# Introduction to Docker

## What is Docker?

Docker:

* Docker, Inc. the company
* Docker, the container runtime and orchestration engine
* Docker, the open-source project (Moby)

## The Company

Docker, Inc.:

* Based in San Francisco
* Founded by Solomon Hykes
* Start as a PaaS provider called dotCloud
* dotCloud leveraged Linux containers
* Their internal tool used to manage containers was nick-named Docker
* In 2013 dotCloud was rebranded as Docker

## The Runtime and Orchestration Engine

The Docker runtime and orchestration engine:

* Most people are referring to the Docker Engine
* Two main editions:
  + Enterprise Edition (EE)
  + Community Edition (CE)
* Both are released quarterly:
  + CE is supported for 4 months
  + EE is supported for 12 months

## The Open-Source Project

Moby:

* The upstream project of Docker
* Breaks Docker down into more modular components
* Code is available on GitHub:
  + <https://github.com/moby/moby>

## Why Use Docker?

Docker Use Cases:

* Dev/Prod parity:
  + Dev and Production environment are the same
  + Bugs in Production can be replicated in Development
* Simplifying Configuration:
  + Lets you put your environment and configuration into code and deploy it
  + Allows the same Docker configuration to be used in a variety of environments
  + Decouples infrastructure requirements from the application environment
* Code Pipeline Management:
  + Build standards and repeatable processes
* Developer Productivity
* App Isolation
* Server Consolidation
* Debugging Capabilities
* Multi-tenancy

# Installing Docker

In this lesson, we'll install the latest version of Docker CE. The commands used throughtout this video are below.

## Prerequisites

Uninstall old versions:

sudo yum remove -y docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-engine

## Install Docker CE

Add the Utilities needed for Docker:

sudo yum install -y yum-utils \

device-mapper-persistent-data \

lvm2

Set up the stable repository:

sudo yum-config-manager \

--add-repo \

https://download.docker.com/linux/centos/docker-ce.repo

Install Docker CE:

sudo yum -y install docker-ce

Enable and start Docker:

sudo systemctl start docker && sudo systemctl enable docker

Add cloud\_user to the docker group:

sudo usermod -aG docker cloud\_user

# Docker Architecture

In this lesson we will take a high-level look at the Docker Architecture.

## Architecture Overview

Docker architecture:

* Client-server architecture
* Client talks to the Docker daemon
* The Docker daemon handles:
  + Building
  + Running
  + Distributing
* Both communicate using a REST API:
  + UNIX sockets
  + Network interface

The Docker daemon (dockerd):

* Listens for Docker API requests and manages Docker objects:
  + Images
  + Containers
  + Networks
  + Volumes

The Docker client (docker):

* Is how users interact with Docker
* Sends commands to dockerd

Docker registries:

* Stores Docker images
* Public registry such as DockerHub
* Let you run your own private registry

Docker objects:

* Images:
  + Read-only template with instructions for creating a Docker container
  + Image is based on another image
  + Create your own images
  + Use images published to a registry
  + Use a Dockerfile to build images
* Containers:
  + Runnable instance of an image
  + Connect a container to networks
  + Attach storage
  + Create a new image based on its current state
  + Isolated from other containers and the host machine
* Services
  + Scale containers across multiple Docker daemons
  + Docker Swarm
  + Define the desired state
  + Service is load-balanced

Docker Swarm:

* Multiple Docker daemons (Master and Workers)
* The daemons all communicate using the Docker API
* Supported in Docker 1.12 and higher

# The Docker Engine

In this lesson, we will go over the Docker Engine and it's various components.

## Under The Hood

Docker engine:

* Modular in design:
  + Batteries included but replaceable
* Based on open-standards outline by the Open Container Initiative
* The major components:
  + Docker client
  + Docker daemon
  + containerd
  + runc
* The components work together to create and run containers

## A Brief History of the Docker Engine

The first release of Docker:

* The Docker daemon:
  + Monolithic binary
  + Docker client
  + Docker API
  + Container runtime
  + Image builds
  + Much more...
* LXC:
  + Namespaces
  + Control groups (cgroups)
  + Linux-specific

## Refactoring of the Docker Engine

LXC was later replaced with libcontainer:

* Docker 0.9
* Platform agnostic

Issues with the monolithic Docker daemon:

* Harder to innovate
* Slow
* Not what the ecosystem wanted

Docker became more modular:

* Smaller more specialized tools
* Pluggable architecture

Open Container Initiative:

* Image spec
* Container runtime spec
* Version 1.0 release in 2017
* Docker Inc. heavily contributed
* Docker 1.11 (2016) used the specification as much as possible

runc:

* Implemenation of the OCI container-runtime-spec
* Lightweght CLI wrapper for libcontainer
* Create containers

containerd:

* Manages container lifecycle
  + Start
  + Stop
  + Pause
  + Delete
* Image management
* Part of the 1.11 release

shim:

* Implemenation of daemonless Containers
* containerd forks an instance of runc for each new container
* runc process exits after the container is created
* shim process becomes the container parent
* Responsible for:
  + STDIN and STDOUT
  + Reporting exit status to the Docker daemon

## Running Containers

docker container run -it --name <NAME> <IMAGE>:<TAG>

Creating a container:

* CLI use for executing a command
* Docker client uses the appropriate API payload
* POSTs to the correct API endpoint
* Docker deamon receives instructions
* Docker deamon calls containerd to start a new container
* Docker daemon uses gRPC (a CRUD style API)
* containerd creates an OCI bundle from the Docker image
* Tells runc to create a container using the OCI bundle
* runc interfaces with the OS kernal to get the constructs needed to create a container
  + This includes namespaces, cgroups, etc.
* Container process starts as a child process
* runc exits once the container starts
* Process is complete, and container is running

# Docker Images and Containers

## What are Docker images?

Docker Images:

* Files comprised of multiple layers
* Execute code in a Docker container
* Built from the instructions
* Use images to create an instance of a container

## Docker images and layers

* Image are made of multiple layers.
* Each layer represents an instruction in the image’s Dockerfile.
* Each layer except, the very last one, is read-only.
* Each layer is only a set of differences from the layer before it.
* Layers are stacked on top of each other.
* Containers add new writable layers on top of the underlying layers
* All changes made to a running container is made to the Container layer

## What are containers?

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

## Container and layers

* Top writable layer
* All changes are stored in the writable layer
* The writable layer is deleted when the container is deleted
* The image remains unchanged

# Docker Hub

In this lesson, we will take a quick look at Docker Hub and its available features.

## What is Docker Hub?

Docker Hub:

* Public Docker registry
* Provided by Docker
* Features:
  + Repositories
  + Teams and Organizations
  + Official Images
  + Publisher Images
  + Builds
  + Webhooks
* <https://hub.docker.com/signup>

# Docker Commands

In this lesson we will start working with Docker commands. We'll see the management commands, along with the ones for managing images and containers.

Get a list of all of the Docker commands:

docker -h

## Management command were introduced in Docker engine v1.13

Management Commands:

* builder Manage builds
* config Manage Docker configs
* container Manage containers
* engine Manage the docker engine
* image Manage images
* network Manage networks
* node Manage Swarm nodes
* plugin Manage plugins
* secret Manage Docker secrets
* service Manage services
* stack Manage Docker stacks
* swarm Manage Swarm
* system Manage Docker
* trust Manage trust on Docker images
* volume Manage volumes

docker image:

* build Build an image from a dockerfile
* history Show the history of an image
* import Import the contents from a tarball to create a filesystem image
* inspect Display detailed information on one or more images
* load Load an image from a tar file or STDIN
* ls List images
* prune Remove unused images
* pull Pull an image or a repository from a registry
* push Push an image or a repository to a registry
* rm Remove one or more images
* save Save one or more images to a tar file (streamed to STDOUT by default)
* tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

docker container:

* attach Attach local standard input, output, and error streams to a running container
* commit Create a new image from a container's changes
* cp Copy files/folders between a container and the local filesystem
* create Create a new container
* diff Inspect changes to files or directories on a container's filesystem
* exec Run a command in a running container
* export Export a container's filesystem as a tar archive
* inspect Display detailed information on one or more containers
* kill Kill one or more running containers
* logs Fetch the logs of a container
* ls List containers
* pause Pause all processes within one or more containers
* port List port mappings or a specific mapping for the container
* prune Remove all stopped containers
* rename Rename a container
* restart Restart one or more containers
* rm Remove one or more containers
* run Run a command in a new container
* start Start one or more stopped containers
* stats Display a live stream of container(s) resource usage statistics
* stop Stop one or more running containers
* top Display the running processes of a container
* unpause Unpause all processes within one or more containers
* update Update configuration of one or more containers
* wait Block until one or more containers stop, then print their exit codes

**Creating Containers**

In this lesson, we will take a deeper look into creating containers, by exploring a few of the flags that will alter it's behavior when created.

docker container run:

* --help Print usage
* --rm Automatically remove the container when it exits
* -d, --detach Run container in background and print container ID
* -i, --interactive Keep STDIN open even if not attached
* --name string Assign a name to the container
* -p, --publish list Publish a container's port(s) to the host
* -t, --tty Allocate a pseudo-TTY
* -v, --volume list Mount a volume (the bind type of mount)
* --mount mount Attach a filesystem mount to the container
* --network string Connect a container to a network (default "default")

Create a container and attach to it:

docker container run –it busybox

Create a container and run it in the background:

docker container run –d nginx

Create a container that you name and run it in the background:

docker container run –d –name myContainer busybox

**Exposing and Publishing Container Ports**

Building on what we've already learned, this lesson will focus on exposing ports on a container, as well as how to publish them.

Exposing:

* Expose a port or a range of ports
* This does not publish the port
* Use --expose [PORT]

docker container run --expose 1234 [IMAGE]

Publishing:

* Maps a container's port to a host`s port
* -p or --publish publishes a container's port(s) to the host
* -P, or --publish-all publishes all exposed ports to random ports

docker container run -p [HOST\_PORT]:[CONTAINER\_PORT] [IMAGE]

docker container run -p [HOST\_PORT]:[CONTAINER\_PORT]/tcp -p [HOST\_PORT]:[CONTAINER\_PORT]/udp [IMAGE]

docker container run -P

Lists all port mappings or a specific mapping for a container:

docker container port [Container\_NAME]

**Executing Container Commands**

In this lesson we'll see three different ways to execute commands on containers.

Executing a command:

* Dockerfile
* During a Docker run
* Using the exec command

Commands can be:

* One and done Commands
* Long running Commands

Start a container with a command:

docker container run [IMAGE] [CMD]

Execute a command on a container:

docker container exec -it [NAME] [CMD]

Example:

docker container run -d -p 8080:80 nginx

docker container ps

docker container exec -it [NAME] /bin/bash

docker container exec -it [NAME] ls /usr/share/nginx/html/

[Nginx Dockerfile](https://github.com/nginxinc/docker-nginx/blob/baa050df601b5e798431a9db458e16f53b1031f6/mainline/stretch/Dockerfile)

# Container Logging

In this lesson, you will learn how to view the logs of a container to get vital output of your application. You will also learn about some of the logging best practices of containerized applications.

Create a container using the weather-app image.

docker container run --name weather-app -d -p 80:3000 linuxacademycontent/weather-app

Show information logged by a running container:

docker container logs [NAME]

Show information logged by all containers participating in a service:

docker service logs [SERVICE]

Logs need to be output to STDOUT and STDERR.

Nginx Example:

RUN ln -sf /dev/stdout /var/log/nginx/access.log \

&& ln -sf /dev/stderr /var/log/nginx/error.log

Debug a failed container deploy:

docker container run -d --name ghost\_blog \

-e database\_\_client=mysql \

-e database\_\_connection\_\_host=mysql \

-e database\_\_connection\_\_user=root \

-e database\_\_connection\_\_password=P4sSw0rd0! \

-e database\_\_connection\_\_database=ghost \

-p 8080:2368 \

ghost:1-alpine

Useful links:  
[12 Factor Logs](https://12factor.net/logs)  
[Weather App Code](https://github.com/linuxacademy/content-intermediate-docker-quest/tree/logging)  
[Ruby Logging](https://ruby-doc.org/stdlib-2.6/libdoc/logger/rdoc/Logger.html)  
[Python Logging](https://docs.python.org/2/howto/logging.html)

# Networking Overview

In this lesson, we will go over the components and theory of how networking functions in Docker.

## Docker Networking 101

Docker Networking:

* Open-source pluggable architecture
* Container Network Model (CNM)
* libnetwork implements CNM
* Drivers extend the network topologies

Network Drivers:

* bridge
* host
* overlay
* macvlan
* none
* Network plugins

## Container Network Model

Defines three building blocks:

* Sandboxes
* Endpoints
* Networks

# Networking Commands

## Networking Basics

ifconfig

List all Docker network commands:

docker network -h

connect Connect a container to a network create Create a network disconnect Disconnect a container from a network inspect Display detailed information on one or more networks ls List networks prune Remove all unused networks rm Remove one or more networks

List all Docker networks on the host:

docker network ls

docker network ls --no-trunc

Getting detailed info on a network:

docker network inspect [NAME]

Creating a network:

docker network create br00

Deleting a network:

docker network rm [NAME]

Remove all unused networks:

docker network prune

## Adding and Removing containers to a network

Create a container with no network:

docker container run -d --name network-test03 -p 8081:80 nginx

Create a new network:

docker network create br01

Add the container to the bridge network:

docker network connect br01 network-test03

Inspect network-test03 to see the networks:

docker container inspect network-test03

Remove network-test03 from br01:

docker network disconnect br01 network-test03

# Networking Containers

In this lesson, we will dig deeper into container networking by supplying our own subnet and gateway when creating a new network. We will then move on to networking two different containers using an internal network. This will allow one container to be publicly accessible while the other one is not.

## Creating a network and defining a Subnet and Gateway

Create a bridge network with a subnet and gateway:

docker network create --subnet 10.1.0.0/24 --gateway 10.1.0.1 br02

Run ifconfig to view the bridge interface for br02:

ifconfig

Inspect the br02 network:

docker network inspect br02

Prune all unused networks:

docker network prune

Create a network with an IP range:

docker network create --subnet 10.1.0.0/16 --gateway 10.1.0.1 \

--ip-range=10.1.4.0/24 --driver=bridge --label=host4network br04

Inspect the br04 network:

docker network inspect br04

Create a container using the br04 network:

docker container run --name network-test01 -it --network br04 centos /bin/bash

Install Net Tools:

yum update -y

yum install -y net-tools

Get the IP info for the container:

ifconfig

Get the gateway info the container:

netstat -rn

Get the DNS info for the container:

cat /etc/resolv.conf

## Assigning IPs to a container:

Create a new container and assign an IP to it:

docker container run -d --name network-test02 --ip 10.1.4.102 --network br04 nginx

Get the IP info for the container:

docker container inspect network-test02 | grep IPAddr

Inspect network-test03 to see that br01 was removed:

docker container inspect network-test04

## Networking two containers

Create an internal network:

docker network create -d bridge --internal localhost

Create a MySQL container that is connected to localhost:

docker container run -d --name test\_mysql \

-e MYSQL\_ROOT\_PASSWORD=P4sSw0rd0 \

--network localhost mysql:5.7

Create a container that can ping the MySQL container:

docker container run -it --name ping-mysql \

--network bridge --network localhost \

centos

Create a container that can't ping the MySQL container:

docker container run -it --name cant-ping-mysql \

centos

Create a Nginx container that is not publicly accessible:

docker container run -d --name private-nginx -p 8081:80 --network localhost nginx

Inspect private-nginx:

docker container inspect private-nginx

# Storage Overview

In this lesson, we will look a how Docker handles storage for persistent and non-persistent data.

## Docker Storage 101

Categories of data storage:

* Non-persistent
  + Local storage
  + Data that is ephemeral
  + Every container has it
  + Tied to the lifecycle of the contain
* Persistent
  + Volumes
    - Volumes are decoupled from containers

## Non-persistent Data

Non-persistent data:

* By default all container use local storage
* Storage locations:
  + Linux: /var/lib/docker/[STORAGE-DRIVER]/
  + Windows: C:\ProgramData\Docker\windowsfilter\
* Storage Drivers:
  + RHEL uses overlay2.
  + Ubuntu uses overlay2 or aufs.
  + SUSE uses btrfs.
  + Windows uses its own.

## Persistent Data Using Volumes

Volumes:

* Use a volume for persistent data:
  + Create the volume first, then create your container.
* Mounted to a directory in the container
* Data is written to the volume
* Deleting a container does not delete the volume
* First-class citizens
* Uses the local driver
* Third party drivers:
  + Block storage
  + File storage
  + Object storage
* Storage locations:
  + Linux: /var/lib/docker/volumes/
  + Windows: C:\ProgramData\Docker\volumes

# Volume Commands

Volumes are the preferred method of maintaining persistent data in Docker. In this lesson, we will begin learning how to use the volume subcommand to list, create, and remove volumes.

## Volume Basics

List all Docker volume commands:

docker volume -h

* create: Create a volume.
* inspect: Display detailed information on one or more volumes.
* ls: List volumes.
* prune: Remove all unused local volumes.
* rm: Remove one or more volumes.

List all volumes on a host:

docker volume ls

Create two new volumes:

docker volume create test-volume1

docker volume create test-volume2

Get the flags available when creating a volume:

docker volume create -h

Inspecting a volume:

docker volume inspect test-volume1

Deleting a volume:

docker volume rm test-volume

Removing all unused volumes:

docker volume prune

# Using Bind Mounts

Bind mounts have been around since the early days of Docker. They have limited functionality compared to volumes. With bind mount, a file or directory on the host machine is mounted into a container.

Volumes use a new directory that is created within Docker’s storage directory on the host machine, and Docker manages that directory’s contents.

Using the mount flag:

mkdir target

docker container run -d \

--name nginx-bind-mount1 \

--mount type=bind,source="$(pwd)"/target,target=/app \

nginx

docker container ls

Bind mounts won't show up when listing volumes:

docker volume ls

Inspect the container to find the bind mount:

docker container inspect nginx-bind-mount1

Create a new file in /app on the container:

docker container exec -it nginx-bind-mount1 /bin/bash

cd target

touch file1.txt

ls

exit

Using the volume flag:

docker container run -d \

--name nginx-bind-mount2 \

-v "$(pwd)"/target2:/app \

nginx

Create /app/file3.txt in the container:

docker container exec -it nginx-bind-mount2 touch /app/file3.txt

ls target2

Create an nginx.conf file:

mkdir nginx

cat << EOF > nginx/nginx.conf

user nginx;

worker\_processes 1;

error\_log /var/log/nginx/error.log warn;

pid /var/run/nginx.pid;

events {

worker\_connections 1024;

}

http {

include /etc/nginx/mime.types;

default\_type application/octet-stream;

log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

'$status $body\_bytes\_sent "$http\_referer" '

'"$http\_user\_agent" "$http\_x\_forwarded\_for"';

access\_log /var/log/nginx/access.log main;

sendfile on;

#tcp\_nopush on;

keepalive\_timeout 65;

#gzip on;

include /etc/nginx/conf.d/\*.conf;

}

EOF

Create an Nginx container that creates a bind mount to nginx.conf:

docker container run -d \

--name nginx-bind-mount3 \

-v "$(pwd)"/nginx/nginx.conf:/etc/nginx/nginx.conf \

nginx

Look at the bind mount by inspecting the container:

docker container inspect nginx-bind-mount3

**Using Volumes for Persistent Storage**

In this lesson, we will take a deeper look into using volumes with our Docker containers. Volumes are the preferred method for maintaining persistent data.

Volumes are easier to back up or migrate than bind mounts. You can manage volumes using Docker CLI commands or the Docker API. They work on both Linux and Windows containers. Volumes can be more safely shared among multiple containers. Volume drivers allow for:

* Storing volumes on remote hosts or cloud providers
* Encrypting the contents of volumes
* Add other functionality

New volumes can have their content pre-populated by a container.

Create a new volume for an Nginx container:

docker volume create html-volume

Creating a volume using that volume mount:

docker container run -d \

--name nginx-volume1 \

--mount type=volume,source=html-volume,target=/usr/share/nginx/html/ \

nginx

Inspect the volume:

docker volume inspect html-volume

List the contents of html-volume:

sudo ls /var/lib/docker/volumes/html-volume/\_data

Creating a volume using that volume flag:

docker container run -d \

--name nginx-volume2 \

-v html-volume:/usr/share/nginx/html/ \

nginx

Edit index.html:

sudo vi /var/lib/docker/volumes/html-volume/\_data/index.html

Inspect nginx-volume2 to get the private IP:

docker container inspect nginx-volume2

Login into nginx-volume1 and go to the html directory:

docker container exec -it nginx-volume1 /bin/bash

cd /usr/share/nginx/html

cat index.hml

Install Vim:

apt-get update -y

apt-get install vim -y

Using a readonly volume:

docker run -d \

--name=nginx-volume3 \

--mount source=html-volume,target=/usr/share/nginx/html,readonly \

nginx

Login into nginx-volume3 and go to the html directory:

docker container exec -it nginx-volume3 /bin/bash

cd /usr/share/nginx/html

cat index.hml

Install Vim:

apt-get update -y

apt-get install vim -y

# Introduction to the Dockerfile

In this lesson we will start learning about building images using a Dockerfile.

## What is the Dockerfile?

Dockerfiles are instructions. They contains all of the commands used to build an image.

* Docker images consist of read-only layers.
* Each represents a Dockerfile instruction.
* Layers are stacked.
* Each layer is a result of the changes from the previous layer.
* Images are built using the docker image build command.

## Dockerfile Layers

Dockerfile:

FROM ubuntu:15.04

COPY . /app

RUN make /app

CMD python /app/app.py

* FROM creates a layer from the ubuntu:15.04 Docker image.
* COPY adds files from your Docker client’s current directory.
* RUN builds your application with make.
* CMD specifies what command to run within the container.

## Best Practices

General guidelines:

* Keep containers as ephemeral as possible.
* Follow Principle 6 of the 12 Factor App.
* Avoid including unnecessary files.
* Use .dockerignore.
* Use multi-stage builds.
* Don’t install unnecessary packages.
* Decouple applications.
* Minimize the number of layers.
* Sort multi-line arguments.
* Leverage build cache.

# Working with Instructions

FROM: Initializes a new build stage and sets the Base Image

RUN: Will execute any commands in a new layer

CMD: Provides a default for an executing container. There can only be one CMD instruction in a Dockerfile

LABEL: Adds metadata to an image

EXPOSE: Informs Docker that the container listens on the specified network ports at runtime

ENV: Sets the environment variable <key> to the value <value>

ADD: Copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

COPY: Copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

ENTRYPOINT: Allows for configuring a container that will run as an executable

VOLUME: Creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers

USER: Sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any RUN, CMD, and ENTRYPOINT instructions that follow it in the Dockerfile

WORKDIR: Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that follow it in the Dockerfile

ARG: Defines a variable that users can pass at build-time to the builder with the docker build command, using the --build-arg <varname>=<value> flag

ONBUILD: Adds a trigger instruction to the image that will be executed at a later time, when the image is used as the base for another build

HEALTHCHECK: Tells Docker how to test a container to check that it is still working

SHELL: Allows the default shell used for the shell form of commands to be overridden

To set up the environment:

sudo yum install git -y

mkdir docker\_images

cd docker\_images

mkdir weather-app

cd weather-app

git clone https://github.com/linuxacademy/content-weather-app.git src

Create the Dockerfile:

vi Dockerfile

Dockerfile contents:

# Create an image for the weather-app

FROM node

LABEL org.label-schema.version=v1.1

RUN mkdir -p /var/node

ADD src/ /var/node/

WORKDIR /var/node

RUN npm install

EXPOSE 3000

CMD ./bin/www

Build the weather-app image:

docker image build -t linuxacademy/weather-app:v1 .

List the images:

docker image ls

Create the weather-app container:

docker container run -d --name weather-app1 -p 8081:3000 linuxacademy/weather-app:v1

List all running containers:

docker container ls

[ADD or Copy](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#add-or-copy)

# Environment Variables

To make new software easier to run, you can use ENV to update the PATH environment variable for the software that your container installs.

Setup your environment:

cd docker\_images

mkdir env

cd env

Use the --env flag to pass an environment variable when building an image:

--env [KEY]=[VALUE]

Use the ENV instruction in the Dockerfile:

ENV [KEY]=[VALUE]

ENV [KEY] [VALUE]

Clone the weather-app:

git clone https://github.com/linuxacademy/content-weather-app.git src

Create the Dockerfile

vi Dockerfile

Dockerfile contents:

# Create an image for the weather-app

FROM node

LABEL org.label-schema.version=v1.1

ENV NODE\_ENV="development"

ENV PORT 3000

RUN mkdir -p /var/node

ADD src/ /var/node/

WORKDIR /var/node

RUN npm install

EXPOSE $PORT

CMD ./bin/www

Create the weather-app container:

docker image build -t linuxacademy/weather-app:v2 .

Inspect the container to see the environment variables:

docker image inspect linuxacademy/weather-app:v2

Deploy the weather-dev application:

docker container run -d --name weather-dev -p 8082:3001 --env PORT=3001 linuxacademy/weather-app:v2

Inspect the development container to see the environment variables:

docker container inspect weather-dev

Deploy the weather-app to production:

docker container run -d --name weather-app2 -p 8083:3001 --env PORT=3001 --env NODE\_ENV=production linuxacademy/weather-app:v2

Inspect the production container to see the environment variables:

docker container inspect weather-app2

Get the logs for weather-app2:

docker container logs weather-app2

docker container run -d --name weather-prod -p 8084:3000 --env NODE\_ENV=production linuxacademy/weather-app:v2

# Build Arguments

In this lesson, we will explore using build arguments to paramerterize an image build.

Use the --build-arg flag when building an image:

--build-arg [NAME]=[VALUE]

Use the ARG instruction in the Dockerfile:

ARG [NAME]=[DEFAULT\_VALUE]

Navigate to the args directory:

cd docker\_images

mkdir args

cd args

Clone the weather-app:

git clone https://github.com/linuxacademy/content-weather-app.git src

Create the Dockerfile:

vi Dockerfile

Dockerfile:

# Create an image for the weather-app

FROM node

LABEL org.label-schema.version=v1.1

ARG SRC\_DIR=/var/node

RUN mkdir -p $SRC\_DIR

ADD src/ $SRC\_DIR

WORKDIR $SRC\_DIR

RUN npm install

EXPOSE 3000

CMD ./bin/www

Create the weather-app container:

docker image build -t linuxacademy/weather-app:v3 --build-arg SRC\_DIR=/var/code .

Inspect the image:

docker image inspect linuxacademy/weather-app:v3 | grep WorkingDir

Create the weather-app container:

docker container run -d --name weather-app3 -p 8085:3000 linuxacademy/weather-app:v3

Verify that the container is working by executing curl:

curl localhost:8085

# Working with Non-privileged Users

In this lesson, you will learn how to use the USER instruction to create a non-privileged user. Rather than using root, we can use a non-privileged user to configure and run an application.

Setup your environment:

cd docker\_images

mkdir non-privileged-user

cd non-privileged-user

Create the Dockerfile:

vi Dockerfile

Dockerfile contents:

# Creates a CentOS image that uses cloud\_user as a non-privileged user

FROM centos:latest

RUN useradd -ms /bin/bash cloud\_user

USER cloud\_user

Build the new image:

docker image build -t centos7/nonroot:v1 .

Create a container using the new image:

docker container run -it --name test-build centos7/nonroot:v1 /bin/bash

Connecting as a privileged user:

docker container start test-build

docker container exec -u 0 -it test-build /bin/bash

Set up the environment:

cd ~/docker\_images

mkdir node-non-privileged-user

cd node-non-privileged-user

Create the Dockerfile:

vi Dockerfile

Dockerfile contents:

# Create an image for the weather-app

FROM node

LABEL org.label-schema.version=v1.1

RUN useradd -ms /bin/bash node\_user

USER node\_user

ADD src/ /home/node\_user

WORKDIR /home/node\_user

RUN npm install

EXPOSE 3000

CMD ./bin/www

git clone https://github.com/linuxacademy/content-weather-app.git src

Build the weather-app image using the non-privileged user node\_user:

docker image build -t linuxacademy/weather-app-nonroot:v1 .

Create a container using the linuxacademy/weather-app-nonroot:v1 image:

docker container run -d --name weather-app-nonroot -p 8086:3000 linuxacademy/weather-app-nonroot:v1

# Order of Execution

This lesson focuses on the order that instructions are executed in when building an image. Some instructions may have unintended consequences that can cause your build to fail.

Setup your environment:

cd docker\_images

mkdir centos-conf

cd centos-conf

Create the Dockerfile:

vi Dockerfile

Dockerfile contents:

# Creates a CentOS image that uses cloud\_user as a non-privileged user

FROM centos:latest

RUN mkdir -p ~/new-dir1

RUN useradd -ms /bin/bash cloud\_user

USER cloud\_user

RUN mkdir -p ~/new-dir2

RUN mkdir -p /etc/myconf

RUN echo "Some config data" >> /etc/myconf/my.conf

Build the new image:

docker image build -t centos7/myconf:v1 .

# Using the Volume Instruction

In this lesson, we will use the VOLUME instruction to automatically create a mount point in a Docker image. When a container is created using this image, a volume will be created and mounted to the specified directory.

Set up your environment:

cd docker\_images

mkdir volumes

cd volumes

Create the Dockerfile:

vi Dockerfile

Build an Nginx image that uses a volume:

FROM nginx:latest

VOLUME ["/usr/share/nginx/html/"]

Build the new image:

docker image build -t linuxacademy/nginx:v1 .

Create a new container using the linuxacademy/nginx:v1 image:

docker container run -d --name nginx-volume linuxacademy/nginx:v1

Inspect nginx-volume:

docker container inspect nginx-volume

List the volumes:

docker volume ls | grep [VOLUME\_NAME]

Inspect the volumes:

docker volume inspect [VOLUME\_NAME]

**Entrypoint vs. Command**

In this lesson, we will begin working with the ENTRYPOINT instruction. Though ENTRYPOINT functions very similarly to CMD it's behaviors are vary different.

* ENTRYPOINT allows us to configure a container that will run as an executable.
* We can override all elements specified using CMD.
* Using the docker run --entrypoint flag will override the ENTRYPOINT instruction.

Setup your environment:

cd docker\_images

mkdir entrypoint

cd entrypoint

Create the Dockerfile:

vi Dockerfile

Dockerfile contents:

# Create an image for the weather-app

FROM node

LABEL org.label-schema.version=v1.1

ENV NODE\_ENV="production"

ENV PORT 3001

RUN mkdir -p /var/node

ADD src/ /var/node/

WORKDIR /var/node

RUN npm install

EXPOSE $PORT

ENTRYPOINT ./bin/www

Clone the image:

git clone https://github.com/linuxacademy/content-weather-app.git src

Build the image:

docker image build -t linuxacademy/weather-app:v4 .

Deploy the weather-app:

docker container run -d --name weather-app4 linuxacademy/weather-app:v4

Inspect weather-app4:

docker container inspect weather-app4 | grep Cmd

docker container inspect weather-app-nonroot

docker container inspect weather-app4

Create the weather-app container:

docker container run -d --name weather-app5 -p 8083:3001 linuxacademy/weather-app:v4 echo "Hello World"

Inspect weather-app5:

docker container inspect weather-app5

Create the volumes for Prometheus:

docker volume create prometheus

docker volume create prometheus\_data

sudo chown -R nfsnobody:nfsnobody /var/lib/docker/volumes/prometheus/

sudo chown -R nfsnobody:nfsnobody /var/lib/docker/volumes/prometheus\_data/

Create the Prometheus container:

docker run --name prometheus -d -p 8084:9090 \

-v prometheus:/etc/prometheus \

-v prometheus\_data:/prometheus/data \

prom/prometheus \

--config.file=/etc/prometheus/prometheus.yml \

--storage.tsdb.path=/prometheus/data

Inspect Prometheus:

docker container inspect prometheus

[Prometheus Dockerfile](https://github.com/prometheus/prometheus/blob/master/Dockerfile)

# Using .dockerignore

In this lesson, we'll create a .dockerignore file, so that we can exclude files we don't want copied over when building an image.

Setup your environment:

cd docker\_images

mkdir dockerignore

cd dockerignore

git clone https://github.com/linuxacademy/content-weather-app.git src

cd src

git checkout dockerignore

cd ../

Create the .dockerignore file:

vi .dockerignore

Add the following to .dockerignore:

# Ignore these files

\*/\*.md

\*/.git

src/docs/

\*/tests/

Create the Dockerfile:

vi Dockerfile

Dockerfile contents:

# Create an image for the weather-app

FROM node

LABEL org.label-schema.version=v1.1

ENV NODE\_ENV="production"

ENV PORT 3000

RUN mkdir -p /var/node

ADD src/ /var/node/

WORKDIR /var/node

RUN npm install

EXPOSE $PORT

ENTRYPOINT ["./bin/www"]

Build the image:

docker image build -t linuxacademy/weather-app:v5 .

Create the weather-app container:

docker container run -d --name weather-app-ignore linuxacademy/weather-app:v5

List the contents of /var/node:

docker container exec weather-app-ignore ls -la /var/node

**Building Images**

In this lesson, we will learn some alternate ways of building images.

To build one:

docker image build -t <NAME>:<TAG> .

Useful flags:

* -f, --file string: This is the name of the Dockerfile (Default is PATH/Dockerfile).
* --force-rm: Always remove intermediate containers.
* --label list: Set metadata for an image.
* --rm: Remove intermediate containers after a successful build (default is true).
* --ulimit ulimit: This sets ulimit options (default is []).

cd docker\_images/weather-app

cp Dockerfile Dockerfile.test

docker image build -t linuxacademy/weather-app:path-example1 -f Dockerfile.test .

docker image build -t linuxacademy/weather-app:path-example2 --label com.linuxacademy.version=v1.8 -f Dockerfile.test .

Building image by piping the Dockerfile through STDIN:

docker image build -t <NAME>:<TAG> -<<EOF

Build instructions

EOF

Example:

docker image build -t linuxacademy/nginx:stind --rm -<<EOF

FROM nginx:latest

VOLUME ["/usr/share/nginx/html/"]

EOF

Building an image using a URL:

docker image build -t <NAME>:<TAG> <GIT\_URL>#<REF>

docker image build -t <NAME>:<TAG> <GIT\_URL>#:<DIRECTORY>

docker image build -t <NAME>:<TAG> <GIT\_URL>#<REF>:<DIRECTORY>

Example:

docker image build -t linuxacademy/weather-app:github https://github.com/linuxacademy/content-weather-app.git#remote-build

Building an image from a zip file:

docker image build -t <NAME>:<TAG> - < <FILE>.tar.gz

Example:

cd docker\_images

mkdir tar\_image

cd tar\_image

git clone https://github.com/linuxacademy/content-weather-app.

cd content-weather-app

git checkout remote-build

tar -zcvf weather-app.tar.gz Dockerfile src

docker image build -t linuxacademy/weather-app:from-tar - < weather-app.tar.gz

**Using Multi-Stage Builds**

In this lesson, we will learn how to build smaller images using multi-stage builds.

* By default, the stages are not named
* Stages are numbered with integers
* Starting with 0 for the first FROM instruction
* Name the stage by adding as to the FROM instruction
* Reference the stage name in the COPY instruction

Set up your environment:

cd docker\_images

mkdir multi-stage-builds

cd multi-stage-builds

git clone https://github.com/linuxacademy/content-weather-app.git src

Create the Dockerfile:

vi Dockerfile

Dockerfile contents:

# Create an image for the weather-app using multi-stage build

FROM node AS build

RUN mkdir -p /var/node/

ADD src/ /var/node/

WORKDIR /var/node

RUN npm install

FROM node:alpine

ARG VERSION=V1.1

LABEL org.label-schema.version=$VERSION

ENV NODE\_ENV="production"

COPY --from=build /var/node /var/node

WORKDIR /var/node

EXPOSE 3000

ENTRYPOINT ["./bin/www"]

Build the image:

docker image build -t linuxacademy/weather-app:multi-stage-build --rm --build-arg VERSION=1.5 .

List images to see the size difference:

docker image ls

Create the weather-app container:

docker container run -d --name multi-stage-build -p 8087:3000 linuxacademy/weather-app:multi-stage-build

# Tagging

In this lesson, we will talk about how to use the tag command, and best practices to keep in mind when tagging.

Add a name and an optional tag with -t or --tag, in the name:tag format:

docker image build -t <name>:<tag>

docker image build --tag <name>:<tag>

List your images:

docker image ls

Use our Git commit hash as the image tag:

git log -1 --pretty=%H

Use the Docker tag to a create a new tagged image:

docker tag <SOURCE\_IMAGE><:TAG> <TARGET\_IMAGE>:<TAG>

Get the commit hash:

cd docker\_images/weather-app/src

git log -1 --pretty=%H

cd ../

Build the image using the Git hash as the tag:

docker image build -t linuxacademy/weather-app:<GIT\_HASH> .

Tag the weather-app as the latest using the image tagged with the commit hash:

docker image tag linuxacademy/weather-app:<GIT\_HASH> linuxacademy/weather-app:latest

# Distributing Images on Docker Hub

In this lesson, we'll walk through how to tag and push an image to Docker Hub. You will need a Docker Hub account.

Create a Docker Hub account:

https://hub.docker.com/

Docker Push:

docker image push <USERNAME>/<IMAGE\_NAME>:<TAG>

Creating an image for Docker Hub:

docker image tag <IMAGE\_NAME>:<TAG> <USERNAME>/<IMAGE\_NAME>:<TAG>

Set up your environment:

cd docker\_images

mkdir dockerhub

cd dockerhub

git clone https://github.com/linuxacademy/content-weather-app.git src

Create the Dockerfile:

vi Dockerfile

Dockerfile contents:

# Create an image for the weather-app using multi-stage build

FROM node AS build

RUN mkdir -p /var/node/

ADD src/ /var/node/

WORKDIR /var/node

RUN npm install

FROM node:alpine

ARG VERSION=V1.1

LABEL org.label-schema.version=$VERSION

ENV NODE\_ENV="production"

COPY --from=build /var/node /var/node

WORKDIR /var/node

EXPOSE 3000

ENTRYPOINT ["./bin/www"]

Git the weather-app code:

git clone https://github.com/linuxacademy/content-weather-app.git src

Use the Git commit hash as the image tag:

cd src

git log -1 --pretty=%H

cd ../

Build the image:

docker image build -t <USERNAME>/weather-app:<HASH> --build-arg VERSION=1.5 .

Tag the image before pushing it to Docker Hub:

docker image tag linuxacademy/weather-app:<HASH> <USERNAME>/weather-app:<HASH>

Push the image to Docker Hub:

docker login <USERNAME>

docker image push <USERNAME>/weather-app:<HASH>

Tag the latest image:

docker image tag <USERNAME>/weather-app:<HASH> <USERNAME>/weather-app:latest

Push the latest image to Docker Hub:

docker login <USERNAME>

docker image push <USERNAME>/weather-app:latest

# Image History

In this lesson, see how to get more information about an image by looking at its history.

Show the history of an image:

docker image history <IMAGE>

docker image history --no-trunc <IMAGE>

docker image history --quiet <IMAGE>

docker image history --quiet --no-trunc <IMAGE>

Get the image history for Node:

docker image history node:latest

Get the image history for weather-app:

docker image history rivethead42/weather-app:latest

Get the image history weather-app:v1 with the no-truncm flag:

docker image history --no-trunc linuxacademy/weather-app:v1

Save the output using the no-truncm flag to a file:

docker image history --no-trunc linuxacademy/weather-app:v1 > output.txt

View the contents:

vi output.txt

Use the quiet flag to list the image IDs:

docker image history --quiet linuxacademy/weather-app:v1

Use the quiet flag to list the image IDs, then save the output to a file using the no-truncm flag:

docker image history --quiet --no-trunc linuxacademy/weather-app:v1

# Saving and Loading Images

In this lesson, we will learn how to save an image to a tar file, and see how to load it back in.

Save one or more images to a tar file:

docker image save <IMAGE> > <FILE>.tar

docker image save <IMAGE> -o <FILE>.tar

docker image save <IMAGE> --output <FILE>.tar

Load an image from a tar file:

docker image load < <FILE>.tar

docker image load -i <FILE>.tar

docker image load --input <FILE>.tar

Setup:

mkdir output

cd output

Archive the rivethead42/weather-app:latest image:

docker image save rivethead42/weather-app:latest --output weather-app-latest.tar

Inspect the tar file:

tar tvf weather-app-latest.tar

Compress the tar file:

gzip weather-app-latest.tar

Delete the image:

docker image rm [USERNAME]/weather-app:latest

Load the weather-app image from a tar file:

docker image load --input weather-app-latest.tar.gz

docker image ls | grep [USERNAME]/weather-app

docker image rm rivethead42/weather-app:latest docker image ls | grep rivethead42/weather-app

# Inspecting Container Processes

In this lesson, we'll take a look at a few ways we can examine the running processes in a container.

Docker Top:

docker container top <NAME>

Docker Stats:

docker container stats <NAME>

Create a new CentOS container:

docker container run -itd --name container\_process centos /bin/bash

Execute docker container top:

docker container top container\_process

Attach to container\_process:

docker container exec -it container\_process /bin/bash

Attach to the container using attach:

docker container attach container\_process

Restart the container:

docker container start container\_process

Attach to the container\_process container:

docker container exec -it container\_process /bin/bash

Run top on the container:

top

exit

Get stats on a container:

docker container stats container\_process

**Having Containers Start Automatically**

In this lesson, we will look at how to set restart policies for containers, and how that will effect their behavior when the docker service is restarted.

To configure the restart policy for a container, use the --restart flag:

* no: Do not automatically restart the container. (the default)
* on-failure: Restart the container if it exits due to an error, which manifests as a non-zero exit code.
* always: Always restart the container if it stops.
* unless-stopped: Similar to always, except that when the container is stopped, it is not restarted even after the Docker daemon restarts.

Automatically Restarting a container:

docker container run -d --name <NAME> --restart <RESTART> <IMAGE>

Make sure a container always restarts:

docker container run -d --name always-restart --restart always rivethead42/weather-app:latest

Make sure a container restarts unless it's stopped:

docker container run -d --name unless-stopped --restart unless-stopped rivethead42/weather-app:latest

Stop and restart your Docker service:

sudo systemctl restart docker

List your containers:

docker container ls

Stop the *unless-stopped* container:

docker container stop unless-stopped

Stop and restart your Docker service:

sudo systemctl restart docker

List your containers:

docker container ls

Stop the unless-stopped container:

docker container stop always-restart

Stop and restart your Docker service:

sudo systemctl restart docker

List your containers:

docker container ls

# Docker Events

In this lesson, we'll see how to listen for events using the events command.

Get real-time events from the server:

docker system events

docker system events --since '<TIME\_PERIOD>'

Start a new CentOS container:

docker container run -itd --name docker\_events centos /bin/bash

Listen for events:

docker system events

Generate Events:

docker container exec docker\_events /bin/bash

docker container attach docker\_events

docker container start docker\_events

Filters Events:

docker system events --filter <FILTER\_NAME>=<FILTER>

Filter for container events:

docker system events --filter type=container --since '1h'

Generate an event:

docker container exec docker\_events ls /

Filter for container events:

docker system events --filter type=container --filter event=start --since '1h'

List / on docker\_events:

docker container exec docker\_events ls /

Filter for attach events:

docker system events --filter type=container --filter event=attach

Connect to docker\_events using /bin/bash:

docker container exec -it docker\_events /bin/bash

Attach to docker\_events:

docker container attach docker\_events

Connect to docker\_events using /bin/bash:

docker container exec -it docker\_events /bin/bash

Attach to docker\_events:

docker container attach docker\_events

Use multiple filters:

docker system events --filter type=container --filter event=attach --filter event=die --filter event=stop

Start docker\_events:

docker container start docker\_events

Attach to docker\_events:

docker container attach docker\_events

Documentation:

[docker events](https://docs.docker.com/engine/reference/commandline/events/)

[Engine API v1.24](https://docs.docker.com/engine/api/v1.24/)

# Managing Stopped Container

In this lesson, we will manage stopped containers by starting, deleting, or pruning them.

Remove one or more containers:

docker container rm <NAME>

List the rm flags:

docker container rm -h

Start one or more stopped containers:

docker container start <NAME>

Remove all stopped containers:

docker container prune

List the IDs of all containers:

docker container ls -a -q

List all stopped containers:

docker container ls -a -f status=exited

List the IDs of stopped containers:

docker container ls -a -q -f status=exited

Get a count of all stopped containers:

docker container ls -a -q -f status=exited | wc -l

Get a listing of our containers:

docker container ls -a -f status=exited | grep prometheus

Start Prometheus:

docker container start prometheus

Fin stopped weather-app containers with grep:

docker container ls -a -f status=exited | grep weather-app

Remove stopped weather-app containers:

docker container rm [CONTAINER\_IDS]

Prune all stopped containers:

docker container prune

# Managing Docker with Portainer

In this lesson, we'll install Portainer and use it manage our Docker host.

Create a volume for Portainers data:

docker volume create portainer\_data

Create the Portainers container:

docker container run -d --name portainer -p 8080:9000 \

--restart=always \

-v /var/run/docker.sock:/var/run/docker.sock \

-v portainer\_data:/data portainer/portainer

docker container ls

The ls should output:

rivethead42/weather-app:latest

NODE\_ENV production

# Updating Containers with Watchtower

In this lesson, we'll see how to use Watchtower to keep a container up-to-date when its image gets updated.

Clone Express app:

git clone https://github.com/linuxacademy/content-express-demo-app.git watchtower

cd watchtower

git checkout dockerfile

Build the Docker image:

docker login -u [USERNAME]

docker image build -t rivethead42/my-express .

docker image push rivethead42/my-express

Create the container:

docker container run -d --name watched-app -p 80:3000 --restart always rivethead42/my-express

Create Watchtower:

docker container run -d --name watchtower \

--restart always \

-v /var/run/docker.sock:/var/run/docker.sock \

v2tec/watchtower -i 15

Add a .dockerignore file:

vi .dockerignore

.dockerignore contents:

Dockerfile

.git

.gitignore

Edit app.js and add a comment:

vi app.js

app.js contents:

//This is a comment

//

...

Add the file newfile.js:

touch newfile.js

Rebuild the image:

docker image build -t rivethead42/my-express --no-cache .

docker image push rivethead42/my-express

Check to see if the container was restarted with the new image:

docker container ls

Verify the changes by attaching to watched-app:

docker container exec -it watched-app /bin/bash

# Installing Docker Compose

In this lesson, we will learn about installing Docker Compose and why we should use it.

Download the latest version of Docker Compose:

sudo curl -L "https://github.com/docker/compose/releases/download/1.23.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

Apply executable permissions:

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

Test Docker Compose:

docker-compose --version

**Compose Commands**

In this lesson, we will start using compose by creating a compose file. Then we will create and manage the services by using the most commonly used commands:

* build: Build or rebuild services
* bundle: Generate a Docker bundle from the Compose file
* config: Validate and view the Compose file
* create: Create services
* down: Stop and remove containers, networks, images, and volumes
* events: Receive real time events from containers
* exec: Execute a command in a running container
* help: Get help on a command
* images: List images
* kill: Kill containers
* logs: View output from containers
* pause: Pause services
* port: Print the public port for a port binding
* ps: List containers
* pull: Pull service images
* push: Push service images
* restart: Restart services
* rm: Remove stopped containers
* run: Run a one-off command
* scale: Set number of containers for a service
* start: Start services
* stop: Stop services
* top: Display the running processes
* unpause: Unpause services
* up: Create and start containers
* version: Show the Docker-Compose version information

Setup your environment:

mkdir -p compose/commands

cd compose/commands

Create a docker-compose file:

vi docker-compose.yml

docker-compose.yml contents:

version: '3'

services:

web:

image: nginx

ports:

- "8080:80"

volumes:

- nginx\_html:/usr/share/nginx/html/

links:

- redis

redis:

image: redis

volumes:

nginx\_html: {}

Create a compose service:

docker-compose up -d

List containers created by compose:

docker-compose ps

Stopping a compose service:

docker-compose stop

Starting a compose service:

docker-compose start

Restarting a compose service:

docker-compose restart

Delete a compose service:

docker-compose down

# Creating a Compose File

In this lesson we will look at the basics of creating a compose file.

Setup your environment:

cd compose

git clone https://github.com/linuxacademy/content-weather-app.git weather-app

cd weather-app

git checkout compose

Create a docker-compose.yml file:

vi docker-compose.yml

docker-compose.yml contents:

version: '3'

services:

weather-app:

build:

context: .

args:

- VERSION=v2.0

ports:

- "8081:3000"

environment:

- NODE\_ENV=production

Create the compose container:

docker-compose up -d

List compose services:

docker-compose ps

Verify the weather-app is working:

curl http://localhost:8081

Rebuild the image:

docker-compose build

Rebuild the image with no cache:

docker-compose build --no-cache

# Using Volumes and Networking with Compose

In this lesson, we will learn how to use volumes and networks in a docker compose file.

Setup your environment:

mkdir -p compose/ghost

cd compose/ghost

Create a docker-compose.yml file:

vi docker-compose.yml

docker-compose.yml:

version: '3'

services:

ghost:

container\_name: ghost

image: ghost:latest

ports:

- "80:2368"

environment:

- database\_\_client=mysql

- database\_\_connection\_\_host=mysql

- database\_\_connection\_\_user=root

- database\_\_connection\_\_password=P4SSw0rd0!

- database\_\_connection\_\_database=ghost

volumes:

- ghost-volume:/var/lib/ghost

networks:

- ghost\_network

- mysql\_network

depends\_on:

- mysql

mysql:

container\_name: mysql

image: mysql:5.7

environment:

- MYSQL\_ROOT\_PASSWORD=P4SSw0rd0!

volumes:

- mysql-volume:/var/lib/mysql

networks:

- mysql\_network

volumes:

ghost-volume:

mysql-volume:

networks:

ghost\_network:

mysql\_network:

Create the compose container:

docker-compose up -d

List compose services:

docker-compose ps

List the volumes:

docker volumes ls

List the volumes:

docker network ls

[Docker Compose Documentation](https://docs.docker.com/compose/compose-file/)

# Introduction to Docker Swarm

In this lesson, we'll look at the highlights of Docker Swarm before we start working with it.

## Swarm 101

Swarm has two major components:

* An enterprise grade secure cluster:
  + Manage one or more Docker nodes as a cluster
  + Encrypted distributed cluster store
  + Encrypted networks
  + Secure join tokens
* An orchestration engine for creating mircoservices:
  + API for deploying and managing microservices
  + Declarative manifest files for defining apps
  + Provides availability to scale apps, and perform rolling updates and rollbacks

Swarm was initially a separate product layered on Docker, since Docker 1.12 it has become a part of the engine.

## The Cluster

* A swarm consists of one or more Docker nodes.
* Nodes are either a managers or a worker.
* Managers:
  + Manage the state of the cluster
  + Dispatch tasks to workers
* Workers:
  + Accepts and execute tasks
* State is held in etcd
* Swarm uses Transport Layer Security (TLS):
  + Encrypted communication
  + Authenticated nodes
  + Authorized roles

## Orchestration

* The atomic unit of scheduling is a swarm service.
* The service construct adds the following to a container:
  + scaling
  + rolling updates
  + rollback
  + updates
* A container wrapped in a service is a task or a replica.

# Running Docker in Swarm Mode

In this lesson, we will create two new docker servers. These servers will be used in a swarm configuration. Then we will initialize the swarm manager and have the two new nodes join the swarm.

## Install the Swarm Worker Node

Now create two new servers in Cloud Playground that will be used as worker nodes.

## Prerequisites

Uninstall old versions:

sudo yum remove -y docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-engine

## Install Docker CE

Add the Docker repository:

sudo yum install -y yum-utils \

device-mapper-persistent-data \

lvm2

Set up the stable repository:

sudo yum-config-manager \

--add-repo \

https://download.docker.com/linux/centos/docker-ce.repo

Install Docker CE:

sudo yum -y install docker-ce

Enable and Start Docker:

sudo systemctl start docker && sudo systemctl enable docker

Add cloud\_user to the docker group:

sudo usermod -aG docker cloud\_user

Initialize the manager:

docker swarm init \

--advertise-addr [PRIVATE\_IP]

Add the worker to the cluster:

docker swarm join --token [TOKEN] \

[PRIVATE\_IP]:2377

List the nodes in the swarm:

docker node ls

# Managing Swarm Nodes:

In this lesson, we learn how to manage the nodes in the swarm.

Docker node commands:

* demote: Demotes one or more nodes from manager in the swarm
* inspect: Displays detailed information on one or more nodes
* ls: Lists nodes in the swarm
* promote: Promotes one or more nodes to manager in the swarm
* ps: Lists tasks running on one or more nodes, defaults to current node
* rm: Removes one or more nodes from the swarm
* update: Updates a node

Docker swarm commands:

* ca: Displays and rotate the root CA
* init: Initializes a swarm
* join: Joins a swarm as a node and/or manager
* join-token: Manages join tokens
* leave: Leaves the swarm
* unlock: Unlocks swarm
* unlock-key: Manages the unlock key
* update: Updates the swarm

## Managing swarm nodes:

Listing nodes:

docker node ls

Inspecting a node:

docker node inspect [NODE\_NAME]

Promoting a worker to a manager:

docker node promote [NODE\_NAME]

Demoting a manager to a worker:

docker node demote [NODE\_NAME]

Removing a node form the swarm (node must be demoted first):

docker node rm -f [NODE\_NAME]

Make a node leave the swarm:

docker swarm leave

Getting the join-token:

docker swarm join-token [worker|manager]

Make the node rejoin the swarm:

docker swarm join --token [TOKEN] \

<PRIVATE\_IP>:2377

# Working with Services

In this lesson, we'll see how to create and manage a service running in Docker Swarm.

Docker service commands:

* create: Creates a new service
* inspect: Displays detailed information on one or more services
* logs: Fetches the logs of a service or task
* ls: Lists services
* ps: Lists the tasks of one or more services
* rm: Removes one or more services
* rollback: Reverts changes to a service's configuration
* scale: Scales one or multiple replicated services
* update: Updates a service

Creating a service:

docker service create -d --name [NAME] \

-p [HOST\_PORT]:[CONTAINER\_PORT] \

--replicas [REPLICAS] \

[IMAGE] [CMD]

List services:

docker service ls

Inspecting a service:

docker service inspect [NAME]

Get logs for a service:

docker service logs [NAME]

List all tasks of a service:

docker service ps [NAME]

Scale a service up or down:

docker service scale [NAME]=[REPLICAS]

Update a service:

docker service update [OPTIONS] [NAME]

Create nginx\_service:

docker service create -d --name nginx\_service -p 8080:80 --replicas 2 nginx:latest

List the swarm services:

docker service ls

Inspect nginx\_service:

docker service inspect nginx\_service

Find the network:

docker network ls --no-trunc | grep [NETOWRK\_ID]

View the running tasks for nginx\_service:

docker service ps nginx\_service

Scale nginx\_service to 3 replicas:

docker service scale nginx\_service=3

# Using Networks in Swarm Mode

In this lesson, we'll look more into overlay networks and how they are used with a swarm.

Create a overlay network:

docker network create -d overlay [NAME]

Creating a service with an overlay network:

docker service create -d --name [NAME] \

--network [NETWORK] \

-p [HOST\_PORT]:[CONTAINER\_PORT] \

--replicas [REPLICAS] \

[IMAGE] [CMD]

Add a service to a network:

docker service update --network-add [NETWORK] [SERVICE]

Remove a service from a network:

docker service update --network-rm [NETWORK] [SERVICE]

Create a overlay network:

docker network create -d overlay my\_overlay

Create an encrypted overlay network:

docker network create -d overlay --opt encrypted encrypted\_overlay

Inspect encrypted\_overlay:

docker network inspect encrypted\_overlay

Inspect my\_overlay:

docker network inspect my\_overlay

Create a service using my\_overlay:

docker service create -d --name nginx\_overlay --network my\_overlay -p 8081:80 --replicas 2 nginx:latest

Adding the my\_overlay network to nginx\_service:

docker service update --network-add my\_overlay nginx\_service

Inspect nginx\_service:

docker service inspect nginx\_service

Removing the ingress network from nginx\_service:

docker service update --network-rm my\_overlay nginx\_service

Inspect nginx\_service:

docker service inspect nginx\_service

Remove encrypted\_overlay:

docker network rm encrypted\_overlay

# Using Volumes in Swarm Mode

In this lesson we will start learning about plugins, and using volumes in swarm mode. The local driver only creates a volume on the node that a command is executed on. This requires using a third party driver that is specific to the environment.

Add Plugins:

docker plugin install [PLUGIN] [OPTIONS]

List plugins:

docker plugin ls

Volume Plugins:

* Hedvig
* Pure Storage
* HPE Nimble Storage
* Nutanix DVP
* Blockbridge
* NexentaStor
* StorageOS
* Rex-Ray

Install the Splunk plugin:

docker plugin install store/splunk/docker-logging-plugin:2.0.0 --alias splunk-logging-plugin

Disable a plugin:

docker plugin disable [ID]

Remove a plugin:

docker plugin rm [ID]

Digital Ocean example:

docker plugin install rexray/dobs \

DOBS\_REGION=<DO\_REGION> \

DOBS\_TOKEN=<DIGITAL\_OCEAN\_TOKEN> \

DOBS\_CONVERTUNDERSCORES=true

Create a volume using a driver:

docker volume create -d [DRIVER] [NAME]

docker service create -d --name [NAME] \

--mount type=[TYPE],src=[SOURCE],dst=[DESTINATION] \

-p [HOST\_PORT]:[CONTAINER\_PORT] \

--replicas [REPLICAS] \

[IMAGE] [CMD]

Create a volume on the manager:

docker volume create -d local portainer\_data

Create a portainers service that uses a volume:

docker service create \

--name portainer \

--publish 8000:9000 \

--constraint 'node.role == manager' \

--mount type=volume,src=portainer\_data,dst=/data \

--mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \

portainer/portainer \

-H unix:///var/run/docker.sock

[Volume Drivers](https://hub.docker.com/search?q=volume%20plugins&type=plugin&category=volume) [Rex-Ray Volume Driver](https://rexray.readthedocs.io/en/stable/user-guide/schedulers/docker/plug-ins/)

# Deploying Stacks in Docker Swarm

In this lesson, we will learn how to deploy stacks to Docker Swarm using Docker Compose.

Docker stack commands:

* deploy: Deploys a new stack or update an existing stack
* ls: Lists stacks
* ps: Lists the tasks in the stack
* rm: Removes one or more stacks
* services: Lists the services in the stack

Setup environment:

mkdir -p swarm/prometheus

cd swarm/prometheus

Create the prometheus.yml file:

vi prometheus.yml

prometheus.yml contents:

global:

scrape\_interval: 15s

scrape\_timeout: 15s

evaluation\_interval: 15s

scrape\_configs:

- job\_name: prometheus

scrape\_interval: 5s

static\_configs:

- targets:

- prometheus\_main:9090

- job\_name: nodes

scrape\_interval: 5s

static\_configs:

- targets:

- [MANAGER]:9100

- [WORKER1]:9100

- [WORKER2]:9100

- job\_name: cadvisor

scrape\_interval: 5s

static\_configs:

- targets:

- [MANAGER]:8081

- [WORKER1]:8081

- [WORKER2]:8081

Create a compose file:

vi docker-compose.yml

docker-compose.yml contents:

version: '3'

services:

main:

image: prom/prometheus:latest

container\_name: prometheus

ports:

- 8080:9090

command:

- --config.file=/etc/prometheus/prometheus.yml

- --storage.tsdb.path=/prometheus/data

volumes:

- ./prometheus.yml:/etc/prometheus/prometheus.yml:ro

- data:/prometheus/data

depends\_on:

- cadvisor

- node-exporter

cadvisor:

image: google/cadvisor:latest

container\_name: cadvisor

deploy:

mode: global

restart: unless-stopped

ports:

- 8081:8080

volumes:

- /:/rootfs:ro

- /var/run:/var/run:rw

- /sys:/sys:ro

- /var/lib/docker/:/var/lib/docker:ro

node-exporter:

image: prom/node-exporter:latest

container\_name: node-exporter

deploy:

mode: global

restart: unless-stopped

ports:

- 9100:9100

volumes:

- /proc:/host/proc:ro

- /sys:/host/sys:ro

- /:/rootfs:ro

command:

- '--path.procfs=/host/proc'

- '--path.sysfs=/host/sys'

- --collector.filesystem.ignored-mount-points

- "^/(sys|proc|dev|host|etc|rootfs/var/lib/docker/containers|rootfs/var/lib/docker/overlay2|rootfs/run/docker/netns|rootfs/var/lib/docker/aufs)($$|/)"

grafana:

image: grafana/grafana

container\_name: grafana

ports:

- 8082:3000

volumes:

- grafana\_data:/var/lib/grafana

- grafana\_plugins:/var/lib/grafana/plugins

environment:

- GF\_SECURITY\_ADMIN\_PASSWORD=P4ssW0rd0!

depends\_on:

- prometheus

- cadvisor

- node-exporter

volumes:

data:

grafana\_data:

grafana\_plugins:

Deploy the stack:

docker stack deploy --compose-file docker-compose.yml prometheus

List stacks:

docker stack ls

List services:

docker service ls

Fix volume permissions:

sudo chown nfsnobody:nfsnobody -R /var/lib/docker/volumes/prometheus\_data

# Introduction to Docker Security

We'll begin exploring ways to secure Docker by using security features native to both the operating system and Docker itself.

## Docker Security 101

Security is all about layers

Linux security:

* Namespaces
* Control Groups
* Mandatory Access Control (MAC)
* Seccomp

Docker security:

* Docker Swarm
* Docker Content Trust
* Docker Security Scanner
* Docker secrets

## Namespaces

Docker creates a set of namespaces and control groups for the container. Docker containers are an organized collections of namespaces.

* Namespaces provide isolation.
* Each container also gets its own network stack.

Docker on Linux namespaces:

* Process ID (pid)
* network (net)
* Filesystem/mount (mount)
* Inter-process Communication (ipc)
* User (user)
* UTS (uts)

## Control Groups

Control Groups are about setting limits for:

* CPU
* RAM
* Disk I/O

They help to mitigate denial-of-service attacks, and are important on multi-tenant platforms.

## Capabilities

Capabilities turn the binary “root/non-root” dichotomy into a fine-grained access control system. In most cases, containers do not need “real” root privileges at all. This means root within a container has much less privileges than the real root. It also means that even if an intruder manages to escalate to root within a container, it is much harder to do serious damage, or to escalate to the host.

## Mandatory Access Control systems

Two major MAC technologies are:

* SELinux
* AppArmor

## Seccomp

This limits the syscalls a container can make to the host's kernel. All new containers get a default seccomp configured

## Docker Swarm

Swarm Mode:

* Cryptographic node Ids
* Mutual authentication via TLS
* Secure join tokens
* CA configuration with automatic certificate rotation
* Encrypted cluster store
* Encrypted networks

docker swarm update --cert-expiry [INT]h

## Docker Secrets

These store sensitive data like:

* Passwords
* TLS Certificates
* API Keys

Secrets Workflow:

1. A secret is created and posted to the Swarm.
2. The secret is encrypted and stored.
3. A service is created and the secret is attached.
4. Secrets are stored in-flight.
5. The secret is mounted into the container of a service.
6. When the task is complete, the in-memory is torn down.

# Working with Docker Security

In this lesson we will start implementing some of the Docker security practices.

## Seccomp Profile

docker container run --security-opt seccomp=[PROFILE] [IMAGE] [CMD]

Testing Seccomp:

docker container run --rm -it alpine sh

whoami

mount /dev/sda1 /tmp

swapoff -a

Using a custom Seccomp profile:

mkdir -p seccomp/profiles/chmod

cd seccomp/profiles/chmod

wget https://raw.githubusercontent.com/moby/moby/master/profiles/seccomp/default.json

Remove chmod, fchmod and fchmodat from the syscalls whitelist. Syscalls starts at line 52.

Applying the custom Seccomp profile:

docker container run --rm -it --security-opt seccomp=./default.json alpine sh

chmod +r /usr

## Capabilities:

Dropping Capabilities:

docker container run --cap-drop=[CAPABILITY] [IMAGE] [CMD]

Test mknod:

docker container run --rm -it alpine sh

mknod /dev/random2 c 1 8

Disable mknod:

docker container run --rm -it --cap-drop=MKNOD alpine sh

mknod /dev/random2 c 1 8

[Runtime privilege and Linux capabilities](https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities)

## Control Groups

Limiting CPU and memory:

docker container run -it --cpus=[VALUE] --memory=[VALUE][SIZE] --memory-swap [VALUE][SIZE] [IMAGE] [CMD]

Setting memory limits on a container:

docker container run -d --name resource-limits --cpus=".5" --memory=512M --memory-swap=1G rivethead42/weather-app

Inspect resource-limits:

docker container inspect resource-limits

[Runtime constraints on resources](https://docs.docker.com/engine/reference/run/#runtime-constraints-on-resources) [More info on resource constraints](https://docs.docker.com/config/containers/resource_constraints/)

# Running Docker Bench for Security

Running Docker Bench Security:

docker container run --rm -it --network host --pid host --userns host --cap-add audit\_control \

-e DOCKER\_CONTENT\_TRUST=$DOCKER\_CONTENT\_TRUST \

-v /var/lib:/var/lib \

-v /var/run/docker.sock:/var/run/docker.sock \

-v /usr/lib/systemd:/usr/lib/systemd \

-v /etc:/etc --label docker\_bench\_security \

docker/docker-bench-security

[Docker Bench Security](https://github.com/docker/docker-bench-security)

# Docker Content Trust

In this lesson you will learn how to sign images and enable Docker Content Trust to ensure that the images you are pulling have been verified.

## Commands

Creating a Key:

docker trust key generate [NAME]

Importing a Key:

docker trust key load [PEM] --name [NAME]

Add a signer:

docker trust signer add --key [PEM] [NAME] [REPOSITORY]

Remove a signer:

docker trust signer remove [NAME] [REPOSITORY]

Signing an image:

docker trust sign [IMAGE]:[TAG]

## Using DCT to sign an image

Tag the image that will be signed:

docker image tag [USERNAME]/weather-app:latest [USERNAME]/dct:latest

Create a Key:

docker trust key generate [NAME]

Add your signer user:

docker trust signer add --key [NAME].pub [NAME] [USERNAME]/dct

Sign and push your image to Docker Hub:

docker trust sign [USERNAME]/dct:

export DOCKER\_CONTENT\_TRUST=1

docker image push [USERNAME]/dct:latest

Remove the Docker image [USERNAME]/weather-app:

docker image rm [USERNAME]/weather-app:latest

Pull [USERNAME]/weather-app:

docker image pull [USERNAME]/weather-app

Remove the Nginx image:

docker image rm nginx:latest

Pull the image:

docker image pull nginx:latest

## Enabling DCT

vi /etc/docker/daemon.json

/etc/docker/daemon.json:

{

"content-trust": {

"mode": "enforced"

}

}

# Working with Secrets

In this lesson, we will start working with Docker Secrets to protect sensitive data, such as passwords and API Keys.

Docker secrets commands:

* create: Create a secret from a file or STDIN as content
* inspect: Display detailed information on one or more secrets
* ls: List secrets
* rm: Remove one or more secrets

Creating a secret:

STDIN | docker secret create [NAME] -

List secrets:

docker secret ls

Inspecting a secret:

docker secret inspect [NAME]

Using secrets:

docker service create --name [NAME] --secret [SECERT] [IMAGE]

Deleting a secret:

docker secret rm [NAME]

Setup environment:

mkdir Secrets

cd secrets

Create a secret using STDIN:

openssl rand -base64 20 | docker secret create my\_secret\_data -

Create a secret using a file:

openssl rand -base64 20 > secret.txt

docker secret create my\_secret\_data2 secret.txt

Create a service using a secret:

docker service create --name redis --secret my\_secret\_data redis:alpine

Find the node the service is running on:

docker service ps redis

Remove secret my\_secret\_data2:

docker secret rm my\_secret\_data2

Generate password files:

openssl rand -base64 20 > db\_password.txt

openssl rand -base64 20 > db\_root\_password.txt

Create a Wordpress Stack:

vi docker-compose.yml

docker-compose.yml contents:

version: '3.1'

services:

db:

image: mysql:5.7

volumes:

- db\_data:/var/lib/mysql

networks:

mysql\_internal:

aliases: ["db"]

environment:

MYSQL\_ROOT\_PASSWORD\_FILE: /run/secrets/db\_root\_password

MYSQL\_DATABASE: wordpress

MYSQL\_USER: wordpress

MYSQL\_PASSWORD\_FILE: /run/secrets/db\_password

secrets:

- db\_root\_password

- db\_password

wordpress:

depends\_on:

- db

image: wordpress:latest

networks:

mysql\_internal:

aliases: ["wordpress"]

wordpress\_public:

ports:

- "8001:80"

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: wordpress

WORDPRESS\_DB\_PASSWORD\_FILE: /run/secrets/db\_password

secrets:

- db\_password

secrets:

db\_password:

file: db\_password.txt

db\_root\_password:

file: db\_root\_password.txt

volumes:

db\_data:

networks:

mysql\_internal:

driver: "overlay"

internal: true

wordpress\_public:

driver: "overlay"

Deploy stack:

docker stack deploy --compose-file docker-compose.yml wp

**Updating Containers with Watchtower**

In this lesson, we'll see how to use Watchtower to keep a container up-to-date when its image gets updated.

Clone Express app:

git clone https://github.com/linuxacademy/content-express-demo-app.git watchtower

cd watchtower

git checkout dockerfile

Build the Docker image:

docker login -u [USERNAME]

docker image build -t rivethead42/my-express .

docker image push rivethead42/my-express

Create the container:

docker container run -d --name watched-app -p 80:3000 --restart always rivethead42/my-express

Create Watchtower:

docker container run -d --name watchtower \

--restart always \

-v /var/run/docker.sock:/var/run/docker.sock \

v2tec/watchtower -i 15

Add a .dockerignore file:

vi .dockerignore

.dockerignore contents:

Dockerfile

.git

.gitignore

Edit app.js and add a comment:

vi app.js

app.js contents:

//This is a comment

//

...

Add the file newfile.js:

touch newfile.js

Rebuild the image:

docker image build -t rivethead42/my-express --no-cache .

docker image push rivethead42/my-express

Check to see if the container was restarted with the new image:

docker container ls

Verify the changes by attaching to watched-app:

docker container exec -it watched-app /bin/bash