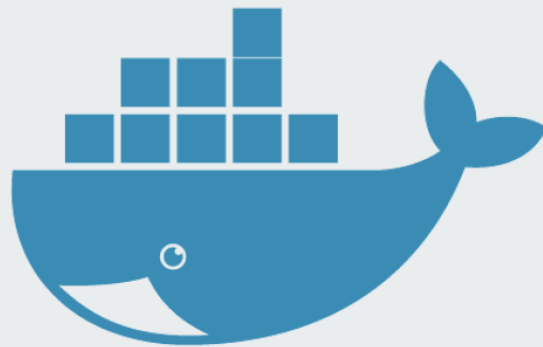


Big Picture Docker

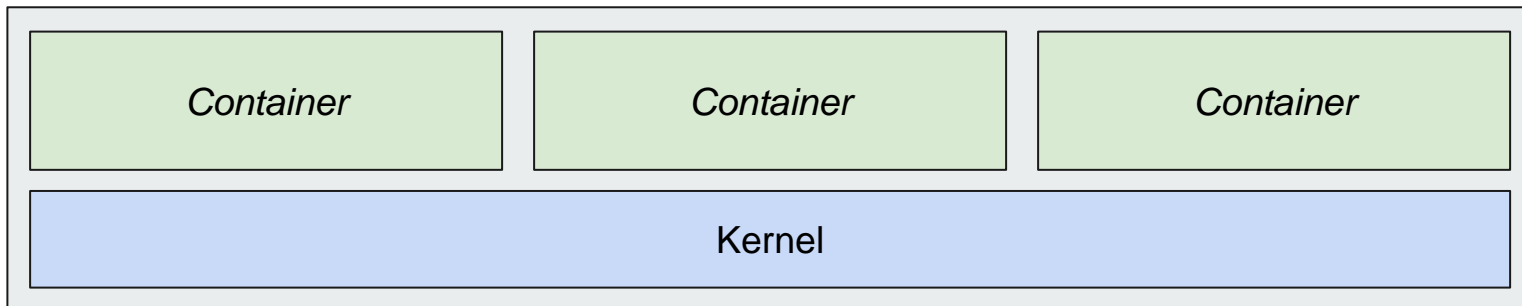
What, Why, Who, & Features





What is Docker?

- Docker is a platform that lets you package, develop, and run applications in containers.
- A container is a virtual environment on top of the OS kernel to capture all of its software - libraries, dependencies, etc.





Why does Docker matter?

- A more lightweight approach than virtual machines to isolated coding and project environments.
- Portability to the major architectures and operating systems.
- Helps achieve continuous integration and deployment for development operations.




Who Does Docker affect most?

- Developers and software engineers.
- Operators and devops engineers.
- Startups and enterprises.

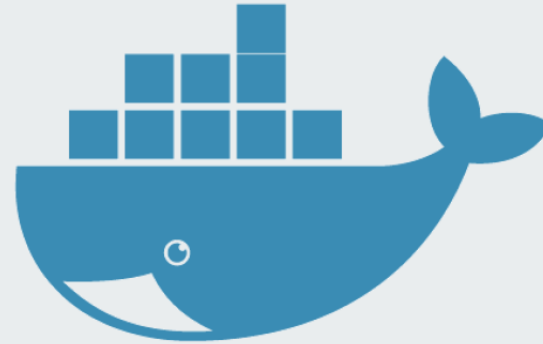


Main features

- Create containers and images.
- Docker-compose for multicontainer applications.
- Docker swarm to utilize multiple machines running Docker.



Docker Containers and the Docker Engine





What is a Docker Container?

A Docker container is a loosely isolated environment running within a host machine's kernel that allows us to run application-specific code.



The Kernel

- The kernel is the software at the core of an operating system, with complete control.
- The CPU is the core circuitry which executes program instructions.
- Docker runs on top of original machine's kernel - making it the *host machine*.

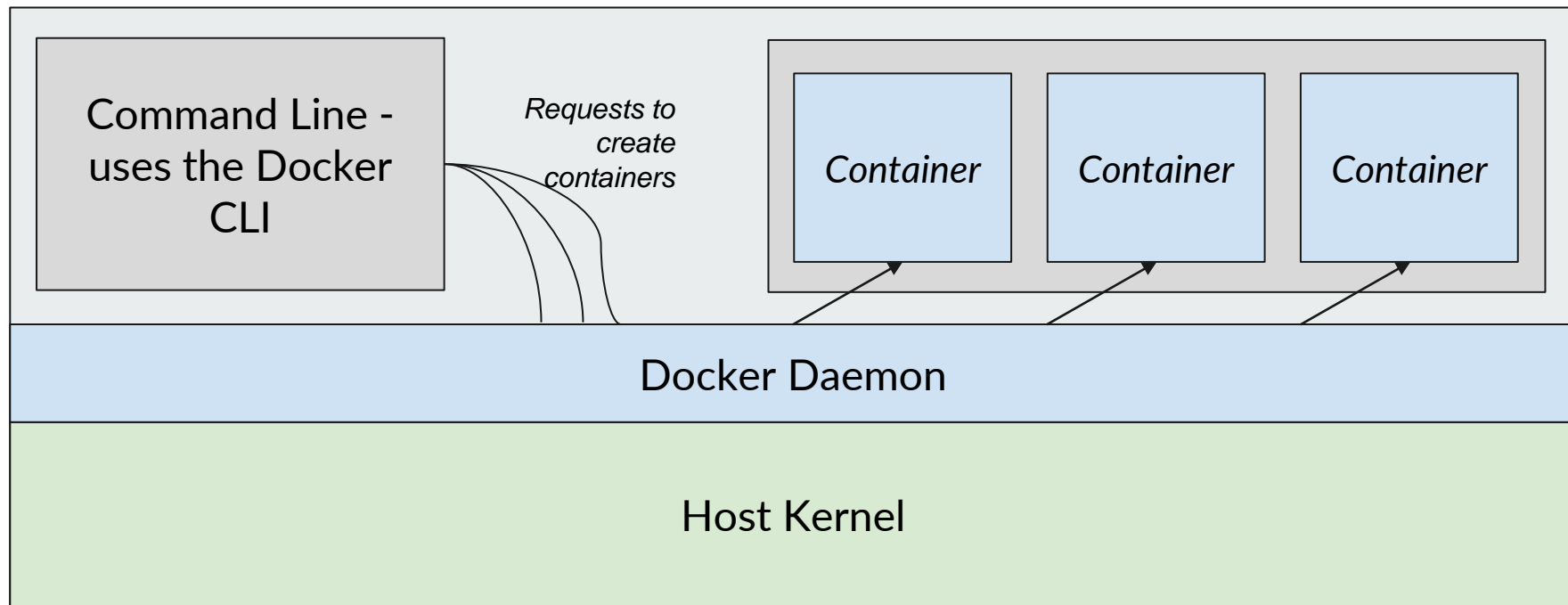


The Docker Engine

- Consists of the Docker server, an API, and command line interface.
- The server is also called the Docker daemon.
 - daemon - background processes on an operating system.
- Docker daemon is like a construction team on the host machine.



The Docker Engine on an Operating System





Docker Container Environments

- The processes of one container cannot affect the processes of another.
- A container has limits on resource usage, like the CPU and memory.
- Application-specific code.



What is a Docker Container?

A Docker container is a loosely isolated environment running within a host machine's kernel that allows us to run application-specific code.



Why are containers useful?

- Portability to multiple operating system environments.
- Less time setting up, more time coding.
- Development, continuous integration, deployment environments.



Containers vs. Virtual Machines

- Virtual machines abstract an entire computer system.

Run a 2GB image of software

Virtual Machines

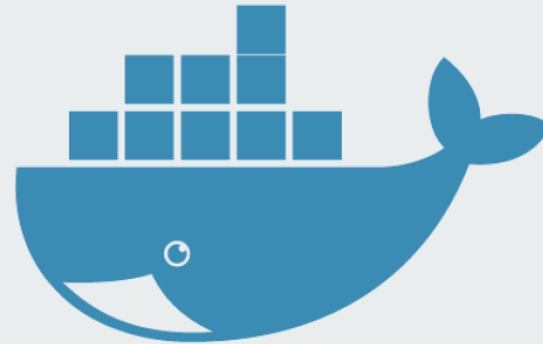
- Require 2GB to run the image
- 5VMs x 2GB = 10GB

Containers

- Use 2GB to run the image.
- $2G + 0.001G * N^n$ Containers



Docker Images





Docker Images

- are “ready-only templates with instructions for creating a Docker container.”
- define the container code, libraries, environment variables, configuration files, and more.



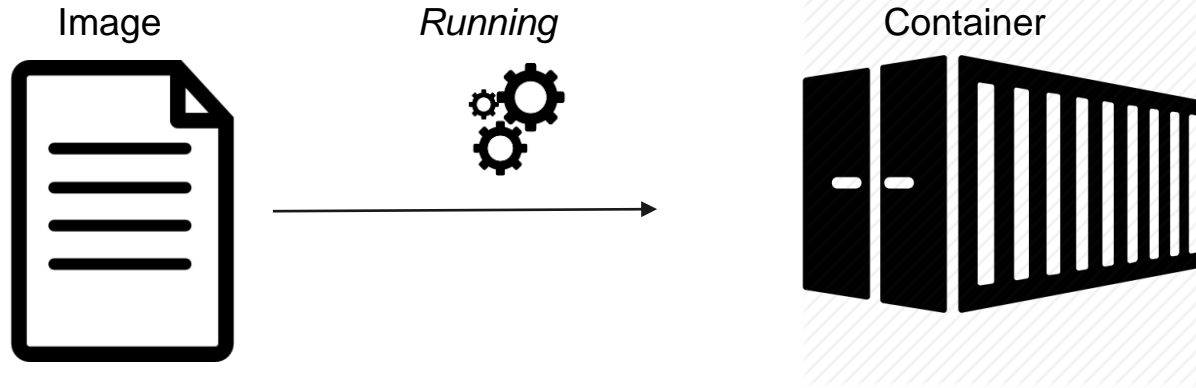
WEINX



Dockerfile

- Outlines instructions for how an image will create a container.

Image to Container Relationship





Dockerhub

- Collects images from Docker users and hosts them as online repositories.

-- Run a Ubuntu Container and a Container Shell --

`We created our first Docker container, and ran a shell within it.
Below are the main commands that we used.

Search for the ubuntu image on the command line with Docker:

- `docker search ubuntu`

Create an interactive ubuntu container, and run bash. Name it foo:

- `docker create -it --name=foo ubuntu bash.

Start the created container:

- `docker start foo`

Then attach to the container's shell:

- `docker attach foo`

List out running containers:

- `docker container ls`

List out containers (whether or not they're running)

- `docker container ls -a`

-- Run a Ubuntu Container Continued --

Containers can be created, run, started and attached to with one step using `docker run.`

We should also remove containers when we finish using them.

Below are the main commands we went over:

Create, start, run, and attach to a container in one step:

- `docker run --name=bar -it ubuntu bash`

Stop a container:

- `docker stop foo`

Remove a container:

- `docker rm foo`

Force remove a container (stop and remove in one step):

- `docker rm -f foo`

Docker Containers and Images | Summary and Commands

Docker Containers and Images | Summary and Commands

Nice work on completing the first section of this course on Docker containers and images. In this section, we learned plenty of important concepts.

We got a grasp of how Docker works, from the running docker engine, the creation of docker images, and the execution of running containers themselves. We learned about the great engineering benefit of isolated environments through containers. Plus we also got a taste of some of the great engineering ideas of Docker, such as caching previous images to optimize performance, and sharing common files across containers.

Moving on, we'll explore Docker images more deeply. We'll create our own customized container images, and dive into more advanced features.

In the meantime, here's a summary of the commands we've used thus far:

Docker Containers and Images

Docker Containers

Create an interactive terminal container with a name, an image, and a default command:

- Usage: `docker create -it --name=<name> <image> <command>`
- Example: `docker create -it --name=foo ubuntu bash`

List all running containers:

- `docker container ls`
- (list all containers, running or not): `docker container ls -a`

Start a docker container:

- Usage: docker start <container name or id>
- Example: docker start foo

Attach to a docker container:

- Usage: docker attach <container name or id>
- Example: docker attach foo

Remove a container:

- Usage: docker rm <container name or id>
- Example: docker rm foo
- Force remove: docker rm foo -f

Run a new container:

- Usage: docker run <image> <command>
- Example with options: docker run --name=bar -it ubuntu bash

Remove all containers:

- docker container ls -aq | xargs docker container rm

Docker Images**Remove all images:**

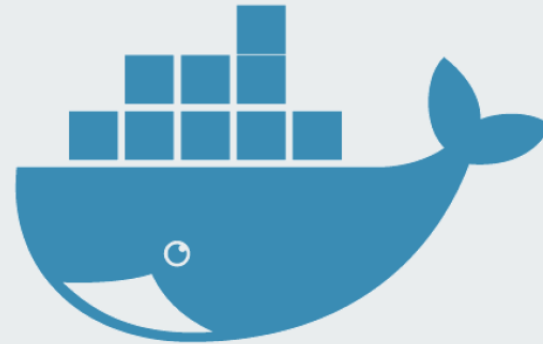
- docker image ls -aq | xargs docker rmi -f

Search for a docker image:

- Usage: docker search <image>
- Example: docker search ubuntu



Docker Images and the Dockerfile





Docker Images

- A “ready-only template with instructions for creating a Docker container.”



Dockerfile

- A text document with various commands that assemble to build a container.
- `$ docker create image... → FROM`
- `$ docker create image command → CMD`
- COPY, RUN, and more...

-- Build a File Server Image --

We build our first Docker image with a custom Dockerfile.

This image used `serve.js`, a `node.js` module used to serve file directories, and static html pages.

Below are the steps we used to create the serve image.

- Create a ``serve`` directory with an inner ``display`` directory.
- Place contents to be served into the ``display`` directory.
- Create a Dockerfile at the root of the project with the following content:
<https://github.com/15Dkatz/docker-guides/blob/master/serve/Dockerfile>
- Build the image: ``docker build . -t docker/serve``
- Run a container with the image, and map the host's 3001 port to the container's 5000 port:
``docker run --name=serve -p=3001:5000 docker/serve``
- visit <http://localhost:5000> in a browser.

-- Build an express.js Image --

We built a new image, using express.js to build a containerized web server. Here is the project we created:

<https://github.com/15Dkatz/docker-guides/tree/master/express>

After cloning the repository, build and run the container, and view it:

- `docker build . -t docker/express`
- `docker run --name=express -d -p 3002:80 docker/express`
- Go to <http://localhost:3002>

While building the express container, we covered a few important concepts. Here are some extra notes:

- In order to make the container accessible, we hosted it on `0.0.0.0`, allow external connections.
- We also used the recommended exec form for the CMD option of the Dockerfile:
CMD ["node", "server.js"]

-- Layered Image Cache and the .dockerignore file --

We went over the layered image cache and the .dockerignore file.

The layered image cache with Docker helps optimize the build process. By caching each phase in the image build, Docker only reruns steps that have actually changed upon subsequent builds.

With the .dockerignore file, we can tell Docker to skip over certain files when copying over the files needed to create images for containers. This is useful when we have large directories like a .git/ or node_modules/ folder that we don't want to send to the container's file system.

-- Build a PHP Image --

We built a php Docker image. We also learned about the importance of the ``EXPOSE 80`` line in the Dockerfile. Although this doesn't actually publish the container's port, it's still a very useful line. It allows developers to look at the Dockerfile as documentation and understand exactly what ports need to be exposed to work properly.

Here's the completed project:

<https://github.com/15Dkatz/docker-guides/tree/master/php>

After cloning the project, build and run the container:

- ``docker build . -t docker/php``
- ``docker run --name=php -p=3003:80 docker/php``
- Visit <http://localhost:3003>

-- Build a Python Flask Image --

We built a Python Flask Image. We also saw how the ``WORKDIR`` option in the Dockerfile can allow us to set a working directory for the container.

This ensures that following commands like ``COPY`` or ``CMD`` are set in the context of that working directory.

Here's the completed project:

<https://github.com/15Dkatz/docker-guides/tree/master/flask>

After cloning the project, build and run the container:

- ``docker build . -t docker/flask``
- ``docker run --name=flask -p=3004:80 docker/flask``

Docker Images in Depth | Summary and Commands

Docker Images in Depth | Summary and Commands

Excellent job on completing this section on taking a deeper dive into docker images. In this section, we learned plenty of important lessons about images.

For one, images are highly important since they provide the blueprints for docker containers. They are created by editing a customized document called the Dockerfile. Examples of dockerfiles, as well as hundreds of useful images are all stored on dockerhub - ready for us to explore.

Next, we'll explore more advanced features of containers. Soon, we'll even allow multiple containers to interact in order to have really complex Docker application setups.

In the meantime, here's a collection of the Docker commands we have used thus far:

Docker Containers

Create an interactive terminal container with a name, an image, and a default command:

- Usage: `docker create -it --name=<name> <image> <command>`
- Example: `docker create -it --name=foo ubuntu bash`

List all running containers:

- `docker container ls`
- (list all containers, running or not): `docker container ls -a`

Start a docker container:

- Usage: `docker start <container name/id>`
- Example: `docker start foo`

Attach to a docker container:

- Usage: `docker attach <container name/id>`
- Example: `docker attach foo`

Remove a container:

- Usage: `docker rm <container name/id>`
- Example: `docker rm foo`
- Force remove: `docker rm foo -f`

Run a new container:

- Usage: `docker run <image> <command>`
- Example with options: `docker run --name=bar -it ubuntu bash`

Remove all containers:

- `docker container ls -aq | xargs docker container rm`

Execute a command in a running container:

- Usage: `docker exec <container name/id> <command>`
- Example (interactive, with tty): `docker exec -it express bash`

Docker Images

Remove a docker image:

- Usage: `docker image rmi <image id>`
- Example (only uses first 3 characters of image id): `docker rmi 70b`

Remove all images:

- `docker image ls -aq | xargs docker rmi -f`

Search for a docker image on dockerhub:

- Usage: `docker search <image>`
- Example: `docker search ubuntu`

List docker images:

- `docker image ls`

Build a Docker image:

- Usage: `docker build <path>`
- Example (also tags and names the build): `docker build . -t org/serve:0.0.0`

Dockerfiles

Specify a base image:

- Usage: FROM <base image>
- Example: FROM node:latest

Set a working directory for the container:

- Usage: WORKDIR <dir>
- Example: WORKDIR /usr/src/app

Run a command for the container image:

- Usage: RUN command
- Command: RUN npm install -g serve

Copy files into the container:

- Usage: COPY <local files/directories> <container files/directories>
- Example: COPY ./display ./display

Inform that a port should be exposed

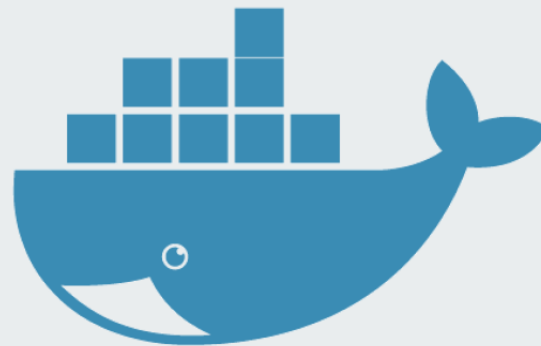
- Usage: EXPOSE <port>
- Example: EXPOSE 80

Specify a default command for the container:

- Usage (shell format): CMD <default command>
- Example: CMD serve ./display
- Usage (exec format, *recommended*): CMD ["default command", "arguments"]
- Example: CMD ["node", "server.js"]



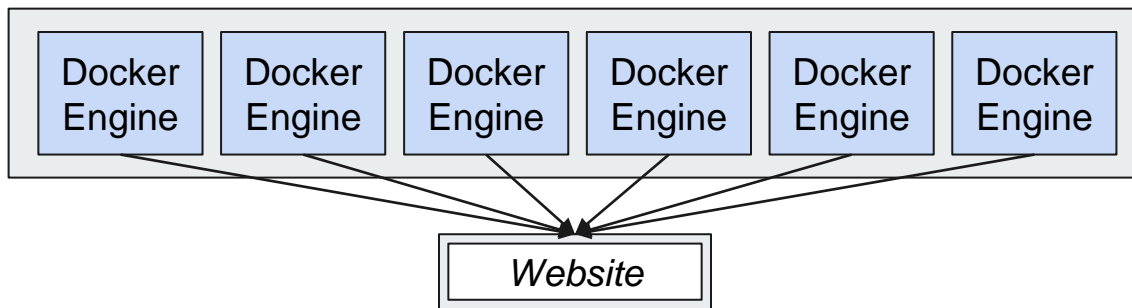
Docker Swarm





Swarm mode

- A feature within Docker to maintain a cluster of machines running the Docker engine.
- The *swarm* is the cluster itself.





Swarm mode

- A **node** “is an instance of the Docker engine participating in the swarm.”
- Nodes will have a **manager** or **worker** role.
- A **service** runs software in the swarm based off of an image.
- A **replica** takes the image software and clones it onto nodes.
- A **task** is an instance of a Docker container running in a node.



Why Docker Swarm?

- Extreme ability to scale.
- Highly reliable.
- Great features: load balancing, service discovery, and more.

-- Grab an Amazon EC2 Cloud Linux Instance (x3) --

As a result of this video, we have three amazon ec2 linux instances that we can turn into nodes for a docker swarm cluster. Essentially, we'll soon install docker on all three systems, and then group them together. This "grouping" will create a swarm, which allows us to pool each of three machine's docker engines to balance the load for running services and applications.

Here are the steps we took to create each ec2 instance, and prepare it for becoming a node in a docker swarm:

- Get an AWS account.
 - Go to `Instances` on the sidebar.
 - Launch an instance. Select `AWS Linux` (the first option usually)
 - Select `t2.micro` (free tier eligible)
 - Click `Edit security groups`
 - Add the following rules, accepting all connections: SSH port 22, HTTP port, 80, TCP port 2377, UDP port 4789, and TCP+UDP on 7946.
 - Click `launch`
 - Create a new key pair and name it something similar to `swarm-nodes`. Download the .pem file.
 - Click `launch`
 - Make the .pem file private: `chmod 400 swarm-nodes.pem`
 - Go to the instances, and look at the instance description (click on its checkbox to open the description).
Make a note of the `public ipv4 address`.
 - Finally, ssh into the instance (Make sure you're in the directory where swarm-nodes.pem is located):
`ssh -i swarm-nodes.pem ec2-user@<public ipv4 address>`.
- Should look something like `ssh -i `swarm-nodes.pem ec2-user@192.10.0.31`

If all goes well, you're now running a shell within the ec2 instance.

-- Install Docker on Ec2 Instances --

After getting access to the ec2 instance, we need to install docker and start the docker daemon.

Here are the steps to install docker on the ec2 instance:

- `sudo yum install -y docker`
- `sudo service docker start`
- `sudo usermod -G docker ec2-user`
- Re-connect to the instance.
 - Try `docker` and `docker container ls`

Note that we also wrote a convenient bash script to quickly connect to our various instances with one command. For example, to connect to the first instance:

- `./cswarm.sh 1`

Or the second:

- `./cswarm.sh 2`

Here's the script (substitute the ip addresses for the ones of your instances):

<https://github.com/15Dkatz/docker-guides/blob/master/swarm/cswarm.sh>

-- Docker Swarm - Scale Services and Explore the Routing Mesh --

We created our first docker swarm, and explored its various features such as the routing mesh. First, we initialized a swarm in our first ec2 instance. And then in our other two instances, we joined that swarm as worker nodes.

To initialize the swarm:

- Go to the first ec2 instance
- Run ``ifconfig`` and make a note of the ip address under ``eth0``
- ``docker swarm init --advertise-addr=<ip address>``, something like:
``docker swarm init --advertise-addr=172.31.17.31``
- make a note of the printed ``docker swarm join...`` command, as this will be run in other instances to join the swarm as workers.

In the other instances, join the swarm as workers with the copied command from the step above. Something like:

```
`docker swarm join --token  
SWMTKN-1-592fo0c3lguwi9cw58jpbq9dolo5jzyw5fzbk9dwiw8bm4xxpad-94vn587o9o3r73h3e5esujxm9  
172.31.17.31:2377`
```

Back in the manager node (the first ec2 instance):

- List the nodes in the swarm: ``docker node ls``
- ``docker network ls``
- ``docker network inspect ingress`` (Note the "Peers" section)
- Start a service: ``docker service create --name=site --publish=80:80 nginx``
- Visit the public ip address of any of the ec2 instances to see the routing mesh in action.
- Create replicas of the running nginx container across the swarm:
``docker service update --replicas 6 site``

Docker Swarm | Summary and Commands

Great work on completing this section on Docker Swarm. In this section, we went over quite a few important concepts.

For one, we explored docker swarm and learned how having more than one machine running docker can really help engineering efforts. With docker swarm, we have a system for extreme scalability and reliability. Not to mention, we great benefits such as the swarm routing mesh, and load balancing between nodes (docker hosts connected to the swarm).

NOTE: Be sure to terminate your amazon ec2 instances once you're done with them. Even if it's less than a dollar, you probably don't want charges going to your credit card for unused instances. You can terminate the instances by updating their instance state with the "Actions" dropdown in the management console.

Finally, here's the summary of commands that we've used so far. Head to the bottom of the summary to see new notes on Docker swarm.

Docker Containers

Create an interactive terminal container with a name, an image, and a default command:

- Usage: `docker create -it --name=<name> <image> <command>`
- Example: `docker create -it --name=foo ubuntu bash`

List all running containers:

- `docker container ls`
- (list all containers, running or not): `docker container ls -a`

Start a docker container:

- Usage: `docker start <container name/id>`
- Example: `docker start foo`

Attach to a docker container:

- Usage: `docker attach <container name/id>`
- Example: `docker attach foo`

Remove a container:

- Usage: `docker rm <container name/id>`
- Example: `docker rm foo`
- Force remove: `docker rm foo -f`

Run a new container:

- Usage: docker run <image> <command>
- Example with options: docker run --name=bar -it ubuntu bash

Remove all containers:

- docker container ls -aq | xargs docker container rm

Execute a command in a running container:

- Usage: docker exec <container name/id> <command>
- Example (interactive, with tty): docker exec -it express bash

Docker Images

Remove a docker image:

- Usage: docker image rmi <image id>
- Example (only uses first 3 characters of image id): docker rmi 70b

Remove all images:

- docker image ls -aq | xargs docker rmi -f

Search for a docker image on dockerhub:

- Usage: docker search <image>
- Example: docker search ubuntu

List docker images:

- docker image ls

Build a Docker image:

- Usage: docker build <path>
- Example (also tags and names the build): docker build . -t org/serve:0.0.0

Dockerfiles

Specify a base image:

- Usage: FROM <base image>
- Example: FROM node:latest

Set a working directory for the container:

- Usage: WORKDIR <dir>
- Example: WORKDIR /usr/src/app

Run a command for the container image:

- Usage: RUN command
- Command: RUN npm install -g serve

Copy files into the container:

- Usage: COPY <local files/directories> <container files/directories>
- Example: COPY ./display ./display

Inform that a port should be exposed

- Usage: EXPOSE <port>
- Example: EXPOSE 80

Specify a default command for the container:

- Usage (shell format): CMD <default command>
- Example: CMD serve ./display
- Usage (exec format, *recommended*): CMD ["default command", "arguments"]
- Example: CMD ["node", "server.js"]

ross-Container Storage

Volumes

Create a volume

- Usage: docker volume create <volume name>
- Example: docker volume create shared-vol

Inspect a volume

- Usage: docker volume inspect <volume name>
- Example: docker volume inspect shared-vol

Mount a container with a volume using docker run

- Usage: --mount source=<volume name>, target=<container dir>
- Example:
docker run -it --name=foo --mount source=shared-vol,target=/src/shared ubuntu bash

Bind Mounts

Mount a container with a bind mount using docker run

- Usage: --mount type=bind source=<host dir>, target=<container dir>
- Example:
docker run -it --name=foo --mount type=bind source=/Users/foo/bindmountdir, \ target=/src/mountdir ubuntu bash

Tmpfs mounts

Mount a container with a tmpfs mount using docker run

- Usage: --mount type=tmpfs, destination=<container dir>
- Example:
docker run -it --name=baz --mount type=tmpfs, destination=/tmpdir ubuntu bash

Docker Networking

List docker networks

- docker network ls

Inspect a docker network

- Usage: docker network inspect <network name>
- Example: docker network inspect bridge

Create a docker network

- Usage: docker network create <network name>
- Example: docker network create privatenw

Run a container with a custom docker network:

- Usage: --network=<network name>
- Example: docker run --network=privatenw -it --name=goo busybox

Docker Compose

Start a compose application

- At the root (where docker-compose.yml is located): docker-compose up

Start a compose application and rebuild images:

- Docker-compose up --build

docker-compose.yml

Version

- Current version is 3. So at the top of the file, specify: version: '3'

Services with builds

- Have a services key in the file. List out services one indent at a time.

Dependencies

- Use the depends_on key and specify dependencies with a list. Each container dependency is marked by a dash, such as: -backend

Volumes

- Have a volume key per service.
- Connect a Docker host directory to a container directory, by joining them with a colon.
- Example: ./dockerhostdir:/containerdir

Networks

- Declare networks at the bottom of the file.
- Specify each service's network(s) with the networks option for each service.

Docker Swarm

Initialize a swarm in a node

- Usage: docker swarm init --advertise-addr=<node ip>
- Example: docker swarm init --advertise-addr=172.31.17.31

After initializing the swarm, you will find a join command for worker/other manager nodes

- Example: docker swarm join --token SWMTKN-1-592fo0c31guwi9cw58jpaw89fafzyw5fzbk9dwiw8bm4xxpad-94vn587o9o3r73h3e5esujxm9 172.31.17.31:2377

List docker nodes from a manger:

- docker node ls

Create a service for the swarm:

- Usage: docker service create --name=<service name> --publish=<host port:service port> <service image>
- Example: docker service create --name=site --publish=80:80 nginx

List services:

- docker service ls

List the running tasks for a service:

- Usage: docker service ps <service name>
- Example: docker service ps site

Update a service

- Usage: docker service update [options] <service name>
- Example: docker service update --replicas=6 site