What is Kubernetes?

What is Kubernetes

- Kubernetes is an open source **orchestration** system for Docker Containers
    - It lets you schedule **containers** on cluster of machines
    - You can run **multiple containers** on one machine
    - You can run long running **services** (like web applications)
    - Kubernetes will **manage** the state of the containers
        - Can start the containers on specific nodes
        - Will restart a container when it gets killed

What is Kubernetes

- Instead of just running a few docker containers on one host manually, Kubernetes is a platform that will manage the containers for you

- Kubernetes cluster can start with one node until thousands of nodes

- Some other popular docker orchestrators are:
  - Docker Swarm
  - Mesos

Kubernetes Advantages

- You can run **Kubernetes** anywhere:
  - On-premise (own datacenters)
  - Public Cloud (AWS, Google Cloud)
  - Hybrid: Public & Private
- Open Source
- Backed by Google

Digital Transformation

Digital Transformation

Requires an Evolution in...

APPLICATIONS

New ways of developing, delivering and integrating applications

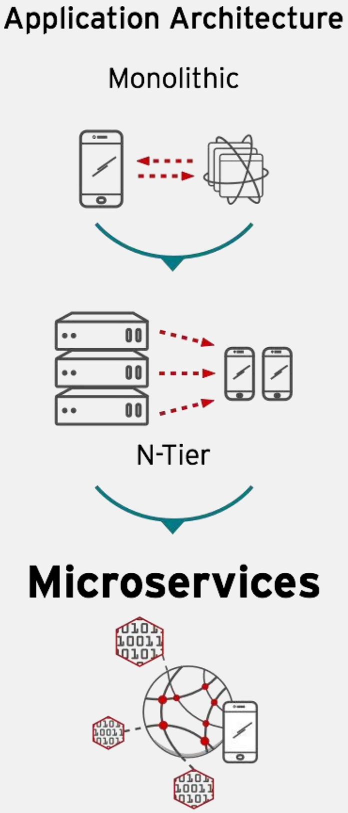
PROCESS

More agile processes across both IT and the business

INFRASTRUCTURE

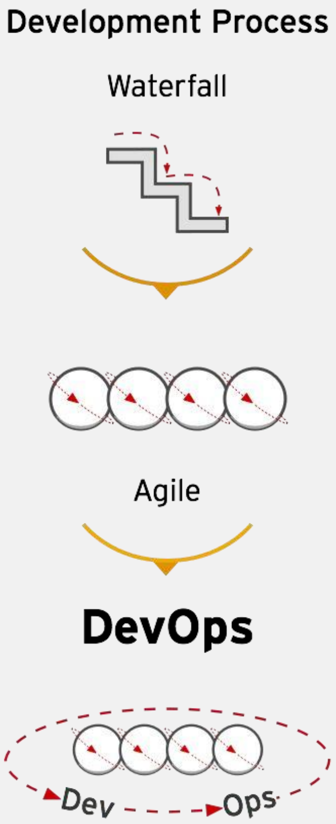
Modernize existing and build new cloud based infrastructure

Application Architecture



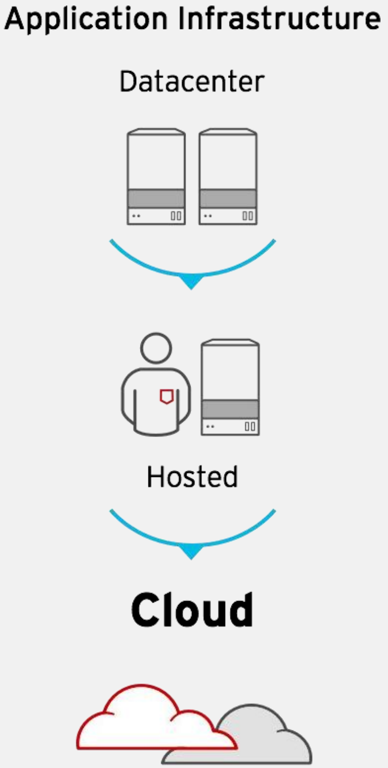
- Shift from monolithic applications to microservices
- Independently deployable and updatable, limited dependencies
- Optimized for agility & accelerated time to market

Development Process



- Shift to more agile development and deployment processes
- Increased collaboration between Development & Operations
- Move from Continuous Integration to Continuous Deployment

Platform Infrastructure



Application Infrastructure

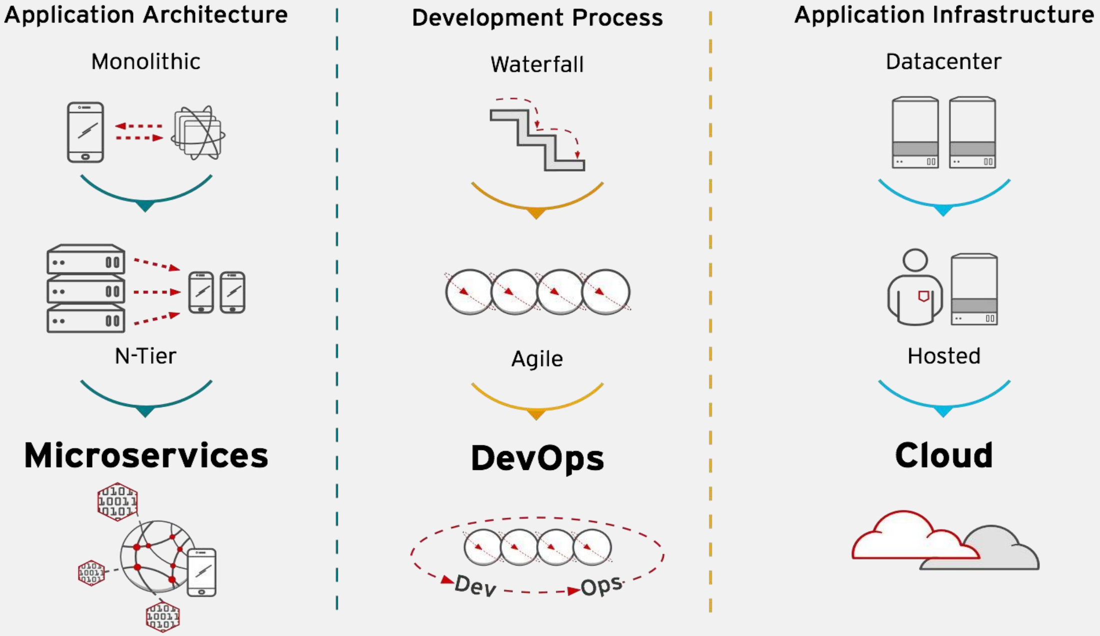
Datacenter

Hosted

Cloud

- Shift from virtualization to scale-out cloud infrastructure
- Rapid growth in public cloud usage for enterprises
- Hybrid cloud deployments span private & multiple public clouds

IT Must Evolve to Stay Ahead of These Trends



Application Architecture

Monolithic

N-Tier

Microservices

Development Process

Waterfall

Agile

DevOps

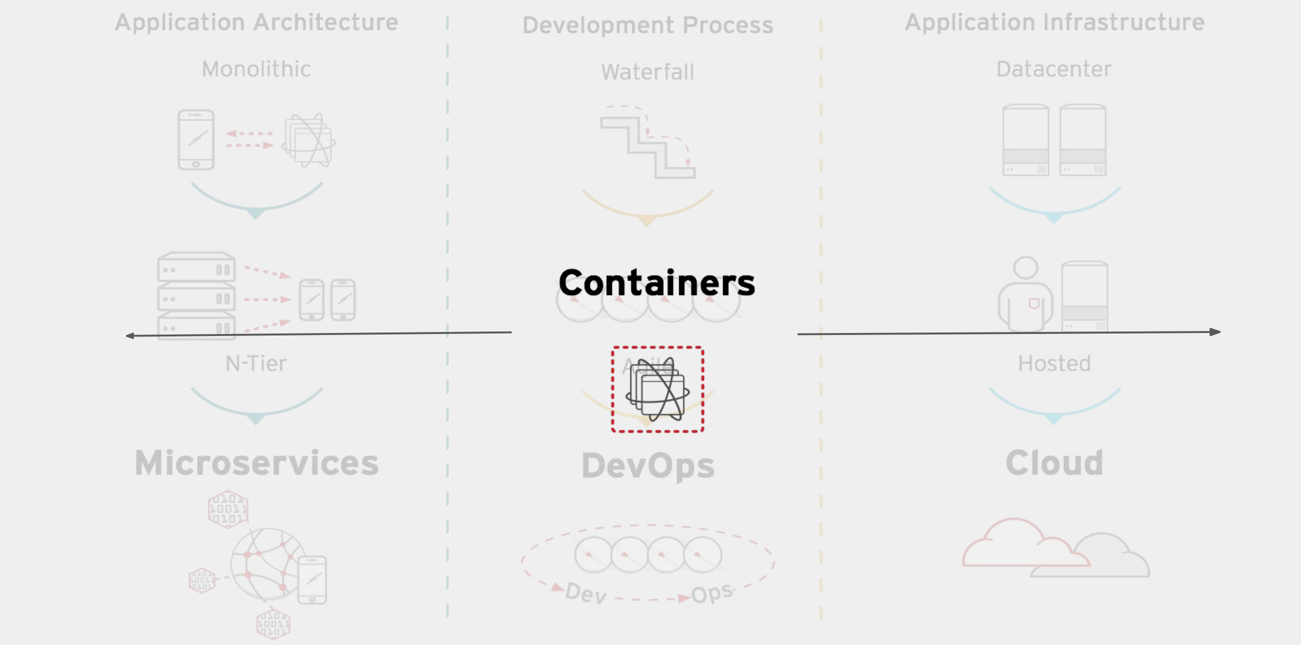
Application Infrastructure

Datacenter

Hosted

Cloud

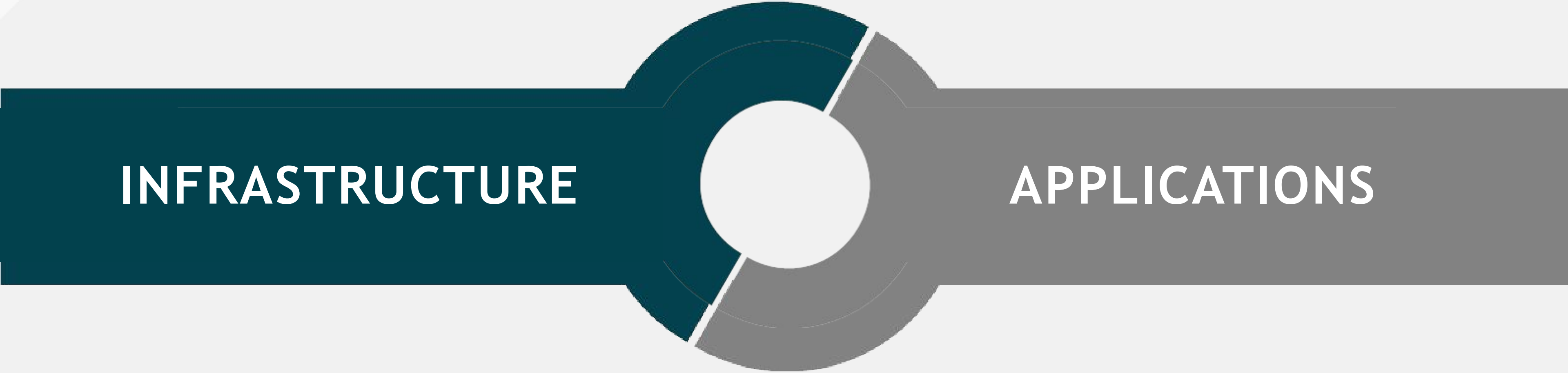
Containers - Transform Apps, Infrastructure & Process



LINUX CONTAINERS

WHAT ARE CONTAINERS?

It Depends Who You Ask



INFRASTRUCTURE

- Application processes on a shared kernel
- Simpler, lighter, and denser than VMs
- Portable across different environments

APPLICATIONS

- Package apps with all dependencies
- Deploy to any environment in seconds
- Easily accessed and shared

Introduction to Docker

The Challenge

Multiplicity of Stacks

Static website

nginx 1.5 + modsecurity + openssl + bootstrap 2

Background workers

Python 3.0 + celery + pyredis + libcurl + ffmpeg + libopencv + nodejs + phantomjs

User DB

postgresql + pgv8 + v8

Web frontend

Ruby + Rails + sass + Unicorn

Queue

Redis + redis-sentinel

Analytics DB

hadoop + hive + thrift + OpenJDK

API endpoint

Python 2.7 + Flask + pyredis + celery + psycopg + postgresql-client

Do services and apps interact appropriately?

Multiplicity of hardware environments

Development VM

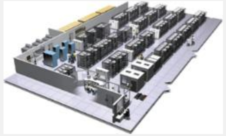
QA server

Customer Data Center

Public Cloud

Disaster recovery

Production Servers



Production Cluster

Contributor’s laptop

Can I migrate smoothly and quickly?

The Challenge

Multiplicity of Stacks

Static website

nginx 1.5 + modsecurity + openssl + bootstrap 2

Background workers

Python 3.0 + celery + pyredis + libcurl + ffmpeg + libopencv + nodejs + phantomjs

User DB

postgresql + pgv8 + v8

Web frontend

Ruby + Rails + sass + Unicorn

Queue

Redis + redis-sentinel

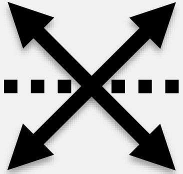
Analytics DB

hadoop + hive + thrift + OpenJDK

API endpoint

Python 2.7 + Flask + pyredis + celery + psycopg + postgresql-client

Do services and apps interact appropriately?



Multiplicity of hardware environments

Development VM

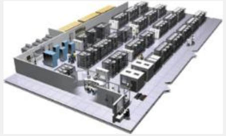
QA server

Customer Data Center

Public Cloud

Disaster recovery

Production Servers



Production Cluster

Contributor's laptop

Can I migrate smoothly and quickly?

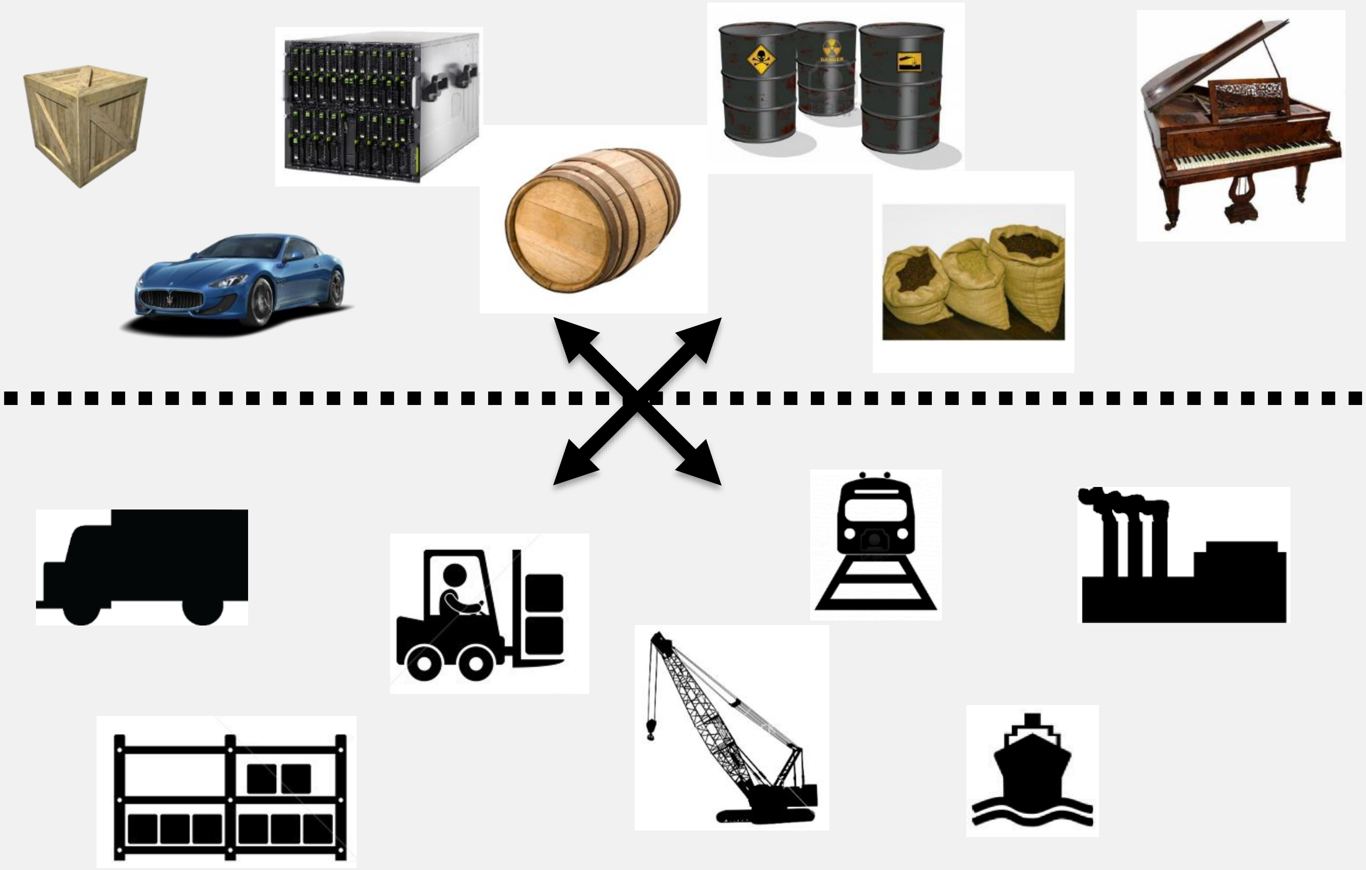
The Matrix From Hell

	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	?	?	?	?	?	?	?

Cargo Transport Pre-1960

Multiplicity of Goods

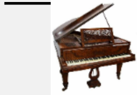
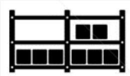
Do I worry about how goods interact (e.g. coffee beans next to spices)



Multipilicity of methods for transporting/storing

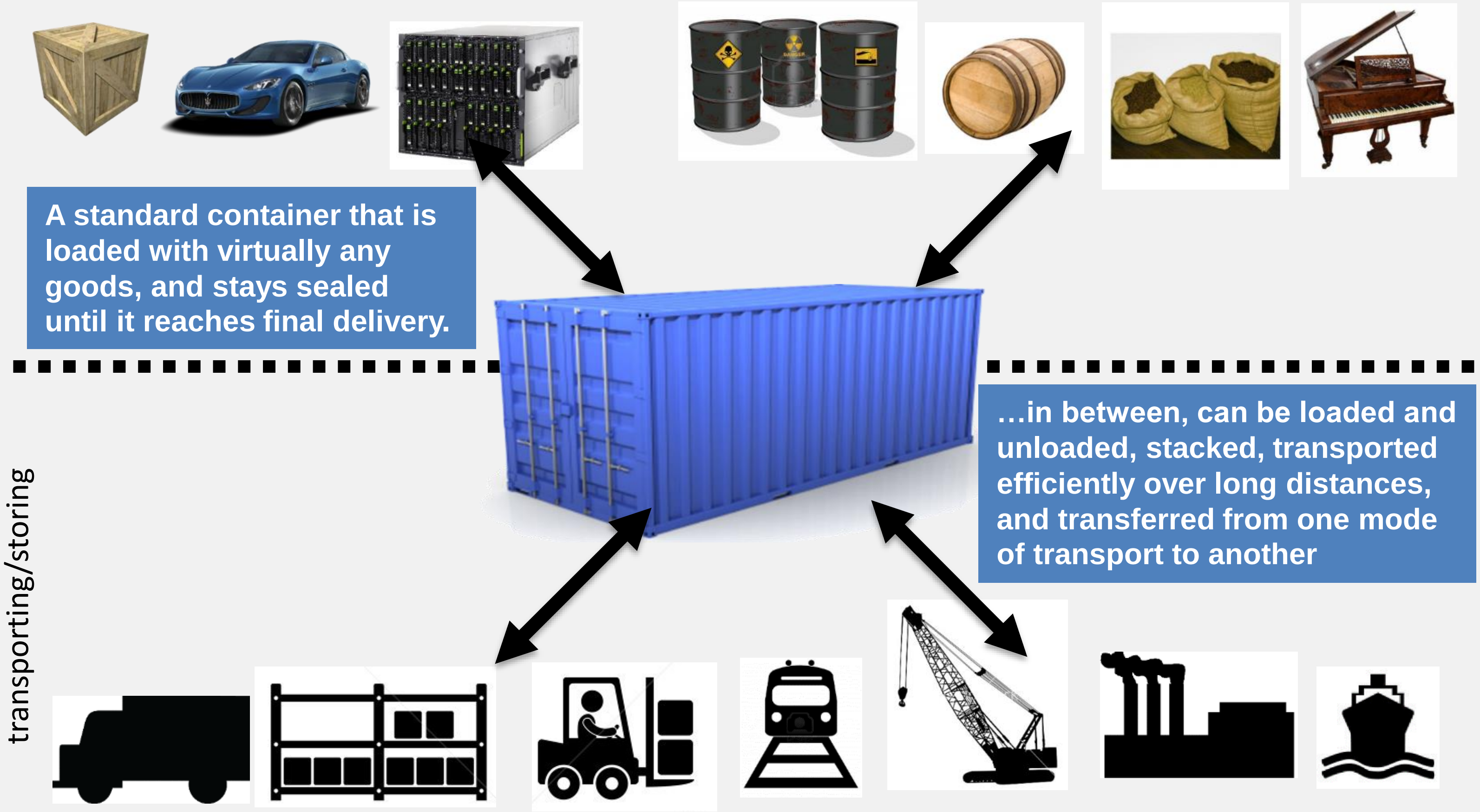
Can I transport quickly and smoothly (e.g. from boat to train to truck)

Also a matrix from hell

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

Solution: Intermodal Shipping Container

Multiplicity of Goods



A standard container that is loaded with virtually any goods, and stays sealed until it reaches final delivery.

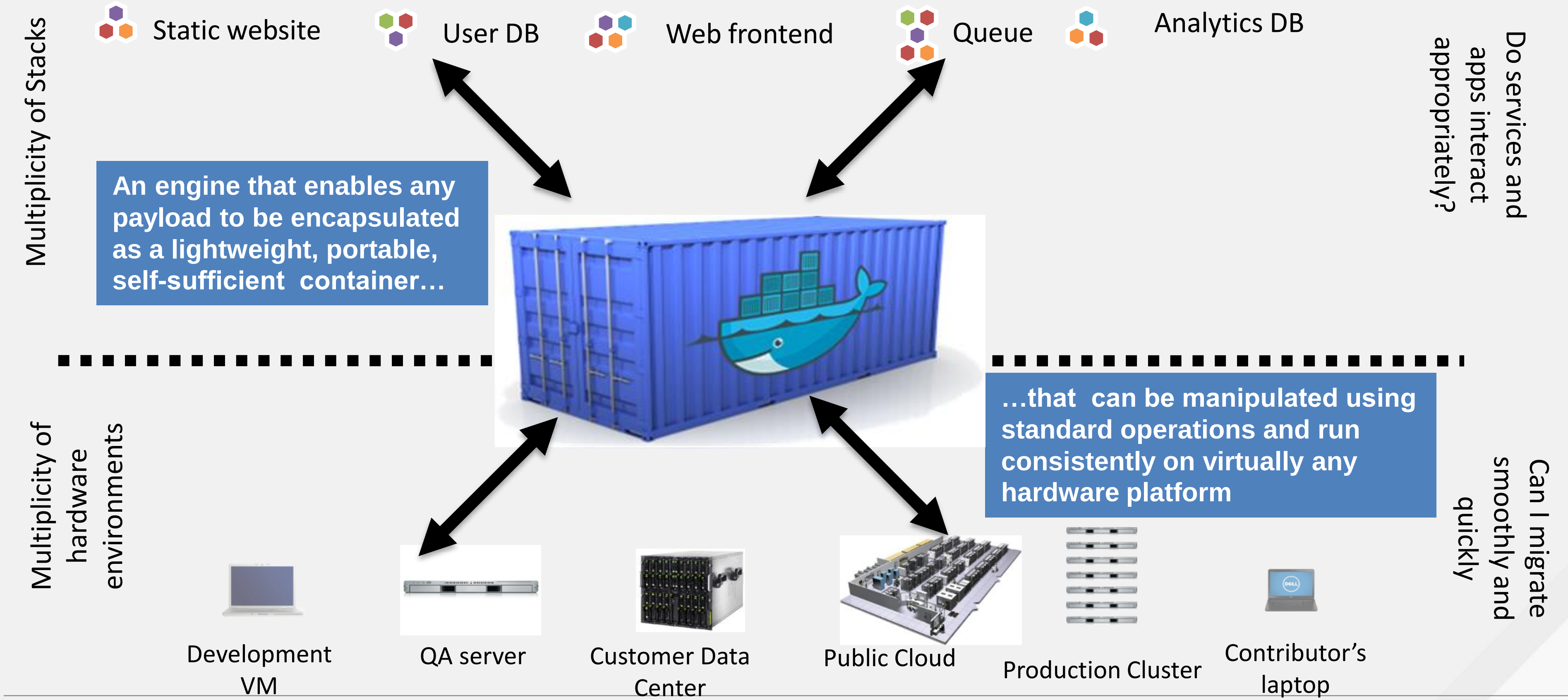
Do I worry about how goods interact (e.g. coffee beans next to spices)

Multiplicity of methods for transporting/storing

…in between, can be loaded and unloaded, stacked, transported efficiently over long distances, and transferred from one mode of transport to another

Can I transport quickly and smoothly (e.g. from boat to train to truck)

Docker is a shipping container system for code



Multiplicity of Stacks

Static website

User DB

Web frontend

Queue

Analytics DB

Do services and apps interact appropriately?

An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…

…that can be manipulated using standard operations and run consistently on virtually any hardware platform

Multiplicity of hardware environments

Development VM

QA server

Customer Data Center

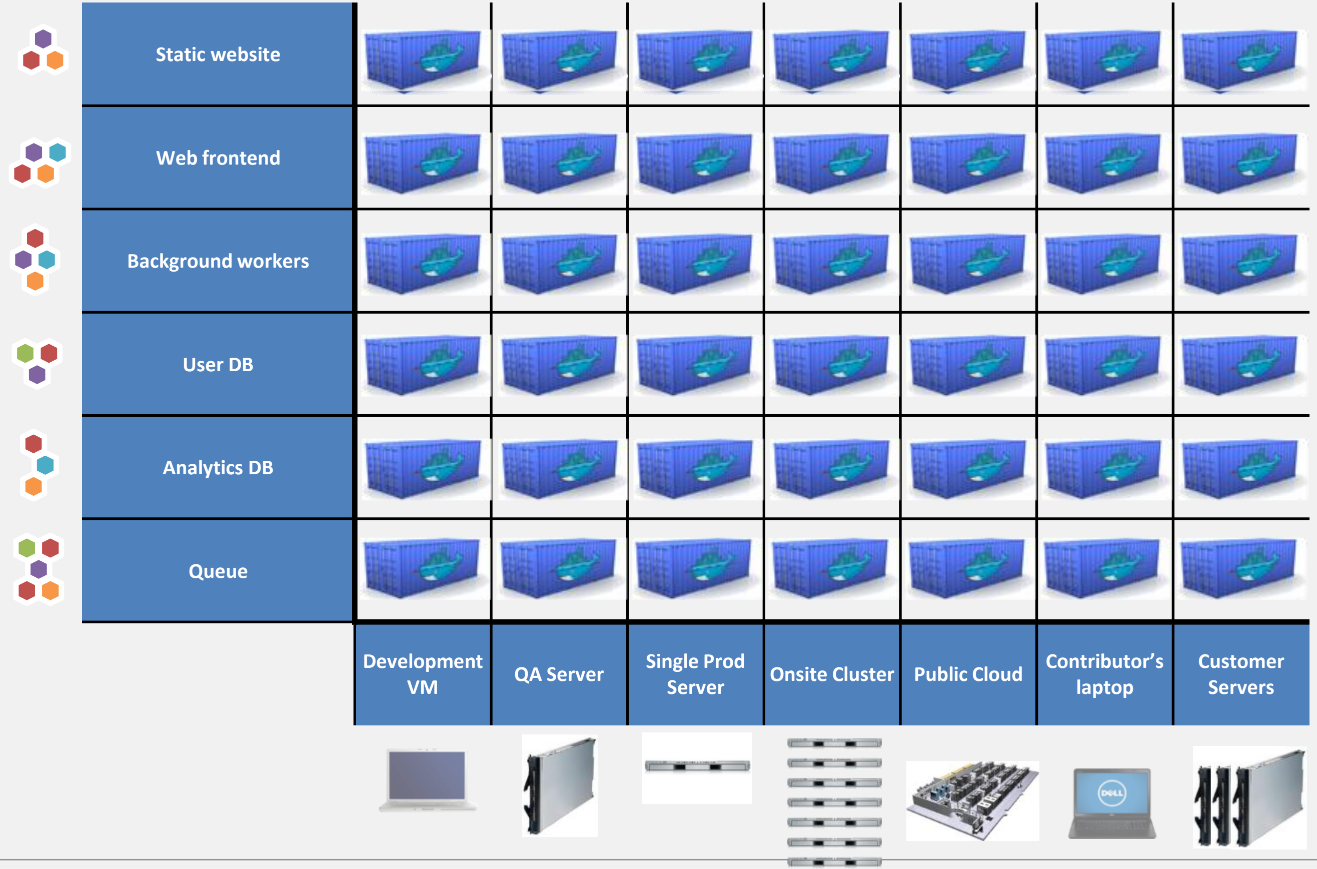
Public Cloud

Production Cluster

Contributor's laptop

Can I migrate smoothly and quickly

Docker eliminates the matrix from Hell



	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							

Why Developers Care

Build once…(finally) 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

* With the 0.7 release, we support any x86 server running a modern Linux kernel (3.2+ generally. 2.6.32+ for RHEL 6.5+, Fedora, & related)

Why Devops Cares?

Configure once…run anything

- Make the entire lifecycle more efficient, consistent, and repeatable
- Increase the quality of code produced by developers.
- Eliminate inconsistencies between development, test, production, and customer environments
- Support segregation of duties
- Significantly improves the speed and reliability of continuous deployment and continuous integration systems
- Because the containers are so lightweight, address significant performance, costs, deployment, and portability issues normally associated with VMs

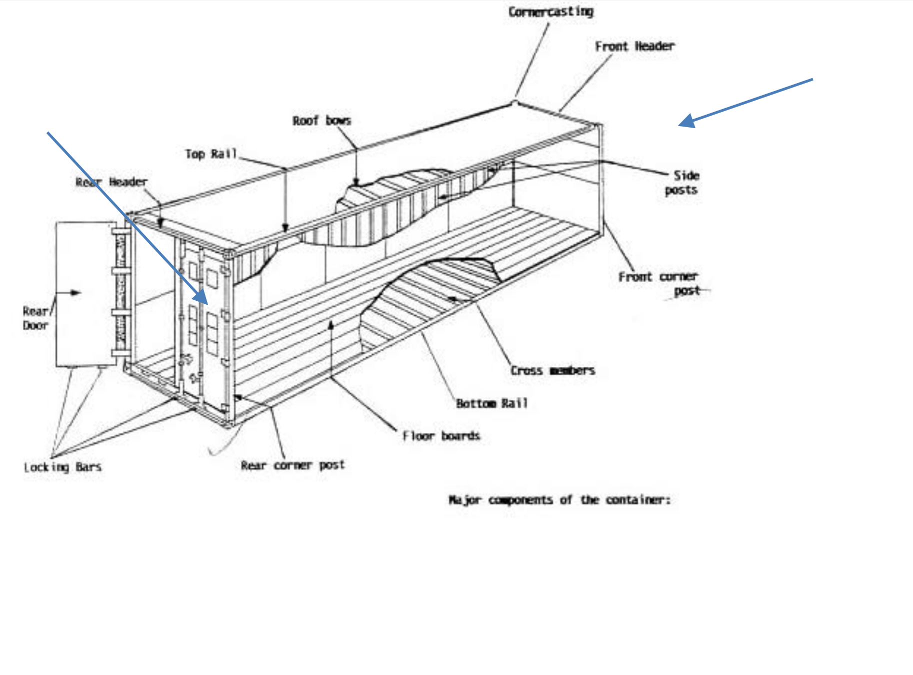
Why it works—separation of concerns

Prashant the Developer

Worries about what's "inside" the container

- His code
- His Libraries
- His Package Manager
- His Apps
- His Data

All Linux servers look the same



Amit the Ops Guy

Worries about what's "outside" the container

- Logging
- Remote access
- Monitoring
- Network config

All containers start, stop, copy, attach, migrate, etc. the same way

More technical explanation

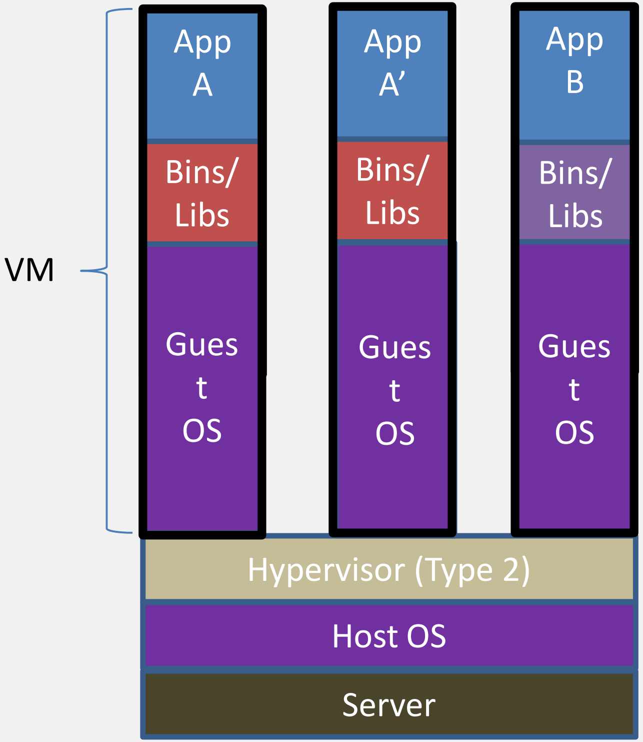
WHY

- Run everywhere
  - Regardless of kernel version (2.6.32+)
  - Regardless of host distro
  - Physical or virtual, cloud or not
  - Container and host architecture must match*
- Run anything
  - If it can run on the host, it can run in the container
  - i.e. if it can run on a Linux kernel, it can run

WHAT

- High Level—It’s a lightweight VM
  - Own process space
  - Own network interface
  - Can run stuff as root
  - Can have its own /sbin/init (different from host)
  - <<machine container>>
- Low Level—It’s chroot on steroids
  - Can also *not* have its own /sbin/init
  - Container=isolated processes
  - Share kernel with host
  - No device emulation (neither HVM nor PV) from host)
  - <<application container>>

Containers vs. VMs



VM

App
A

Bins/
Libs

Gues
t
OS

App
A’

Bins/
Libs

Gues
t
OS

App
B

Bins/
Libs

Gues
t
OS

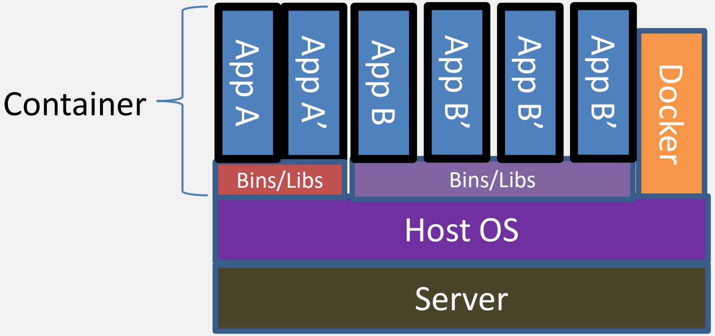
Hypervisor (Type 2)

Host OS

Server

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

…result is significantly faster deployment, much less overhead, easier migration, faster restart



Container

App A

App A’

App B

App B’

App B’

App B’

Docker

Bins/Libs

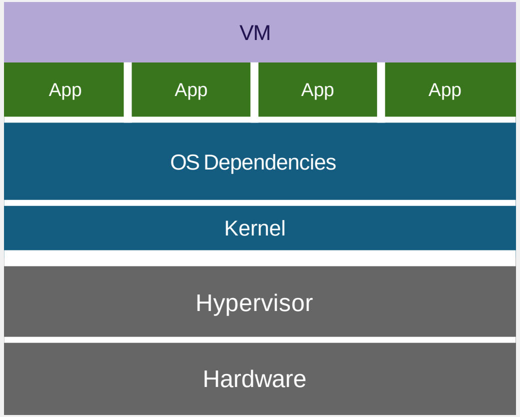
Bins/Libs

Host OS

Server

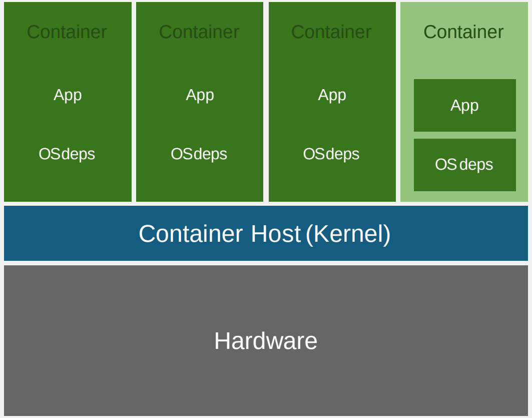
VIRTUAL MACHINES AND CONTAINERS

VIRTUAL MACHINES



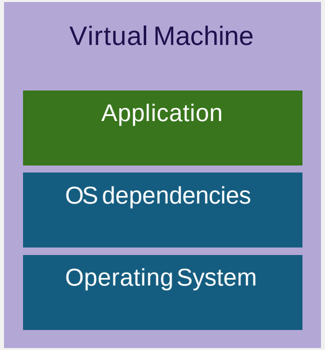
Virtual machines areisolated
apps are not

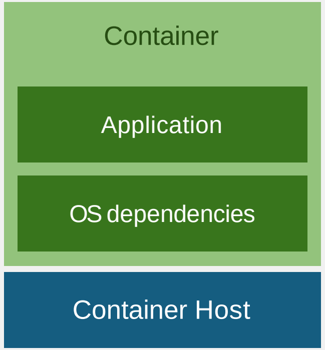
CONTAINERS



containers are isolated
so are the apps

VIRTUAL MACHINES AND CONTAINERS

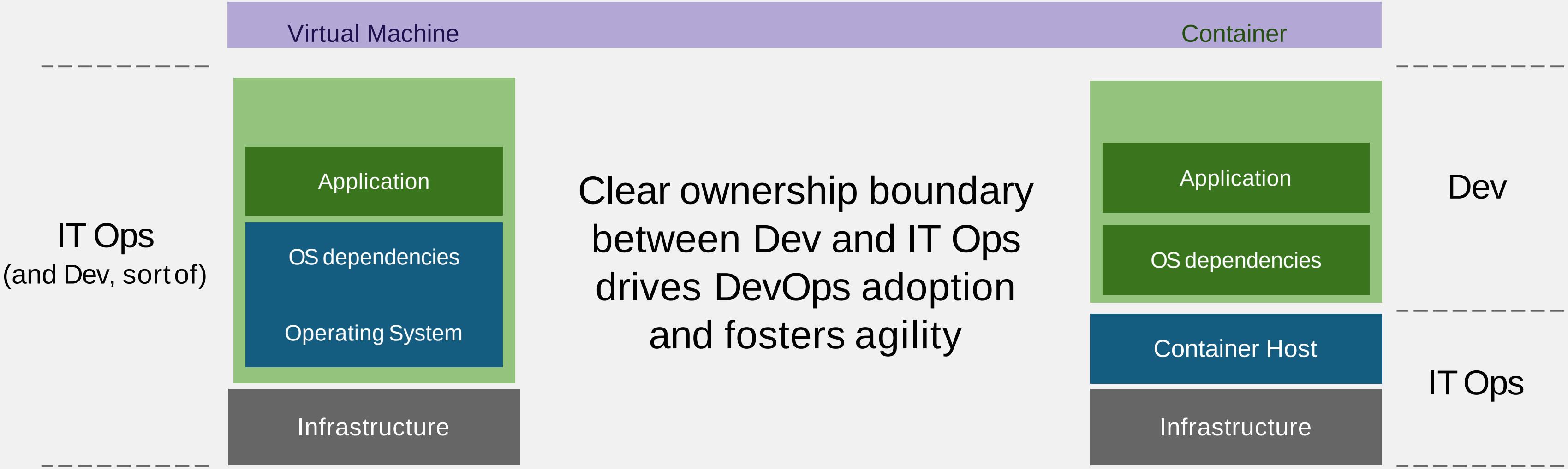




- + VM Isolation
- − Complete OS
- − Static Compute
- − Static Memory
- − High Resource Usage

- + Container Isolation
- + Shared Kernel
- + Burstable Compute
- + Burstable Memory
- + Low Resource Usage

VIRTUAL MACHINES AND CONTAINERS



Virtual Machine

IT Ops
(and Dev, sort of)

Application

OS dependencies

Operating System

Infrastructure

Container

Dev

Application

OS dependencies

Container Host

IT Ops

Infrastructure

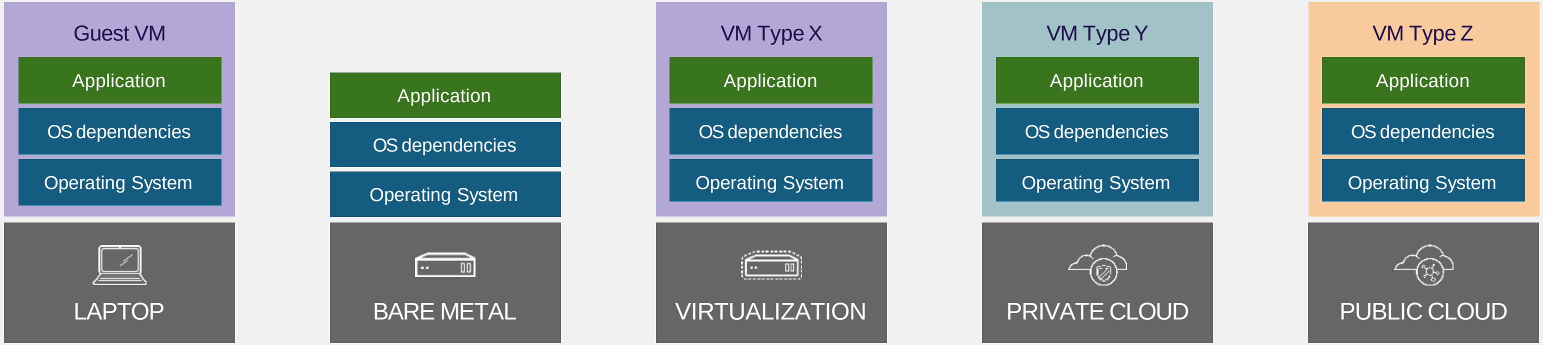
Clear ownership boundary between Dev and IT Ops drives DevOps adoption and fosters agility

🟦 Optimized for stability

🟩 Optimized for agility

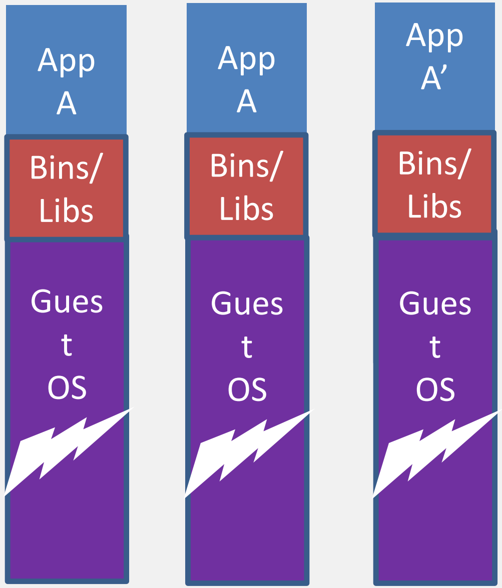
APPLICATION PORTABILITY WITH VM

Virtual machines are NOT portable across hypervisor and do NOT provide portable packaging for applications



Why are Docker containers lightweight?

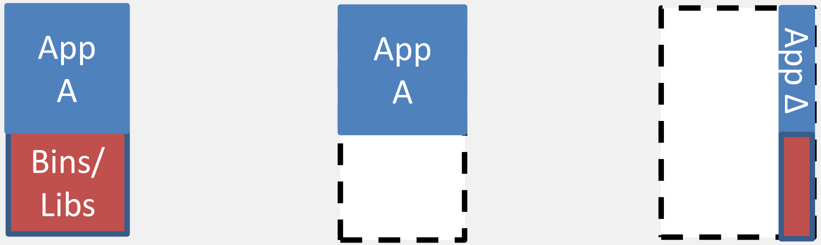
VMs



VMs

Every app, every copy of an
app, and every slight modification
of the app requires a new virtual server

Containers



Original App

(No OS to take
up space, resources,
or require restart)

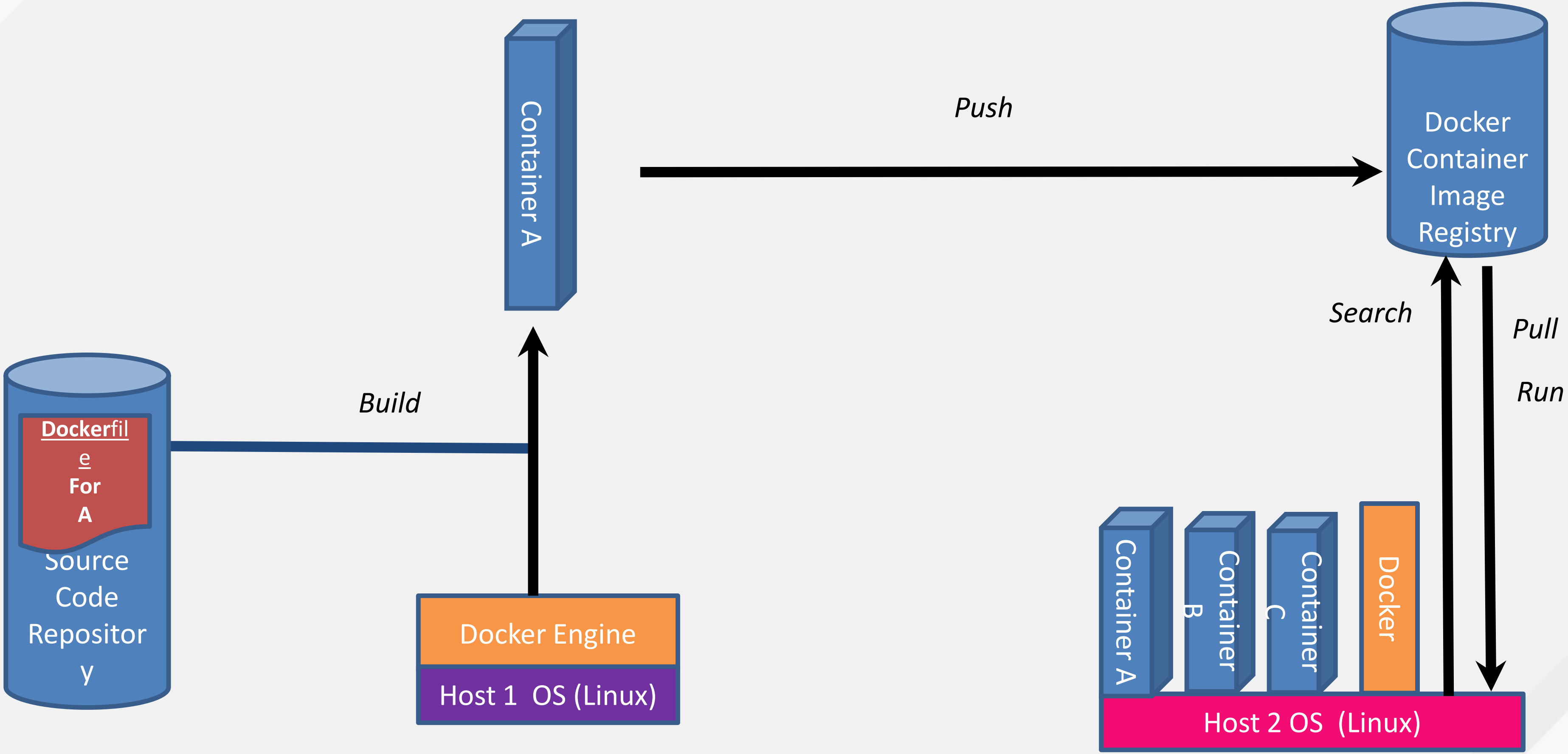
Copy of App

No OS. Can
Share bins/libs

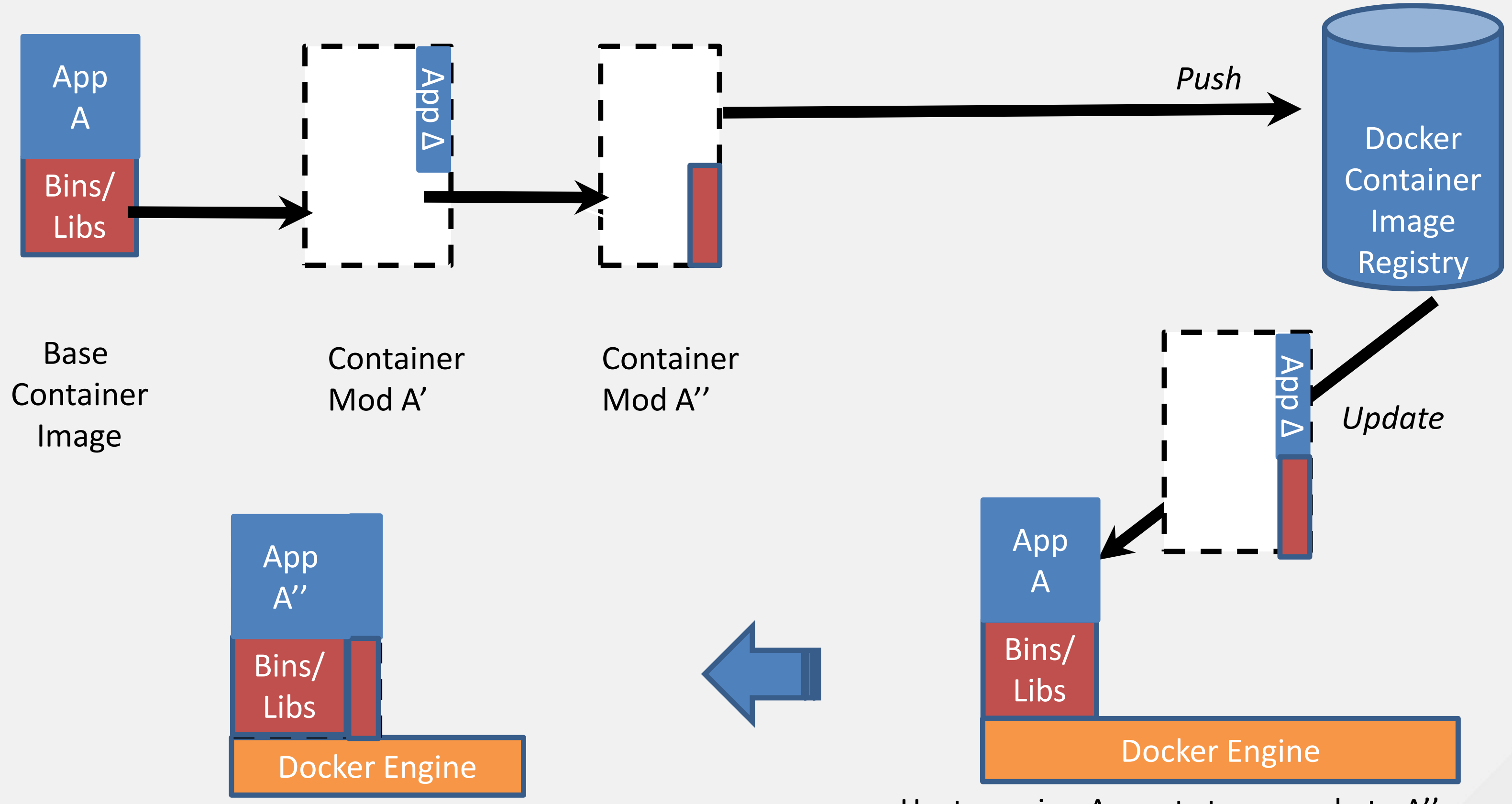
Modified App

Copy on write
capabilities allow
us to only save the diffs
Between container A
and container
A’

What are the basics of the Docker system?



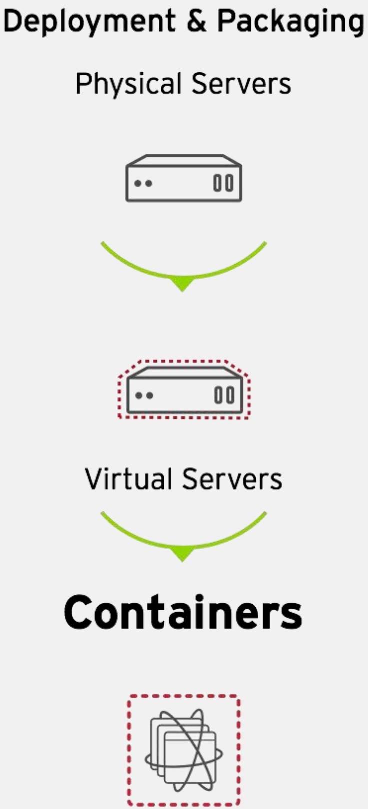
Changes and Updates



Host is now running A’’

Host running A wants to upgrade to A’’.
Requests update. Gets only diffs

Containers - An Evolution in Application Deployment



Deployment & Packaging

Physical Servers

Virtual Servers

Containers

- Enable efficiency and automation for microservices, but also support traditional applications
- Enable faster and more consistent deployments from Development to Production
- Enable application portability across 4 infrastructure footprints: Physical, Virtual, Private & Public Cloud

Install Docker

Install Docker

Installing Docker

Docker is easy to install.

It runs on:

- ❑ A variety of Linux distributions.
- ❑ OS X via a virtual machine.
- ❑ Microsoft Windows via a virtual machine.

Installing Docker on Linux

It can be installed via:

- Distribution supplied packages on virtually all distros.
- (Includes at least: Arch Linux, CentOS, Debian, Fedora, Gentoo, openSUSE, RHEL, Ubuntu.)
- Packages supplied by Docker.
- Installation script from Docker.
- Binary download from Docker (it's a single file).

Installing Docker on your Linux distribution

On Red Hat and derivatives.

❑$ sudo yum install docker

On Debian and derivatives.

❑$ sudo apt-get install docker.io

Installation script from Docker

You can use the curl command to install on several platforms.

❑$ curl -s https://get.docker.io/ubuntu/ | sudo sh

This currently works on:
Ubuntu;
Debian;
Fedora;
Gentoo.

Installing on OS X and Microsoft Windows

Docker doesn't run natively on OS X and Microsoft Windows.
To install Docker on these platforms we run a small virtual machine using a tool called Boot2Docker.

Docker architecture

Docker is a client-server application.

The Docker daemon

The Docker server.

Receives and processes incoming Docker API requests.

The Docker client

Command line tool - the docker binary.

Talks to the Docker daemon via the Docker API.

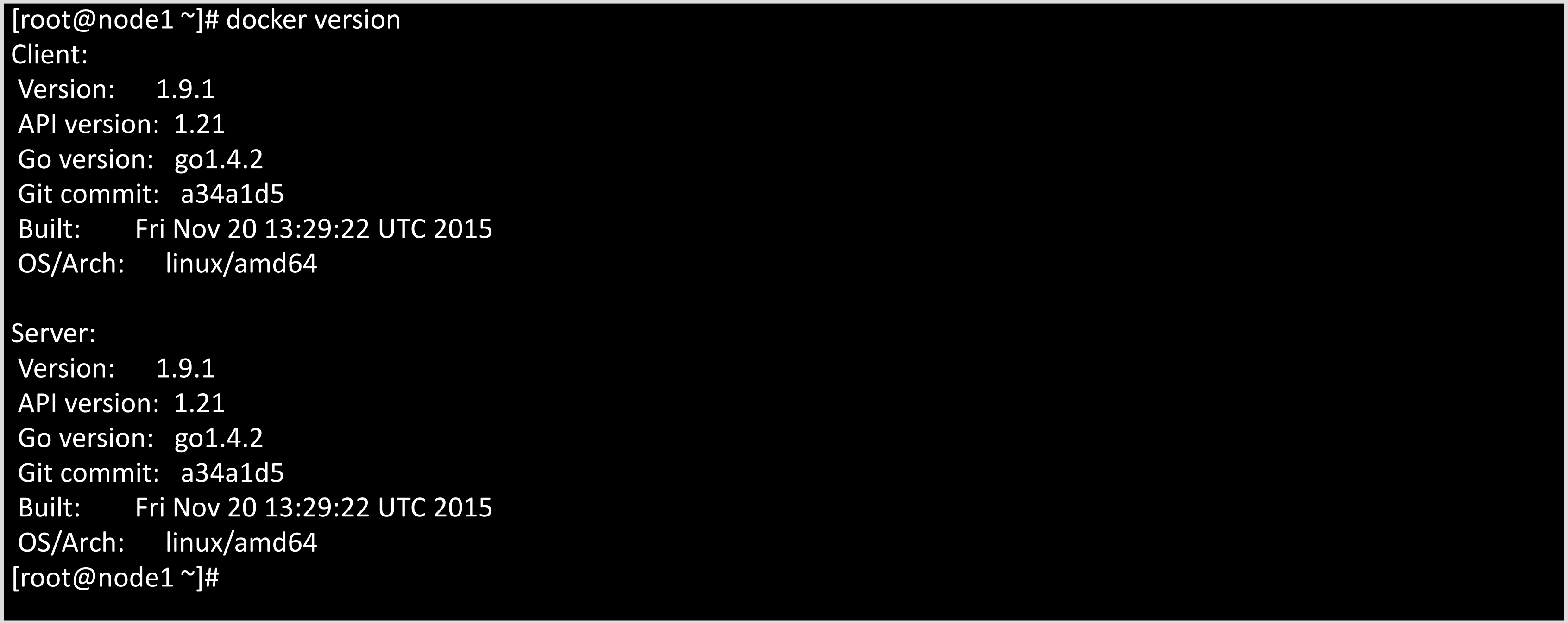
Docker Hub Registry

Public image registry.

The Docker daemon talks to it via the registry API.

Test Docker is working

Using the docker **client:**



```
[root@node1 ~]# docker version
Client:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:29:22 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:29:22 UTC 2015
 OS/Arch:      linux/amd64
[root@node1 ~]#
```

The docker group

Warning!
The **docker** user is **root** equivalent.
It provides **root** level access to the host.
You should restrict access to it like you would protect **root**.

Add the Docker group

```
$ sudo groupadd docker
```

Add ourselves to the group

```
$ sudo gpasswd -a $USER docker
```

Restart the Docker daemon

```
$ sudo systemctl restart docker.service
```

Hello World again without sudo

Hello World again without sudo

```
[amit@node1 ~]$ docker run ubuntu echo hello world
hello world
[amit@node1 ~]$
```

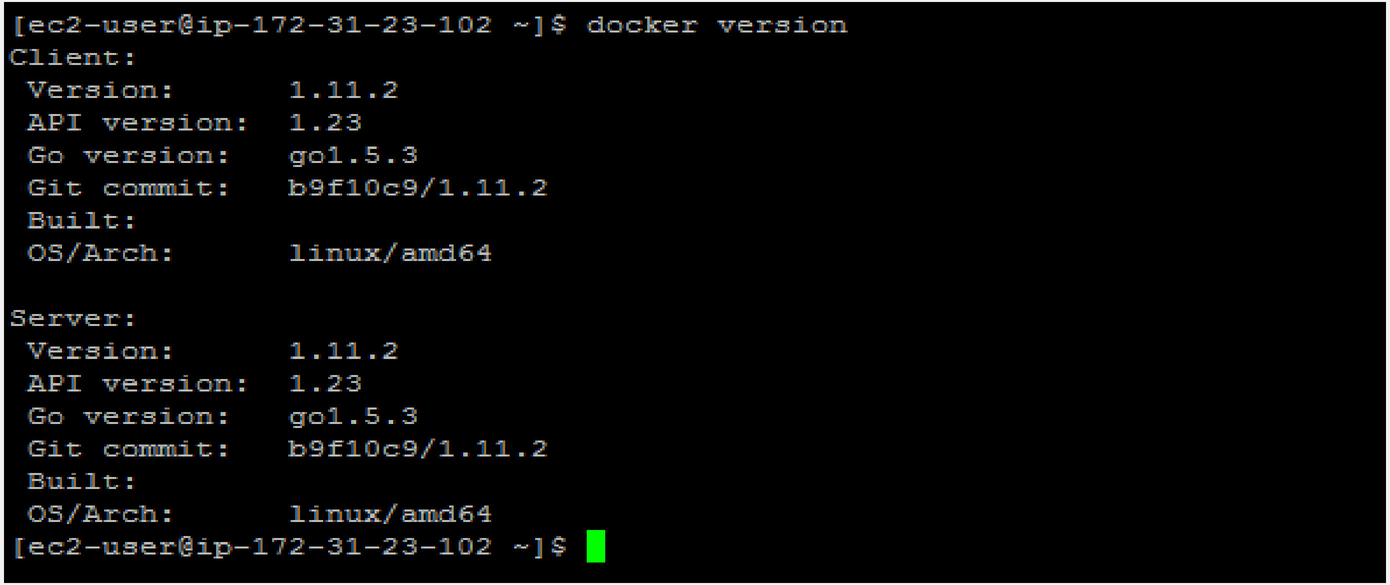
Section summary

We've learned how to:

Install Docker.

Run Docker without **sudo**.

Demo.

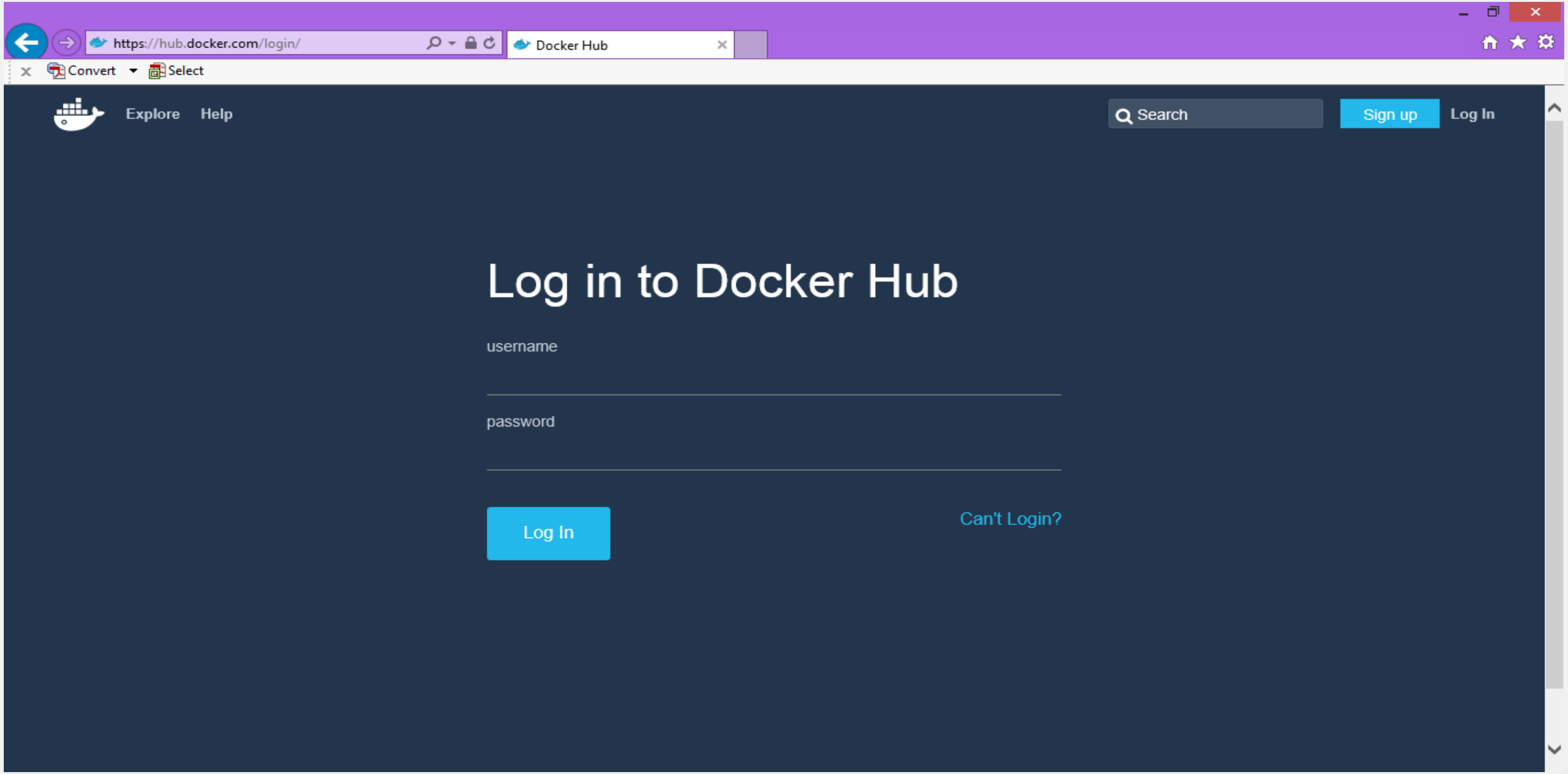


```
[ec2-user@ip-172-31-23-102 ~]$ docker version
Client:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.3
 Git commit:   b9f10c9/1.11.2
 Built:
 OS/Arch:      linux/amd64

Server:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.3
 Git commit:   b9f10c9/1.11.2
 Built:
 OS/Arch:      linux/amd64
[ec2-user@ip-172-31-23-102 ~]$ 
```

Introducing Docker Hub

Introducing Docker Hub

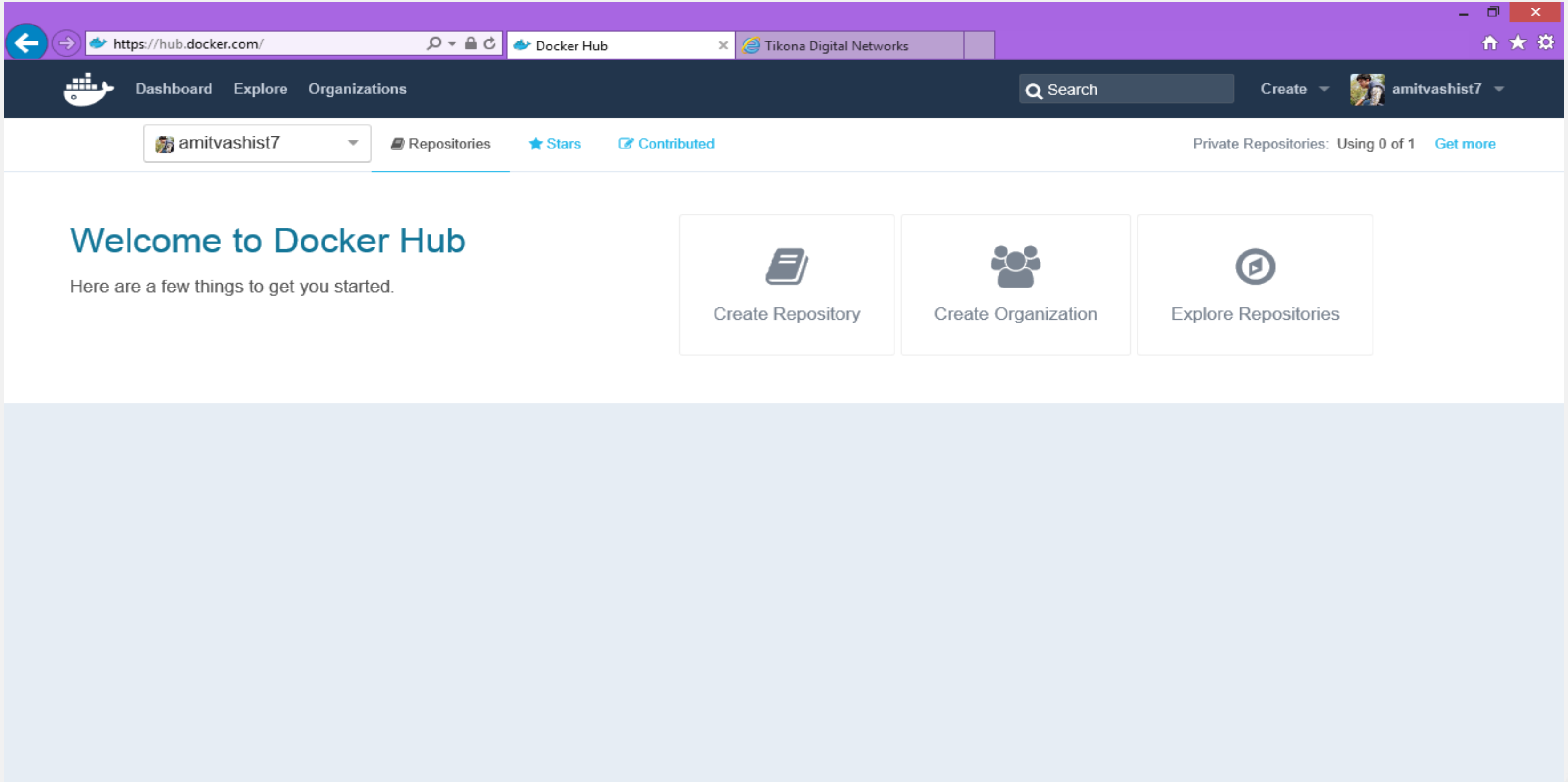


Log in to Docker Hub

username

password

Introducing Docker Hub



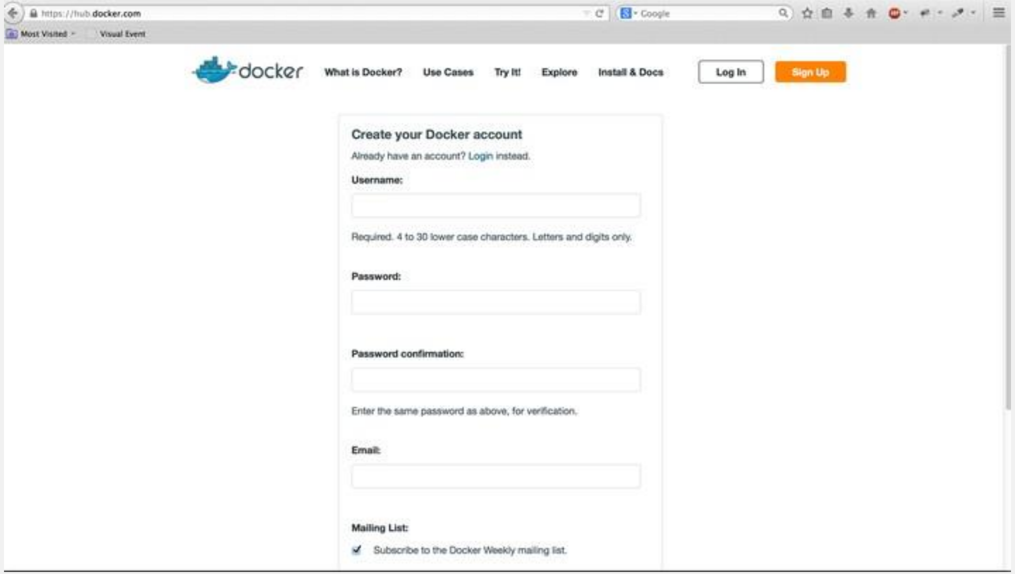
Welcome to Docker Hub

Here are a few things to get you started.

Introducing Docker Hub

Sign up for a Docker Hub account

Having a Docker Hub account will allow us to store our images in the registry.
To sign up, you'll go to hub.docker.com and fill out the form.
Note: if you have an existing Index/Hub account, this step is not needed.



Create your Docker account

Already have an account? Login instead.

Username:

Required. 4 to 30 lower case characters. Letters and digits only.

Password:

Password confirmation:

Enter the same password as above, for verification.

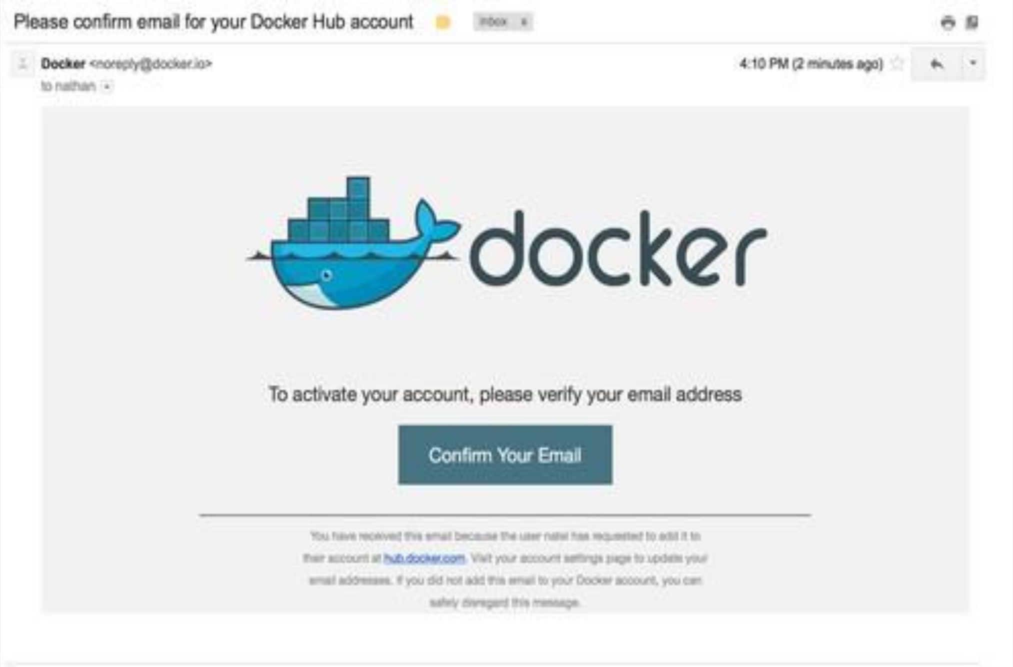
Email:

Mailing List:

☑ Subscribe to the Docker Weekly mailing list.

Introducing Docker Hub

Activate your account through e-mail.
Check your e-mail and click the confirmation link.



Please confirm email for your Docker Hub account

Docker <noreply@docker.io>
to nathan

4:10 PM (2 minutes ago)

To activate your account, please verify your email address

You have received this email because the user natel has requested to add it to their account at hub.docker.com. Visit your account settings page to update your email addresses. If you did not add this email to your Docker account, you can safely disregard this message.

Introducing Docker Hub

Let's use our new account to login to the Docker Hub!

```
$ docker login
Username: my_docker_hub_login Password:
Email: my@email.com Login Succeeded
```

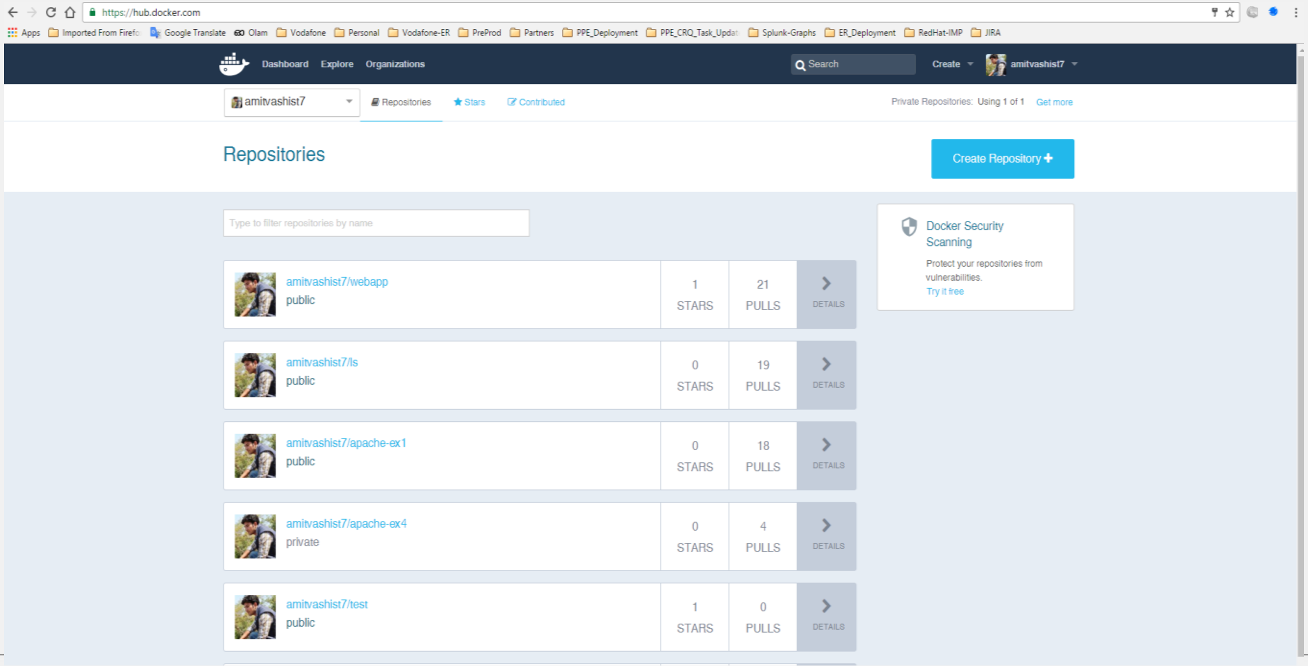
You should protect this file!

```
[amit@node1 ~]$ docker login
Username: amitvashist7
Password:
Email: amitvashist7@gmail.com
WARNING: login credentials saved in
/home/amit/.docker/config.json
Login Succeeded
[amit@node1 ~]$

[amit@node1 ~]$ ls -ltr .docker/config.json
-rw-------. 1 amit amit 137 Dec 23 01:25 .docker/config.json
[amit@node1 ~]$
```

Section summary

We've learned how to:
Register for an account on Docker Hub.
Login to your account from the command line.



Private Repositories: Using 1 of 1

Repositories

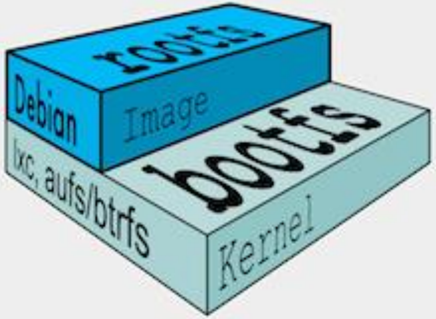
amitvashist7/webapp public	1 STARS	21 PULLS
amitvashist7/ls public	0 STARS	19 PULLS
amitvashist7/apache-ex1 public	0 STARS	18 PULLS
amitvashist7/apache-ex4 private	0 STARS	4 PULLS
amitvashist7/test public	1 STARS	0 PULLS

Docker Security Scanning

Protect your repositories from vulnerabilities.

Docker Image

Getting started with Images



Getting started with Images

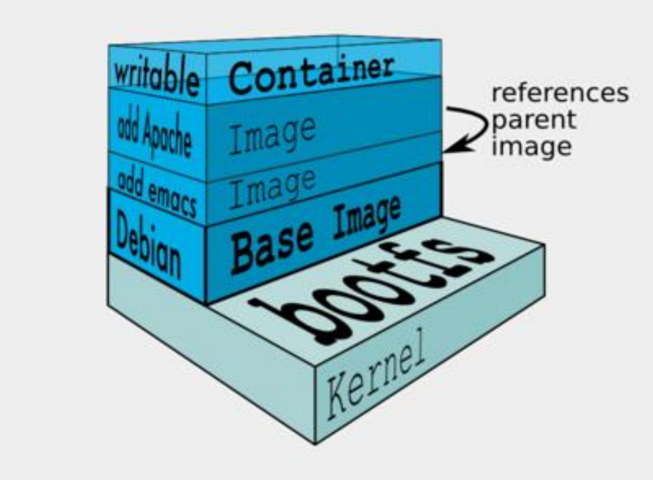
Images

What are they?

An image is a collection of files.

Base images (ubuntu, busybox, fedora etc.) are what you build your own custom images on top of.

Images are layered, and each layer represents a diff (what changed) from the previous layer. For instance, you could add **emacs & apache** on top of a base image.



Getting started with Images

So what's the difference between Containers and Images?

Containers represent an encapsulated set of **processes** based on an image.

You spawn them with the **docker run** command.

In our Initial example, you created a shiny new container by executing **docker run**. It was based on the **ubuntu** image, and we ran the **echo** command.

Images are like **templates** or **stencils** that you can create containers from.

Getting started with Images

How do you store and manage images?
Images can be stored:
On your Docker host.
In a Docker registry.

You can use the Docker client to manage images.

Getting started with Images

Images belong to a namespace

There are three namespaces:

Root-like

```
ubuntu
```

User

```
amitvashist7/apache-ex1
```

Self-Hosted

```
registry.example.com:5000/my-private-image
```

Getting started with Images

Root namespace

The root namespace is for official images. They are put there by Docker Inc., but they are generally authored and maintained by third parties.

Those images include some barebones **distro** images, for instance:
ubuntu
fedora
centos

Those are ready to be used as bases for your own images.

Getting started with Images

Downloading images

In order to download Image:

The busybox image, implicitly, when we did docker run busybox.

The ubuntu image, explicitly, when we did docker pull ubuntu.

Download a user image.

```
$ docker pull amitvashist7/apache-ex1

Pulling repository amitvashist7/apache-ex1
8144a5b2bc0c:  Download complete
511136ea3c5a:  Download complete
8abc22fbb042:  Download complete
58394af37342:  Download complete
6ea7713376aa:  Download complete
71ef82f6ed3c:  Download complete
```

Getting started with Images

Image and tags

Images can have tags.

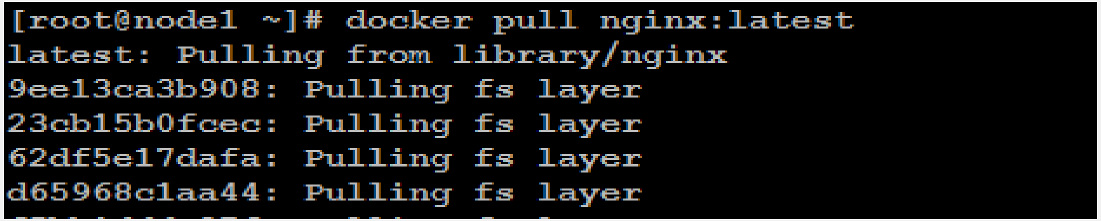
Tags define image variants.

When using images it is always best to be specific

Downloading an image tag

As seen previously, images are made up of layers.

Docker has downloaded all the necessary layers.



```
[root@node1 ~]# docker pull nginx:latest
latest: Pulling from library/nginx
9ee13ca3b908: Pulling fs layer
23cb15b0fcec: Pulling fs layer
62df5e17dafa: Pulling fs layer
d65968c1aa44: Pulling fs layer
```

Section summary

We've learned how to:

Understand images and image tags.

Search for images.

Download an image.

Understand Docker image name spacing.

Docker Container

A container to call your own

A container to call your own

Dockerizing an application is the process of converting an application to run within a Docker container.

Containers
Containers are created with the **docker run** command.
Containers have two modes they run in:

- ❑ **Daemonized.**
- ❑ **Interactive.**

A container to call your own

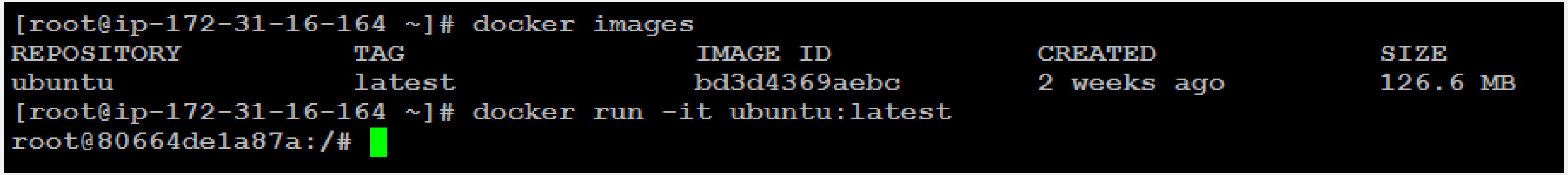
Daemonized containers
Runs in the background.
The docker run command is launched with the -d command line flag.
The container runs until it is stopped or killed.

Interactive containers
Runs in the foreground.
Attached a pseudo-terminal, i.e. let you get input and output from the container.
The container also runs until its controlling process stops or it is stopped or killed.

A container to call your own

Launching a container

Let's create a new container from the ubuntu image:



```
[root@ip-172-31-16-164 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              bd3d4369aebc        2 weeks ago         126.6 MB
[root@ip-172-31-16-164 ~]# docker run -it ubuntu:latest
root@80664de1a87a:/#
```

The **-i** flag sets Docker's mode to interactive.

The **-t** flag creates a pseudo terminal (or PTY) in the container.

We've specified the **ubuntu:12.04** image from which to create our container.

We passed a command to run inside the container, /bin/bash.

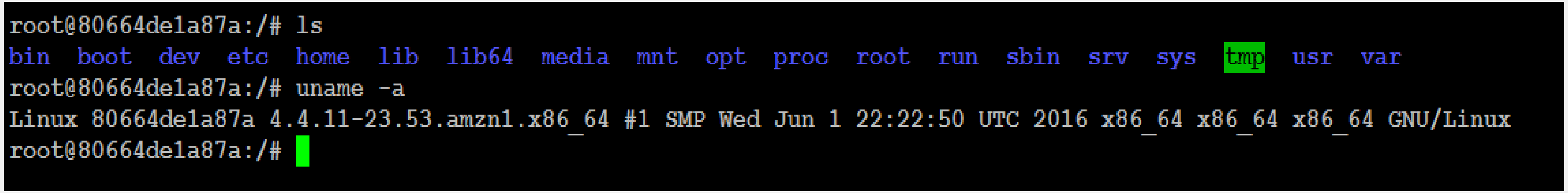
That command has launched a Bash shell inside our container.

The hexadecimal number after root@ is the container's identifier.

> **(The actual ID is longer than that. Docker truncates it for convenience, just like git or hg will show shorter ID instead of full hashes.)**

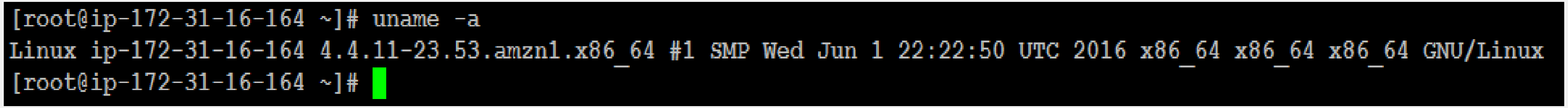
A container to call your own

Let’s run some commands inside our container



```
root@80664de1a87a:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@80664de1a87a:/# uname -a
Linux 80664de1a87a 4.4.11-23.53.amzn1.x86_64 #1 SMP Wed Jun 1 22:22:50 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
root@80664de1a87a:/# 
```

Now let's **exit** the container.
Check the kernel version and hostname again, outside the container:



```
[root@ip-172-31-16-164 ~]# uname -a
Linux ip-172-31-16-164 4.4.11-23.53.amzn1.x86_64 #1 SMP Wed Jun 1 22:22:50 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
[root@ip-172-31-16-164 ~]# 
```

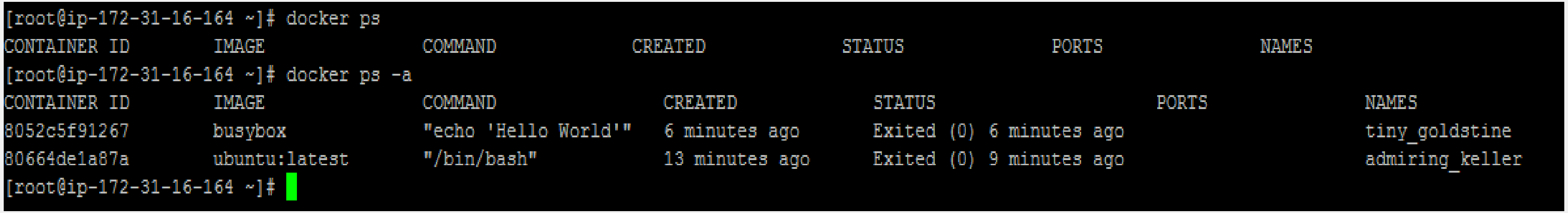
The kernel version is the same. Hostname is different.

A container to call your own

Container status

You can see container status using the **docker ps** command.

We can also use the **docker ps** command with the **-a** flag. The **-a** flag tells Docker to list all containers both running and stopped.



```
[root@ip-172-31-16-164 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
[root@ip-172-31-16-164 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS                     PORTS               NAMES
8052c5f91267        busybox             "echo 'Hello World'"   6 minutes ago       Exited (0) 6 minutes ago                       tiny_goldstine
80664de1a87a        ubuntu:latest       "/bin/bash"            13 minutes ago      Exited (0) 9 minutes ago                       admiring_keller
[root@ip-172-31-16-164 ~]#
```

A container to call your own

Container naming

You can now give memorable names to your containers using the new -**name** flag for docker run.

If no name is specified Docker will automatically generate a name.

docker run -itd --name job1 ubuntu /bin/bash

```
[root@ip-172-31-16-164 ~]# docker run -itd --name job1 ubuntu /bin/bash
17de35728b8dd96cfa40dcb0b822fe0eaf11e8002fbb5224227b6b43fd9cb334
[root@ip-172-31-16-164 ~]#


[root@ip-172-31-16-164 ~]# docker ps -l
CONTAINER ID      IMAGE            COMMAND          CREATED           STATUS            PORTS            NAMES
17de35728b8d       ubuntu           "/bin/bash"       14 seconds ago     Up 14 seconds                       job1
[root@ip-172-31-16-164 ~]#
```

A container to call your own

What does docker ps tell us?

We can see a lot of data returned by the docker ps command.

```
[root@ip-172-31-16-164 ~]# docker ps -l
CONTAINER ID     IMAGE          COMMAND         CREATED          STATUS          PORTS          NAMES
17de35728b8d      ubuntu         "/bin/bash"      14 seconds ago    Up 14 seconds                    job1
[root@ip-172-31-16-164 ~]#
```

Let's focus on some items:

CONTAINER ID is a unique identifier generated by Docker for our container. You can use it to manage the container.

IMAGE is the image used to create that container.

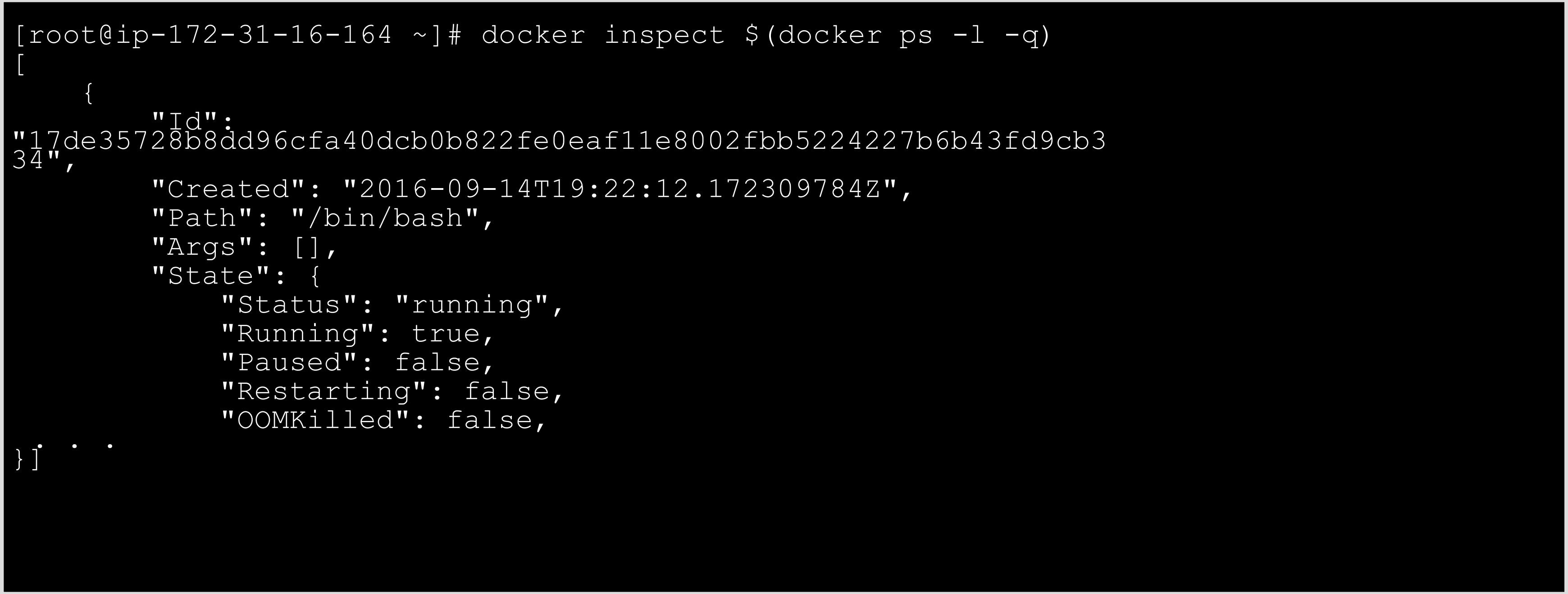
COMMAND is the exact command that we asked Docker to run: /bin/bash.

You can name your containers (with the --name option).If you don't, Docker will generate a random name for you, like **job1**. That name shows up in the **NAMES** column.

A container to call your own

Inspecting our container

You can also get a lot more information about our container by using the **docker inspect** command.

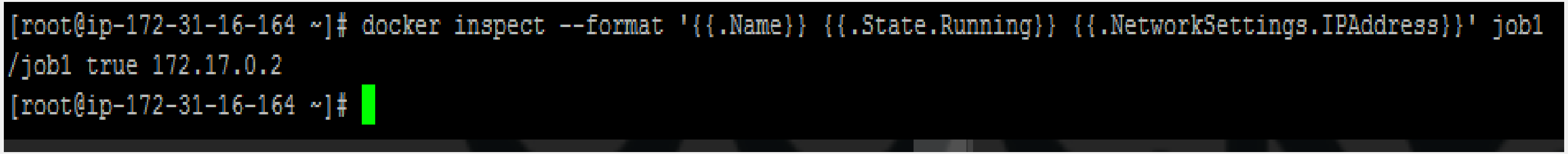


```
[root@ip-172-31-16-164 ~]# docker inspect $(docker ps -l -q)
[
    {
        "Id":
"17de35728b8dd96cfa40dcb0b822fe0eaf11e8002fbb5224227b6b43fd9cb3
34",
        "Created": "2016-09-14T19:22:12.172309784Z",
        "Path": "/bin/bash",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
  . . .
}]
```

A container to call your own

Inspecting something specific

We can also use the **docker inspect** command to find specific things about our container, for example:
docker inspect --format '{{.Name}} {{.State.Running}} {{.NetworkSettings.IPAddress}}' job1



```
[root@ip-172-31-16-164 ~]# docker inspect --format '{{.Name}} {{.State.Running}} {{.NetworkSettings.IPAddress}}' job1
/job1 true 172.17.0.2
[root@ip-172-31-16-164 ~]#
```

Here we've used the **--format** flag and specified a single value from our inspect hash result. This will return its value, in this case a Boolean value for the container's status.

A container to call your own

Restarting our container
You can (re-)start a stopped container using its ID.

```
$ docker start <yourContainerID>
```

Or using its name.

```
$ docker start  job1
```

The container will be restarted using the same options you launched it with.

Section summary

We've learned how to:

Understand the different types of containers.

Start a container.

See a container's status.

Inspect a container.

(Re)Start and attach to a container.

Working with Docker Images

Docker Image

Objectives

At the end of this lesson, you will be able to:

Understand the instructions for a Dockerfile.

Create your own Dockerfiles.

Build an image from a Dockerfile.

Pull and push images to the Docker Hub.

Docker Image

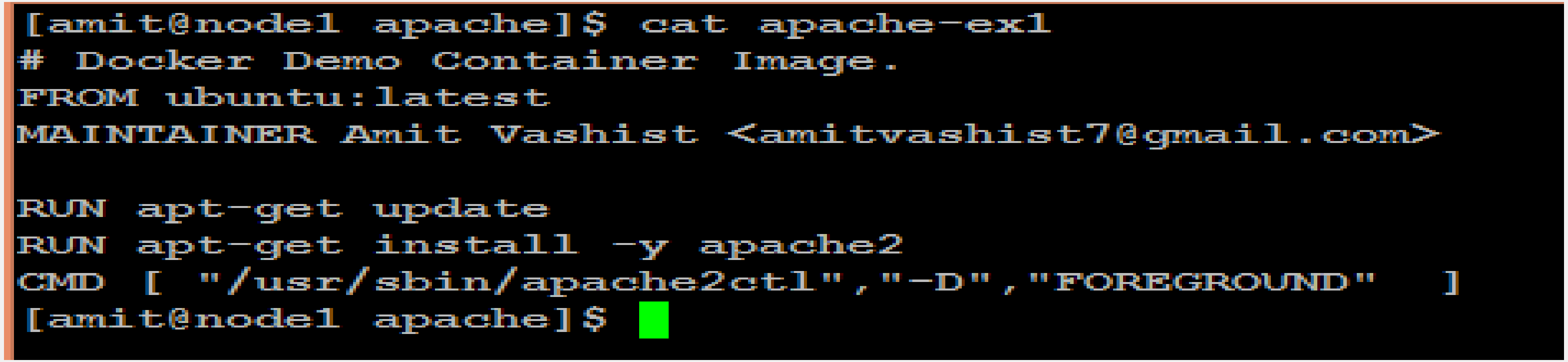
Docker Image

Let's see how to build our own images using:

A **Dockerfile** which holds Docker image definitions. You can think of it as the "build recipe or manifest" for a Docker image. It contains a series of instructions telling Docker how an image is   constructed.
The **docker build** command which builds an image from a **Dockerfile**.

Docker Image

Our first Dockerfile



```
[amit@node1 apache]$ cat apache-ex1
# Docker Demo Container Image.
FROM ubuntu:latest
MAINTAINER Amit Vashist <amitvashist7@gmail.com>

RUN apt-get update
RUN apt-get install -y apache2
CMD [ "/usr/sbin/apache2ctl","-D","FOREGROUND"   ]
[amit@node1 apache]$
```

FROM specifies a source image for our new image. It's mandatory.
MAINTAINER tells us who maintains this image.
Each **RUN** instruction executes a command to build our image.
CMD defines the default command to run when a container is launched from this image.
EXPOSE lists the network ports to open when a container is launched from this image.

Docker Image

Building our Dockerfile

We use the **docker build** command to build images.

```
$ docker build -t="amitvashist7/apache-ex1"  -f apache-ex1 .
```

The -t flag tags an image.

The . indicates the location of the **Dockerfile** being built.

We can also build from other sources.

```
$ docker build -t web https://hub.docker.com/r/amitvashist7/apache-ex4/
```

Here we've specified a GitHub repository to build.

Docker Image

In the last section we created a new image for our web application. This image would be useful to the whole team but how do we share it? Using the **Docker Hub!**

Pulling images

```
$ docker pull ubuntu:14.04
```

This will connect to the Docker Hub and download the ubuntu:14.04 image to allow us to build containers from it.
We can also do the reverse and push an image to the Docker Hub so that others can use it.

Docker Image

Before pushing a Docker image ...

We push images using the docker push command.

Images are uploaded via HTTP and authenticated.

You can only push images to the user namespace, and with your own username.

This means that you cannot push an image called web. It has to be called my_docker_hub_login/web.

Docker Image

Name your image properly

Here are different ways to ensure that your image has the right name.

Of course, in the examples below, replace my_docker_hub_login with your actual login on the Docker Hub.

If you have previously built the web image, you can re-tag it:

```
$ docker tag web my_docker_hub_login/web
```

Or, you can also rebuild it from scratch:

```
$ docker build -t my_docker_hub_login/web \
  git://github.com/docker-training/
staticweb.git
```

Docker Image

Pushing a Docker image to the Docker Hub

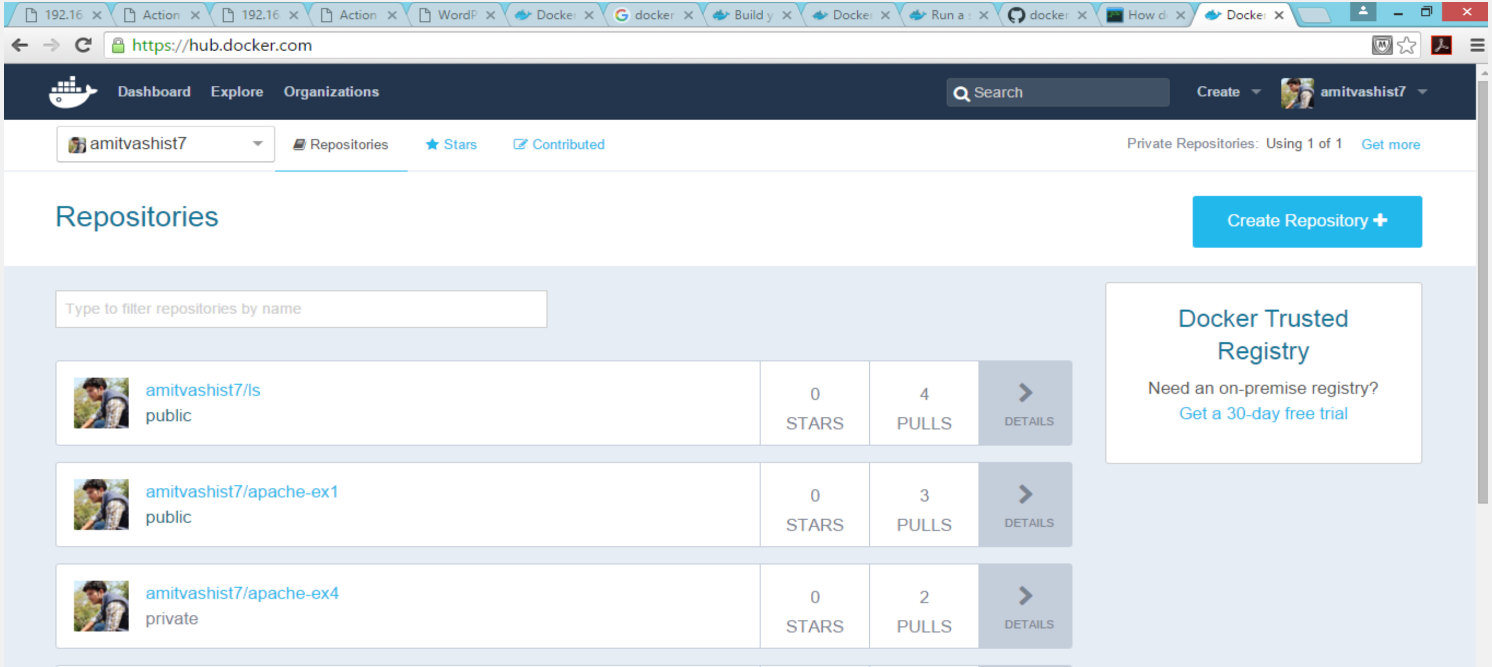
Now the image is named proper, we can push it:

```
[amit@node1 apache]$ docker push amitvashist7/apache-ex1:latest
The push refers to a repository [docker.io/amitvashist7/apache-ex1] (len: 1)
eb6af5eba876: Pushed
dff2fcb00ad3: Pushing [==================>                               ] 5.382 MB/14.35 MB
```

Docker Image

Your account screen

This is the master account screen. Here you can see your repositories and recent activity.



Private Repositories: Using 1 of 1

Repositories

Type to filter repositories by name

amitvashist7/ls
public
0 STARS
4 PULLS

amitvashist7/apache-ex1
public
0 STARS
3 PULLS

amitvashist7/apache-ex4
private
0 STARS
2 PULLS

Docker Trusted Registry

Need an on-premise registry?

Get a 30-day free trial

Section summary

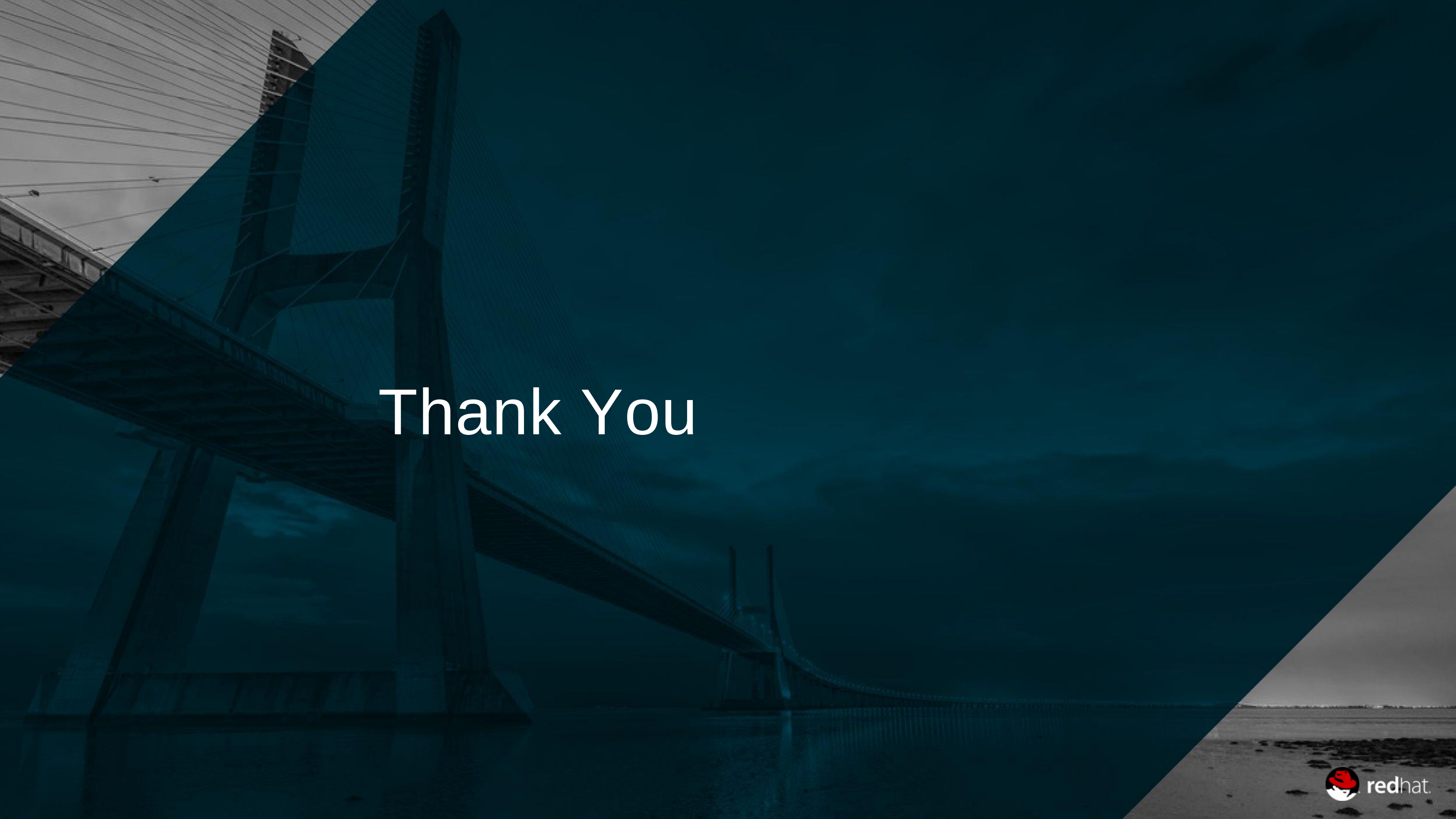
We've learned how to:

Understand the instructions for a Dockerfile.

Create your own Dockerfiles.

Build an image from a Dockerfile.

Pull and push images to the Docker Hub.



Thank You