Docker

What is docker?

- Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.
- Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.
- By doing so, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.
- And importantly, Docker is open source. This means that anyone can contribute to Docker and extend it to meet their own needs if they need additional features that aren't available out of the box.

Who is Docker for?

- Docker is a tool that is designed to benefit both developers and system administrators, making it a part of many DevOps (developers + operations) toolchains.
- For developers, it means that they can focus on writing code without worrying about the system that it will ultimately be running on. It also allows them to get a head start by using one of thousands of programs already designed to run in a Docker container as a part of their application.
- For operations staff, Docker gives flexibility and potentially reduces the number of systems needed because of its small footprint and lower overhead.

Docker Introduction

- The software industry has changed.
- Before:
  - monolithic applications
  - long development cycles
  - slowly scaling up
- Now:
  - decoupled services
  - fast, iterative improvements
  - quickly scaling out
- Deployment becomes very complex
  - Many different stacks.
  - Many different targets.

Docker Introduction

- Deployment becomes very complex
  - Many different stacks.
  - Many different targets.
- Solution-
  - **“Build once, configure once and run anywhere….”**
- Earlier Option
  - Use VMs

What is docker?

- Docker is a computer program that performs operating-system-level virtualization also known as containerization.
- **Docker** is an open platform for developers and sysadmins to build, ship, and run distributed applications, whether on laptops, data center VMs, or the cloud.
- It is developed by Docker, Inc. Docker is primarily developed for Linux, where it uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces, and a union-capable file system such as OverlayFS and others to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines (VMs).

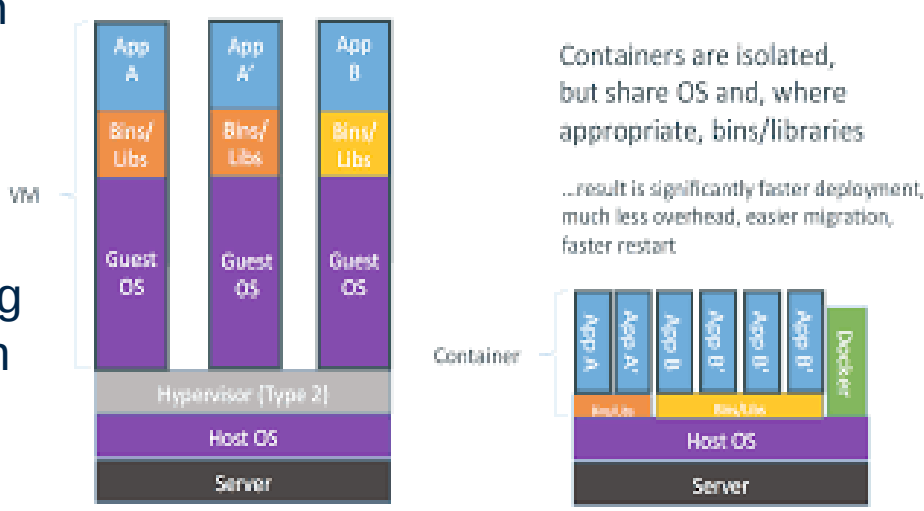
Docker History

- History
  - Solomon Hykes started Docker in France as an internal project within dotCloud, a platform-as-a-service company, with initial contributions by other dotCloud engineers including Andrea Luzzardi and Francois-Xavier Bourlet.
  - The software debuted to the public in Santa Clara at PyCon in 2013.
  - Docker was released as open source in March 2013 and is written in the Go programming language.

VM vs Docker

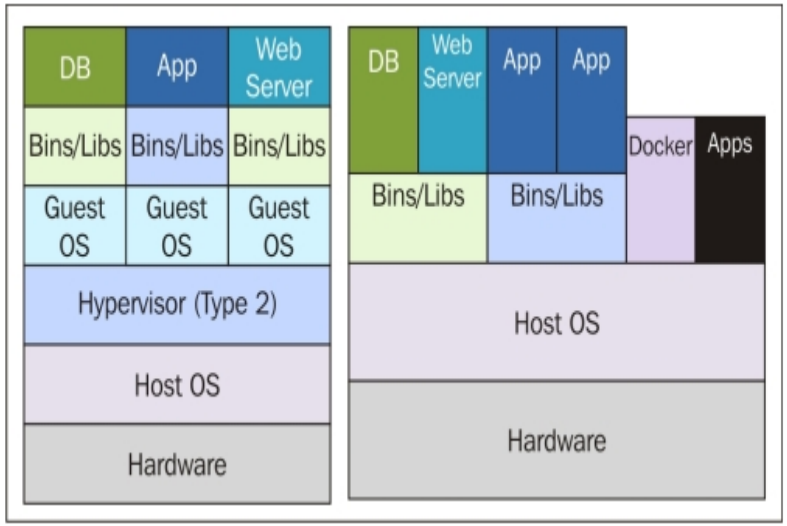
- **Shortcomings of Virtual Machines (VMs):**

1. A VM's size can grow very large when trying to handle all of the required dependencies and packages.
2. They are resource intensive because they consume a great deal of CPU and memory. A complex scenario, like scaling an application for multiple providers, can result in more complexities, such as running out of disk space.
3. From a developer's perspective, their tools for building and testing applications are limited.
4. They produce significant performance overhead, especially when performing IO operations.



Containerization vs Virtualization

Virtual Machines (VMs)	Containers
Represents hardware-level virtualization	Represents operating system virtualization
Heavyweight	Lightweight
Slow provisioning	Real-time provisioning and scalability
Limited performance	Native performance
Fully isolated and hence more secure	Process-level isolation and hence less secure



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 your windows machine

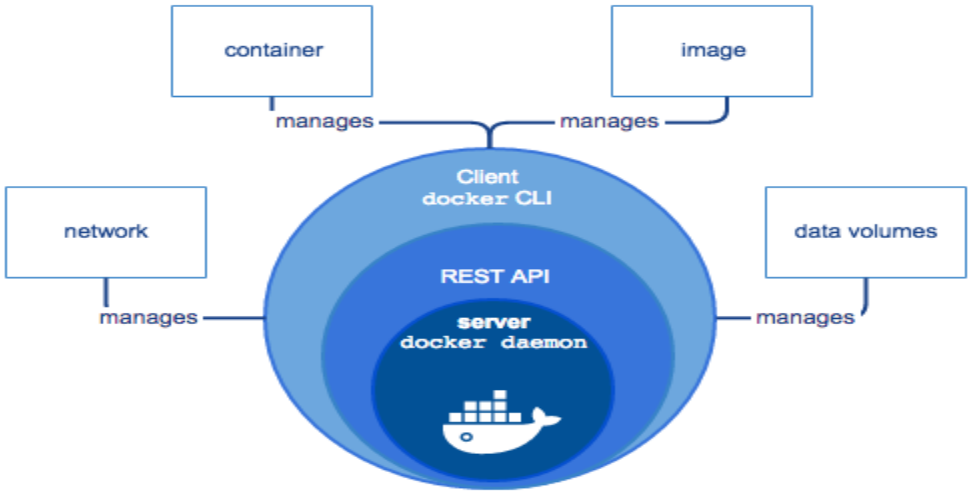
- Use Docker Toolbox, which installs the following components:
  - VirtualBox + Boot2Docker VM image (runs Docker Engine)
  - Kitematic GUI
  - Docker CLI
  - Docker Machine
  - Docker Compose
  - A handful of clever wrappers
- When you execute docker version from the terminal:
  - the CLI prepares a request for the REST API,
  - environment variables tell the CLI where to send the request,
  - the request goes to the Boot2Docker VM in VirtualBox,
  - the Docker Engine in the VM processes the request.
- Reminder: all communication happens over the API!

Installing Docker on Linux Ubuntu

- Ubuntu Linux – 18.04
  - sudo snap install docker
- Centos -7
  - Start by updating your system packages and install the required dependencies:
    - sudo yum update
    - sudo yum install yum-utils device-mapper-persistent-data lvm2
  - Add the Docker stable repository to your system:
    - sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
  - Install the latest version of Docker CE (Community Edition) using yum by typing:
    - sudo yum install docker-ce
  - Start the Docker daemon and enable it to automatically start at boot time:
    - sudo systemctl start docker
    - sudo systemctl enable docker
  - To verify that the Docker service is running type:
    - sudo systemctl status docker

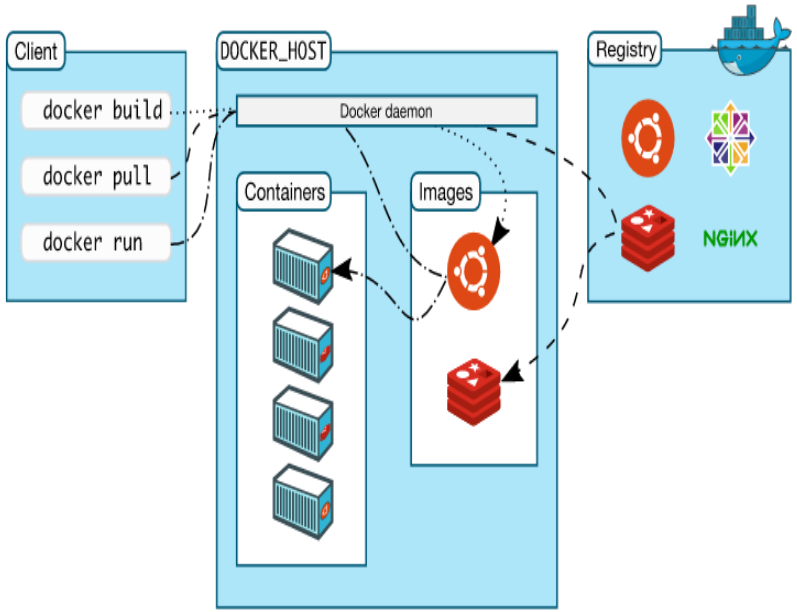
What is Docker Engine?

- *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.
- 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.



Docker's architecture

- Docker uses a client-server architecture.
- The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate via sockets or through a REST API.



Docker’s architecture

- Docker images
  - A Docker image is a read-only template with instructions for creating a Docker container.
  - For example, an image might contain an Ubuntu operating system with Apache web server and your web application installed. You can build or update images from scratch or download and use images created by others.
  - An image may be based on, or may extend, one or more other images.
  - A docker image is described in text file called a Dockerfile, which has a simple, well-defined syntax.

Docker’s architecture

- Docker containers
  - A Docker container is a runnable instance of a Docker image.
  - You can run, start, stop, move, or delete a container using Docker API or CLI commands.
  - When you run a container, you can provide configuration metadata such as networking information or environment variables.
  - Each container is an isolated and secure application platform, but can be given access to resources running in a different host or container, as well as persistent storage or databases.

Docker’s architecture

- Docker registries
  - A docker registry is a library of images. A registry can be public or private, and can be on the same server as the Docker daemon or Docker client, or on a totally separate server.
- Docker services
  - A Docker service allows a swarm of Docker nodes to work together, running a defined number of instances of a replica task, which is itself a Docker image.
  - You can specify the number of concurrent replica tasks to run, and the swarm manager ensures that the load is spread evenly across the worker nodes.
  - To the consumer, the Docker service appears to be a single application. Docker services are the scalability component of Docker.

How does a Docker image work?

- Docker images are read-only templates from which Docker containers are instantiated.
- Each image consists of a series of layers. Docker uses union file systems to combine these layers into a single image.
- Union file systems allow files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.
- These layers are one of the reasons Docker is so lightweight.
- When you change a Docker image, such as when you update an application to a new version, a new layer is built and replaces only the layer it updates.
- The other layers remain intact. To distribute the update, you only need to transfer the updated layer. Layering speeds up distribution of Docker images. Docker determines which layers need to be updated at runtime.

How does a Docker image work?

- An image is defined in a Dockerfile.
- Every image starts from a base image, such as ubuntu, a base Ubuntu image, or fedora, a base Fedora image.
- You can also use images of your own as the basis for a new image, for example if you have a base Apache image you could use this as the base of all your web application images. The base image is defined using the FROM keyword in the dockerfile.
  - Each instruction creates a new layer in the image. Some examples of Dockerfile instructions are:
  - Specify the base image (FROM)
  - Specify the maintainer (MAINTAINER)
  - Run a command (RUN)
  - Add a file or directory (ADD)
  - Create an environment variable (ENV)
  - What process to run when launching a container from this image (CMD)

How does a Docker registry work?

- A Docker registry stores Docker images. After you build a Docker image, you can push it to a public registry such as Docker Hub or to a private registry running behind your firewall. You can also search for existing images and pull them from the registry to a host.

- Docker store allows you to buy and sell Docker images. For image, you can buy a Docker image containing an application or service from the software vendor, and use the image to deploy the application into your testing, staging, and production environments, and upgrade the application by pulling the new version of the image and redeploying the containers.

How does a container work?

- A container uses the host machine's Linux kernel, and consists of any extra files you add when the image is created, along with metadata associated with the container at creation or when the container is started.
- Each container is built from an image. The image defines the container's contents, which process to run when the container is launched, and a variety of other configuration details.
- The Docker image is read-only. When Docker runs a container from an image, it adds a read-write layer on top of the image (using a UnionFS as we saw earlier) in which your application runs.

What happens when you run a container?

- When you use the docker run CLI command or the equivalent API, the Docker Engine client instructs the Docker daemon to run a container.

- $ docker **run** -i -t ubuntu /bin/bash

- This example tells the Docker daemon to run a container using the ubuntu Docker image, to remain in the foreground in interactive mode (-i), and to run the /bin/bash command.

- Background Containers

A non-interactive 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).

View the logs of a container

- 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 elevenly 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
- We can also see stopped containers, with the -a (--all) option.
  - $ docker ps -a

Try it

- **Creating a Database Server**
- **Load mysql image and run it in docker container in background**
  - docker run --name db -d -e MYSQL_ROOT_PASSWORD=123 -p 3306:3306 mysql:latest
- Check container status
  - $ docker ps
- Open another docker shell prompt and connect to this sever by creating another container
  - $ docker exec -it db /bin/bash
  - root@36e68b966fd0:/# mysql -uroot -p123

    mysql> show databases;

    mysql> exit
  - root@36e68b966fd0:/# exit

Try it

- Check the logs produced by mysql server container
- Check what all containers are running
- Stop the containers
- Start the containers again and test it is working

Understanding Docker Images

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.

- 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 ayer.
  - A new image is created by stacking the new layer on top of the old image.

Creating other images

- docker commit
  - Saves all the changes made to a container into a new layer.
  - Creates a new image (effectively a copy of the container).
- docker build
  - Performs a repeatable build sequence.
  - This is the preferred method!
- We will explain both methods in a moment.

Images namespaces

- There are three namespaces:

- Root-like
  - Ubuntu

- User (and organizations)
  - jpetazzo/clock

- Self-Hosted
  - registry.example.com:5000/my-private-image

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.

Some docker image commands

- Showing current images
  - $ docker images
- Searching for images
  - $ docker search zookeeper
- Downloading images
  - There are two ways to download images.
  - Explicitly, with docker pull. - $ docker pull debian:jessie
  - Implicitly, when executing docker run and the image is not found locally.

Building Image Interactively

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

  $ docker run -it <newImageId>

  root@fcfb62f0bfde:/# figlet hello

- We can use the tag command:

  $ docker tag <newImageId> figlet

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

  $ docker commit <containerId> figlet

- And then run it using its tag:

  $ docker run -it figlet

Building Images With A Dockerfile

- 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.

Building Images With A Dockerfile

- 1. Create a directory to hold our Dockerfile.

  $ mkdir myimage

- 2. Create a Dockerfile inside this directory.

  $ cd myimage

  create “Dockerfile” in this directory by using text editor and type following and save it

  ```
  FROM ubuntu
  RUN apt-get update
  RUN apt-get install figlet
  ```

- 3. Build it
  - $ docker build -t figlet . [note . at end – refers current dir for docker file]
- 4. Run it

  $ docker run -ti figlet

  root@91f3c974c9a1:/# figlet hello

Dockerfile Instructions

- FROM : This instruction is used to set the base image for subsequent instructions. It is mandatory to set this in the first line of a Dockerfile. You can use it any number of times though.

  FROM ubuntu

- MAINTAINER : This is a non-executable instruction used to indicate the author of the Dockerfile.
  - MAINTAINER <name>
- RUN : This instruction lets you execute a command on top of an existing layer and create a new layer with the results of command execution.
  - FROM ubuntu
  - RUN apt-get update
  - CMD
- Note: there can be only one CMD instruction in a Dockerfile, if you add more, only the last one takes effect.

Dockerfile Instructions

- LABEL :You can assign metadata in the form of key-value pairs to the image using this instruction. It is important to notice that each LABEL instruction creates a new layer in the image, so it is best to use as few LABEL instructions as possible.
  - LABEL version="1.0" description="This is a sample desc"
- EXPOSE :While running your service in the container you may want your container to listen on specified ports. The EXPOSE instruction helps you do this.
  - EXPOSE 6456
- ENV : This instruction can be used to set the environment variables in the container.
  - ENV var_home="/var/etc"

Dockerfile Instructions

- COPY :This instruction is used to copy files and directories from a specified source to a destination (in the file system of the container).
  - COPY preconditions.txt /usr/temp
- ADD : This instruction is similar to the COPY instruction with few added features like remote URL support in the source field and local-only tar extraction. But if you don’t need a extra features, it is suggested to use COPY as it is more readable.
  - ADD http://www.site.com/downloads/sample.tar.xz /usr/src
- ENTRYPOINT : You can use this instruction to set the primary command for the image.
  - For example, if you have installed only one application in your image and want it to run whenever the image is executed, ENTRYPOINT is the instruction for you.

Dockerfile Instructions

- Note: arguments are optional, and you can pass them during the runtime with something like docker run <image-name>.
- Also, all the elements specified using CMD will be overridden, except the arguments. They will be passed to the command specified in ENTRYPOINT.
  - CMD "Hello World!"
  - ENTRYPOINT echo
- VOLUME : You can use the VOLUME instruction to enable access to a location on the host system from a container. Just pass the path of the location to be accessed.
  - VOLUME /data
- USER : This is used to set the UID (or username) to use when running the image.
  - USER daemon

Dockerfile Instructions

- WORKDIR : This is used to set the currently active directory for other instructions such as RUN, CMD, ENTRYPOINT, COPY and ADD.
  - WORKDIR /user
  - WORKDIR home
  - RUN pwd : This will output the path as /user/home.
- ONBUILD : This instruction adds a trigger instruction to be executed when the image is used as the base for some other image. It behaves as if a RUN instruction is inserted immediately after the FROM instruction of the downstream Dockerfile. This is typically helpful in cases where you need a static base image with a dynamic config value that changes whenever a new image has to be built (on top of the base image).
  - ONBUILD RUN rm -rf /usr/temp

Naming and inspecting containers

- When we create a container, if we don't give a specific name, Docker will pick one for us.
- Specifying a name
  - You can set the name of the container when you create it.
  - $ docker run --name ticktock jpetazzo/clock
  - If you specify a name that already exists, Docker will refuse to create the container.
- The docker inspect command will output a very detailed JSON map.
  - $ docker inspect <containerID>
- Stopping the container
  - $ docker stop <yourContainerID>
- And remove it.
  - $ docker rm <yourContainerID>

Docker Networking

- To get list of networks
  - $ docker network ls
- To connect two containers to the bridge network
  - $ docker run -dit --name alpine1 alpine ash

    docker run -dit --name alpine2 alpine ash
- Check containers started
  - $ docker container ls
- Inspect the bridge network to see what containers are connected to it
  - $ docker network inspect bridge
- The containers are running in the background. Use the docker attach command to connect to alpine1
  - $ docker attach alpine1
- / #

Docker Networking

- Use the ip addr show command to show the network interfaces for alpine1as they look from within the container
  - # ip addr show
- From within alpine1, make sure you can connect to the internet by pinging google.com. The -c 2 flag limits the command to two ping attempts.
  - # ping -c 2 google.com

Docker Networking

- To create the alpine-net network of bridge type. You do not need the --driver bridge flag since it's the default, but this example shows how to specify it.
  - $ docker network create --driver bridge alpine-net
  - $ docker network ls
  - $ docker run -dit --name alpine1 --network alpine-net alpine ash
  - $ docker run -dit --name alpine2 --network alpine-net alpine ash

Container Networking Basics

- A simple, static web server
  - Run the Docker Hub image nginx, which contains a basic web server:
  - $ docker run -d -P nginx
  - 66b1ce719198711292c8f34f84a7b68c3876cf9f67015e752b94e189d35a204e
  - Docker will download the image from the Docker Hub.
  - -d tells Docker to run the image in the background.
  - -P tells Docker to make this service reachable from other computers. (-P is the short version of --publish-all.)
- Finding our web server port
  - $ docker ps
  - CONTAINER ID IMAGE ... PORTS ...
  - e40ffb406c9e nginx ... 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp ...
  - The web server is running on ports 80 and 443 inside the container.
  - Those ports are mapped to ports 32769 and 32768 on our Docker host.

Container Networking Basics

- Connecting to our web server (GUI)
  - Point your browser to the IP address of your Docker host (docker-machine ip)
  - http://host-ip:32769/ - for port 80
- Finding the web server port in a script
  - $ docker port <containerID> 80
  - 32769
- Manual allocation of port numbers
  - If you want to set port numbers yourself, no problem:
    - $ docker run -d -p 80:80 nginx
    - $ docker run -d -p 8000:80 nginx

Container Networking Basics

- Finding the container's IP address
  - We can use the docker inspect command to find the IP address of the container.
    - $ docker inspect --format '{{ .NetworkSettings.IPAddress }}' <yourContainerID>
    - 172.17.0.3
- Pinging our container
  - We can test connectivity to the container using the IP address we've just discovered.
    - $ ping <ipAddress>

Assign static IP to Docker container

- First you need to create you own docker network (mynet123)

  docker network create –subnet 172.18.0.0/16 mynet123
  docker run --net mynet123 --ip 172.18.0.22 -it ubuntu bash

- then in ubuntu shell
  ip addr

- Additionally you could use
  --hostname to specify a hostname
  --add-host to add more entries to /etc/hosts

Using local registry server

- A registry is an instance of the registry image, and runs within Docker.
- Before you can deploy a registry, you need to install Docker on the host.

- Run a local registry
  - Use a command like the following to start the registry container:
  - $ docker run -d -p 5000:5000 --restart=always --name registry registry:2
  - The registry is now ready to use.

Using local registry server

- Pull the ubuntu:16.04 image from Docker Hub.
  - $ docker pull ubuntu:16.04
  - Tag the image as localhost:5000/my-ubuntu.
  - This creates an additional tag for the existing image.When the first part of the tag is a hostname and port, Docker interprets this as the location of a registry, when pushing.
  - $ docker tag ubuntu:16.04 localhost:5000/my-ubuntu

Using local registry server

- Push the image to the local registry running at localhost:5000:
  - $ docker push localhost:5000/my-ubuntu
- Remove the locally-cached ubuntu:16.04 and localhost:5000/my-ubuntu images, so that you can test pulling the image from your registry.
  - $ docker rmi ubuntu:16.04 $ docker rmi localhost:5000/my-ubuntu
- Pull the localhost:5000/my-ubuntu image from your local registry.
  - $ docker pull localhost:5000/my-ubuntu

Stop a local registry

- To stop the registry, use the same docker stop command as with any other container.
  - $ docker stop registry
- To remove the container, use docker rm.
  - $ docker stop registry && docker rm -v registry

Linking docker containers

- Docker has a linking system that allows you to link multiple containers together and send connection information from one to another.
- By linking containers, you provide a secure channel via which Docker containers can communicate to each other.
- Think of a sample web application. You might have a Web Server and a Database Server. When we talk about linking Docker Containers, what we are talking about here is the following:
  1. We can launch one Docker container that will be running the Database Server.
  2. We will launch the second Docker container (Web Server) with a link flag to the container launched in Step 1. This way, it will be able to talk to the Database Server via the link name.
- This is a generic and portable way of linking the containers together rather than via the networking port.

Linking docker containers

- Let us begin first by launching the popular NoSQL Data Structure Server Redis.

  $ docker pull redis

- Next, let us launch a Redis container (named redis1) in detached mode as follows:

  $ docker run -d --name redis1 redis

- We can check that redis1 container has started

  $ docker ps

  Notice that it has started on port 6379.

Linking docker containers

- let us run a another container, a busybox container as shown below:

  $ docker run -it --link redis1:redis --name redisclient1 busybox

- The value provided to the **—link** flag is **sourcecontainername:containeraliasname**.The **container aliasname** has been selected as **redis** and it could be any name of your choice.
- The container (**redisclient1**) will lead you to the shell prompt.
- Let us observe first what entry has got added in **/etc/hosts** file of the **redisclient1** container:

  / # cat /etc/hosts

  …….

  172.17.0.21 **redis** 37f174130f75 redis1

  / #

Linking docker containers

- Notice an entry at the end, where the container **redis1** has got associated with the **redis** name.
- Now, if you do a ping by the host name i.e. alias name (redis)—

  / # ping redis
- If you print out the environment variables you will see the following.

```
/ # set
……
REDIS_ENV_REDIS_VERSION='3.0.3'
REDIS_NAME='/redisclient1/redis'
REDIS_PORT='tcp://172.17.0.21:6379'
REDIS_PORT_6379_TCP='tcp://172.17.0.21:6379'
REDIS_PORT_6379_TCP_ADDR='172.17.0.21'
REDIS_PORT_6379_TCP_PORT='6379'
REDIS_PORT_6379_TCP_PROTO='tcp'
```

Linking docker containers

- Let us launch a container based on the **redis** image

  $ docker run -it --link redis1:redis --name client1 redis sh

- Next thing is to launch the redis client (**redis-cli**) and connect to our redis server.

  # redis-cli -h redis

  redis:6379>

- Now, let us execute some standard Redis commands:

  redis:6379> PING

  PONG

  redis:6379> set myvar DOCKER

  OK

  redis:6379> get myvar

  "DOCKER"

Linking docker containers

- Let us exit this container and launch another client (client2) that wants to connect to the same Redis Server that is still running in the first container that we launched and to which we added a string key / value pair.
- $ docker run -it --link redis1:redis --name client2 redis sh
- Once again, we execute a few commands to validate things:

```
# redis-cli -h redis
redis:6379> get myvar
"DOCKER"
redis:6379>
```

Volume Mounting -Sharing Data Between the Host and the Docker Container

- Let's say you wanted to use the official Docker Nginx image but you wanted to keep a permanent copy of the Nginx's log files to analyze later. By default the nginx Docker image logs to the /var/log/nginx directory, but this is /var/log/nginx inside the Docker Nginx container. Normally it's not reachable from the host filesystem.
- Let's create a folder to store our logs and then run a copy of the Nginx image with a shared volume so that Nginx writes its logs to our host's filesystem instead of to the /var/log/nginx inside the container:

  mkdir ~/nginxlogs
- Then start the container:

  docker run -d -v ~/nginxlogs:/var/log/nginx -p 5000:80 -i nginx

  We set up a volume that links the /var/log/nginx directory from inside the Nginx container to the ~/nginxlogs directory on the host machine.

Volume Mounting -Sharing Data Between the Host and the Docker Container

- So, we now have a copy of Nginx running inside a Docker container on our machine
- Let's use curl to do a quick test request:

  $curl localhost:5000
- This will create some logs entries
- If you look in the ~/nginxlogs folder on the host machine and take a look at the access.log file you'll see a log message from Nginx showing our request

  $cat ~/nginxlogs/access.log

- If you make any changes to the ~/nginxlogs folder, you'll be able to see them from inside the Docker container in real-time as well.

Using Docker Compose for Development Stacks

- Dockerfiles are great to build a single container.
- But when you want to start a complex stack made of multiple containers, you need a different tool. This tool is Docker Compose.
- What is Docker Compose?
  - Docker Compose (formerly known as fig) is an external tool. It is optional (you do not need Compose to run Docker and containers) but we recommend it highly!
  - The general idea of Compose is to enable a very simple, powerful onboarding workflow:
    - 1. Clone your code.
    - 2. Run docker-compose up.
    - 3. Your app is up and running!

Docker Compose Tool

- Installing in aws-ec2 instance

- sudo curl -L https://github.com/docker/compose/releases/download/1.21.0/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose

- sudo chmod +x /usr/local/bin/docker-compose

Compose overview

- This is how you work with Compose:
  - You describe a set (or stack) of containers in a YAML file called dockercompose.yml.
  - You run docker-compose up.
  - Compose automatically pulls images, builds containers, and starts them.
  - Compose can set up links, volumes, and other Docker options for you.
  - Compose can run the containers in the background, or in the foreground.
  - When containers are running in the foreground, their aggregated output is shown.

The docker-compose.yml file

- Each section of the YAML file must contain either build, or image.
  - build indicates a path containing a Dockerfile.
  - image indicates an image name (local, or on a registry).
  - The other parameters are optional.
  - They encode the parameters that you would typically add to docker run.
  - Sometimes they have several minor improvements.

```
www:
  build: www
  ports:
    - 8000:5000
  links:
    - redis
  user: nobody
  command: python counter.py
  volumes:
    - ./www:/src
```

The docker-compose.yml file

- Container parameters
  - command indicates what to run (like CMD in a Dockerfile).
  - ports translates to one (or multiple) -p options to map ports. You can specify local ports (i.e. x:y to expose public port x).
  - volumes translates to one (or multiple) -v options. You can use relative paths here.
  - links translates to one (or multiple) --link options. You can refer to other Compose containers by their name.
- For the full list, check http://docs.docker.com/compose/yml/.

The docker-compose.yml file

- Compose commands
  - We already saw docker-compose up, but another one is docker-compose build. It will execute docker build for all containers mentioning a build path.

- It is common to execute the build and run steps in sequence:

  docker-compose build && docker-compose up

- Another common option is to start containers in the background:

  docker-compose up –d

- Compose makes it easier; with docker-compose ps you will see only the status of the containers of the current stack:
  - $ docker-compose ps

Compose Cleaning up

- Cleaning up
  - If you have started your application in the background with Compose and want to stop it easily, you can use the kill command:

```
$ docker-compose kill
```

  - Likewise, docker-compose rm will let you remove containers (after confirmation):

```
$ docker-compose rm
```

Docker Machine

- Docker Machine is a tool that lets you install Docker Engine on virtual hosts, and manage the hosts with docker-machine commands.
- You can use Machine to create Docker hosts on your local Mac or Windows box, on your company network, in your data center.
- Using docker-machine commands, you can start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon, and configure a Docker client to talk to your host.
- Point the Machine CLI at a running, managed host, and you can run docker commands directly on that host.
- For example,
  - run docker-machine env default

  to point to a host called default and follow on-screen instructions to complete env setup, and run docker ps, docker run hello-world, and so forth.

Installing Docker Machine

- Download the Docker Machine binary and extract it to your PATH.

- If you are running Linux:

```
$ base=https://github.com/docker/machine/releases/download/v0.16.0 &&
  curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine &&
  sudo install /tmp/docker-machine /usr/local/bin/docker-machine
```

- If you are running Windows with Git BASH:

```
$ base=https://github.com/docker/machine/releases/download/v0.16.0 &&
  mkdir -p "$HOME/bin" &&
  curl -L $base/docker-machine-Windows-x86_64.exe > "$HOME/bin/docker-machine.exe"
&&
  chmod +x "$HOME/bin/docker-machine.exe"
```

Get started with Docker Machine

- Create a machine.

  $ docker-machine create --driver virtualbox default

- Use docker-machine ls to list available machines.
- Get the environment commands for your new VM

  $ $docker-machine env default

- Connect your shell to the new machine.

  $ eval "$(docker-machine env default)“

- docker-machine ssh
  - To log into or run a command on a machine using SSH.

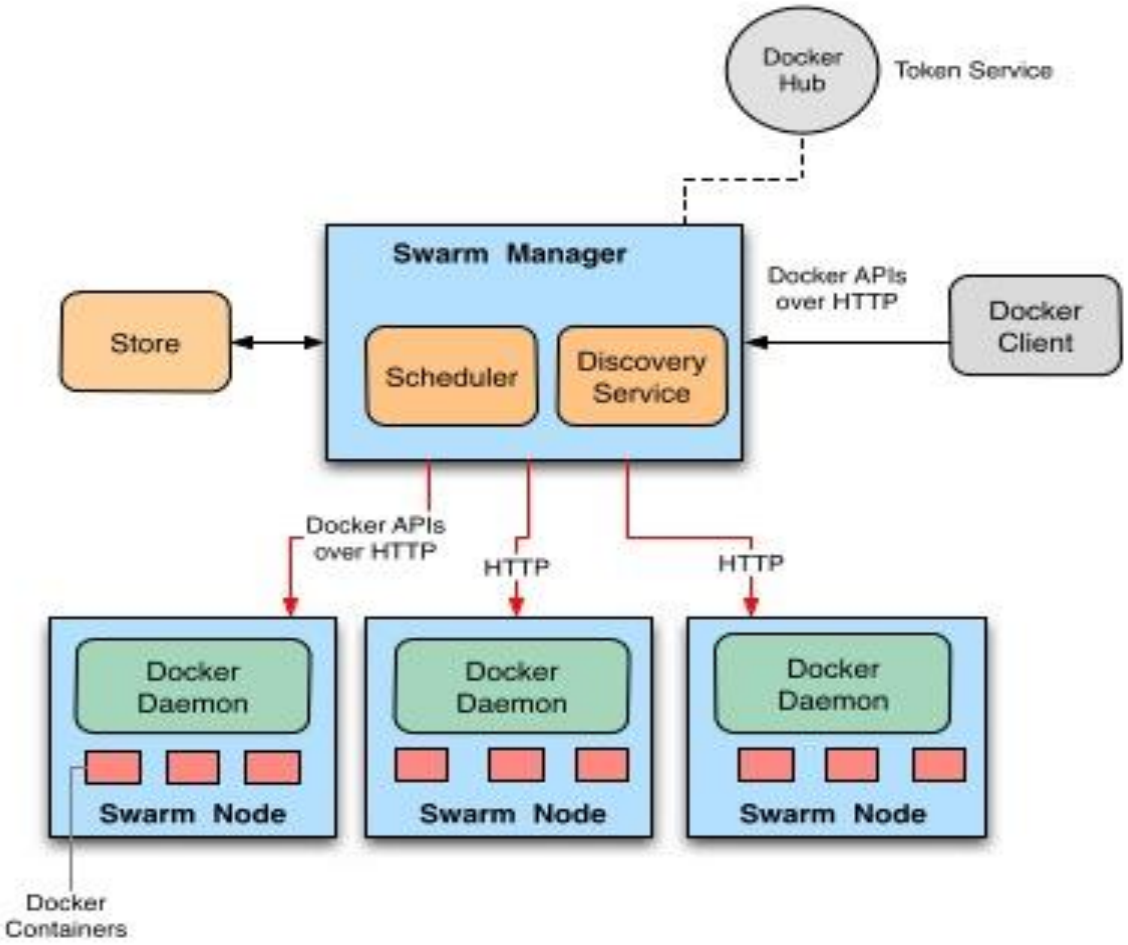
    $ docker-machine ssh machinename
  - You can also specify commands to run

    $ docker-machine ssh dev free

Docker SWARM

- Docker Swarm is a clustering and scheduling tool for Docker containers.
- With Swarm, IT administrators and developers can establish and manage a cluster of Docker nodes as a single virtual system.
- Clustering is an important feature for container technology, because it creates a cooperative group of systems that can provide redundancy, enabling Docker Swarm failover if one or more nodes experience an outage.
- An IT administrator controls Swarm through a swarm manager, which orchestrates and schedules containers. The swarm manager allows a user to create a primary manager instance and multiple replica instances in case the primary instance fails. In Docker Engine's swarm mode, the user can deploy manager and worker nodes at runtime.

Docker Swarm Architecture - Exploded



Docker SWARM –Setting up cluster

- Create 3 Docker Machines, where one of them will act as the Manager (Leader) and the other will be worker nodes.
- Create manager1 docker machine

  docker-machine create --driver hyperv manager1

  or – depending on driver available

  docker-machine create --driver virtualbox manager1
- Create worker1 and workder2 docker machine similarly

  docker-machine create --driver virtualbox worker1

  docker-machine create --driver virtualbox worker2
- Fire the **docker-machine ls** command to check on the status of all the Docker machines

Docker SWARM –Setting up cluster

- Note down the IP Address of the manager1, since you will be needing that.

- One way to get the IP address of the manager1 machine is as follows:

  $ docker-machine ip manager1

- You can SSH into any of the Docker Machines

  $docker-machine ssh <machine-name>

- The first thing to do is initialize the Swarm. We will SSH into the manager1 machine and initialize the swarm in there.

  $ docker-machine ssh manager1

Docker SWARM –Setting up cluster

- Perform the following steps-on manager1 host:

  $ docker swarm init --advertise-addr **MANAGER_IP**

- Read the output produced- you will find ‘docker swarm join …’. You can obtain it using ‘docker swarm join-token **worker’** also.
- Copy docker swarm join command…
- Adding Worker Nodes to our Swarm
  - SSH into the worker1 machine i.e. docker-machine ssh worker1
  - Then fire the command ‘ docker swarm join… ‘
- After making all worker nodes join the Swarm. SSH to manager1 fire the following command to check on the status of my Swarm

  $ docker node ls

Docker SWARM –Setting up cluster

- Create a Service – execute following on manager1

  $docker service create --replicas 5 -p 80:80 --name web nginx

- You can find out the status of the service

  docker@manager1:~$ docker service ls

- You can also see the status of the service and how it is getting orchestrated to the different nodes by using the following command:

  docker@manager1:~$ docker service ps web

- Accessing the service
  - You can access the service by hitting any of the manager or worker nodes. It does not matter if the particular node does not have a container scheduled on it. That is the whole idea of the swarm.
  - Try out a curl to any of the Docker Machine IPs (manager1 or worker1/2) or hit the URL (http://<machine-ip>) in the browser. You should be able to get the standard NGINX Home page.

Docker SWARM –Setting up cluster

- Scaling up and Scaling down

  docker service scale web=8

  docker service ls

  docker service ps web

- Inspecting nodes
  - You can inspect the nodes anytime via the docker node inspect command.

  - $ docker node inspect self
  - $ docker node inspect worker1

- Remove the Service

  docker@manager1:~$ docker service rm web

- Applying Rolling Updates

  $ docker service update --image <imagename>:<version> web

Summary

- Using docker it easy to
  - Build your own PaaS.
  - Create a web-based environment
  - Deploy applications
  - Automate application deployment.
  - Reproduce environments between dev, production, QA, etc..