Docker

Docker Community Edition

Installation and Configuration

Installing Docker on CentOS

Documentation:

- **Docker CE for CentOS Setup Requirements**

Here is a basic guide that covers on how to install Docker CE on CentOS 7:

1. Install the required packages:

```
 sudo yum install -y  () \
device-mapper-persistent-data \
lvm2
```

2. Add the Docker CE `yum` repository:

```
sudo yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

3. Install the Docker CE packages:

```
sudo yum install -y docker-ce-18.09.5 docker-ce-cli-18.09.5 containerd.io
```

4. Start and enable the Docker service:

```
sudo systemctl start docker
sudo systemctl enable docker
```

To grant a user permission to run Docker commands, add the user to the Docker group. The user will have access to Docker after their next login.

```
sudo usermod -a -G docker <user>
```

We can test our Docker installation by running a simple container. This container should output some text, and then exit.

```
docker run hello-world
```

Installing Docker on Ubuntu

Documentation:

- **Docker CE Install Link**

Here is a basic guide to installing Docker CE on Ubuntu:

1. Install the required packages:

```
sudo apt-get update

sudo apt-get -y install \
  apt-transport-https \
  ca-certificates \
  curl \
  gnupg-agent \
  software-properties-common
```

2. Add the Docker repo's GNU Privacy Guard (GPG) key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

It's a good idea to verify the key fingerprint. This is an optional step, but highly recommended. We should receive an output indicating that the key was found:

```
sudo apt-key fingerprint 0EBFCD88
```

1. Add the Docker Ubuntu repository:

```
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

2. Install packages:

```
sudo apt-get update

sudo apt-get install -y docker-ce=5:18.09.5~3-0~ubuntu-bionic \
docker-ce-cli=5:18.09.5~3-0~ubuntu-bionic containerd.io
```

3. To provide a user with permission to run Docker commands, add the user to the Docker group. The user will have access to Docker after their next login.

```
sudo usermod -a -G docker <user>
```

4. We can test our Docker installation by running a simple container. This container should output some text, and then exit.

```
docker run hello-world
```

Selecting a Storage Driver

Documentation:

- **Docker Storage Drivers Overview Link**

Storage Driver Basics

Storage Driver: A pluggable driver that handles internal storage for containers.

Currently, the default driver for CentOS and Ubuntu systems is `overlay2`.

The `devicemapper` storage driver is sometimes used on CentOS/RedHat systems, especially in older Docker versions. We can determine the current storage driver with `docker info`:

```
    docker info | grep "Storage"
```

Using a Daemon Flag to Set the Storage Driver

One way to select a different storage driver is to pass the `--storage-driver` flag over to the Docker daemon. For example, we can modify Docker's `systemd` unit file: `/usr/lib/systemd/system/docker.service`.

Remember, add the flag `--storage-driver <driver name>` to the call to `dockerd`.

Using the Daemon Config File to Set the Storage Driver

Note: This is the recommended method for setting the storage driver.

1. Create or edit `/etc/docker/daemon.json`:

```
sudo vi /etc/docker/daemon.json
```

2. Add the value `"storage-driver": "<driver name>"`:

This example sets the storage driver to `devicemapper`.

```
{
  "storage-driver": "devicemapper"
}
```

1. After any changes are made to `/etc/docker/daemon.json`, remember to restart Docker. It is also a good idea to check the status of Docker after restarting, as a malformed config file will cause Docker to encounter startup failure. Use the following commands:

```
sudo systemctl restart docker
sudo systemctl status docker
```

Running a Container

Documentation:

- **Docker Run Reference Link**

Docker Run

Here are some key commands and processes:

`docker run IMAGE[:TAG] [COMMAND] [ARGS]` : Runs a container.

- **IMAGE**: Run a container using an image called `hello-world`. In this example, the tag is unspecified, so the latest tag will automatically be used.

```
docker run hello-world
```

- **COMMAND and ARGS**: Run a command inside the container. This command runs a container using the `busybox` image. Inside the container, plus it runs the command `echo` with the arguments `hello world!`.

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

- **TAG**: This command specifies a certain tag, running a container with tag `1.15.11` of the `nginx` image.

```
docker run nginx:1.15.11
```

- **-d**: Runs the container in detached mode. The command immediately exits, and the container continues to run in the background.

- **--name NAME**: Gives the container a specified name instead of the usual randomly-assigned name.

- **--restart RESTART**: Specifies when Docker should automatically restart the container. Valid

values include:

- **no** (default): Never restart the container.

- **on-failure**: Only if the container fails (exits with a non-zero exit code).

- **always**: Always restart the container whether it succeeds or fails. Also starts the container automatically upon daemon startup.
- **unless-stopped**: Always restart the container whether it succeeds or fails, and on daemon startup unless the container is manually stopped.
- **-p HOST_PORT:CONTAINER_PORT**: Publish a container's port. This process is necessary to access a port on a running container. The `HOST_PORT` is the port that listens on the host machine, and traffic to that port is mapped to the `CONTAINER_PORT` on the container.
- **--memory MEMORY**: Set a hard limit on memory usage.
- **--memory-reservation MEMORY**: Set a soft limit on memory usage. This limit is used only if Docker detects memory contention on the host.

Here's an example of these `docker run` flags in action:

```
docker run -d --name nginx --restart unless-stopped -p 8080:80 --memory 500M \
--memory-reservation 256M nginx
```

Managing Containers

Some of the commands for managing running containers:

- `docker ps`: List running containers.
- `docker ps -a`: List all containers, including stopped containers.
- `docker container stop <container name or ID>`: Stop a running container.
- `docker container start <container name or ID>`: Start a stopped container.
- `docker container rm <container name or ID>`: Delete a container (must be stopped first).

Upgrading the Docker Engine

Documentation:

- **Upgrade Docker CE Link**

1. To upgrade the Docker engine, first we must stop the Docker service and install newer versions of `docker-ce` and `docker-ce-cli`.

```
sudo systemctl stop docker
sudo apt-get install -y docker-ce=<new version> docker-ce-cli=<new version>
```

2. Check the current version:

```
docker version
```

Configuring Logging Drivers (Splunk, Journald, etc.)

Documentation:

- **Logging Drivers Configuration Link**

Logging driver: A pluggable driver that handles log data from services and containers in Docker.

Determine the current default logging driver:

```
docker info | grep Logging
```

To set a new default driver, modify `/etc/docker/daemon.json` . The `log-driver` option sets the driver, and `log-opts` can be used to provide driver-specific configuration.

For example:

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3",
    "labels": "production_status",
    "env": "os,customer"
  }
}
```

Remember to utilize `sudo systemctl restart docker` after making any changes to `/etc/docker/daemon.json.`

We can also override the default driver setting for individual containers using the `--log-driver' and '--log-opt` flags with `docker run.`

```
      docker run --log-driver json-file --log-opt max-size=10m nginx
```

Namespaces and Cgroups

Documentation:

- **Docker Overview Link**
- **Namescape Remapping Link**

Namespaces: A Linux related technology that isolates processes by partitioning the resources that are available to them. Namespaces prevent processes from interfering with one another. Docker leverages namespaces to isolate resources for containers.

Some namespaces used by Docker:

- **pid**: Process isolation.
- **net**: Network interfaces.
- **ipc**: Inter-process communication.
- **mnt**: Filesystem mounts.
- **uts**: Kernel and version identifiers.
- **user namespaces**: Requires special configuration. Allows container processes to run as root inside the container while mapping that user to an unprivileged user on the host.

Control Groups (cgroups): Control groups limit processes to a specific set of resources. Docker uses `cgroups` to enforce rules around resource usage by containers, such as limiting memory or CPU usage.

Image Creation, Management, and Registry

Introduction to Docker Images

Documentation:

- **Images, Containers, and Storage Drivers Link**

Image: An executable package containing all of the software that's needed to run a container. Run a container using an image with:

```
docker run IMAGE
```

Download an image with:

```
docker image pull IMAGE
docker pull IMAGE
```

Layered File System: Images and containers use a layered file system. Each layer contains only the differences from the previous layer.

View file system layers in an image with:

```
docker image history IMAGE
```

The Components of a Dockerfile

Documentation:

- **Dockerfile Builds Reference Link**

Dockerfile: A file that defines a series of directives and is used to build an image. An example of a `Dockerfile`:

```
# Simple nginx image
FROM ubuntu:bionic

ENV NGINX_VERSION 1.14.0-0ubuntu1.2

RUN apt-get update && apt-get install -y curl
RUN apt-get update && apt-get install -y nginx=$NGINX_VERSION

CMD ["nginx", "-g", "daemon off;"]
```

Build an image with:

```
docker build -t TAG_NAME DOCKERFILE_LOCATION
```

Dockerfile Directives:

- `FROM`: Specifies the base image to build from.
- `ENV`: Sets environment variables that are visible in later build steps as well as during container runtime.
- `RUN`: Executes a command and commits the result to the image file system.

- `CMD`: Sets the default command for containers, and this gets overridden if a command gets specified at container runtime.
- `ENTRYPOINT` : Sets the default executable for containers. This can still be overridden at container runtime, but requires a special flag. When `ENTRYPOINT` and `CMD` are both used, `ENTRYPOINT` sets the default executable, and `CMD` sets default arguments.

More Dockerfile Directives

Documentation:

- **Dockerfile Builds Reference Link**

Directives:

- `EXPOSE`: Documents ports that are intended to be published at runtime.

  Note: that this does not actually publish the ports.

- `WORKDIR`: Sets the working directory, both for subsequent build steps and for the container at runtime. We can use `WORKDIR` multiple times. If the begins with a forward slash `/` , then it will set an absolute path. Otherwise, it will set the working directory relative to the previous working directory.
- `COPY`: Copies files from the build host into the image file system.
- `ADD`: Copies files from the build host into the image file system. Unlike `COPY`, `ADD` can also extract an archive into the image file system and add files from a remote URL.
- `STOPSIGNAL`: Sets a custom signal that will be used to stop the container process.
- `HEALTHCHECK`: Sets a command that will be used by the Docker daemon to check whether the container is healthy.

Building Efficient Images

Documentation:

- **Best Practices for Dockerfiles Link**
- **Using Multi-stage Builds Link**

Multi-Stage Build: A build from a Dockerfile with more than one the `FROM` directive. It is used to selectively copy files into final stage, keeping the resulting image as small as possible.

Managing Images

Documentation:

- **Docker Image Commands Link**

Here are some key commands for image management:

- `docker image ls`: List images on the system.
- `docker image ls -a`: Includes intermediate images.
- `docker image inspect IMAGE`: Get detailed information about an image.
- `docker image inspect IMAGE --format "GO_TEMPLATE"`: Provide a Go template to retrieve specific data fields about the image.
- `docker image rm IMAGE`: Delete an image. An image can only face deletion if no containers or other image tags reference it.
- `docker rmi IMAGE`: Delete an image.
- `docker image rm -f IMAGE`: Force deletion of an image, even if gets referenced by something else.

- `docker image prune`: Find and delete dangling or unused images.

Flattening a Docker Image to a Single Layer

Docker does not provide an official method for turning a multi-layered image into a single layer. We can work around this by running a container from the image, exporting the container's file system, and then importing that data as a new image.

1. Set up a new project directory to create a basic image:

```
cd ~/
mkdir alpine-hello
cd alpine-hello
vi Dockerfile
```

2. Create a Dockerfile that will result in a multi-layered image:

```
FROM alpine:3.9.3
RUN echo "Hello, World!" > message.txt
CMD cat message.txt
```

3. Build the image and check how many layers it has:

```
docker build -t nonflat .
docker image history nonflat
```

4. Run a container from the image and export its file system to an archive:

```
docker run -d --name flat_container nonflat
docker export flat_container > flat.tar
```

5. Import the archive to a new image and check how many layers the new image has:

```
cat flat.tar | docker import - flat:latest
docker image history flat
```

Introduction to Docker Registries

Documentation:

- **Registry Deployment Link**
- **Registry Configuration Link**
- **Insecure Registry Testing Link**

Docker Registry: A central location for storing and distributing images.

Docker Hub: The default, public registry that's operated by Docker.

We can operate our own private registry for free using the `registry` image. Run

a simple registry with:

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

Configure our registry using environment variables with:

```
docker run -d -p 5000:5000 --restart=always --name registry \
-e REGISTRY_LOG_LEVEL=debug registry:2
```

Using Docker Registries

Documentation:

- **Docker Push Commands Link**

- **Docker Pull Commands Link**
- **Docker Login Commands Link**
- **Insecure Registry Testing Link**
- **Docker Search Commands Link**

We can search Docker Hub from the command line with:

```
docker search SEARCH_TERM
```

Authenticate against a registry. We can omit the `REGISTRY_URL` to authenticate with Docker Hub:

```
docker login REGISTRY_URL
```

There are two ways to authenticate with a private registry that uses an untrusted or self-signed certificate.

- Secure: Adds the registry's public certificate to `/etc/docker/certs.d/<registry public hostname>.`
- Insecure: Adds the registry to the `insecure-registries` list in `daemon.json` , or pass it to `dockerd` with the `--insecure-registry` flag.

Push an image to a registry:

```
docker push IMAGE
```

Introduction to Docker Services

Documentation:

- **Swarm Service Deployment Link**
- **Services Overview Link**
- **Service Creation Link**

Service: A collection of one or more replica containers running the same image in a swarm.

Task: A replica container that is running as part of a service.

Here are some common tasks and the commands that are associated with them:

1. Create a service with:

```
docker service create IMAGE
```

2. To provide a name for a service, we can use:

```
docker service create --name NAME IMAGE
```

3. Set the number of replicas with:

```
docker service create --replicas REPLICAS IMAGE
```

4. Publish a port for the service. By default, the port will listen on all nodes in the cluster (workers and managers). Requests can be routed to a container on any node, regardless of which node is accessed by the client. The format will look like this:

```
docker service create -p 8080:80 IMAGE
```

Note: We can use templates to pass dynamic data to certain flags when creating a service, for instance this example sets an environment variable containing the node hostname for each replica container.

```
docker service create --name node-hostname --replicas 3 --env
NODE_HOSTNAME="{{.Node.Hostname}}" \
nginx
```

1. We can list services in the cluster with:

```
docker service ls
```

2. We can list the tasks/containers for a service with:

```
docker service ps SERVICE
```

3. To inspect a service:

```
docker service inspect SERVICE
docker service inspect --pretty SERVICE
```

4. To change a service:

```
docker service update --replicas 2 SERVICE
```

5. There are two different ways to change the number of replicas for a service:

```
docker service update --replicas 2 SERVICE
docker service scale SERVICE=REPLICAS
```

6. Delete a service with:

```
docker service rm SERVICE
```

7. We can create global services. Instead of running a specific number of replicas, they run exactly one task on each node in the cluster. The command appears as:

```
docker service create --mode global IMAGE
```

Using docker inspect

Documentation:

- **Swarm Service Deployment Link**
- **Services Overview Link**
- **Service Creation and Commands Link**

Docker inspect provides a way to get detailed information about Docker objects. We can use the general form `docker inspect OBJECT` or an object-type-specific form `docker container inspect CONTAINER`.

Use the `--format` flag with Docker inspect to retrieve specific data fields using a Go template:

```
docker service inspect --format='{{.ID}}' SERVICE
```

Docker Compose

Documentation:

- **Docker Compose Link**

Docker Compose: A tool used to manage complex, multi-container applications running on a single host. To

use Docker Compose, first define the application in a compose file.

An example of a simple `docker-compose.yml`:

```
version: '3'
services:
  web:
    image: nginx
    ports:
    - "8080:80"
  redis:
    image: redis:alpine
```

Run an application using a compose file from the directory where the file is located:

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

List compose applications:

```
docker-compose ps
```

Stop a Docker compose application from the directory where the compose file is located:

```
docker-compose down
```

Introduction to Docker Stacks

Documentation:

- **Stacks and Prerequisites Link**
- **Docker Stack Commands Link**

Stack: A complex, multi-service application running in a swarm.

We can define a stack using a Docker Compose file, and then run it in the swarm with:

```
docker stack deploy -c COMPOSE_FILE STACK_NAME
```

Remember that we can redeploy the stack with the same compose file to make changes to it.

Delete a stack with:

```
docker stack rm STACK
```

Node Labels

Documentation:

- **Docker Node Update Link**
- **Placement Constraints Link**

Node Label: Custom metadata about a node in the cluster. To

list current nodes, enter:

```
docker node ls
```

To add a label to a node, input:

```
docker node update --label-add LABEL_NAME=LABEL_VALUE NODE
```

To view existing labels, run:

```
docker node inspect --pretty NODE
```

We can use node constraints to control which nodes a service's tasks will run on based upon node labels. Here is an example:

```
docker service create --constraint node.labels.availability_zone==east nginx
```

To use various expressions for constraints, such as inequality, we can run, for instance:

```
--constraint node.labels.availability_zone!=east
```

Use `--placement-pref` to spread tasks evenly based on the value of a specific label:

```
docker service create --placement-pref spread=node.labels.availability_zone --replicas 3
nginx
```

Storage and Volumes

Docker Storage in Depth

Documentation:

- **Docker Storage Drivers Link**
- **Object Storage and File Systems Link**

Here are what default storage drivers consist of in terms of systems:

- Latest versions of Ubuntu and CentOS — `overlay2`
- CentOS 7 and earlier — `devicemapper`
- Ubuntu 14.04 and earlier - `aufs`

Storage Models

Filesystem Storage:

- Data is stored in the form of regular files on the host disk.
- Used by `overlay2` and `aufs`.
- Efficient use of memory.
- Inefficient with write-heavy workloads.

- Block Storage:
  - Stores data in blocks using special block storage devices.
  - Used by `devicemapper`.
  - Efficient with write-heavy workloads.
- Object Storage:
  - Stores data in an external object-based store.
  - Application must be designed to use object-based storage.
  - Flexible and scalable.

We can inspect containers and images to locate the actual location of their data files on disk.

Configuring the Device Mapper

Documentation:

- **Device Mapper Storage Driver**

`devicemapper`: A block storage driver used by CentOS 7 and earlier.

Uses two modes:

- `loop-lvm`: This is the default mode, but it is recommended for testing only, not for production use.
- `direct-lvm`: A production-ready mode, which requires additional configuration and a special block storage device.

Below is sample configuration to enable `direct-lvm` in `daemon.json`.

```
Remember, this assumes that there is a block storage device called `/dev/xvdb`.
```

```
{
  "storage-driver": "devicemapper",
  "storage-opts": [
    "dm.directlvm_device=/dev/xvdb",
    "dm.thinp_percent=95",
    "dm.thinp_metapercent=1",
    "dm.thinp_autoextend_threshold=80",
    "dm.thinp_autoextend_percent=20",
    "dm.directlvm_device_force=true"
  ]
}
```

Docker Volumes

Documentation:

- **Managing Data Storage Link**
- **Using Bind Mounts Link**
- **Using Volumes Link**

There are two different types of data mounts on Docker:

- Bind Mount: Mounts a specific directory on the host to the container. It is useful for sharing configuration files, plus other data between the container and host.

- Named Volume: Mounts a directory to the container, but Docker controls the location of the volume on disk dynamically.

There are different syntaxes for adding bind mounts or volumes to containers:

-v syntax

A bind mount. The fact that the source begins with a forward slash `/` makes this a bind mount.

```
docker run -v /opt/data:/tmp nginx
```

A named volume. The fact that the source is just a string means that this is a volume. If no volume exists with the provided name, then it will be automatically created.

```
docker run -v my-vol:/tmp nginx
```

--mount syntax

A bind mount:

```
docker run --mount source=/opt/data,destination=/tmp nginx
```

A named volume:

```
docker run --mount source=my-vol,destination=/tmp nginx
```

We can mount the same volume to multiple containers, allowing them to share data.

We can also create and manage volumes by themselves without running a container.

Here are some common and useful commands:

- `docker volume create VOLUME`: Creates a volume.
- `docker volume ls`: Lists volumes.
- `docker volume inspect VOLUME`: Inspects a volume.

- `docker volume rm VOLUME`: Deletes a volume.

Image Cleanup

Documentation:

- **Docker System DF Link**

To display Docker's disk usage on a system, enter:

```
docker system df
```

To display a more detailed disk usage report, input:

```
docker system df -v
```

To delete dangling/unused images, run:

```
docker image prune
```

Once we have verified that the images are not being used by a container, we can use the following command to delete all of them:

```
docker image prune -a
```

Networking

Docker Networking

Documentation:

- **Docker Container Networks Link**

Docker Container Networking Model (CNM): A conceptual model that describes the components and concepts of Docker networking.

There are multiple implementations of the Docker CNM:

- Sandbox: An isolated unit containing all networking components associated with a single container.
- Endpoint: Connects one sandbox to one network.
- Network: A collection of endpoints that can communicate with each other.
- Network Driver: A pluggable driver that provides a specific implementation of the CNM.
- IPAM Driver: Provides IP Address management. Allocates and assigns IP addresses.

Built-In Network Drivers

Documentation:

- **Docker Network Drivers Overview Link**
- **Use Cases for Docker Network Drivers**

Native Network Drivers: Network drivers that come shipped with Docker.

Host

This driver connects the container directly to the host's networking stack. It provides no isolation between containers or between containers and the host.

```
docker run --net host nginx
```

Bridge

This driver uses virtual bridge interfaces to establish connections between containers running on the same host.

```
docker network create --driver bridge my-bridge-net
docker run -d --network my-bridge-net nginx
```

Overlay

This driver uses a routing mesh to connect containers across multiple Docker hosts, usually in a Docker swarm.

```
docker network create --driver overlay my-overlay-net
docker service create --network my-overlay-net nginx
```

MACVLAN

This driver connect containers directly to the host's network interfaces, but uses special configuration to provide isolation.

```
docker network create -d macvlan --subnet 192.168.0.0/24 --gateway 192.168.0.1 -o
parent=eth0 \
my-macvlan-net
docker run -d --net my-macvlan-net nginx
```

None

This driver provides sandbox isolation, but it does not provide any implementation for networking between containers or between containers and the host.

```
docker run --net none -d nginx
```

Creating a Docker Bridge Network

Documentation:

- **Using Bridge Networks Link**

Bridge is the default driver, so any network that is created without specifying the driver will be a bridge network.

Create a bridge network:

```
docker network create my-net
```

Run a container on the bridge network:

```
docker run -d --network my-net nginx
```

By default, container and services on the same network can communicate with each other simply using their container or service names. Docker provides DNS resolution on the network that allows this to work. We can supply a network alias to provide an additional name by which a container or service is reached.

```
docker run -d --network my-net --network-alias my-nginx-alias nginx
```

Here are some useful commands for when one must interact with Docker networks:

- `docker network ls`: Lists networks.
- `docker network inspect NETWORK`: Inspects a network.
- `docker network connect CONTAINER NETWORK`: Connects a container to a network.
- `docker network disconnect CONTAINER NETWORK`: Disconnects a container from a network.
- `docker network rm NETWORK`: Deletes a network.

Deploying a Service on a Docker Overlay Network

Documentation:

- **Using Overlay Networks Link**

To create an overlay network, run:

```
docker network create --driver overlay NETWORK_NAME
```

To create a service that uses the network, enter:

```
docker service create --network NETWORK_NAME IMAGE
```

Exposing Containers Externally

Documentation:

- **Docker Run Commands Link**

- **Docker Service Command Options Link**
- **Docker Port Commands Link**

Publish a port on a container. The host port is the port that will listen on the host. Requests to that port on the host will be forwarded to the CONTAINER_PORT inside the container.

```
docker run -d -p HOST_PORT:CONTAINER_PORT IMAGE
```

`-P`, `--publish-all`: Publish all ports documented using `EXPOSE` for the image. These ports are all published to random ports.

We can also publish ports for services. By default, the routing mesh causes the host port to listen on all nodes in the cluster. Requests can be forwarded to any task, even if they are on another node.

```
docker service create -p HOST_PORT:CONTAINER_PORT IMAGE
```

We can publish service ports in host mode. This mode is much more restrictive. It does not use a routing mesh. Requests to the host port on a host are forwarded to the task running on that same host. Therefore, in this mode, we cannot have more than one task for the service per host, and hosts that are not running a task for the service won't listen on the host port.

```
docker service create -p mode=host,published=HOST_PORT,target=CONTAINER_PORT IMAGE
```

Network Troubleshooting

Documentation:

- **Troubleshooting Container Networking Link**

To view container logs, enter:

```
docker logs CONTAINER
```

To view logs for all tasks of a service, execute:

```
docker service logs SERVICE
```

To view Docker daemon logs, input:

```
sudo jounralctl -u docker
```

We can use the `nicolaka/netshoot` image to perform network troubleshooting. It comes packaged with a variety of useful networking-related tools.

We can inject a container into another container's networking sandbox for troubleshooting purposes:

```
docker run --network container:CONTAINER_NAME nicolaka/netshoot
```

Configuring Docker to Use External DNS

Documentation:

- **Container DNS Configuration Link**
- **Daemon DNS Options Link**

We can customize the DNS server that will be used by our containers. Set

the system-wide default DNS for Docker containers in `daemon.json`:

```
{
  "dns": ["8.8.8.8"]
}
```

Set the DNS for an individual container.

```
docker run --dns 8.8.4.4 IMAGE
```