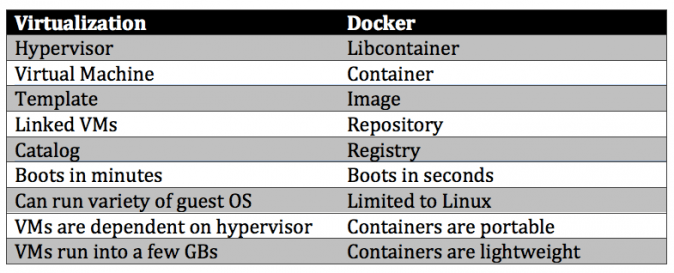
**Docker Containers vs Virtual Machines**



When it comes to comparing the two, it could be said that Docker Containers have much more potential than Virtual Machines. It’s evident as Docker Containers are able to share a single kernel and share application libraries. Containers present a lower system overhead than Virtual Machines and performance of the application inside a container is generally same or better as compared to the same application running within a Virtual Machine.

There is one key metric where Docker Containers are weaker than Virtual Machines, and that’s “Isolation”. Intel’s VT-d and VT- x technologies have provided Virtual Machines with ring-1 hardware isolation of which, it takes full advantage. It helps Virtual Machines from breaking down and interfering with each other. Docker Containers yet don’t have any hardware isolation, thus making them receptive to exploits.

As compared to virtual machines, containers can be faster and less resource heavy as long as the user is willing to stick to a single platform to provide the shared OS. A virtual machine could take up several minutes to create and launch whereas a container can be created and launched just in a few seconds. Applications contained in containers offer superior performance, compared to running the application within a virtual machine.

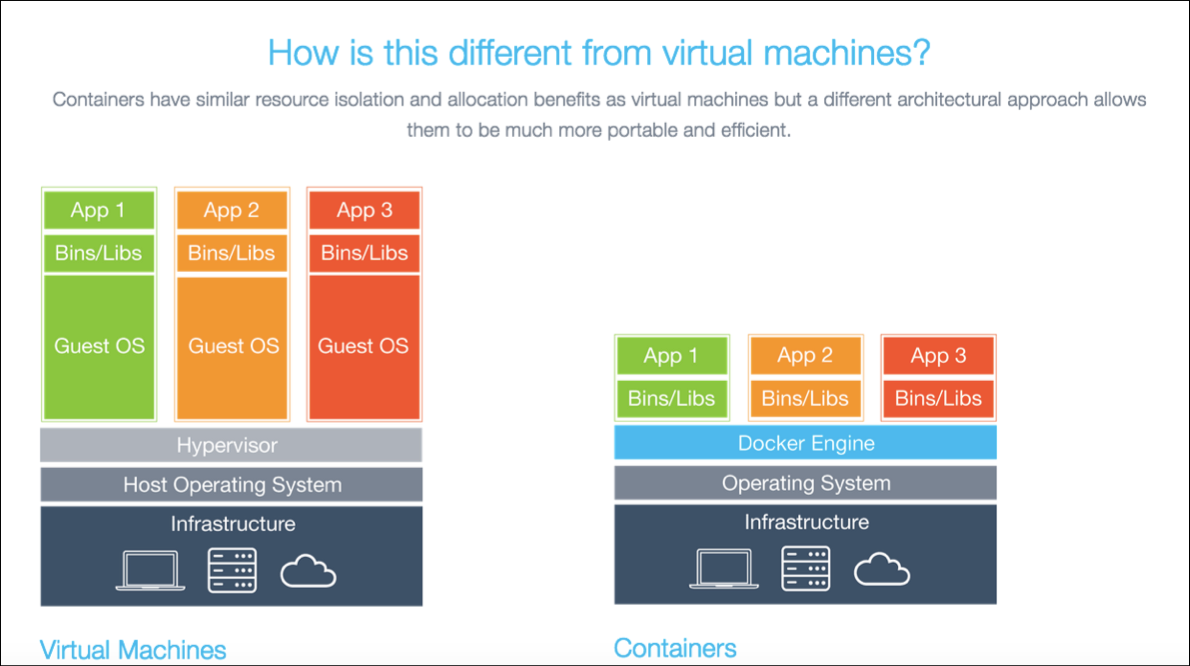
There is an estimation being done by Docker that application running in a container can go twice as fast as one in a virtual machine. Also, a single server can pack more than one containers as OS is not duplicated for each application.

Docker Containers are transforming the DevOps (developer operations) landscape as an important tool in DevOps arsenal. The use cases for Docker Containers within the realm of DevOps are plenty. Running apps over Docker Containers and then deploying anywhere (Cloud or on-premise or any flavor of linux) is a reality now.

Working in heterogeneous environment, Virtual Machines provide high flexibility whereas Docker containers’ prime focus is on applications and their dependencies.

With Docker ecosystem you will never need to move around Gigabytes on "small changes" and you don't need to worry about losing performance by packaging applications into a Docker container on runtime. You don't need to worry about versions of that image. And finally you will even often be able to reproduce complex production environments even on your linux laptop

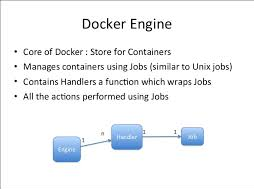
And of course you can start docker containers in VMs (it's a good idea). Reduce your server provisioning on VM level. All the above could be managed by Docker.



**Docker Engine**

Docker Engine is a lightweight runtime and robust tooling that builds and runs your Docker containers. Docker Engine runs on Linux to create the operating environment for your distributed applications. The in-host daemon communicates with the Docker client to execute commands to build, ship and run containers.

The Docker engine replaces the need of a heavyweight hypervisor. It uses the Linux namespaces and control groups.



**Docker Build, Run and Ship**

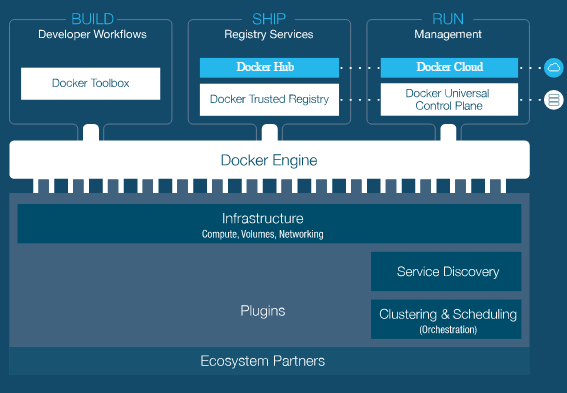
What is build, Run and Ship in Docker???

Basically Build refers to the process of building an image. Run refers to running the image as container and Ship refers to pushing the app into the Docker hub and which can be pulled from anywhere.

**Build:** Docker allows you to compose your application from micro services, without worrying about inconsistencies between development and production environment and without locking into any platforms or language

**Ship:** Docker lets you design the entire cycle of application development, testing and distribution, and manage it with a consistent user interface

**Run:** Docker offers you the ability to deploy scalable services, securely and reliably, on a wide variety of platforms



How to build an image and run a container can be understood when knowing about Docker file.

**Docker File**

How an application is provided as image? How an applications jar file can be pulled as an image? The answers to these questions can be found when understanding the Dockerfile.

For example let’s take Jenkins, this application can be pulled as an image from Docker global repository. It is achieved by executing the commands in the Dockerfile. Below is the Dockerfile for Jenkins

# The image for Jenkins is started by using the Java image as the base image. Which means Jenkins image is built on top of the Java image

FROM java:8-jdk

RUN apt-get update && apt-get install -y wget git curl zip && rm -rf /var/lib/apt/lists/\*

# Setting up environment variables for Jenkins.

ENV JENKINS\_HOME /var/jenkins\_home

ENV JENKINS\_SLAVE\_AGENT\_PORT 50000

# Jenkins is run with user `jenkins`, uid = 1000# If you bind mount a volume from the host or a data container,

# ensure you use the same uid

RUN useradd -d "$JENKINS\_HOME" -u 1000 -m -s /bin/bash Jenkins

# Jenkins home directory is a volume, so configuration and build history

# can be persisted and survive image upgrades

# The concept of Volume is the latest update in docker

VOLUME /var/jenkins\_home

# `/usr/share/jenkins/ref/` contains all reference configuration we want

# to set on a fresh new installation. Use it to bundle additional plugins

# or config file with your custom jenkins Docker image.

RUN mkdir -p /usr/share/jenkins/ref/init.groovy.d

ENV TINI\_SHA 066ad710107dc7ee05d3aa6e4974f01dc98f3888

# Use tini as subreaper in Docker container to adopt zombie processes

RUN curl -fL https://github.com/krallin/tini/releases/download/v0.5.0/tini-static -o /bin/tini && chmod +x /bin/tini \ && echo "$TINI\_SHA /bin/tini" | sha1sum -c -

COPY init.groovy /usr/share/jenkins/ref/init.groovy.d/tcp-slave-agent-port.groovy

ENV JENKINS\_VERSION 1.642.2ENV JENKINS\_SHA e72e06e64d23eefb13090459f517b0697aad7be0

# could use ADD but this one does not check Last-Modified header

# see <https://github.com/docker/docker/issues/8331>

RUN curl -fL http://repo.jenkins-ci.org/public/org/jenkins-ci/main/jenkins-war/${JENKINS\_VERSION}/jenkins-war-${JENKINS\_VERSION}.war -o /usr/share/jenkins/jenkins.war \ && echo "$JENKINS\_SHA /usr/share/jenkins/jenkins.war" | sha1sum -c –

ENV JENKINS\_UC <https://updates.jenkins-ci.org>

RUN chown -R jenkins "$JENKINS\_HOME" /usr/share/jenkins/ref

# for main web interface:

EXPOSE 8080

# will be used by attached slave agents:

EXPOSE 50000

ENV COPY\_REFERENCE\_FILE\_LOG $JENKINS\_HOME/copy\_reference\_file.log

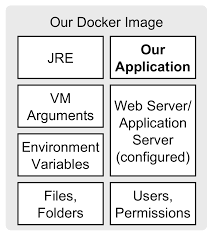
USER jenkins

COPY jenkins.sh /usr/local/bin/jenkins.sh

ENTRYPOINT ["/bin/tini", "--", "/usr/local/bin/jenkins.sh"]

# from a derived Dockerfile, can use `RUN plugins.sh active.txt` to setup /usr/share/jenkins/ref/plugins from a support bundle

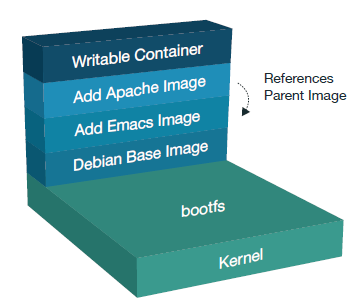
COPY plugins.sh /usr/local/bin/plugins.sh



The above image explains what is inside our image based on a Dockerfile.

**Docker Image**

* Docker images are the basics of the containers at rest
* These artifacts are stored and managed in a registry.
* An image is a collection of files + some metadata. (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.
* Images are conceptually similar to classes in Object Oriented Programming
* Layers are conceptually similar to inheritance in Object Oriented Programming
* Each image consists of a series of layers. Docker makes use of 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



The above diagram explains how images are created and the layers.

**Docker Containers**

A Docker container wraps up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. By encapsulating and isolating everything in a container, this guarantees that the container will always run the same, regardless of the environment it is running in.

Docker containers are built from Docker images, which use union filesystems and share image layers which further attribute to their lightweight nature.

Containers are created from an image with the “docker run” command. Once initiated, the Docker Engine spins up that container from the defined image (it can also spin them down). Every command is executed to the Docker Engine including creating new containers, scaling existing containers, stopping, removing and much more.

Docker containers make distributed applications:

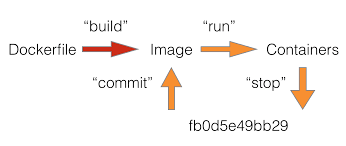
**Composable:** Lightweight containers start instantly and use minimal resources because they share the kernel of the underlying operating system

**Dynamic:** Quick to spin up and easy to scale and change make containers ideal for agile teams

**Portable:** Able to run on any physical, virtual or cloud environment running Docker.

Containers can uniquely turn very diverse set of application services into standardized software units. From the outside the containers all look the same which regardless of the binaries, libraries and code inside the container. This makes it easy to build tooling around container to move them along every stage of the application lifecycle. You can create simple single container or complex multi-container applications.

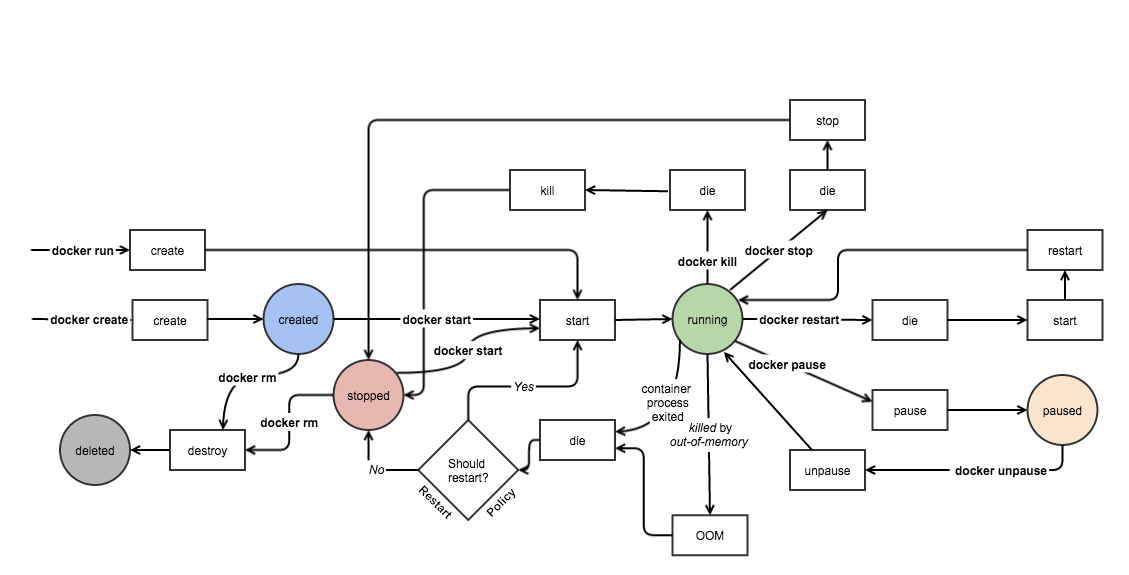
Containers are conceptually similar to instances in Object Oriented Programming language



The above image explains how an image is created using a Dockerfile and how a container is started using the image.

So on building a Dockerfile an image is created and it will be at rest and can be updated. Once the image is made to run it will turn into a container

**Docker Life Cycle:**

****

**Docker Volume**

Docker images are the files that make up an application. For example, if you download the Docker [Nginx image](https://hub.docker.com/_/nginx/) from the Docker Hub (the public Docker Registry), you'll be downloading an image that contains a minimal Linux distribution and the binaries that Nginx itself needs to run. A container is like another layer that sits on top of this image without ever modifying the original image itself. If you're familiar with the concept of snapshots in VirtualBox or other virtualization tools, this works similarly. To save your changes you actually have to create a new Docker image (not container) that includes your newly changed files.

That's fine if you're interested in creating new Docker images, but what if you just want to do something like use the public Nginx Docker image to serve your own webpage? Having to copy your webpages into place every single time you start your Docker container is far from ideal. This is where data volumes come in. Docker data volumes provide you with a separate place to store other data. You can tell Docker to use any folder as a container, and from then on any files written to that location will be persisted.

For instance lets run a container which has “Ubuntu”

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

This command does a bunch of things in one go. The run portion with the -t -i options tells Docker to prepare to run an image in interactive mode with a terminal. The ubuntu command line option is the name of the image to run, and the /bin/bash is the name of the executable inside that image that we want to start. Docker will automatically attempt to download images that it can't find locally from the public Docker Registry, so running this command will result in Docker downloading an Ubuntu image from the Docker Registry and then launching bash inside it, which gives us a standard shell.

The prompt will change to something similar to:

Ubuntu container bash shell

root@cbceb444ec6a:/#

Let's go ahead and write a file to the root of the filesystem inside the Docker container:

* echo "I'm going to disappear" > /byebye

You can see that the file /byebye was created and its contents are present via the usual shell commands. For example, inside the bash shell for your Ubuntu container, if you list the contents of the / directory with the command:

* ls /

You will see the following:

Output of ls /

bin boot byebye dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

If you issue the command:

* cat /byebye

You will see the following:

Output of `cat /byebye`

I'm going to disappear

Go ahead and type exit at the shell now, and you'll be dropped back to your host system's shell.

Now, scroll back up and repeat the steps in this section again, except don't create the file /byebye again. You will see that it no longer exists. For example, the command:

* cat /byebye

now shows the following:

Output of cat /byebye

cat: /byebye: No such file or directory

Be sure to exit from the Ubuntu container bash shell:

* exit

## Learning the Types of Docker Data Volumes

There are three main use cases for Docker data volumes:

1. To keep data around, even through container restarts
2. To share data between the host filesystem and the Docker container
3. To share data with other Docker containers

The third case is a little more advanced, so we won't go into it in this tutorial, but the first two are quite common.

In the first (and simplest) case you just want the data to hang around even if you shut down or restart the container, so it's often easiest to let Docker manage where the data gets stored.

## Keeping Data Persistent

There's no way to directly create a "data volume" in Docker, so instead we create a data volume containerwith a volume attached to it. For any other containers that you then want to connect to this data volume container, use the Docker's --volumes-from option to grab the volume from this container and apply them to the current container. This is a bit unusual at first glance, so let's run through a quick example of how we could use this approach to make our byebye file stick around even if the container is restarted.

First, create a new data volume container to store our volume:

* docker create -v /tmp --name datacontainer ubuntu

This created a container named datacontainer based off of the ubuntu image and in the directory/tmp.

Now, if we run a new Ubuntu container with the --volumes-from flag and run bash again as we did earlier, anything we write to the /tmp directory will get saved to the /tmp volume of our datacontainercontainer.

First, start the ubuntu image:

* docker run -t -i --volumes-from datacontainer ubuntu /bin/bash

The -t command line options calls a terminal from inside the container. The -i flag makes the connection interactive.

At the bash prompt for the ubuntu container, create a file in /tmp:

* echo "I'm not going anywhere" > /tmp/hi

Go ahead and type exit to return to your host machine's shell. Now, run the same command again:

* docker run -t -i --volumes-from datacontainer ubuntu /bin/bash

This time the hi file is already there:

* cat /tmp/hi

You should see:

Output of cat /tmp/hi

I'm not going anywhere

You can add as many --volumes-from flags as you'd like (for example, if you wanted to assemble a container that uses data from multiple data containers). You can also create as many data volume containers as you'd like.

The only caveat to this approach is that you can only choose the mount path inside the container (/tmp in our example) when you create the data volume container.

## Sharing Data Between the Host and the Docker Container

The other common use for Docker containers is as a means of sharing files between the host machine and the Docker container. This works differently from the last example. There's no need to create a "data-only" container first. You can simply run a container of any Docker image and override one of its directories with the contents of a directory on the host system.

As a quick real-world example, 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

This run command is a little different from the ones we've used so far, so let's break it down piece by piece:

* -v ~/nginxlogs:/var/log/nginx — We set up a volume that links the /var/log/nginxdirectory from inside the Nginx container to the ~/nginxlogs directory on the host machine. Docker uses a : to split the host's path from the container path, and the host path always comes first.
* -d — Detach the process and run in the background. Otherwise, we would just be watching an empty Nginx prompt and wouldn't be able to use this terminal until we killed Nginx.
* -p 5000:80 — Setup a port forward. The Nginx container is listening on port 80 by default, and this maps the Nginx container's port 80 to port 5000 on the host system.

If you were paying close attention, you may have also noticed one other difference from the previous runcommands. Up until now we've been specifying a command at the end of all our run statements (usually/bin/bash) to tell Docker what command to run inside the container. Because the Nginx image is an official Docker image, it follows Docker best practices, and the creator of the image set the image to run the command to start Nginx automagically. We can just drop the usual /bin/bash here and let the creators of the image choose what command to run in the container for us.

So, we now have a copy of Nginx running inside a Docker container on our machine, and our host machine's port 5000 maps directly to that copy of Nginx's port 80. Let's use curl to do a quick test request:

* curl localhost:5000
* You'll get a screenful of HTML back from Nginx showing that Nginx is up and running. But more interestingly, if you look in the ~/nginxlogs folder on the host machine and take a look at theaccess.log file you'll see a log message from Nginx showing our request:
* cat ~/nginxlogs/access.log

You will see something similar to:

Output of `cat ~/nginxlogs/access.log`

172.17.42.1 - - [23/Oct/2015:05:22:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.35.0" "-"

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.

**Docker Data Backup, Restore or Migrate**

Another useful function we can perform with volumes is use them for backups, restores or migrations. We do this by using the --volumes-from flag to create a new container that mounts that volume, like so:

$ docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

Here we’ve launched a new container and mounted the volume from thedbstore container. We’ve then mounted a local host directory as /backup. Finally, we’ve passed a command that uses tar to backup the contents of thedbdata volume to a backup.tar file inside our /backup directory. When the command completes and the container stops we’ll be left with a backup of our dbdata volume.

You could then restore it to the same container, or another that you’ve made elsewhere. Create a new container.

$ docker run -v /dbdata --name dbstore2 ubuntu /bin/bash

Then un-tar the backup file in the new container’s data volume.

$ docker run --rm --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"

You can use the techniques above to automate backup, migration and restore testing using your preferred tools.

**Docker Swarm and Node**

Docker Swarm is a clustering and scheduling tool for [Docker](http://searchenterpriselinux.techtarget.com/definition/Docker) containers. With Swarm, IT administrators and developers can establish and manage a [cluster](http://searchexchange.techtarget.com/definition/cluster) of Docker host systems, or nodes, as a single virtual system

Clustering is an important feature for [container technology](http://searchservervirtualization.techtarget.com/definition/container-based-virtualization-operating-system-level-virtualization) because it creates a cooperative group of systems that can provide redundancy if one or more nodes fail. Clustering also provides the ability to add or subtract container iterations, also called scalability, as computing demands change

Swarm uses scheduling capabilities to ensure that there are sufficient resources for distributed containers. Swarm assigns containers to underlying nodes and optimizes resources by automatically scheduling container workloads to run on the most appropriate host. This provides basic workload balancing for containerized applications, ensuring containers are launched on systems with adequate resources while maintaining necessary performance levels.

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](http://whatis.techtarget.com/definition/instance) and multiple replica instances in case the primary instance fails.

**Docker Discovery Services**

Docker Discovery Service is used to connect and communicate with the host nodes and maintains a list of IP’s in that cluster and shares it with the swarm manager. The Swarm Manager uses this list to assign the tasks to the nodes based on the scheduler.Multiple service discovery managers are [supported](https://docs.docker.com/swarm/discovery/) by Docker Swarm. These managers track registered services, members, and sessions between replicas of themselves. Swarm Joins and Swarm Managers connect with these instances and the cluster of Service Discovery Managers handles the rest (including outages and adding/removing Swarm Nodes while in operation).

**Docker Swarm Scheduling Strategy**

Docker Swarm Strategy is used to rank the nodes that are connected in the cluster. The ranking is based on the computing strategy that we give. Whenever a new container is up, it will be assigned to a node which has the highest ranking based on the strategy. Swarm Strategy tells how the nodes should be used.

Swarm uses three different strategies to determine which nodes each container should run on:

* **Spread** -- Acts as the default setting and balances containers across the nodes in a cluster, based on the nodes' available CPU and RAM, as well as the number of containers it is currently running. The benefit of the Spread strategy is that if the node fails, only a few containers are lost.
* **BinPack** -- BinPack schedules containers to fully use each node. Once a node is full, it moves on to the next in the cluster. The benefit of BinPack is that it uses a smaller amount of infrastructure and leaves more space for larger containers on unused machines.
* **Random**– This strategy chooses a node at random.

**Docker Swarm Scheduling Filter**

Swarm filter is a subset of Swarm Strategy. Swarm filter tells how to choose the node for running the container. Swarm filters are divided into two types. They are Node Filters and Container Configuration Filters

**Container Configuration Filters:**

* **Affinity**– To ensure containers run on the same network node, the Affinity filter tells one container to run next to another based on an identifier, image or label.
* **Port** – With the Port filter, ports represent a unique resource. When a container tries to run on a port that's already occupied, it will move to the next node in the cluster.
* **Dependency** – When containers depend on each other, this filter schedules them on the same node.

**Node Filters:**

* **Constraint** – Also known as node tags, constraints are [key/value pairs](http://searchenterprisedesktop.techtarget.com/definition/key-value-pair) associated to particular nodes. A user can select a subset of nodes when building a container and specify one or multiple key/value pairs.
* **Health** – In the event that a node is not functioning properly, this filter will prevent scheduling containers on it.

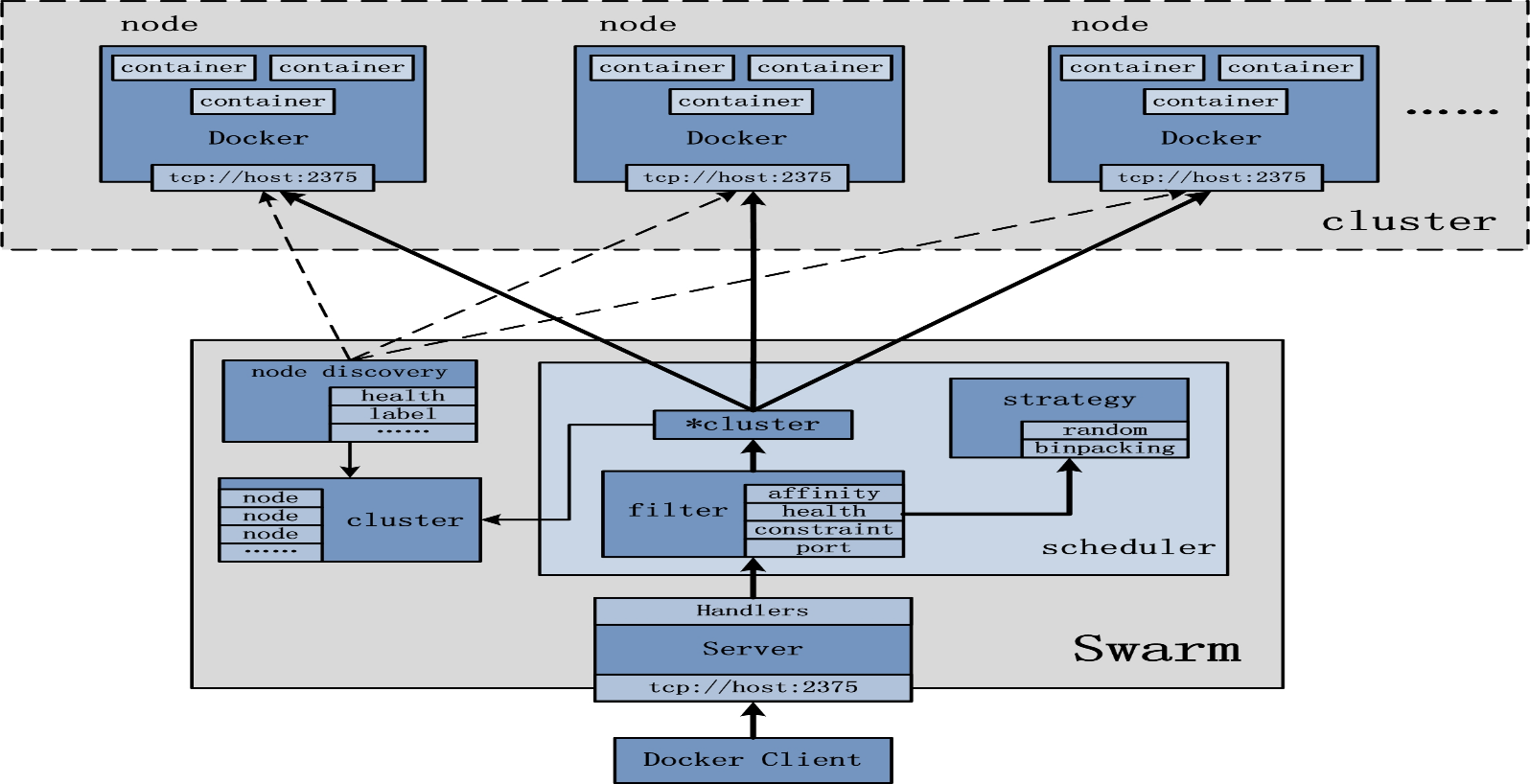
Docker Swarm uses the standard Docker API, which means it can support other tools, such as Dokku, Docker Compose, Krane and Jenkins.

Need to know terminologies:

1. **Docker Daemon** (daemon) – A process that handles container management on a single host or vm.  
2. **Swarm Join** (join) – A process that handles registering a single host with a Service Discovery Manager and exposing the host’s Docker Daemon as an available service.  
3. **Swarm Node** (node) – This is not an official Docker term but a logical association for a host machine in the Swarm that is only responsible for running containers. At a minimum, a Swarm Node needs to have the Docker Daemon and the Swarm Join running on it.  
4. **Swarm Manager** (manager) – The service a user uses for managing containers across the registered Swarm Node(s). This is the endpoint for interfacing with a Swarm environment.  
5. **Swarm Cluster Token** (token) – A Docker Swarm can be deployed without running your own Service Discovery Managers; however, this means the token will be shared over an encrypted connection with Docker Hub. This is a useful way for getting started with a single host environment. Please consider that this means your token and environment are now dependent on a third party. It is not recommended for production use. If you want to run a distributed Swarm using a token, then all Swarm Joins and Swarm Managers need to specify the **token://<your token string>** instead of **consul://<consul uri>/<swarm name e.g. myswarm>** . The advantage is simplicity when you do not need to integrate with a service discovery tool to get going. The disadvantage is each Swarm Node in the Cluster needs to know the token after a new Swarm Cluster has been created

6. **Overlay Network** – This is the new native Docker networking type for deploying containers that are linked only with the other Containers on the network. This is utilizing VXLAN technology and is really interesting for its ability to separate containers (even on the same node) as well linking containers across multiple Swarm Nodes. The containers deployed with an overlay network can see the other linked containers on the network by using the entries defined in the /etc/hosts file in the container

**Docker Swarm Architecture:**



Docker Swarm has a **Manager**, a pre-defined Docker Host, and is a single point for all administration. Swarm manager orchestrates and schedules containers on the entire cluster, and can be configured in [High Availability](https://docs.docker.com/swarm/multi-manager-setup/). The containers are deployed on **Nodes** that are additional Docker Hosts.

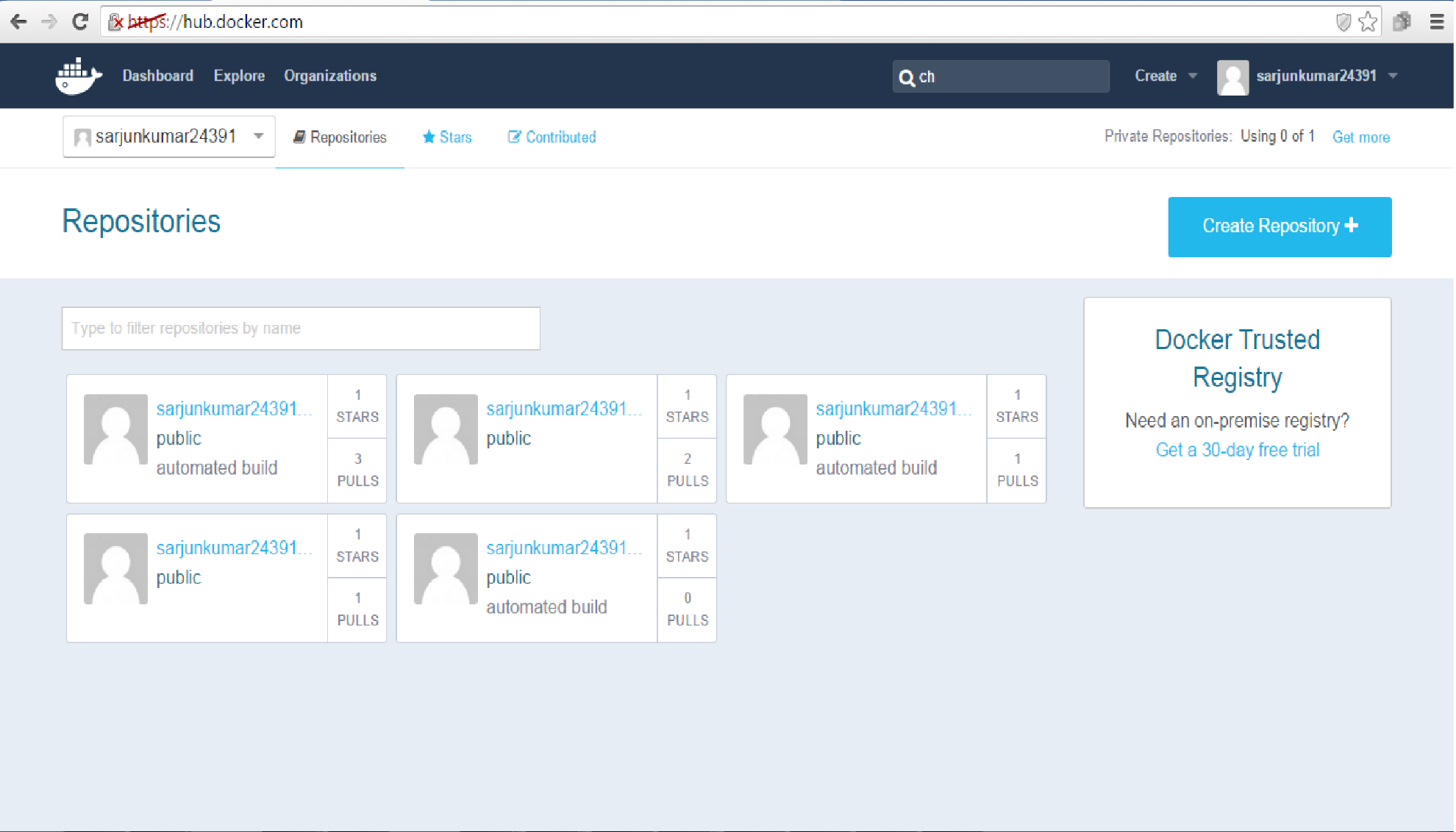
Swarm talks to a hosted **Discovery Service** that maintains a list of IPs in your cluster.

**Docker Compose**

This is a tool for defining and running multi-container Docker applications. This tool takes a simple configuration yml file and deploys the containers like a manifest. It is the deployment tool for utilizing the new multi-host networking features that allow the Swarm Nodes to have applications running across distributed hosts. Compose also handles placement strategies for ensuring your containers are distributed evenly (or not) across the Swarm Nodes. This allows for container redundancy at the host level which is good for production resiliency.

**Docker HUB**

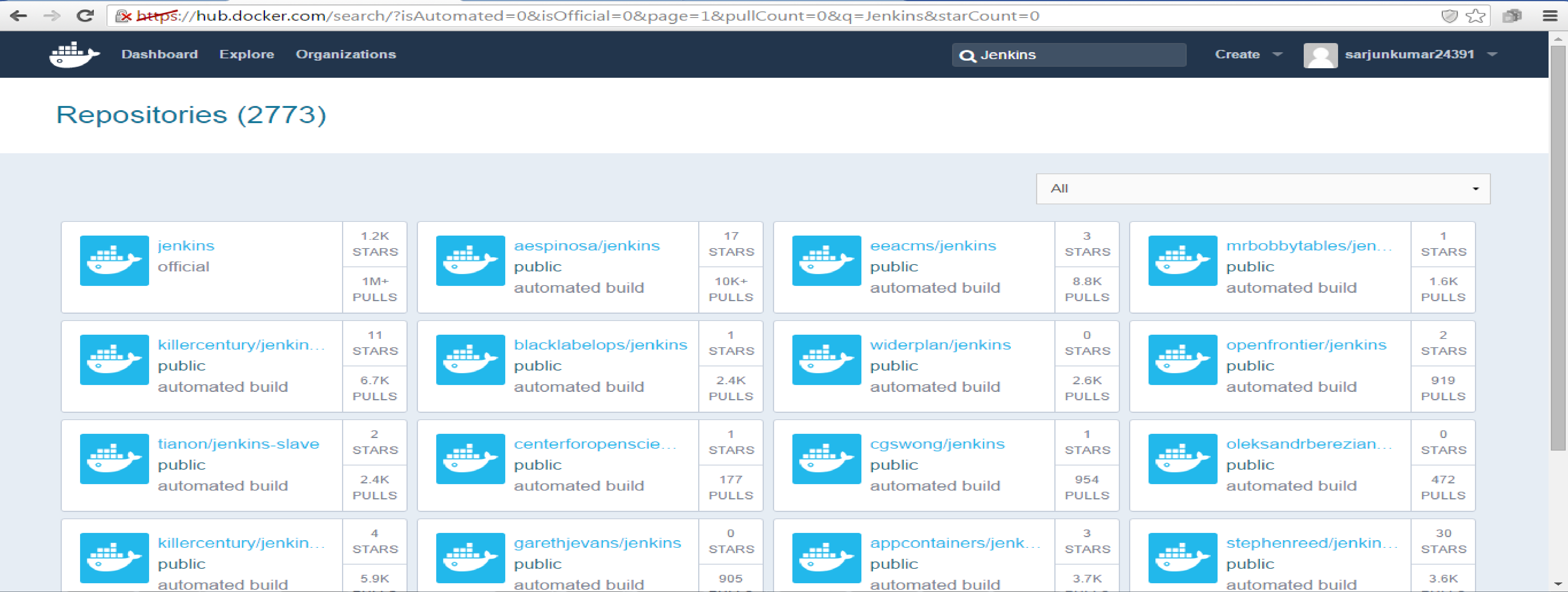
Docker HUB is a public repository from where we can pull images, push updated images, trigger auto build and to create our own repository.



**Docker Repository**

Docker Repository will have all the types of images which are public, officially trusted by Docker, Automated Build Images. Images can be searched in Docker Repository based on the following criteria.

* 1. Public
  2. Automated Build
  3. Official (Officially approved images by Docker)
  4. Starts (Images with most number of starts)



**Docker Automated Build**

Using Automated Build we can connect to the SCM and trigger a build to create an image. Docker HUB can be integrated with “github”. From github the source application can be pulled and image will be created in Docker HUB based on the Dockerfile in the github source code. If the code doesn’t have a Dockerfile or the repository is not accessible, then image cannot be built.

