Internet of Things and Services

Service-oriented architectures

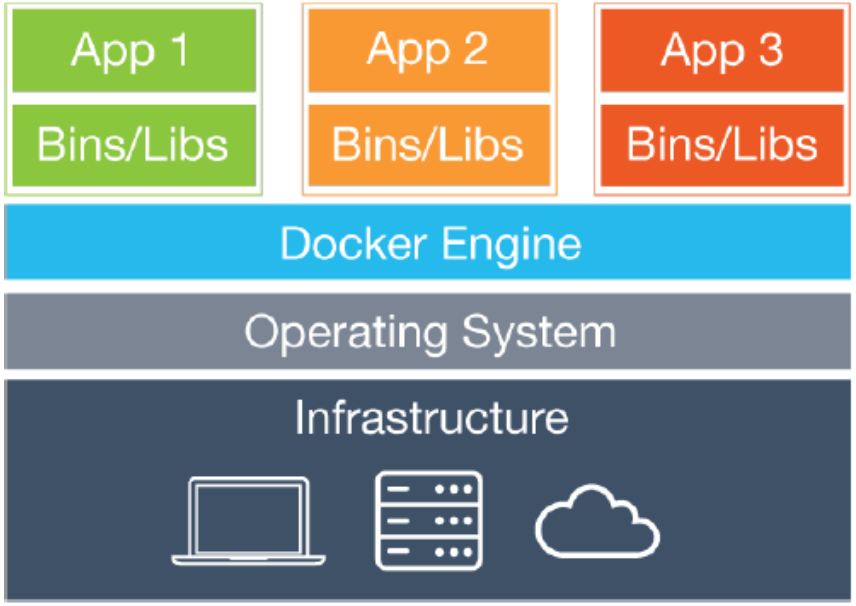
Docker

Container-based virtualization

Department of Computer Science
Faculty of Electronic Engineering, University of Nis

Docker

- Lightweight, open and secure container-based virtualization
  - Containers include the application and all of its dependencies, but share the OS kernel with other containers
  - Containers run as an isolated process in user space on the host OS
  - Containers are also not tied to any specific infrastructure

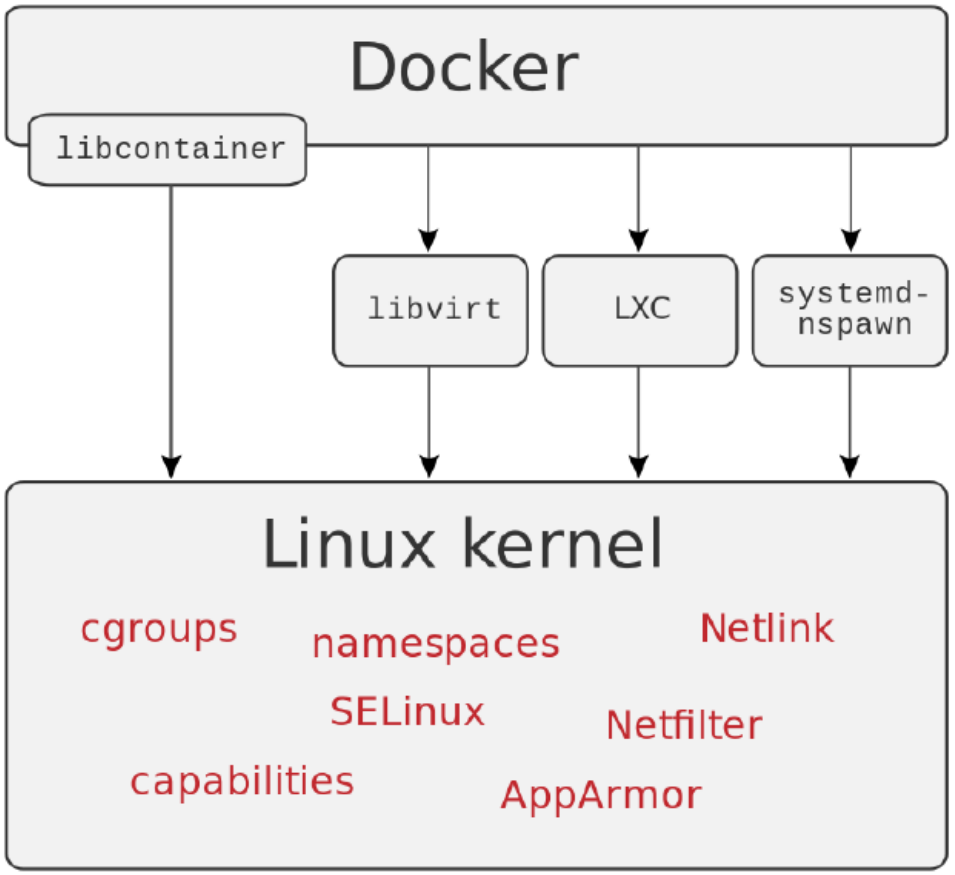


Docker internals

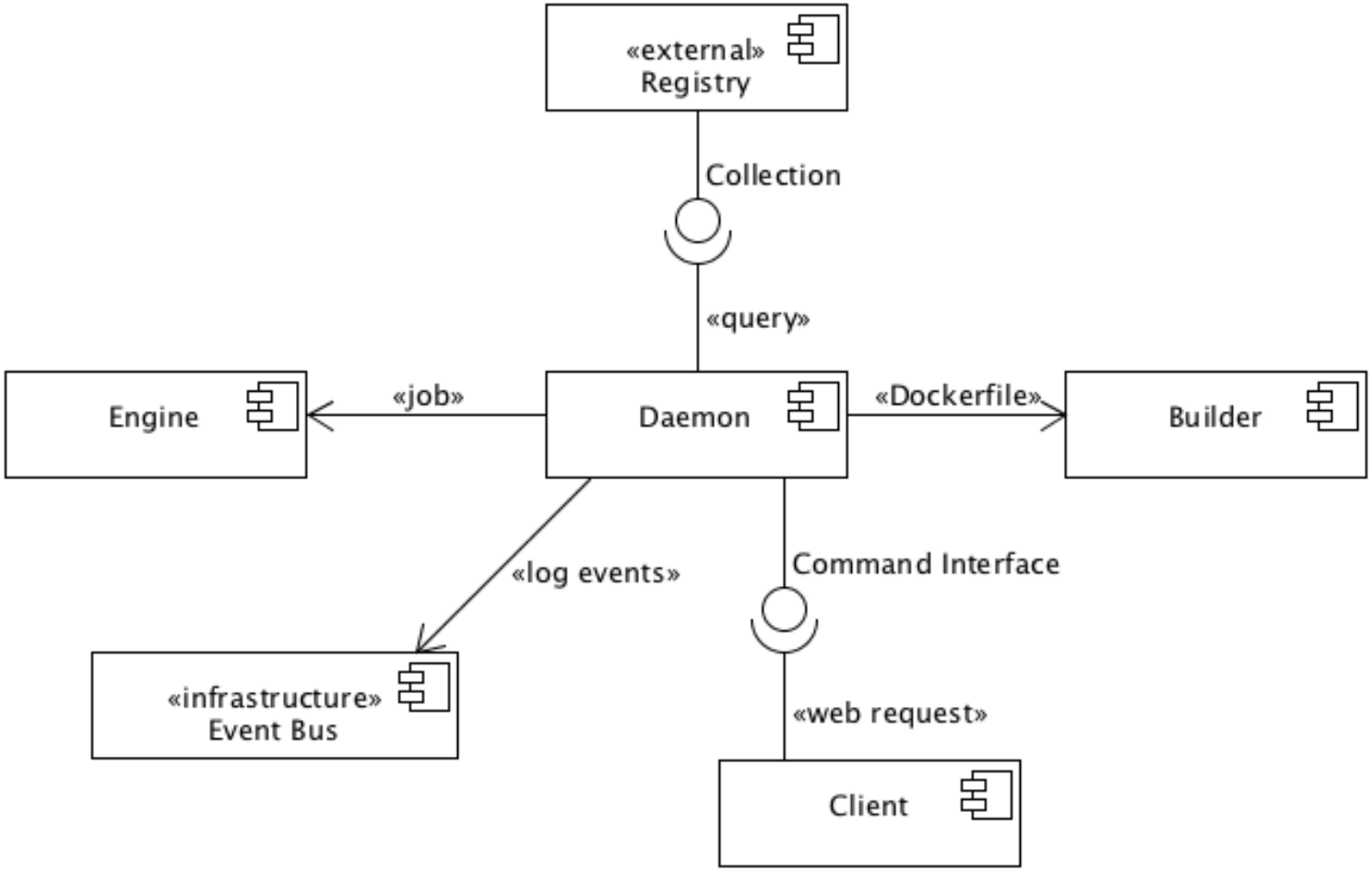
- Docker is written in Go language
- With respect to other OS-level virtualization solutions, Docker is a higher-level platform that exploits Linux kernel mechanisms such as cgroups and namespaces
  - First versions based on Linux Containers (LXC)
  - Then based on its own *libcontainer* runtime that uses Linux kernel *namespaces* and *cgroups* directly
- Docker adds to LXC
  - Portable deployment across machines
  - Versioning, i.e., git-like capabilities
  - Component reuse
  - Shared libraries, see Docker Hub hub.docker.com

Docker internals

- *libcontainer* (now included in *opencontainers/runc*): cross-system abstraction layer aimed to support a wide range of isolation technologies

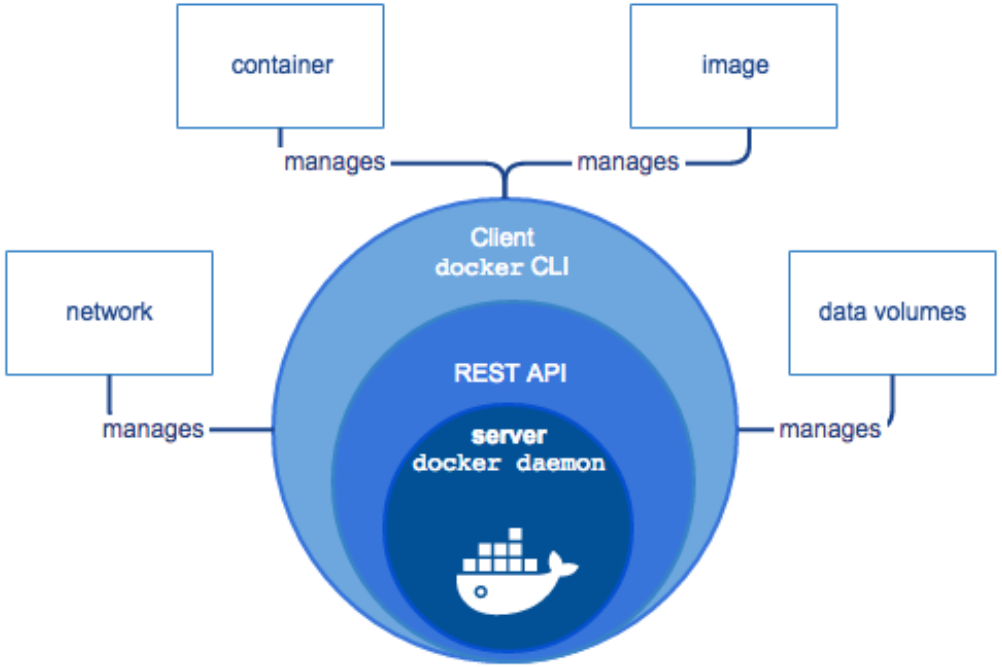


Docker component diagram



Docker engine

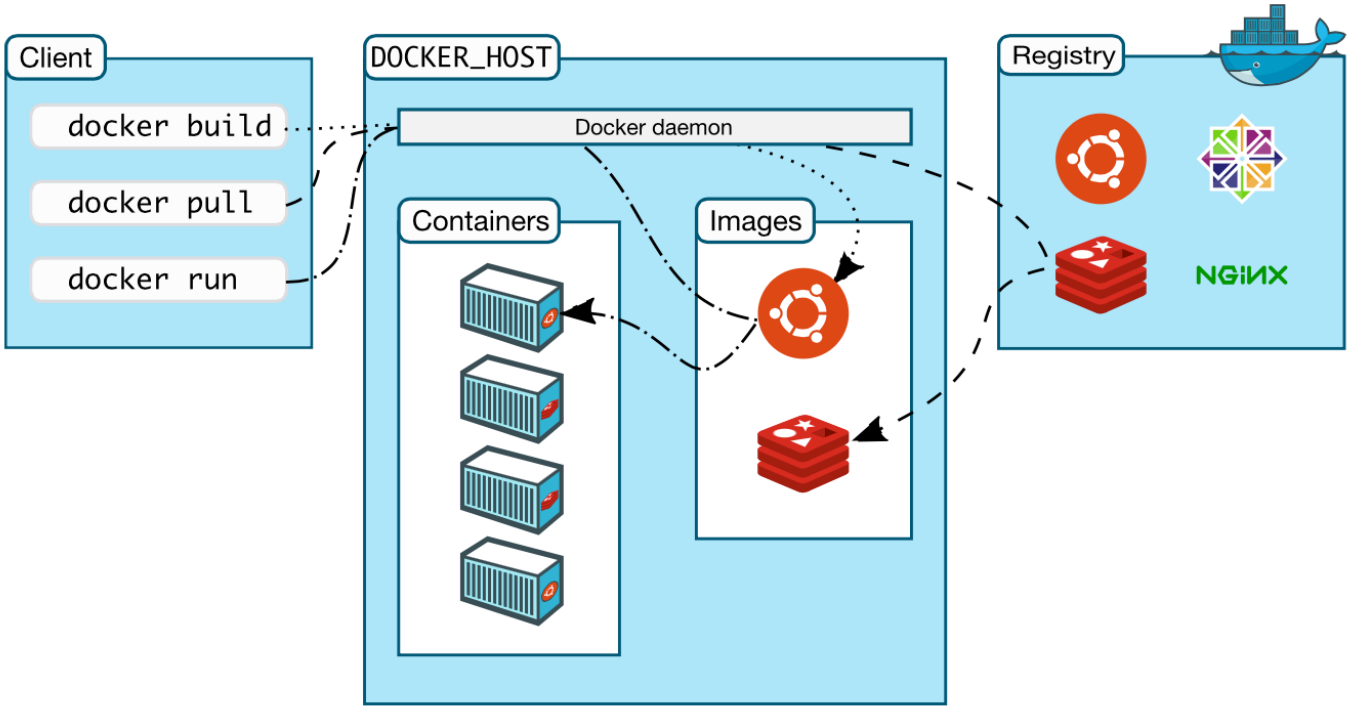
- *Docker Engine:* client-server application composed by:
  - A server, called docker daemon
  - A REST API which specifies interfaces that programs can use to control and interact with the daemon
  - A command line interface (CLI) client



See https://docs.docker.com/engine/docker-overview/

Docker architecture

- Docker uses a client-server architecture
  - The Docker *client* talks to the Docker *daemon*, which builds, runs, and distributes Docker containers
  - Client and daemon communicate via sockets or REST API



Docker

Docker image

- Read-only template used to create a Docker container
- The ***Build*** component of Docker
  - Enables the distribution of apps with their runtime environment
    - Incorporates all the dependencies and configuration necessary to apps to run, eliminating the need to install packages and troubleshoot
  - Target machine must be Docker-enabled
- Docker can build images automatically by reading instructions from a **Dockerfile**
  - A text file with simple, well-defined syntax
- Images can be pulled and pushed towards a public/private registry
- Image name: [registry/][user/]name[:tag]
  - Default for tag is latest

Docker image: Dockerfile

- Image can be created from a Dockerfile and a context
  - Dockerfile: instructions to assemble the image
  - Context: set of files (e.g., application, libraries)
- Often, an image is based on another image (e.g., ubuntu)
- Dockerfile syntax

  # Comment

  INSTRUCTION arguments
- Instructions in a Dockerfile run in order
- Some instructions

  FROM: to specify parent image (mandatory)

  RUN: to execute any command in a new layer on top of current image and commit results

  ENV: to set environment variables

  EXPOSE: container listens on specified network ports at runtime

  CMD: to provide defaults for executing container

Docker image: Dockerfile

- Example of Dockerfile to build the image of a container that will run a Python app (https://docs.docker.com/get-started/)

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim
# Set the working directory to /app
WORKDIR /app
# Copy the current directory contents into the container at /app
ADD . /app
# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt
# Make port 80 available to the world outside this container
EXPOSE 80
# Define environment variable
ENV NAME World
# Run app.py when the container launches
CMD ["python", "app.py"]
```

Docker image: build

- Build image from Dockerfile

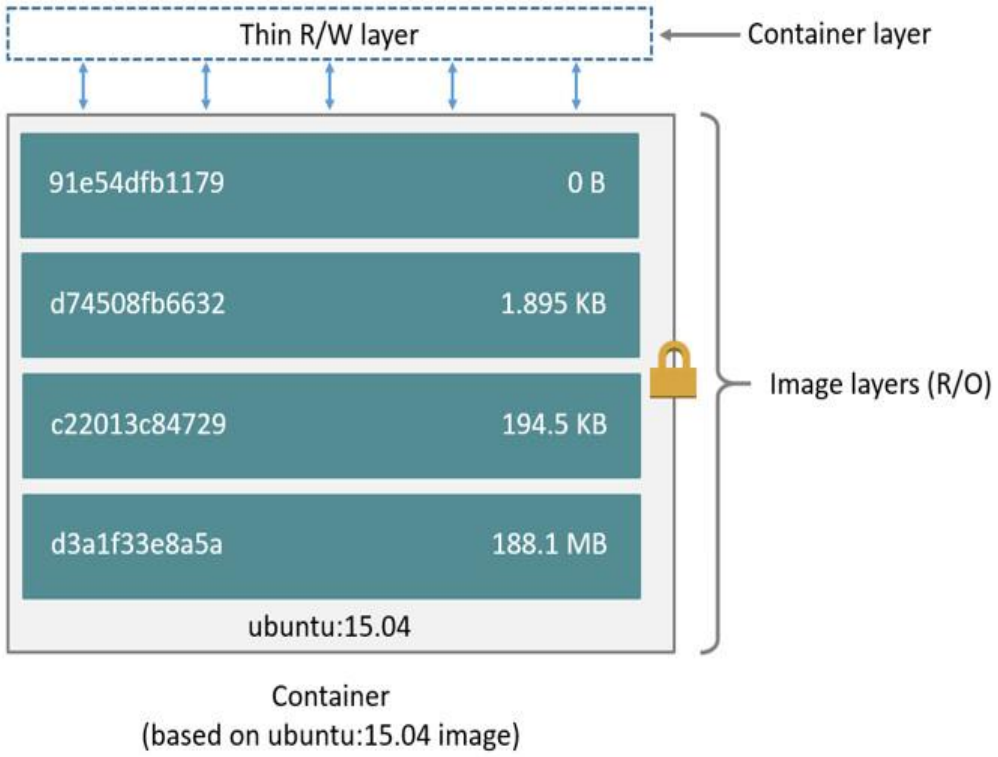
```
$ docker build [OPTIONS] PATH | URL | -
```

- E.g., to build the image for Python app (see Dockerfile in previous slide)

$ docker build -t friendlyhello .

Docker image: layers

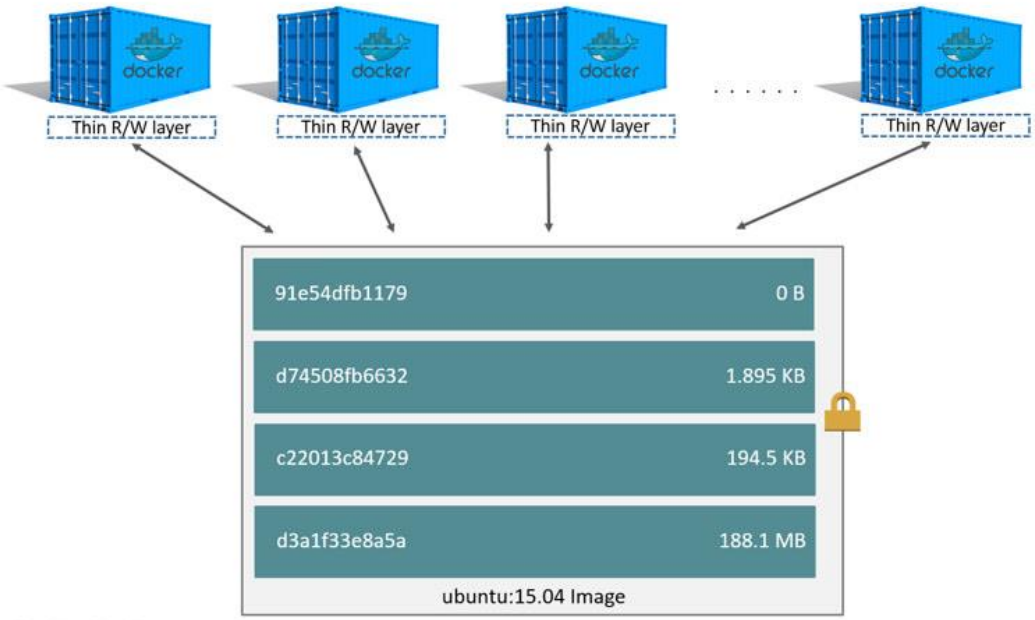
- Each image consists of a *series of layers*
- Docker uses *union file systems* to combine these layers into a single unified view
  - Layers are stacked on top of each other to form a base for a container's root file system
  - Based on *copy-on-write* (COW) principle



Docker image: layers

- Layering pros
  - Enable layer sharing and reuse, installing common layers only once and saving bandwidth and storage space
  - Manage dependencies and separate concerns
  - Facilitate software specializations

See https://docs.docker.com/storage/storagedriver/



Docker

Docker image: layers and Dockerfile

- Each layer represents an instruction in the image's Dockerfile
- Each layer except the very last one is read-only

- To inspect an image, including image layers

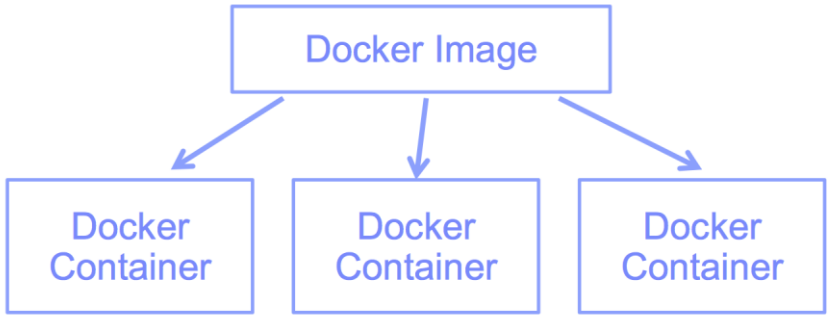
```
$ docker inspect imageid
```

Docker image: storage

- Containers should be stateless. Ideally:
  - Very little data is written to container's writable layer
  - Data should be written on Docker volumes
- Nevertheless: some workloads require to write data to the container's writable layer
- The storage driver controls how *images* and *containers* are stored and managed on the Docker host
- Multiple choices for the storage driver
  - Including AuFS and Overlay2 (at file level), Device Mapper, btrfs and zfs (at block level)
  - Storage driver's choice can affect the performance of containerized applications
  - See https://docs.docker.com/storage/storagedriver/

Docker container and registry

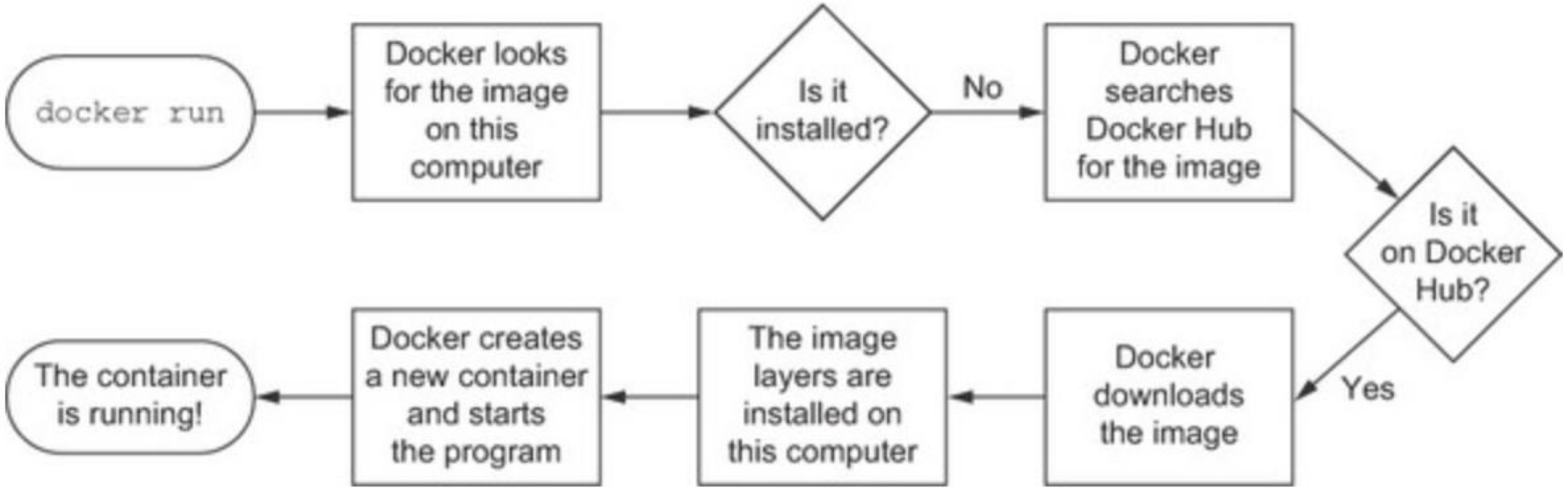
- Docker **container**: runnable instance of a Docker image
  - Run, start, stop, move, or delete a container using Docker API or CLI commands
  - The ***Run*** component of Docker
  - Docker containers are stateless: when a container is deleted, any data written not stored in a *data volume* is deleted along with the container



- Docker **registry**: stateless server-side application that stores and lets you distribute Docker images
  - Open library of images
  - The ***Distribute*** component of Docker
  - Docker-hosted registries: Docker Hub, Docker Store (open source and enterprise verified images)

Docker: run command

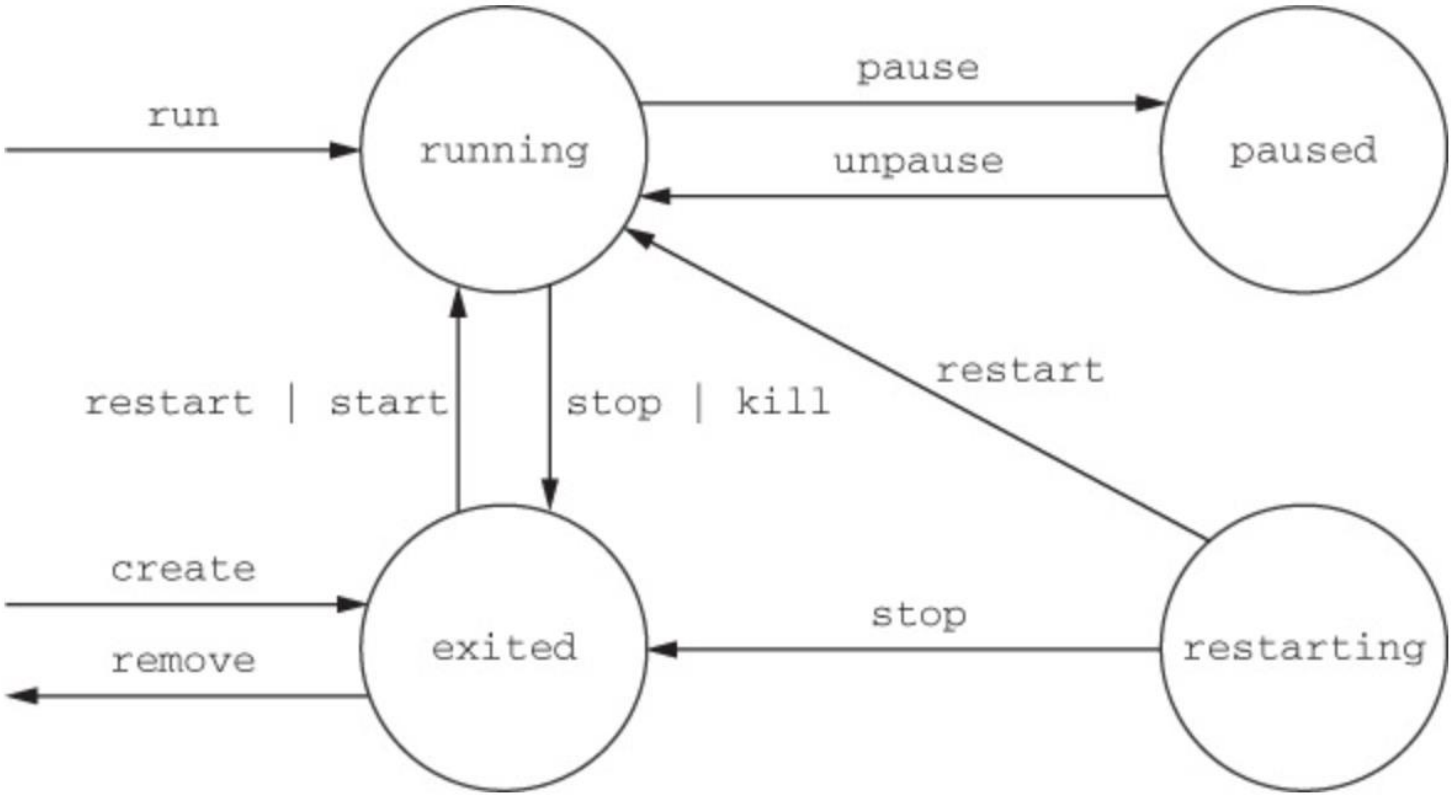
- When you run a container whose image is not yet installed but is available on Docker Hub



Docker in Action by Jeff Nickoloff

Docker containers

- State transitions



Docker in Action by Jeff Nickoloff

Commands: Docker info

- Obtain system-wide info on Docker installation

```
$ docker info
```

- Including:
  - How many images, containers and their status
  - Storage driver
  - Operating system, architecture, total memory
  - Docker registry
  - Docker Swarm status

Commands: image handling

- List images on host (i.e., local repository)

```
$ docker images
```

- List every image, including intermediate image layers:

```
$ docker images -a
```

- Options to list images by name and tag, to list image digests (sha256), to filter images, to format the output, e.g.,

```
$ docker images --filter reference=ubuntu
```

- Remove an image (can also use `imagename` instead of `imageid`)

```
$ docker rmi imageid
```

Command: run

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARGS]
```

- Most common options

--name	assign a name to the container
-d	detached mode (in background)
-i	interactive (keep STDIN open even if not attached)
-t	allocate a pseudo-tty
--expose	expose a range of ports inside the container
-p	publish a container's port or a range of ports to the host
-v	bind and mount a volume
-e	set environment variables
--link	add link to other containers

- The "Hello World" container

```
$ docker run alpine /bin/echo 'Hello world'
```

  - alpine: lightweight Linux distro with reduced image size

Commands: containers management

- List containers
  - Only running containers: `$ docker ps`
  - Alternatively, `$ docker container ls`
  - All containers (even stopped or killed containers):

    ```
    $ docker ps -a
    ```
- Container lifecycle
  - Stop running container

    ```
    $ docker stop containerid
    ```
  - Start stopped container

    ```
    $ docker start containerid
    ```
  - Kill running container

    ```
    $ docker kill containerid
    ```
  - Remove container (need to stop it before attempting removal)

    $ docker rm containerid

Can also use `containername` instead of `containerid`

Commands: containers management

- Inspect a container
  - Most detailed view of the environment in which a container was launched

```
$ docker inspect containerid
```

- Copy files from and to docker container

```
$ docker cp containerid:path localpath
$ docker cp localpath containerid:path
```

Examples of using Docker

- Run a nginx Web server inside a container
  - Also bind the container to a specific port

```
$ docker run –d –p 80:80 --name web nginx
```

- Send HTTP request through Web browser
  - First retrieve the hostname of the host machine
- Send HTTP request through an interactive container using Docker internal network

```
$ docker run -i -t --link web:web --name web_test busybox
/ # wget -O - http://web:80/
/ # exit
```

`--link`: legacy flag to manually create links between the containers
`wget`: -O FILE Save to FILE ('-' for stdout)

  - Instead of using --link, let us define a bridge network

```
$ docker network create my_net
$ docker run -d –p 80:80 --name web --net=my_net nginx
$ docker run –i -t --net=my-net --name web_test busybox
/ # ...
```

Examples of using Docker

- Send HTTP request through an Alpine Linux container with curl installed and set as entrypoint

```
$ docker run --rm byrnedo/alpine-curl http://…
```

- Check container logs

```
$ docker logs containerid
```

Examples of using Docker

- Running Apache web server with minimal index page
  - Define container image with Dockerfile
    - Define image starting from Ubuntu, install and configure Apache
    - Incoming port set to 80 using EXPOSE instruction

```
FROM ubuntu

# Install dependencies
RUN apt-get update
RUN apt-get -y install apache2
# Install apache and write hello world message
RUN echo 'Hello World!' > /var/www/html/index.html
# Configure apache
RUN echo '. /etc/apache2/envvars' > /root/run_apache.sh
RUN echo 'mkdir -p /var/run/apache2' >> /root/run_apache.sh
RUN echo 'mkdir -p /var/lock/apache2' >> /root/run_apache.sh
RUN echo '/usr/sbin/apache2 -D FOREGROUND' >> /root/run_apache.sh
RUN chmod 755 /root/run_apache.sh

EXPOSE 80
CMD /root/run_apache.sh
```

Examples of using Docker

- Build container image from Dockerfile

```
$ docker build -t hello-apache .
```

- Run container and bind

```
$ docker run -d -p 80:80 hello-apache
```

  - Option –p: publish container port (80) to host port (80)

Examples of using Docker

- Stop and remove a container

```
$ docker ps
```

```
$ docker ps
CONTAINER ID  IMAGE        COMMAND                  CREATED        STATUS        PORTS                  NAMES
2b2b39d05fe0  hello_world  "/bin/sh -c /root/ru…"  3 seconds ago  Up 2 seconds  0.0.0.0:80->80/tcp  pedantic_austin
```

```
$ docker stop containerid
$ docker ps -a
$ docker rm containerid
```

- Stop all containers

```
$ for i in $(docker ps -q); do docker stop $i; done
```

Examples of using Docker

- Running a Python web app in Docker
  - See https://docs.docker.com/get-started/part2/
  - Define container image with Dockerfile
    - Define image with Python runtime and Python app
    - Incoming port set to 80 using EXPOSE instruction
  - Write app code in Python using Flask and Redis packages and create file requirements.txt to specify those packages needed by app
    - Redis: in-memory data store, used to keep the counter of the number of visits to web app
  - Build container image

```
$ docker build -t friendlyhello .
```

  - Run container and bind

```
$ docker run -d -p 4000:80 friendlyhello
```

    - Option –p: publish container port (80) to host port (4000)

Multi-container Docker applications

- How to run multi-container Docker apps?

1. Docker Compose
   - Deployment only on single host
2. Docker Swarm
   - Native orchestration tool for Docker
   - Deployment on multiple hosts
3. Kubernetes
   - Deployment on multiple hosts

Docker Compose

- To coordinate the execution of multiple containers, we can use Docker Compose
  - See https://docs.docker.com/compose/
- Docker Compose
  - Not bundled within Docker installation (on Linux)
  - https://docs.docker.com/compose/install/
- Allows to easily express the containers to be instantiated at once, and the relations among them
- Runs the composition on a single machine (i.e., single Docker engine)
- Use Docker Swarm if you need to deploy containers on multiple nodes

Docker Compose

- Specify how to compose containers in a easy-to-read file named docker-compose.yml
- To start Docker composition (background -d):

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

- To stop Docker composition:

```
$ docker-compose down
```

- By default, Docker Compose looks for docker-compose.yml in current working directory
  - Change file using -f flag

Docker Compose

- Different versions of the Docker compose file format
  - Latest: version 3 is supported from Docker Compose 1.13

```
version: '3'

services:
    storm-nimbus:
        image: storm
        container_name: nimbus
        command: storm nimbus
        depends_on:
            - zookeeper
        links:
            - zookeeper
        ports:
            - "6627:6627"
```



```
zookeeper:
    image: zookeeper
    container_name: zookeeper
    ports:
        - "2181:2181"

worker1:
    image: storm
    command: storm supervisor
    depends_on:
        - storm-nimbus
        - zookeeper
    links:
        - storm-nimbus
        - zookeeper
```

Docker Compose file format:
https://docs.docker.com/compose/compose-file/

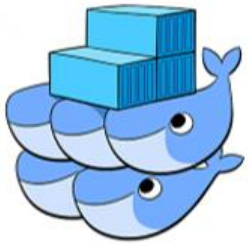
Docker Compose: example

- Simple Python web app running on Docker Compose
  - Two containers: Python web app and Redis
  - Use Flask framework and maintain a hit counter in Redis
  - See https://docs.docker.com/compose/gettingstarted/
- Steps:
  - Write Python app
  - Define Python container image with Dockerfile
  - Define services in docker-compose.yml file
    - Two services: web (image defined by Dockerfile) and redis (image pulled from Docker Hub)
  - Build and run your app with Compose

    ```
    $ docker-compose up -d
    ```
  - Send HTTP requests using curl (now counter is increased)
  - Stop Compose

    ```
    $ docker-compose down
    ```

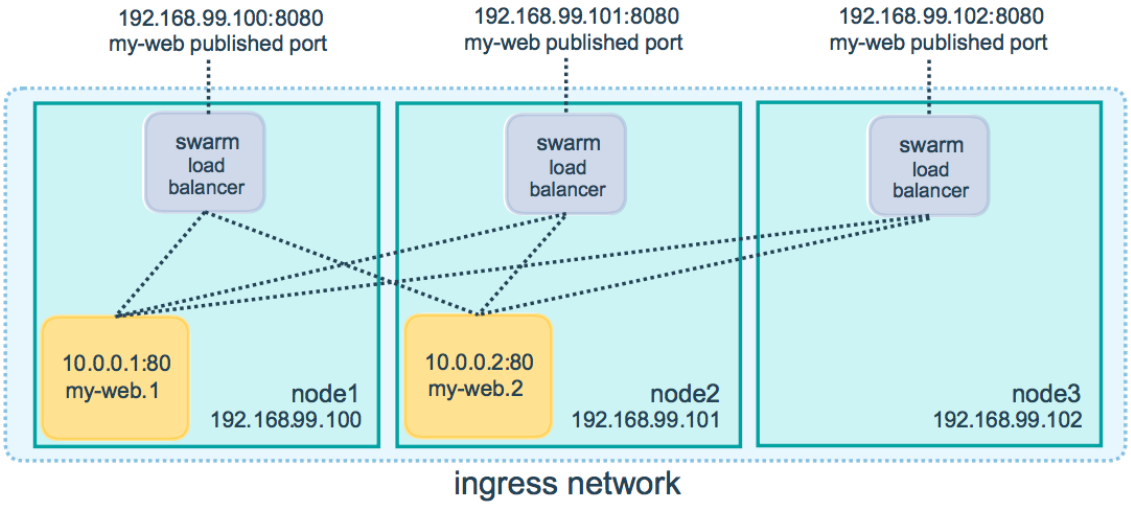
Docker Swarm



- Docker includes **swarm mode** for natively managing a cluster of Docker Engines, called *swarm*
  - See https://docs.docker.com/engine/swarm/
- Tasks: containers running in a service
- Basic features of swarm mode:
  - **Scaling:** number of tasks for each service
  - **State reconciliation:** Swarm monitors cluster state and reconciles any differences w.r.t. desired state (e.g., replace containers after host failure)
  - **Multi-host networking:** to specify an overlay network among services
  - **Load balancing:** allows to expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute containers among nodes

Docker Swarm

- A swarm consists of multiple Docker hosts which run in swarm mode
- Node: instance of Docker engine
  - **Manager node** dispatches tasks to worker nodes
  - **Worker nodes** receive and execute tasks
- Load balancing
  - Swarm manager can automatically assign the service a (configurable) PublishedPort
  - External components can access the service on PublishedPort. All nodes in the swarm route ingress connections to a running task



Docker

Docker Swarm: Swarm cluster

- Create a swarm: manager node

```
$ docker swarm init --advertise-addr <MANAGER-IP>
Swarm initialized: current node (<nodeid>) is now a manager.
To add a worker to this swarm, run the following command:

    docker swarm join --token <token> <manager-ip>:port
```

- Create a swarm: worker node

```
$ docker swarm join --token <token> <manager-ip>:port
```

- Inspect status

```
$ docker info
```

```
$ docker node ls

ID            HOSTNAME      STATUS    AVAILABILITY    MANAGER STATUS
<nodeid> *    controller    Ready     Active          Leader
<nodeid>      storage       Ready     Active
```

Docker Swarm: Swarm cluster

- Leave the swarm

```
$ docker swarm leave
```

  - If the node is a manager node, warning about maintaining the quorum (to override warning, --force flag)

- After a node leaves the swarm, you can run docker node rm command on a manager node to remove the node from the node list

```
$ docker node rm node-id
```

Docker Swarm: manage services

- Deploy a service to the swarm (from manager node)

```
$ docker service create -d --replicas 1 \
     --name helloworld alpine ping docker.com
```

- List running services

```
$ docker service ls
```

```
ID            NAME          MODE         REPLICAS  IMAGE           PORTS
<serviceid>   helloworld    replicated   1/1       alpine:latest
```

Docker Swarm: manage services

- Inspect service

```
$ docker service inspect --pretty <SERVICE-ID>
$ docker service ps <SERVICE-ID>
```

```
ID          NAME           IMAGE          NODE        DESIRED ST CURRENT ST   ERROR   PORTS
<cont.id1>  helloworld.1   alpine:latest  controller  Running    Running …
<cont.id2>  helloworld.2   alpine:latest  storage     Running    Running …
```

- Inspect container

```
$ docker ps <cont.id1>
```

```
# Manager node
CONTAINER ID  IMAGE          COMMAND             CREATED     STATUS     ... NAMES
<cont.id1>    alpine:latest  "ping docker.com"   2 min ago   Up 2 min   helloworld.1.iuk1sj…

# Worker node
CONTAINER ID  IMAGE          COMMAND             CREATED     STATUS     ... NAMES
<cont.id2>    alpine:latest  "ping docker.com"   2 min ago   Up 2 min   helloworld.2.skfos4…
```

Docker Swarm: manage services

- Scale service

```
$ docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>
```

  - Swarm manager will automatically enact the updates

- Apply rolling updates to a service

```
$ docker service update --limit-cpu 2 redis
$ docker service update --replicas 2 helloworld
```

- Roll back an update

```
$ docker service rollback [OPTIONS] <SERVICE-ID>
```

- Remove a service

```
$ docker service rm <SERVICE-ID>
```

References

- Docker Documentation
  - https://docs.docker.com/
  - https://docs.docker.com/get-started/
- Docker for beginners
  - https://docker-curriculum.com/
- Ian Miell and Aidan Hobson Sayers, *Docker in Practice*, 2nd Edition, Manning Publications, 2019