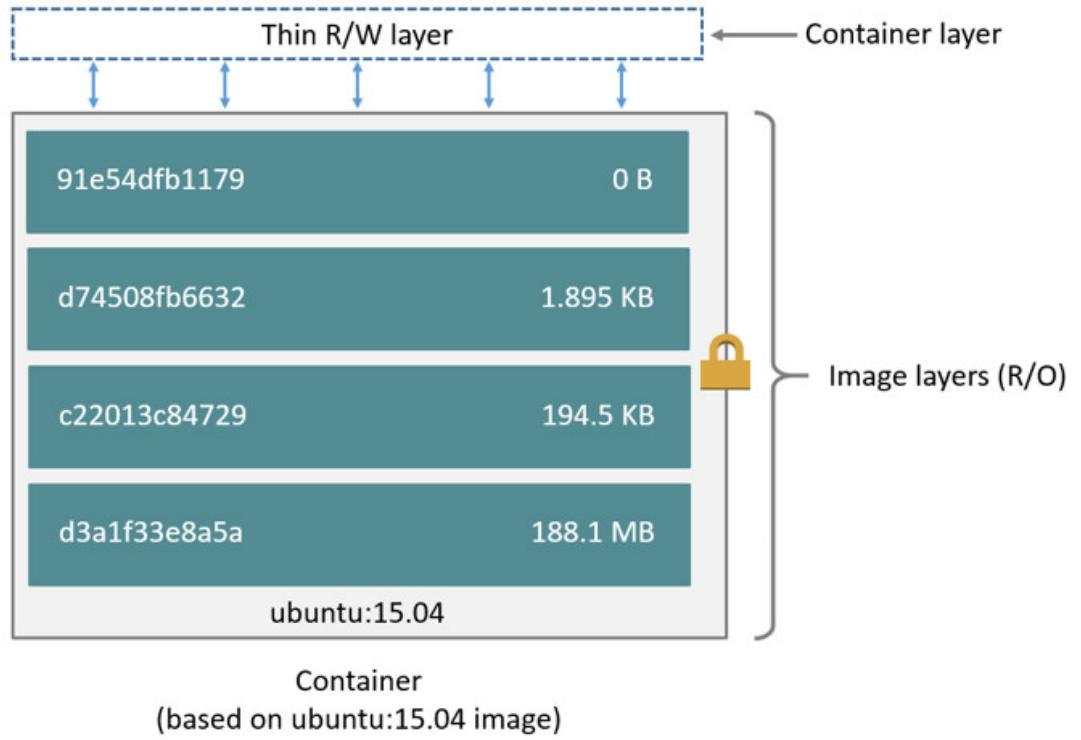
```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

- ❑ FROM creates a layer; ubuntu:15.04 image
- ❑ COPY command adds some files from (.)
- ❑ The RUN command builds your application
- ❑ CMD specifies what to run in the container

- ❑ Finally, the last layer specifies what command to run within the container

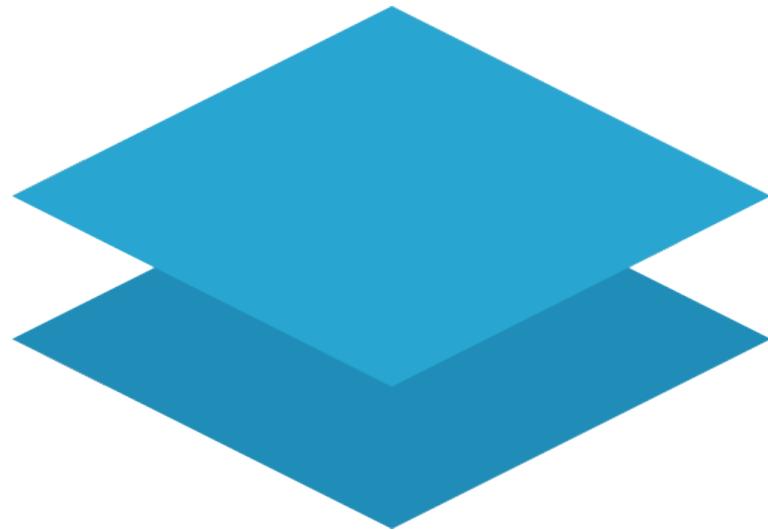
Images and Layers

- ❑ Each layer is only a set of differences from the layer before it
- ❑ When you create a new container, you add a new writable layer on top of the underlying layers
- ❑ All changes made are written to this thin writable container layer
- ❑ The diagram below shows a container based on the Ubuntu 15.04 image:
- ❑ A *storage driver* handles the details about the way these layers interact with each other



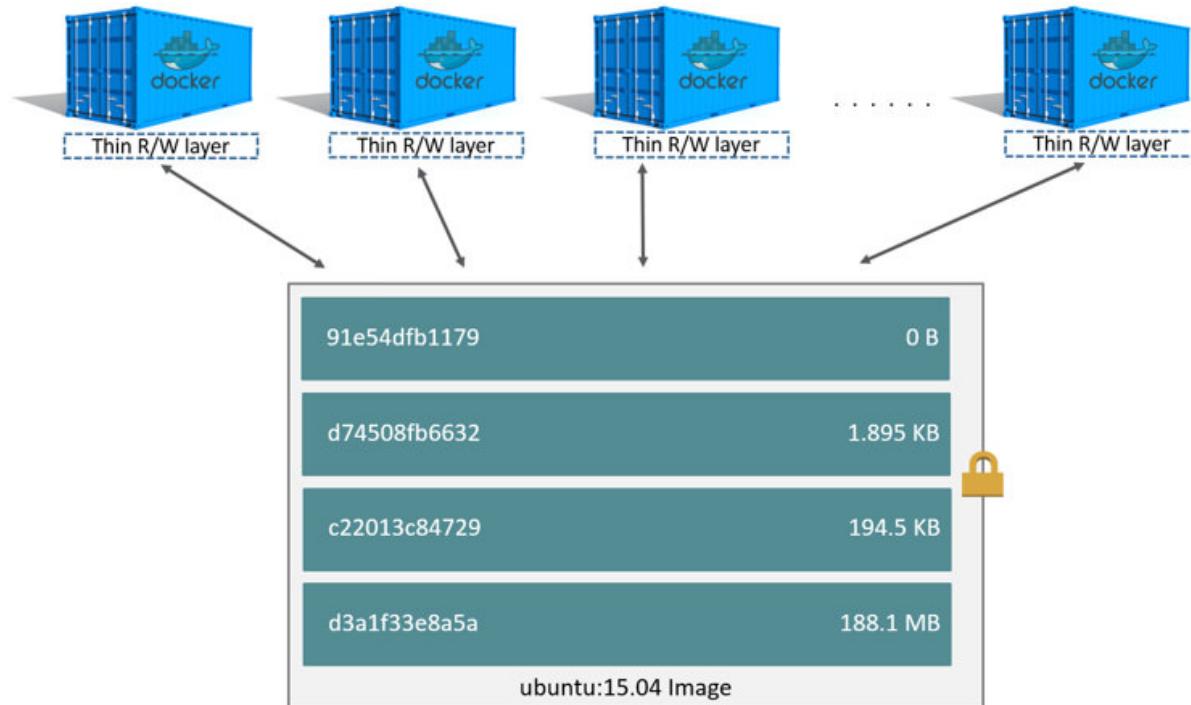
Containers and Layers

- ❑ The major difference between a container and an image is the top writable layer
- ❑ When the container is deleted, the writable layer is also deleted
- ❑ The underlying image remains unchanged
- ❑ Multiple containers can share access to the same underlying image and yet have their own data state



Containers and Layers

- The diagram below shows multiple containers sharing a Ubuntu 15.04 image:

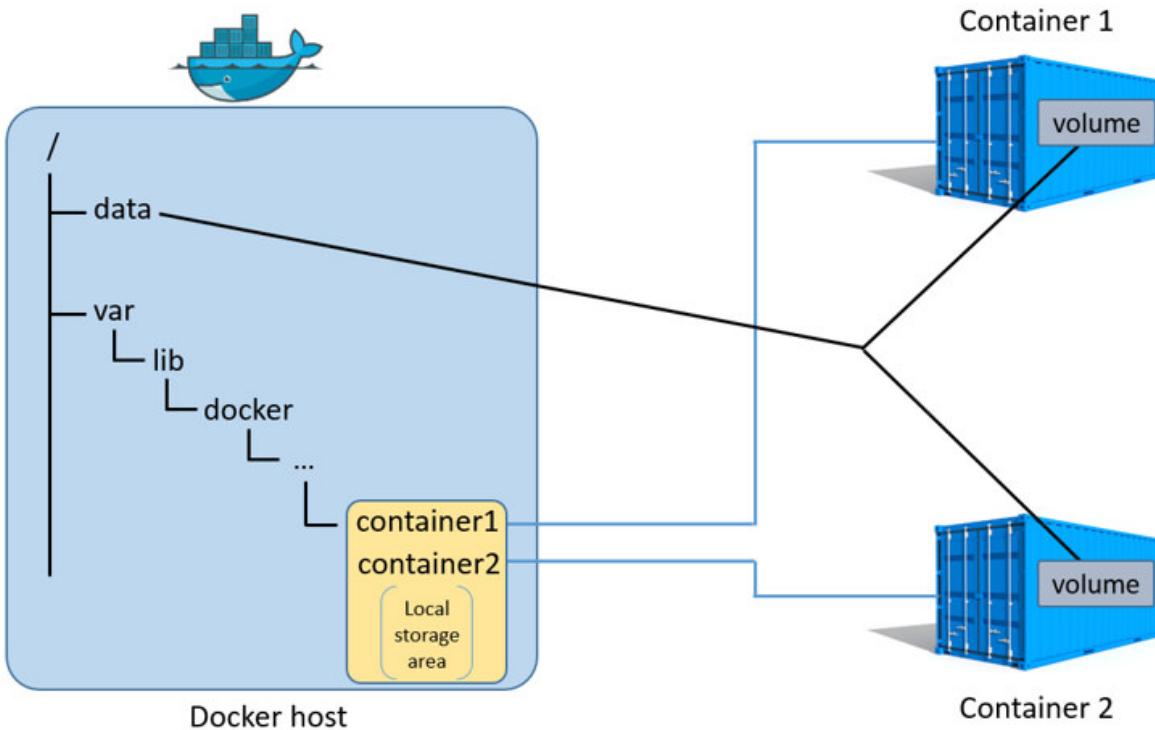


Data Volumes and Storage Drivers

- ❑ Any data written to the container, not stored in a *data volume* is **deleted!**
- ❑ A data volume is a directory or file in the Docker host's filesystem that is mounted directly into a container
- ❑ Data volumes are not controlled by the storage driver
- ❑ Reads and writes to data volumes **bypass** the storage driver and operate at native host speeds
- ❑ You can mount any number of data volumes into a container
- ❑ Multiple containers can also share one or more data volumes.

Data Volumes and Storage Drivers

- The diagram below shows a single Docker host running two containers:



- Data volumes reside outside of the local storage area on the Docker host
- Any data stored in data volumes persists on the Docker host

Hands-on Exercise(s)

End of Chapter

Docker Registry

CREATE A LOCAL AND EXTERNAL REGISTRY

Docker Registry Introduction

Docker says:

“Docker Trusted Registry (DTR) is a commercial product that enables complete image management workflow, featuring LDAP integration, image signing, security scanning, and integration with Universal Control Plane. DTR is offered as an add-on to Docker Enterprise subscriptions of Standard or higher.”



What & Why

What it is:

The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. The Registry is open-source, under the permissive Apache license.

Why use it:

You should use the Registry if you want to:

- Tightly control where your images are being stored
- Fully own your images distribution pipeline
- Integrate image storage and distribution tightly into your in-house development workflow

Alternatives & Requirements

- ❑ Users looking for a zero maintenance, ready-to-go solution are encouraged to head-over to the **Docker Hub**, which provides a free-to-use, hosted Registry, plus additional features (organization accounts, automated builds, and more).
 - ❑ Users looking for a commercially supported version of the Registry should look into **Docker Trusted Registry**.
 - ❑ The Registry is compatible with Docker engine **version 1.6.0 or higher**.
-
- ❑ Now, we've looked at DockerHub, so let's look now at Docker Trusted Registry!

Basic Commands

❑ Start Registry

```
$ docker run -d -p 5000:5000 --name registry registry:2
```

❑ Pull Image From DockerHub

```
$ docker pull ubuntu
```

❑ Tag Image to Point to the Registry

```
$ docker image tag ubuntu localhost:5000/myfirstimage
```

Basic Commands

□ Push it

```
$ docker push localhost:5000/myfirstimage
```

□ Pull it Back

```
$ docker pull localhost:5000/myfirstimage
```

□ Now Stop the Registry and Remove All Data

```
$ docker image tag ubuntu localhost:5000/myfirstimage
```

Docker Trusted Registry

Docker says:

“Docker Trusted Registry* (DTR) is the enterprise-grade image storage solution from Docker. You install it behind your firewall so that you can securely store and manage the Docker images you use in your applications.”



*The following slides are based on **DTR 2.4.1**

Image Management

- ❑ DTR can be installed on-premises, or on a virtual private cloud.
- ❑ You can store your Docker images securely, behind your firewall
- ❑ You can use DTR as part of your CI and CD processes
- ❑ DTR has a web based user interface, too!

Availability & Efficiency

Availability

DTR is highly available through the use of multiple replicas of all containers and metadata such that if a machine fails, DTR continues to operate and can be repaired

Efficiency

DTR has the ability to cache images closer to users to reduce the amount of bandwidth used during docker pulls DTR has the ability to clean up unreferenced manifests and layers

Built-in Access Control

- ❑ DTR uses the same authentication mechanism as Docker Universal Control Plane
- ❑ Users can be managed manually or synched from LDAP or Active Directory
- ❑ DTR uses Role Based Access Control (RBAC) to allow you to implement fine-grained access control policies for who has access to your Docker images

Security Scanning

- ❑ DTR's built in security scanner can be used to discover versions of software
- ❑ It scans each layer and aggregates the results to give you a complete picture
- ❑ Vulnerable databases are kept up to date through periodic updates
- ❑ This serves to give you unprecedented insight into your exposure

Deploy Docker Registry Server

Deploy Registry Server

- ❑ Before you can deploy a registry, you need to install Docker on the host
- ❑ A registry is an instance of the registry **image**, and runs within Docker
- ❑ This topic provides basic information about deploying and configuring a registry

Run a Local Registry

- ❑ Use a command like the following to start the registry container:

```
$ docker run -d -p 5000:5000 --restart=always  
--name registry registry:2
```

- ❑ That's it. No, seriously!

Interacting with a Registry

From Docker Hub to Registry

- ❑ You can pull an image from Docker Hub and push it to your registry
- ❑ The following example pulls the `ubuntu:16.04` image from Docker Hub and re-tags it as `my-ubuntu`, then pushes it to the local registry
- ❑ Finally, the `ubuntu:16.04` and `my-ubuntu` images are deleted locally and the `my-ubuntu` image is pulled from the local registry
- ❑ To get started, we'll pull the `ubuntu:16.04` image down from Docker Hub

```
$ docker pull ubuntu:16.04
```

From Docker Hub to Registry

- ❑ Tag the image as `localhost:5000/my-ubuntu`
- ❑ This creates an additional tag for the existing image!
- ❑ When the first part of the **tag** is a hostname and port, Docker interprets this as the location of a registry, when pushing:

```
$ docker tag ubuntu:16.04 localhost:5000/my-ubuntu
```

- ❑ Push the image to the local registry running at `localhost:5000`:

```
$ docker push localhost:5000/my-ubuntu
```

From Docker Hub to Registry

- ❑ Tag the image as localhost:5000/my-ubuntu
- ❑ This creates an additional tag for the existing image
- ❑ When the first part of the tag is a hostname and port, Docker interprets this as the location of a registry, when pushing

```
$ docker image remove ubuntu:16.04
```

```
$ docker image remove localhost:5000/my-ubuntu
```

- ❑ Push the image to the local registry running at localhost:5000:

```
$ docker pull localhost:5000/my-ubuntu
```

Stop a Local Registry

- ❑ To stop the registry, use the same `docker container stop` command as with any other container

```
$ docker container stop registry
```

- ❑ It's that easy!
- ❑ To remove the container, use `docker container rm`:

```
$ docker container stop registry &&  
docker container rm -v registry
```

Basic Configuration

Start the Registry Automatically

- ☐ To configure the container, you can pass additional or modified options to the `docker run` command
- ☐ If you want to use the registry as part of your permanent infrastructure, you should set it to restart **automatically** when Docker restarts or if it exits
- ☐ This example uses the `--restart always` flag to set a restart policy for the registry:

```
$ docker run -d \
-p 5000:5000 \
--restart=always \
--name registry \
registry:2
```



Customize the Published Port

- ❑ If you are already using port 5000, or you want to run multiple local registries to separate areas of concern, you can customize the registry's port settings
- ❑ This example runs the registry on port 5001 and names it `registry-test`
- ❑ **Remember**, the first part of the `-p` value is the host port and the second part is the port within the container
- ❑ Within the container, the registry listens on port 5000 by **default**

```
$ docker run -d \
-p 5001:5000 \
--name registry-test \
registry:2
```

Customize the Published Port

- ❑ If you want to change the port the registry listens on within the container, you can use the environment variable `REGISTRY_HTTP_ADDR` to change it
- ❑ This command causes the registry to listen on port 5001 within the container:

```
$ docker run -d \
-e REGISTRY_HTTP_ADDR=0.0.0.0:5001 \
-p 5001:5001 \
--name registry-test \
registry:2
```



Not secure!

A red handwritten-style arrow points from the word "Not" in the red text "Not secure!" to the `REGISTRY_HTTP_ADDR` environment variable in the Docker command line.

Customize the Storage Location

- ❑ Registry data is persisted as a docker volume on the host filesystem
- ❑ If you want to store your registry contents at a specific location on your host filesystem, you might decide to use a bind mount instead
- ❑ A bind mount is more dependent on the filesystem layout of the Docker host, but more performant in many situations
- ❑ The following example bind mounts the host directory `/mnt/registry` into the registry container at `/var/lib/registry/`

```
$ docker run -d -p 5000:5000 \
--restart=always --name registry \
-v /mnt/registry:/var/lib/registry \
registry:2
```

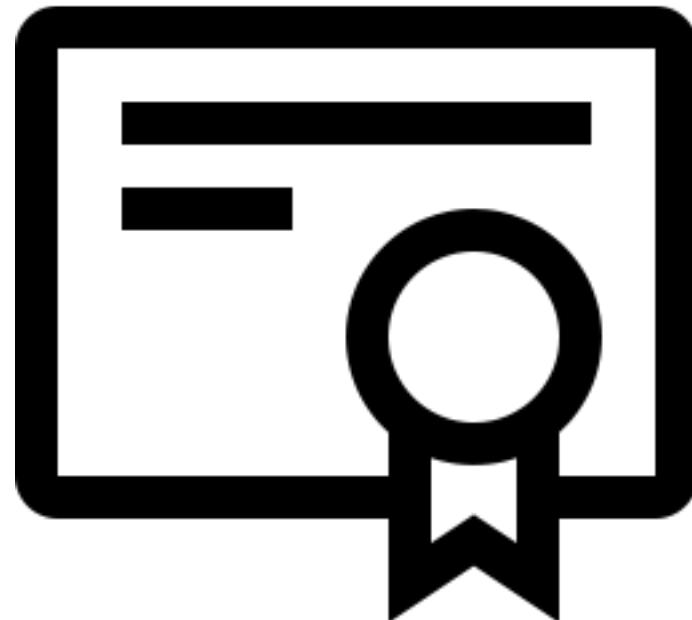
Run Externally Accessible Registry

Run Externally Accessible Registry

- ❑ Running a registry only accessible on localhost has limited usefulness
- ❑ In order to make a registry accessible to external hosts, you must first secure it using Transport Layer Security (TLS)
- ❑ This example is extended in [Run a registry as a service](#) below.

Get a Certificate

- These examples assume the following:
 - Your registry URL is `https://myregistry.domain.com/`
 - Your DNS, routing, and firewall settings allow access to the registry's host on 5000
 - You have already obtained a certificate from a certificate authority (CA)



Get a Certificate

- ❑ Firstly, we'll create a certs directory:

```
$ mkdir -p certs
```

- ❑ Copy the .crt and .key files from the CA into the certs directory.
- ❑ The following steps assume that the files are named:
 - domain.crt
 - domain.key
- ❑ In this situation, we'll stop any registry currently running before we proceed

Get a Certificate

- ❑ Restart the registry, directing it to use the TLS certificate
- ❑ This command bind-mounts the certs/directory into the container at /certs/, and sets environment variables
- ❑ The registry runs on port 443, the default HTTPS port:

```
$ docker run -d \
--restart=always \
--name registry \
-v `pwd`/certs:/certs \
-e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
-e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
-p 443:443 \
registry:2
```

Get a Certificate

- ❑ Docker clients can now pull from and push to your registry using its external address
- ❑ The following commands demonstrate this:

```
$ docker pull ubuntu:16.04
$ docker tag ubuntu:16.04 myregistrydomain.com/my-ubuntu
$ docker push myregistrydomain.com/my-ubuntu
$ docker pull myregistrydomain.com/my-ubuntu
```

Run the Registry as a Service

Run the Registry as a Service

- ❑ Swarm services provide several advantages over standalone containers
- ❑ They use a declarative model, which means that you define the desired state and Docker works to keep your service in that state
- ❑ Services provide automatic load balancing scaling, and the ability to control the distribution of your service, among other advantages.
- ❑ Services also allow you to store sensitive data such as TLS certificates in secrets.

Run the Registry as a Service

- ❑ The storage back-end you use determines whether you use a fully scaled service or a service with either only a single node or a node constraint
- ❑ If you use a distributed storage driver, such as Amazon S3, you can use a fully replicated service
- ❑ Each worker can write to the storage back-end without causing write conflicts
- ❑ If you use a local bind mount or volume, each worker node writes to its own storage location, which means that each registry contains a different data set
- ❑ You can solve this problem by using a single-replica service and a node constraint to ensure that only a single worker is writing to the bind mount
- ❑ The following example starts a registry as a single-replica service, which is accessible on any swarm node on port 80
- ❑ It assumes you are using the same TLS certificates as in the previous examples

Run the Registry as a Service

- ❑ First, save the TLS certificate and key as secrets:

```
$ docker secret create domain.crt certs/domain.crt  
$ docker secret create domain.key certs/domain.key
```

- ❑ Next, add a label to the node where you want to run the registry
- ❑ To get the node's name, use `docker node ls`
- ❑ Substitute your node's name for `node1` below:

```
$ docker node update --label-add registry=true node1
```

Run the Registry as a Service

- ❑ Next, create the service, granting it access to the two secrets and constraining it to only run on nodes with the label `registry=true`
- ❑ Besides the constraint, you are also specifying that only a single replica should run at a time
- ❑ The example bind-mounts `/mnt/registry` on the swarm node to `/var/lib/registry/` within the container
- ❑ Bind mounts rely on the pre-existing source directory, so be sure `/mnt/registry` exists on node1
- ❑ You might need to create it before running the following `docker service create` command

Run the Registry as a Service

- By default, secrets are mounted at /run/secrets/<secret-name>

```
$ docker service create \
--name registry \
--secret domain.crt \
--secret domain.key \
--constraint 'node.labels.registry==true' \
--mount type=bind,src=/mnt/registry,dst=/var/lib/registry \
-e REGISTRY_HTTP_ADDR=0.0.0.0:80 \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/run/secrets/domain.crt \
-e REGISTRY_HTTP_TLS_KEY=/run/secrets/domain.key \
--publish published=80,target=80 \
--replicas 1 \
registry:2
```

- You can access the service on port 80 of any swarm node

Restricting Access

Restricting Access

- ❑ Except for registries running on secure local networks, registries should always implement access restrictions
- ❑ The simplest way to achieve access restriction is through basic authentication
- ❑ This example uses native basic authentication using `htpasswd` to store the secrets.



Restricting Access

- ❑ Create a password file with one entry for the user testuser, with password testpassword:

```
$ mkdir auth  
$ docker run \  
    --entrypoint htpasswd \  
    registry:2 -Bbn testuser testpassword > auth/htpasswd
```

- ❑ Stop the previous registry:

```
$ docker container stop registry
```

- ❑ Now, let's start our registry with the basic authentication!

Restricting Access

□ Start the registry with basic authentication

```
$ docker run -d \
-p 5000:5000 \
--restart=always \
--name registry \
-v `pwd`/auth:/auth \
-e "REGISTRY_AUTH=htpasswd" \
-e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
-e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
-v `pwd`/certs:/certs \
-e REGISTRY_HTTP_CERTIFICATE=/certs/domain.crt \
-e REGISTRY_HTTP_KEY=/certs/domain.key \
registry:2
```

□ Try to pull an image from the registry, or push an image to the registry

Restricting Access

- These commands fail! But, why?
- Because it works! We have got to log in...
- Log in to the registry

```
$ docker login myregistrydomain.com:5000
```

- Provide the username and password from the first step
- Now we can pull an image from the registry or push an image to the registry

Deploy Using a Compose File

Deploy Using a Compose File

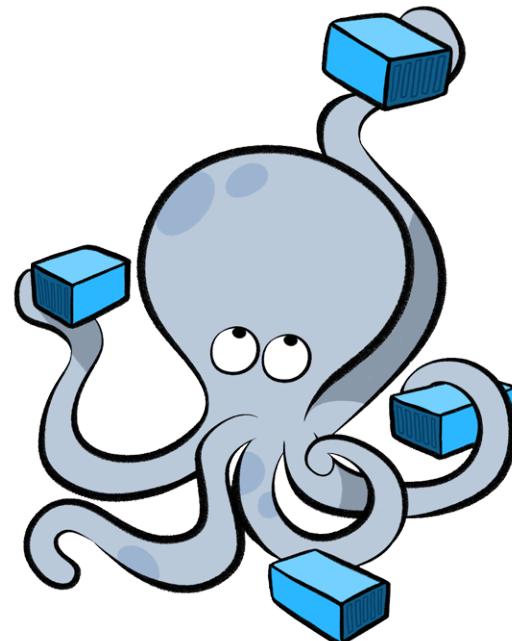
- ❑ If your registry invocation is advanced, it may be easier to Docker Compose
- ❑ Use the following example docker-compose.yml as a template:

```
registry:  
  restart: always  
  image: registry:2  
  ports:  
    - 5000:5000  
  environment:  
    REGISTRY_HTTP_TLS_CERTIFICATE: /certs/domain.crt  
    REGISTRY_HTTP_TLS_KEY: /certs/domain.key  
    REGISTRY_AUTH: htpasswd  
    REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd  
    REGISTRY_AUTH_HTPASSWD_REALM: Registry Realm  
  volumes:  
    - /path/data:/var/lib/registry  
    - /path/certs:/certs  
    - /path/auth:/auth
```

Deploy Using a Compose File

- ❑ Start it up by issuing the following in the directory containing the docker-compose.yml file:

```
$ docker-compose up -d
```



*daemon
mode*

Hands-on Exercise(s)

End of Chapter

Docker Compose

MANAGING MULTI-CONTAINER DOCKER APPLICATIONS

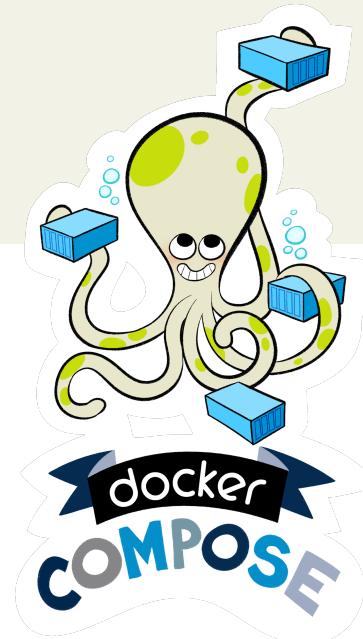
Overview

Overview

- ❑ So, we've seen a bit of it already, but what is Docker Compose?

“Docker Compose is a tool for defining and running complex applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running.”

— github.com/docker/compose



Compose Introduction

- ❑ When working with **multiple** containers, it can be difficult to manage starting, along with the configuration, of variables and links.
- ❑ To solve this problem, Docker has a tool called Docker Compose, to manage the **orchestration** and launching of containers.

Compose Introduction

- ❑ Docker Compose is **based** on a `docker-compose.yml` file.
- ❑ This file **defines** all of the containers and settings you need to launch your set of clusters.
- ❑ The properties **map** onto how you use the `docker run` commands, however, are now stored in source control and shared along with your code.

Getting to Know Compose

- ❑ An application using Docker containers will typically consist of multiple containers
- ❑ With Docker Compose, there is no need to write shell scripts to start your containers
- ❑ All the containers are defined in a configuration file
- ❑ *Services*, and then docker-compose script is used to start, stop, and restart the application
- ❑ And all the services in that application, and all the containers within that service

Compose Commands

Command	Description	Command	Description
build	Build or rebuild services	restart	Restart services
help	Get help on a command	rm	Remove stopped containers
kill	Kill containers	run	Run a one-off command
logs	View output from containers	scale	Set number of containers for services
port	Print the public port for a port binding	start	Start services
ps	List containers	stop	Stop services
pull	Pulls service images	up	Create and start containers

Compose Commands

- ❑ An application using Docker containers will typically consist of multiple containers
- ❑ With Docker Compose, there is no need to write shell scripts to start your containers
- ❑ All the containers are defined in a configuration file
- ❑ *Services*, and then docker-compose script is used to start, stop, and restart the application
- ❑ And all the services in that application, and all the containers within that service

Installation of Docker Compose

Installation of Docker Compose

- You can run Compose on macOS, Windows, and 64-bit Linux

Prerequisites

- Docker Compose relies on Docker Engine for any meaningful work, so make sure you have Docker Engine installed either locally or remote, depending on your setup
- On desktop systems like Docker for Mac and Windows, Docker Compose is included as part of those desktop installs
- On Linux systems, first install the Docker for your OS as described on the Get Docker page, then come back here for instructions on installing Compose on Linux systems
- To run Compose as a non-root user, see "Manage Docker as a non-root user"

Installation of Docker Compose on Linux

- ❑ On **Linux**, you can download the Docker Compose binary
- ❑ Run this command to download the latest version of Docker Compose:

```
$ sudo curl -L https://github.com/docker/compose/releases/download/1.19.0/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

- ❑ Apply executable permissions to the binary:

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

- ❑ Optionally, install command completion for the `bash` and `zsh` shell.
- ❑ Test the installation.

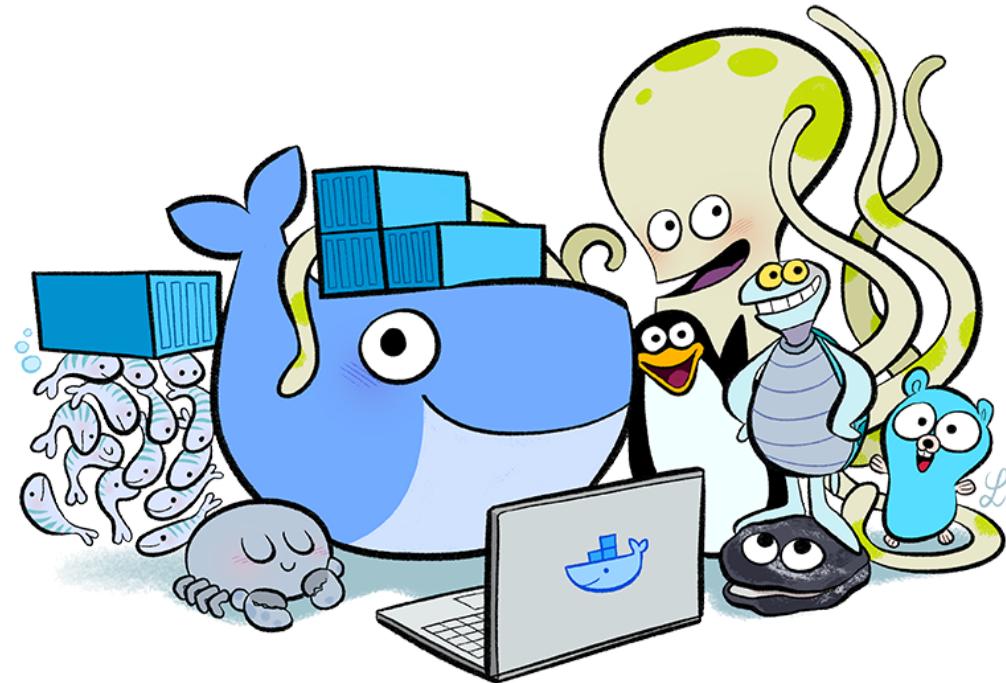
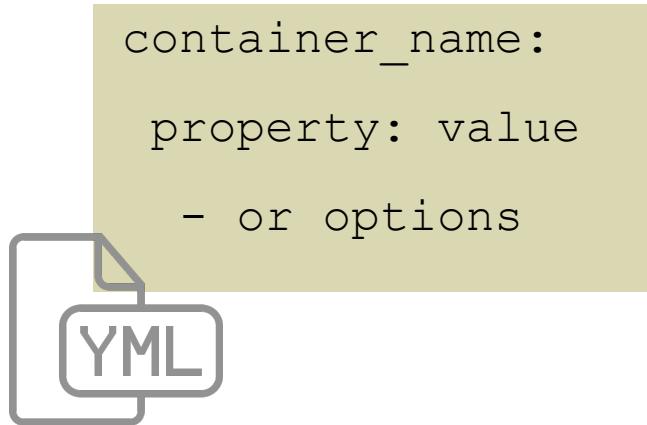
```
$ docker-compose --version
```

```
docker-compose version 1.19.0, build 1719ceb
```

Docker Compose Configuration File

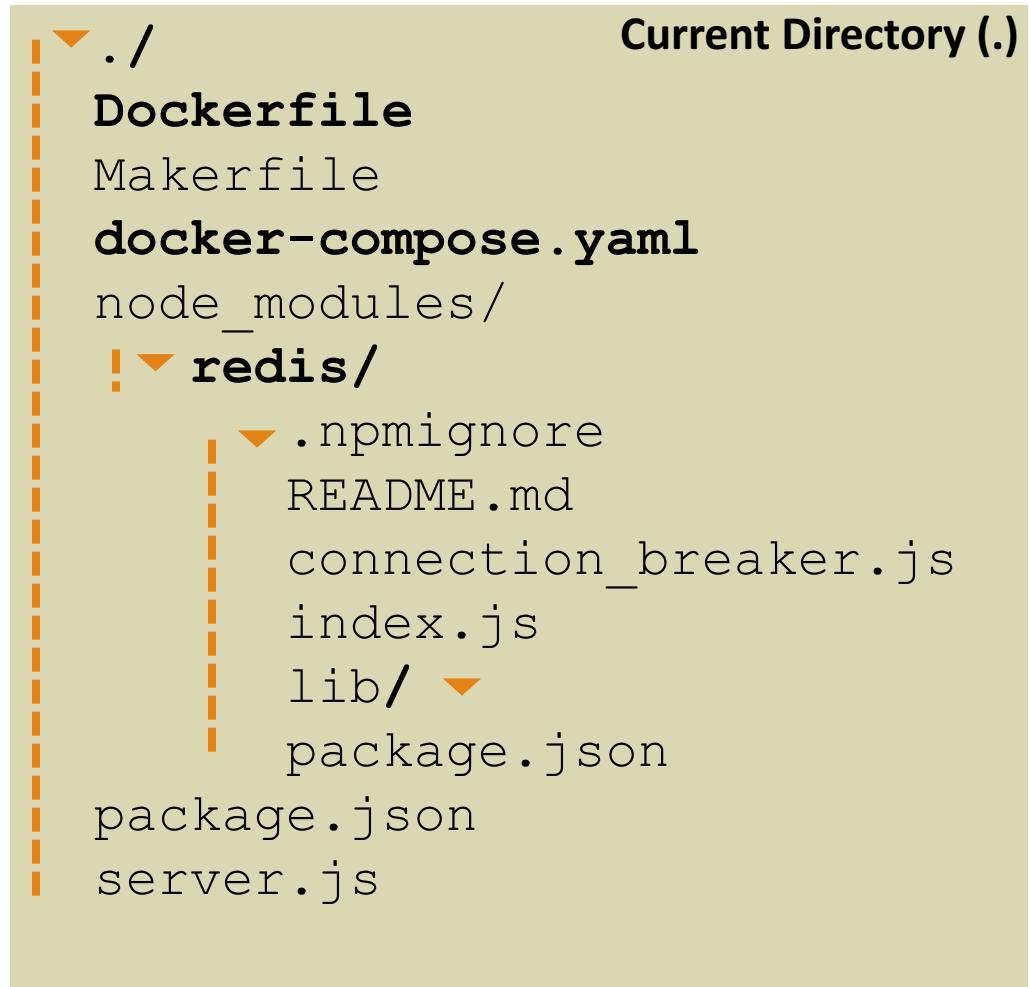
YAML Format

- The **format** of YAML (Yet Another Markup Language), looks like this:



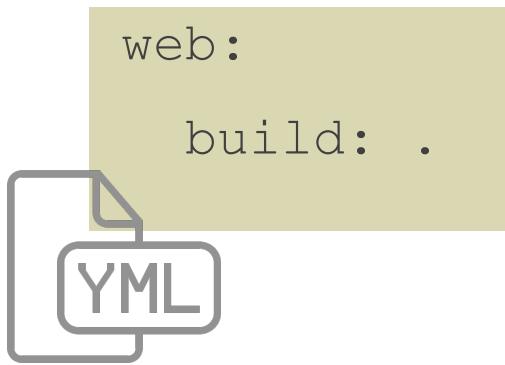
Defining Our First Container

- In this **example**, let's say we have a Node.js application, which requires connecting to Redis.
- To start, we need to **define** our docker-compose.yml file to launch the Node.js application.



Formatting Basic

- ❑ Given the **format** shared previously, the file needs to name the container 'web' and set the build property to the current directory.



- ❑ Copying the following code into our YAML file, will **define** a container called web, which is based on the build of the current directory.
- ❑ NOTE: Remember we have a Node.js app in our current directory.

Defining Container Properties

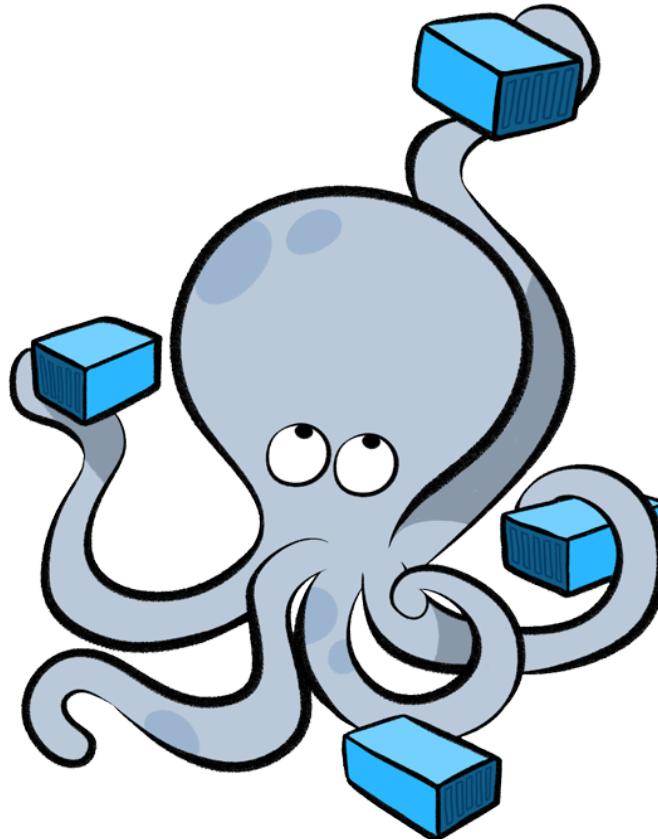
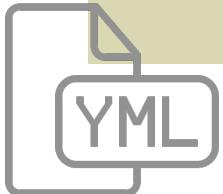
- ❑ Docker Compose supports all of the properties which can be defined using `docker run`.
- ❑ To link two containers together to specify a `links` property and list required connections.
- ❑ For example, the following would link to the `redis` source container defined in the same file and assign the same name to the alias.

```
links:  
  - redis
```

Defining Container Properties

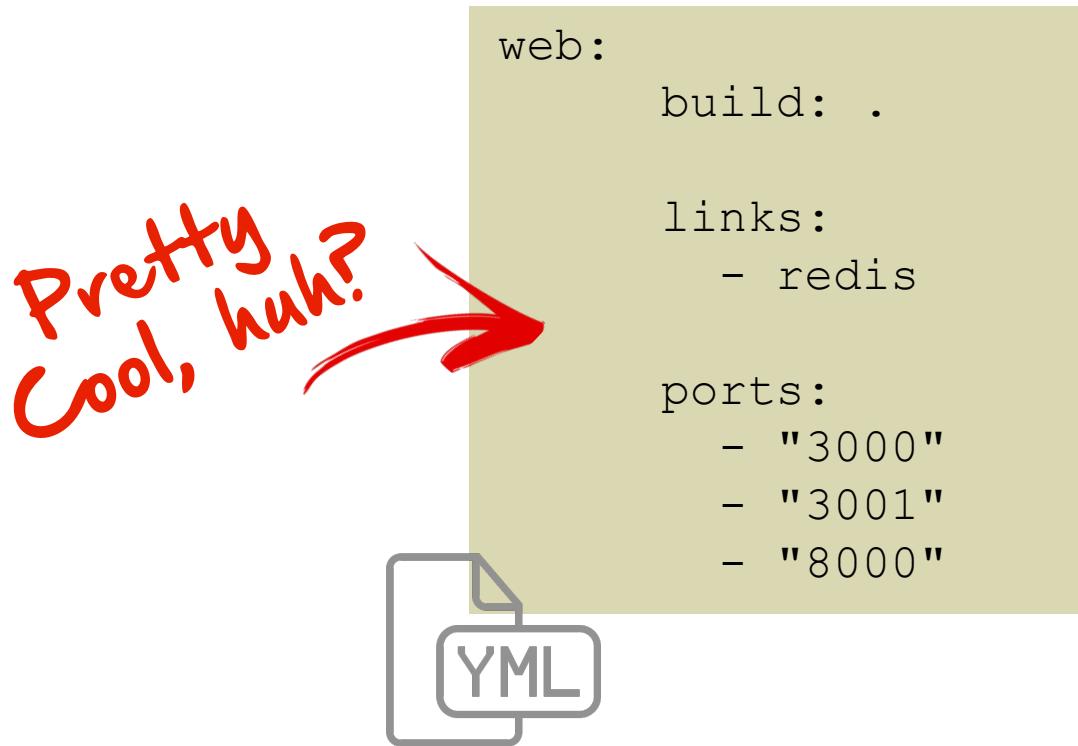
- The same format is used for other properties, such as **ports**:

```
ports:  
  - "3000"  
  - "3001"  
  - "8000"
```



First Container Overview

- ❑ So, after all that, our yaml file would look something like this



- ❑ This is just the very beginning!

Building a Multi-Container Application

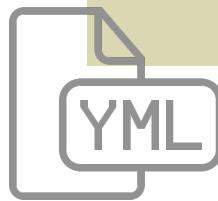
Defining a Second Container

- ❑ In the previous step, we used the `Dockerfile` in the current directory (`.`) as the **base** for our container
- ❑ In this step, we want to use an existing **image** from Docker Hub as a second container
- ❑ To define the second container you simply use the same **format** as before on a new line
- ❑ The YAML format is flexible enough to define multiple **containers** within the same file

Second Container Properties

- Defining the **second** container with the name `redis`, which will use the image `redis`. Following the YAML format, the container details would be:

```
redis:  
  image: redis:alpine  
  volumes:  
    - /var/redis/data:/data
```



Starting Compose Up

- With the `docker-compose.yml` file we created in place, you can launch all the applications with a single command of `up`, like this:

```
docker compose up
```

- If you wanted to bring up a single container, then we could use `up <name>`.

```
docker compose up <name>
```

Docker Compose Management

- ❑ Docker Compose starts containers, but it also provides a commands to **manage** all the containers.
- ❑ To see the details of the **launched** containers we could use:

```
docker-compose ps
```

- ❑ To access all the **logs** via a single snapshot stream we could use:

```
docker-compose logs
```
- ❑ Other commands follow the same pattern, and as always can be seen by typing the command itself: `docker-compose`.

Scaling With Compose

- ❑ Let's talk about how Compose can also be used to **scale** up the number of running containers.
- ❑ The scale option allows you to specify the service and then the **number** of instances you want.
- ❑ If the number is **greater** than the instances already running, it will launch additional containers.
- ❑ And if the number is **less**, it will stop the unrequired containers.

Scaling With Compose

- To **scale** the number of web containers we're running we'd use this command:

```
docker-compose scale web=3
```

- And we could scale it back **down** using:

```
docker-compose scale web=1
```



Stop and Remove

- ❑ As when we launch an application with start, to **stop** a set of containers we can use the command:

```
docker-compose stop
```

- ❑ And to **remove** all the containers we can use the command:

```
docker-compose rm
```

Multi-Container Application Management

MANAGE APPS SPANNING MULTIPLE CONTAINERS

Multi-Container Application Using Compose

Multi-Container Application Using Compose

- ❑ The application used in this section is a Java EE application talking to a database
- ❑ The application publishes a REST endpoint that can be invoked using ‘curl’
- ❑ It is deployed using WildFly Swarm that communicates to MySQL database.
- ❑ WildFly Swarm and MySQL will be running in two separate containers, and thus making this a multi-container application

Multi-Container Application Using Compose

- ❑ The entry point to Docker Compose is a Compose file, usually called `docker-compose.yml`
- ❑ Create a new directory `javaee`. In that directory, create a new file `docker-compose.yml` in it.
- ❑ Use the following contents:

Reference: `cs_09-35.yaml`

Our Compose File

- ❑ Two services in this Compose are defined by the name db and web attributes
- ❑ Image name for each service defined using image attribute
- ❑ The mysql:8 image starts the MySQL server
- ❑ Environment attribute defines environment variables to initialize MySQL server
- ❑ Java EE application uses the db service as specified in the connection-url as specified
- ❑ The arungupta/docker-javaee:dockerconeu17 image starts WildFly Swarm application server
- ❑ Port forwarding is achieved using ports attribute
- ❑ depends_on attribute allows to express dependency between services

Start Our Application

- ❑ All services in the application can be started, in detached mode, by giving the command:

```
$ docker-compose up -d
```

- ❑ An alternate Compose file name can be specified using `-f` option
- ❑ An alternate directory where the compose file exists can be specified using `-p` option
- ❑ This shows the output as:

```
Creating network "javaee_default" with the default driver
Creating db ...
Creating db ... done
Creating javaee_web_1 ...
Creating javaee_web_1 ... done
```

Start Our Application

- ❑ We've learned we can verify using `docker-compose ps`
- ❑ This provides a consolidated view of all the services, and containers within each of them
- ❑ Alternatively, the containers in this application are verified by using the usual `docker container ls` command
- ❑ Service logs can be seen using `docker-compose logs` command
- ❑ `depends_on` attribute in the Compose definition file ensures the container start up order

Verify Our Application

- ❑ Now that the WildFly Swarm and MySQL have been configured, let's access the application!
- ❑ You need to specify IP address of the host where WildFly is running
- ❑ The endpoint can be accessed in this case as:

```
$ curl -v http://localhost:8080/resources/employees
```

- ❑ The output is shown as:

Reference: cs_09-39.bash

- ❑ This shows the result from querying the database
- ❑ Um, what if we just want one?

Verify Our Application

- ❑ A single resource can be obtained:

```
$ curl -v http://localhost:8080/resources/employees/1
```

- ❑ It shows the output:

```
Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /resources/employees/1 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Connection: keep-alive
< Content-Type: application/xml
< Content-Length: 104
< Date: Sat, 07 Oct 2017 00:06:33 GMT
< * Curl_http_done: called premature == 0 * Connection #0 to host localhost left intact
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><employee><id>1</id><name>Penny</name></employee>
```

Reference: cs_09-40.bash

Shutdown Our Application

- ❑ Shutdown the application using docker-compose down:

```
Stopping javaee_web_1 ... done
Stopping db ... done
Removing javaee_web_1 ... done
Removing db ... done
Removing network javaee_default
```

- ❑ This stops the container in each service and removes all the services
- ❑ It also deletes any networks that were created as part of this application.

Multi-Container Application Using Compose and Swarm Mode

Multi-Container Application Using Swarm

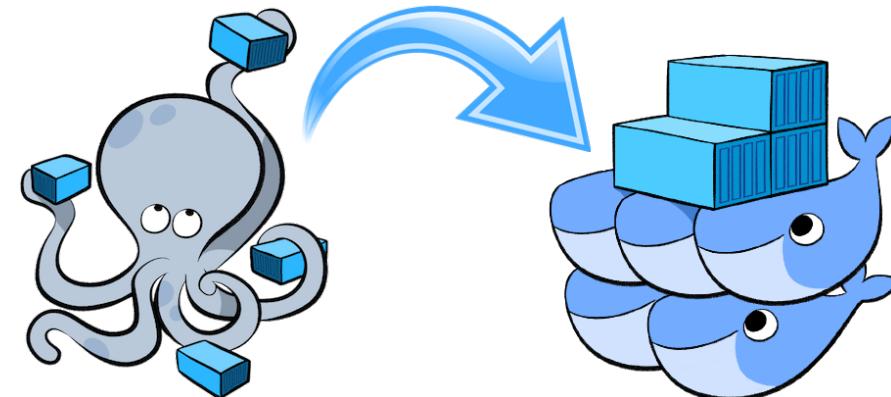
- ❑ This section will deploy an application that will provide a CRUD/REST interface on a data bucket in MySQL
- ❑ This is achieved by using a Java EE application deployed on WildFly to access the database

Initialize Swarm

- ❑ Initialize Swarm mode using the following command:

```
$ docker swarm init
```

- ❑ This starts a Swarm Manager!



Multi-Container Application Using Swarm

- ❑ Find some information about this one-node cluster using the command `docker info`
- ❑ The log output is large, and gives extensive information as we've already seen:

Reference: `cs_09-info.bash`

- ❑ This cluster has 1 node, and that is a manager
- ❑ By default, the manager node is also a worker node
- ❑ This means the containers can run on this node

Multi-Container Application Using Swarm

- ❑ This section describes how to deploy a multi-container application using Docker Compose to Swarm mode use Docker CLI
- ❑ Create a new directory and cd to it:

```
$ mkdir webapp  
$ cd webapp
```

- ❑ Create a new Compose definition `docker-compose.yml` using our previous configuration file:

Reference: `cs_09-35.yaml`

Multi-Container Application Using Swarm

- ❑ This application can be started as:

```
$ docker stack deploy --compose-file=docker-compose.yml webapp
```

- ❑ This shows the output as:

```
Ignoring deprecated options:  
container_name: Setting the container name is not supported.  
Creating network webapp_default  
Creating service webapp_db  
Creating service webapp_web
```

- ❑ WildFly Swarm and MySQL services are started on this node
- ❑ A new overlay network is created

Verify Service and Containers in Application

- ❑ Based on our previous work with Compose, how can we verify our service and containers application?
- ❑ Verify that the WildFly and MySQL services are running using `docker service ls`
- ❑ More details about the service can be obtained using `docker service inspect webapp_web`
- ❑ Logs for the service can be seen using `docker service logs -f webapp_web`

Access Application

- ❑ Now that the WildFly and MySQL servers have been configured, let's access the application
- ❑ You need to specify IP address of the host where WildFly is running
- ❑ The endpoint can be accessed in this case as:

```
$ curl -v http://localhost:8080/resources/employees
```

- ❑ The output is shown as:

Reference: cs_09-curl.bash

- ❑ This shows all employees stored in the database!

Multi-Container Application Using Swarm

- ❑ If you only want to stop the application temporarily while keeping any networks that were created as part of this application, you can:

```
$ docker service scale webapp_db=0 webapp_web=0
```

- ❑ ...the recommended way is to set the amount of service replicas to 0
- ❑ It shows the output:

```
webapp_db scaled to 0
```

```
webapp_web scaled to 0
```

Since `--detach=false` was not specified, tasks will be scaled in the background.

In a future release, `--detach=false` will become the default.

Multi-Container Application Using Swarm

- ❑ Shutdown the application using `docker stack rm webapp`:
- ❑ Output:

```
Removing service webapp_db
Removing service webapp_web
Removing network webapp_default
```

- ❑ This stops the container in each service and removes the services. It also deletes any networks that were created as part of this application

Hands-on Exercise(s)

End of Chapter

Docker on the Cloud

MANAGE BUILDS, SWARMS, INFRASTRUCTURE, AND NODES

Introduction to Docker Cloud

Introduction to Docker Cloud

Docker says:

“Docker Cloud provides a hosted service with build and testing facilities for Dockerized application images; tools to help you set up and manage host infrastructure; and application lifecycle features to automate deploying services created from images.”

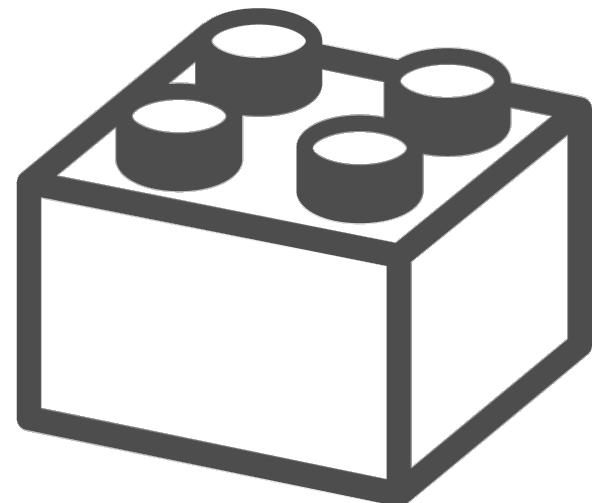


What is a node?

- ❑ A node is an individual Linux host used to deploy and run your applications
- ❑ Docker Cloud does not provide hosting services, so all of your applications, services, and containers run on your own hosts
- ❑ Your hosts can come from several different sources, including physical servers, virtual machines or cloud providers

What is a node cluster?

- ❑ When launching a node from a cloud provider you actually create a node cluster
- ❑ Node Clusters are groups of nodes of the same type and from the same cloud provider
- ❑ Node clusters allow you to scale the infrastructure by provisioning more nodes with a drag of a slider
- ❑ Pretty cool, huh?



What is a node cluster?

Use cloud service providers

- ❑ Docker Cloud makes it easy to provision nodes from existing cloud providers
- ❑ You can provision new nodes directly from within Docker Cloud
- ❑ Native support for AWS, DigitalOcean, Azure, Packet.net, and IBM SoftLayer.

Use your own hosts (“Bring your own nodes”)

- ❑ You can also provide your own node or nodes
- ❑ This means you can use any Linux host connected to the Internet as a Docker Cloud node as long as you can install a Cloud agent
- ❑ The agent registers itself with your Docker account, and allows you to use Docker Cloud to deploy containerized applications

What is a service?

- ❑ Services are logical groups of containers from the same image
- ❑ Services make it simple to scale your application across different nodes
- ❑ In Docker Cloud you drag a slider to increase or decrease the availability, performance, and redundancy of the application
- ❑ Services can also be linked one to another even if they are deployed on different nodes, regions, or even cloud providers

Docker ID

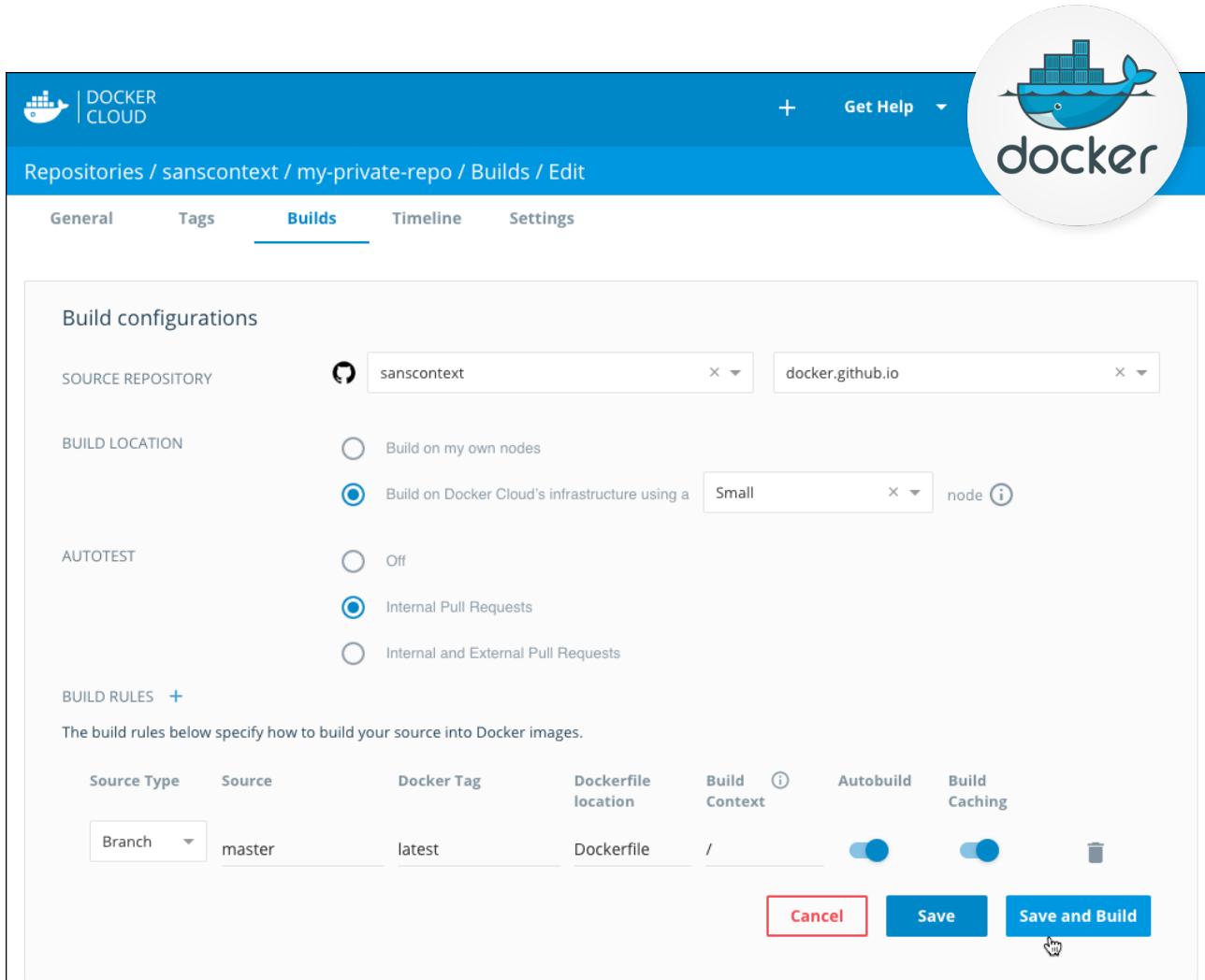
- ❑ Docker Cloud uses your Docker ID for access and access control, and this allows you to link your Hub and Cloud accounts.
- ❑ If you already have an account on Docker Hub, you can use the same credentials to log in to Docker Cloud.
- ❑ If you don't have a Docker ID yet, you can sign up for one from the Cloud website
- ❑ Or using the `docker login` command in the Docker CLI.
- ❑ The name you choose for your Docker ID becomes part of your account namespace

Advantages of Docker on the Cloud

Images, Builds, and Testing

- ❑ Docker Cloud uses the hosted Docker Cloud Registry
- ❑ Allowing you to publish Dockerized images online, either publicly or privately
- ❑ Docker Cloud can also store pre-built images
- ❑ Or, link to your source code so it can build the code into Docker images
- ❑ Optionally testing the resulting images before pushing them to a repository

Images, Builds, and Testing



The screenshot shows the Docker Cloud interface for managing build configurations. The top navigation bar includes the Docker logo, 'DOCKER CLOUD', a '+' button, 'Get Help', and a search bar featuring a whale icon and the word 'docker'.

The main page displays the build configuration for the repository 'sanscontext / my-private-repo'. The 'Builds' tab is selected, showing the following settings:

- SOURCE REPOSITORY:** sanscontext (selected from a dropdown) and docker.github.io (selected from a dropdown).
- BUILD LOCATION:** Build on Docker Cloud's infrastructure using a 'Small' node.
- AUTOTEST:** Internal Pull Requests (selected).
- BUILD RULES:** A table for defining build rules:

Source Type	Source	Docker Tag	Dockerfile location	Build Context	Autobuild	Build Caching
Branch	master	latest	Dockerfile	/	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

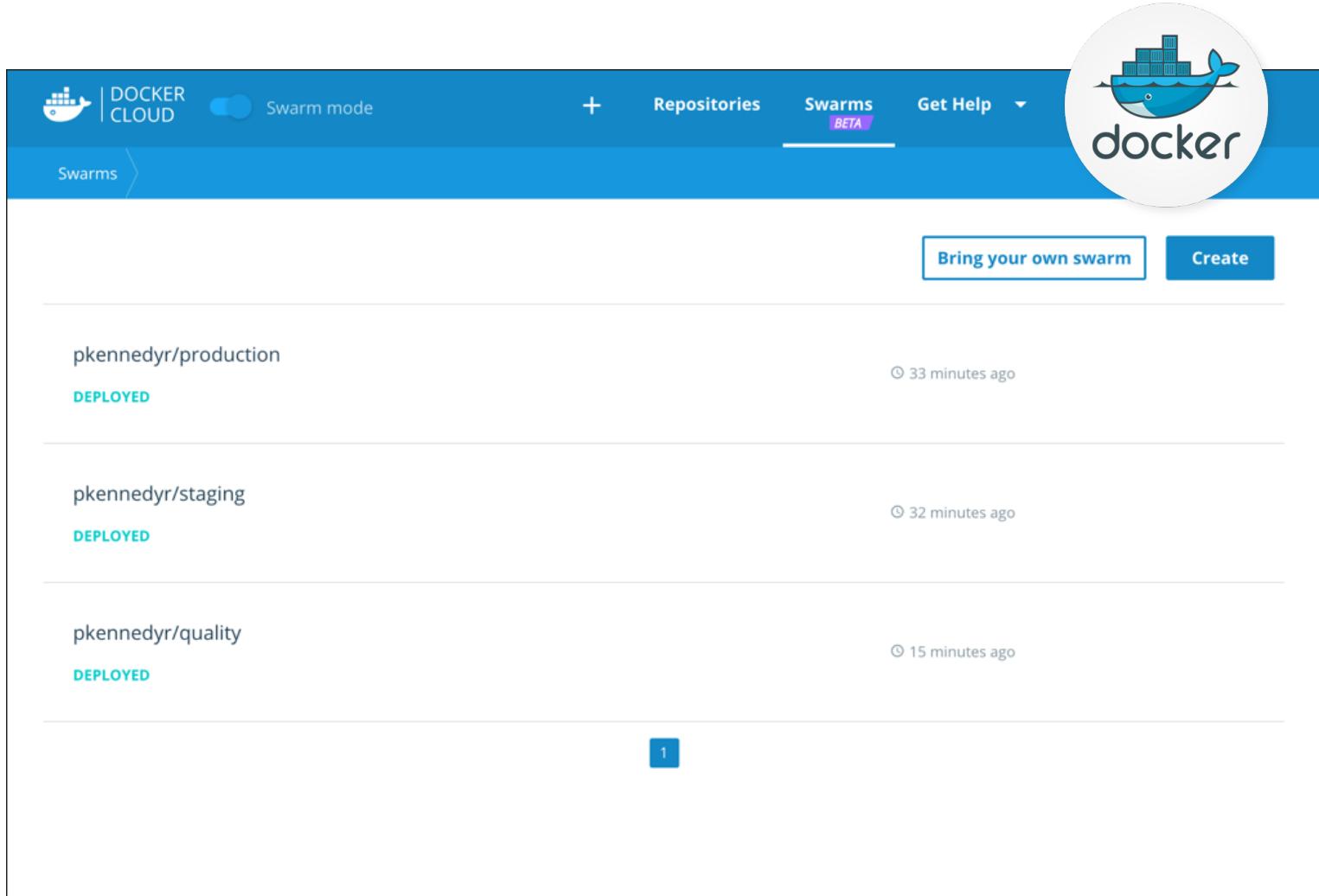
At the bottom right of the build configuration panel, there are three buttons: 'Cancel' (red), 'Save' (blue), and 'Save and Build' (blue, with a cursor icon hovering over it).

Swarm Management (Beta Swarm Mode)

- ❑ With Beta Swarm Mode, you can create new swarms from within Docker Cloud
- ❑ You can register existing swarms to Docker Cloud, or provision swarms to your cloud providers
- ❑ Your Docker ID authenticates and securely accesses personal or team swarms
- ❑ Docker Cloud allows you to connect your local Docker Engine to any swarm you have access to in Docker Cloud



Swarm Management (Beta Swarm Mode)



The screenshot shows the Docker Cloud interface with the "Swarms" tab selected. The top navigation bar includes "DOCKER CLOUD" with a ship icon, a "Swarm mode" toggle switch, and tabs for "+", "Repositories", "Swarms" (labeled "BETA"), and "Get Help". A large "docker" logo is on the right. Below the header, there are two buttons: "Bring your own swarm" and "Create". The main content area displays three deployed Docker images:

- pkennedy/production** - DEPLOYED, 33 minutes ago
- pkennedy/staging** - DEPLOYED, 32 minutes ago
- pkennedy/quality** - DEPLOYED, 15 minutes ago

A small blue page number "1" is at the bottom center.

Infrastructure management (Standard Mode)

- ❑ Before doing anything with your images, you need somewhere to run them
- ❑ Docker Cloud allows you to link to your infrastructure
- ❑ Or, like to your cloud services provider so you can provision new nodes automatically
- ❑ Once you have nodes set up, you can deploy images directly from Docker Cloud repositories

Infrastructure management (Standard Mode)

The screenshot shows the Docker Cloud Node Clusters interface. At the top, there's a header with the Docker logo, "DOCKER CLOUD", a "+" button, "Get Help", and a "docker" logo. Below the header, the title "Node Clusters" is displayed. On the left, there are filters: a checkbox for "Select All", a search bar "Filter by name...", and buttons for "Actions", "Bring your own node", and "Create".

The main area lists three environments:

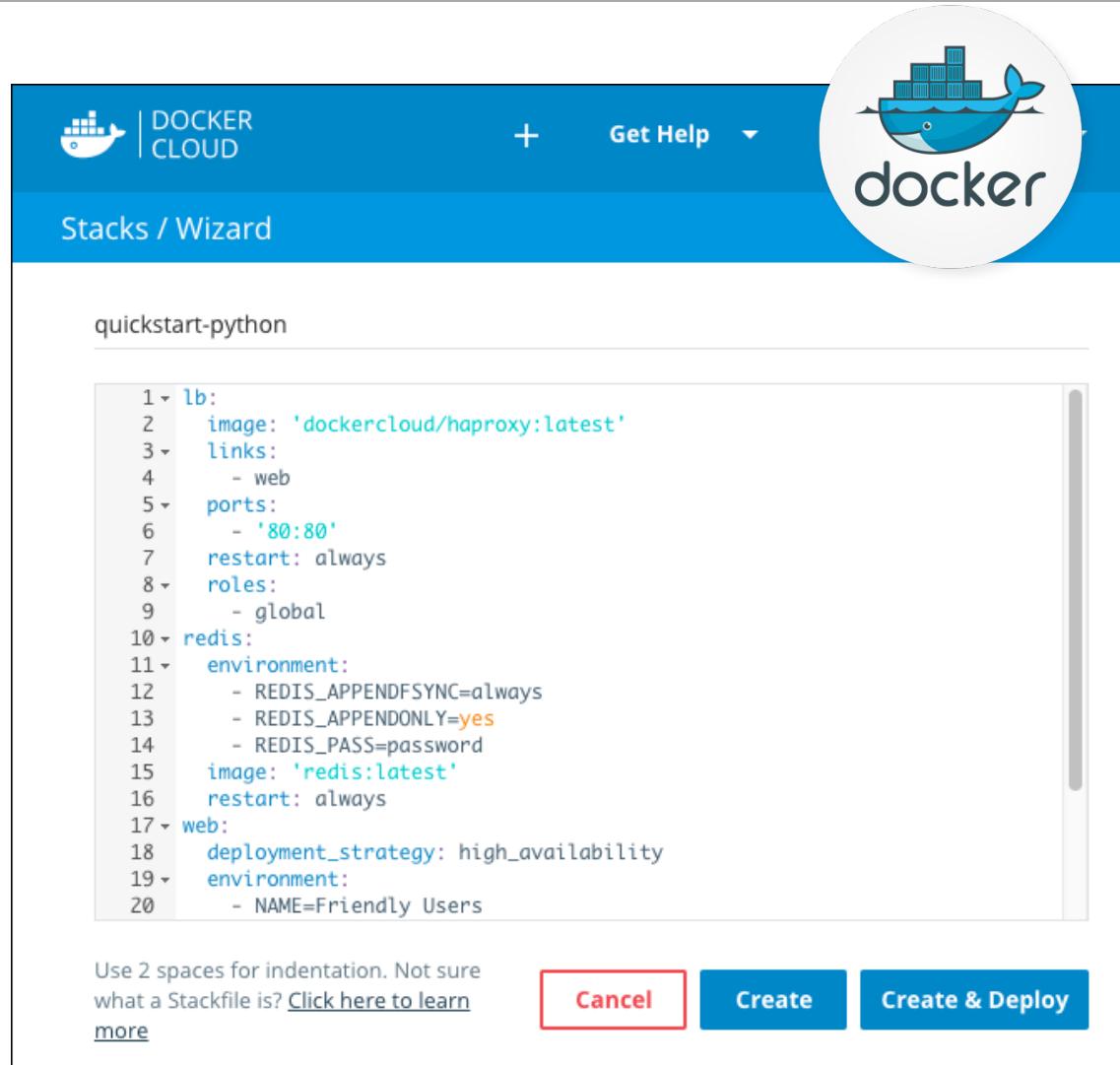
- production**: DEPLOYED, 2 nodes. One node is listed as "us-east-1 • T2.NANO" with "byon=false, prod" and was created "3 minutes ago".
- testing**: DEPLOYED, 2 nodes. One node is listed as "us-east-1 • T2.NANO" with "test, testing" and was created "2 minutes ago".
- dev-env**: DEPLOYED, 1 node. One node is listed as "us-east-1 • T2.NANO" with "dev" and was created "3 minutes ago".

Each node entry includes a "Details" icon.

Services, Stacks, and Applications (Standard Mode)

- ❑ Images are just one layer in containerized applications
- ❑ Once you've built an image, you can use it to deploy services
- ❑ Or, use Docker Cloud's **stackfiles** to combine it with other services and microservices, to form a full application

Services, Stacks, and Applications (Standard Mode)



The screenshot shows the Docker Cloud Stacks / Wizard interface. At the top, there's a navigation bar with the Docker Cloud logo, a '+' button, 'Get Help', and a 'docker' logo. Below the header, the title 'Stacks / Wizard' is displayed, followed by the stack name 'quickstart-python'. The main content area contains a code editor showing a Stackfile:

```
1 lb:
2   image: 'dockercloud/haproxy:latest'
3   links:
4     - web
5   ports:
6     - '80:80'
7   restart: always
8   roles:
9     - global
10  redis:
11    environment:
12      - REDIS_APPENDFSYNC=always
13      - REDIS_APPENDONLY=yes
14      - REDIS_PASS=password
15    image: 'redis:latest'
16    restart: always
17  web:
18    deployment_strategy: high_availability
19    environment:
20      - NAME='Friendly Users'
```

Below the code editor, a note says 'Use 2 spaces for indentation. Not sure what a Stackfile is? [Click here to learn more](#)'. At the bottom right are three buttons: 'Cancel' (red), 'Create' (blue), and 'Create & Deploy' (blue).

Manage Builds & Images

Manage Builds & Images

- ❑ Docker Cloud provides a hosted registry service where you can create repositories to store your Docker images
- ❑ You can choose to push images to the repositories, or link to your source code and build them directly in Docker Cloud
- ❑ You can build images manually, or set up automated builds to rebuild your Docker image on each git push to the source code
- ❑ You can also create automated tests, and when the tests pass use autodeploy to automatically update your running services when a build passes its tests

Docker Cloud - Automated Builds

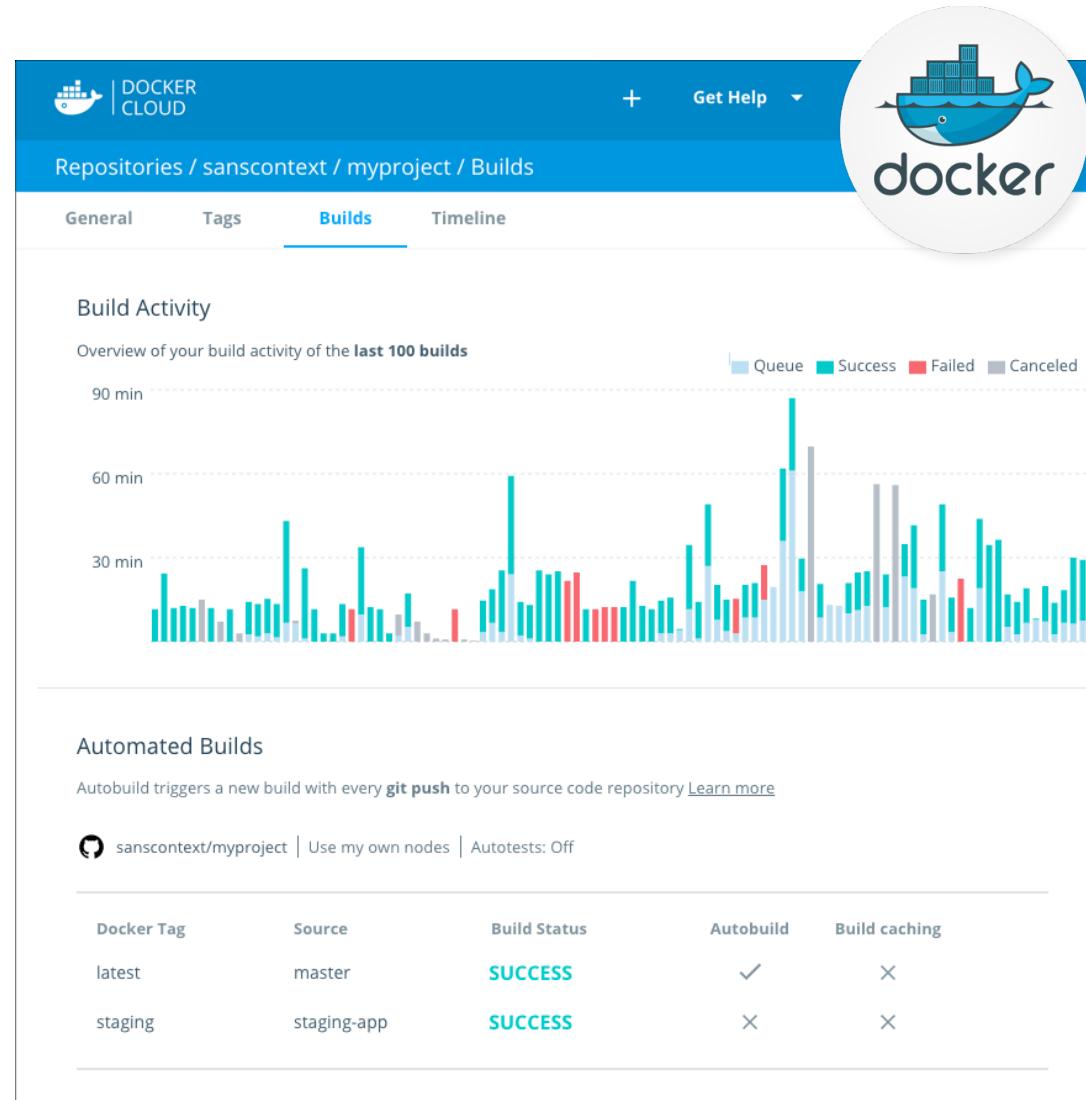
- ❑ When you set up autobuilds, you create a list of branches and tags that you want to build into Docker images
- ❑ When you push code to a source code branch for one of those listed image tags, the push uses a webhook to trigger a new build
- ❑ The built image is then pushed to the Docker Cloud registry or to an external registry

Docker Cloud - Automated Builds

- ❑ If you have automated tests configured, these run after building but before pushing to the registry
- ❑ Use these tests to create a CI workflow where a build that fails its tests does **not** push the built image
- ❑ Automated tests do not push images to the registry on their own.
- ❑ You can also just use `docker push` to push pre-built images to these repositories, even if you have automatic builds set up
- ❑ Here's a look at the Docker Cloud build dashboard...

Next,
Slide! 

Docker Cloud – Automated Builds



The screenshot shows the Docker Cloud interface for a repository named 'sanscontext/myproject'. The top navigation bar includes the Docker logo, 'DOCKER CLOUD', a '+' button, 'Get Help', and a search bar featuring a Docker whale icon.

The main content area has tabs for 'General', 'Tags', **Builds**, and 'Timeline'. The 'Builds' tab is selected, displaying 'Build Activity' for the last 100 builds. A bar chart tracks build times in 30-minute increments, with categories for Queue (light blue), Success (teal), Failed (red), and Canceled (gray). The chart shows a significant peak in build times between 60 and 90 minutes.

The 'Automated Builds' section indicates that Autobuild triggers a new build with every **git push** to the source code repository. It shows configuration for 'sanscontext/myproject' using its own nodes and no autotests.

Docker Tag	Source	Build Status	Autobuild	Build caching
latest	master	SUCCESS	✓	✗
staging	staging-app	SUCCESS	✗	✗

Manage Swarms

Manage Swarms

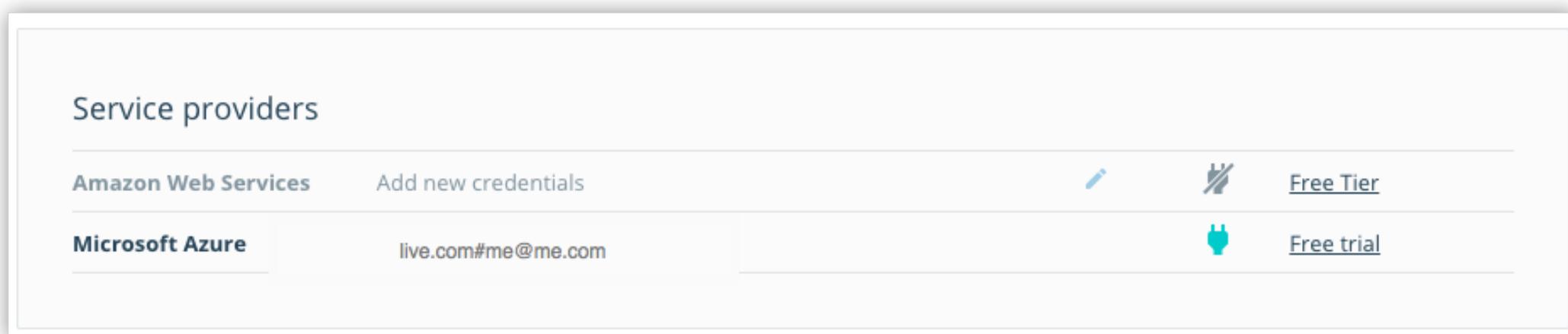
- ❑ Docker Cloud now allows you to connect to clusters of Docker Engines in swarm mode
- ❑ With Beta Swarm Mode in Docker Cloud, you can provision swarms to popular cloud providers, or register existing swarms to Docker Cloud
- ❑ Use your Docker ID to authenticate and securely access personal or team swarms

Create a New Swarm on Azure in Docker Cloud

- ❑ You can now create *new* Docker Swarms from within Docker Cloud as well as register existing swarms
- ❑ When you create a swarm, Docker Cloud connects to the Cloud provider on your behalf, and uses the provider's APIs and a provider-specific template to launch Docker instances
- ❑ The instances are then joined to a swarm and the swarm is configured using your input
- ❑ When you access the swarm from Docker Cloud, the system forwards your commands directly to the Docker instances running in the swarm

Create a New Swarm on Azure in Docker Cloud

- ❑ To create a swarm, you need to give Docker Cloud permission to deploy swarm nodes on your behalf in your cloud services provider account
- ❑ If you haven't yet linked Docker Cloud to Azure, you'll first need to follow those steps
- ❑ Once it's linked, it shows up on the **Swarms -> Create** page as a connected service provider



Create a Swarm

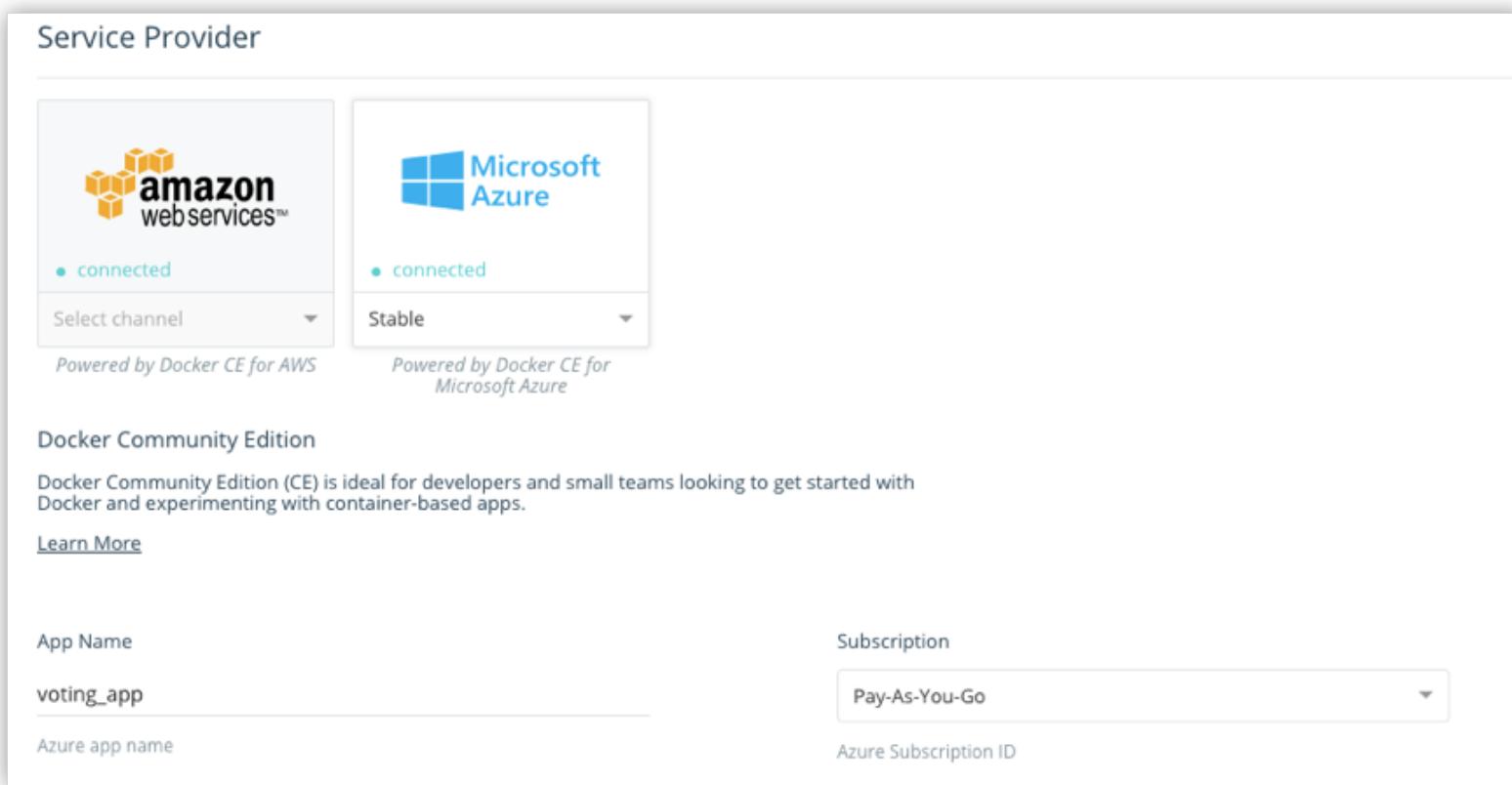
□ Here's the quick rundown:

1. If necessary, log in to Docker Cloud and switch to Swarm Mode.
2. Click **Swarms** in the top navigation, then click **Create**.
3. Enter a name for the new swarm.



Create a Swarm

4. Select Microsoft Azure as the service provider, select a channel (Stable or Edge) from the drop-down menu, provide an App name, and select the Azure Subscription you want to use.



Create a Swarm

5. Make sure that **Create new resource group** is selected, provide a name for the group, and select a location from the drop-down menu.

Resource Group

Create new resource group Use existing resource group

swarm_vote_resources

Azure Resource Group

Location

West US

Region where your swarm is provisioned

Create a Swarm

6. Choose how many swarm managers and worker nodes to deploy.

Swarm Size

Number of Swarm managers?

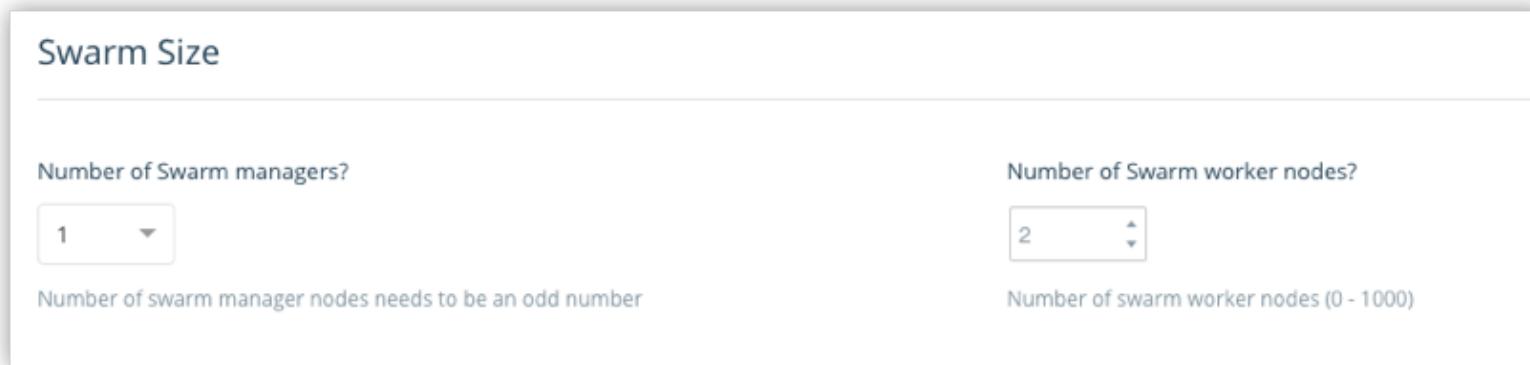
1 ▾

Number of swarm manager nodes needs to be an odd number

Number of Swarm worker nodes?

2 ▾

Number of swarm worker nodes (0 - 1000)



7. Configure swarm properties, SSH key and resource cleanup.

Swarm Properties

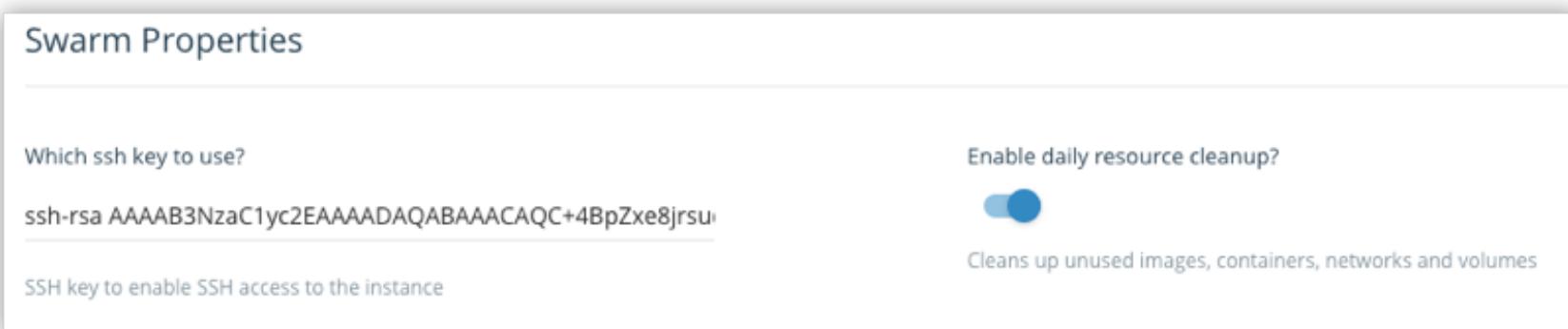
Which ssh key to use?

ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQACQC+4BpZxe8jrsu...

SSH key to enable SSH access to the instance

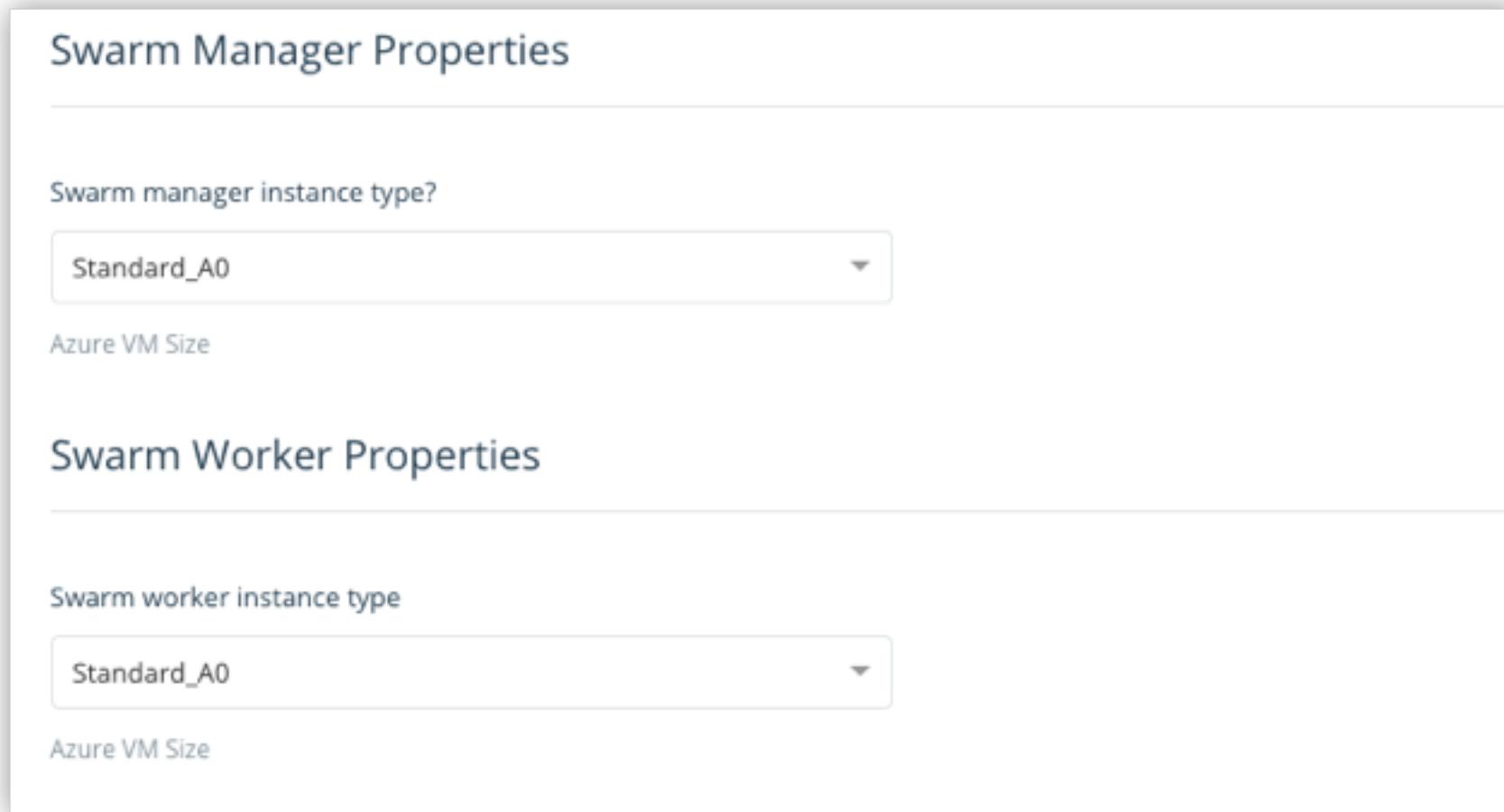
Enable daily resource cleanup?

Cleans up unused images, containers, networks and volumes



Create a Swarm

8. Select the machine sizes for the managers, and for the worker.



Create a Swarm

8. Click Create.

The screenshot shows the Docker Cloud interface with the 'Swarms' tab selected. A speech bubble contains the text 'Here's our swarm!'. Another speech bubble in the top right corner says 'Pretty cool, if you ask me!'.

Name	Provider	Registered
orangesnap/swarm_webapps	microsoft Azure	3 hours ago
orangesnap/vote_swarm	microsoft Azure	20 minutes ago

1

Manage Infrastructure

Manage Infrastructure (standard mode)

- ❑ Cloud uses an agent and system containers to deploy and manage nodes (hosts) on your behalf
- ❑ All nodes accessible to your account are connected by an overlay or mesh network

Deploy nodes from Docker Cloud

- ❑ When you use Docker Cloud to deploy nodes on a hosted provider, the service stores your cloud provider credentials
- ❑ And then deploys nodes for you using the services' API to perform actions on your behalf

Manage Infrastructure (standard mode)

Bring your own host

- ❑ If you are using Bring Your Own Host, Docker Cloud provides a script that:
 - installs the Docker Cloud Agent on the host
 - downloads and installs the latest Docker CS Engine version and the AUFS storage driver
 - sets up TLS certificates and the Docker security configuration
 - registers the host with Docker Cloud under your user account
- ❑ Once this connection is established, the Docker Cloud Agent manages the node and performs updates when the user requests them
- ❑ And it can also create and maintain a reverse tunnel to Docker Cloud if firewall restrictions prevent a direct connection

Manage Infrastructure (standard mode)

Internal networking

- ❑ Docker Cloud communicates with the Docker daemon running in the node using the following IPs, on port **2375/tcp**
- ❑ If the port is not accessible, Docker Cloud creates a secure reverse tunnel from the nodes to Docker Cloud
- ❑ When you add a node on Docker Cloud, the node joins the Weave private overlay network for containers in other nodes by connecting on ports **6783/tcp** and **6783/udp**

52.204.126.235/32

52.6.30.174/32

52.205.192.142/32

52.205.2.114/32

Manage Infrastructure (standard mode)

Internal Overlay Network

- ❑ Docker Cloud creates a per-user overlay network which connects all containers across all of the user's hosts
- ❑ This network connects all of your containers on the 10.7.0.0/16 subnet, and gives every container a local IP
- ❑ This IP persists on each container even if the container is redeployed and ends up on a different host
- ❑ Every container can reach any other container on any port within the subnet

Manage Infrastructure (standard mode)

External Access

- ❑ The easiest way to access nodes is to ensure that your public ssh key is available to them
- ❑ You can quickly copy your public key to all of the nodes in your Docker Cloud account by running the **authorizedkeys** container



Manage Nodes and Apps

Manage Nodes and Apps (standard mode)

- ❑ These topics cover the traditional, pre-Swarm model for deploying and managing nodes, services, and applications in Docker Cloud

Applications in Docker Cloud

- ❑ Applications in Docker Cloud are usually several Services linked together using the specifications from a **Stackfile** or a Compose file
- ❑ You can also create individual services using the Docker Cloud Services wizard
- ❑ And you can attach Volumes to use as long-lived storage for your services
- ❑ If you are using autobuild and autotest features, you can also see autodeploy to automatically redeploy the application

Docker Machine

Docker Machine Overview

- ❑ Docker Machine is a **tool** that lets you install Docker Engine on virtual hosts, and manage the hosts with `docker-machine` commands.
- ❑ You can use Machine to **create** Docker hosts on your local Mac or Windows box.
- ❑ Also, on your company network, in your data center, or on **cloud** providers like AWS or Digital Ocean.

Docker Machine Overview

- ❑ Using `docker-machine` **commands**, you can:
 - ❑ Start
 - ❑ Inspect
 - ❑ Stop
 - ❑ Restart a managed host
 - ❑ Upgrade the Docker client and daemon
 - ❑ Configure a Docker client to talk to your host.

Setting Machine Environment

- ❑ Point the Machine CLI at a running, managed host, and you can run docker commands directly on that host.
- ❑ For example, run docker-machine env default to point to a host called default.
 - ❑ NOTE: You may have to name the VM specifically, like this:

```
docker-machine env <name>
```

Setting Machine Environment

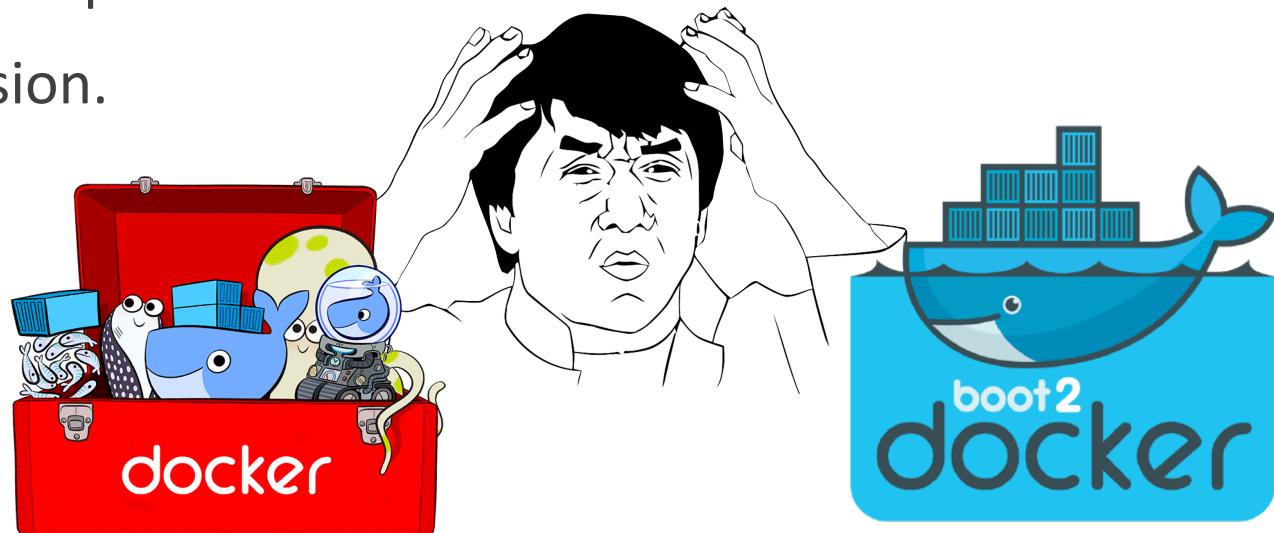
- ❑ Once you follow the on-screen instructions to complete the env setup, you can run docker ps, docker run hello-world, and so forth.

- ❑ Output:

```
export DOCKER_TLS_VERIFY="1"  
  
export DOCKER_HOST="tcp://XXX.XXX.XX.XXX:XXXX"  
  
export DOCKER_CERT_PATH= <PATH>  
  
export DOCKER_MACHINE_NAME= <NAME>  
  
# Run this command to configure your shell:  
  
# eval $(docker-machine env dev)
```

The Good Old Days

- ❑ Machine was the **only** way to run Docker on Mac or Windows previous to Docker v1.12.
- ❑ Docker for Mac & Docker for Windows are available as **native** apps *now* and are the better choice for this use case on newer desktops and laptops.
- ❑ The installers for Docker for Mac and Docker for Windows include Docker Machine, along with Docker Compose.
- ❑ So, no more need for confusion.

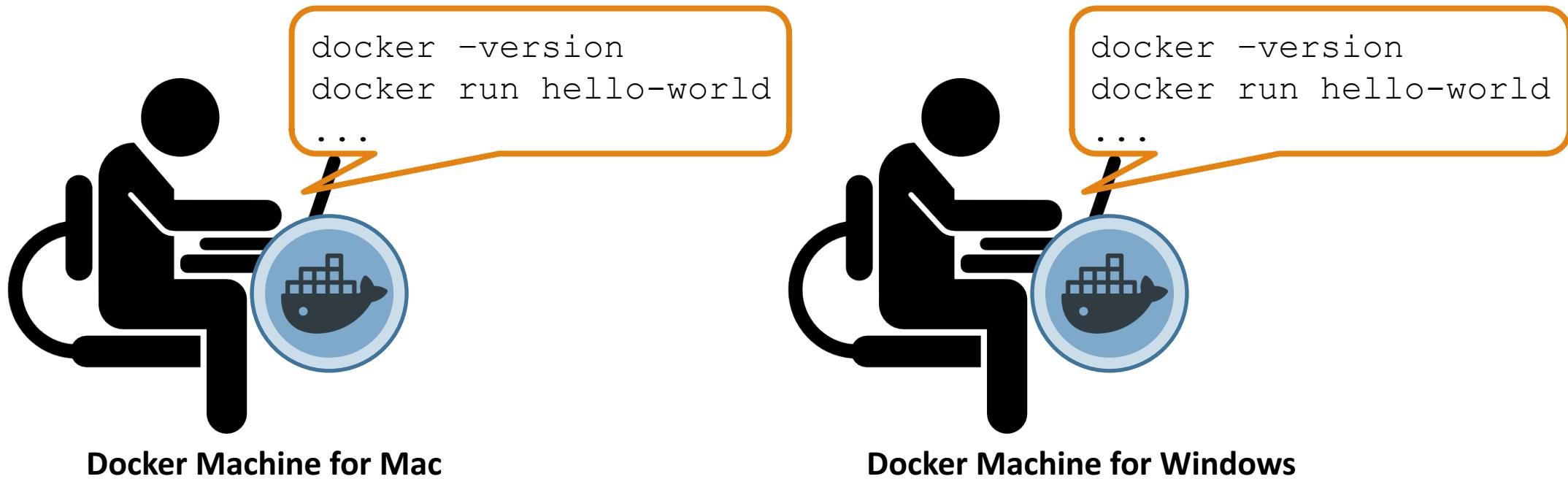


Why Use It, Now?

- ❑ Docker Machine enables you to provision multiple remote Docker hosts on various flavors of Linux.
 - ❑ Additionally, Machine allows you to run Docker on older Mac or Windows systems, as described in the previous topic.
-
- ❑ Basically, Docker Machine has two broad use cases:

Why Use It, Now?

#1 - You own an older desktop system and want to run Docker on Mac or Windows

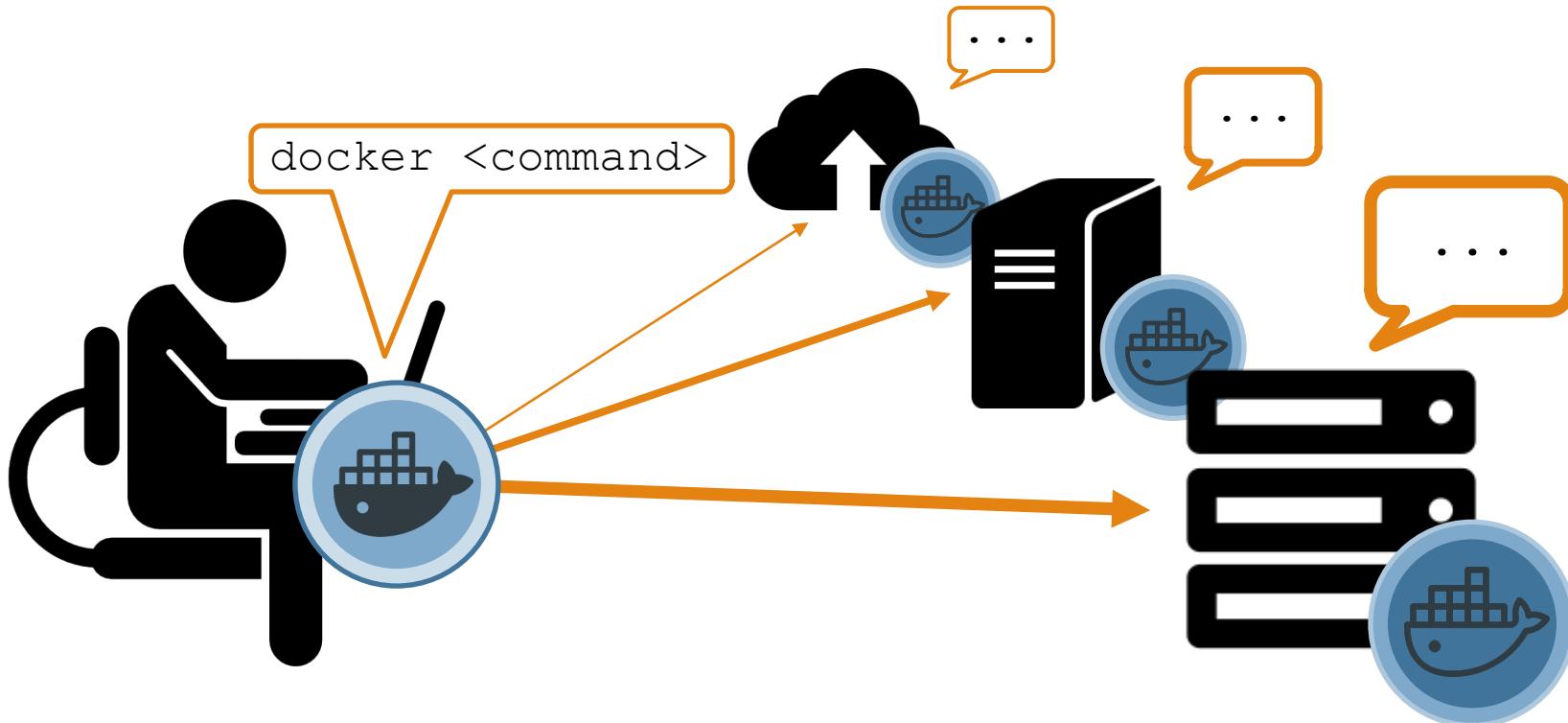


The Older Desktop Option

- ❑ If the computer doesn't meet the requirements for DockerforMac and Docker for Windows apps, then you need Docker Machine in order to run Docker Engine locally.
- ❑ Installing Docker Machine on a Mac or Windows box with the Docker Toolbox installer, provisions a local virtual machine with Docker Engine.
- ❑ This gives you the ability to connect it, and run docker commands.

Why Use It, Now?

#2 - You want to provision Docker hosts on remote systems



Remote Host Option

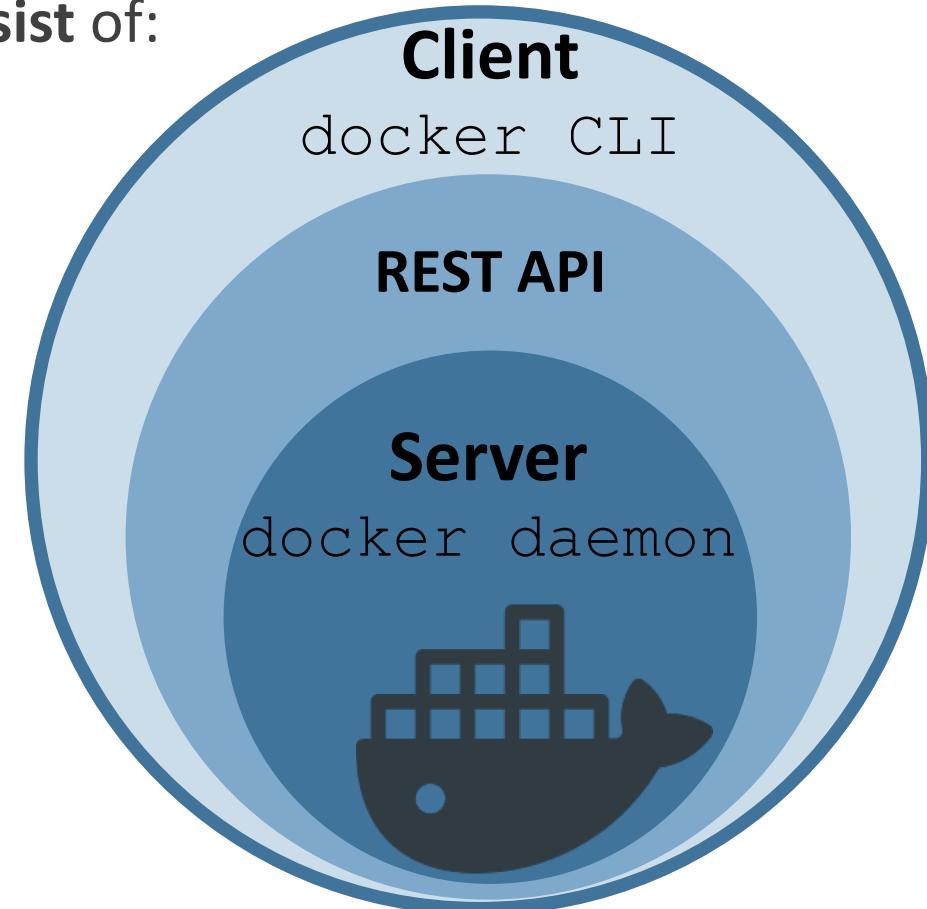
- ❑ Rememeber, Docker **Engine** runs natively on Linux systems.
- ❑ If you have a Linux box as your **primary** system, and want to run Docker commands, all you need to do is download and install Docker Engine.
- ❑ However, if you want an efficient way to provision multiple Docker hosts on a network, in the cloud or even locally, you **need** Docker Machine.

Remote Host Option

- ❑ Whether your primary system is Mac, Windows, or Linux, you can install Docker Machine on it and use `docker-machine` commands to provision and **manage** large numbers of Docker hosts.
- ❑ Machine automatically creates hosts, installs Docker Engine on them, then configures the Docker clients. Each managed host (*machine*) is the **combination** of a Docker host and a configured client.

Docker Engine vs. Docker Machine

- ❑ When people say “*Docker*” they typically mean Docker **Engine**
- ❑ Engine, the client-server application **consist** of:
 - ❑ the Docker **daemon**
 - ❑ a REST **API** that specifies interfaces for interacting with the daemon
 - ❑ a command line interface (**CLI**) client that talks to the daemon (through the REST API wrapper)



Docker Engine vs. Docker Machine

- ❑ Docker Engine accepts `docker` commands from the CLI, such as `docker run <image>`, `docker ps` to list running containers, `docker images` to list images, and so on
- ❑ **Docker Machine** is a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them).
- ❑ Typically, you install Docker Machine on your local system. Docker Machine has its own command line client `docker-machine`
- ❑ You can use Machine to install Docker Engine on one or more virtual systems. These virtual systems can be local (as when you use Machine to install and run Docker Engine in VirtualBox on Mac or Windows) or remote (as when you use Machine to provision Dockerized hosts on cloud providers). The Dockerized hosts themselves can be thought of, and are sometimes referred to as, managed “*machines*”.

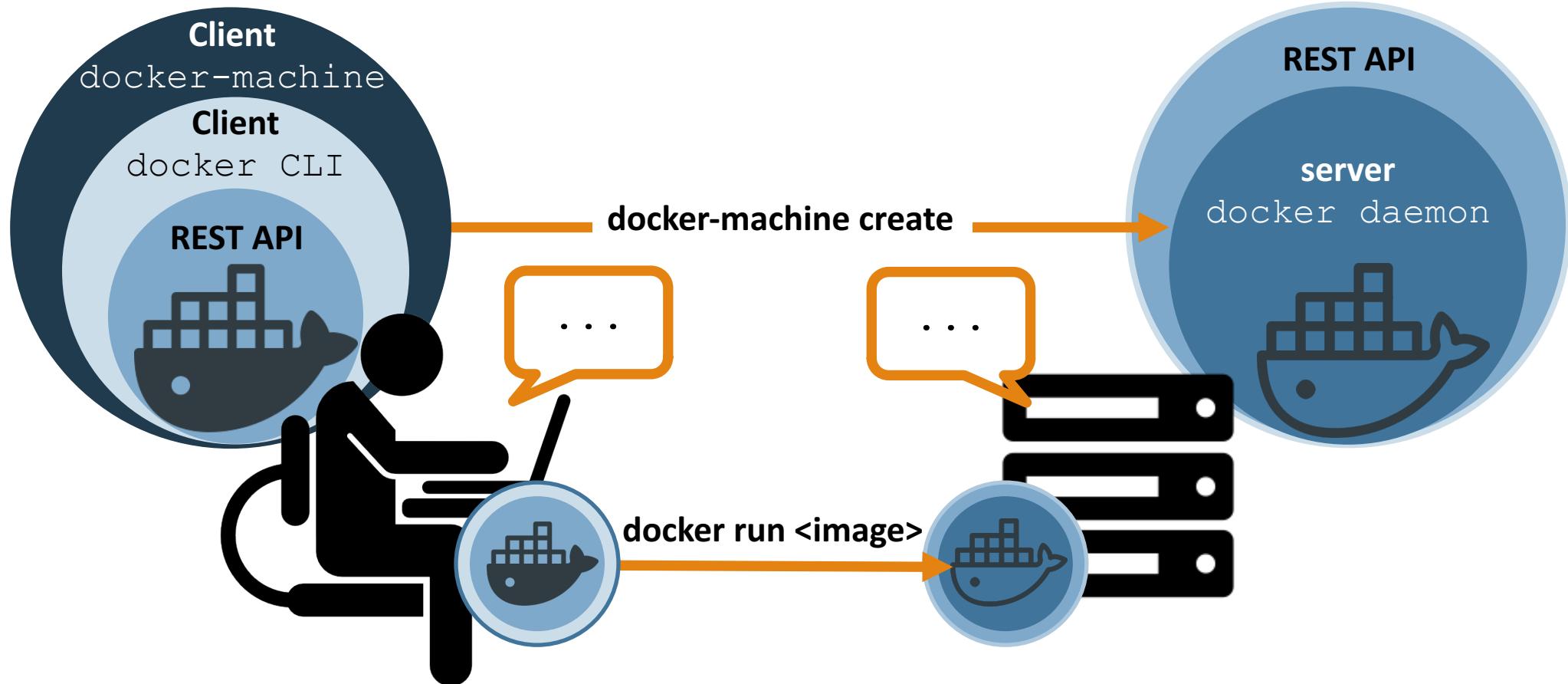
Docker Engine vs. Docker Machine

- ❑ Docker Engine accepts `docker` commands from the CLI, such as `docker run <image>`, `docker ps` to list running containers, `docker images` to list images, and so on
- ❑ **Docker Machine** is a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them).
- ❑ Typically, you install Docker Machine on your local system. Docker Machine has its own command line client `docker-machine`

Docker Engine vs. Docker Machine

- ❑ You can use Machine to install Docker Engine on one or more virtual systems.
- ❑ Virtual systems can be local (as when you use Machine to install and run Docker Engine in VirtualBox on Mac or Windows)
- ❑ Or, they can be remote (as when you use Machine to provision Dockerized hosts on cloud providers).
- ❑ The Dockerized hosts themselves can be thought of, and are sometimes referred to as, managed “*machines*”.

Remote Machine Host Diagram



Getting Started Using Local VM

- ❑ Let's take a look at using `docker-machine` to create, use and manage a Docker host inside of a **local** virtual machine.



Docker Machine Rust

- ❑ With the advent of Docker for Mac and Docker for Windows as replacements for Docker Toolbox, we recommend that you use these for your primary Docker workflows.
- ❑ You can use these applications to run Docker natively on your local system without using Docker Machine at all.

Docker Machine Rust

- ❑ However, if you want to create *multiple* local machines, you still need Docker Machine to create and manage machines for multi-node experimentation.
- ❑ Both Docker for Mac and Docker for Windows include the newest version of Docker Machine, so when you install either of these, you get `docker-machine`.

Prerequisite Information

- ❑ Keep the following considerations in mind when using Machine to create local VMs.
 - ❑ **Docker for Windows** - You can use `docker-machine create` with the `hyperv` driver to create additional local machines.
 - ❑ **Docker for Mac** - You can use `docker-machine create` with the `virtualbox` driver to create additional local machines.

Windows Prerequisite Information

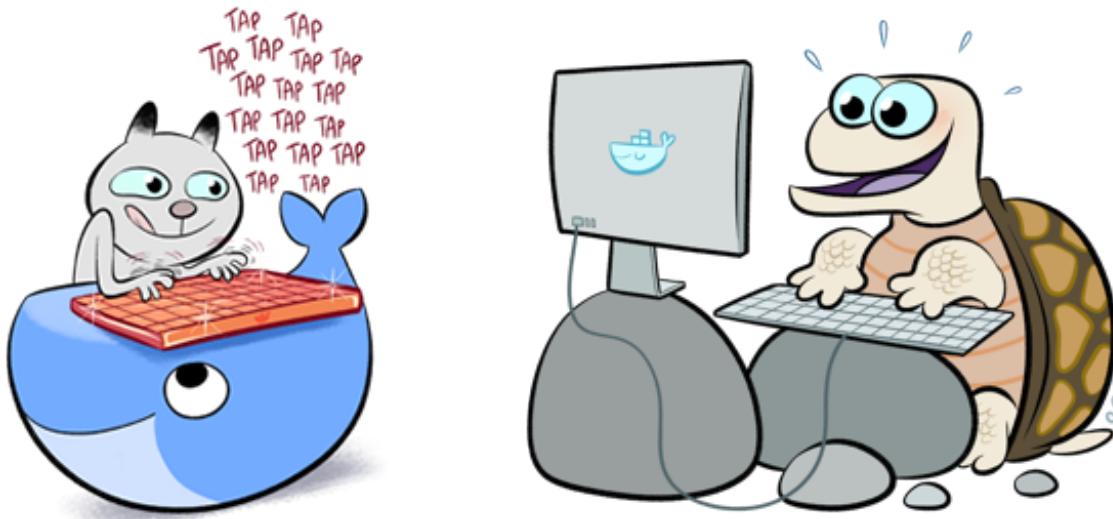
Docker for Windows

- Docker for Windows uses Microsoft Hyper-V for virtualization, and Hyper-V is not compatible with Oracle VirtualBox.
- Therefore, you cannot run the two solutions simultaneously.
- But you can still use `docker-machine` to create more local VMs by using the Microsoft Hyper-V driver.

Mac Prerequisite Information

❑ Docker for Mac

- ❑ Docker for Mac uses HyperKit, a lightweight macOS virtualization solution built on top of the Hypervisor.framework in macOS 10.10 Yosemite and higher.
- ❑ Currently, there is no docker-machine create driver for HyperKit, so you will use virtualbox driver to create local machines.
- ❑ You can run both HyperKit and Oracle VirtualBox on the same system.



Running Containers

- ❑ To run a Docker container, you:
 - ❑ Create a new (or start an existing) Docker virtual machine.
 - ❑ Switch your environment to your new VM.
 - ❑ Use the docker client to create, load, and manage containers.
- ❑ Once you create a machine, you can reuse it as often as you like.
- ❑ Like any VirtualBox VM, it maintains its configuration between uses.

- ❑ The examples here show how to create and start a machine, run Docker commands, and work with containers.

Machine Containers Example

- ❑ Let's look at how to create a new container, using Docker Machine.
- ❑ First, open a command shell or terminal window.
- ❑ Use `docker-machine ls` to list available machines:

```
docker-machine ls
```

- ❑ Output (No machines have been created):

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS

Machine Containers Example

- To create a Machine, we'd need to start by following the steps below:
 - **Run** docker-machine create <command>
 - **Pass the appropriate driver to the --driver flag** and provide a machine name.
 - If this is your first machine, **name** it default as shown in the example.
 - If you already have a “default” machine, choose **another** name for this new machine.
 - If you are using Docker for **Mac**, use `virtualbox` as the driver, as shown in this example.
 - On Docker for **Windows** systems that support Hyper-V, use the `hyperv` driver

Creating a New Machine

- ❑ So, our command should look something like this:

```
docker-machine create --driver virtualbox default
```

- ❑ Output:

```
Running pre-create checks...
```

```
Creating machine...
```

```
...
```

- ❑ This downloads a lightweight Linux distribution boot2docker, with the Docker daemon installed, plus creates and starts a VirtualBox.

Print Newly Created Machine

- ❑ To list available machines, we want to use `docker-machine ls` again.
 - ❑ This time we should find our container listed.
-
- ❑ Output:

```
NAME      ACTIVE   DRIVER      STATE      URL      ...
default    *        virtualbox  Running    tcp://XXX.XXX.XX.XXX:XXXX ...
```

Get Docker Machine Env Variables

- ❑ Let's look at the environment variable for your new VM.
- ❑ To do that, we will use the following command:

```
docker-machine env <machine-name>
```

- ❑ Output:

```
export DOCKER_TLS_VERIFY="..."  
export DOCKER_HOST="..."  
export DOCKER_CERT_PATH="..."  
export DOCKER_MACHINE_NAME="..."  
  
# Run this command to configure your shell:  
# eval "$(docker-machine env default)"
```

Connecting Env Settings

- ❑ Now, we need to connect our shell to the new machine.
- ❑ We'll use the following command, which can be found in the output:

```
eval "$(docker-machine env default)"
```

- ❑ Okay! Now, our environment variables are set for the current shell that the Docker client will read which specify the TLS settings.
- ❑ You need to do this each time you open a new shell or restart your machine.
- ❑ You can now run Docker commands on this host.

Experiment With Machine Containers

- ❑ Run a container with `docker run` to verify your set up.
- ❑ Use `docker run` to download and run `busybox` with a simple `echo` command, like this:

```
docker run busybox echo hello world
```

- ❑ Output:

```
Unable to find image 'busybox' locally  
Pulling repository busybox  
e72ac664f4f0: Download complete  
511136ea3c5a: Download complete  
df7546f9f060: Download complete  
e433a6c5b276: Download complete  
hello world
```

Introducing Machine Cloud Drivers

- ❑ When you install Docker Machine, you get a set of drivers for various cloud providers.
- ❑ Also available, Docker Machine driver plugins for use with other cloud platforms are available from 3rd party contributors.
 - ❑ NOTE: These are use-at-your-own-risk plugins, not maintained by or formally associated with Docker.



Exploring the Cloud

- ❑ Docker Machine driver plugins are available for many cloud platforms, so you can use Machine to provision cloud hosts.
- ❑ When you use Docker Machine for provisioning, you create cloud hosts with Docker Engine installed on them.
- ❑ We'll need to create an account with the cloud provider.

Exploring the Cloud

- ❑ We will provide account verification, security credentials, and configuration options for the providers as flags to `docker-machine create`.
- ❑ The flags are unique for each cloud-specific driver. For instance, to pass a Digital Ocean access token you use the `--digitalocean-access-token` flag.
- ❑ Let's look at some examples on the next slide...

Digital Ocean Example

- ❑ For Digital Ocean, this command creates a Droplet (cloud host) called “docker-sandbox”:

```
docker-machine create --driver digitalocean  
--digitalocean-access-token xxx docker-sandbox
```



Amazon Web Services (AWS) Example

- ❑ For AWS, this command creates an instance called “aws-sandbox” in EC2:

```
docker-machine create --driver amazonec2 --amazonec2-access-key  
AKI***** --amazonec2-secret-key 8T93C***** aws-sandbox
```



Docker Machine Swarm

- ❑ Docker Machine can also provision Docker Swarm clusters.
- ❑ This can be used with any driver and will be secured with TLS.

Start and Stop

- ❑ If you are finished using a host for the time being, you can stop it with `docker-machine stop` and later start it again with `docker-machine start`.
- ❑ Stop the machine, like this:

```
docker-machine stop <machine-name>
```

- ❑ And we would start it back up, like this:

```
docker-machine start <machine-name>
```

Azure Support for Docker

Why Docker for Azure?

- ❑ The Docker for Azure project was created and is being actively developed to ensure that Docker users can enjoy a fantastic out-of-the-box experience on Azure
- ❑ It is now generally available and can now be used by everyone
- ❑ As an informed user, you might be curious to know what this project offers you for running your development, staging, or production workloads

Native for Docker

- ❑ Docker for Azure provides a Docker-native solution that avoids operational complexity and adding unneeded additional APIs to the Docker stack
- ❑ Docker for Azure allows you to interact with Docker instead of distracting you with the need to navigate extra layers on top of Docker
- ❑ The skills that you've already learned, using Docker automatically carry over to using Docker on Azure

Skip the Boilerplate and Maintenance

- ❑ Docker for Azure bootstraps all of the recommended infrastructure to start using Docker on Azure automatically
- ❑ You don't need to worry about rolling your own instances, security groups, or load balancers when using Docker for Azure
- ❑ Likewise, setting up and using Docker swarm mode functionality for container orchestration is managed across the cluster's lifecycle when you use Docker for Azure. Docker has already coordinated the various bits of automation you would otherwise be gluing together on your own to bootstrap Docker swarm mode on these platforms. When the cluster is finished booting, you can jump right in and start running docker service commands.
- ❑ We also provide a prescriptive upgrade path that helps users upgrade between various versions of Docker in a smooth and automatic way. Instead of experiencing "maintenance dread" as you ponder your future responsibilities upgrading the software you are using, you can easily upgrade to new versions when they are released.

Skip the Boilerplate and Maintenance

- ❑ Docker for Azure bootstraps all of the recommended infrastructure to start using Docker on Azure automatically
- ❑ You don't need to worry about rolling your own instances, security groups, or load balancers when using Docker for Azure
- ❑ Likewise, setting up and using Docker swarm mode functionality for container orchestration is managed across the cluster's
- ❑ When the cluster is finished booting, you can jump right in and start running docker service commands
- ❑ Also provides a prescriptive upgrade path that helps users upgrade between various versions of Docker in a smooth and automatic way

Hands-on Exercise(s)

End of Chapter

Docker Swarm

CONNECTING AND MANAGING MULTIPLE CONTAINERS

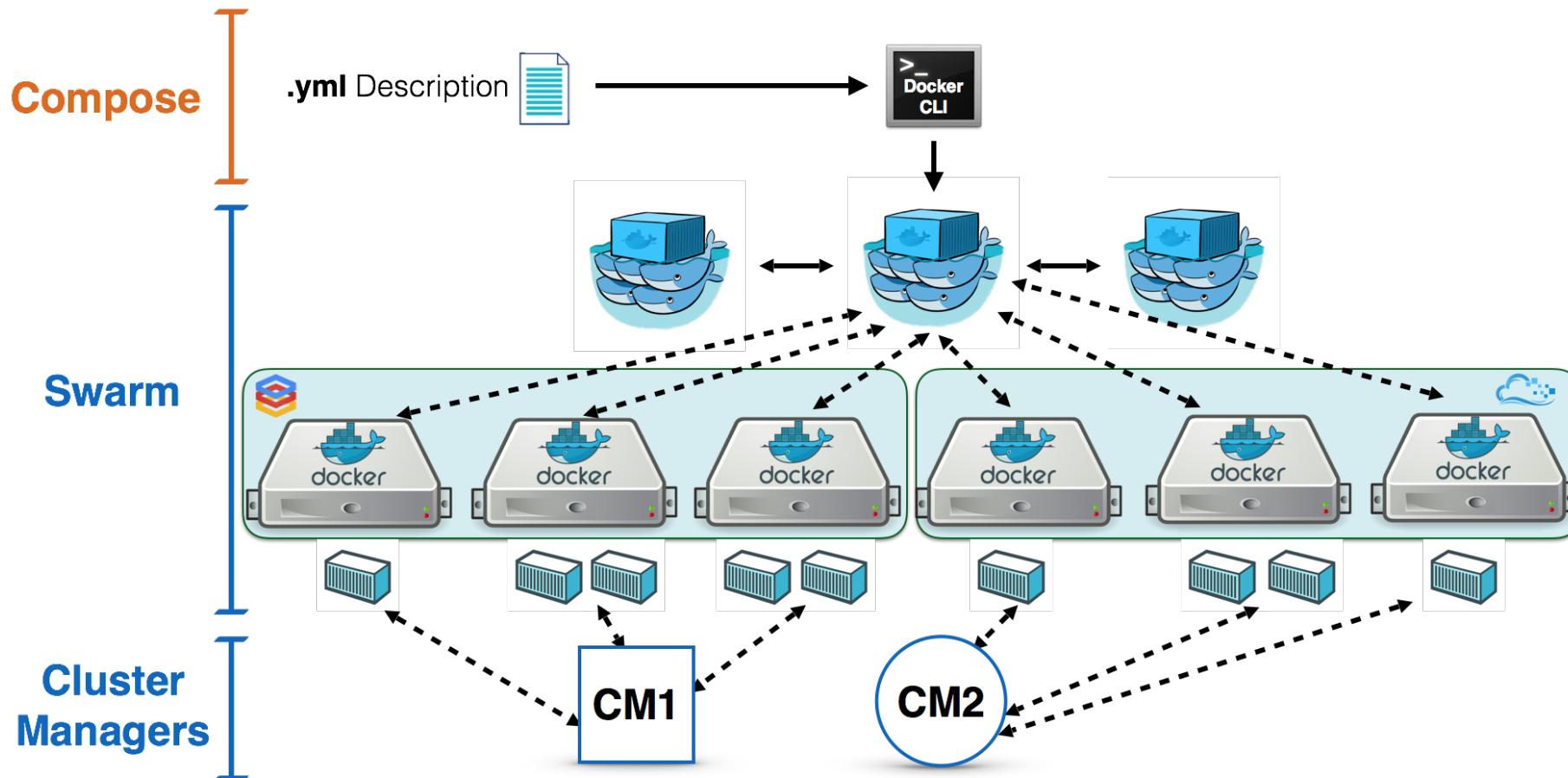
Overview

Recap Docker

- ❑ Containers run on **single** Docker host
- ❑ Containers are **ephemeral**
- ❑ Containers can have external persistence
- ❑ Containers do not contain
- ❑ Operating system **matters**

Swarm Overview

- Describe swarm in a sentence of two, here.



Initialize Swarm Process

- ❑ Turn single host Docker host into a Multi-host with Docker Swarm Mode.
- ❑ All containers are only **deployed** onto the engine.
- ❑ Swarm Mode turns it into a multi-host **cluster-aware** engine.
- ❑ The first node to initialize the Swarm Mode becomes the **manager**.
- ❑ As new nodes join the cluster, they can **adjust** their roles between managers or workers.
- ❑ You **should** run 3-5 managers in a production environment to ensure high availability.

Create Swarm Mode Cluster

- ❑ Swarm Mode is built **into** the Docker CLI. You can find an overview the possibility commands via:

```
docker swarm --help
```

- ❑ The most important one is how to **initialize** Swarm Mode. Initialization is done via *init*, like this:

```
docker swarm init
```

- ❑ After running the command, the Docker Engine knows how to work with a cluster and becomes the **manager**.
- ❑ The results of an initialization is a **token** used to add additional nodes in a secure fashion.

Swarm Init Output

- Once you initialize Docker Swarm, the token output will look something like this:

```
WSwarm initialized: current node (gpnuxrri0tzaszblvhdrwqsh) is  
now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join \  
--token SWMTKN-1-  
1xffnpg61r5wgel28ehuh8o3ip5z2iiwmp1bemnksfho5akdqy-  
3zkn8ssrkcnjei34f5rmd4n3q \  
172.17.0.65:2377
```

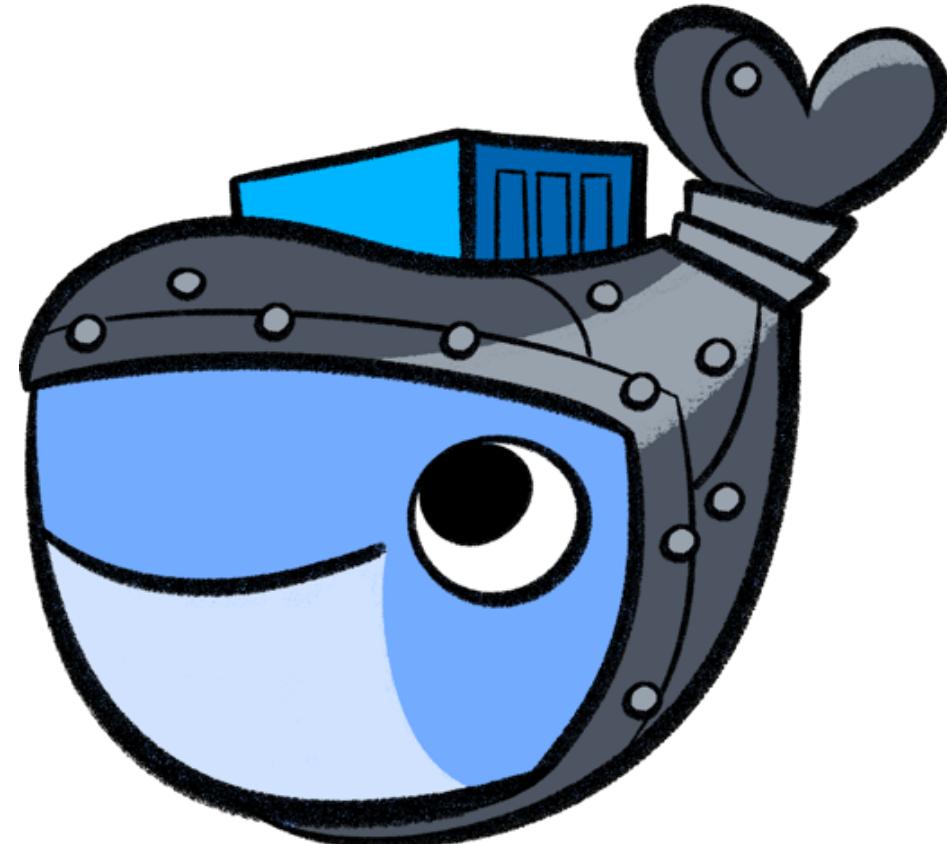
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

Creating a Cluster

- ❑ With Swarm Mode enabled, it is possible to **add** additional nodes and issues commands across all of them.
- ❑ If nodes happen to disappear, for example because of a **crash**, the containers which were running on those hosts will be automatically rescheduled onto other available nodes.
- ❑ The rescheduling ensures you do not lose capacity and provides **high-availability**.
- ❑ On each additional node, you wish to add to the cluster, use the Docker CLI to **join** the existing group.
- ❑ Joining is done by pointing the other host to a **current manager** of the cluster.

Creating a Cluster

- ❑ Docker now uses an **additional** port, 2377, for managing the Swarm.
- ❑ The port should be blocked from public access and only accessed by **trusted** users and nodes.
- ❑ We recommend using VPNs or private networks to **secure** access.



Join Cluster

- ❑ The first task is to **obtain** the token required to add a node to the cluster.
- ❑ For demonstration purposes, we'll ask the manager what the **token** is via *swarm join-token*.

```
token=$(docker -H 172.17.0.65:2345 swarm join-token -q  
worker) && echo $token
```

- ❑ NOTE: In production, this token should be stored securely and only accessible by trusted individuals.

Join Cluster

- ❑ On the second host, **join** the cluster by requesting access via the manager.
- ❑ The **token** is provided as an additional parameter, like this:

```
docker swarm join 172.17.0.65:2377 --token $token
```

Join Cluster

- ❑ By **default**, the manager will automatically accept new nodes being added to the cluster.
- ❑ You can **view** all nodes in the cluster using:

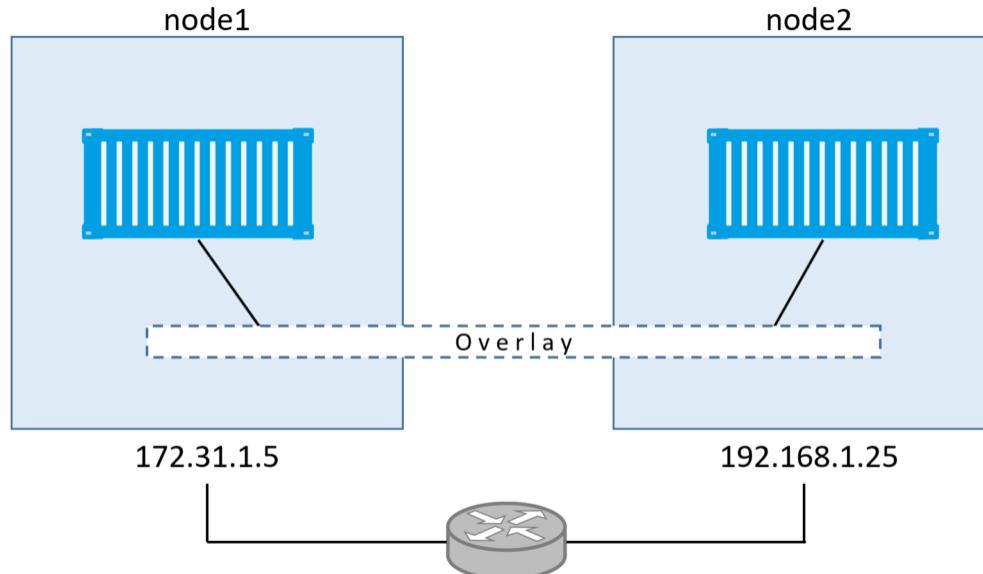
```
docker node ls
```

- ❑ Output:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER
STATUS5re1humv246m6clujn0zng9jy	docker	Ready		
Activerbavg04vwiijyzzjwdjoc3io2 *	docker	Ready	Active	Leader

Overlay Network

- ❑ Swarm Mode also introduces another **networking** model.
- ❑ In **previous** versions, Docker required the use of an external key-value store, such as Consul, to ensure consistency across the network.
- ❑ The need for consensus and KV has now been incorporated **internally** into Docker and no longer depends on external services.



Overlay Network

- ❑ The improved networking approach follows the **same** syntax as previously.
- ❑ The **overlay** network is used to enable containers on different hosts to communicate.
- ❑ Under the covers, this is a Virtual Extensible LAN (**VXLAN**), designed for large scale cloud based deployments.

Overlay Network

- The following command will create a new overlay network called **skynet**:

```
docker network create -d overlay skynet
```

- All containers **registered** to this network can communicate with each other, regardless of which node they are deployed onto.

Deploy Service

- ❑ By default, Docker uses a **spread** replication model for deciding which containers should run on which hosts.
- ❑ The spread approach ensures that containers are deployed across the cluster **evenly**.
- ❑ If one of the nodes are **removed** from the cluster, the containers running are spread across the other available nodes.



Deploy Service

- ❑ The new **Services** concept is used to run containers across the cluster.
- ❑ This is a **higher-level** concept than containers.
- ❑ A service allows you to define how applications should be deployed at **scale**.
- ❑ By updating the service, Docker updates the container required in a **managed** way.

Deployment

- ❑ As an **example**, let's deploy the Docker Image katacoda/docker-http-server.
- ❑ We will give it a friendly **name** like, http .
- ❑ And we will **attach** it to the newly created skynet **network**.
- ❑ For ensuring replication and availability, we will run two instances of **replicas**, of the container across our cluster.

Deployment

- ❑ Finally, we will load balance these two containers **together** on port 80.
 - ❑ Sending an HTTP request to any of the nodes in the cluster will process the request by **one** of the containers within the cluster.
-
- ❑ NOTE: The node which accepts the request might not be the node where the container responses. Instead, Docker load-balances requests across all available containers.
 - ❑ So, what should our **code** look like?

Deployment

□ Here it is:

```
docker service create --name http --network skynet  
--replicas 2 -p 80:80 katacoda/docker-http-server
```

Named it “http”

Assigned it to “skynet” network

Created 2 replicas

Exposed port 80

Indicated which Image to use

Deployment Verification

- You can **view** the services running on the cluster using the CLI command

```
docker service ls
```

- Output:

ID	NAME	MODE	REPLICAS	IMAGE
414yr9hv663w	http	replicated	2/2	katacoda/docker-http-server:latest

Deployment Verification

- ❑ As containers are started you will see them using the `ps` command. You should see one instance of the container on **each** host.
- ❑ **Listing** containers on the first host using `docker ps` will produce:

CONTAINER ID	IMAGE...
<host-1-id>	Image:name

- ❑ Listing containers on the **second** host using `docker ps` produces:

CONTAINER ID	IMAGE...
<host-2-id>	Image:name

Deployment Verification

- ❑ If we issue an HTTP request to the **public** port, it will be processed by the two containers:

```
curl <localhost>
```

- ❑ Output:

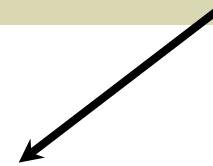
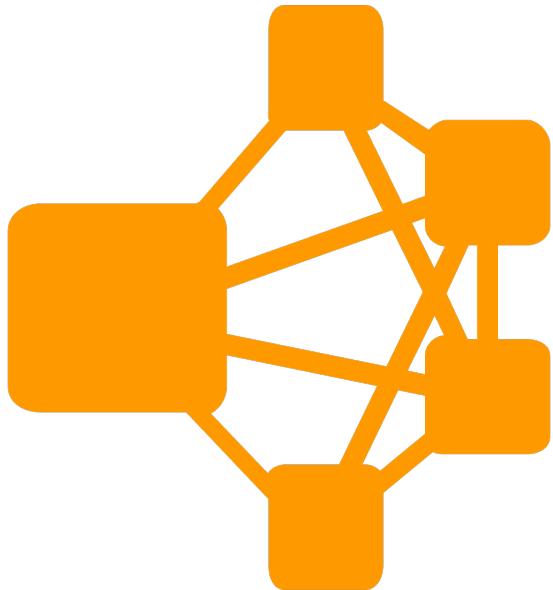
```
<h1>This request was processed by host: "host-1-id"</h1>
```

- ❑ If we issue the same command again, what do you think the results will be?

Deployment Verification

- ❑ Everything happens just like before, except this request was processed by host 2's container instance:

```
<h1>This request was processed by host: "host-2-id"</h1>
```



Notice which container id is given. Again, using `ps` we could verify this.

What if we do it again?

Health Inspection

- ❑ The Service concept allows you to inspect the health and state of your cluster and the running applications.
- ❑ You can view the list of all the tasks associated with a service across the cluster:

```
docker service ps http
```

- ❑ NOTE: In this case, each task is a container.
- ❑ Output:

ID	NAME	IMAGE	NODE
<service-id-1>	http.1	katacoda/docker-http-server:latest	docker
<service-id-2>	http.2	katacoda/docker-http-server:latest	docker

Health Inspection

- You can view more details and the configuration of a service by using:

```
docker service inspect --pretty http
```

- Output:

```
ID: 9b60zdcritvo1sskwsta44f
Name: http
Service Mode: Replicated
Replicas: 2
...
```

Health Inspection

- On each node, you can ask what tasks it is currently running.

```
docker node ps self
```

- NOTE: Self refers to the manager node Leader:

- Output:

ID	NAME	IMAGE	NODE ...
<service-id-1>	http.2	katacoda/docker-http-server:latest	docker ...

Health Inspection

- Using the ID of a node you can query individual hosts

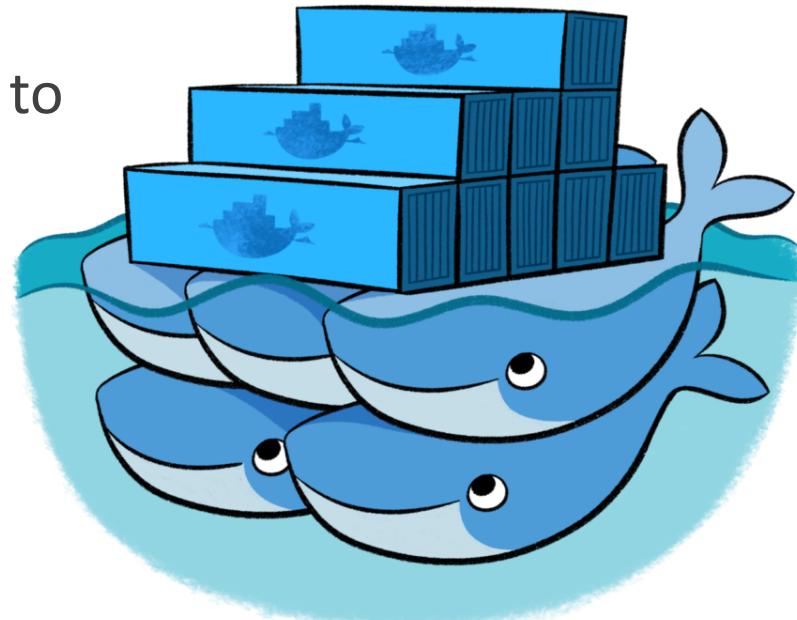
```
docker node ps $(docker node ls -q | head -n1)
```

- Output:

ID	NAME	IMAGE	NODE ...
<service-id-2>	http.1	katacoda/docker-http-server:latest	docker ...

Scale Service

- ❑ A Service also allows us to scale how many instances of a task is running across the cluster.
- ❑ As it understands how to launch containers and which containers are running, it can easily start, or remove, containers as required.
- ❑ At the moment the scaling is manual.
 - ❑ However, the API could be hooked up to an external system such as a metrics dashboard.



Scale Service

- ❑ In our example, we had two load-balanced containers running, processing our requests `curl docker`.
- ❑ Command below would scale our `http` service to be run across five containers:

```
docker service scale http=5
```

- ❑ On each host, you will see additional nodes being started.
- ❑ The load balancer will automatically be updated.
- ❑ Requests will now be processed across the new containers.

Result Summary

- ❑ The result of this scenario is a two-node Swarm cluster which can run load-balanced containers that can be scaled up and down.

Hands-on Exercise(s)

End of Chapter
