TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Container-based virtualization: Docker

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2023/24

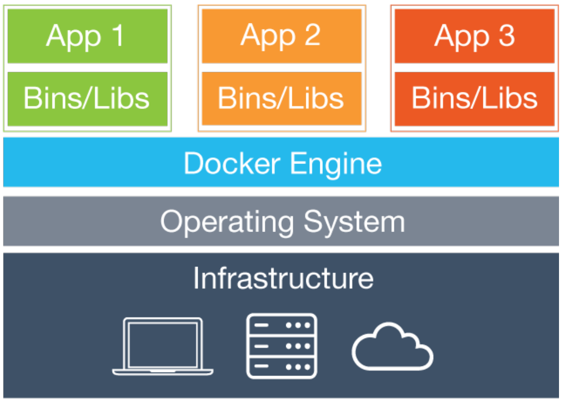
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Case study: Docker

docker

- Lightweight, open and secure container-based virtualization
  - Container includes an application and its dependencies, but shares OS kernel with other containers
  - Container runs as isolated process in user space on host OS
  - Container is not tied to any specific infrastructure



Docker features

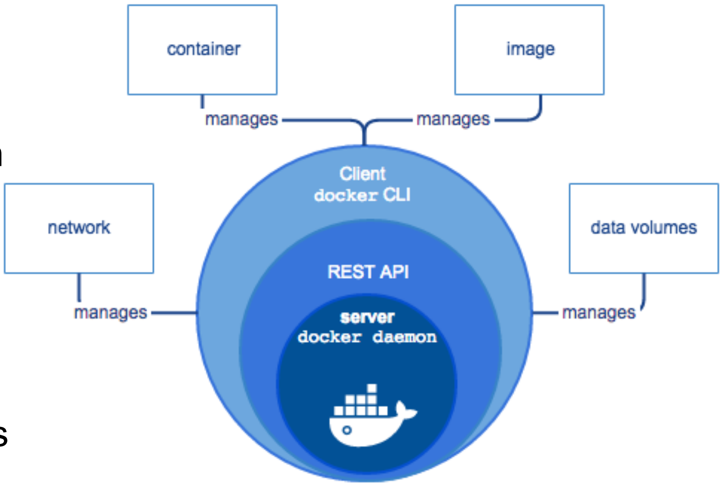
- Portable deployment across machines
- Versioning, i.e., git-like capabilities
- Component reuse
- Shared libraries, see Docker Hub
- Supports Open Container Initiative (OCI), a set of standards for containers

Docker internals

- Written in Go
- Exploits Linux kernel mechanisms such as cgroups and namespaces
  - First versions were based on Linux Containers (LXC)
  - Then based on `libcontainer`, a container runtime which provides a native Go implementation for creating containers with namespaces, cgroups, capabilities, and filesystem access controls and allows you to manage the lifecycle of the container
  - `libcontainer` *is* ow included in `runc`: CLI tool for spawning and running containers according to OCI specification

Docker Engine

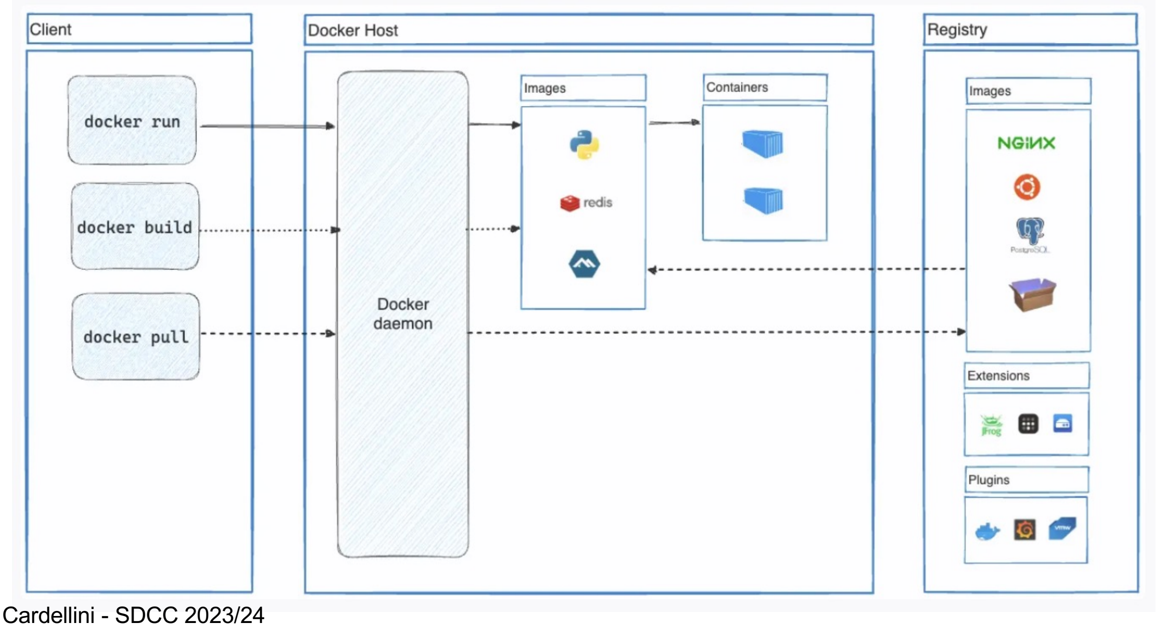
- Docker Engine acts a client-server application composed by:
  - Server, called Docker daemon (`dockerd`), which listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes
  - REST API which specifies interfaces that programs can use to control and interact with the daemon
  - Command line interface (CLI) client



See docs.docker.com/get-started/overview/#docker-architecture

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 image

- Read-only template used to create a Docker container
- ***Build*** component of Docker
  - Enables apps distribution 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 created from Dockerfile and context
  - Dockerfile: instructions to assemble the image
  - Context: set of files (e.g., application, libraries)
  - Often, an image is based on a parent image (e.g., alpine)
- Dockerfile syntax

```
# Comment
INSTRUCTION arguments
```

- Instructions in 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

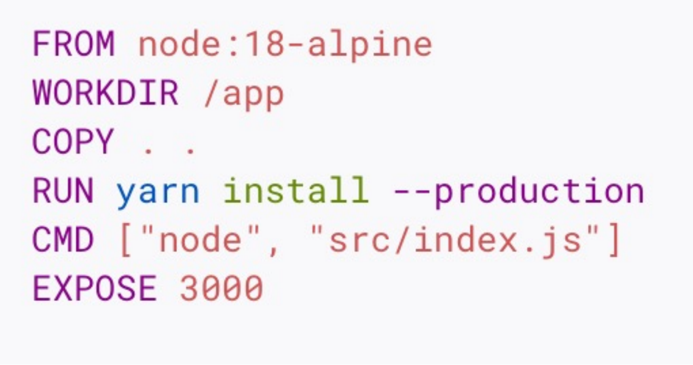
  `ENV`: to set environment variables

  `EXPOSE`: container listens on specified network ports at runtime

  `CMD`: to provide defaults for executing container

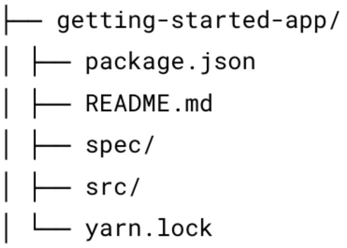
Docker image: Dockerfile

- Example: Dockerfile to build the image of a container that will run as application a simple todo list manager written in Node.js



```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Directory with app code



```
├── getting-started-app/
│ ├── package.json
│ ├── README.md
│ ├── spec/
│ ├── src/
│ └── yarn.lock
```

See docs.docker.com/get-started/02_our_app

Docker image: build

- Build image from Dockerfile and context
  - Build's context is the set of files located in the specified PATH or URL

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

  - E.g., to build the image for Node.js app (see previous slide)

```
$ docker build -t getting-started .
```

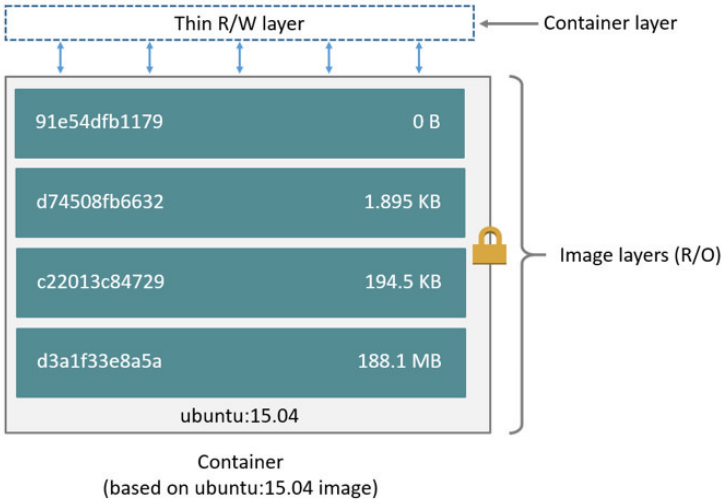
  - If the name of the Dockerfile is not Dockerfile use `-f`, e.g.,

```
$ docker build -t getting-started -f myDockerfile .
```

See docs.docker.com/engine/reference/commandline/build/

Docker image: layers

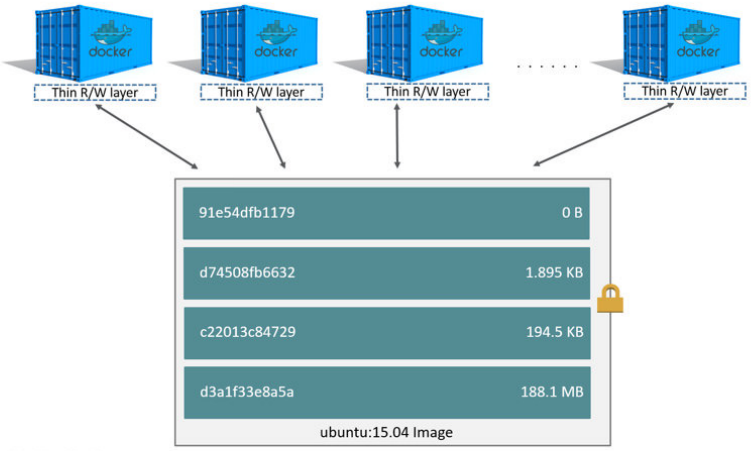
- Each image consists of a *series of layers*
- Docker uses *union file systems* to combine these layers into a single unified view
  - Layers are stacked on top of each other to form a base for a container's root file system
  - Based on *copy-on-write* (CoW) strategy



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 docs.docker.com/storage/storagedriver



Docker image: layers and Dockerfile

- Each layer represents an instruction in Dockerfile
  - Except `CMD` instruction, which specifies what command to run within container: it only modifies image's metadata, without producing an image layer
- Each layer except the very last one is read-only
- Writable layer on top (aka *container layer*) is added when container is created
  - Changes made to running container (e.g., writing a file) are written to writable layer
  - Does not persist after container is deleted
  - Suitable for storing *ephemeral data* generated at runtime
- To inspect an image, including image layers

```
$ docker inspect imageid
```

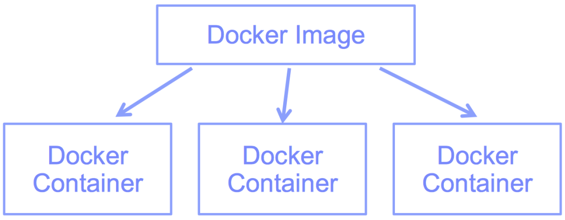
Docker image: storage

- Containers are usually stateless (why? easier to scale, restart from failure, migrate)
  - Very little data written to container's writable layer
  - Data usually written on **Docker volumes**
  - Nevertheless: some workloads require to write data to container's writable layer
- Storage driver controls how *images* and *containers* are stored and managed on Docker host
- Multiple choices for storage driver
  - Including Overlay2 (at file level, preferred for all Linux distros), Device Mapper, btrfs and zfs (at block level)
  - Storage driver's choice can affect performance of containerized apps: optimized for space efficiency, but write speeds can be lower than native file system performance

See docs.docker.com/storage/storagedriver/select-storage-driver

Docker container and registry

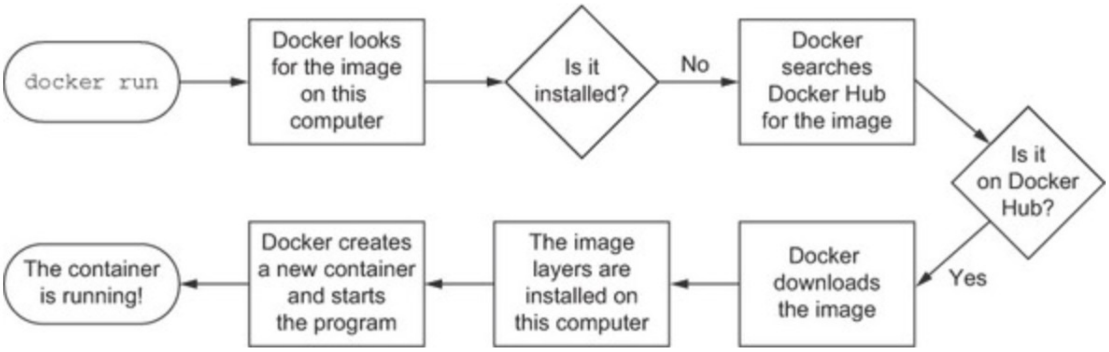
- Docker **container**: runnable instance of Docker image
  - ***Run*** component of Docker
  - Run, start, stop, move, or delete a container using Docker API or CLI commands
  - Docker containers are stateless: when a container is deleted, any data written not stored in a data volume is deleted



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

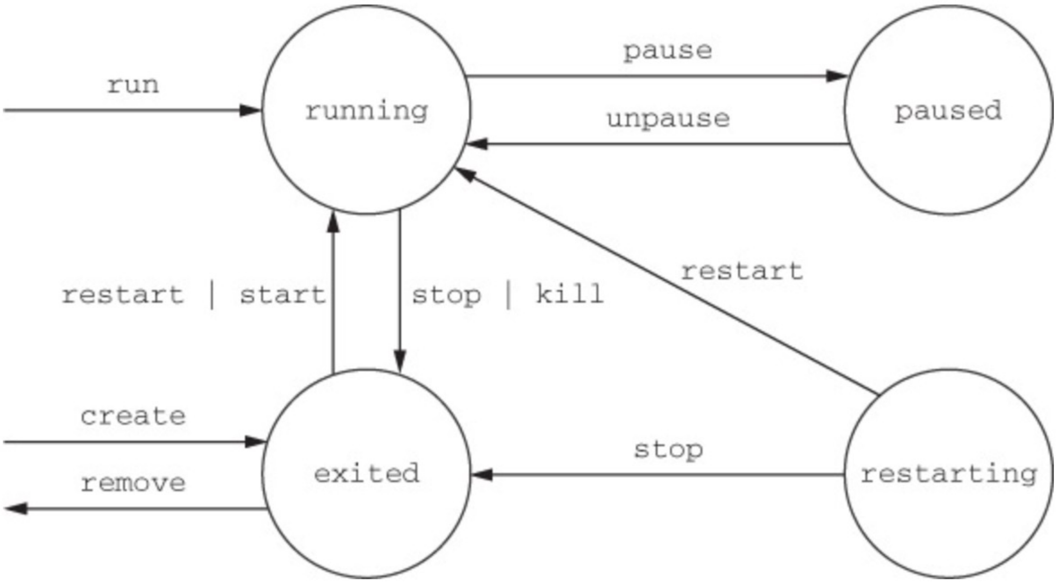
Docker: run command

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


docker run
Docker looks for the image on this computer
Is it installed?
No
Docker searches Docker Hub for the image
Is it on Docker Hub?
Yes
Docker downloads the image
The image layers are installed on this computer
Docker creates a new container and starts the program
The container is running!

Courtesy of “Docker in Action” by J. Nickoloff

State transitions of Docker containers



Courtesy of “Docker in Action” by J. 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

Commands: image handling

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

```
$ docker images
```

alternatively, `$ docker image ls`

- List every image, including intermediate image layers

```
$ docker image ls -a
```

- Options to list images by name and tag, to list image digests (sha256), to filter images, to format the output
  - E.g., to list untagged images (`<none>`) that have no relationship to any tagged images (no longer used but consume disk space)

```
$ docker images --filter "dangling=true"
```

- Remove an image

```
$ docker rmi imageid
```

alternatively, `$ docker image rm imageid`

can also use *`imagename`* instead of *`imageid`*

Command: run

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

- Most common options

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

See docs.docker.com/engine/reference/commandline/run/

Commands: containers management

- List containers
  - Only running containers: `$ docker ps`
    alternatively, `$ docker container ls`
  - All containers (including stopped or killed containers):
    `$ docker ps -a`
- Manage 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

- Stop and remove a running container

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

- Stop all containers

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

- Execute command in a running container

```
$ docker exec [OPTIONS] CONTAINER [COMMAND] [ARGS]
```

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 container

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

Docker networking

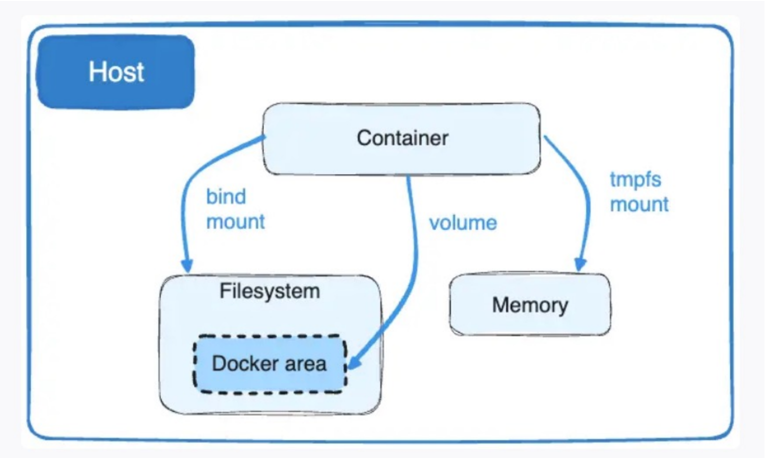
- Container networking: ability for containers to connect to and communicate with each other or to non-Docker workloads
- Published ports
  - In `docker run`, use `--publish` or `-p` flag to make port available to services outside of Docker
  - E.g.: `-p 8080:80` map port 8080 on host to TCP port 80 in container
  - Issue: publishing container ports is insecure by default
    - Include localhost IP address so that only host can access container port, e.g.: `-p 127.0.0.1:8080:80`
- IP address and hostname
  - Container receives IP address out of network IP subnet
  - Docker daemon performs dynamic subnetting and IP address allocation for containers
  - Container hostname defaults to be container ID in Docker

Docker networking: network drivers

- Docker's networking is pluggable using drivers
- Several network drivers, including
  - `bridge`: default network driver, used when application runs in a container that needs to communicate with other containers on the same host
    - Software bridge which lets containers connected to same bridge network communicate, while providing isolation from containers that are not connected to that bridge network
  - `host`: remove network isolation between container and host and use host networking directly

Docker volumes

- Preferred mechanism for persisting data generated by and used by Docker containers
  - New directory is created within Docker's storage directory on host machine, and Docker manages directory's content
    - On Linux storage directory is `/var/lib/docker/volumes/`
  - Volume does not need to exist on host, it is created on demand if it does not yet exist



Docker volumes

- To mount a volume to a container, use `-v` (or --volume) flag with `docker run`

  `$ docker run … -v `*`source:destination`*`:[options]`

  - Use `ro` option to mount a read-only volume
  - If a container is started with a volume that does not yet exist, Docker creates the volume
- Commands to manage volumes:
  - Create volume: `$ docker volume create `*`volumename`*
  - List volumes: `$ docker volume ls`
  - Inspect volume: `$ docker volume inspect `*`volumename`*
  - Remove volume: `$ docker volume rm `*`volumename`*
- Volume can be declared in Dockerfile using `VOLUME`
- How to load data into a volume? Can use `docker cp`

See docs.docker.com/engine/reference/commandline/

Docker volumes: pros

✓ Completely managed by Docker

✓ Easy to back up or migrate

✓ Managed using Docker CLI or API

✓ Work on both Linux and Windows containers

✓ Can be shared among multiple containers

✓ Content can be encrypted

✓ Content can be pre-populated

✓ Better choice than persisting data in container’s writable layer

- A volume does not increase container size and its contents exist outside container lifecycle

- Tip: use volumes for write-heavy application (e.g., a write-intensive DB)

Hands-on: hello world

- Download and install Docker
  - Available on multiple platforms

    docs.docker.com/get-docker

    docs.docker.com/get-started

- Test Docker version

```
$ docker --version
```

- Test Docker installation by running hello-world Docker image

```
$ docker run hello-world
```

Hands-on: hello world

- Run “Hello World” container with a command

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

  - alpine: lightweight Linux distro with reduced image size

- Use commands to:
  - List containers and container images
  - Remove containers and container images

Hands-on: networking

- Run nginx Web server inside a container
  - Bind container port to host port

```
$ docker run -dp 80:80 --name web nginx
```

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

Option `-d`: detached mode

1. Send HTTP request through Web browser
   - First retrieve hostname of host machine (e.g., localhost)
2. Send HTTP request to nginx from interactive container using a bridge network

```
$ docker network create -d bridge my_net
$ docker run -dp 80:80 --name web --network=my_net nginx
$ docker run -i -t --network=my_net --name web_test busybox
/ # wget -O - http://web:80/
/ # exit
```

Hands-on: from Dockerfile

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

```
FROM ubuntu:18.04

# Install dependencies
RUN apt-get update -y
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
```

Hands-on: from Dockerfile

2. Build container image from Dockerfile

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

3. Run container and bind

```
$ docker run -dp 80:80 hello-apache
```

4. Execute an interactive shell in running container

```
$ docker exec --it hello-apache /bin/bash
```

- To reduce container's image size let's avoid adding unnecessary layers
  - E.g., in Dockerfile update and install multiple packages in a single RUN instruction
    - Use \ to type out the command in multiple lines

Hands-on: from Dockerfile

```
FROM ubuntu:18.04

# Install dependencies
RUN apt-get update -y && \
 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 && \
 echo 'mkdir -p /var/run/apache2' >> /root/run_apache.sh && \
 echo 'mkdir -p /var/lock/apache2' >> /root/run_apache.sh && \
 echo '/usr/sbin/apache2 -D FOREGROUND' >> /root/run_apache.sh && \
 chmod 755 /root/run_apache.sh

EXPOSE 80

CMD /root/run_apache.sh
```

Hands-on: volumes

- Run nginx container with volume

```
$ docker volume create my-vol
$ docker volume ls
$ docker volume inspect my-vol
$ docker run -d \
--name devtest \
-v my-vol:/app \
nginx:latest
```

  - `my-vol` is the source, `/app` is the target inside container

```
$ docker inspect devtest
```

  - Inspect container to verify that Docker created the volume and it mounted correctly

Docker: reduce image size

- Optimize Docker images
  - Especially important for DevOps engineers at every stage of CI/CD process
  - Not only to reduce image disk space, reduce image transfer and deploy time, but also to improve security
  - Best practice employed by Google and other tech giants
- Techniques
  1. Use minimal base images (e.g., alpine, minideb) or distroless base images
     - Distroless images contain only application and its runtime dependencies; do not contain package managers, shells or any other programs available in standard Linux distro
  2. Minimize number of image layers

Docker: reduce image size

- Techniques
  3. Multistage builds
     - Use intermediate images (build stages) to compile code, install dependencies, and package files; after that, only necessary files required to run app are used in another image with only the required libraries
  4. Exploit image layers' caching
     - Add the lines which are used for installing dependencies and packages earlier inside Dockerfile, before COPY commands
  5. Use `.dockerignore` file
     - Configuration file that describes files and directories that you want to exclude when building a Docker image
  6. Keep application data in a volume, not inside the container
- Tools to minimize image size, e.g., Slim

See devopscube.com/reduce-docker-image-size/

Configure container memory and CPU

- By default, a container has no resource constraints
  – Can use as much resource as host's kernel scheduler allows
- Docker provides ways to control how much memory or CPU a container can use by setting runtime configuration flags of `docker run`
  docs.docker.com/config/containers/resource_constraints
  – Docker Engine implements configuration changes by modifying settings of container's `cgroup`

Configure container memory

- Avoid running out of memory (OOM)
  - Individual containers can be killed (Docker daemon has lower OOM priority, containers default one)
- Docker can enforce hard or soft memory limits
  - Hard limit: container cannot use more than a given amount of user or system memory; `--memory` flag
  - Soft limit: container can use as much memory as it needs unless certain conditions are met, such as when kernel detects contention or low memory on host machine
  - Example: limit container to use at most 500 MB of memory (hard limit) and specify also a soft limit

```
$ docker run -it --memory-reservation="300m" \
  --memory="500m" ubuntu /bin/bash
```

Configure container CPU

- Various constraints to limit container usage of host machine's CPU cycles
- Some options

  `--cpus=<value>`: limit how many CPU resources a container can use (hard limit)

  `--cpu-quota=<value>`: set CPU Completely Fair Scheduler (CFS) quota on container

  `--cpuset-cpus`: limit specific CPUs or cores a container can use

  `--cpu-shares`: set to value >/< 1024 to increase/reduce container's weight, and give it access to greater/less proportion of CPU cycles (soft limit)

  - Example: limit container to use at most 50% of CPU every second

```
$ docker run -it --cpus=".5" ubuntu /bin/bash
```

Alternatively,

```
$ docker run -it --cpu-period=100000 \
--cpu-quota=50000 ubuntu /bin/bash
```

Multi-container Docker applications

- How to run multi-container Docker apps?
- **Docker Compose**
  - Deployment only on single host
- Docker Swarm
  - Native orchestration tool for Docker
  - Deployment on multiple hosts
- Kubernetes
  - Deployment on multiple hosts
  - See next lesson

Docker Compose

- A tool for defining and running multi-container Docker applications docs.docker.com/compose/
- Bundled within Docker Desktop docs.docker.com/compose/install/
- Allows to coordinate a composition of multiple containers which run on a single host (i.e., on a single Docker engine)
  - Easily express the containers to be instantiated at once and their relationships
  - Docker Compose automatically sets up a network and attaches all deployed containers to it
  - To deploy containers on multiple nodes, use either Docker Swarm or Kubernetes

Docker Compose: how to use

- To get started: specify how to compose containers in a YAML file named `compose.yaml`
- Then, manage the whole lifecycle of containerized application through Compose

- To start Docker composition (background `-d`):

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

  - By default, Docker Compose looks for `compose.yaml` in working directory
    - Can specify a different file using `-f` flag

```
$ docker compose -f composefile up -d
```

- To stop running containers:

```
$ docker compose stop
```

- To bring composition down, removing everything

```
$ docker compose down
```

Docker Compose: Compose file

- Compose file allows to configure Docker application's services, networks, volumes, and more
  - Different versions of Compose file format docs.docker.com/compose/compose-file/
  - Latest: Compose V2 implements format defined by Compose Specification and includes legacy versions 2.x and 3.x
- What inside `compose.yaml` (or `compose.yml` or `docker-compose.yml`)?
- YAML file which defines: version (optional), services (required), networks, volumes, configs, secrets

See docs.docker.com/compose/compose-file/03-compose-file/

Docker Compose: Compose file

- Service: abstract definition of computing resource within application which can be scaled or replaced independently from other components
  - Services are backed by a set of containers
  - Compose file must declare a `services` top-level element
- Each service
  - may also include a `build` section, which defines how to create service image
  - may also specify `container_name`, startup and shutdown dependencies between services (`depends_on`), exposed containers `ports`, CPU and memory limits, `volumes` that are accessible to service containers
  - and many other settings, see docs.docker.com/compose/compose-file/05-services/

Docker Compose: Compose file

- Example: Compose file for Apache Storm, a distributed data stream processing framework
  - Master-worker architecture (Nimbus is master) using Zookeeper
  - Let's define a Storm cluster with 3 containers: master, worker and Zookeeper

```
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: full example

- Simple Python web app running on Docker Compose
  - 2 containers: Python web app and Redis
  - Use Flask framework and maintain hit counter in Redis
  - Redis: in-memory, key-value data store

  See docs.docker.com/compose/gettingstarted/

- Steps:
  1. Write Python app
  2. Define Python container image with its Dockerfile →

```
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

Docker Compose: full example

- Steps (cont'd):
  3. Define services in Compose file →
     - Two services: web (image defined by Dockerfile) and redis (official image pulled from Docker Hub)

```
services:
  web:
        build: .
        ports: - 8000:5000"
  redis:
        image: "redis:alpine"
```

  4. Build and run app with Compose
     `$ docker compose up -d`
  5. Send HTTP requests using curl or browser (counter is increased)
  6. List local images `$ docker image ls`
  7. Stop Compose, bringing everything down
     `$ docker compose down`

Docker Compose: full example

- Add volume for app code, so that code can be modified on the fly without rebuilding the image
- Specify restart policy for containers in Compose file
  - Options: `on-failure[:max-retries]`, `always`, `unless-stopped`
- Start multiple replicas of same service using `deploy` specification, e.g.,

```
services:
  frontend:
    image: example/webapp
    deploy:
      mode: replicated
      replicas: 6
```

Kafka as Docker containers

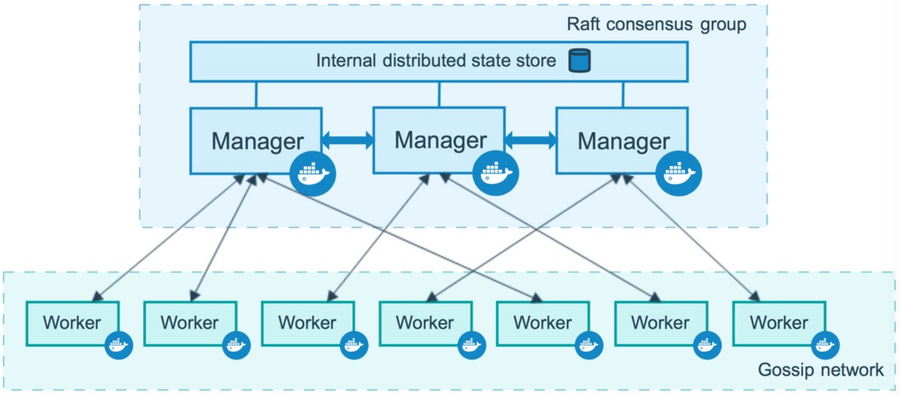
- Different packages already available, e.g.,
  - bitnami.com/stack/kafka/containers
    - Single container, Docker Compose with Zookeeper or KRaft
  - www.conduktor.io/kafka/how-to-start-kafka-using-docker
    - Docker Compose with Zookeeper, single and multiple Zookeeper and Kafka brokers

Docker Swarm

- Swarm mode: advanced feature of Docker to natively manage a cluster of Docker engines called a *swarm* docs.docker.com/engine/swarm/
- Tasks: containers running in a service
- Main features:
  - **Cluster management integrated with Docker**
  - **Declarative service model**
  - **Scaling**: number of tasks for each service (no auto-scaling)
  - **State reconciliation**: Swarm monitors cluster state and reconciles any difference wrt desired state
  - **Multi-host networking**: can specify overlay network for services
  - **Load balancing**: can expose service ports to an external load balancer; internally, the swarm lets you specify how to distribute containers among nodes
  - **Secure**: TLS authentication and encryption

Docker Swarm: architecture

- A swarm consists of multiple Docker engines which run in swarm mode
- Node: instance of Docker engine
  - **Manager node(s):** handles cluster management, including scheduling tasks to worker nodes
    - Multiple managers to improve fault tolerance
    - Raft as consensus algorithm to manage global cluster state
  - **Worker nodes** execute tasks



Docker Swarm: Swarm cluster

- Create swarm: start with 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 swarm: add worker node(s)

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

- Inspect swarm status

```
$ docker info
```

```
$ docker node ls

ID              HOSTNAME     STATUS      AVAILABILITY    MANAGER STATUS
<node.id1> *    manager1     Ready       Active          Leader
<node.id2>      worker1      Ready       Active
```

Docker Swarm: Swarm cluster

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

```
$ docker swarm leave
```

- After a node leaves the swarm, you can run `docker node rm` 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 swarm (from manager node)

```
$ docker service create -d --replicas 2 \
    --name helloworld alpine ping docker.com
```

  – Deploy service `helloworld` with 2 replicas; arguments `alpine ping docker.com` define service as an Alpine Linux container that executes `ping docker.com`

- List running services

```
$ docker service ls
```

```
ID            NAME          REPLICAS     IMAGE     COMMANDS
<service-id>  helloworld    2/2          alpine    ping docker.com
```

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  manager1    Running    Running …
<cont.id2>  helloworld.2  alpine:latest  worker1     Running    Running …
```

- Inspect container

```
$ docker ps <cont-id>
```

```
# 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 number of containers in service

```
$ docker service scale <service-id>=<number-of-tasks>
```

  - Swarm manager will enact the updates

- Apply *rolling updates* (i.e., update without downtime) to service

```
$ docker service update --limit-cpu 2 redis
$ docker service update --replicas 3 helloworld
```

- Roll back an update to previous version of service

```
$ docker service rollback [options] <service-id>
```

- Remove service

```
$ docker service rm <service-id>
```

References

- Docker Docs
- Nickoloff and Kuenzli, Docker in Action 2nd Edition, 2019