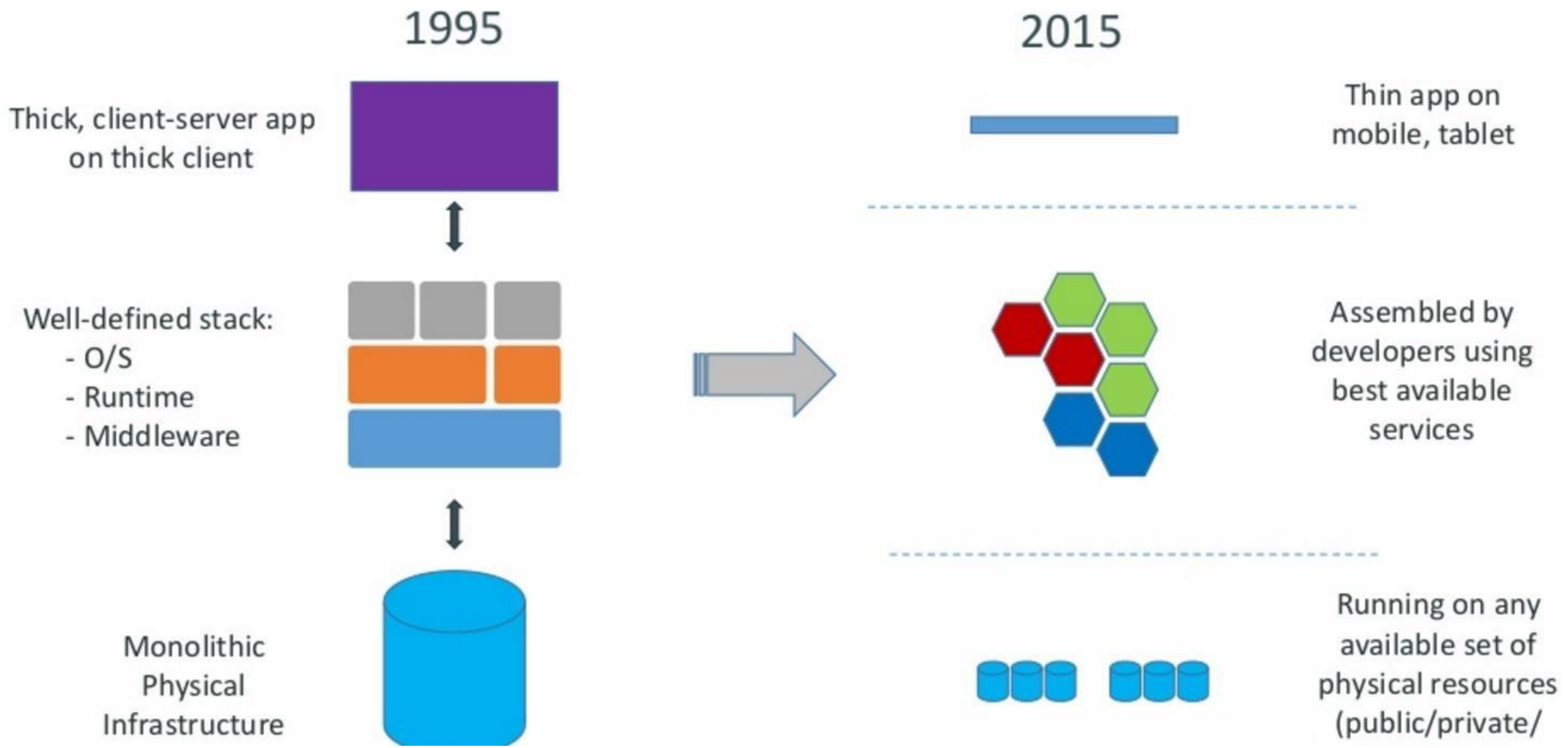
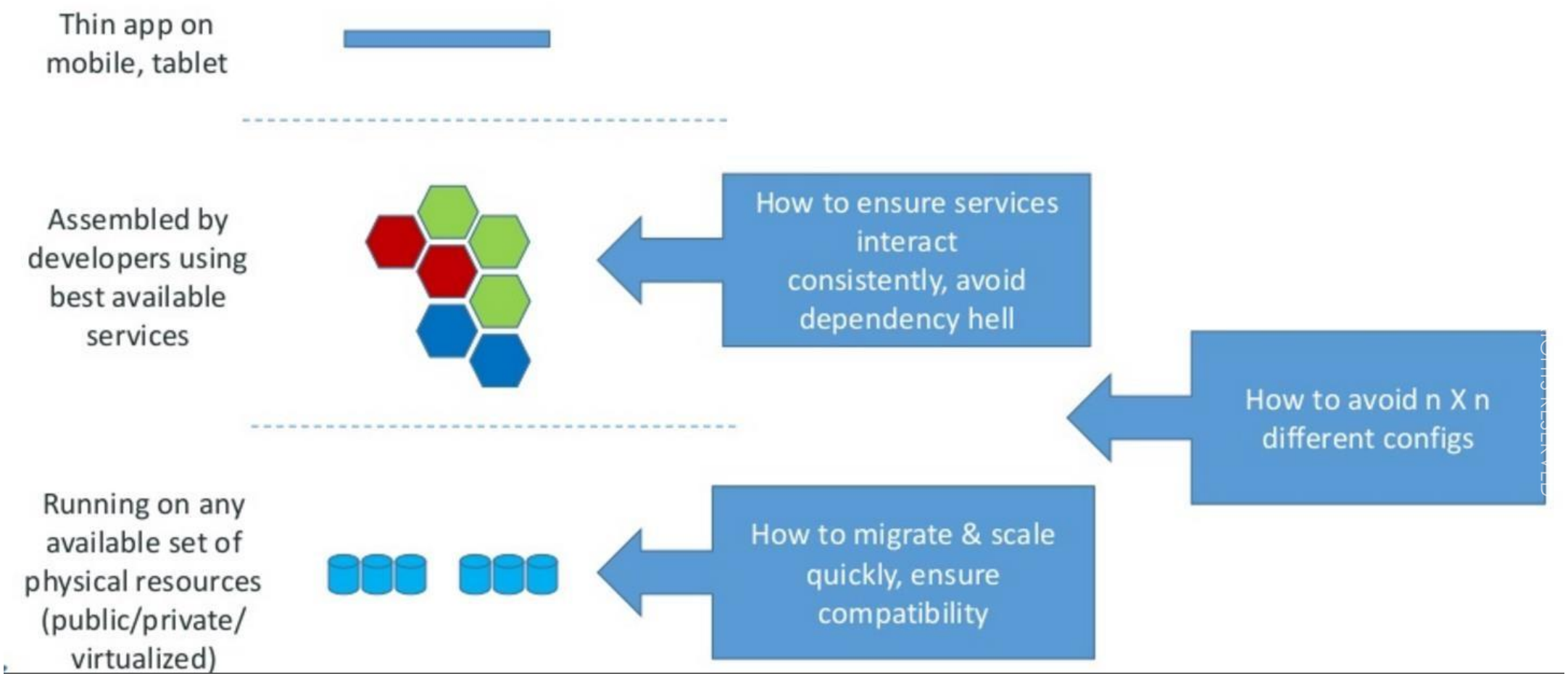
Containers & Dockers

Evolution of Dockers & Containers

Motivation



Motivation –the challenge

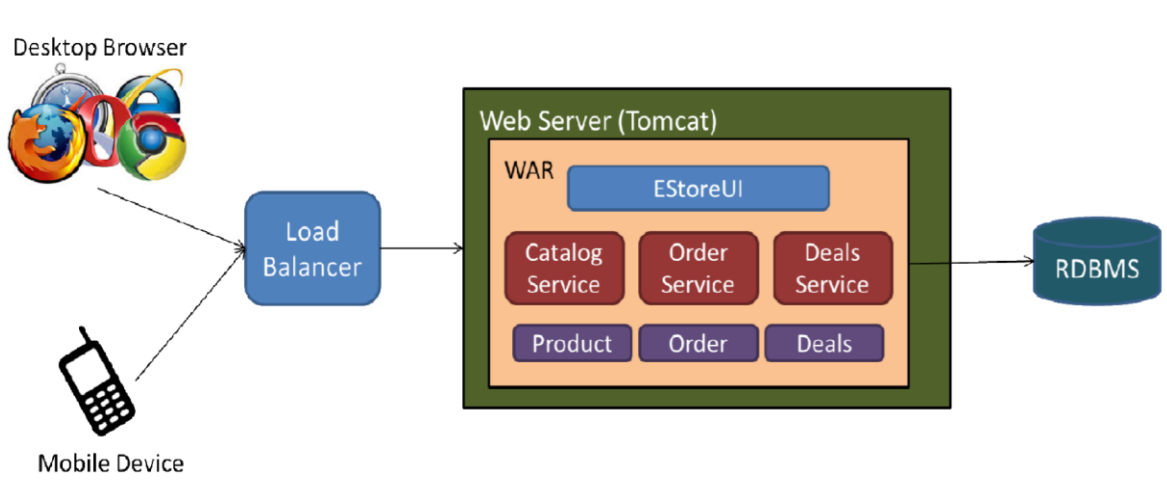


Overview of Microservices

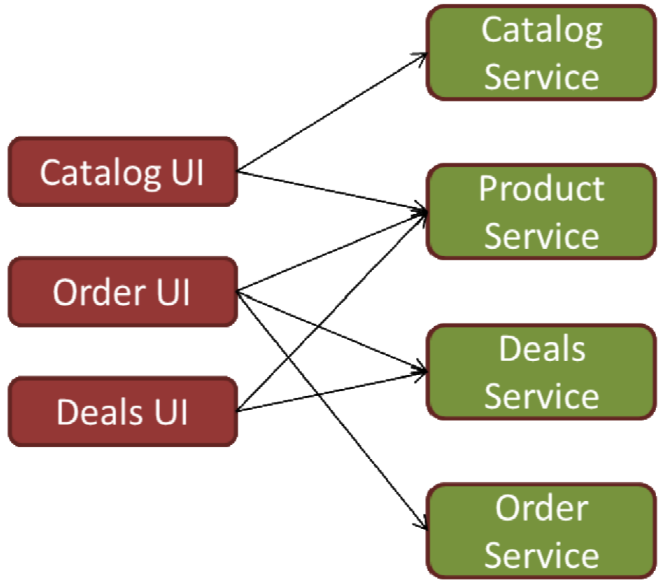
- Microservices are increasingly used in the development world as developers work to create larger, more complex applications that are better developed and managed as a combination of smaller services that work cohesively together for more extensive, application-wide functionality.

- Microservices are an architectural style that develops a single application as a set of small services. Each service runs in its own process. The services communicate with clients, and often each other, using lightweight protocols, often over messaging or HTTP.

Monolithic vs Microservices



Monolithic

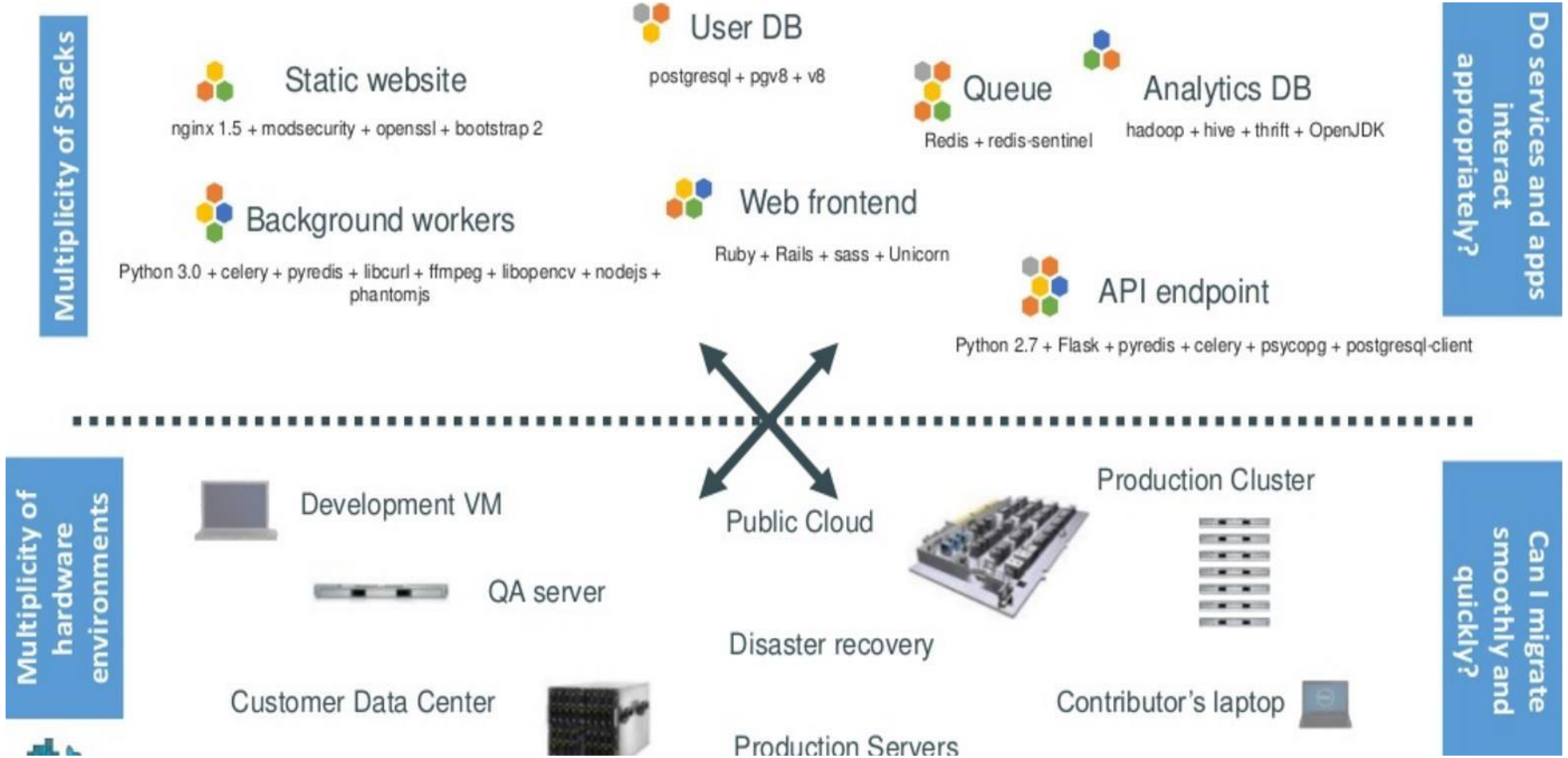


Microservice

Features of Microservices

- Componentization via Services
- Organised around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralised Governance
- Decentralised Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

The challenge continued



But it works fine at my end

May be your machine is not the same as mine

There is a problem with your application. It is not working at our end.

Developer

Operations

Looking for all kinds of solutions…

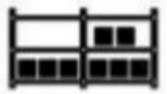
- Too many to consider

	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Understanding….an analogy

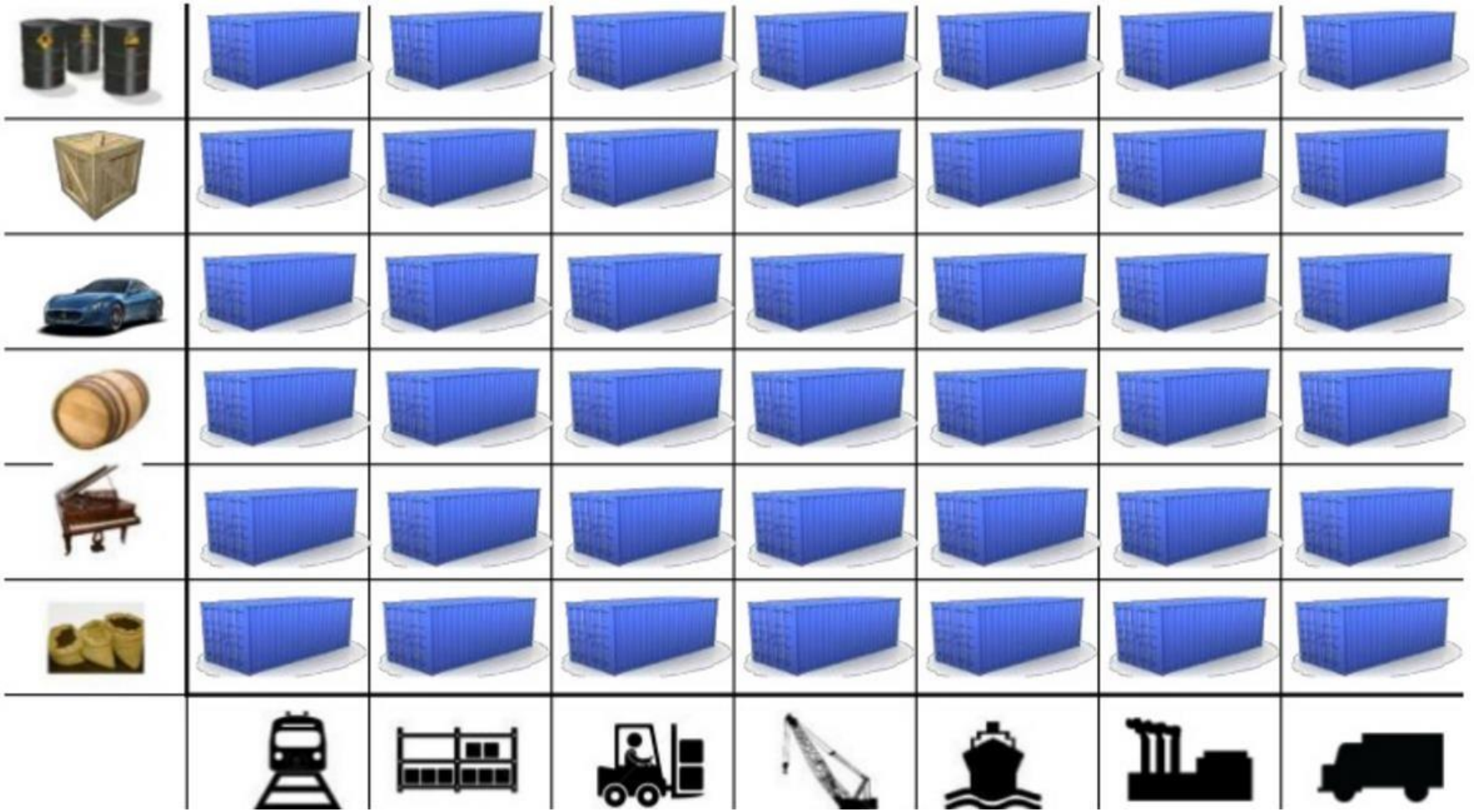
…cargo transport pre-1960

What are the possibilities

	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
							

SOLUTION—shipping containers

This solved the problem



Today shipping is done with containers

- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

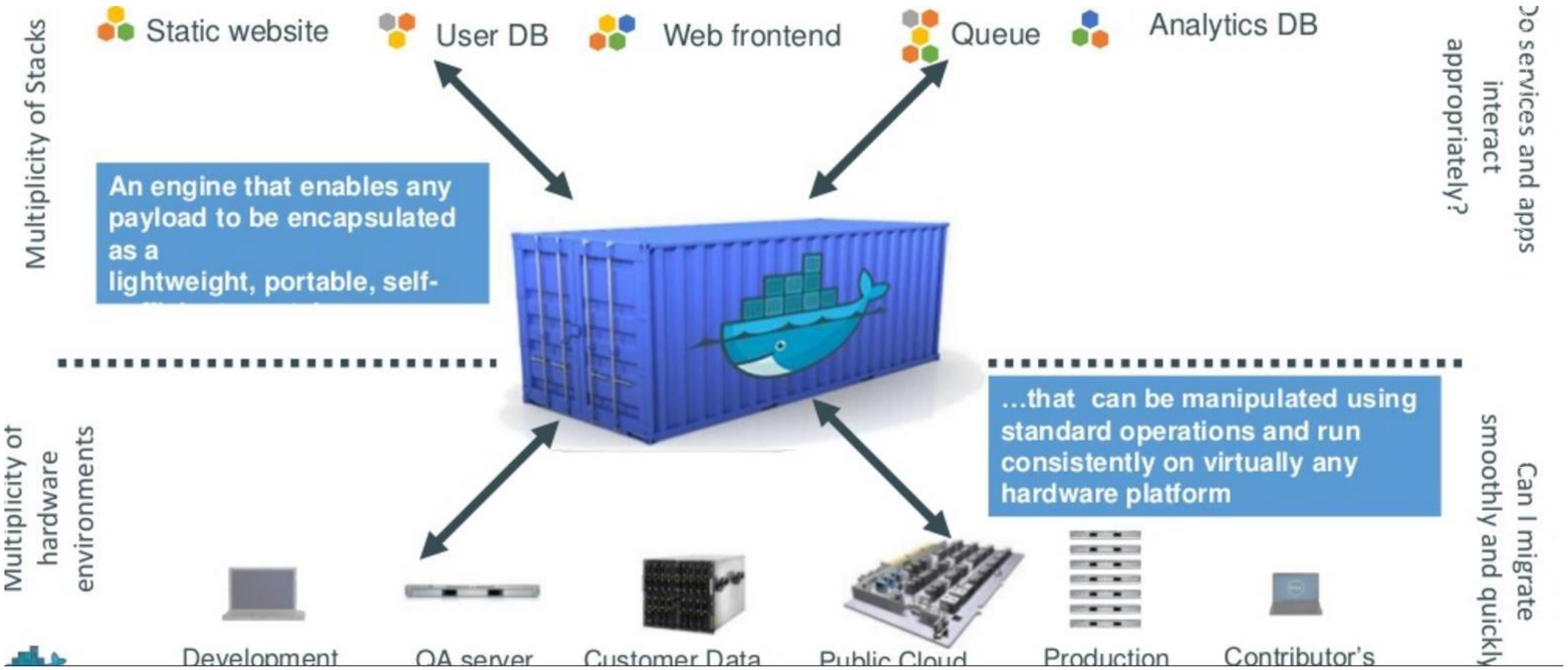
→ massive globalizations

- 5000 ships deliver 200M containers per year

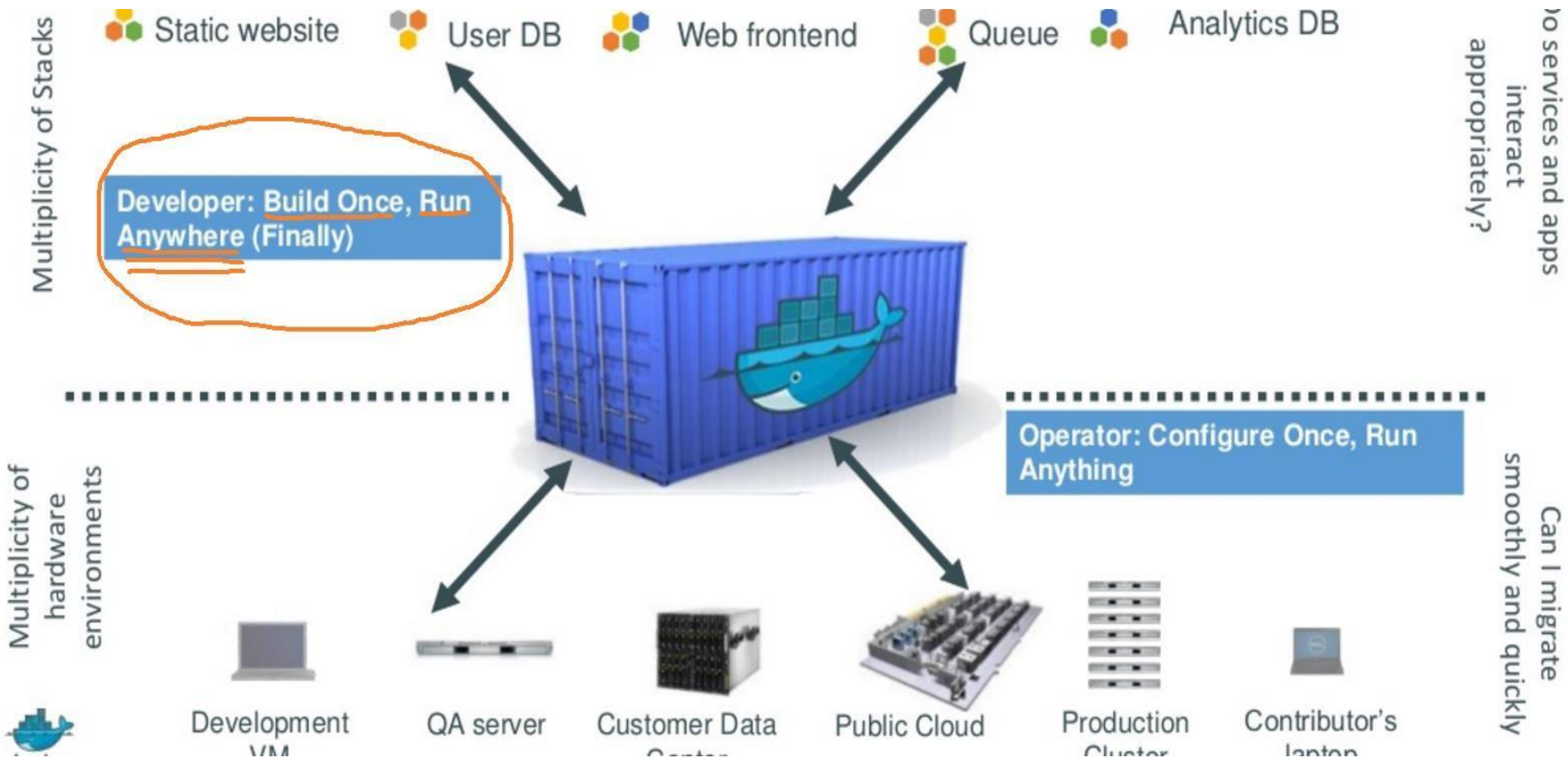
How does this container idea translate to our Problems

	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

How does this container idea translate to our problem—container for code????



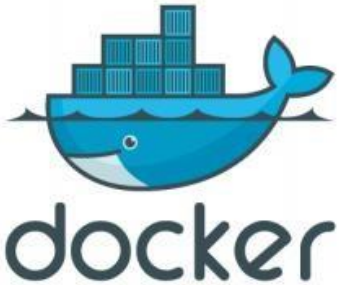
Do once run anywhere



Docker's container ---the concept (and relation to our shipping container)

	Physical Containers	Docker
Content Agnostic	The same container can hold almost any type of cargo	Can encapsulate any payload and its dependencies
Hardware Agnostic	Standard shape and interface allow same container to move from ship to train to semi-truck to warehouse to crane without being modified or opened	Using operating system primitives (e.g. LXC) can run consistently on virtually any hardware—VMs, bare metal, openstack, public IAAS, etc.—without modification
Content Isolation and Interaction	No worry about anvils crushing bananas. Containers can be stacked and shipped together	Resource, network, and content isolation. Avoids dependency hell
Automation	Standard interfaces make it easy to automate loading, unloading, moving, etc.	Standard operations to run, start, stop, commit, search, etc. Perfect for devops: CI, CD, autoscaling, hybrid clouds
Highly efficient	No opening or modification, quick to move between waypoints	Lightweight, virtually no perf or start-up penalty, quick to move and manipulate
Separation of duties	Shipper worries about inside of box, carrier worries about outside of box	Developer worries about code. Ops worries about infrastructure.

Docker supported in many Cloud platforms



Docker container—developer viewpoint

Build once…run anywhere

- A clean, safe, hygienic and portable runtime environment for your app.
- No worries about missing dependencies, packages and other pain points during subsequent deployments.
- Run each app in its own isolated container, so you can run various versions of libraries and other dependencies for each app without worrying
- Automate testing, integration, packaging…anything you can script
- Reduce/eliminate concerns about compatibility on different platforms, either your own or your customers.
- Cheap, zero-penalty containers to deploy services? A VM without the overhead of a VM? Instant replay and reset of image snapshots? That's the power of Docker

Developer viewpoint---(doesn't this quote remind you of Java virtual machine)

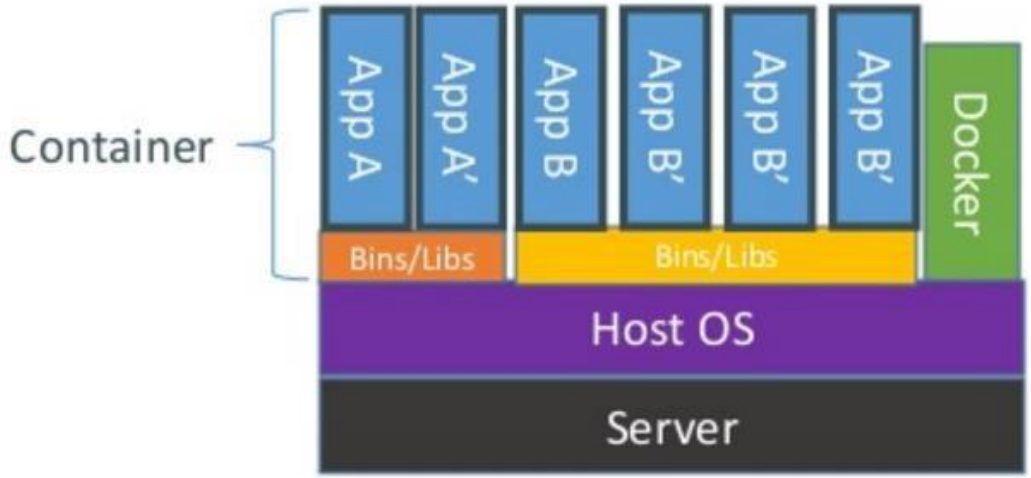
"Docker interests me because it allows simple environment isolation and repeatability. I can create a run-time environment once, package it up, then run it again on any other machine. Furthermore, everything that runs in that environment is isolated from the underlying host (much like a virtual machine). And best of all, everything is fast and simple."

-Gregory Szorc, Mozilla Foundation

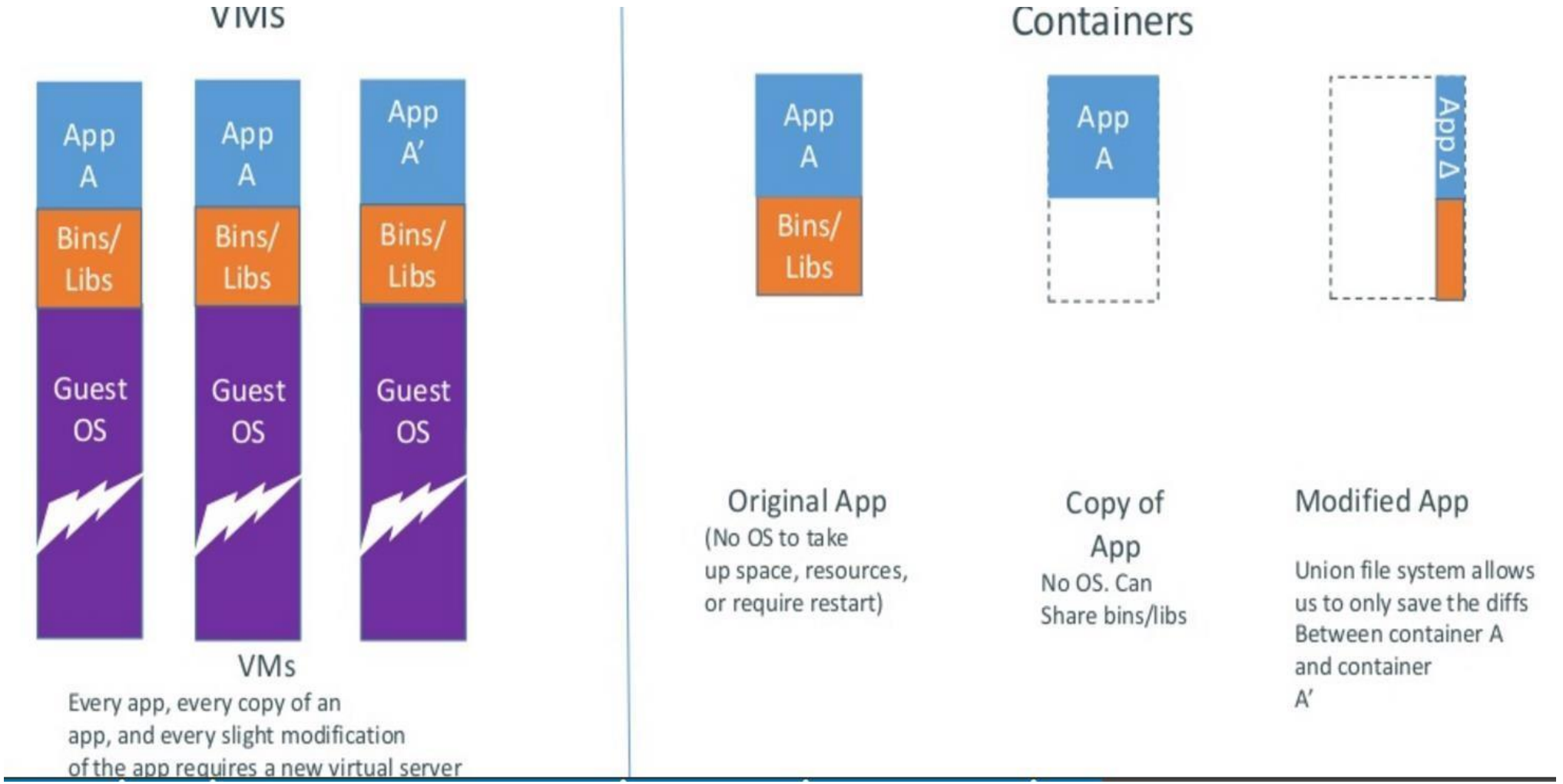
http://gregoryszorc.com/blog/2013/05/19/using-docker-to-build-firefox/

How does Docker containers work?

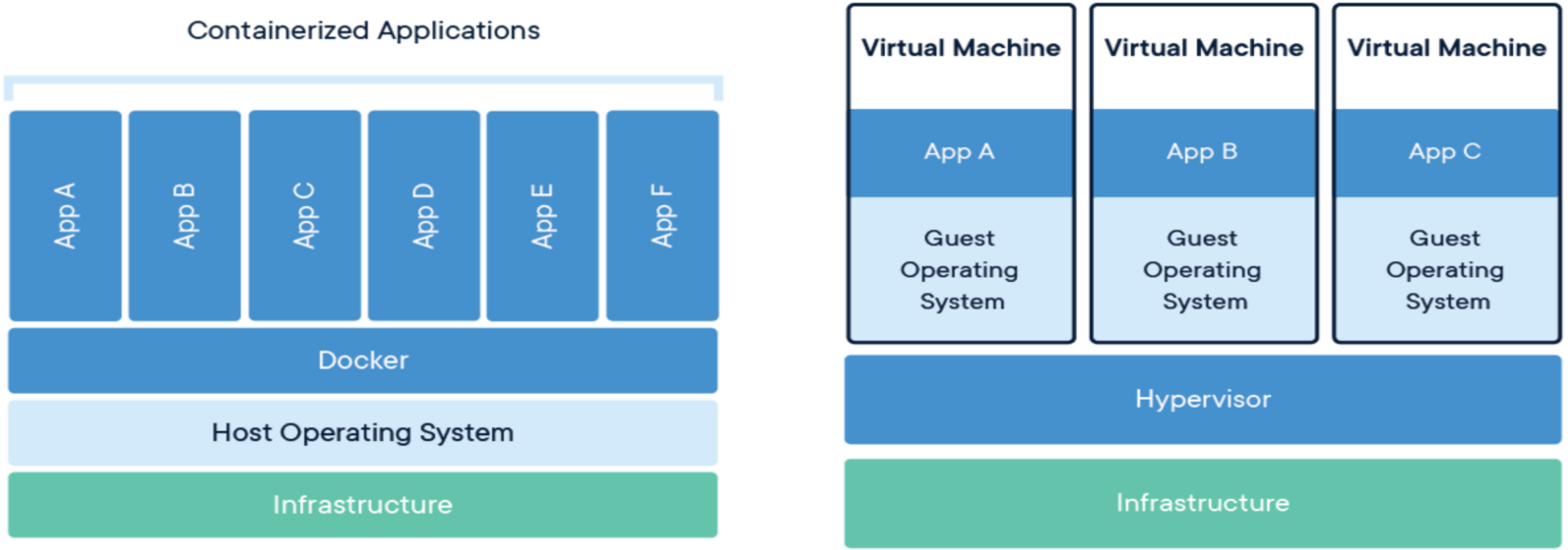
Containers are isolated,
but share OS and, where
appropriate, bins/libraries



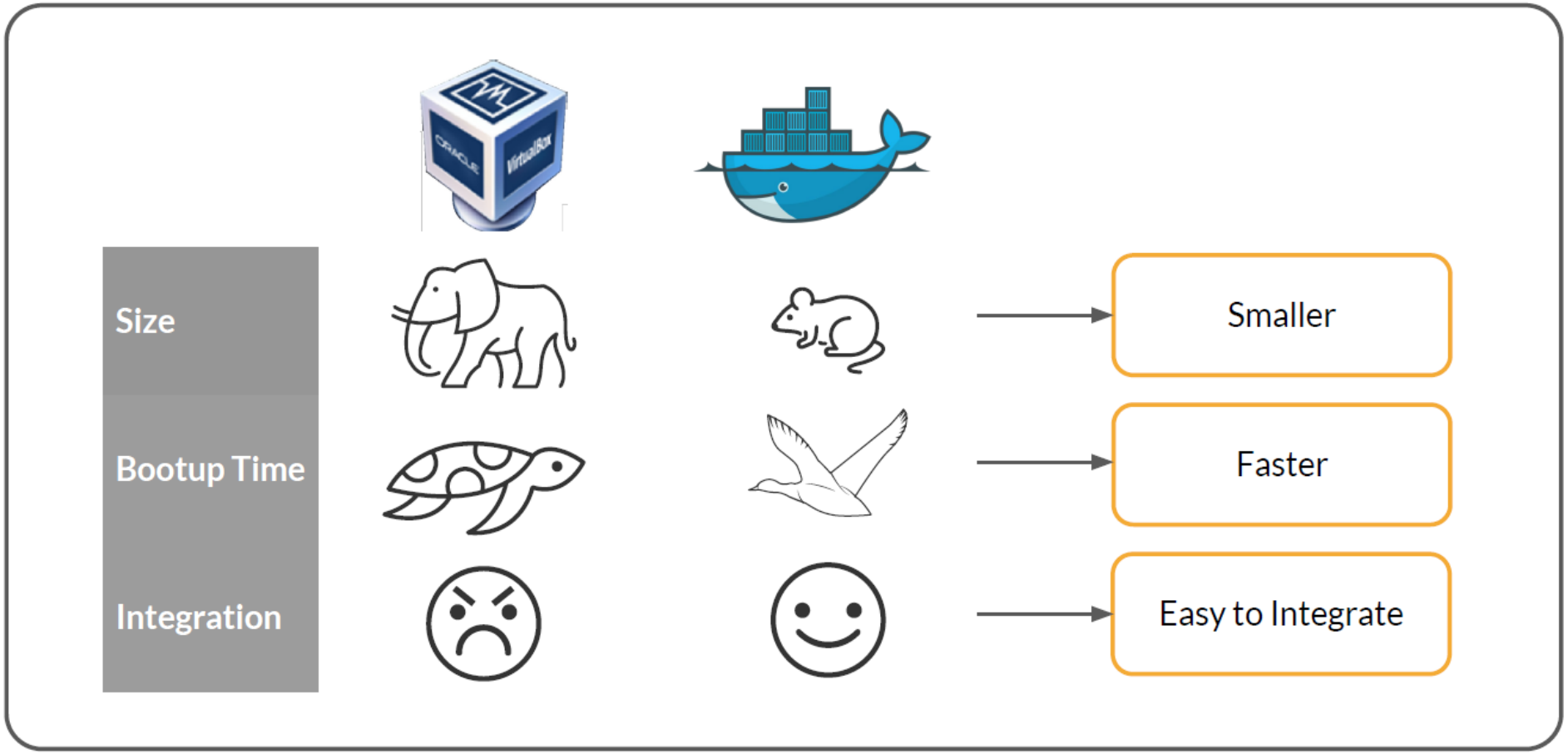
Docker containers are lightweight



Differences between Containers and Virtual Machines



Differences between Containers and Virtual Machines



Use Case

A developer will setup a JBoss software on his system

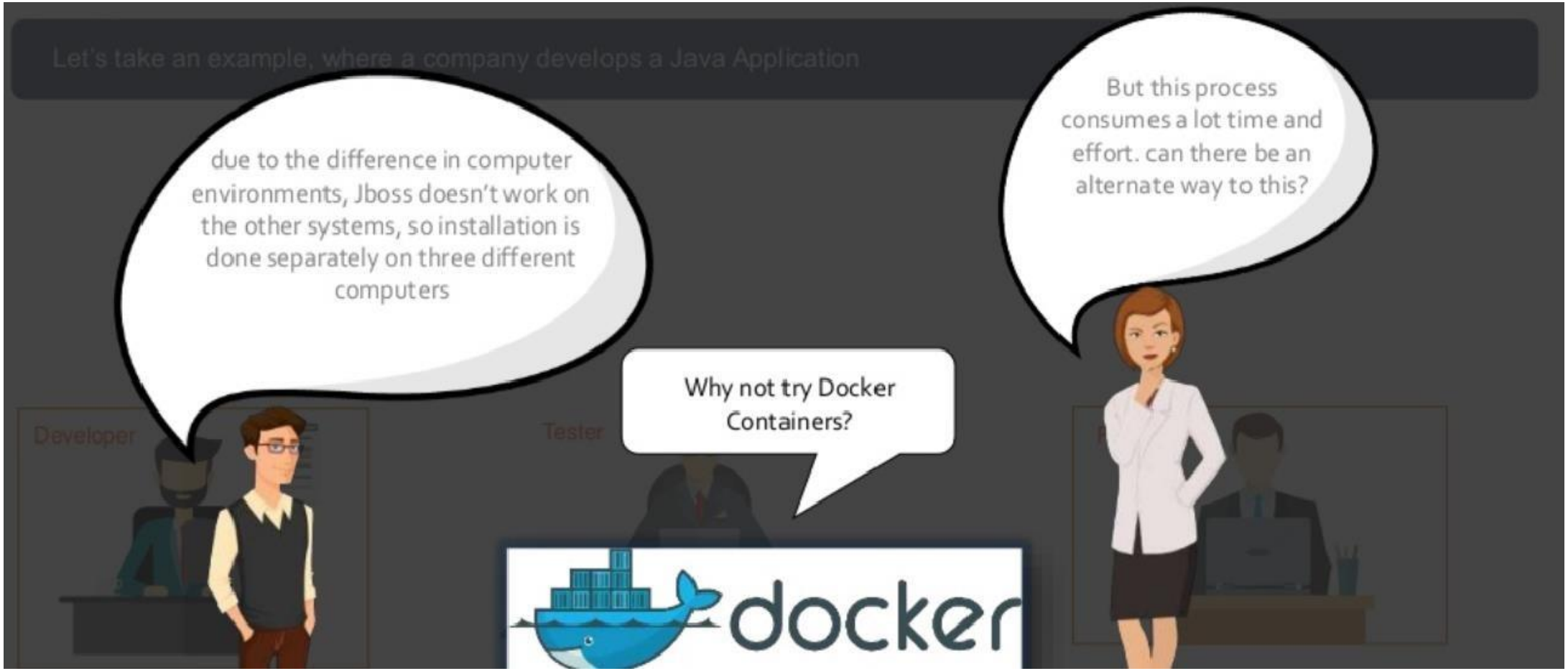
After the application is developed, it is examined by the testing team

Here, the tester repeats the installation process of JBoss

Once the application is tested, it w be deployed by the production tea

To host the Java application, the system admin also has to install JBoss on his system

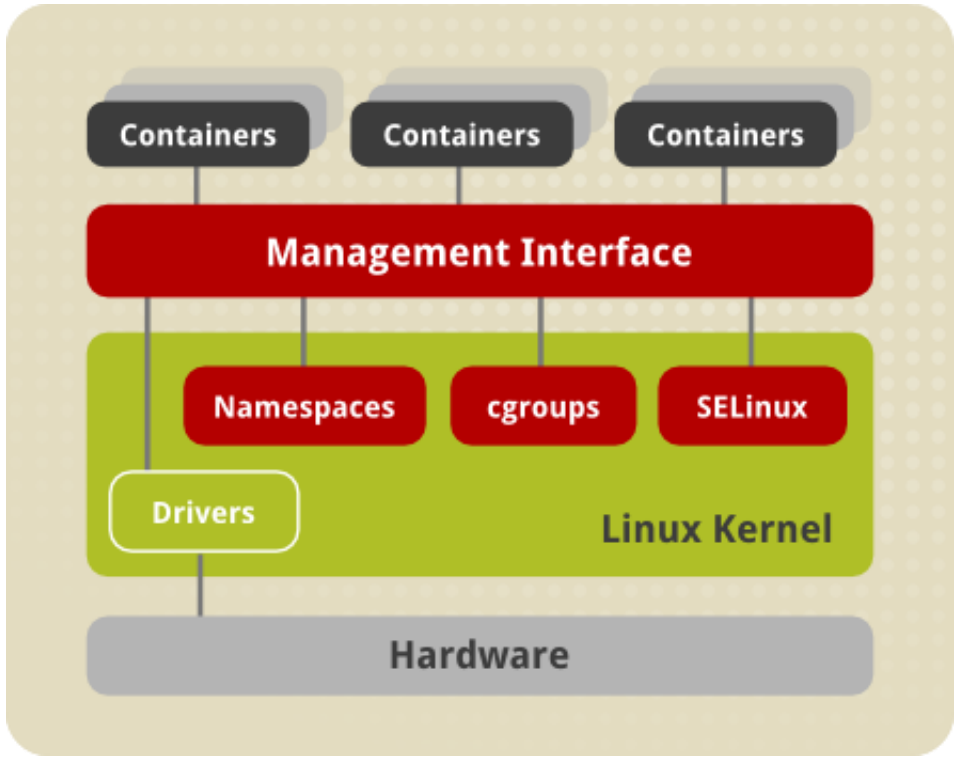
Use Case

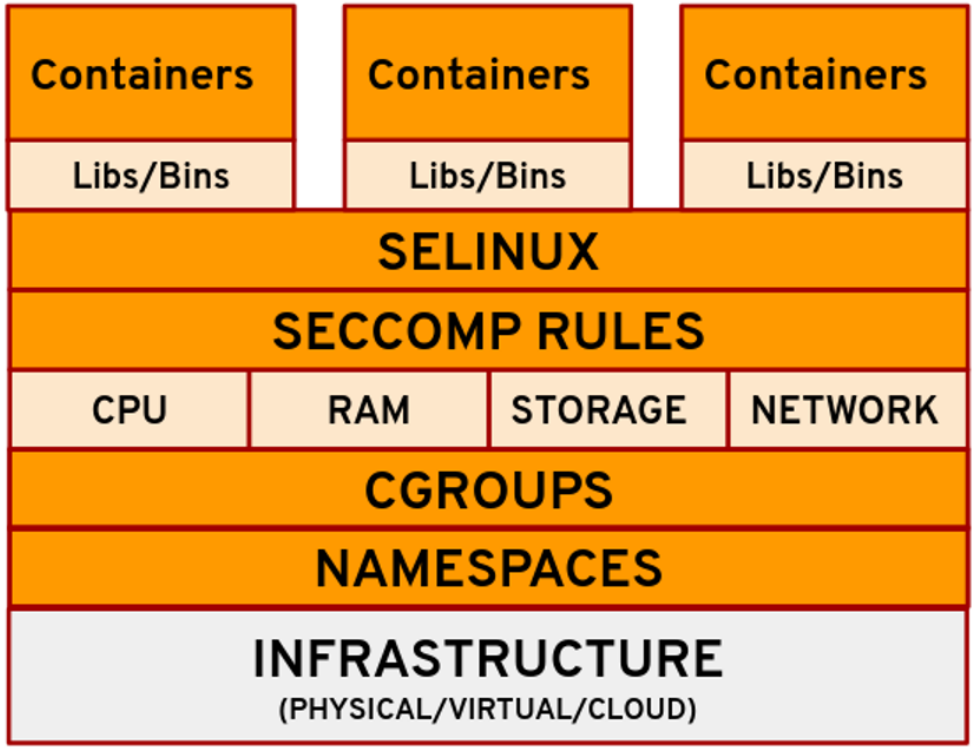


Thank you

Overview of Containers and Dockers

Container Building Blocks





A container is a standard unit of software that packages up code and all its with other containers, each running as isolated processes in user space.

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers
https://opensource.com/article/21/8/container-linux-technology

Namespaces

- Technique where groups of processes are separated in such a way that they cannot see resources in other groups.

- PID namespace provides a separate enumeration of process identifiers within each namespace.

- 6 namespaces: mnt, pid, net, ipc, uts, user

Ref:https://man7.org/linux/man-pages/man7/namespaces.7.html

Cgroups

- Known as control groups
- It can be used to restrict, prioritize and monitor the resource usage on the process level
- Can be used to set limits on resource e.g. cpu, memory, disk I/O throughput and network bandwidth

Container Namespaces

- Namespaces provide a layer of isolation for the containers by giving the container a view of what appears to be its own Linux filesystem. This limits what a process can see and therefore restricts the resources available to it.

- Docker uses a technology called namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container. These namespaces provide a layer of isolation.

- All the containers use the same user namespace and cgroup resource namespaces others running on the host

Finding Container Namespace

```
[root@ip-172-31-21-77 ~]# docker inspect c0f69e5cd207 | grep -i pid
        "Pid": 4452,
        "PidMode": "",
        "PidsLimit": null,

[root@ip-172-31-21-77 ~]# find / -name 4452
/proc/4452
/proc/4452/task/4452
[root@ip-172-31-21-77 ~]# cd /proc/4452

[root@ip-172-31-21-77 4452]# ls
arch_status    environ   mem          patch_state   stat
attr           exe       mountinfo    personality   statm
autogroup      fd        mounts       projid_map    status
auxv           fdinfo    mountstats   root          syscall
cgroup         gid_map   net          sched         task
clear_refs     io        ns           schedstat     timens_offsets
cmdline        latency   numa_maps    sessionid     timers
comm           limits    oom_adj      setgroups     timerslack_ns
```

Access Container Namespaces

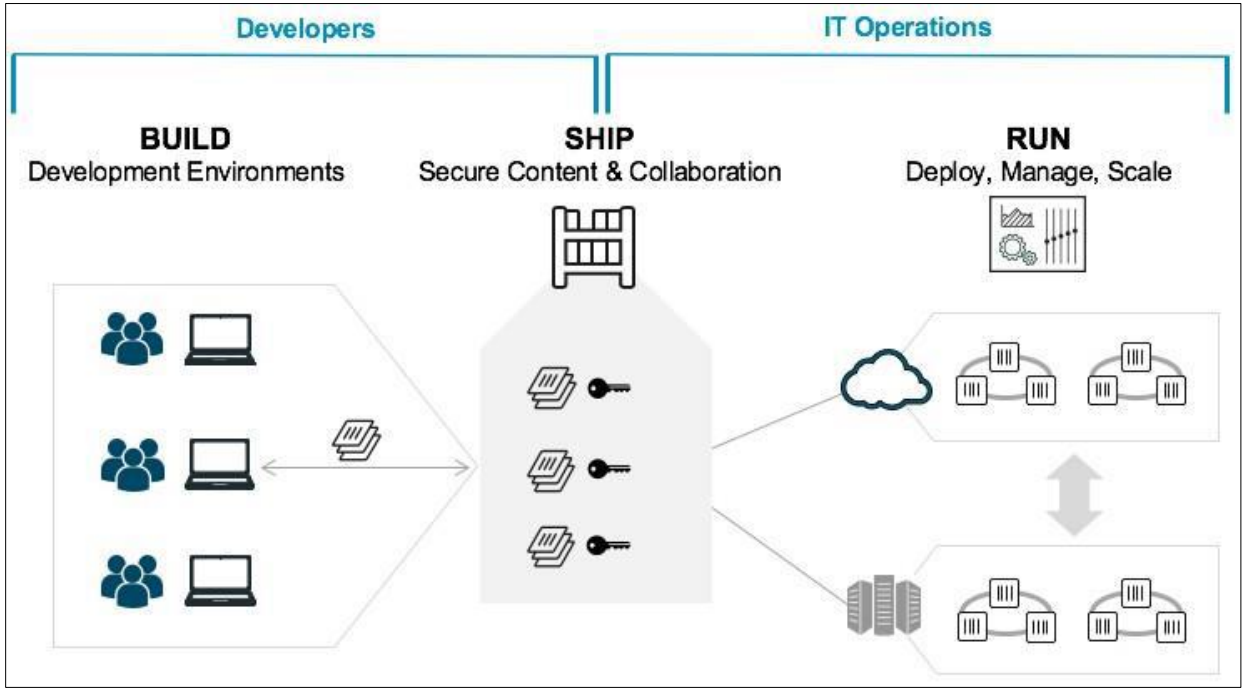
```
[root@ip-172-31-21-77 ns]# ls
cgroup  mnt  pid           time           user
ipc     net  pid_for_children  time_for_children  uts

[root@ip-172-31-21-77 ns]# ls -l
total 0
lrwxrwxrwx 1 root root 0 Jan 12 16:58 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 ipc -> ipc:[4026532372]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 mnt -> mnt:[4026532370]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 net -> net:[4026532375]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 pid -> pid:[4026532373]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 pid_for_children -> pid:[4026532373]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 time -> time:[4026531834]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 time_for_children -> time:[4026531834]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Jan 12 16:58 uts -> uts:[4026532371]
```

What is Docker?

- Docker is an open platform for developing, shipping, and running container applications.
- Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
- By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.
- Docker provides the ability to package and run an application in a loosely isolated environment called a container.

Benefits of Dockers

- Fast, consistent delivery of your applications
- Responsive deployment and scaling
- Running more workloads on the same hardware



Docker Editions

Two editions of Docker

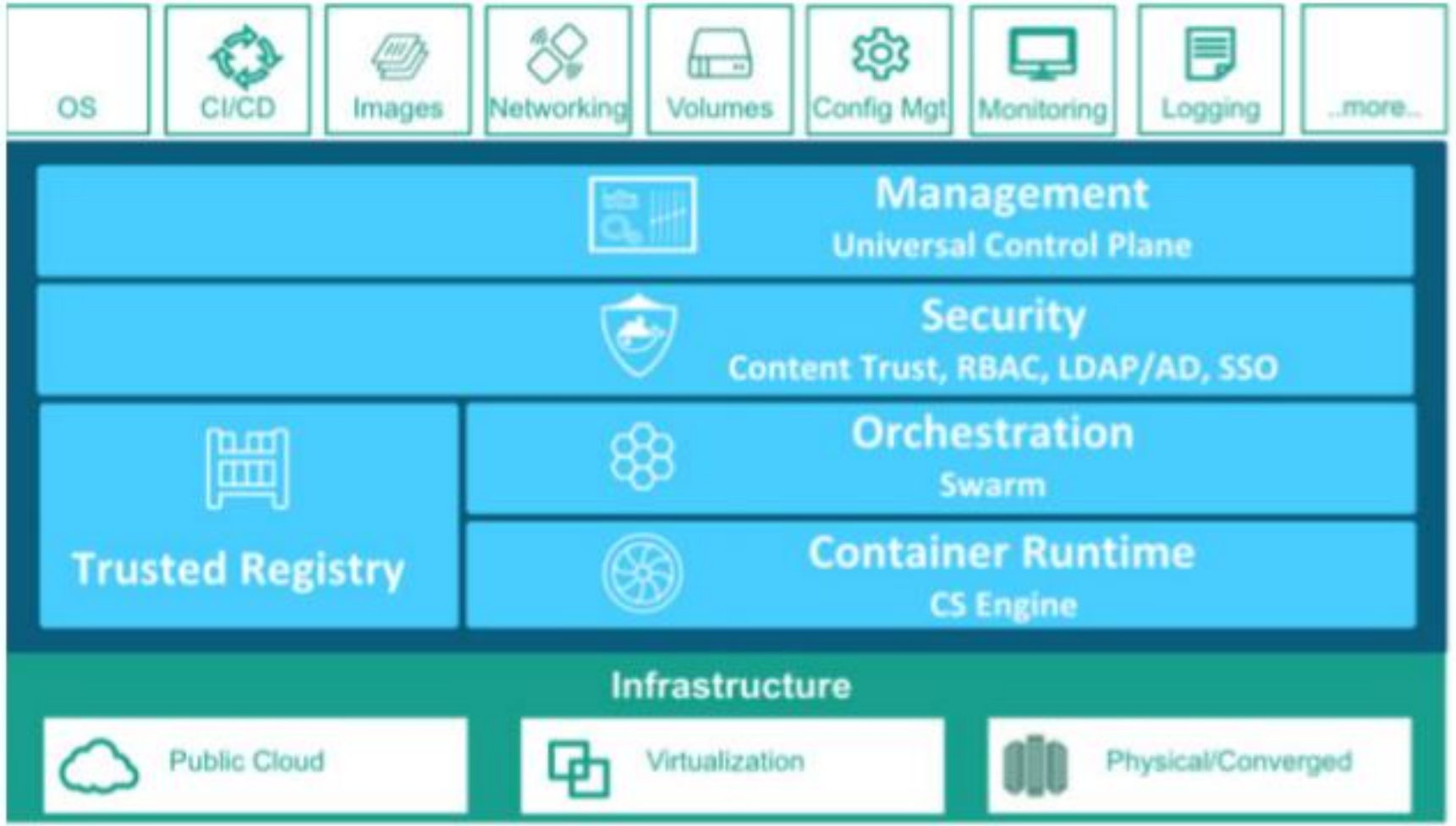
- Docker Community Edition (CE) (Free of cost) is ideal for individual developers and small teams looking to get started with Docker and experimenting with container based apps. Can only use Docker CLI

- Docker Enterprise Edition (is designed for enterprise development and IT teams who build, ship, and run business critical applications in production at scale.Supports both CLI and GUI.

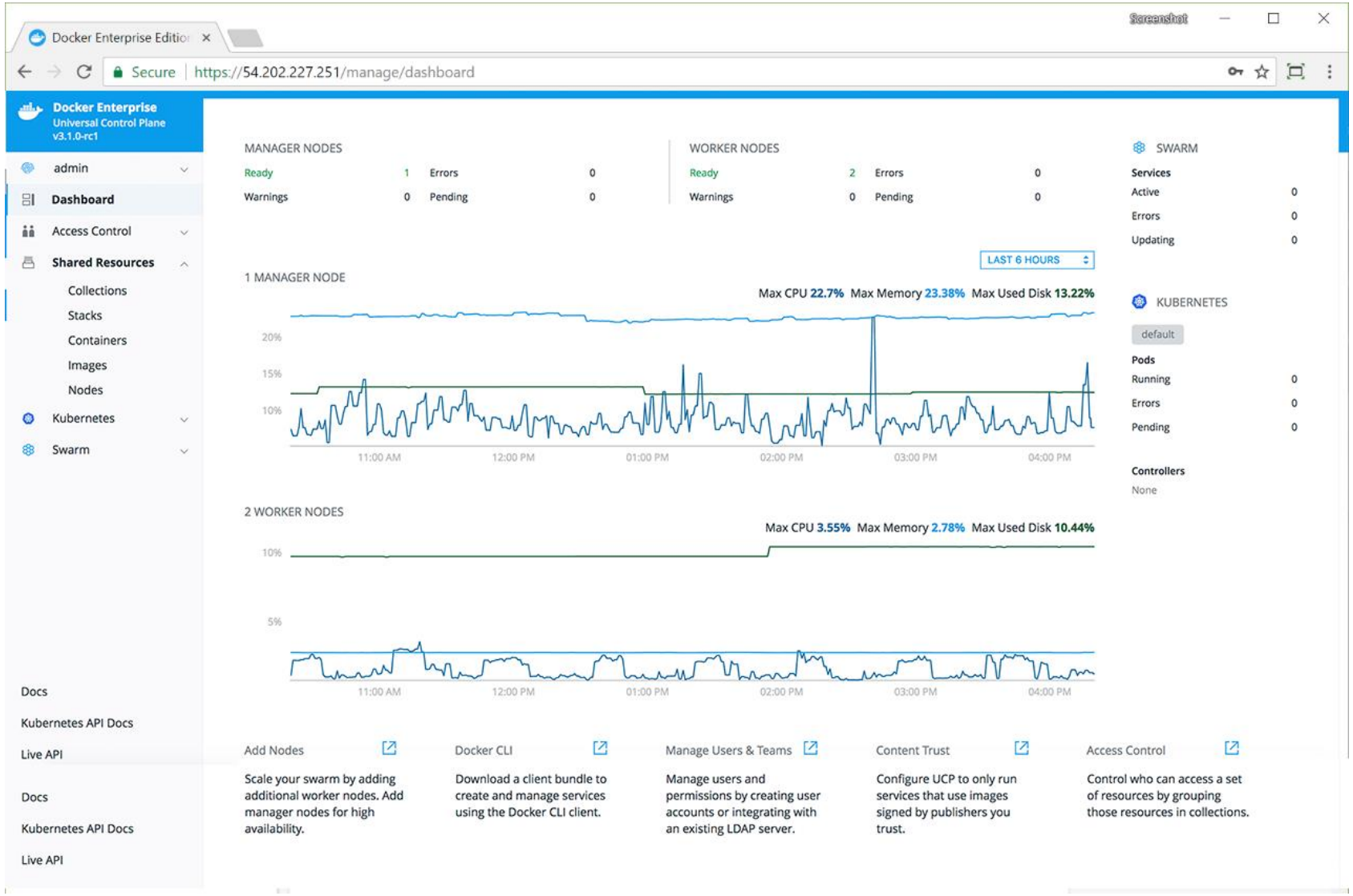
Comparison of Docker Editions

Capabilities	Community Edition	Enterprise Edition Basic	Enterprise Edition Standard	Enterprise Edition Advanced
Container engine and built in orchestration, networking, security	✓	✓	✓	✓
Certified infrastructure, plugins and ISV containers		✓	✓	✓
Image management			✓	✓
Container app management			✓	✓
Image security scanning				✓

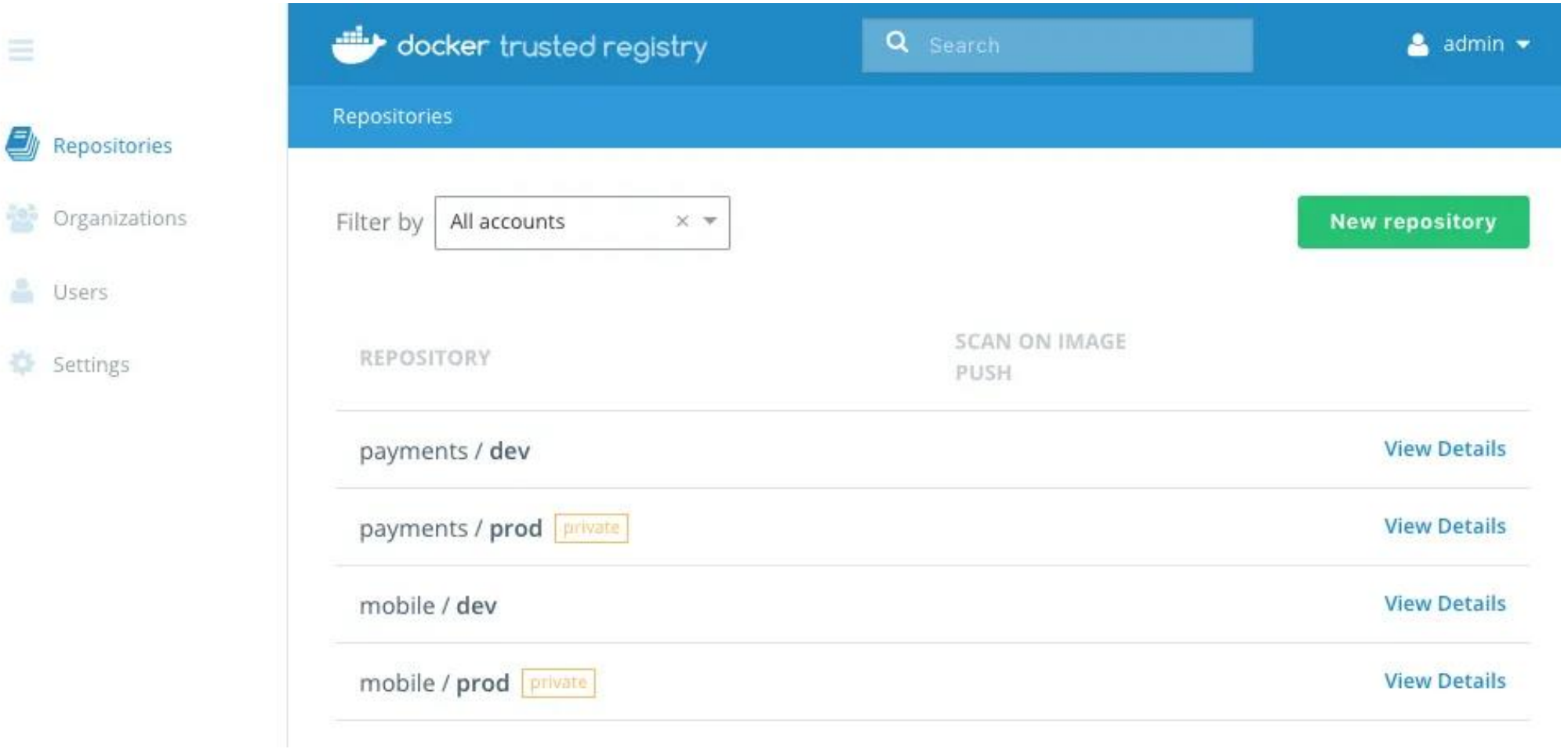
Docker Enterprise Edition Components



UCP Dashboard



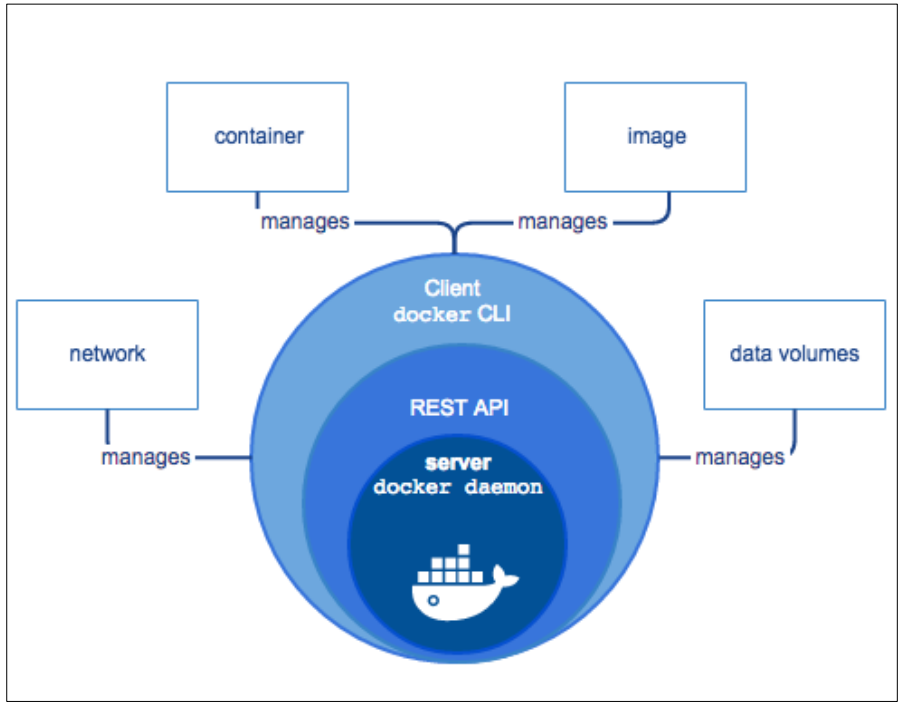
Docker Trusted Registry



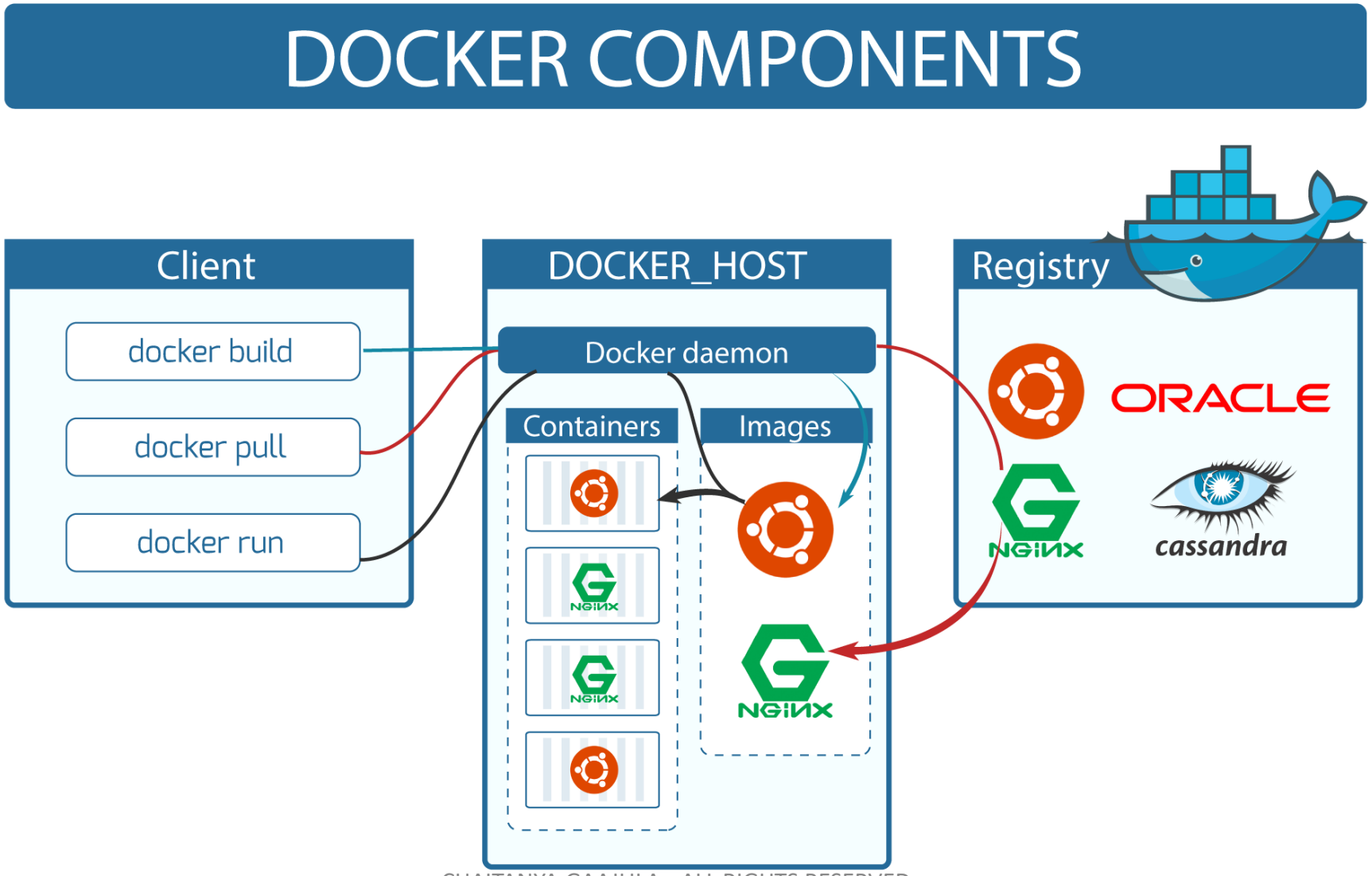
Docker Engine

Docker Engine is a client-server application with these major components:

- A server which is a type of long-running program called a daemon process (the dockerd command).
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the docker command).
- The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands.
- The daemon creates and manages Docker objects, such as images, containers, networks, and volumes.



Docker Architecture & Components



Docker Components

The Docker daemon

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
- A daemon can also communicate with other daemons to manage Docker services.

The Docker client

- The Docker client (docker) is the primary way that many Docker users interact with Docker.
- When you use commands such as docker run, the client sends these commands to dockerd, which carries them out.
- The docker command uses the Docker API.
- The Docker client can communicate with more than one daemon.

Docker Registries

- A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default.

- Default Docker Registry is DockerHub in which you can both public and private repository.

- When docker pull is executed , the required images are pulled from configured registry. When the docker push command is executed , image is pushed to the configured registry.

Docker Images

- An image is a read-only template with instructions for creating a Docker container.

- An image is based on another base OS image, with some additional customization.

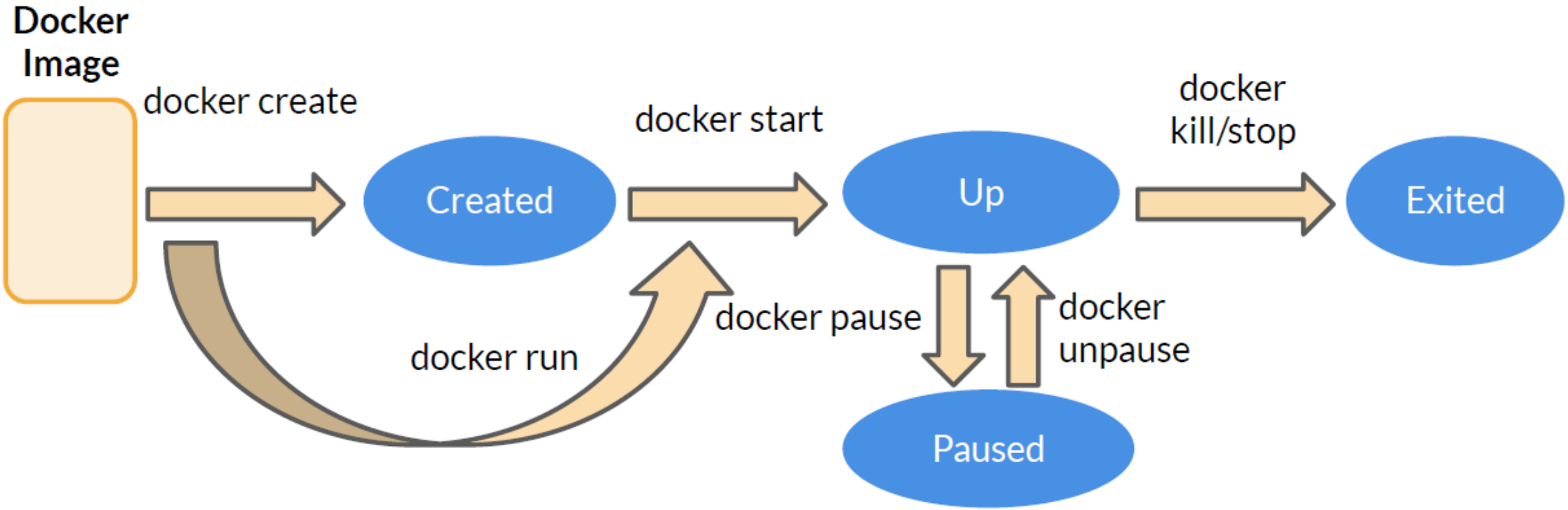
For example –One you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

- Using Dockerfile own images can be built . Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is why images so lightweight, small, and fast, when compared to other virtualization technologies.

Docker Containers

- A container is a runnable instance of an image. One can create, start, stop, move, or delete a container using the Docker API or CLI.
- Containers are lightweight because they don’t need the extra load of a hypervisor, but run directly within the host machine’s kernel
- By default, a container is relatively well isolated from other containers and its host machine.
- A container is defined by its image as well as any configuration options you provide to it when you create or start it.
- When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Docker Container Lifecycle



THANK YOU

Docker Registry

What is an image

- Docker image is a snapshot or template from which new containers can be started. It's a combination of file system and libraries .

- A new image can be created by executing a set of commands contained in a Dockerfile

  For example, this Docker file would take a base Ubuntu
  6. image and install git, resulting in a new image:

  FROM ubuntu:16.04

  RUN apt-get install -y git

- **I**mage is composed of a set of read-only layers. Image layers function as follows:
  - Each image layer is the outcome of one command in the image's Dockerfile
  - Each additional image layer only includes the set of differences from the previous layer. (docker history command can be used to check all layer images)

Types of Images

- ubuntu
- centos
- alpine

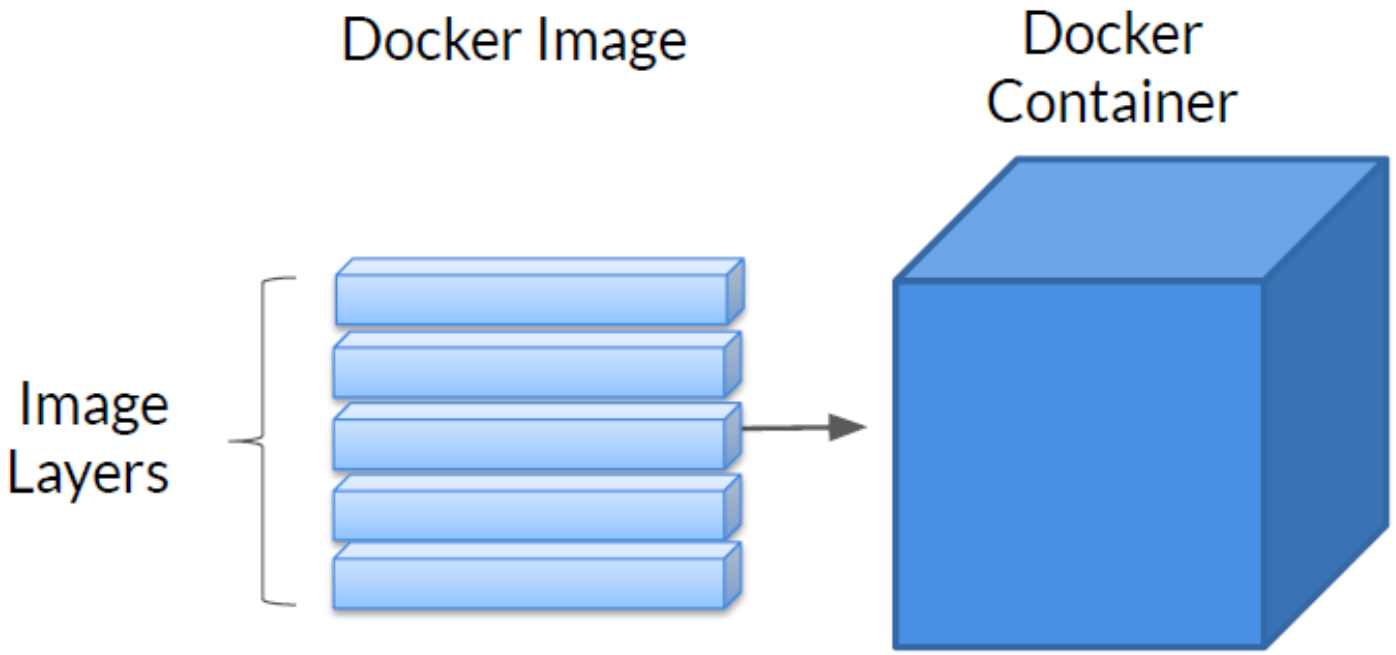
Command: **docker run hello-world**

- The Docker command is specific and tells the Docker program on the Operating System that something needs to be done.
- The **run** command is used to mention that we want to create an instance of an image, which is then called a **container**.
- Finally, "hello-world" represents the image from which the container is made.

Command : **docker images**

This command is used to display all the images currently installed on the system

Docker Image Layers



Docker Layers

- Docker image is made of a series of Docker layers
- Each layer corresponds to an instruction in Dockerfile and is read-only
- Each layer contains the differences between the preceding layer and the current layer
- Docker images are stored at /var/lib/docker/overlay location on Linux host machine

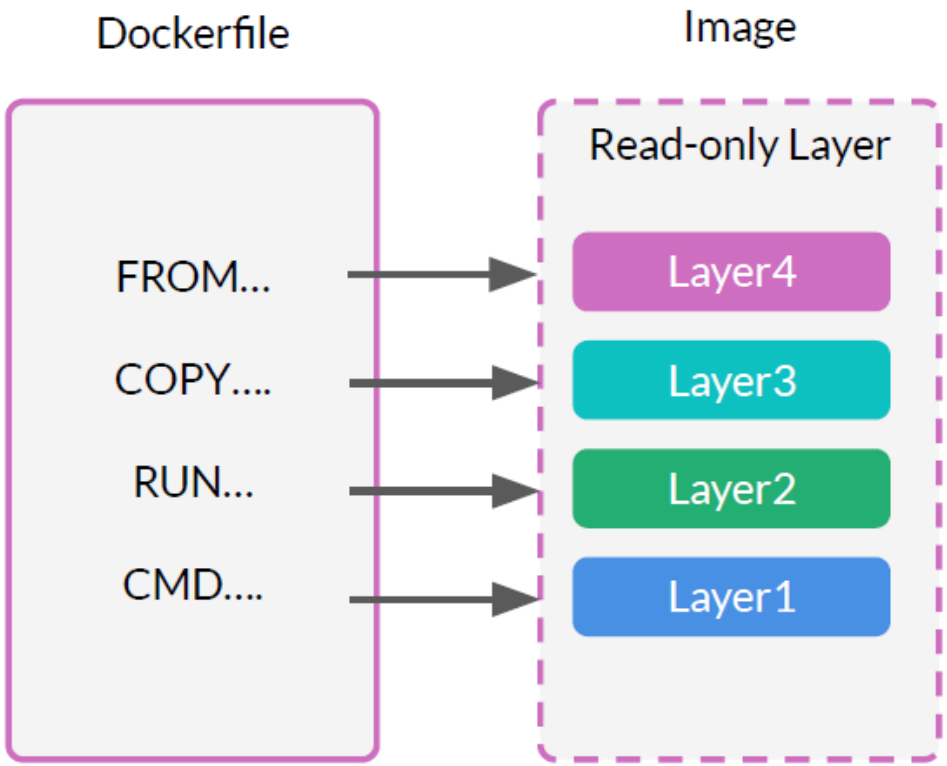
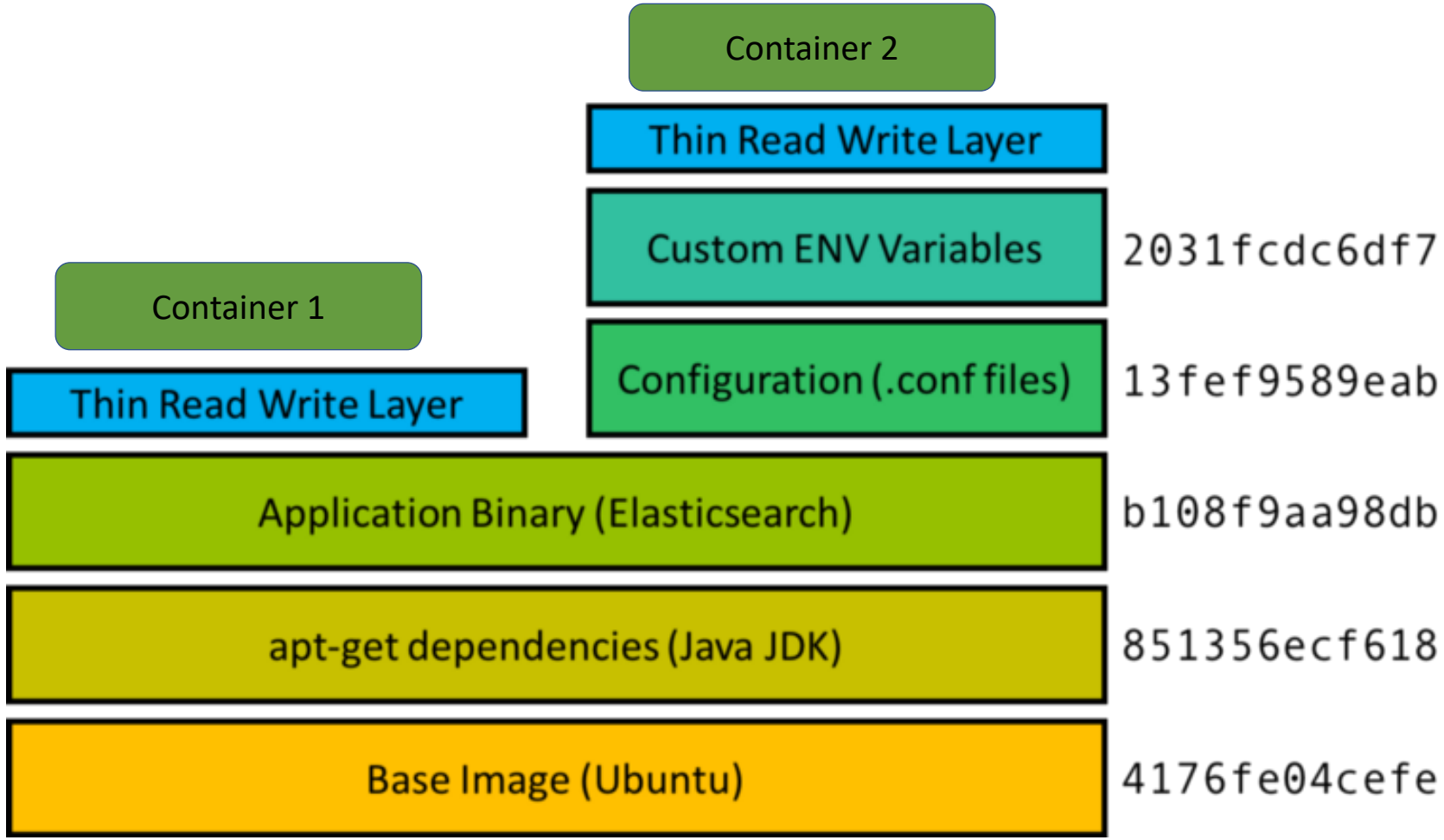
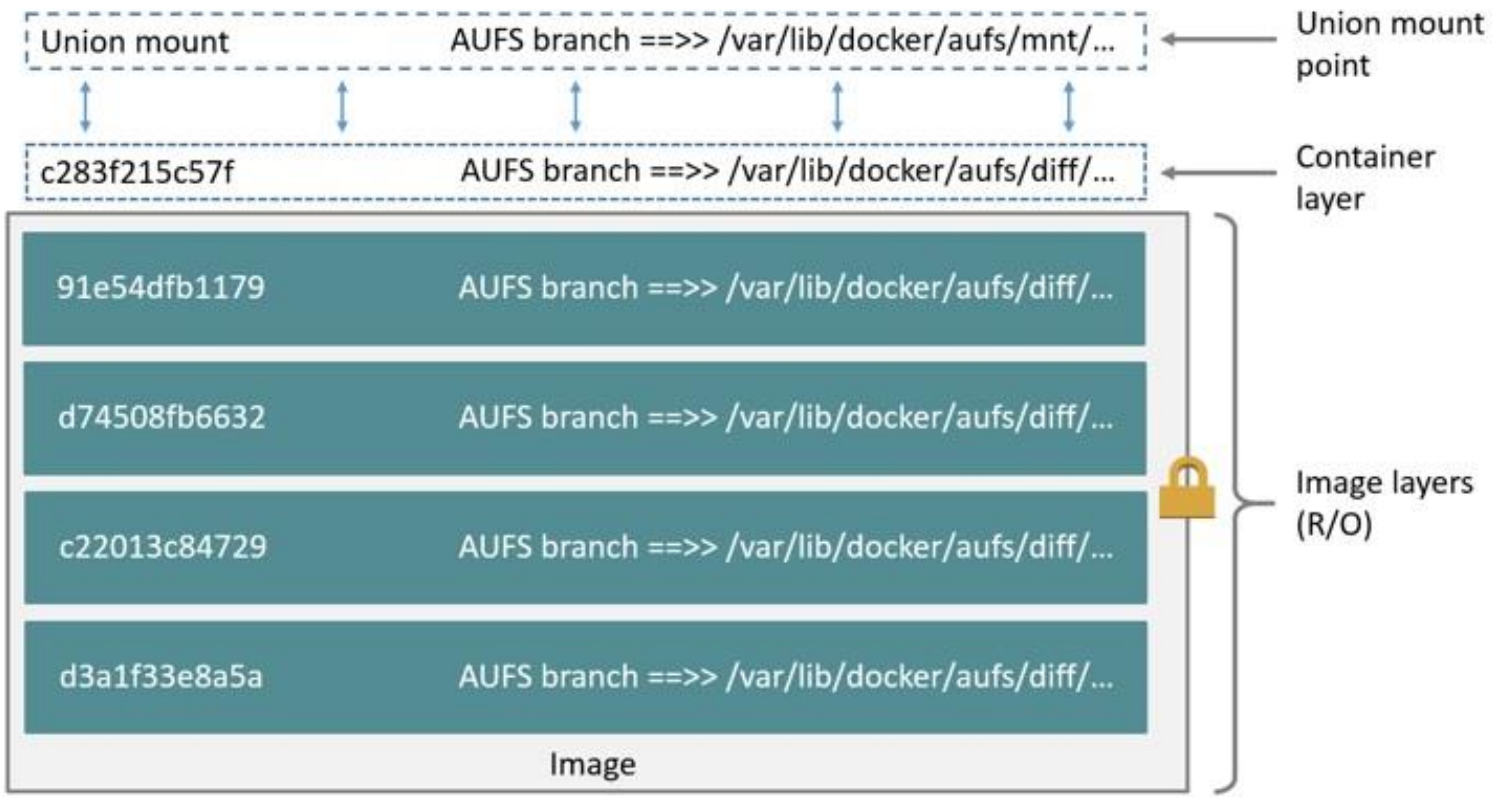


Image and Containers



Overlay File System



Docker Image Commands

Build / Pull	Create / Download a docker image
Run	Create a container from the image
Commit	Save container as image
Push	Upload image to central repository
Stop	Stop the container
Remove	Delete a container

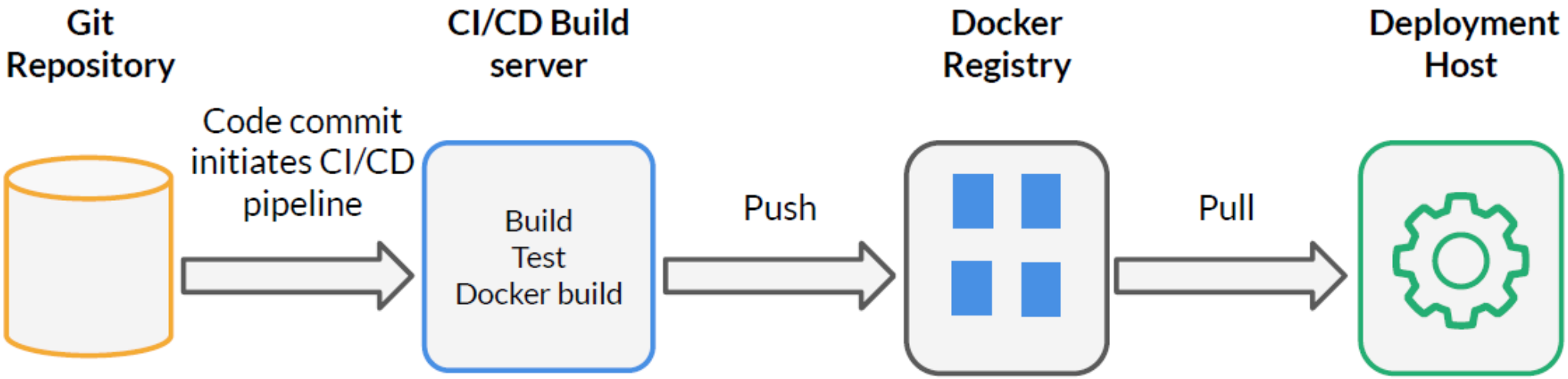
Docker Registries

Docker Registries

- A Docker registry is a place to store and distribute Docker images. It serves as a target for your docker push and docker pull commands
- Repository- Collection of different versions of a single Docker image
- Tags – Named version of an image . Provide meaningful name for easy reference
- Two types of Registries – Public and Private

Public : Anyone can pull and push images . Eg: Docker Hub , Docker Store
Private : Strict Access control is provided for accessing the images Eg: Google Container Registry, AWS ECR

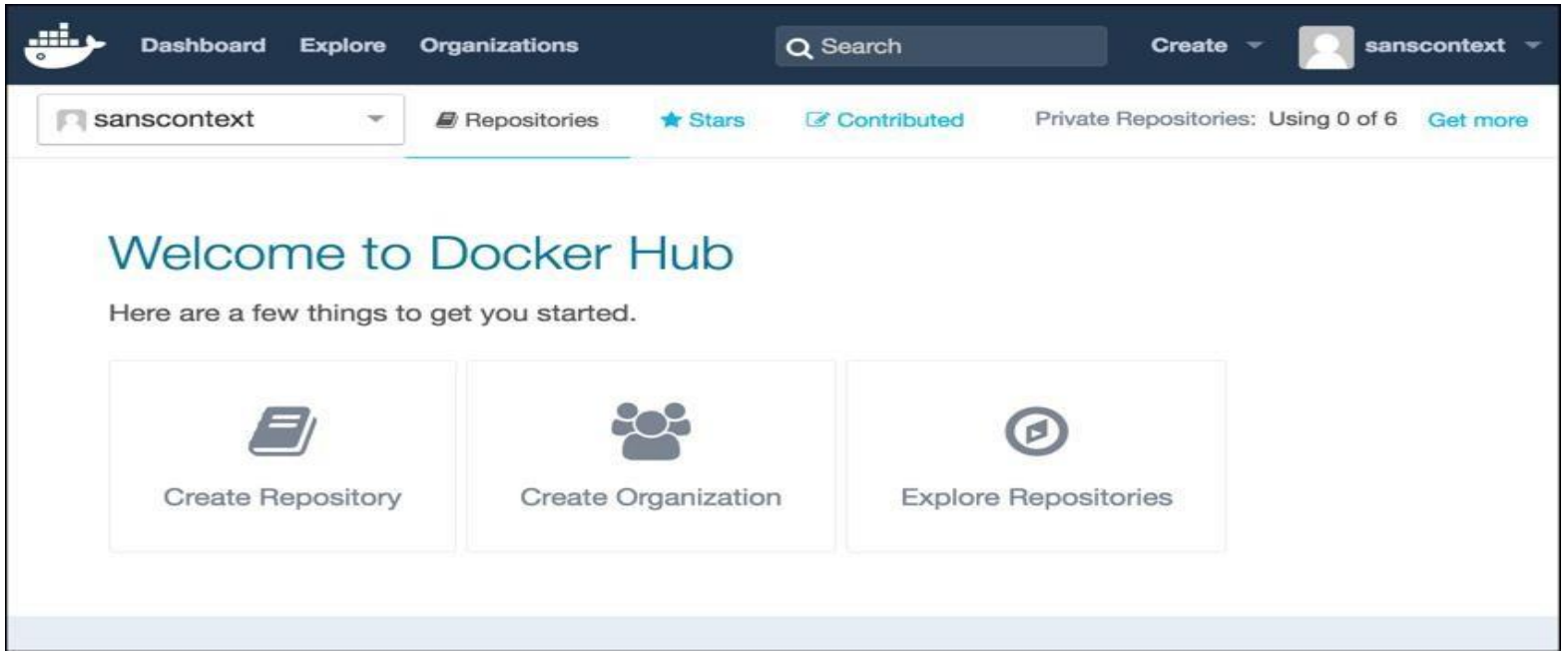
Docker Registry



Can be easily integrated with CI/CD system

Docker Hub

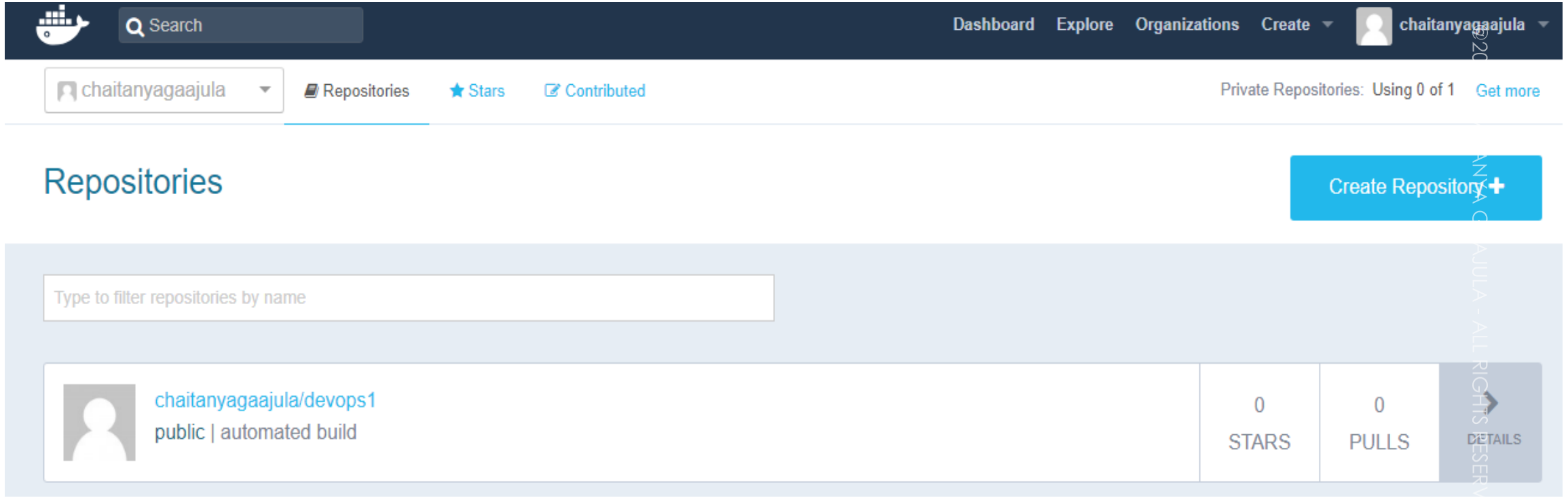
- Docker Hub is a cloud-based repository in which Docker users and partners create, test, store and distribute container images.
- Users can access public, open source image repositories, as well as use a space to create their own private repositories, automated build functions, and work groups.



Docker Hub-Features

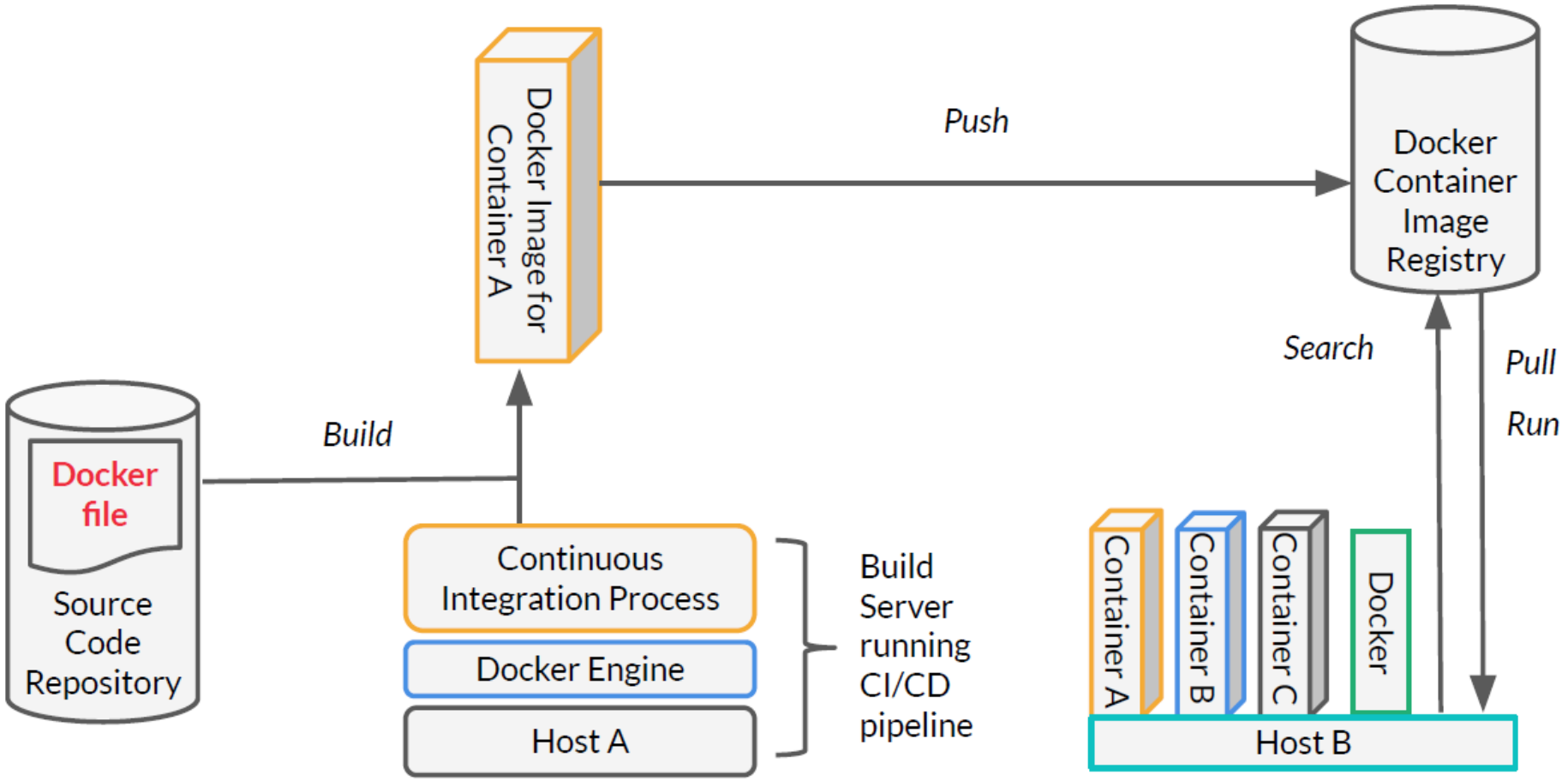
- Image Repositories: Find and pull images from community and official libraries, and manage, push to, and pull from private image libraries to which you have access.
- Automated Builds: Automatically create new images when you make changes to a source code repository.
- Webhooks: A feature of Automated Builds, Webhooks let you trigger actions after a successful push to a repository.
- Organizations: Create work groups to manage access to image repositories.
- GitHub and Bitbucket Integration: Add the Hub and your Docker Images to your current workflows

Docker Hub Repository



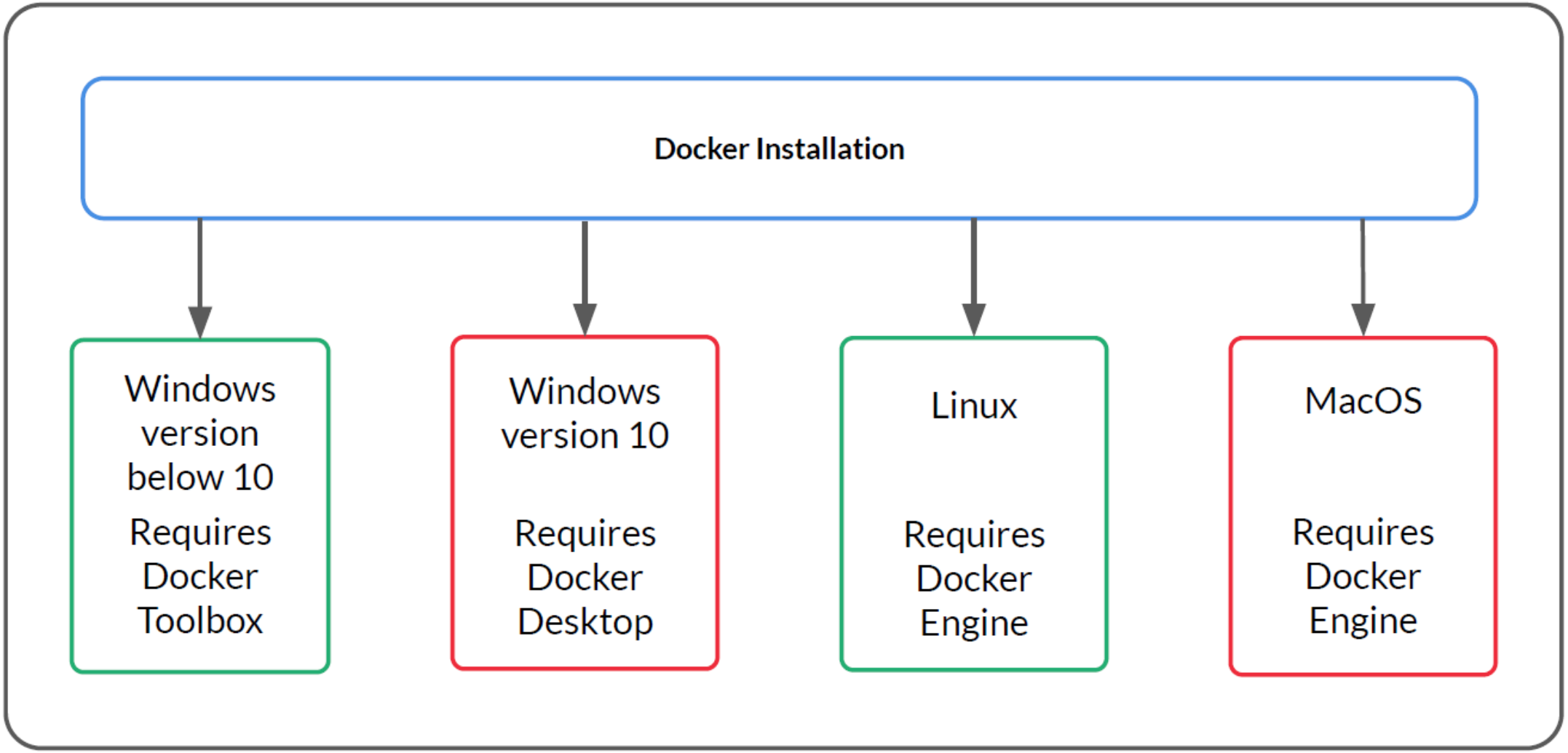
- Docker Hub repositories let you share images with co-workers, customers, or the Docker community at large.
- You can build your images and push them to a Docker Hub repository that you add to your Docker Hub user or organization account.

Docker Push / Pull Images



THANK YOU

Docker Installation


Docker Installation
Windows version below 10
Requires Docker Toolbox
Windows version 10
Requires Docker Desktop
Linux
Requires Docker Engine
MacOS
Requires Docker Engine

Docker for Linux

- Docker was originally a Linux application
- It uses the kernel container functionality
- It requires a 64 bit installation using a kernel version 3.10 or later
- Docker runs on many popular Linux distributions
- It is available as YUM ,RPM, APT, or binary versions

Running Docker as a user

- Docker Daemon binds to a Unix socket instead of a TCP port, which is owned by root user.
- Other users can only access it by using sudo.
- To avoid using sudo every time, create a Unix group called Docker, and add users to it.
- The Docker daemon now, when starts, it gives the ownership for Unix socket read/writable by the Docker group.
- Make sure you log out and log in once.

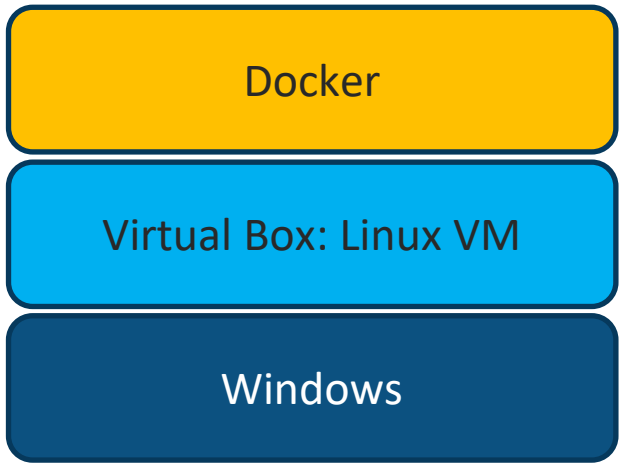
Docker for OS X

- Docker runs natively on OS X
- Is built on the xhyve hypervisor
- Requires a 2010 or newer Mac with Intel MMU and EPT support
- Requires OS X 10.10.3 Yosemite or newer
- Requires at least 4GB of RAM
- Docker instances can't be accessed remotely due to limited network support

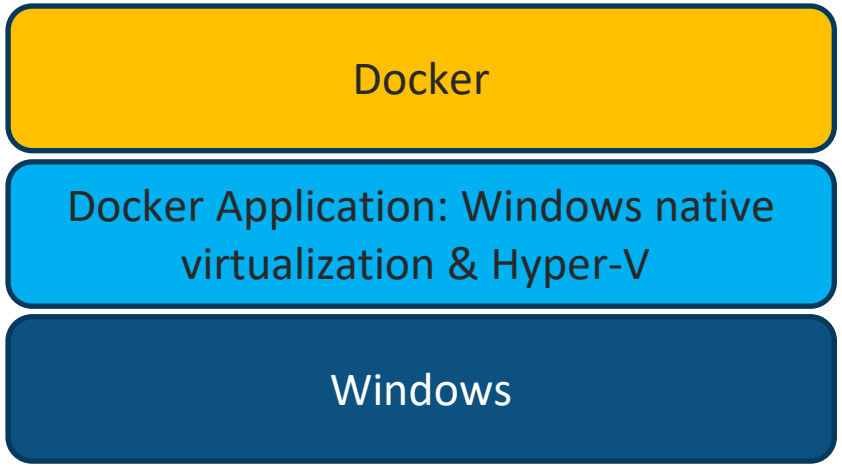
Docker for Windows

- Docker runs natively on Windows
- Requires later versions of Windows 10 Pro or Enterprise
- Docker requires the Hyper-V package
- This is Microsoft's hypervisor for Windows
- It virtualizes the Docker environment and Linux kernel specific features
- Docker can't run alongside VirtualBox VMs

Docker for Windows



All Windows prior to Win 10 via
Docker Toolbox

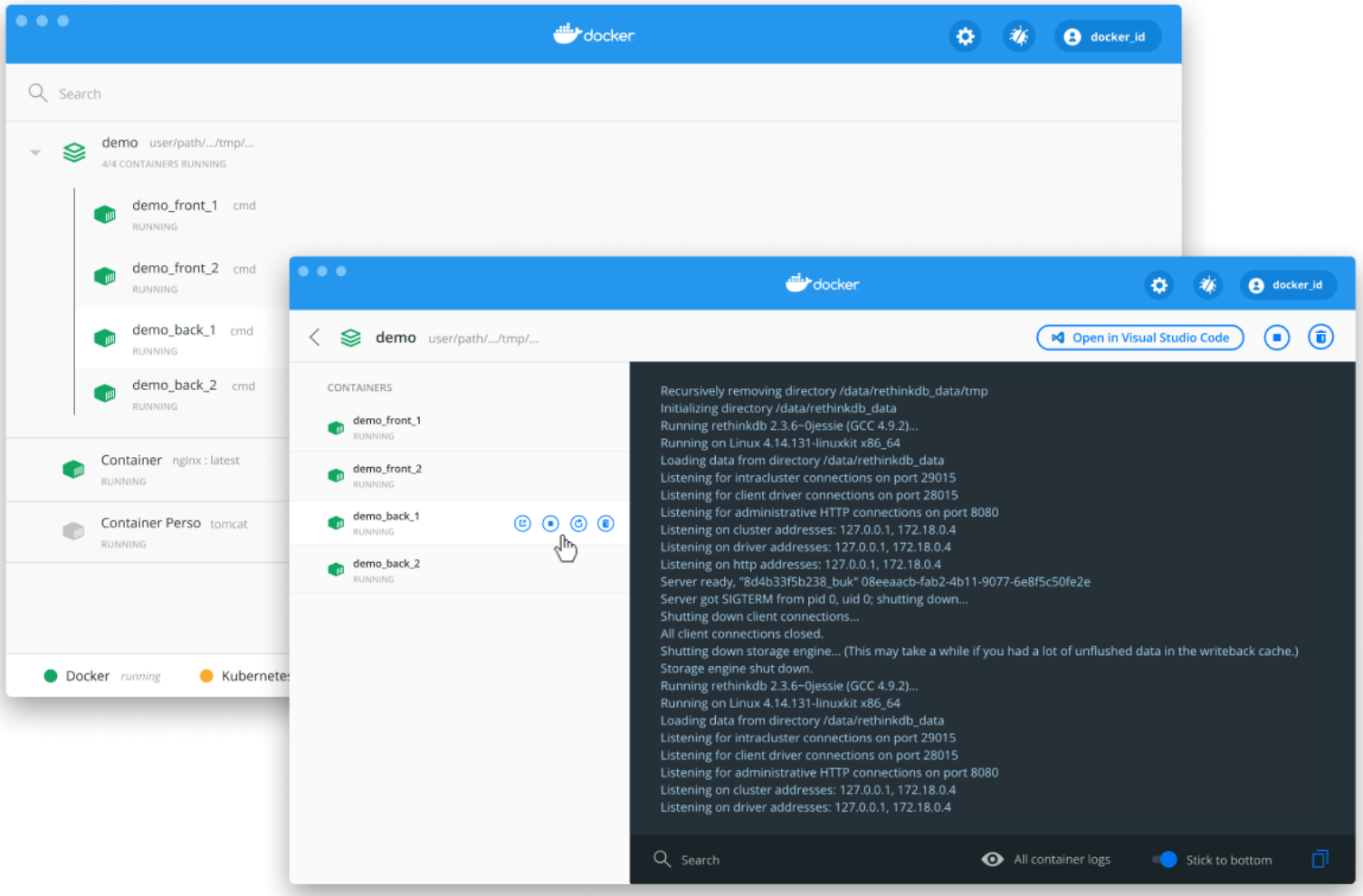


For Windows 10 Professional or
Enterprise and above

Overview of Docker Desktop

- An application for MacOS and Windows machines for the building and sharing of containerized applications and microservices.
- Delivers the speed, choice and security for designing and delivering containerized applications on your desktop.
- Docker Desktop allows you to leverage certified images and templates and your choice of languages and tools.
- Development workflows leverage Docker Hub to extend your development environment to a secure repository for rapid auto-building, continuous integration and secure collaboration.

Working with Docker Desktop



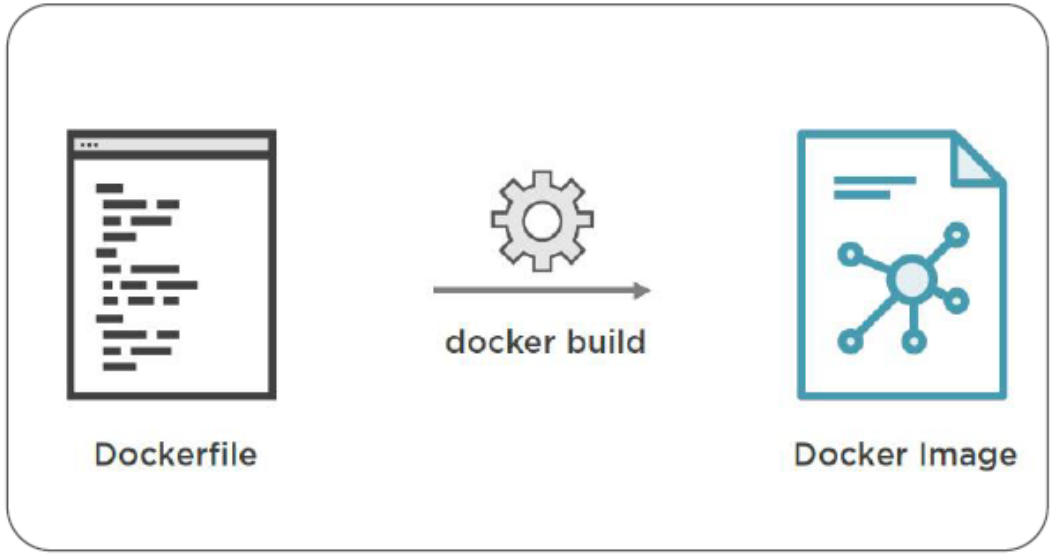
THANK YOU

Dockerfile & Docker Compose

Automating Build using 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.
- Each command in the build file creates a layer of the image

What is a Dockerfile



Dockerfile

```
FROM openjdk:8-jdk-alpine
MAINTAINER upgrad
ADD build/libs/application.jar
/opt/app/application.jar
WORKDIR /opt/app
ENV PATH="${PATH}:${JAVA_HOME}/bin"
ENTRYPOINT [ "java", "-jar",
"/opt/app/application.jar"]
```

Working with Dockerfile

Dockerfile contains build directives

- FROM defines the starting image
- MAINTAINER defines the email address of the builder

FROM centos:latest

MAINTAINER XXX@XXX.com

```
1  FROM ubuntu:14.04
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Env and Command

Env:

- The Dockerfile ENV command sets environment variables in the image.

ENV JAVA_HOME=/usr/java/latest

ENV PATH=/usr/local/sbin:/usr/local/bin

```
1  FROM ubuntu:14.04
2
3  RUN apt-get update
4  RUN apt-get install vim nginx php5-fpm
5  RUN find /etc/php5/cli/conf.d/ -name "*.ini"
6
7  ENV DocumentRoot="/usr/share/nginx/html"
8
9
10
11
12
13
14
15
```

Command :

The Dockerfile CMD command sets the default application.

CMD /bin/bash

Add and Run

Add:

- The Dockerfile ADD command copies files and remote file URLs into the container
- The source files or directories must be in the build context or remote URLs
- The source files can contain UNIX shell wildcards ? * []
- Destination directories must end in a / and will get created if they don't exist Local source files in tar or compressed tar format get unpacked

Eg : ADD apache-maven*.tar.gz /opt/

```
1  FROM ubuntu:14.04
2
3  RUN apt-get update
4  RUN apt-get install vim nginx php5-fpm
5  RUN find /etc/php5/cli/conf.d/ -name "*.ini"
6
7  ENV DocumentRoot="/usr/share/nginx/html"
8
9  ADD ./nginx-site.conf /etc/nginx/default
10 ADD ./source.tar.gz /tmp/
11 ADD http://example.com/file /home/
12
13
14
15
```

Run :

- The Dockerfile RUN command executes a Linux command
- Multiple commands can be separated with a ; - needed for cd
- Commands shouldn't block for input – commands have a –y switch

which answers yes to all questions RUN yum install –y which , RUN rpm –i /tmp/*rpm ,RUN cd /opt; ln –s apache-maven* maven

Shell & Copy

Shell:

- The Dockerfile SHELL command defines the default shell to use
- It must be specified in JSON form
- The default for Linux and windows are shown
- It takes the form SHELL ["executable", "parameters"]

SHELL ["/bin/sh", "-c"]

SHELL ["cmd", "/S", "/C"]

Copy:

- The Dockerfile COPY command copies files into the container
- The source files or directories must be in the build context
- The source files can contain UNIX shell wildcards ? * []
- Destination directories must end in a / and will get created if they don't exist

COPY jdk*.rpm /tmp/

ENTRYPOINT and CMD

- Both CMD and ENTRYPOINT instructions define what command gets executed when running a container

There are few rules that describe their co-operation:

- Dockerfile should specify at least one of CMD or ENTRYPOINT commands
- ENTRYPOINT should be defined when using the container as an executable
- CMD should be used as a way of defining default arguments for an ENTRYPOINT command
- CMD will be overridden when running the container with alternative arguments

ENTRYPOINT

- **ENTRYPOINT Instruction has two forms:**

  - **ENTRYPOINT [“executable”, “param1”, “param2”]:** It is the executable form with default arguments/parameters. These arguments/parameters can be overridden if used with CMD. This is the most preferred form.

  - **ENTRYPOINT command param1 param2 :** It is the shell form, which will execute the given command in a shell.

CMD

- **CMD Instruction has three forms:**
  - **CMD [“executable”, “param1”, “param2”]:** It is an executable form with arguments. This is preferred form.
  - **CMD [“param1”, “param2”]:** It will provide default parameters/arguments to the ENTRYPOINT instruction.
  - **CMD command param1 param2:** It will provide default parameters/arguments to the ENTRYPOINT instruction

```
1  FROM ubuntu:14.04
2
3  RUN apt-get update
4  RUN apt-get install vim nginx php5-fpm
5  RUN find /etc/php5/cli/conf.d/ -name "*.ini"
6
7  ENV DocumentRoot="/usr/share/nginx/html"
8
9  ADD ./nginx-site.conf /etc/nginx/default
10 ADD ./source.tar.gz /tmp/
11 ADD http://example.com/file /home/
12
13 ENTRYPOINT ["/bin/ping","-c","3"]
14 CMD ["localhost"]
15
```

Expose

- EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime.
- One can specify protocol as well
- Does not make ports of the container accessible to the host

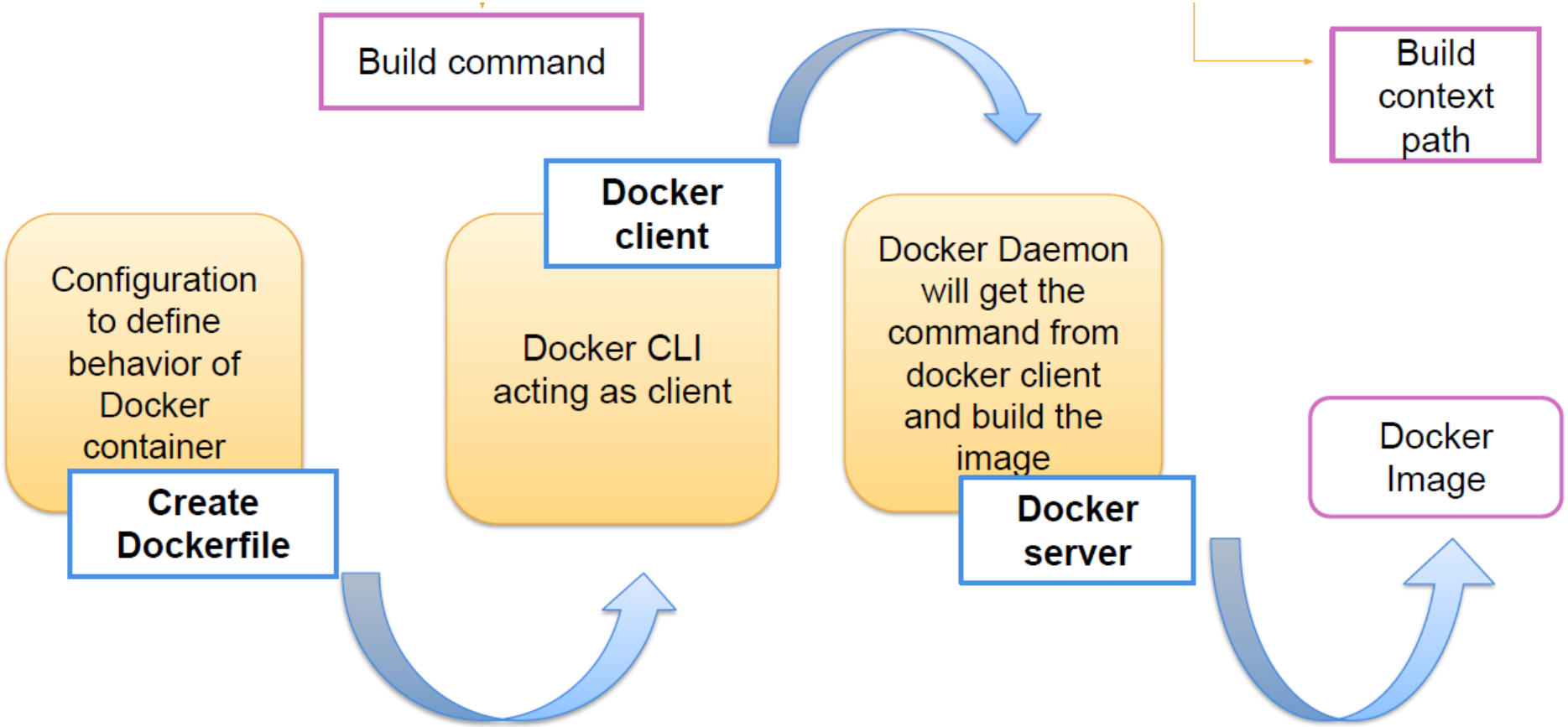
```
1  FROM ubuntu:14.04
2
3  RUN apt-get update
4  RUN apt-get install vim nginx php5-fpm
5  RUN find /etc/php5/cli/conf.d/ -name "*.ini"
6
7  ENV DocumentRoot="/usr/share/nginx/html"
8
9  ADD ./nginx-site.conf /etc/nginx/default
10 ADD ./source.tar.gz /tmp/
11 ADD http://example.com/file /home/
12
13 ENTRYPOINT ["/bin/ping","-c","3"]
14 CMD ["localhost"]
15
16 EXPOSE 80
```

Building Images using Dockerfile

- The build process requires a directory
- All of the directory contents are transferred to the daemon o It must contain a Dockerfile build script
- Images should be tagged image-name:version
- A temporary container is created for each command in the build

eg: docker build –t nginx:0.1 .

Docker Build Process



Sample Output of Docker Build

Sample Output of 'docker build' Command Execution

```
[opc@instance-20201221-1152 sample]$ docker build -t  example/application:latest .
Sending build context to Docker daemon    110MB
Step 1/7 : FROM openjdk:8-jdk-alpine
 ---> a3562aa0b991
Step 2/7 : MAINTAINER example
 ---> Using cache
 ---> a96c5edf5574
Step 3/7 : VOLUME /tmp
 ---> Using cache
 ---> d9c67b9bafdd
Step 4/7 : ADD build/libs/application.jar /opt/app/application.jar
 ---> Using cache
 ---> e5646f3dd0af
Step 5/7 : WORKDIR /opt/app
 ---> Using cache
 ---> 8ca979055bfd
Step 6/7 : ENV PATH="${PATH}:${JAVA_HOME}/bin"
 ---> Using cache
 ---> 98a7f41c3919
Step 7/7 : ENTRYPOINT [ "java", "-jar", "/opt/app/application.jar"]
 ---> Using cache
 ---> 8a9eb21595a1
Successfully built 8a9eb21595a1
Successfully tagged example/application:latest
```

Overview of Docker Compose

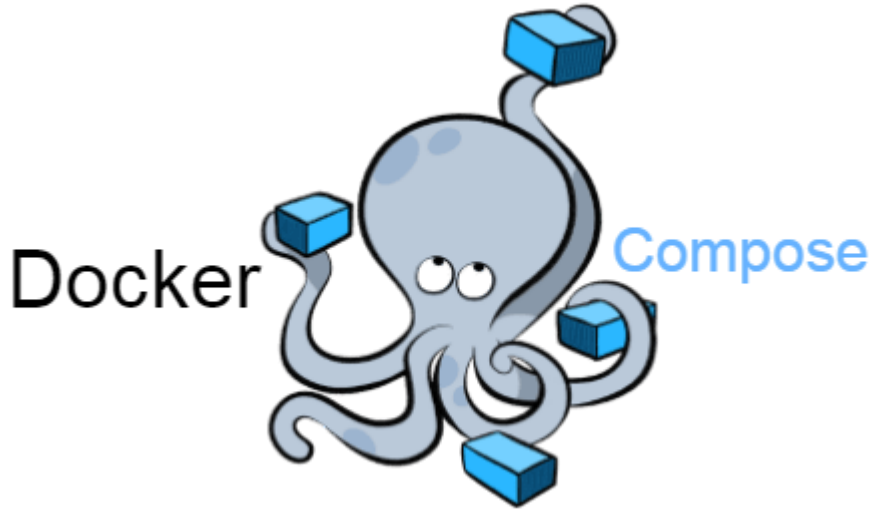
- Compose is a tool for defining and running multi-container Docker applications.

- Create a single a YAML file to configure application's services.

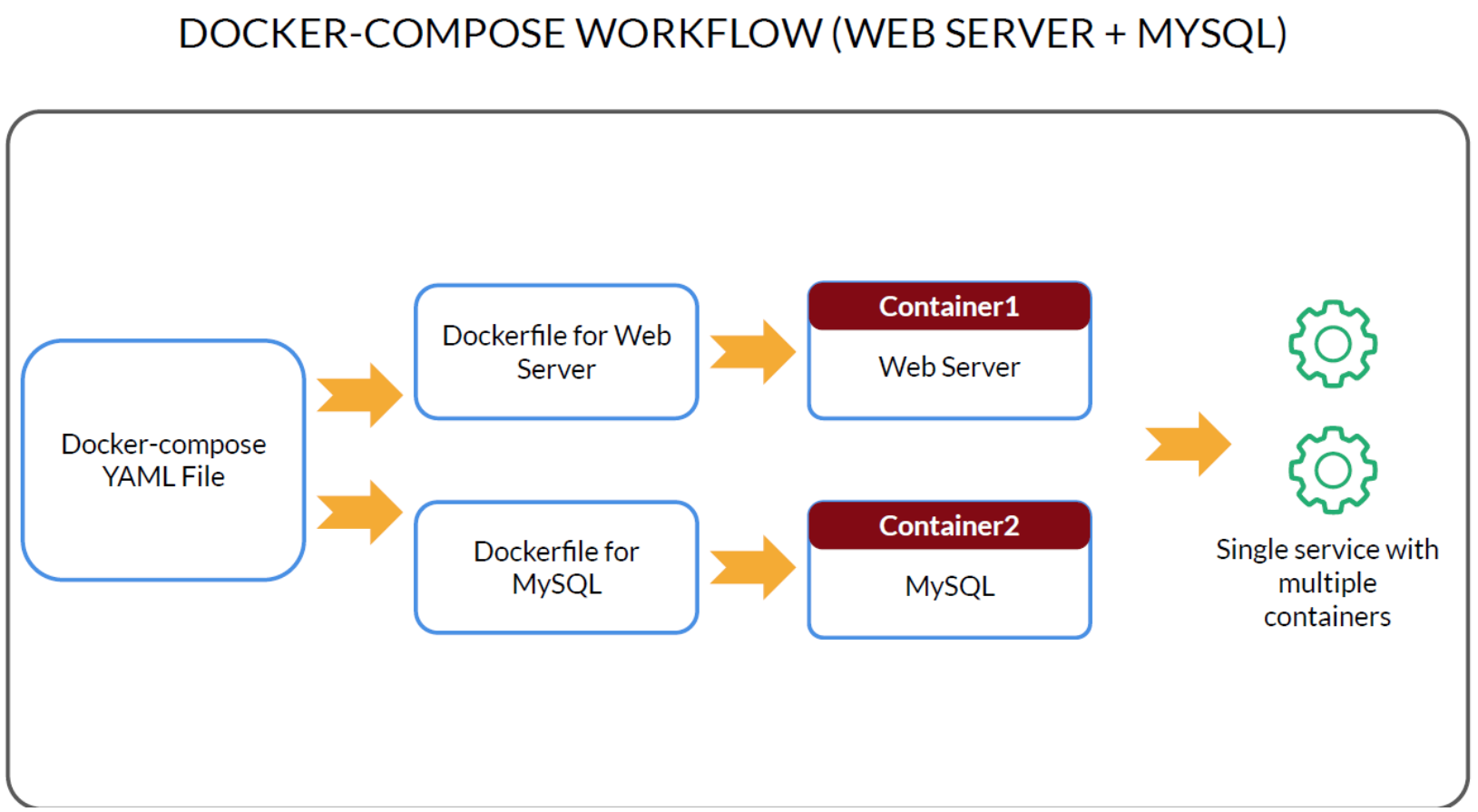
- Works in all environments: production, staging, development, testing, as well as CI workflows.

A docker-compose.yml looks like this:

```
version: '2'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: redis
```



Docker-Compose Workflow



Features of Docker Compose

- Multiple isolated environments on a single host
- Preserve volume data when containers are created
- Only recreate containers that have changed
- Variables and moving a composition between environments

Common use cases

➢ **Development environments:**

Compose file provides a way to document and configure all of the application's service dependencies with a single command (docker-compose up)

➢ **Automated testing environments:**

- Automated end-to-end testing requires an environment in which to run tests.
- Compose provides a convenient way to create and destroy isolated testing environments for your test suite.
- By defining the full environment in a Compose file, you can create and destroy these environments

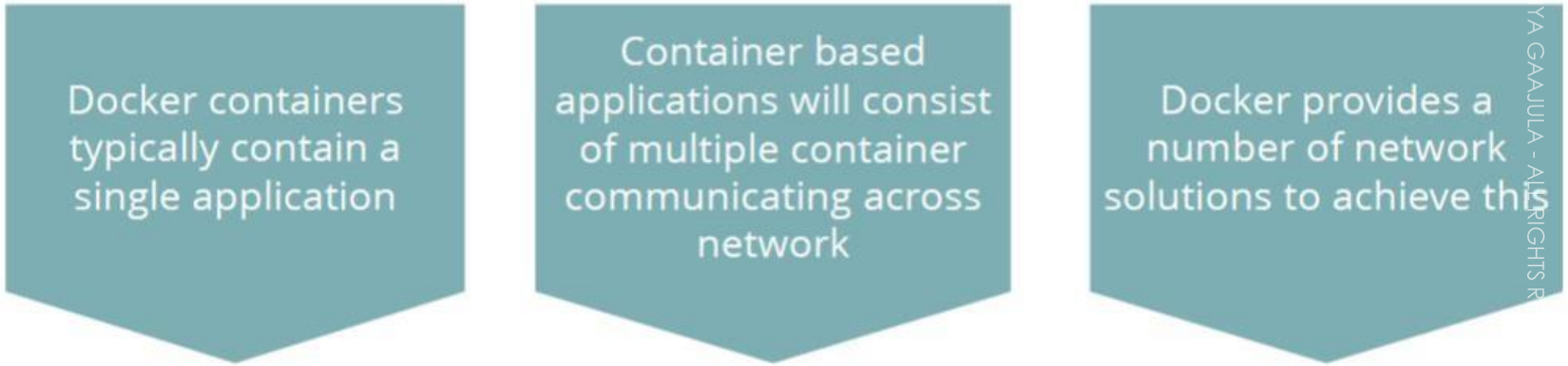
Docker Compose Commands

Command	Description
docker-compose up	Starts all the containers
docker-compose ps	Can be used to verify the status of running containers
docker-compose stop	Can be used to stop the containers
docker-compose logs	Can be used to check the logs of the containers
docker-compose down	Can be used to remove the containers

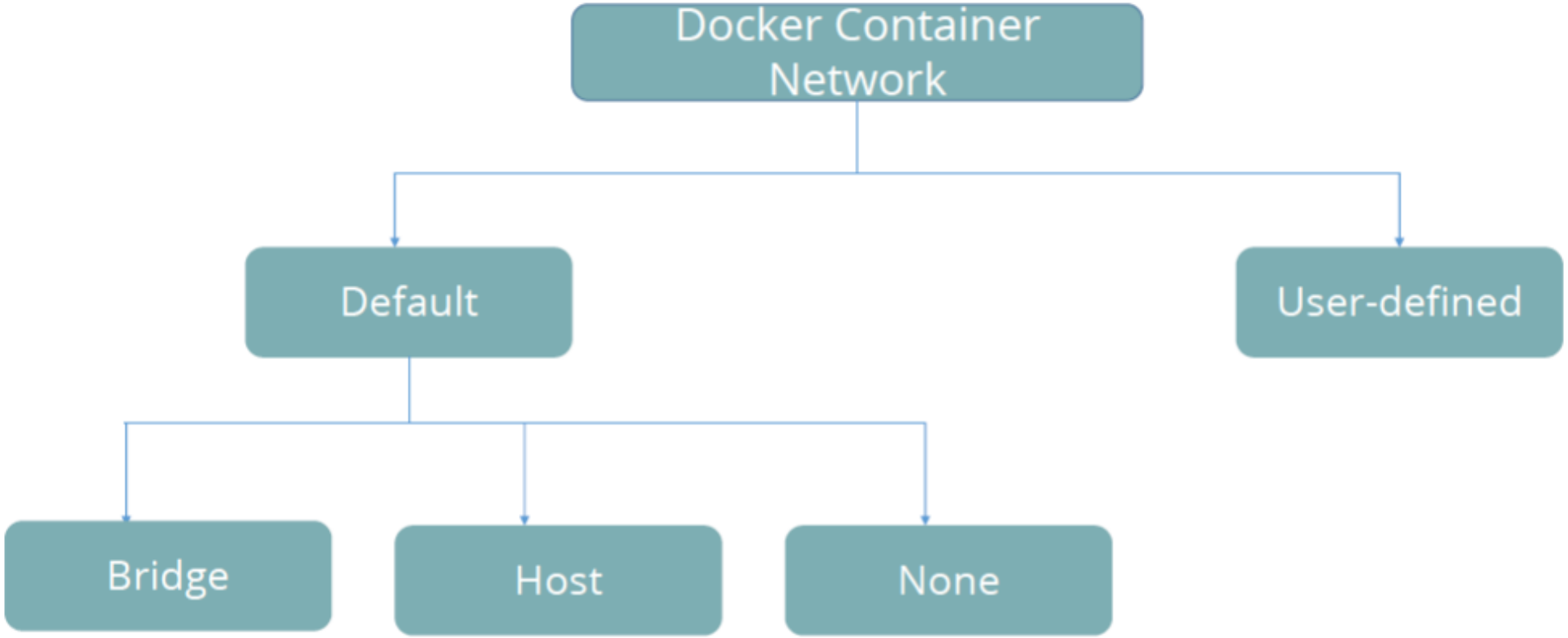
THANK YOU

Docker Networking

Need for Docker Container Network



Types of Docker Container Network



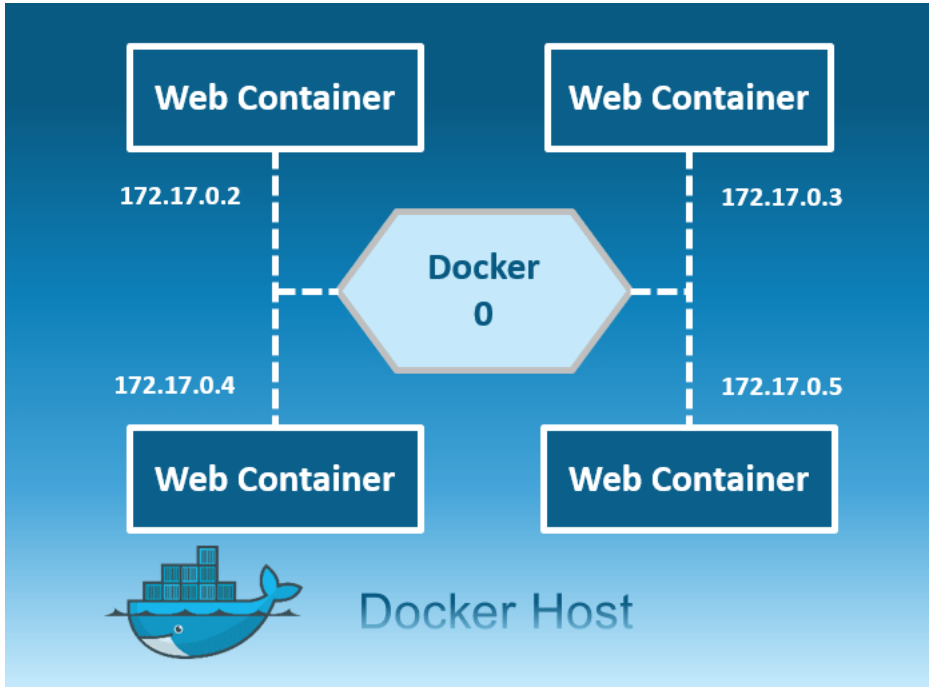
Docker Container Networks—Default

- Docker creates three networks by default, which can't be removed
- The none network is local to the container – it has localhost
- The host network gives the container the same network as the host
- The bridge network is the default
- A docker0 or bridge0 virtual interface is created on the host

```
$ docker network ls
  NETWORK ID    NAME   DRIVER  SCOPE
   9d6a9ab487ba     bridge   bridge   local
   c7956146a031       host      host  local
   115642b21a91       none     null    local
```

Docker Container Networks—bridge

- Connect the container to the bridge via veth interfaces
- Automatically created when docker starts
- docker0 is the default bridge network



Example:

$ docker run -it --name=test2 ubuntu /bin/bash

$ docker network inspect bridge

Default Network

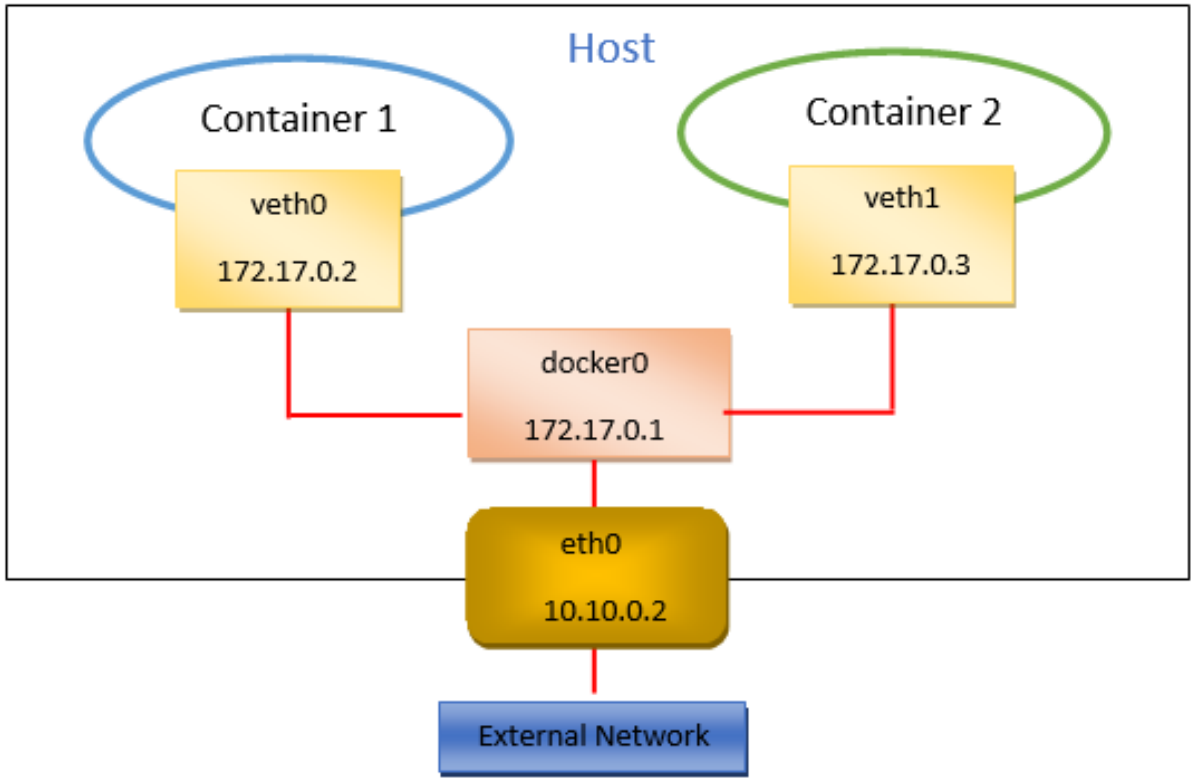
docker0 created automatically on the host machine and ***docker run*** adds containers to this network

- Docker creates a subnet and gateway for the bridge network
- Subnet CIDR block selected randomly from a private range that are not in use on host machine
- The network itself isolates the containers from external networks

Note:

- We can ping to container in same network
- We can ping external network like google.com

Docker0 Bridge Network



Default—Bridge Network

The bridge network creates a subnet and a subnet mask.

$ docker network inspect bridge

[{

"Name": "bridge",

"Id": "9d6a9ab487ba1d00715bfa60833a9cf5daa564d9a02918424ca3d38e26b2b5f8", "Scope": "local",

"Driver": "bridge",

"EnableIPv6": false, "IPAM": {

"Driver": "default",

"Options": null,

"Config": [

{

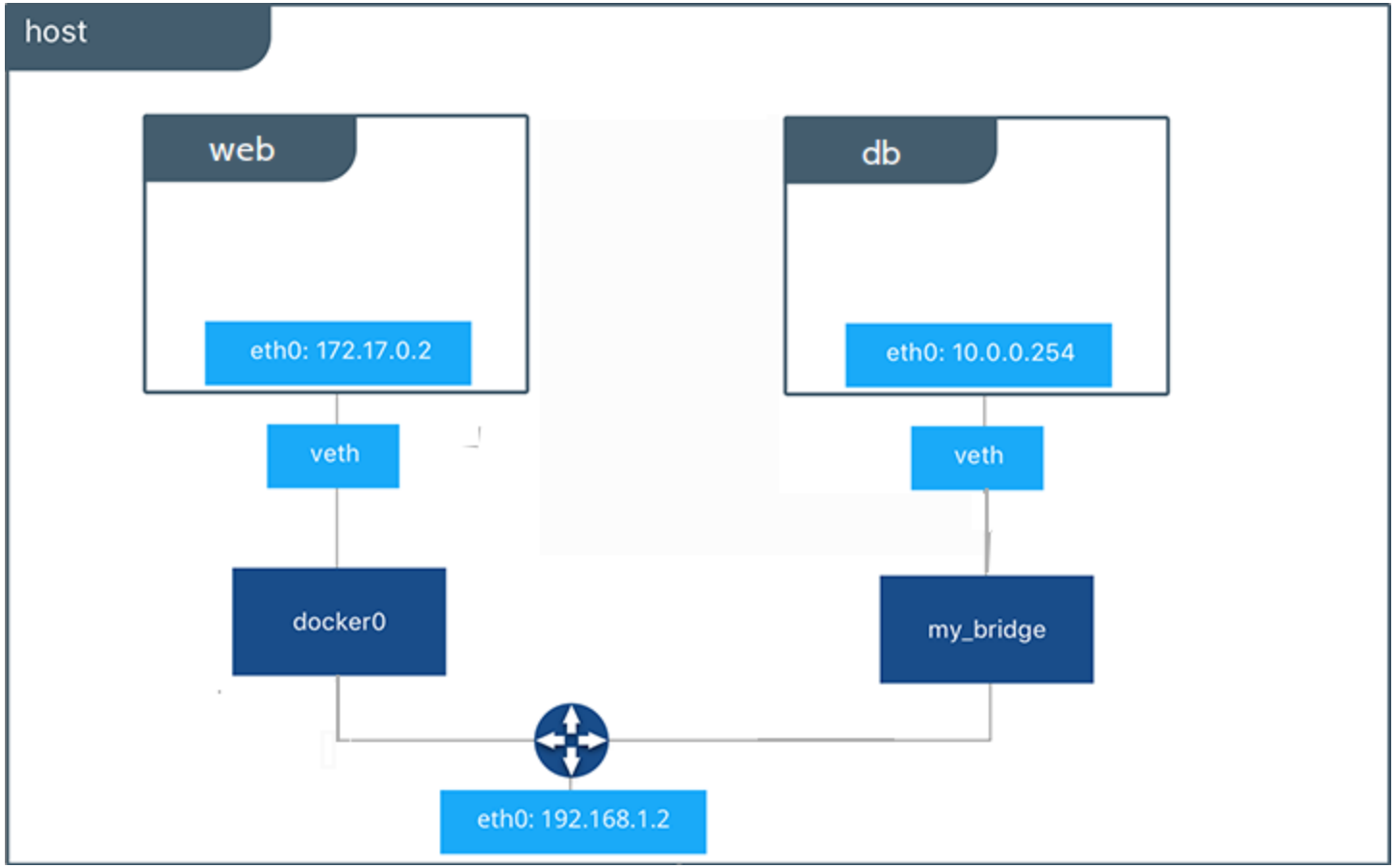
"Subnet": "172.17.0.0/16",

"Gateway": "172.17.0.1"

}

]},

User-Defined Network



Docker Container Networks—User-defined

- User defined networks can be created
- Docker provides drivers including bridge
- Containers can only communicate with other containers on the same network
- Multiple networks can be created
- Containers can be connected to multiple networks
- Can communicate with any container on any connected network

Bridge Network Commands

docker network ls
docker network create --driver bridge nw1
docker network inspect nw1

Create a container to use the new network:

docker run --network=nw1 -it --name=container3 ubuntu

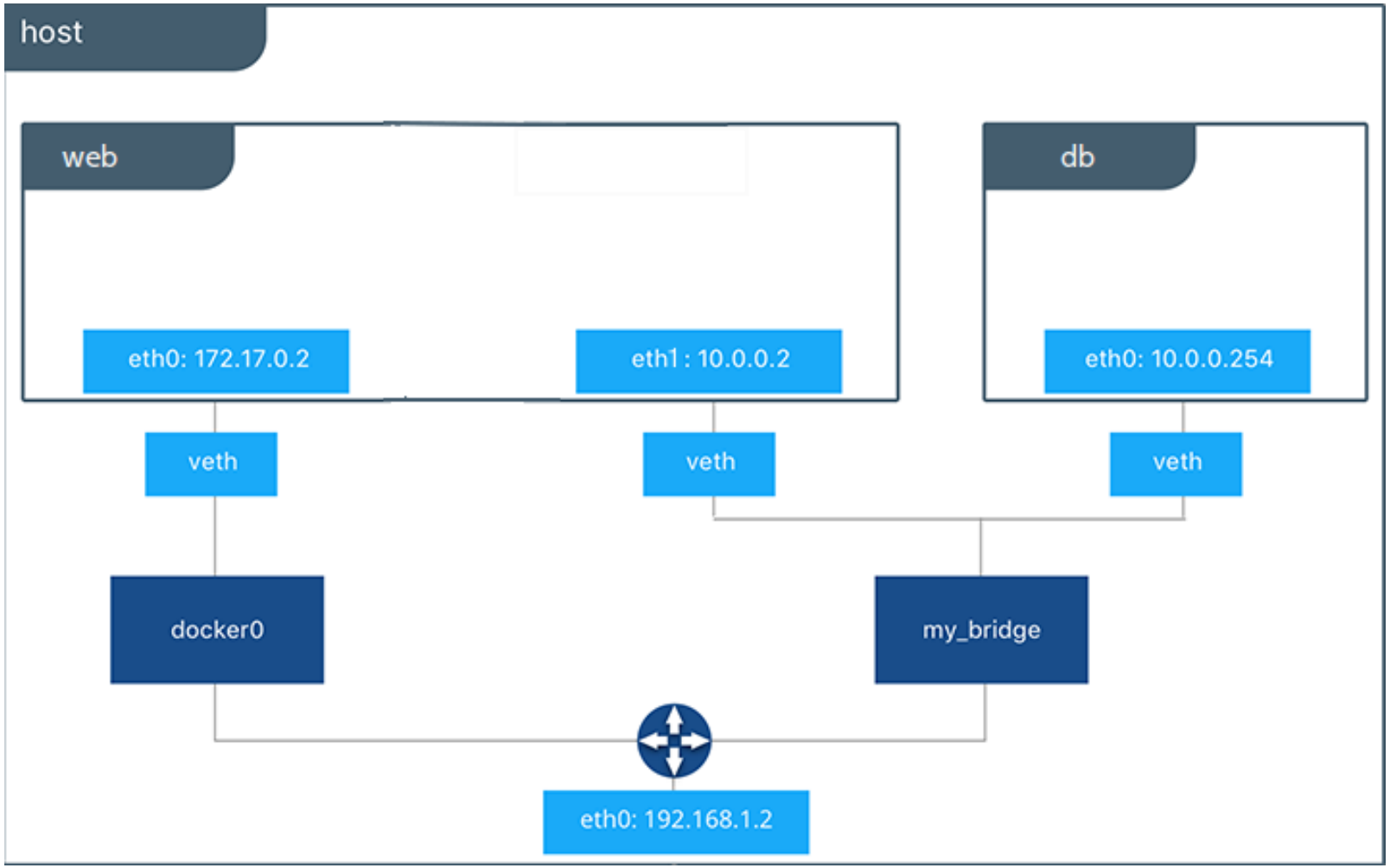
docker network inspect nw1

Default—Bridge Network

- The bridge network assigns MAC and IP addresses to each container.

```
$ docker network inspect bridge
...
"Containers": {
"eb6dc24ff73fff0da60e98b02aa28e76f92f316aeed73f774ef7b3b0220b5b69": {
"Name": "centos",
"EndpointID":
"5b4437f5c54b0923f4558ecc01f9326fafa0fbb4d1f8564131d6dac9bd47e0de",
"MacAddress": "02:42:ac:11:00:02",
"IPv4Address": "172.17.0.2/16",
"IPv6Address": ""
}
},
```

Connecting Bridges (docker network connect)

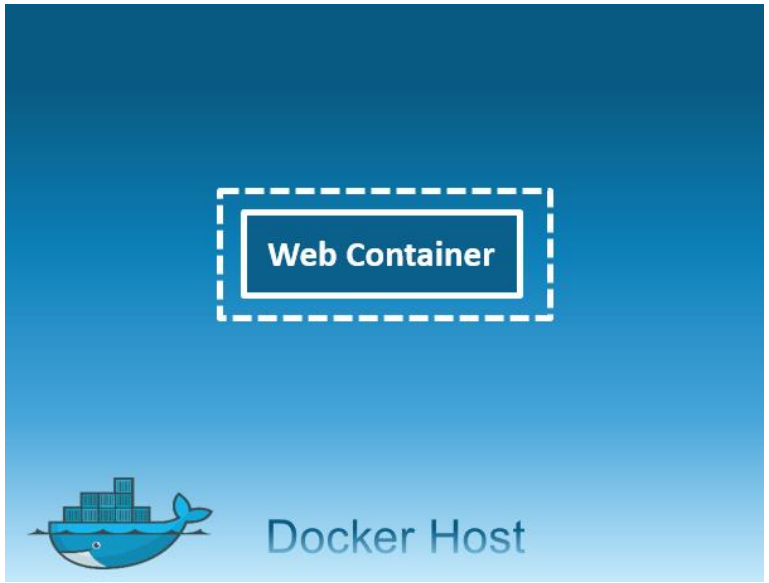


Default—Host Network

- A container attached to a host network has the same network as the host
- For instance, if you run a container which binds to port 80 and you use host networking, the container's application is available on port 80 on the host's IP address.
- Host mode networking can be useful to optimize performance, and in situations where a container needs to handle a large range of ports

Default—None Network

- Container will not have access to any external routes
- Container doesn't have access to any other containers as well
- Container will only have a local loopback interface



Example:

- docker run -it --name=test2 ***--net=none*** ubuntu /bin/bash

Default—None Network

```
$ docker attach nonenetcontainer

root@0cb243cd1293:/# cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

root@0cb243cd1293:/# ip -4 addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

THANK YOU

Container Orchestration

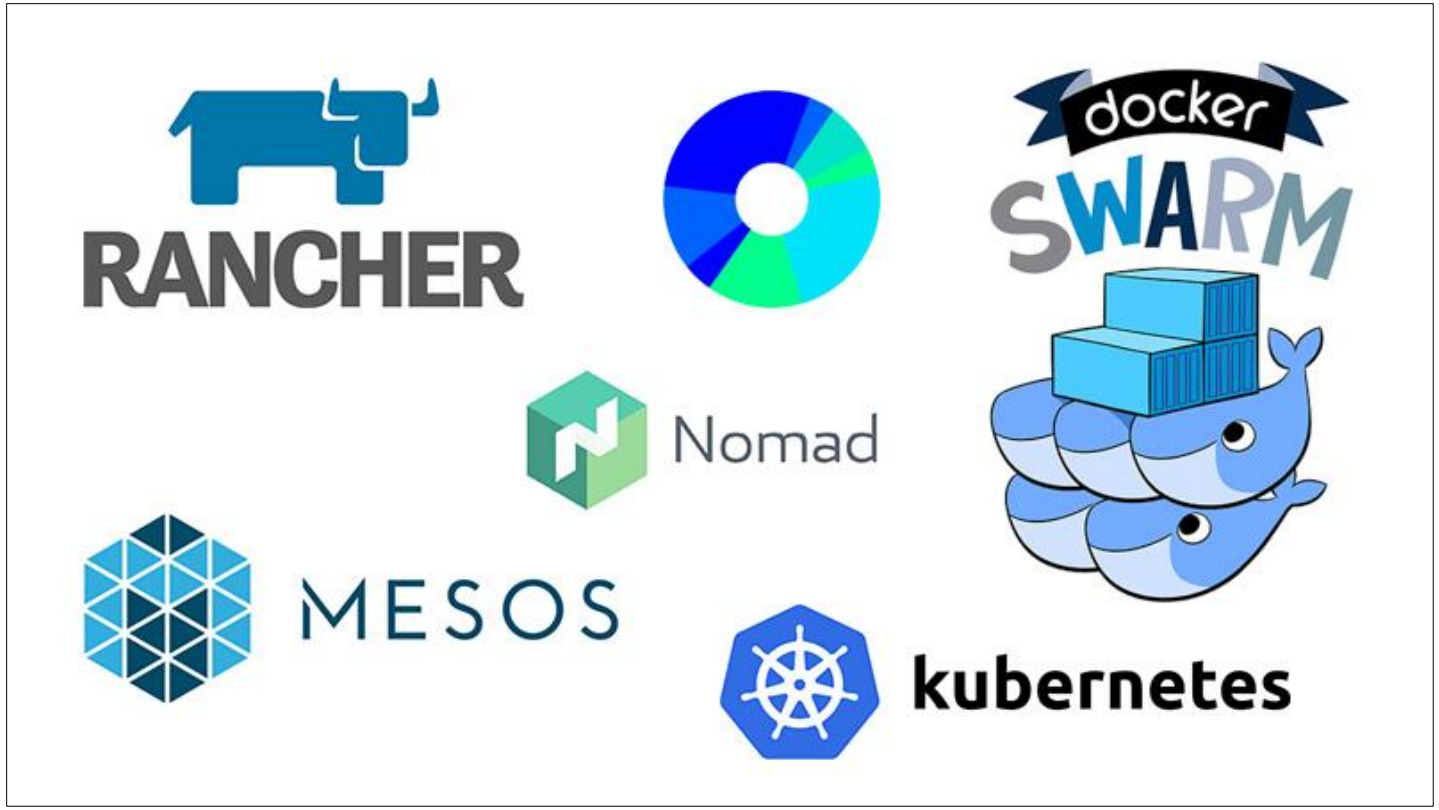
What is Container Orchestration?

Container orchestration is all about managing the lifecycles of containers, especially in large, complex and dynamic environments. Dev/Ops teams use container orchestration to manage and automate many tasks like:

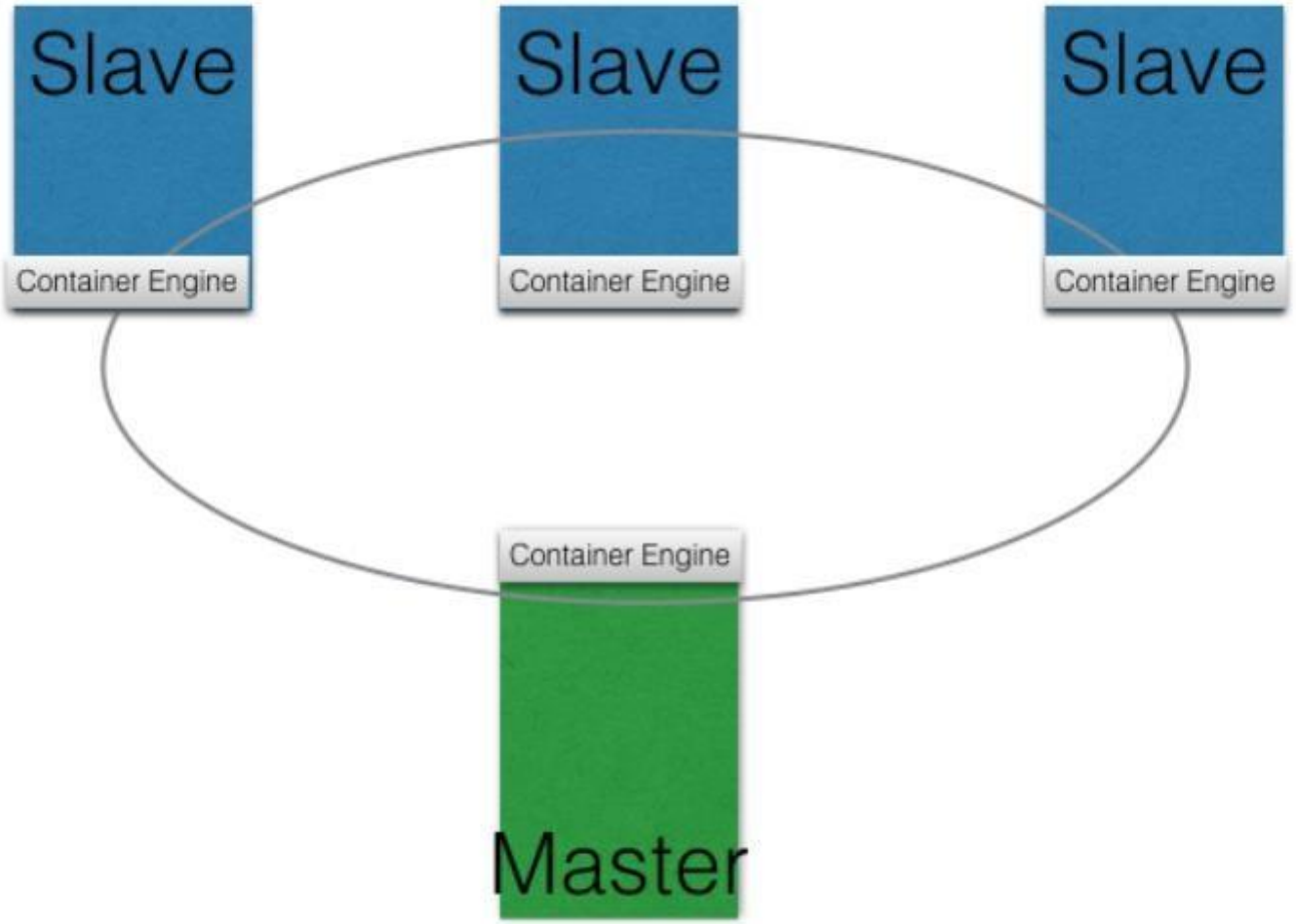
- Provisioning and deployment of containers
- Redundancy and availability of containers
- Scaling up or Scaling down containers.
- Shifting containers from one host to another if host goes down
- Resources allocation to containers
- Expose container with the outside world
- Load balancing and service discovery of containers
- Health check and monitoring of containers and hosts

Container Orchestrators

- Kubernetes
- Docker Swarm
- Mesos Marathon
- Amazon ECS . . . etc

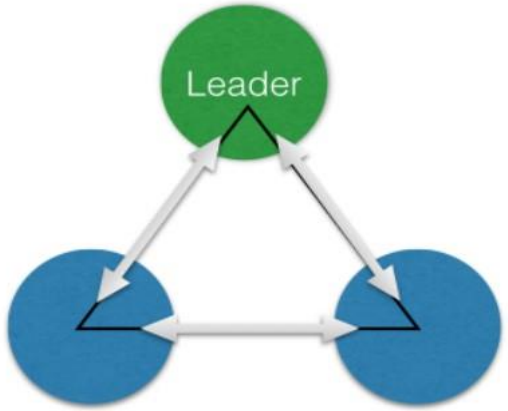


Master / Slave Architecture



Key/Value Store

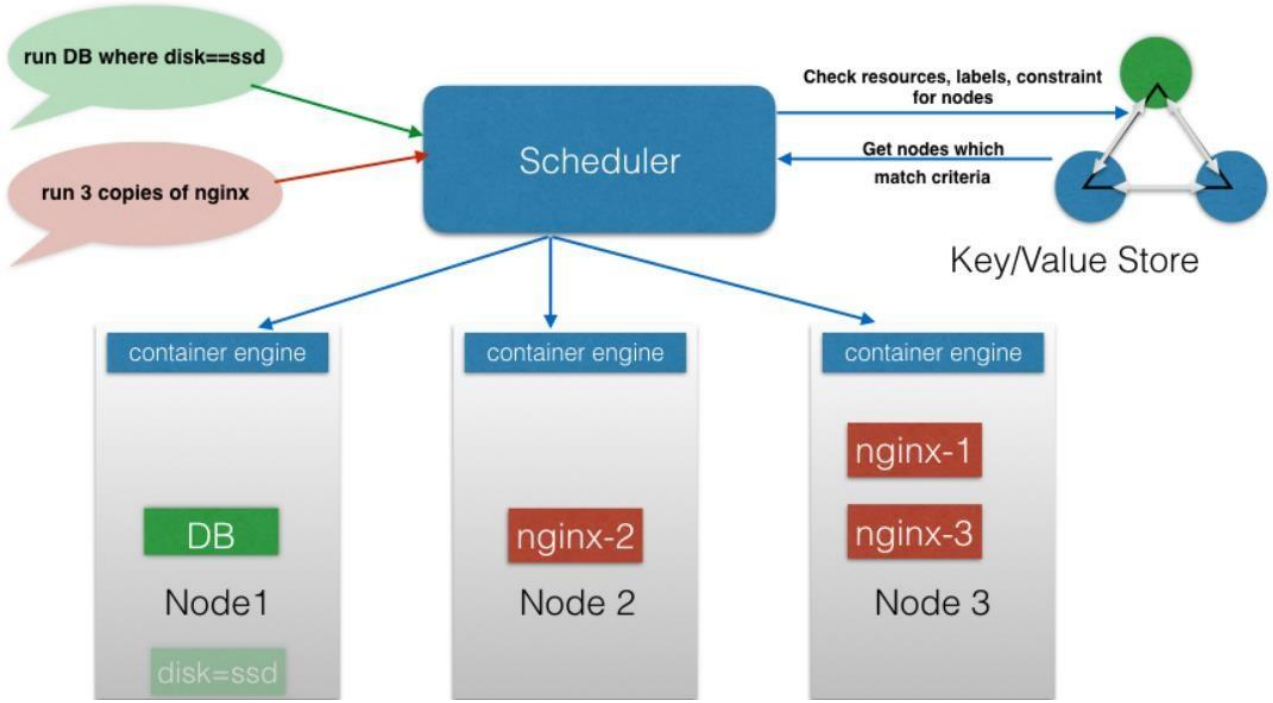
Single source of truth



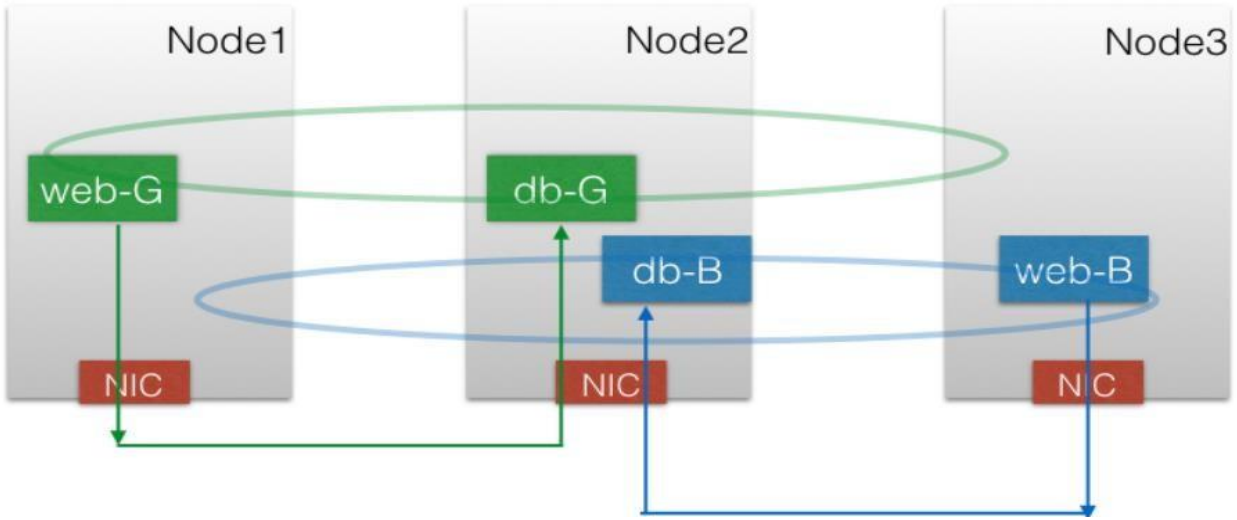
Key/Value Store

- Accessed over HTTP
- Can store any information in key/value format, like :-
  - configuration
  - subnet details
  - Node specific information.
- One can watch on key changes and trigger action based on changes
- Examples
  - etcd

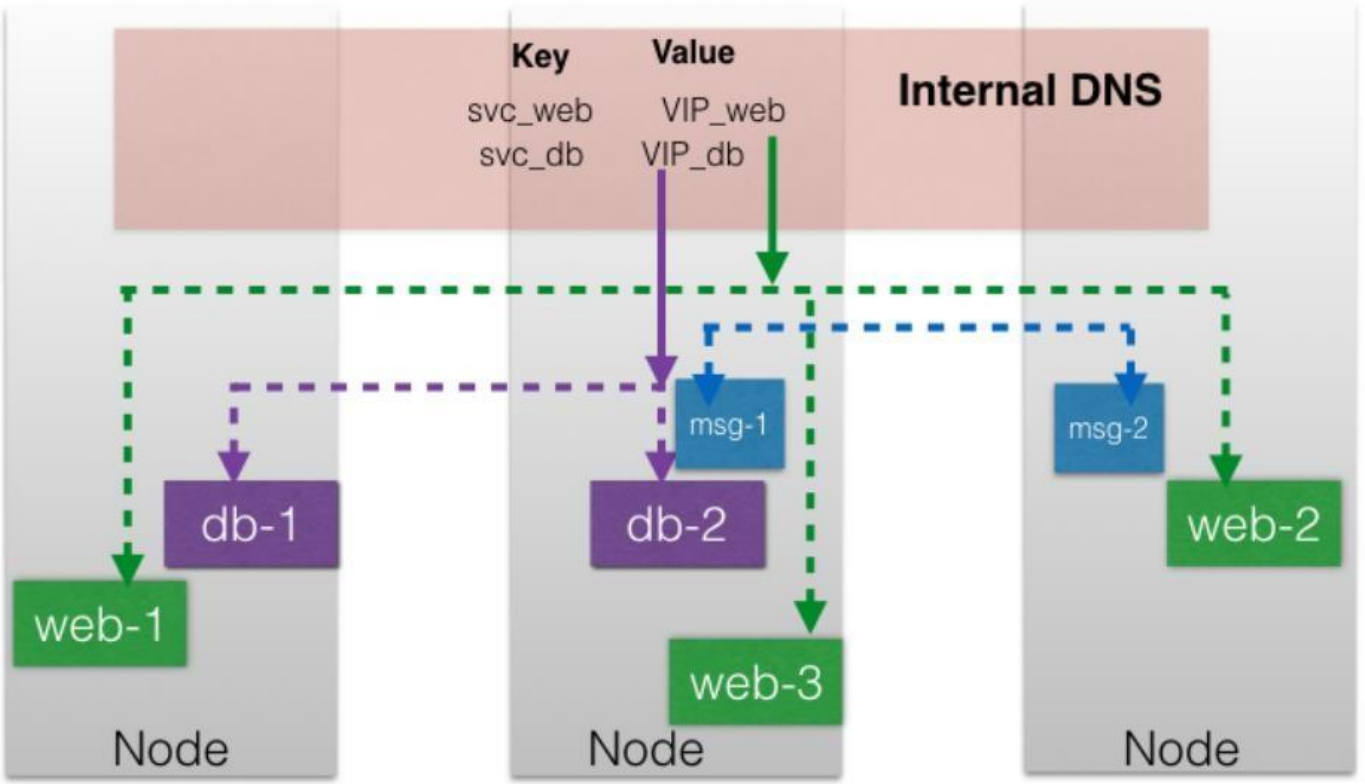
Scheduler



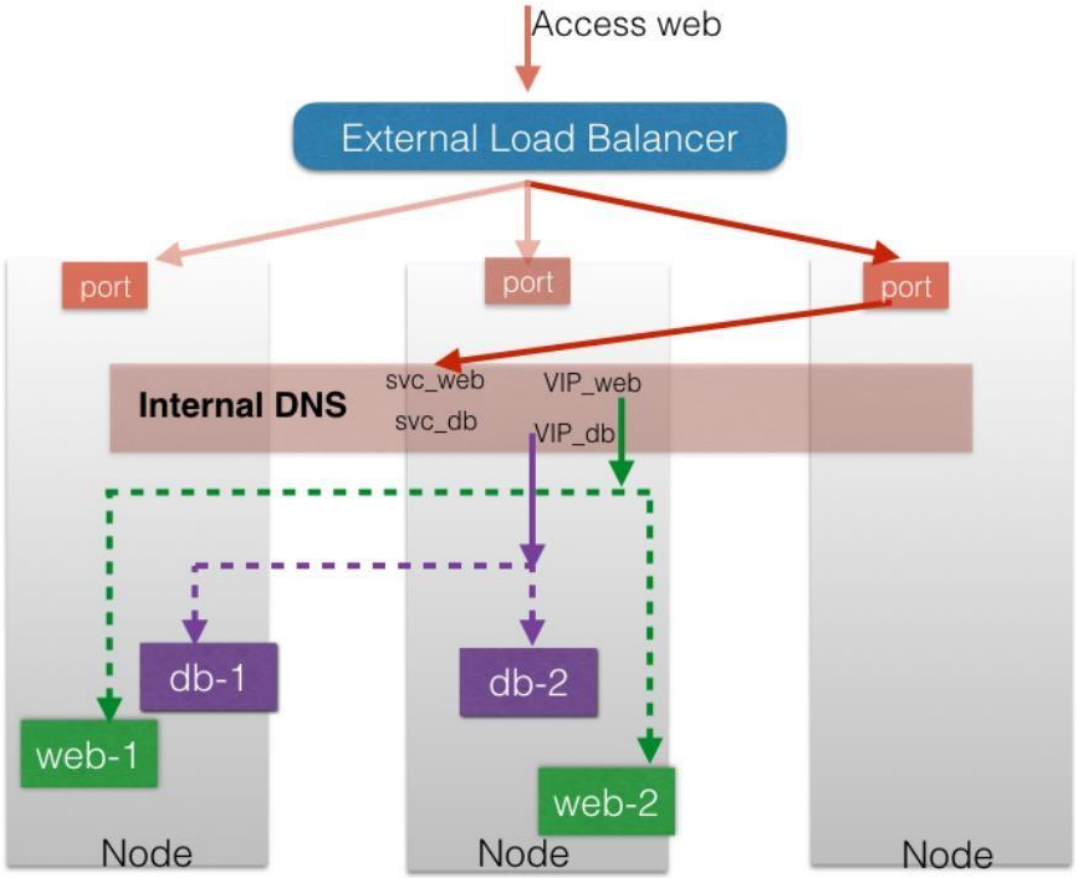
Network for containers of different nodes to talk to each other



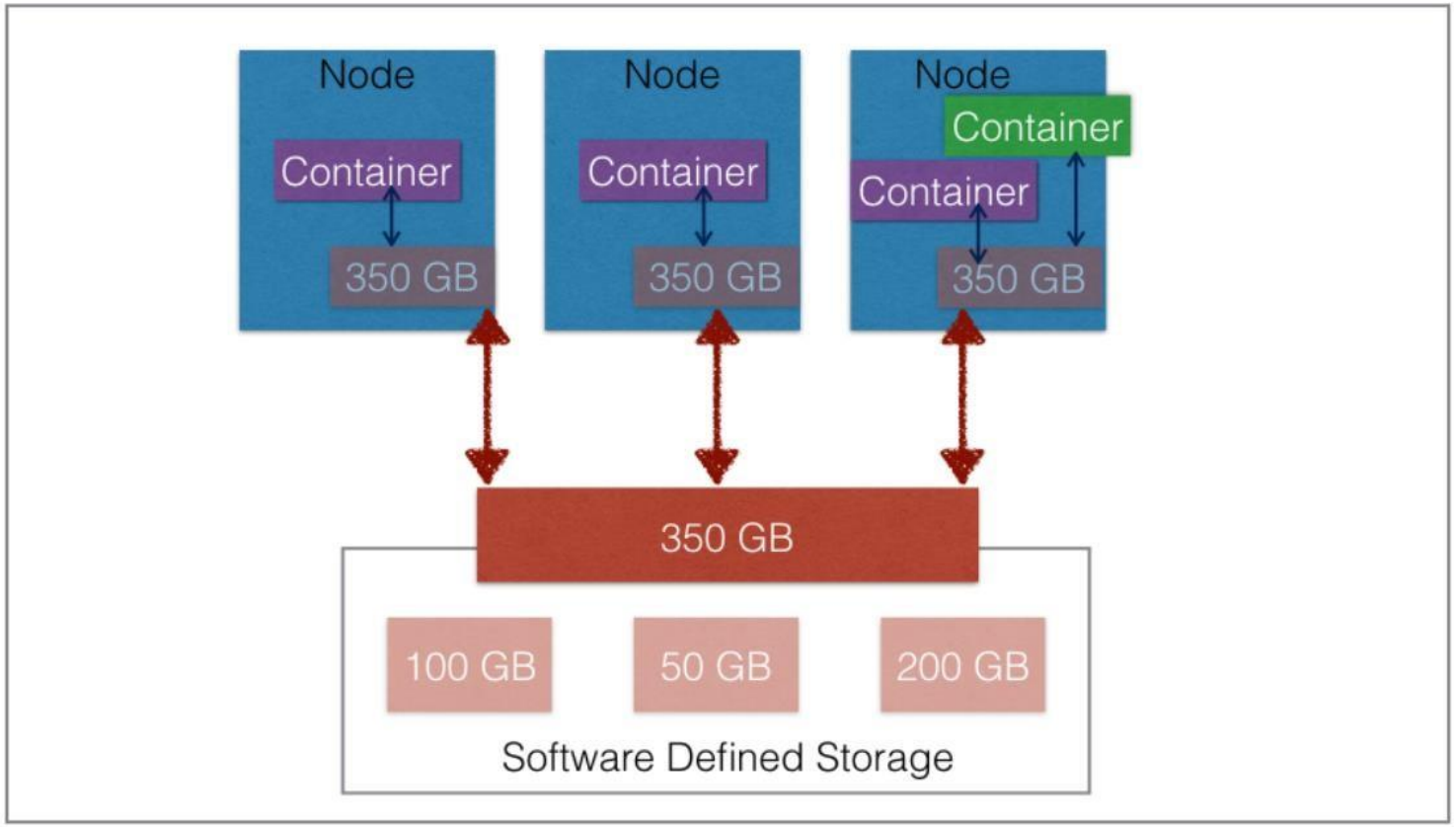
Service Discovery



Access from External World



Access to External Storage

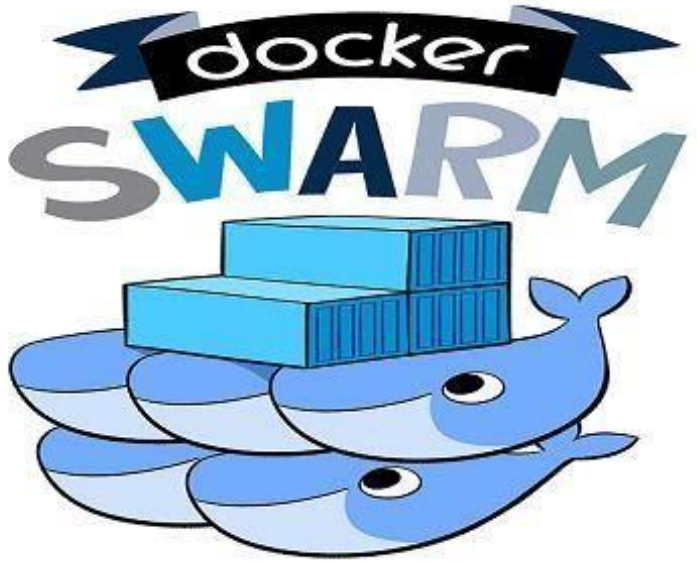


THANK YOU

Docker Swarm

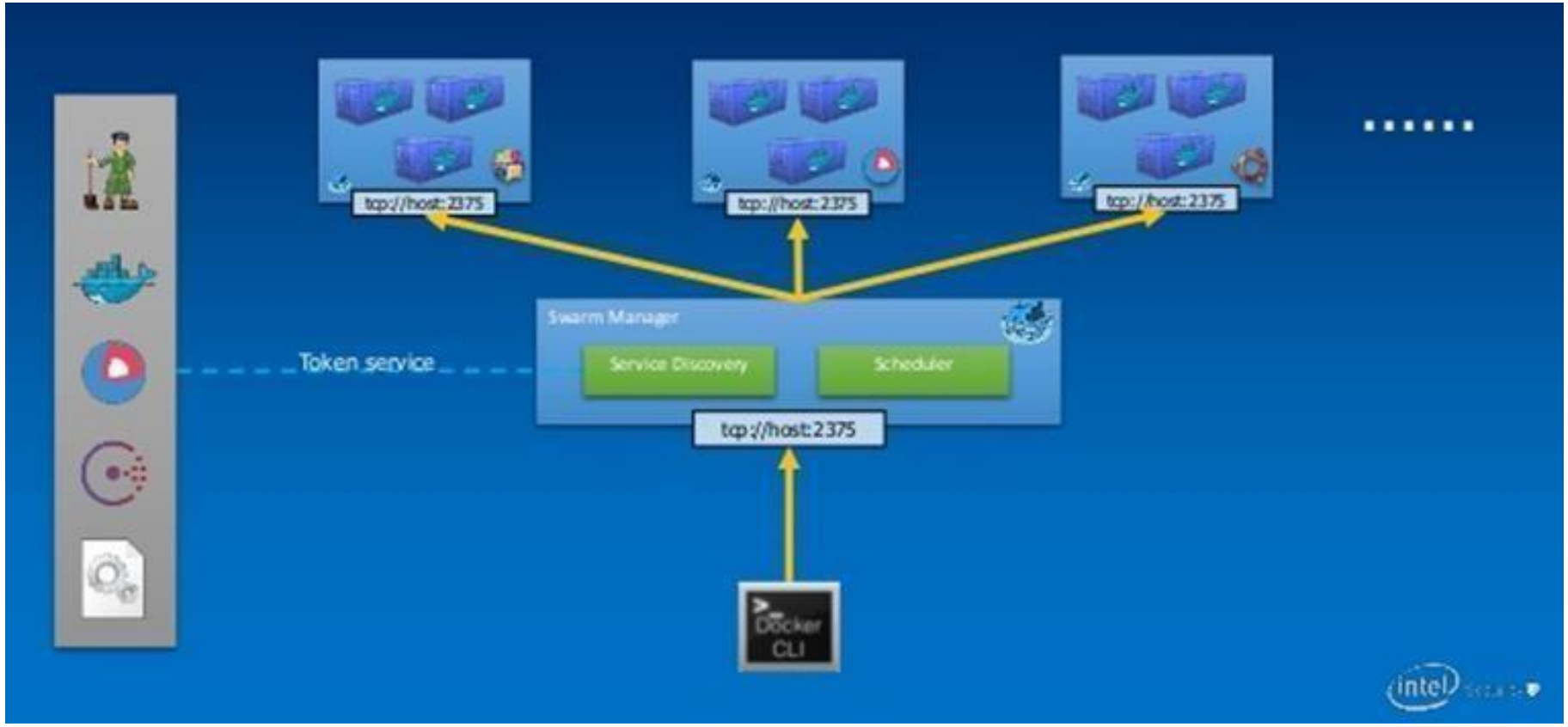
Docker Swarm

- Docker Swarm is clustering for Docker
- It turns several Docker hosts into a single virtual Docker host
- The regular Docker client works transparently with Swarm
- The Swarm is controlled by a Swarm Manager
- Each Docker node communicates with the manager
- It can be installed manually or by using Docker Machine



Swarm Architecture

- The Swarm is controlled by a Swarm Manager
- Each Docker node communicates with the manager
- It can be installed manually or by using Docker Machine



Creating a Swarm

- A machine needs to be designated as a manager
- There can be several managers
- The manager is created by initializing a Swarm
- Swarm must be performed on the manager machine

```
$ docker swarm init
Swarm initialized: current node (5lo6zmzvashexpfm8ipnlni37) is now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-51icto7l3hfkym98e2cq0rflh6a6hkyqzxpyb8jaid3qzx5kmm-
864t0x9ebw4a2ymsms1kvq9s9 \
    192.168.0.38:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.
```

Managing a Swarm

- The manager node is automatically added to the Swarm
- Information about the Swarm can be found using docker info

```
...
Swarm: active
 NodeID: 5lo6zmzvashexpfm8ipnlni37
 Is Manager: true
 ClusterID: e9ff1sv78oxyb1989xa0hvy77
 Managers: 1
 Nodes: 1
...
 Node Address: 192.168.0.38
...
```

Joining a Swarm

- Nodes can be added to the Swarm
- Ensure that firewall rules aren't blocking port 2377 on the manager
- The nodes can be listed

```
$ docker swarm join \
    --token SWMTKN-1-
51icto7l3hfkym98e2cq0rflh6a6hkyqzxpyb8jaid3qzx5kmm-
864t0x9ebw4a2ymsms1kvq9s9 \
    192.168.0.38:2377
 $ docker node ls

 ID                            HOSTNAME                  STATUS   AVAILABILITY   MANAGER STATUS
 5lo6zmzvashexpfm8ipnlni37   * hp                        Ready    Active         Leader
 esc6tvcf01tx09zwzzr90no53     localhost.localdomain     Ready    Active
```

Docker Swarm—Services

- A Docker Service is a container
- Services run in Docker Swarm
- They can only be started on a Swarm Manager node
- Multiple copies of services can be run
- The Swarm Manager replicates the service on other nodes in the Swarm

Docker Swarm - Services (Contd.)

- A service can be added to the manager node
- A service is a container
- You can also inspect the service
- It also appears as a running container

```
$ docker service create --replicas 1 --name helloworld alpine
ping docker.com
1rw0x7pohbf4mvz9isdiecbe2

$ docker service ls
ID              NAME         REPLICAS   IMAGE    COMMAND
1rw0x7pohbf4    helloworld   0/1        alpine   ping docker.com

$ docker service inspect –pretty helloworld
$ docker ps
```

Docker Swarm Services - Scaling

- A service can be scaled
- The service will be duplicated and run on different nodes
- The service can be removed from all nodes

```
$ docker service ps helloworld
$ docker service scale helloworld=2
helloworld scaled to 2

$ docker service ps helloworld

ID                         NAME          IMAGE   NODE                    DESIRED STATE  CURRENT STATE
84lhnmfeob8y4r1tevkvsa0d9  helloworld.1  alpine  hp                      Running        Running
34u5pzw5yldnessr4radaip26  helloworld.2  alpine  localhost.localdomain   Running        Running

$ docker service rm helloworld
```

Applying Rolling Updates

- We will deploy a service based on the Redis 3.0.6 container image
- Then we will upgrade the service to use the Redis 3.0.7 container image using rolling updates
- We configure the rolling update policy at service deployment time
- The --update-delay flag configures the time delay between updates to a service task or sets of tasks
- By default the scheduler updates 1 task at a time
- By passing the --update-parallelism flag to configure the maximum number of service tasks that the scheduler updates simultaneously.

```
$ docker service create --replicas 3 --name redis --update-delay 10s redis:3.0.6
$ docker service inspect --pretty redis
```

```
$ docker service update --image redis:3.0.7 redis
$ docker service inspect --pretty redis
$ docker service update redis
$ docker service ps redis
```

THANK YOU