Introduction to Container & Docker

Agenda

* **Container & Docker:**
* **Introduction to Container & Docker:**
  * Container Benefits.
  * Docker Components.
  * Lab Resources.
* **Docker Engine:**
  * Docker Basic Command.
  * Labs: Docker Basic & Exercise.
* **Docker File:**
  * Usage Dockerfile.
  * Dockerfile Syntax & Commands.
  * Lab: Dockerfile & Exercise.
* **Docker Compose:**
  * Docker Compose Commands.
  * Lab: Docker Compose & Exercise.
* **Container orchestration platform**
* **Automation and pipeline:**
  * What's git?
  * What's gitlab?
  * Gitlab pipelines How can we implement Docker commands on gitlab pipelines?
  * Best practices for gitlab (Branching and Merge requests).

Container

Containers are a lightweight and portable store for an application and its dependencies.

What's the Diff: VMs vs Containers

VMs	Containers
Heavyweight	Lightweight
Limited performance	Native performance
Each VM runs in its own OS	All containers share the host OS
Hardware-level virtualization	OS virtualization
Startup time in minutes	Startup time in milliseconds
Allocates required memory	Requires less memory space
Fully isolated and hence more secure	Process-level isolation, possibly less secure

Docker Inc

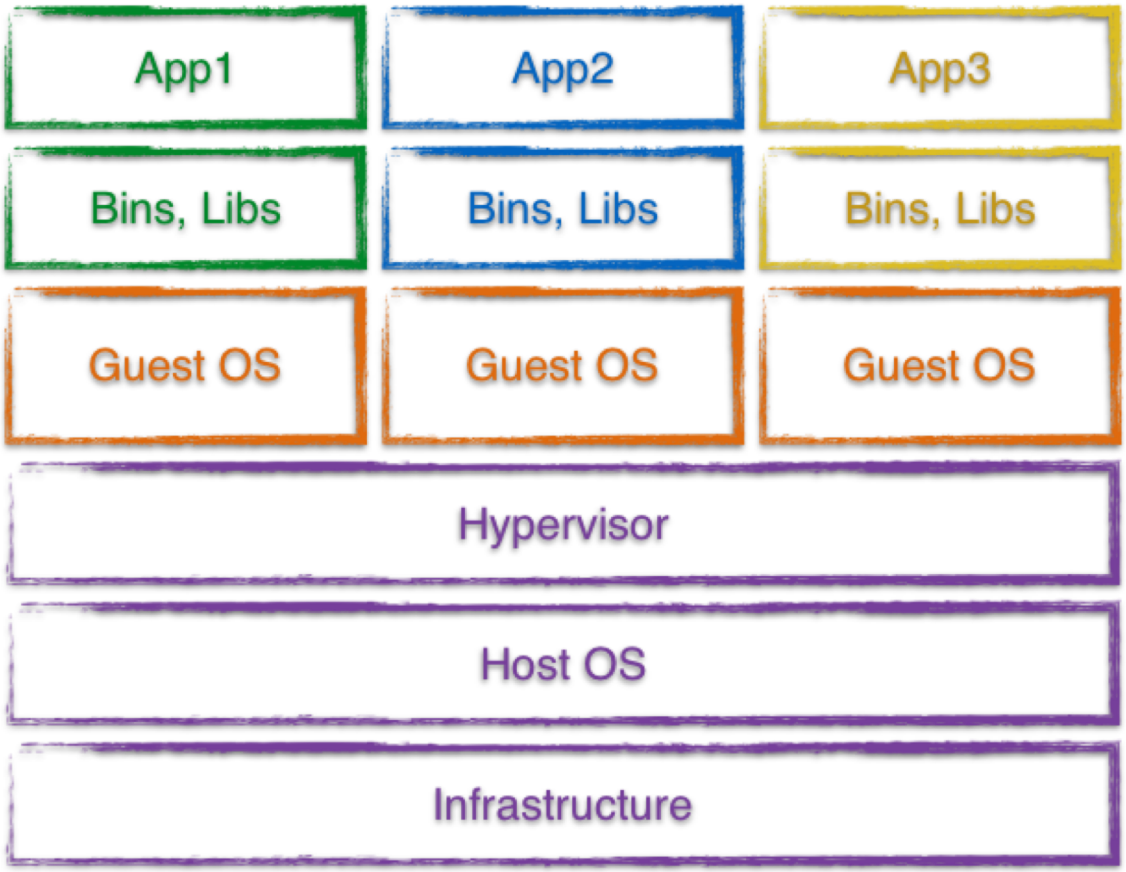
* Founded in 2009.
* Formerly dotCloud Inc.
* Primary sponsor of the Docker Project.
* Together with the community of maintainers and contributors, Docker aims to deliver open tools to help developers build applications with open APIs to help sysadmins better manage these applications.
* Services :- Docker Cloud, Docker Datacenter

Container Benefits

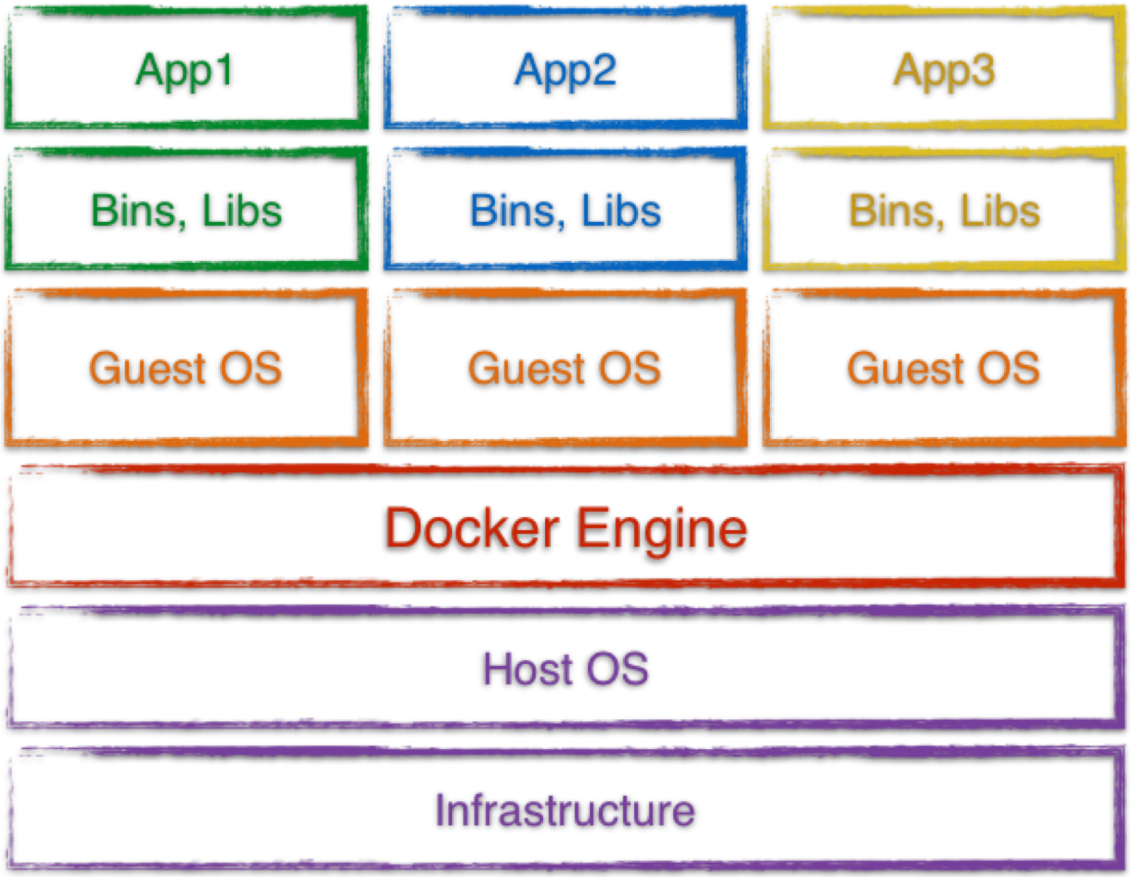
Containers are a streamlined way to build, test, deploy, and redeploy applications on multiple environments from a developer's local laptop to an on-premises data center and even the cloud. Benefits of containers include:

* **Less overhead.** Containers require less system resources than traditional or hardware virtual machine environments because they don't include operating system images.
* **Increased portability.** Applications running in containers can be deployed easily to multiple different operating systems and hardware platforms.
* **More consistent operation.** DevOps teams know applications in containers will run the same, regardless of where they are deployed.
* **Greater efficiency.** Containers allow applications to be more rapidly deployed, patched, or scaled.
* **"But it works on my machine"** stuff should be gone.

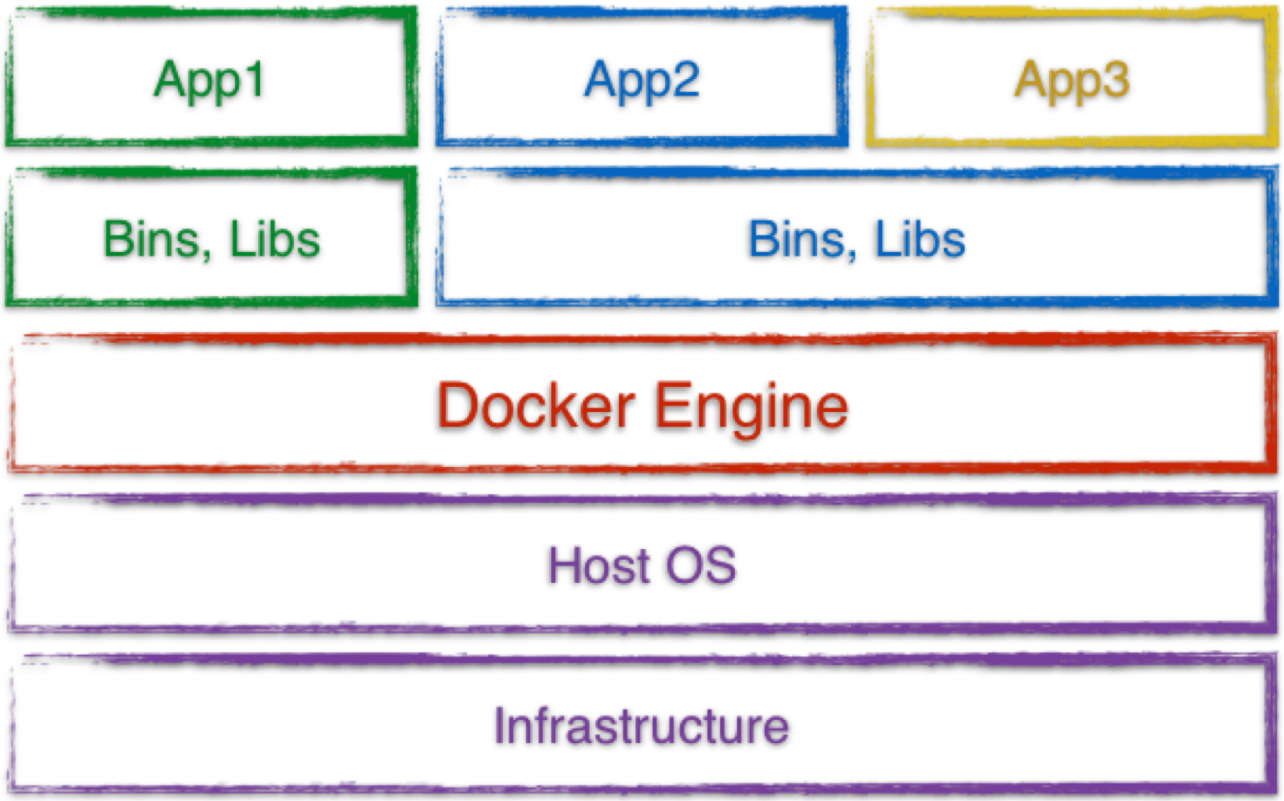
Virtual Machine



Container


App1
App2
App3
Bins, Libs
Bins, Libs
Bins, Libs
Guest OS
Guest OS
Guest OS
Docker Engine
Host OS
Infrastructure

Container


App1
App2
App3
Bins, Libs
Bins, Libs
Docker Engine
Host OS
Infrastructure

Docker Components

* Docker Engine :-
Create Docker images and run Docker containers. Including **Swarm** mode from 1.12.0-rc1

* Docker Compose :-
Defines applications built using multiple containers.

* Docker Hub :-
A hosted registry service for managing and building images.

* Docker Cloud :-
A hosted service for building, testing, and deploying Docker images to your hosts.

* Docker Trusted Registry :-
(DTR) stores and signs your images.

* Docker Machine :-
Automate container provisioning on your network or in the cloud. Available for Windows, Mac OS X, or Linux

Supported Platforms

Docker for Mac

A native application using the OS X sandbox security model which delivers all Docker tools to your Mac.

Docker for Windows

A native Windows application which delivers all Docker tools to your Windows computer.

Docker for Linux

Install Docker on a computer which already has a Linux distribution installed.

Your training Virtual Machine

#	IP	#	IP
1	10.254.145.226	11	10.254.145.218
2	10.254.145.141	12	10.254.145.175
3	10.254.145.10	13	10.254.145.88
4	10.254.145.20	14	10.254.145.214
5	10.254.145.223	15	10.254.145.232
6	10.254.145.231	16	10.254.145.212
7	10.254.145.96	17	10.254.145.180
8	10.254.145.140	18	10.254.145.225
9	10.254.145.71	19	10.254.145.224
10	10.254.145.78	20	10.254.145.203

Connecting to your Virtual Machine

* You need an SSH client.
  * On OS X, Linux, and other UNIX systems, just use ssh:

```
$ ssh  <KAUST-ID>@<ip-address>
```

  * On Windows, if you don't have an SSH client, you can download Putty from www.putty.org.

Checking your Virtual Machine

* Once logged in, make sure that you can run a basic Docker command:

```
$ docker version
Client:
 Version:           18.09.5
 API version:       1.39
 Go version:        go1.10.8
 Git commit:        e8ff056
 Built:             Thu Apr 11 04:43:57 2019
 OS/Arch:           linux/amd64
 Experimental:      false
Server: Docker Engine - Community
 Engine:
  Version:          18.09.5
  API version:      1.39 (minimum version 1.12)
  Go version:       go1.10.8
  Git commit:       e8ff056
  Built:            Thu Apr 11 04:10:53 2019
  OS/Arch:          linux/amd64
  Experimental:     false
```

Docker architecture

* The Docker daemon (or "Engine")
  Receives and processes incoming Docker API requests.

* The Docker client
  Talks to the Docker daemon via the Docker API. We'll use mostly the CLI embedded within the docker binary.

Docker Basics

* Check the version of docker

```
$ docker --version
```

* Run the first docker container… Hello, World!

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

* Check the process of docker container

```
$ docker ps
$ docker ps –a
```

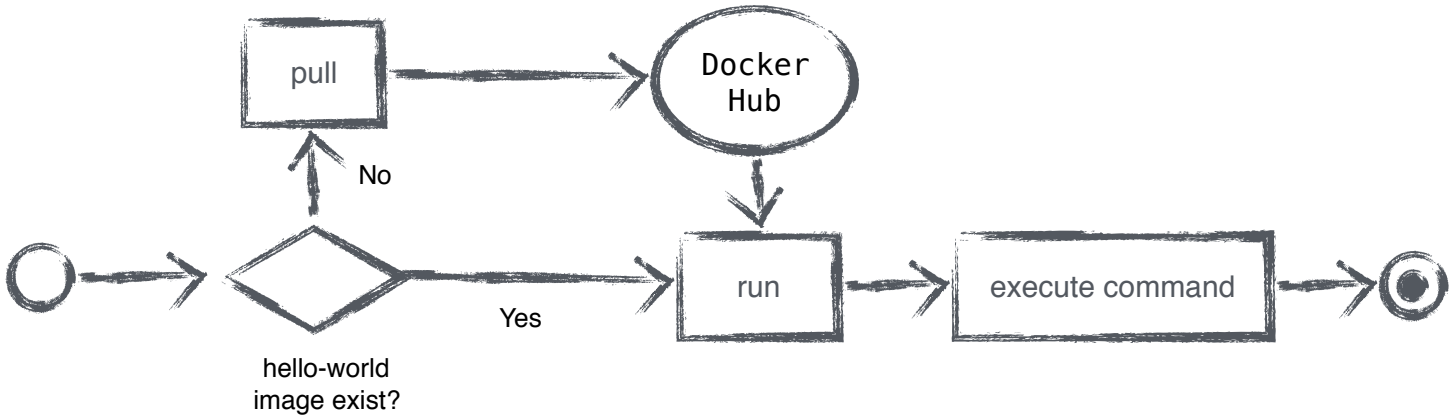
* See the list of Docker Images.

```
$ docker images
```

‘docker run’

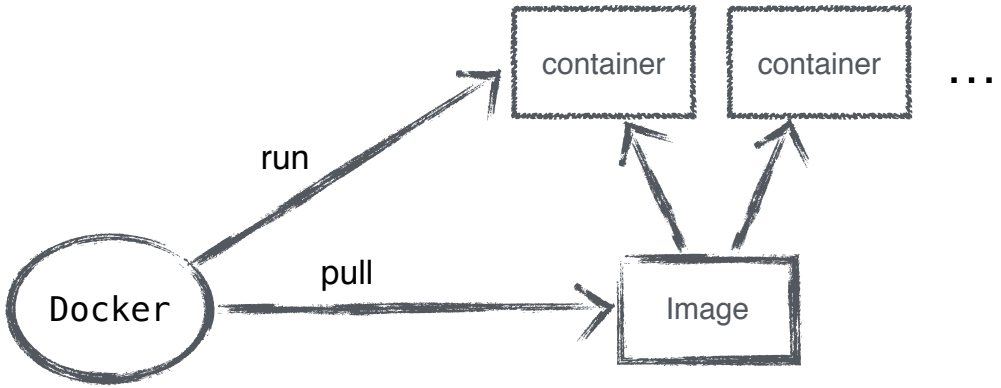
Usage: docker [option] COMMAND [argument …]

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



X

Container State



container

- long running process
- service

Container management commands

command	description
`docker create` *image* [*command*] `docker run` *image* [*command*]	create the container = `create` + `start`
`docker rename` *container new_name* `docker update` *container*	rename the container update the container config
`docker start` *container...* `docker stop` *container...* `docker kill` *container...* `docker restart` *container...*	start the container graceful[2] stop kill (SIGKILL) the container = `stop` + `start`
`docker pause` *container...* `docker unpause` *container...*	suspend the container resume the container
`docker rm` [`-f`[3]] *container...*	destroy the container

docker run — interactive mode

* By default containers are non-interactive
  * stdin is closed immediately
  * terminal signals are not forwarded

```
$docker run -t debian bash
root@6fecc2e8ab22:/# date
^C
$
```

* With -i the container runs interactively
  * stdin is usable
  * terminal signals are forwarded to the container

Inspecting the container

command	description
`docker ps`	list running containers
`docker ps -a`	list all containers
`docker logs` [`-f`[5]] *container*	show the container output (*stdout+stderr*)
`docker top` *container* [*ps options*]	list the processes running inside the containers[6]
`docker stats` [*container*]	display live usage statistics[7]
`docker diff` *container*	show the differences with the image (modified files)
`docker port` *container*	list port mappings
`docker inspect` *container...*	show low-level infos (in json format)

[5] with `-f`, `docker logs` follows the output (à la `tail -f`)

[6] `docker top` is the equivalent of the `ps` command in unix

[7] `docker stats` is the equivalent of the `top` command in unix

Interacting with the container

command	description
`docker attach` *container*	attach to a running container (stdin/stdout/stderr)
`docker cp` *container:path hostpath*|- `docker cp` *hostpath*|- *container:path*	copy files from the container copy files into the container
`docker export` *container*	export the content of the container (tar archive)
`docker exec` *container args...*	run a command in an existing container (**useful** for debugging)
`docker wait` *container*	wait until the container terminates and return the exit code
`docker commit` *container image*	commit a new docker image (snapshot of the container)

Docker Commit

```
$ docker run --name my-container -t -i debian
root@3b397d383faf:/# cat >> /etc/bash.bashrc <<EOF
> echo 'hello!'
> EOF
root@3b397d383faf:/# exit
$ docker start --attach my-container
my-container
hello!
root@3b397d383faf:/# exit
$ docker diff my-container
C /etc
C /etc/bash.bashrc
A /.bash_history
C /tmp
$ docker commit my-container hello
a57e91bc3b0f5f72641f19cab85a7f3f860a1e5e9629439007c39fd76f37c5dd
$ docker rm my-container
my-container
$ docker run --rm -t -i hello
hello!
root@386ed3934b44:/# exit
```

Docker Basics

* In your Docker environment, just run the following command

```
$ docker run busybox echo hello world
hello world
```

* We used one of the smallest, simplest images available: busybox.
* busybox is typically used in embedded systems (phones, routers...)
* We run a single process and echo'ed hello world.

A more useful container

* Let's run a more exciting container:

```
$ docker run -it ubuntu bash
root@04c0bb0a6c07:/#
```

* -it is shorthand for -i -t.
  * -i tells Docker to connect us to the container's stdin.
  * -t tells Docker that we want a pseudo-terminal.

* Try to run figlet in our container:

```
root@04c0bb0a6c07:/# figlet hello
bash: figlet: command not found
root@04c0bb0a6c07:/# apt-get update
root@04c0bb0a6c07:/#apt-get install figlet –y
root@04c0bb0a6c07:/#exit
$ docker commit 04c0bb0a6c07 hello
```

Image tags

* A docker tag is made of two parts: “REPOSITORY:TAG”
* The TAG part identifies the version of the image. If not provided, the default is “:latest”

```
$ docker images
REPOSITORY  TAG           IMAGE ID      CREATED        VIRTUAL SIZE
debian      8             835c4d274060  2 weeks ago    122.6 MB
debian      8.0           835c4d274060  2 weeks ago    122.6 MB
debian      jessie        835c4d274060  2 weeks ago    122.6 MB
debian      rc-buggy      350a74df81b1  7 months ago   159.9 MB
debian      experimental  36d6c9c7df4c  7 months ago   159.9 MB
debian      6.0.9         3b36e4176538  7 months ago   112.4 MB
debian      squeeze       3b36e4176538  7 months ago   112.4 MB
debian      wheezy        667250f9a437  7 months ago   115 MB
debian      latest        667250f9a437  7 months ago   115 MB
debian      7.5           667250f9a437  7 months ago   115 MB
debian      unstable      24a4621560e4  7 months ago   123.6 MB
debian      testing       7f5d8ca9fdcf  7 months ago   121.8 MB
debian      stable        caa04aa09d69  7 months ago   115 MB
debian      sid           f3d4759f77a7  7 months ago   123.6 MB
debian      7.4           e565fbbc6033  9 months ago   115 MB
debian      7.3           b5fe16f2ccba  11 months ago  117.8 MB
```

Inputs/Outputs

* Data volumes (persistent data)
  * mounted from the host filesystem
* Publishing ports (NAT)

docker run — mount external volumes

* Command

```
$ docker run -v /hostpath:/containerpath[:ro]
```

* -v mounts the location /hostpath from the host filesystem at the location /containerpath inside the container
  With the ":ro" suffix, the mount is read-only
* Purposes:
  * store persistent data outside the container
  * provide inputs: data, config files, . . . (read-only mode)

Example:

```
docker run -i -t -v /tmp/data:/data:ro ubuntu
```

docker run — publish a TCP port

* Command

```
$ docker run -p hostport:containerport image-name
```

* Containers are deployed in a private network, they are not reachable from the outside (unless a redirection is set up).

* redirect incoming connections to the TCP port hostport of the host to the TCP port containerport of the container
The listening socket binds to 0.0.0.0 (all interfaces) by default or to ipaddr if given

```
$docker run -d -p 80:80 nginx
```

Labs

1:1

* Create nginx web server with docker on port 8080
  curl localhost:8080

1:2

* Install vim
* Make changes in /usr/share/nginx/html/index.html
  From Welcome to nginx to Welcome to KAUST

1:3

* Commite the change to “mynginx”

1:4

* Login to docker hub
  Userid: kaustdocker
  Password: 123456789
* Tag mynginx to kaustdocker/nginx:<kaust_id>
* Push image to docker hub

1:5

* Create folder /tmp/data
* Mount /tmp/data to /home/ in image
  kaustdocker/nginx:<kaust_id>

Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

Using docker build users can create an automated build that executes several command-line instructions in succession.

Dockerfile

* Dockerfile instructions are executed in order.
* Each instruction creates a new layer in the image.
* Instructions are cached. If no changes are detected then the instruction is skipped and the cached layer used.
* The FROM instruction MUST be the first non-comment instruction.
* Lines starting with # are treated as comments.
* You can only have one CMD and one ENTRYPOINT instruction in a Dockerfile

Instructions affecting the image filesystem

instruction	description
`FROM` *image*|`scratch`	base image for the build
`COPY` *path dst*	copy *path* from the context into the container at location *dst*
`ADD` *src dst*	same as `COPY` but untar archives and accepts http urls
`RUN` *command*	run an arbitrary command inside the container

Instructions setting the default container config

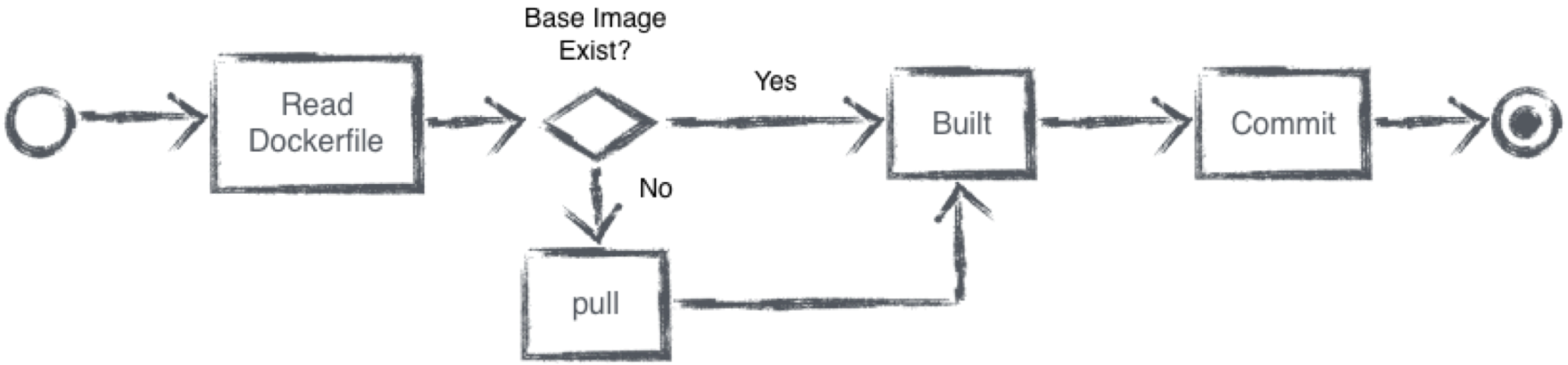
instruction	description
`CMD` *command* `ENTRYPOINT` *command*	command run inside the container entrypoint[13]
`USER` *name[:group]* `WORKDIR` *path* `ENV` *name*="*value*"...	user running the command working directory environment variables
`STOPSIGNAL` *signal*	signal to be sent to terminate the container*(instead of SIGTERM)*
`HEALTHCHECK CMD` *command*	test command to check if the container works well
`EXPOSE` *port...* `VOLUME` *path...*	listened TCP/UDP ports mount-point for external volumes
`LABEL` *name*="*value*"...	arbitrary metadata

Dockerfile Examples

```
FROM ubuntu:14.04
MAINTAINER Hassan
RUN apt-get update && apt-get install -y nodejs npm
ADD ./src/ /src
RUN cd /src; npm install
CMD ["node", "/src/index.js"]
```

Dockerfile
Usage

```
$ docker built -t my_Image .
```



Dockerfile

Multi-stage build

```
#======= Stage 1: build the app from sources =======#
FROM debian:stretch AS builder
# update the package lists an install the build dependencies
RUN apt-get -qqy update
RUN apt-get -qqy install gcc make libacme-dev

# install the sources in /opt/src and build them
COPY . /opt/src
RUN cd /opt/src && ./configure && make
# install the files in a tmp dir and make an archive that we can
deploy elsewhere
RUN cd /opt/src && make install DESTDIR=/tmp/dst \
&& cd /tmp/dst && tar czvf /tmp/myapp.tgz .

#======= Stage 2: final image ================#
FROM debian:stretch
# update the package lists and install the runtime dependencies
RUN apt-get -qqy update
RUN apt-get -qqy install libacme1.0
# install the app built in stage 1
COPY --from=builder /tmp/myapp.tgz /tmp/
RUN cd / && tar zxf /tmp/myapp.tgz
CMD ["myapp"]
```

Labs

2:1

* Using Dockerfile
  * build nginx web server with docker on port 8080 and tag it “mynginx:lab2”
  * Install vim
  * Copy https://raw.githubusercontent.com/kaustdocker/docker101/master/index.html
  * to /usr/share/nginx/html/

Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration

group.yml

```yaml
name: counter

containers:
  web:
    build: .
    command: python app.py
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    links:
      - redis
  redis:
    image: redis:latest
```

Docker Compose

Commands

```
docker-compose start
docker-compose stop
docker-compose pause
docker-compose unpause
docker-compose ps
docker-compose up
docker-compose down
```

Docker Compose

Reference

Building

```
web:
  # build from Dockerfile
  build: .

  # build from custom Dockerfile
  build:
    context: ./dir
    dockerfile: Dockerfile.dev


  # build from image
  image: ubuntu
  image: ubuntu:14.04
  image: tutum/influxdb
```

Ports

```
  ports:
    - "3000"
    - "8000:80"  # guest:host
 # expose ports to linked services (not to host)
  expose: ["3000"]
```

Environment variables

```
 # environment vars
  environment:
    RACK_ENV: development
  environment:
    - RACK_ENV=development

  # environment vars from file
  env_file: .env
  env_file: [.env, .development.env]
```

Commands

```
# command to execute
command: bundle exec thin -p 3000
 command: [bundle, exec, thin, -p, 3000]

# override the entrypoint
entrypoint: /app/start.sh
 entrypoint: [php, -d, vendor/bin/phpunit]
```

Docker Compose Examples

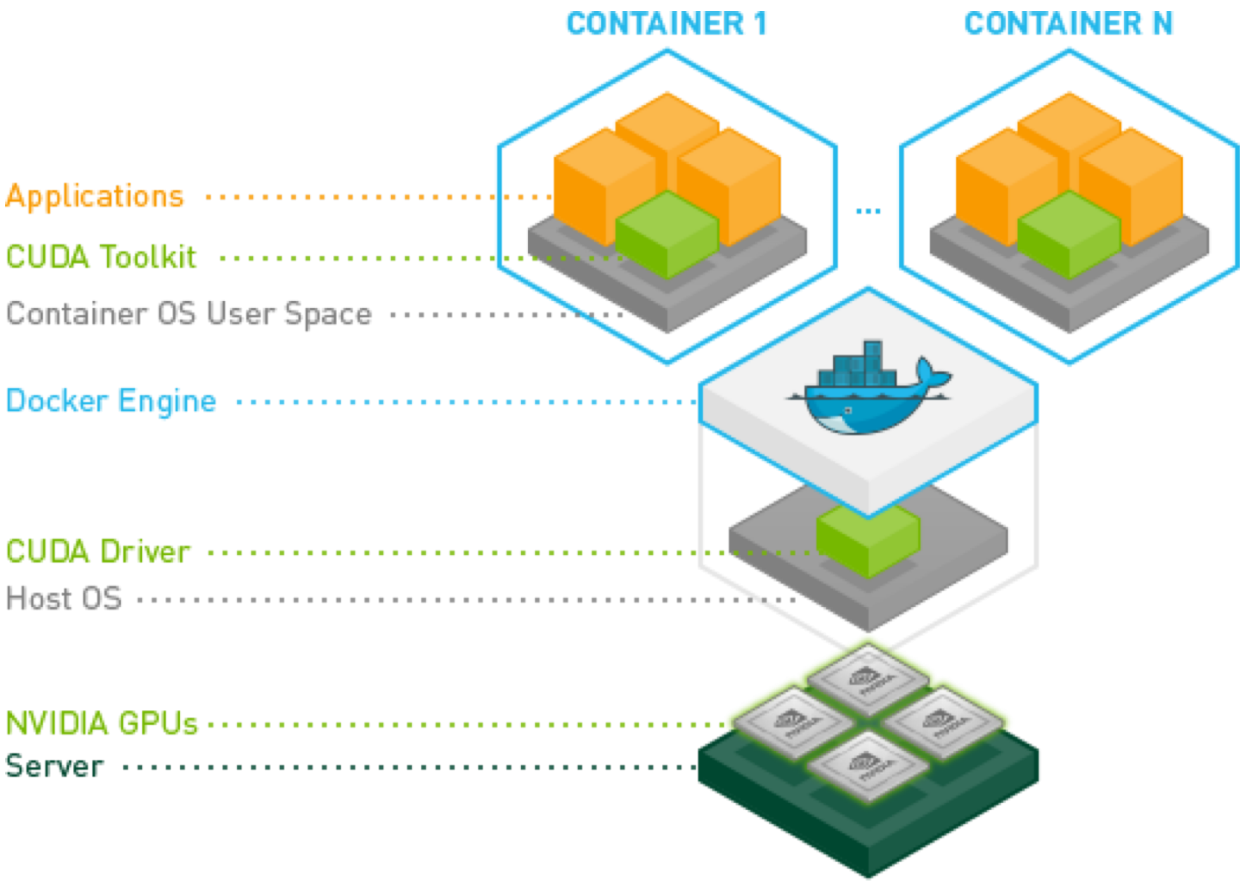
```
version: '3.3’

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD: wordpress
       WORDPRESS_DB_NAME: wordpress
volumes:
    db_data: {}
```

Exercise

https://docs.docker.com/compose/gettingstarted/

https://github.com/kaustdocker/docker101/tree/master/composetest

NVIDIA Container Runtime for Docker



NVIDIA Container Runtime for Docker
Installation

```
# Add the package repositories
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | \
  sudo apt-key add -
distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | \
  sudo tee /etc/apt/sources.list.d/nvidia-docker.list
sudo apt-get update


# Install nvidia-docker2 and reload the Docker daemon configuration
sudo apt-get install -y nvidia-docker2
sudo pkill -SIGHUP dockerd
```

NVIDIA Container Runtime for Docker
Usage

In docker command :

```
docker run --runtime=nvidia --rm nvidia/cuda:9.0-base nvidia-smi
```

In docker-compose :

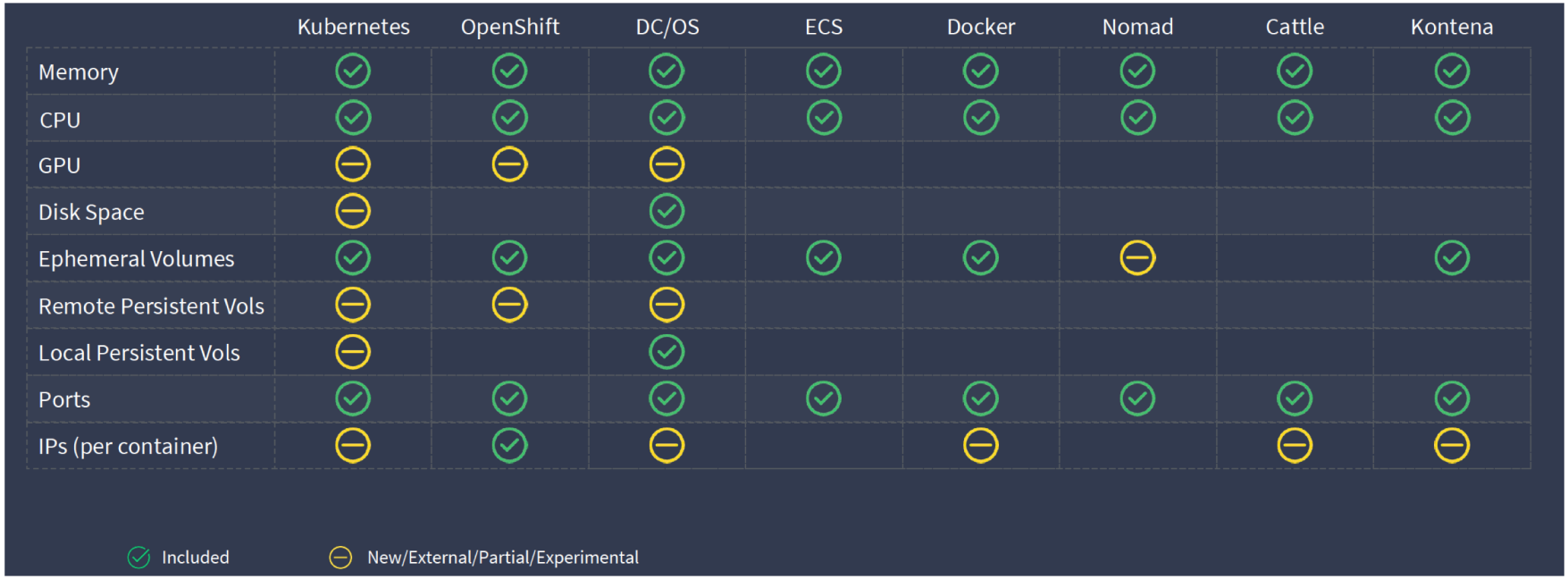
```
---
version: "2.3"
services:
  paraview:
    hostname: paraview
    tty: true
    image: kcr.kaust.edu.sa/apps/paraview
    runtime: nvidia
    environment:
      - DISPLAY=unix${DISPLAY}
      - NVIDIA_VISIBLE_DEVICES=all
    working_dir:
      "/home/${USER}/"
    volumes:
      - "/var/lib/sss/pipes/:/var/lib/sss/pipes/:rw"
      - "/home/${USER}/:/home/${USER}/"
      - "/tmp/.X11-unix:/tmp/.X11-unix:rw"
    command:
      /bin/su - $USER -c "/opt/ParaView-5.6.0-MPI-Linux-64bit/bin/paraview"
```

Container orchestration

Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments. Software teams use container orchestration to control and automate many tasks:

* Provisioning and deployment of containers
* Redundancy and availability of containers
* Scaling up or removing containers to spread application load evenly across host infrastructure
* Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
* Allocation of resources between containers
* External exposure of services running in a container with the outside world
* Load balancing of service discovery between containers
* Health monitoring of containers and hosts
* Configuration of an application in relation to the containers running it

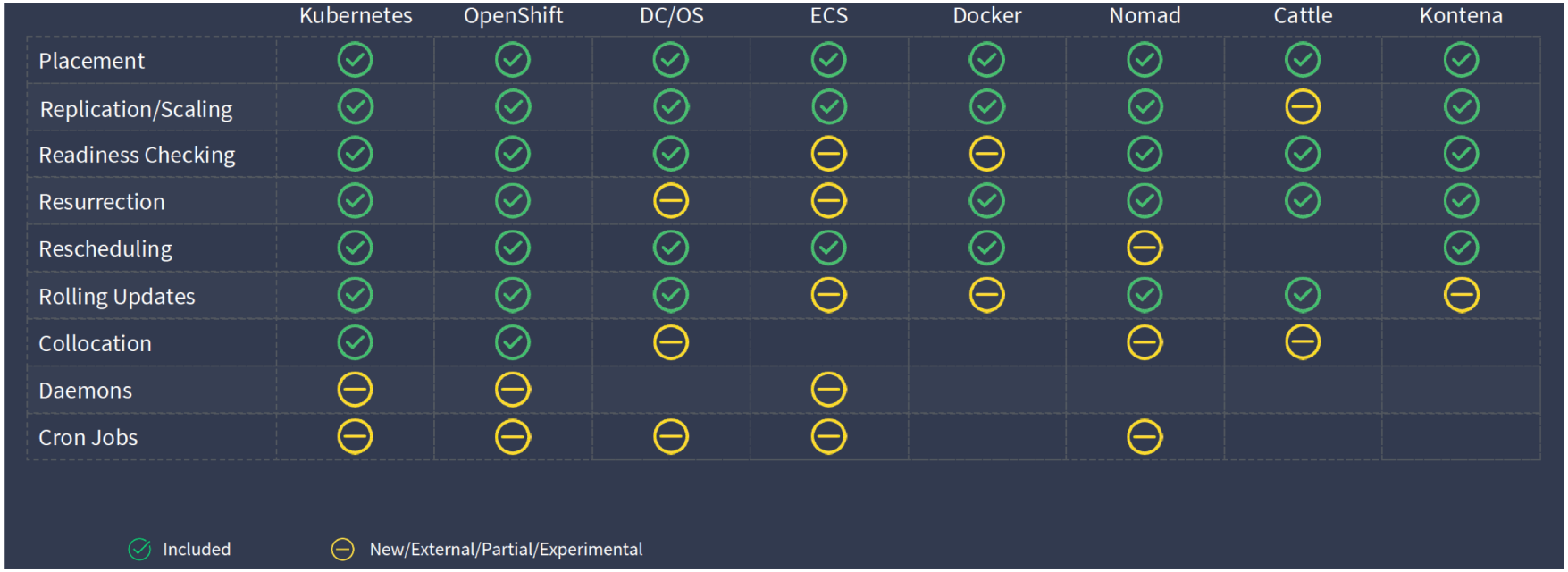
CONTAINER PLATFORM COMPARISON (06/2017)



	Kubernetes	OpenShift	DC/OS	ECS	Docker	Nomad	Cattle	Kontena
Memory	✓	✓	✓	✓	✓	✓	✓	✓
CPU	✓	✓	✓	✓	✓	✓	✓	✓
GPU	⊖	⊖	⊖					
Disk Space	⊖		✓					
Ephemeral Volumes	✓	✓	✓	✓	✓	⊖		✓
Remote Persistent Vols	⊖	⊖	⊖					
Local Persistent Vols	⊖		✓					
Ports	✓	✓	✓	✓	✓	✓	✓	✓
IPs (per container)	⊖	✓	⊖		⊖		⊖	⊖

✓ Included ⊖ New/External/Partial/Experimental

CONTAINER PLATFORM COMPARISON (06/2017)



Nvidia,TensorFlow & Kubernetes