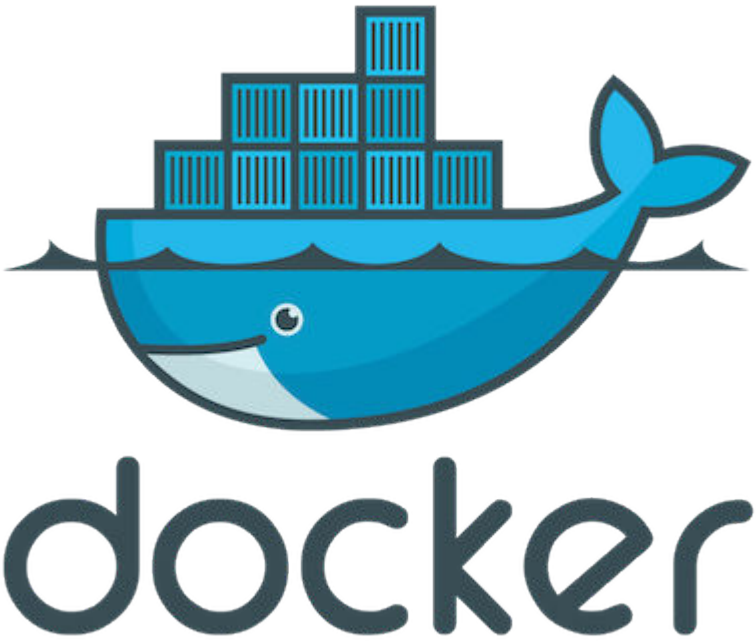
Introduction to Docker



Tutorial Day 1 – Running Galaxy on Docker

Agenda for Today

- Introduction to Containers
- Building a Minimal Host for Docker Containers
- Key commands for using Docker
- Building Docker images and running them as containers
- Shipping Docker Containers
- Running Galaxy on Docker on AWS

Tutorial Overview

You will:

- Learn what Docker is and what it is for.
- Learn the terminology, and define images, containers, etc.
- Learn how to install Docker.
- Use Docker to run containers.
- Use Docker to build containers.
- Use and orchestrate Galaxy on Docker using prebuilt image
- Know what's next: the future of Docker, and how to get help.

Course Agenda

Part 1 - Gaurav

- About Docker
- Using the training virtual machines
- Installing Docker
- Our first containers
- Running containers in the background
- Images and containers
- Building images manually
- Building images automatically
- Basic networking
- Local development workflow

Course Agenda

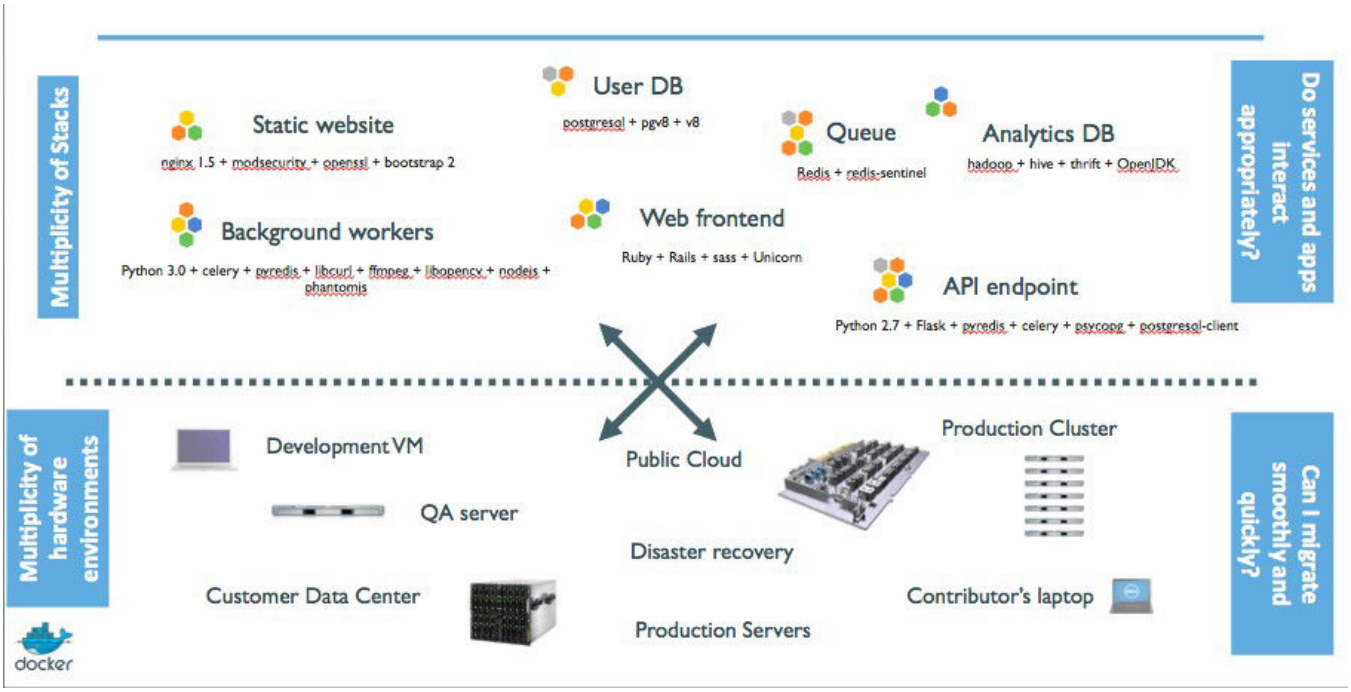
Part 2 - Robert

- Connecting to DockerHub
- Acquire, build and Deploy Galaxy on Docker
- Running a Galaxy pipeline in Docker
- Security

Why Docker?

- The software industry has changed.
- Applications used to be monolithic, with long lifecycles, scaled up.
- Today, applications are decoupled, built iteratively, scaled out.
- As a result, deployment is tough!

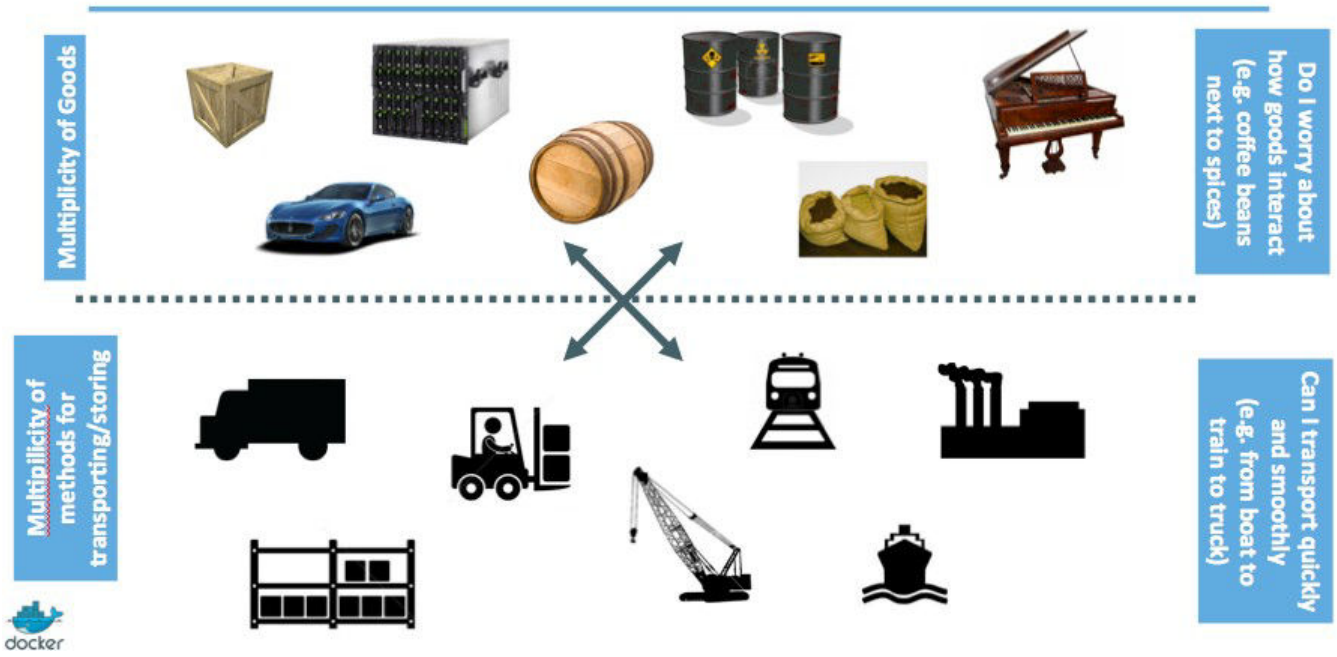
The problem in 2015



The Matrix from Hell

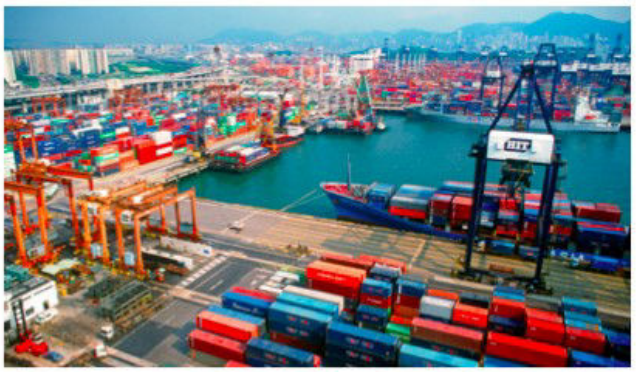
	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

An inspiration and some ancient history!



Intermodal shipping containers

This spawned a Shipping Container Ecosystem!

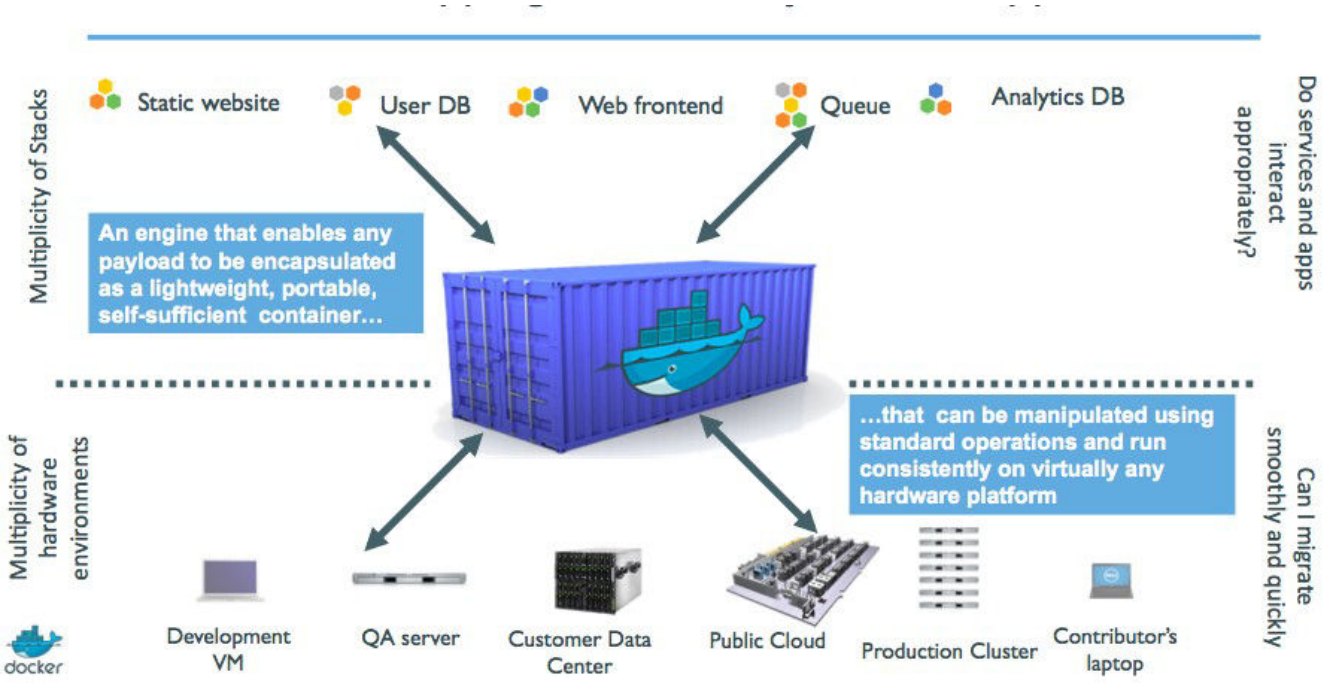


- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

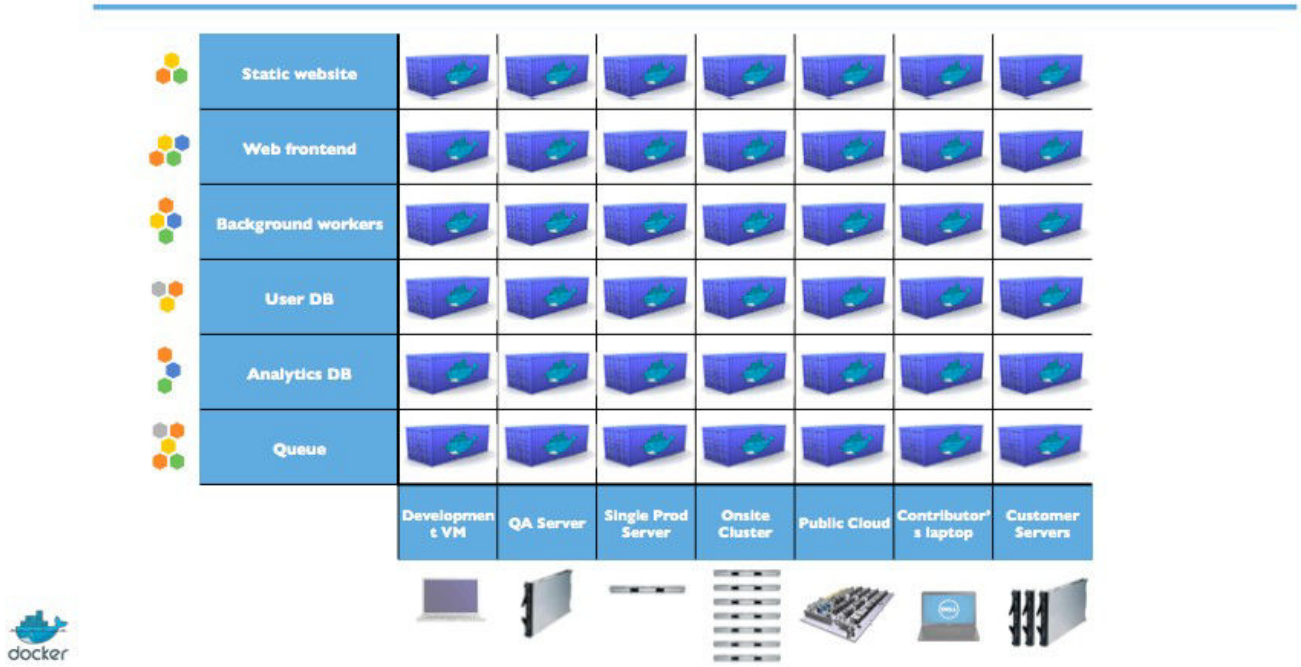
→ massive globalization

- 5000 ships deliver 200M containers per year

A shipping container system for applications



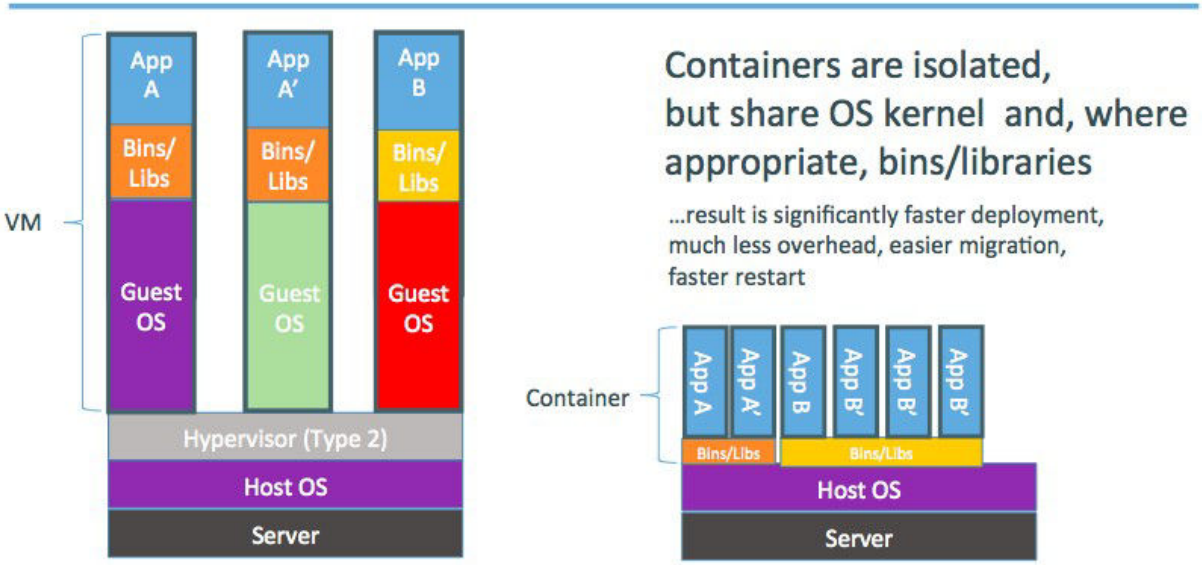
Eliminate the matrix from Hell



Docker high-level roadmap

- Step 1: containers as lightweight VMs
- Step 2: commoditization of containers
- Step 3: shipping containers efficiently
- Step 4: containers in a modern software factory

Step 1: containers as lightweight VMs



- This drew attention from hosting and PAAS industry.
- Highly technical audience with strong ops culture.

Step 2: commoditization of containers

Container technology has been around for a while.
(c.f. LXC, Solaris Zones, BSD Jails, LPAR...)

So what's new?

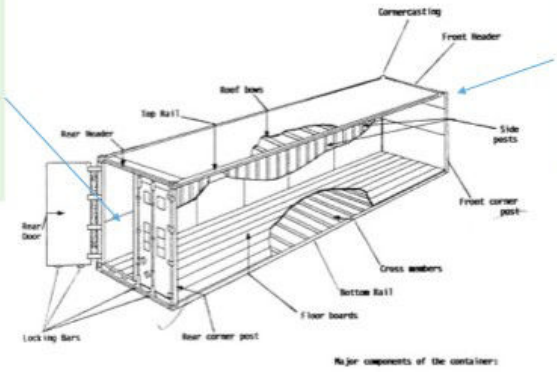
- Standardize the container format, because containers were not portable.
- Analogy:
  - shipping containers are not just steel boxes
  - they are steel boxes that are a standard size, with the same hooks and holes
- Make containers easy to use for developers.
- Emphasis on re-usable components, APIs, ecosystem of standard tools.
- Improvement over ad-hoc, in-house, specific tools.

Running containers everywhere

- Maturity of underlying technology (cgroups, namespaces, copy-on-write systems).
- Ability to run on any Linux server today: physical, virtual, VM, cloud, OpenStack...
- Ability to move between any of the above in a matter of seconds-no modification or delay.
- Ability to share containerized components.
- Self contained environment - no dependency hell.
- Tools for how containers work together: linking, discovery, orchestration...

Technical & cultural revolution: separation of concerns

- Dan 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

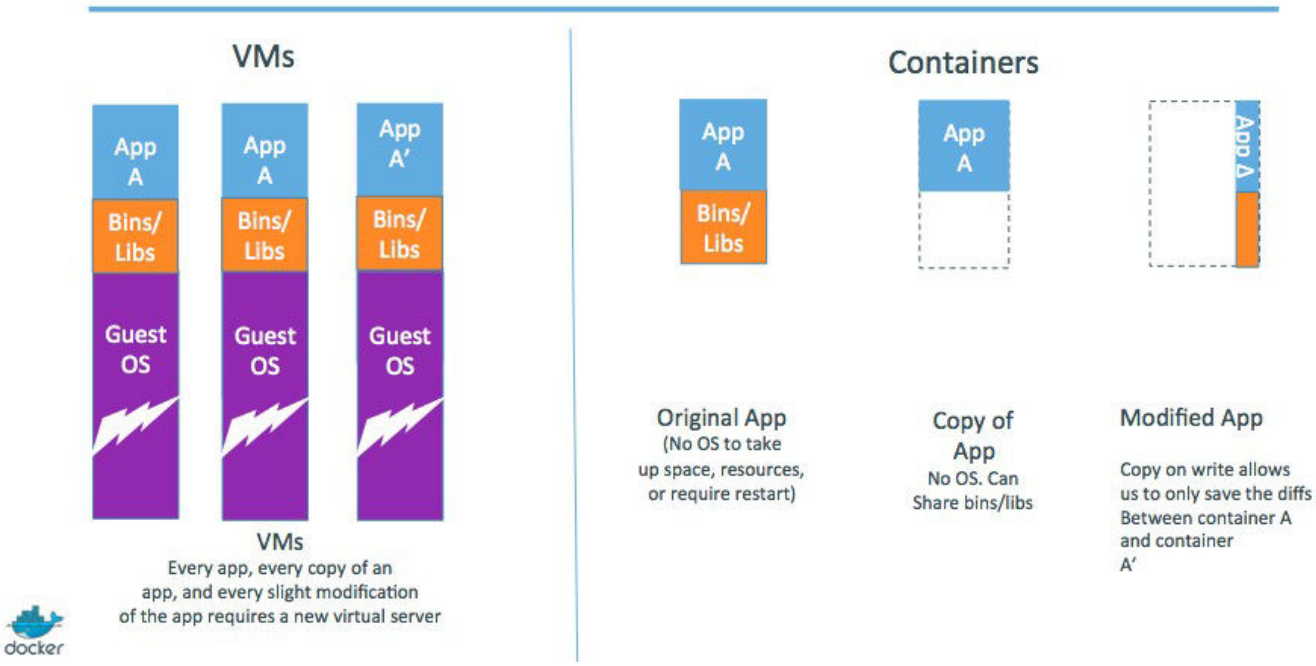


- Oscar the Ops Guy
  - Worries about what's "outside" the container
    - Logging
    - Remote access
    - Monitoring
    - Network config
  - All containers start, stop, copy, attach, migrate, etc. the same way

Step 3: shipping containers efficiently

Ship container images, made of reusable shared layers.

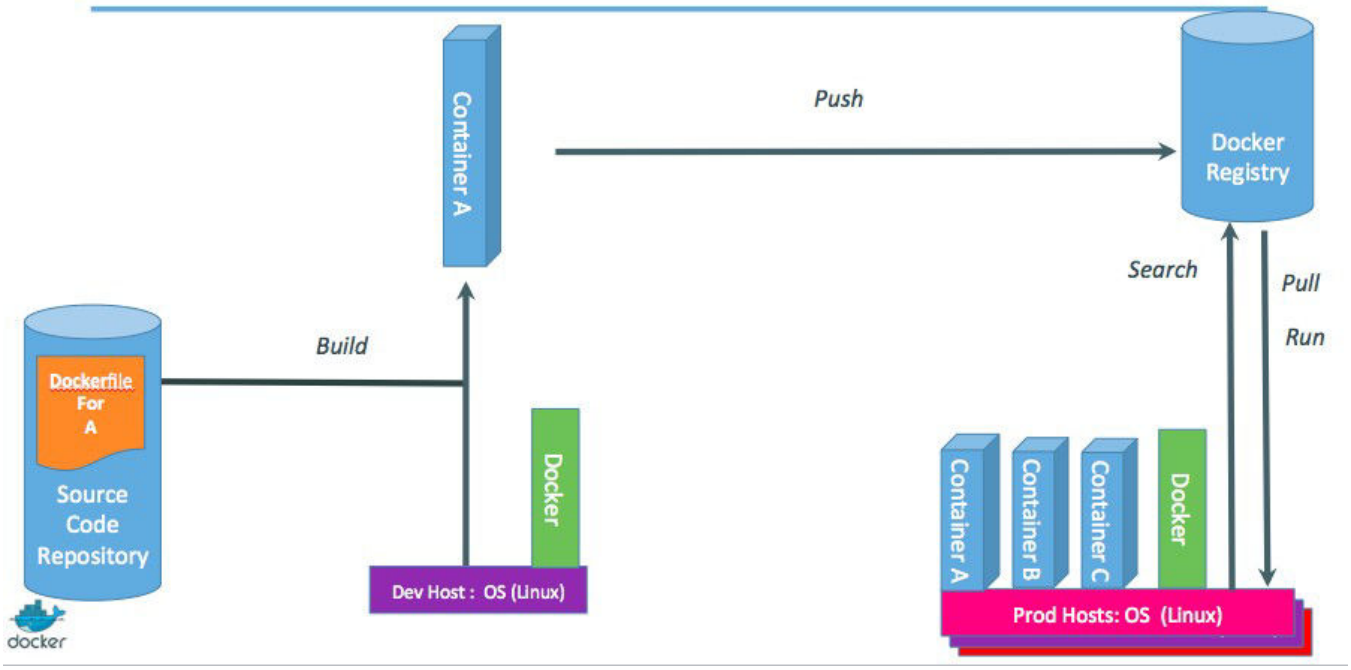
Optimizes disk usage, memory usage, network usage.



Step 4: containers in a modern software factory

The container becomes the new build artefact.

The same container can go from dev, to test, to QA, to prod.



Docker architecture

Docker is a client-server application.

- **The Docker daemon**
  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.

Your training Virtual Machine

Lesson 1: Your training Virtual Machine

If you are following this course as part of an official Docker training or workshop, you have been given credentials to connect to AWS EC2 instance (say Ubuntu)

If you are following this course on your own, without access to an official training Virtual Machine, just skip this lesson, and check "Installing Docker" instead.

Connecting to your Virtual Machine

You need an SSH client.

- On OS X, Linux, and other UNIX systems, just use `ssh`:

```
$ ssh <login>@<ip-address>
```

- Login to the AWS console using your credentials and start a EC2 Ubuntu micro instance

Checking your Virtual Machine

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

```
$ docker version
Client version: 1.4.1
Client API version: 1.16
Go version (client): go1.3.3
Git commit (client): 5bc2ff8
OS/Arch (client): linux/amd64
Server version: 1.4.1
Server API version: 1.16
Go version (server): go1.3.3
Git commit (server): 5bc2ff8
```

- If this doesn't work, raise your hand so that an instructor can assist you!

Install Docker

Lesson 2: Installing Docker

Objectives

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

- Install Docker.
- Run Docker without sudo.

Note: if you were provided with a training VM for a hands-on tutorial, you can skip this chapter, since that VM already has Docker installed, and Docker has already been setup to run without sudo.

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 Fedora:

```
$ sudo yum install docker-io
```

On CentOS 7:

```
$ sudo yum install docker
```

On Debian, Ubuntu 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.com/ | 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 or Microsoft Windows.

To install Docker on these platforms we run a small virtual machine using a tool called Boot2Docker.

Check that Docker is working

Using the docker client:

```
$ docker version
Client version: 1.5.0
Client API version: 1.17
Go version (client): go1.4.1
Git commit (client): a8a31ef
OS/Arch (client): linux/amd64
Server version: 1.5.0
Server API version: 1.17
Go version (server): go1.4.1
Git commit (server): a8a31ef
```

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 service docker restart
```

Log out

```
$ exit
```

Check that Docker works without sudo

```
$ docker version
Client version: 1.5.0
Client API version: 1.17
Go version (client): go1.4.1
Git commit (client): a8a31ef
OS/Arch (client): linux/amd64
Server version: 1.5.0
Server API version: 1.17
Go version (server): go1.4.1
Git commit (server): a8a31ef
```

Section summary

We've learned how to:

- Install Docker.
- Run Docker without sudo.

Our First Containers

Overview: Our First Containers

Objectives

At the end of this lesson, you will have:

- Seen Docker in action.
- Started your first containers.

Hello World

In your Docker environment, just run the following command:

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

That was our first container!

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

A more useful container

Let's run a more exciting container:

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

- This is a brand new container.
- It runs a bare-bones, no-frills `ubuntu` system.

Do something in our container

Try to run `curl` in our container.

```
root@04c0bb0a6c07:/# curl ifconfig.me/ip
bash: curl: command not found
```

Told you it was bare-bones!

Let's check how many packages are installed here.

```
root@04c0bb0a6c07:/# dpkg -l | wc -l
189
```

- `dpkg -l` lists the packages installed in our container
- `wc -l` counts them
- If you have a Debian or Ubuntu machine, you can run the same command and compare the results.

Install a package in our container

We want `curl`, so let's install it:

```
root@04c0bb0a6c07:/# apt-get update
...
Fetched 1514 kB in 14s (103 kB/s)
Reading package lists... Done
root@04c0bb0a6c07:/# apt-get install curl
Reading package lists... Done
...
Do you want to continue? [Y/n]
```

Answer Y or just press `Enter`.

One minute later, `curl` is installed!

```
# curl ifconfig.me/ip
64.134.229.24
```

Exiting our container

Just exit the shell, like you would usually do.

(E.g. with ^D or `exit`)

```
root@04c0bb0a6c07:/# exit
```

- Our container is now in a *stopped* state.
- It still exists on disk, but all compute resources have been freed up.

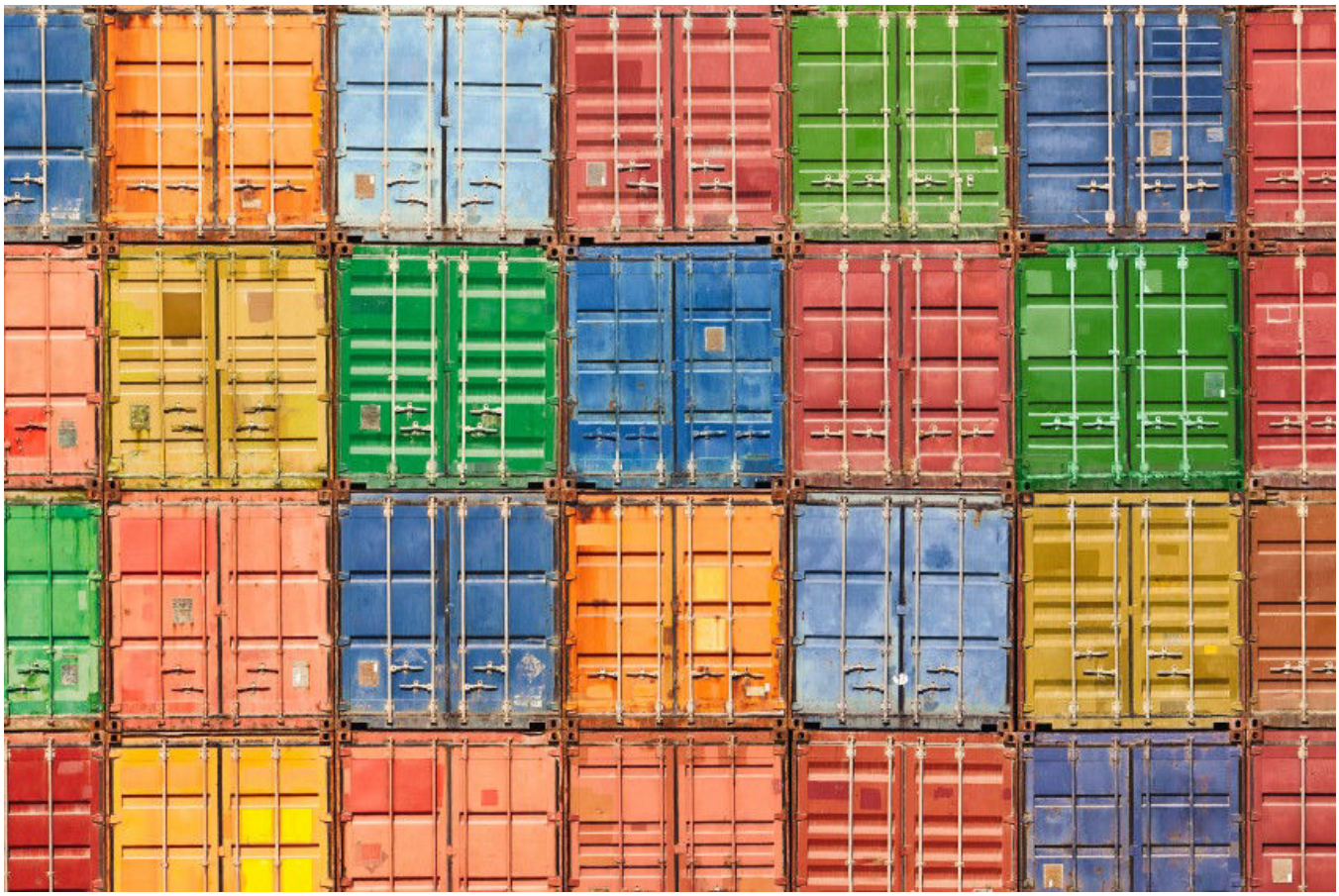
Starting another container

What if we start a new container, and try to run `curl` again?

```
$ docker run -it ubuntu bash
root@b13c164401fb:/# curl
bash: curl: command not found
```

- We started a *brand new container*.
- The basic Ubuntu image was used, and `curl` is not here.
- We will see in the next chapters how to bake a custom image with `curl`.

Background Containers



Background Containers

Our first containers were *interactive*.

We will now see how to:

- Run a non-interactive container.
- Run a container in the background.
- List running containers.
- Check the logs of a container.
- Stop a container.
- List stopped containers.

A non-interactive container

We will run a small custom container.

This container just displays the time every second.

```
$ docker run jpetazzo/clock
Fri Feb 20 00:28:53 UTC 2015
Fri Feb 20 00:28:54 UTC 2015
Fri Feb 20 00:28:55 UTC 2015
...
```

- This container will run forever.
- To stop it, press ^C.
- Docker has automatically downloaded the image `jpetazzo/clock`.
- This image is a user image, created by `jpetazzo`.
- We will tell more about user images (and other types of images) later.

Run a container in the background

Containers can be started in the background, with the `-d` flag (daemon mode):

```
$ docker run -d jpetazzo/clock
47d677dcfba4277c6cc68fcaa51f932b544cab1a187c853b7d0caf4e8debe5ad
```

- We don't see the output of the container.
- But don't worry: Docker collects that output and logs it!
- Docker gives us the ID of the container.

List running containers

How can we check that our container is still running?

With `docker ps`, just like the UNIX `ps` command, lists running processes.

```
$ docker ps
CONTAINER ID  IMAGE                  COMMAND  CREATED        STATUS        ...
47d677dcfba4  jpetazzo/clock:latest  ...      2 minutes ago  Up 2 minutes  ...
```

Docker tells us:

- The (truncated) ID of our container.
- The image used to start the container.
- That our container has been running (Up) for a couple of minutes.
- Other information (COMMAND, PORTS, NAMES) that we will explain later.

Two useful flags for `docker ps`

To see only the last container that was started:

```
$ docker ps -l
CONTAINER ID  IMAGE                  COMMAND  CREATED        STATUS        ...
47d677dcfba4  jpetazzo/clock:latest  ...      2 minutes ago  Up 2 minutes  ...
```

To see only the ID of containers:

```
$ docker ps -q
47d677dcfba4
66b1ce719198
ee0255a5572e
```

Combine those flags to see only the ID of the last container started!

```
$ docker ps -q
47d677dcfba4
```

View the logs of a container

We told you that Docker was logging the container output.

Let's see that now.

```
$ docker logs 47d6
Fri Feb 20 00:39:52 UTC 2015
Fri Feb 20 00:39:53 UTC 2015
...
```

- We specified a *prefix* of the full container ID.
- You can, of course, specify the full ID.
- The `logs` command will output the *entire* logs of the container.
  (Sometimes, that will be too much. Let's see how to address that.)

View only the tail of the logs

To avoid being spammed with eleventy pages of output, we can use the `--tail` option:

```
$ docker logs --tail 3 47d6
Fri Feb 20 00:55:35 UTC 2015
Fri Feb 20 00:55:36 UTC 2015
Fri Feb 20 00:55:37 UTC 2015
```

- The parameter is the number of lines that we want to see.

Follow the logs in real time

Just like with the standard UNIX command `tail -f`, we can follow the logs of our container:

```
$ docker logs --tail 1 --follow 47d6
Fri Feb 20 00:57:12 UTC 2015
Fri Feb 20 00:57:13 UTC 2015
^C
```

- This will display the last line in the log file.
- Then, it will continue to display the logs in real time.
- Use ^C to exit.

Stop our container

There are two ways we can terminate our detached container.

- Killing it using the `docker kill` command.
- Stopping it using the `docker stop` command.

The first one stops the container immediately, by using the `KILL` signal.

The second one is more graceful. It sends a `TERM` signal, and after 10 seconds, if the container has not stopped, it sends `KILL`.

Reminder: the `KILL` signal cannot be intercepted, and will forcibly terminate the container.

Killing it

Let's kill our container:

```
$ docker kill 47d6
47d6
```

Docker will echo the ID of the container we've just stopped.

Let's check that our container doesn't show up anymore:

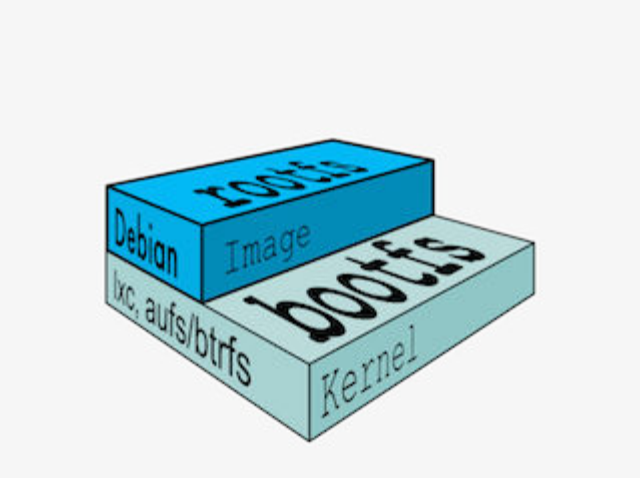
```
$ docker ps
```

List stopped containers

We can also see stopped containers, with the `-a` (`--all`) option.

```
$ docker ps -a
CONTAINER ID  IMAGE                   ...  CREATED      STATUS
47d677dcfba4  jpetazzo/clock:latest  ...  23 min. ago  Exited (0) 4 min. ago
5c1dfd4d81f1  jpetazzo/clock:latest  ...  40 min. ago  Exited (0) 40 min. ago
b13c164401fb  ubuntu:latest          ...  55 min. ago  Exited (130) 53 min. ago
```

Understanding Docker Images



Lesson 3: Understanding Docker Images

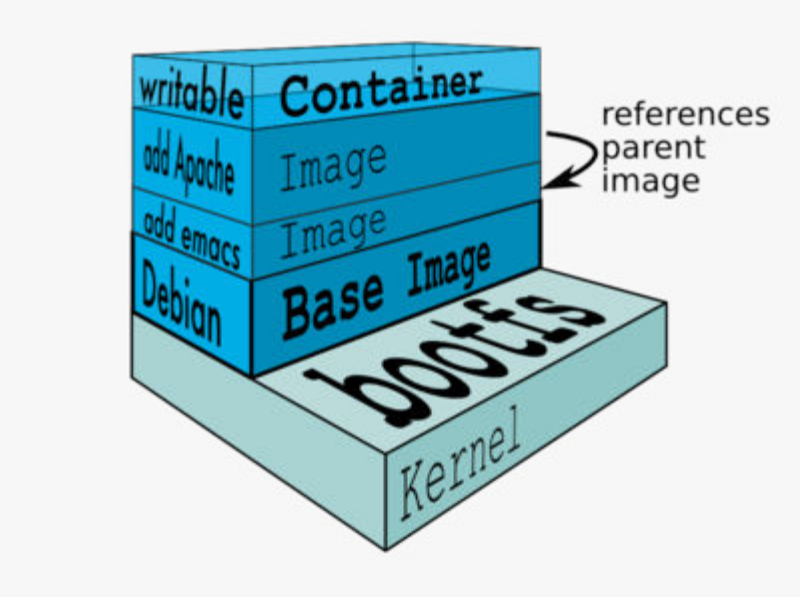
Objectives

In this lesson, we will explain:

- What is an image.
- What is a layer.
- The various image namespaces.
- How to search and download images.

What is an image?

- An image is a collection of files + some meta data.
  (Technically: those files form the root filesystem of a container.)
- Images are made of *layers*, conceptually stacked on top of each other.
- Each layer can add, change, and remove files.
- Images can share layers to optimize disk usage, transfer times, and memory use.



Differences between containers and images

- An image is a read-only filesystem.
- A container is an encapsulated set of processes running in a read-write copy of that filesystem.
- To optimize container boot time, *copy-on-write* is used instead of regular copy.
- `docker run` starts a container from a given image.

Let's give a couple of metaphors to illustrate those concepts.

Image as stencils

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

Object-oriented programming

- Images are conceptually similar to *classes*.
- Layers are conceptually similar to *inheritance*.
- Containers are conceptually similar to *instances*.

Wait a minute…

If an image is read-only, how do we change it?

- We don't.
- We create a new container from that image.
- Then we make changes to that container.
- When we are satisfied with those changes, we transform them into a new layer.
- A new image is created by stacking the new layer on top of the old image.

In practice

There are multiple ways to create new images.

- `docker commit`: creates a new layer (and a new image) from a container.
- `docker build`: performs a repeatable build sequence.
- `docker import`: loads a tarball into Docker, as a standalone base layer.

We will explain `commit` and `build` in later chapters.

`import` can be used for various hacks, but its main purpose is to bootstrap the creation of base images.

Images namespaces

There are three namespaces:

- Root-like

```
ubuntu
```

- User (and organizations)

```
jpetazzo/clock
```

- Self-Hosted

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

Let's explain each of them.

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:

- Small, "swiss-army-knife" images like busybox.
- Distro images to be used as bases for your builds, like ubuntu, fedora…
- Ready-to-use components and services, like redis, postgresql...

User namespace

The user namespace holds images for Docker Hub users and organizations.

For example:

```
jpetazzo/clock
```

The Docker Hub user is:

```
jpetazzo
```

The image name is:

```
clock
```

Self-Hosted namespace

This namespace holds images which are not hosted on Docker Hub, but on third party registries.

They contain the hostname (or IP address), and optionally the port, of the registry server.

For example:

```
localhost:5000/wordpress
```

The remote host and port is:

```
localhost:5000
```

The image name is:

```
wordpress
```

Historical detail

Self-hosted registries used to be called *private* registries, but this was misleading!

- A self-hosted registry can be public or private.
- A registry in the User namespace on Docker Hub can be public or private.

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 download (pull) or upload (push) images.

To be more accurate: you can use the Docker client to tell a Docker server to push and pull images to and from a registry.

Showing current images

Let's look at what images are on our host now.

```
$ docker images
REPOSITORY          TAG     IMAGE ID     CREATED      VIRTUAL SIZE
ubuntu              13.10   9f676bd305a4 7 weeks  ago 178 MB
ubuntu              saucy   9f676bd305a4 7 weeks  ago 178 MB
ubuntu              raring  eb601b8965b8 7 weeks  ago 166.5 MB
ubuntu              13.04   eb601b8965b8 7 weeks  ago 166.5 MB
ubuntu              12.10   5ac751e8d623 7 weeks  ago 161 MB
ubuntu              quantal 5ac751e8d623 7 weeks  ago 161 MB
ubuntu              10.04   9cc9ea5ea540 7 weeks  ago 180.8 MB
ubuntu              lucid   9cc9ea5ea540 7 weeks  ago 180.8 MB
ubuntu              12.04   9cd978db300e 7 weeks  ago 204.4 MB
ubuntu              latest  9cd978db300e 7 weeks  ago 204.4 MB
ubuntu              precise 9cd978db300e 7 weeks  ago 204.4 MB
```

Searching for images

Searches your registry for images:

```
$ docker search zookeeper
NAME                              DESCRIPTION                     STARS  ...
jplock/zookeeper                  Builds a docker image for ...   27
thefactory/zookeeper-exhibitor    Exhibitor-managed ZooKeepe...   2
misakai/zookeeper                 ZooKeeper is a service for...   1
digitalwonderland/zookeeper       Latest Zookeeper - cluster...   1
garland/zookeeper                                                 1
raycoding/piggybank-zookeeper     Zookeeper 3.4.6 running on...   1
gregory90/zookeeper                                               0
```

- "Stars" indicate the popularity of the image.
- "Official" images are those in the root namespace.
- "Automated" images are built automatically by the Docker Hub.
  (This means that their build recipe is always available.)

Downloading images

There are two ways to download images.

- Explicitly, with `docker pull`.
- Implicitly, when executing `docker run` and the image is not found locally.

Pulling an image

```
$ docker pull debian:jessie
Pulling repository debian
b164861940b8: Download complete
b164861940b8: Pulling image (jessie) from debian
d1881793a057: Download complete
```

- As seen previously, images are made up of layers.
- Docker has downloaded all the necessary layers.
- In this example, `:jessie` indicates which exact version of Debian we would like. It is a *version tag*.

Image and tags

- Images can have tags.
- Tags define image variants.
- `docker pull ubuntu` will refer to `ubuntu:latest`.
- The `:latest` tag can be updated frequently.
- When using images it is always best to be specific.

Section summary

We've learned how to:

- Understand images and layers.
- Understand Docker image namespacing.
- Search and download images.

Building Images Interactively

Lesson 4: Building Images Interactively

In this lesson, we will create our first container image.

We will install software manually in a container, and turn it into a new image.

We will introduce commands `docker commit`, `docker tag`, and `docker diff`.

Building Images Interactively

As we have seen, the images on the Docker Hub are sometimes very basic.

How do we want to construct our own images?

As an example, we will build an image that has `wget`.

First, we will do it manually with `docker commit`.

Then, in an upcoming chapter, we will use a `Dockerfile` and `docker build`.

Building from a base

Our base will be the `ubuntu` image.

If you prefer `debian`, `centos`, or `fedora`, feel free to use them instead.

(You will have to adapt `apt` to `yum`, of course.)

Create a new container and make some changes

Start an Ubuntu container:

```
$ docker run -it ubuntu bash
root@<yourContainerId>:#/
```

Run the command `apt-get update` to refresh the list of packages available to install.

Then run the command `apt-get install -y wget` to install the program we are interested in.

```
root@<yourContainerId>:#/ apt-get update && apt-get install -y wget
.... OUTPUT OF APT-GET COMMANDS ....
```

Inspect the changes

Type `exit` at the container prompt to leave the interactive session.

Now let's run `docker diff` to see the difference between the base image and our container.

```
$ docker diff <yourContainerId>
C /root
A /root/.bash_history
C /tmp
C /usr
C /usr/bin
A /usr/bin/wget
...
```

Docker tracks filesystem changes

As explained before:

- An image is read-only.
- When we make changes, they happen in a copy of the image.
- Docker can show the difference between the image, and its copy.
- For performance, Docker uses copy-on-write systems.
  (i.e. starting a container based on a big image doesn't incur a huge copy.)

Commit and run your image

The `docker commit` command will create a new layer with those changes, and a new image using this new layer.

```
$ docker commit <yourContainerId>
<newImageId>
```

The output of the `docker commit` command will be the ID for your newly created image.

We can run this image:

```
$ docker run -it <newImageId> bash
root@fcfb62f0bfde:/# wget
wget: missing URL
...
```

Tagging images

Referring to an image by its ID is not convenient. Let's tag it instead.

We can use the `tag` command:

```
$ docker tag <newImageId> mydistro
```

But we can also specify the tag as an extra argument to `commit`:

```
$ docker commit <containerId> mydistro
```

And then run it using its tag:

```
$ docker run -it mydistro bash
```

What's next?

Manual process = bad.

Automated process = good.

In the next chapter, we will learn how to automate the build process by writing a `Dockerfile`.

Building Docker images

Lesson 5: Building Docker Images

Objectives

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

- Write a `Dockerfile`.
- Build an image from a `Dockerfile`.

`Dockerfile` overview

- A `Dockerfile` is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
- The `docker build` command builds an image from a `Dockerfile`.

Writing our first Dockerfile

Our Dockerfile must be in a **new, empty directory**.

1. Create a directory to hold our Dockerfile.

```
$ mkdir myimage
```

2. Create a Dockerfile inside this directory.

```
$ cd myimage
$ vim Dockerfile
```

Of course, you can use any other editor of your choice.

Type this into our Dockerfile…

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
```

- FROM indicates the base image for our build.
- Each RUN line will be executed by Docker during the build.
- Our RUN commands **must be non-interactive.**
  (No input can be provided to Docker during the build.)

Build it!

Save our file, then execute:

```
$ docker build -t myimage .
```

- `-t` indicates the tag to apply to the image.
- `.` indicates the location of the *build context*.
  (We will talk more about the build context later; but to keep things simple: this is the directory where our Dockerfile is located.)

What happens when we build the image?

The output of `docker build` looks like this:

```
$ docker build -t myimage .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
 ---> e54ca5efa2e9
Step 1 : RUN apt-get update
 ---> Running in 840cb3533193
 ---> 7257c37726a1
Removing intermediate container 840cb3533193
Step 2 : RUN apt-get install -y wget
 ---> Running in 2b44df762a2f
 ---> f9e8f1642759
Removing intermediate container 2b44df762a2f
Successfully built f9e8f1642759
```

- The output of the RUN commands has been omitted.
- Let's explain what this output means.

Sending the build context to Docker

```
Sending build context to Docker daemon 2.048 kB
```

- The build context is the `.` directory given to `docker build`.
- It is sent (as an archive) by the Docker client to the Docker daemon.
- This allows to use a remote machine to build using local files.
- Be careful (or patient) if that directory is big and your link is slow.

Executing each step

```
Step 1 : RUN apt-get update
 ---> Running in 840cb3533193
(...output of the RUN command...)
 ---> 7257c37726a1
Removing intermediate container 840cb3533193
```

- A container (840cb3533193) is created from the base image.
- The RUN command is executed in this container.
- The container is committed into an image (7257c37726a1).
- The build container (840cb3533193) is removed.
- The output of this step will be the base image for the next one.

Running the image

The resulting image is not different from the one produced manually.

```
$ docker run -ti myimage bash
root@91f3c974c9a1:/# wget
wget: missing URL
```

- Sweet is the taste of success!

Using image and viewing history

The `history` command lists all the layers composing an image.

For each layer, it shows its creation time, size, and creation command.

When an image was built with a Dockerfile, each layer corresponds to a line of the Dockerfile.

```
$ docker history myimage
IMAGE         CREATED            CREATED BY                      SIZE
f9e8f1642759  About an hour ago  /bin/sh -c apt-get install -y  6.062 MB
7257c37726a1  About an hour ago  /bin/sh -c apt-get update      8.549 MB
e54ca5efa2e9  8 months ago       /bin/sh -c apt-get update &&8 B
6c37f792ddac  8 months ago       /bin/sh -c apt-get update &&83.43 MB
83ff768040a0  8 months ago       /bin/sh -c sed -is/^#\s*\(d 1.903 kB
2f4b4d6a4a06  8 months ago       /bin/sh -c echo  #!/bin/sh  >  194.5 kB
d7ac5e4f1812  8 months ago       /bin/sh -c #(nop) ADD file:ad  192.5 MB
511136ea3c5a  20 months ago                                      0 B
```

CMD and ENTRYPOINT

Lesson 6: CMD and ENTRYPOINT

Objectives

In this lesson, we will learn about two important Dockerfile commands:

CMD and ENTRYPOINT.

Those commands allow us to set the default command to run in a container.

Defining a default command

When people run our container, we want to automatically execute `wget` to retrieve our public IP address, using ifconfig.me.

For that, we will execute:

```
wget -O- -q http://ifconfig.me/ip
```

- `-O-` tells wget to output to standard output instead of a file.
- `-q` tells wget to skip verbose output and give us only the data.
- `http://ifconfig.me/ip` is the URL we want to retrieve.

Adding CMD to our Dockerfile

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
CMD wget -O- -q http://ifconfig.me/ip
```

- CMD defines a default command to run when none is given.
- It can appear at any point in the file.
- Each CMD will replace and override the previous one.
- As a result, while you can have multiple CMD lines, it is useless.

Build and test our image

Let's build it:

```
$ docker build -t ifconfigme .
...
Successfully built 042dff3b4a8d
```

And run it:

```
$ docker run ifconfigme
64.134.229.24
```

Overriding CMD

If we want to get a shell into our container (instead of running `wget`), we just have to specify a different program to run:

```
$ docker run -it ifconfigme bash
root@7ac86a641116:/#
```

- We specified `bash`.
- It replaced the value of `CMD`.

Using ENTRYPOINT

We want to be able to specify a different URL on the command line, while retaining `wget` and some default parameters.

In other words, we would like to be able to do this:

```
$ docker run ifconfigme http://ifconfig.me/ua
Wget/1.12 (linux-gnu)
```

We will use the `ENTRYPOINT` verb in Dockerfile.

Adding ENTRYPOINT to our Dockerfile

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
ENTRYPOINT ["wget", "-O-", "-q"]
```

- ENTRYPOINT defines a base command (and its parameters) for the container.
- The command line arguments are appended to those parameters.
- Like CMD, ENTRYPOINT can appear anywhere, and replaces the previous value.

Build and test our image

Let's build it:

```
$ docker build -t ifconfigme .
...
Successfully built 36f588918d73
```

And run it:

```
$ docker run ifconfigme http://ifconfig.me/ua
Wget/1.12 (linux-gnu)
```

Great success!

Using CMD and ENTRYPOINT together

What if we want to define a default URL for our container?

Then we will use ENTRYPOINT and CMD together.

- ENTRYPOINT will define the base command for our container.
- CMD will define the default parameter(s) for this command.

CMD and ENTRYPOINT together

Our new Dockerfile will look like this:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y wget
ENTRYPOINT ["wget", "-O-", "-q"]
CMD http://ifconfig.me/ip
```

- ENTRYPOINT defines a base command (and its parameters) for the container.
- If we don't specify extra command-line arguments when starting the container, the value of CMD is appended.
- Otherwise, our extra command-line arguments are used instead of CMD.

Build and test our image

Let's build it:

```
$ docker build -t ifconfigme .
...
Successfully built 6e0b6a048a07
```

And run it:

```
$ docker run ifconfigme
64.134.229.24
$ docker run ifconfigme http://ifconfig.me/ua
Wget/1.12 (linux-gnu)
```

Overriding ENTRYPOINT

What if we want to run a shell in our container?

We cannot just do `docker run ifconfigme bash` because that would try to fetch the URL `bash` (which is not a valid URL, obviously).

We use the `--entrypoint` parameter:

```
$ docker run -it --entrypoint bash ifconfigme
root@6027e44e2955:/#
```