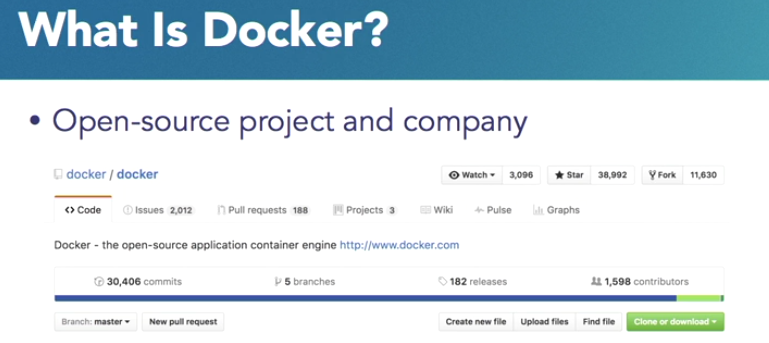
A container is a virtual walled environment for your application. It’s literally a ‘container’ inside the host OS. Thus, your application works like it is in its own self-contained environment, but it’s sharing operating system resources of the host computer.  Because of this, containers are more resource efficient than full blown virtual machines.



Docker is an open-source project that started about three years ago. It used to be called Dark Cloud, and the main idea behind that project was really to build containers for software applications.

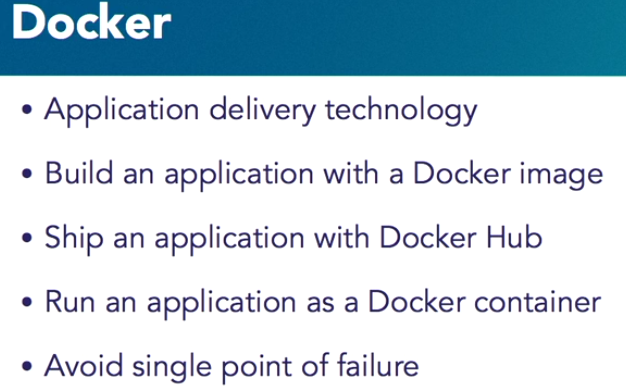
Over a period, the company Dark Cloud was changed to the name Docker and now is an open-source project, and a company literally on GitHub. So, it's <http://www.github.com/docker>

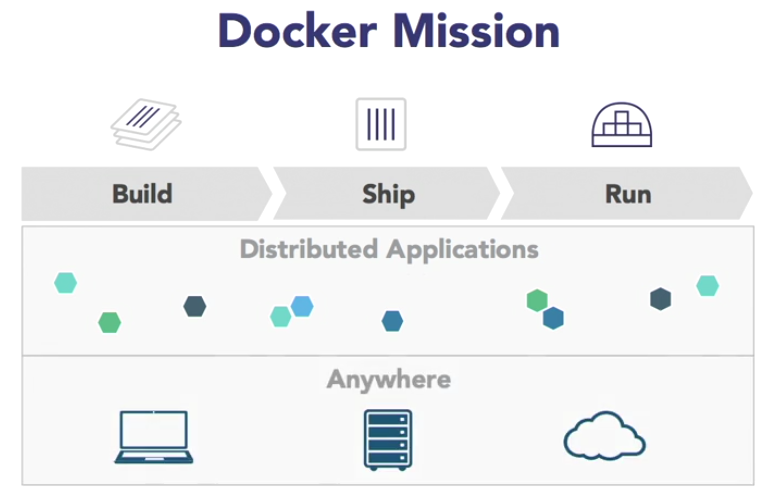
It as an application deployment, or application delivery technology. It is used to build, deploy, and run containers for software applications.

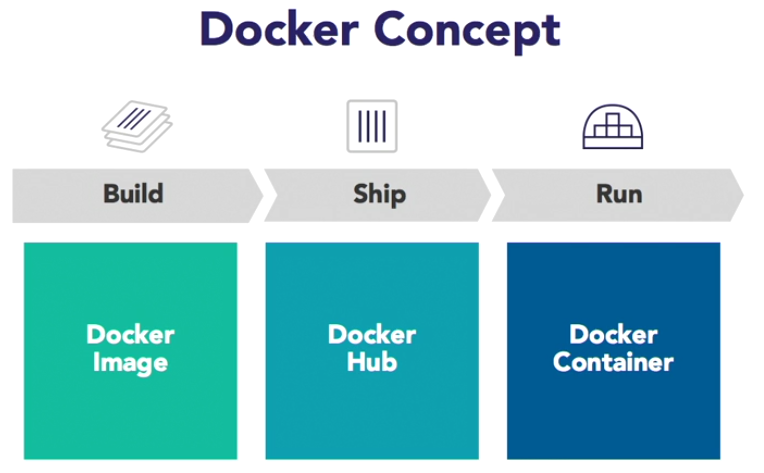
What Docker gives us is a standard way to package our software, deploy the software, and run the software.

What goes in the container, which is called as a container, it is completely our application developers responsibility, but Docker just gives us that standard format by which we can easily do this.

So, what is Docker Mission at a very high level? Well, essentially it wants us to be able to build, ship, and run distributed applications.







when we are building an application, we have multiple components to the application.

1. Web server.
2. Application server
3. Database server.
4. Caching layer.
5. Messaging server
6. Multiple components.

And, typically, we want to run these applications in a distributed way. Why would we run in a distributed? Because we want to avoid a single point of failure. And for a web server, or a database server, or an application server, we will run multiple instances of those. And we want to run multiple instances again, to avoid a single point of failure.

Also, to be able to meet the scalability needs of our application. So that in case more database concurrent accesses are required we can do the horizontal sharing as some of the databases allow.

In terms of build, what Docker defines is a standard Docker image. When we build our application, essentially what we're saying is, "Here is what my application component going to look like, and here is what the configuration of the application looks like, and I'm going to package this together, and I'm going to call that as Docker Image."

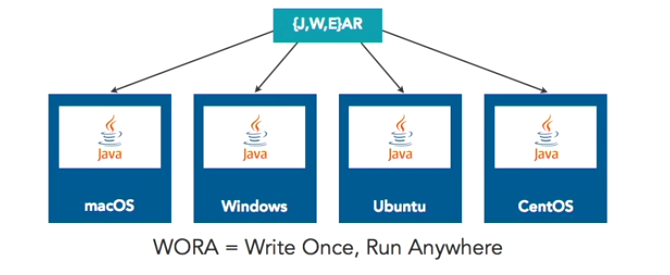
Now once we have built a Docker image, we want to share it with somebody else, or we want to run it in production, or we want to give it to somebody else who can then exactly try the same image.

For that, the shipment concept comes in. By default, that is Docker Hub, or on <https://www.hub.docker.com>

So once the image is built we can share the image using Docker Hub, and once we are at Docker Hub, we can look at what the image looks like, and then we can download the image from Docker Hub.

**Docker is an application delivery technology.**

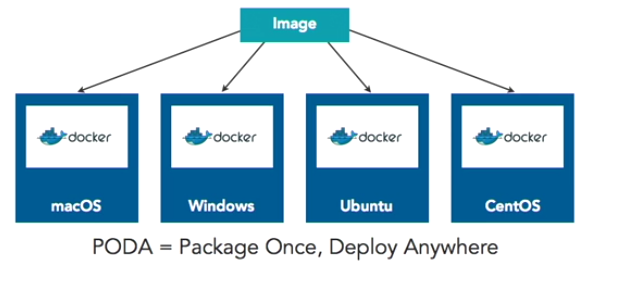
It gives us the ability to build an application using Docker image, ship an application using Docker Hub, or registry is another term for shipping, and then finally, be able to run the application as a Docker container, potentially on multiple hosts so that there is no single point of failure and even multiple instances of a container just to avoid a single point of failure as well.



As a Java developer the concept of WORA: Write Once, Run Anywhere. How does that work?

I have a Java source file. We take the Java source file and give it to the Java compiler. The Java compiler then takes it, generates a .class file. Then, we have different tools like Maven, Gradle, so on, so forth. We take those class files and pack them into a JAR or a WAR or an EAR file.

So, we've taken the Java source file, compiled into class, and then packaged it into a JAR, WAR, or EAR. Now, on a variety of operating systems, whether it's Mac or Windows or Ubuntu or any flavor of Linux for that sake, there is a JVM, or Java Virtual Machine that is running. That JVM now understands the format of this class file, which is a Java bytecode. And, that's what gives us the power of Write our source code once, and the JVM will understand the underlying bytecode and it will run out of the box.



Docker has this capability of creating an image. Images where all our application configuration data is put together.

Just like our JVM understands the class format, the Docker understands the image format. It takes that image format and runs the container for us.

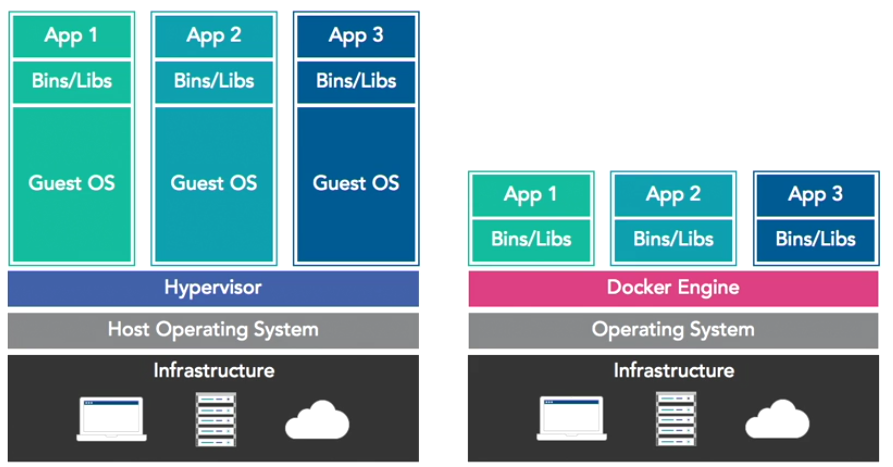
PODA which is Package Once, Deploy Anywhere, is very similar to WORA, but it's not exactly similar because based upon what is the base operating system that we are using for an image, it could run only on that operating system.

**Virtual Machine and Docker**

**On the left**, what we see is a very classical virtual machine environment. So, we have our infrastructure on-premise, cloud, laptop, whatever it is. We have a host operating system running, which is whatever our base operating system is.

On a host operating system, we would run a hypervisor. Let's say, a type two hypervisor like Virtual Box or anything. Let's say my host operating system is Windows or Mac or Linux, and on top of that, I went on a different operating system. For example, let's say my host operating system is OS X and, on top of that, I want to run a Windows operating system. So, for that, I would need to run a type two hypervisor and then, using Virtual Box, I can create a Windows VM on a Mac OS X.

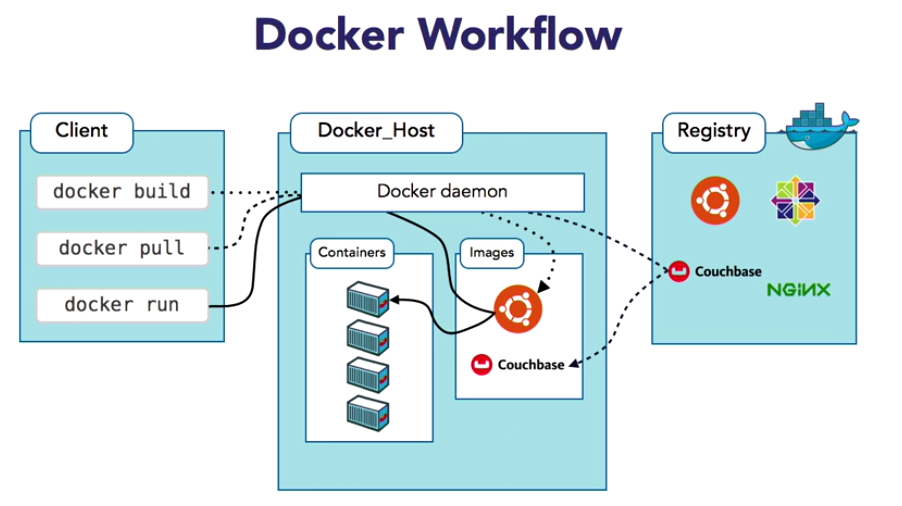
Now, if I want to build an application around an application within Windows, then I would have to packet my application as a full-blown operating system sitting over there. Very basic needs are being filled by the underlying host operating system. So, on a given infrastructure, if we have multiple virtual machines running, there's a limit because each operating system has its own memory requirements, CPU requirements, et cetera. So, we are stalled by the capacity of the underlying operating system.



**On the right-hand side** we have same underlying infrastructure. This is not a host operating system. It's just a simple operating system on which our Docker engine is running. Now, the Docker engine has a whole bunch of images. In that image, instead of a full-blown operating system, we have a minimal operating system. Docker relies upon basic Linux capabilities like cgroups and namespaces.

So, instead of having one gigabyte Ubuntu guest operating system on a typical virtual machine, I can use 120 megabytes Ubuntu base operating system on the Docker side and then packet my application in there. Everything else is provided by Docker engine. What that gives you? Well, first, that gives us a much smaller image, but because of the smaller image and lesser requirements, we can have much more densely packed. You can have agility of the application.

You can have the scalability of the application achieved much more easily. So, essentially, **Docker which is an application delivered technology**, **is very different from a virtual machine technology**. So, I think understanding the concept that how PODA matches with WORA, and how Docker is different from virtual machines, is fundamentally important.



The Docker client is configured to talk to a Docker host. It could be a single docker host, or it could be a cluster of Docker hosts. But the important part to understand is client is a "dumb client". It's a stateless client. There is no state on the client at all. The client says, "I want to run a container", which is a Docker run command.

Now, the client is configured to talk to Docker Host. It gives the command to Docker Host, on Docker Host, there is a Docker daemon running, which is listening for that command. The client gives the command "docker run", but if you have to look under the hood, it is basically a REST API call, which goes as a REST API and Docker daemon is listening on a specific port for that REST request. It understands that request. So, the client said "docker run", okay? Docker says, "In order to run a container I need to have the image". By default, Docker daemon, or Docker Host is configured to talk to a registry, which is Docker hub, which is <https://hub.docker.com/>

So, it queries the registry, says, "Hey do you have this image? Because I need to run this container". Let’s say you want to run a simple Nginx container or a simple Couchbase container, or an Ubuntu container. So, it queries the registry. Registry is where anybody can publish a Docker image.

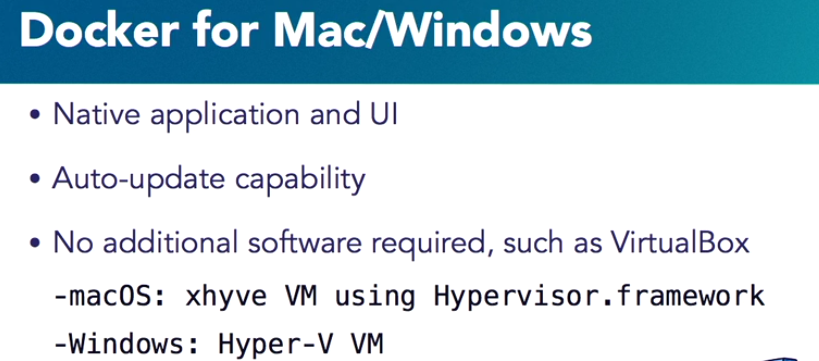
So now, Docker Host is saying, "I want to download an image". It downloads the image on the Docker Host. Remember we said Client is stateless? The state is maintained on Docker Host. We can download as many images on Docker Host via the client. The client gives a command, the image is stored on the Docker host. The image is downloaded once, but we can run as many instances of the container.

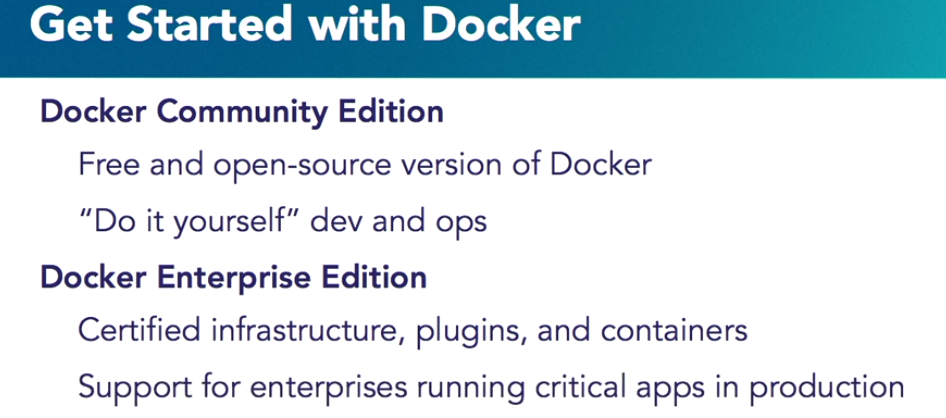
So, let’s think about the flow once again. The client makes a request, docker run, a container, goes to Docker Host. Docker Host says, "I don't have the image. If I have the image I will run the container. If I don't have the image I will download the image from the Docker hub, and once the image is downloaded I will run the container. All this happens behind the scenes.

In our typical day-to-day workflow client makes a request to the host, which downloads an image from the registry, and runs the container.

**OS Specific Docker Installation:**

<https://docs.docker.com/install/>





**Running the First Docker Container:**

docker container run **-it** jboss/wildfly

I want to run the container in the interactive mode and I want to assign a TTY to it. And the reason I would like to assign a TTY to it so that is when I press something on my command keyboard here it goes to the underlying container. So, what I'm doing is, Docker container run jboss/wildfly run it in interactive mode. Now of course it says, the image is not available. So, as you can see it's downloading the image from the Docker hub. By default, it's configured for the Docker hub and that's where it's downloading the image.

**To run in detached mode:**

docker container run **-d** jboss/wildfly

**To check list of containers:**

docker container ls

**To stop the container:**

docker container stop 47bb8a75ead9🡪Container Id

**To check all containers irrespective of it running or not:**

docker container ls -a

**To remove the container:**

docker container rm 47bb8a75ead9

**To run the container by giving user defined name:**

docker container run -d --name **myweb** jboss/wildfly

**To forcefully remove an active controller without stopping:**

docker container rm -f myweb

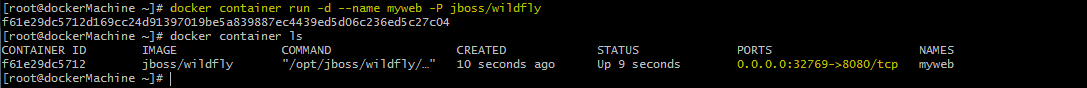
**Bash Mode:**

docker container run -it --name myweb jboss/wildfly bash



**Expose the port of docker so that, we can run it in host OS:**

docker container run -d --name myweb -P jboss/wildfly





So, just by adding that -P, it picked up a default port for us, and our WildFly is now accessible.

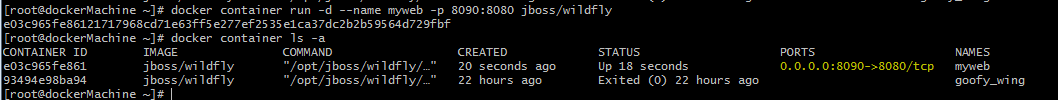
**To run the container in a specific port**

docker container run -d --name myweb -p 8090:8080 jboss/wildfly

8090 in the host machine and 8080 in the container.

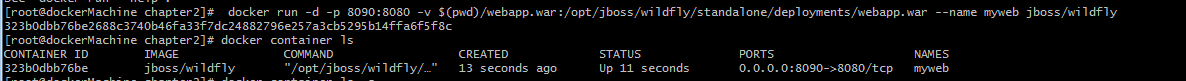
P is used for assigning user defined port.

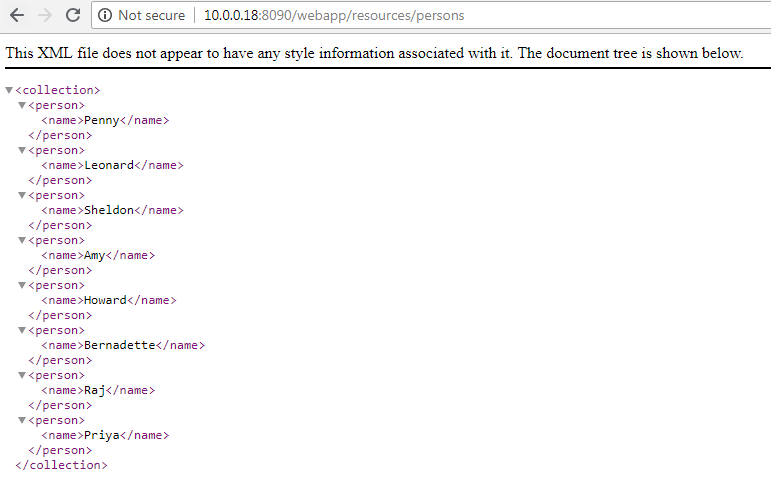
P Random port



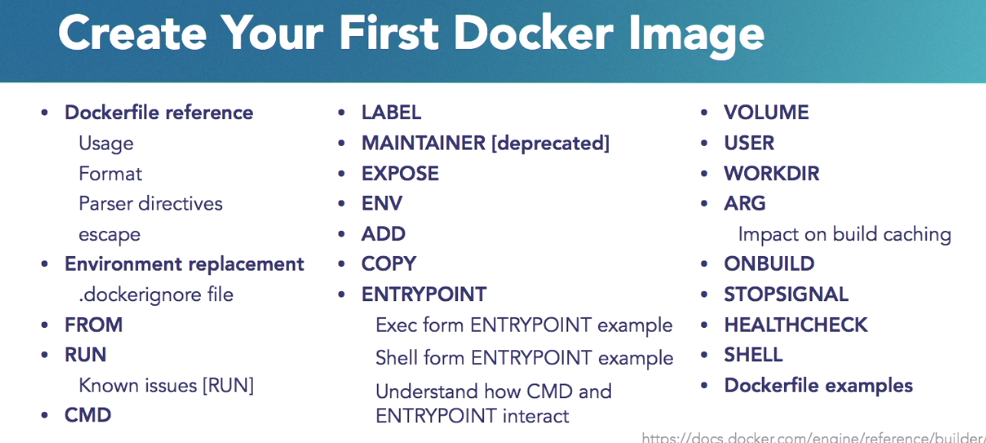
**Volume Mapping:**

docker run -d -p 8090:8080 -v $(pwd)/webapp.war:/opt/jboss/wildfly/standalone/deployments/webapp.war --name myweb jboss/wildfly





**Create my own Image:**



**Building the Docker Image:**

**Step1**: Create a Directory

**Step2**: Go to that Directory

**Step3**: Create a Docker file

FROM ubuntu

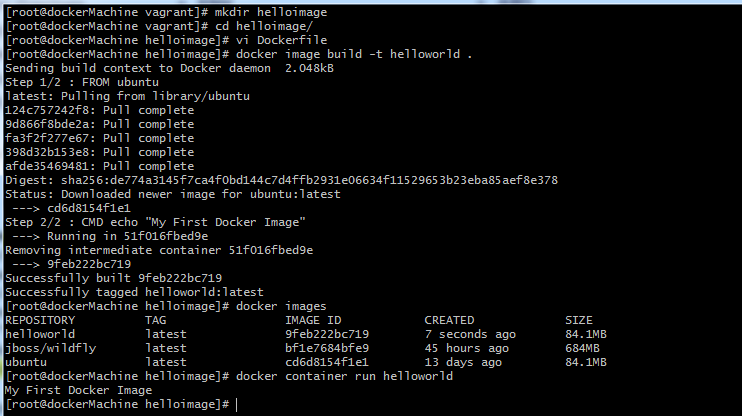
CMD echo “Hello World”

**Step4**: Create the image with name helloworld

docker image build -t helloworld .

-t: Tag Name

. : Current directory



**Creating First Java Docker image:**

**Step1**: Create a Directory

**Step2**: Go to that Directory

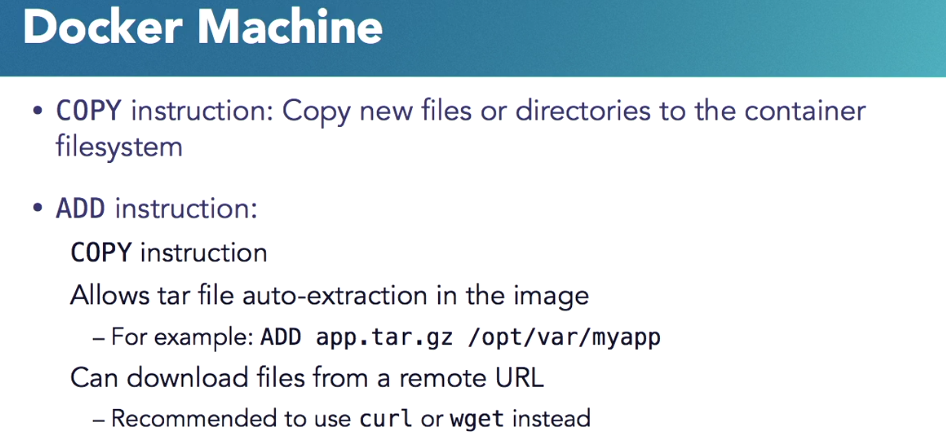
**Step3**: Create a Docker file

FROM openjdk

java -version

**Step4**: Create the image with name hellojava

docker image build -t hellojava .



**Copy files to Image:**

**Step1**: Create a Directory

**Step2**: Go to that Directory

**Step3**: Create a Docker file

FROM jboss/wildfly

COPY webapp.war /opt/jboss/wildfly/standalone/deployments/webapp.war

**Step4**: Create the image with name helloweb

docker image build -t helloweb .

**Step5**: docker container run -p 8090:8080 -d helloweb

Now we can run our own image in 8090 ports.

**How to Install JAVA 8 on CentOS/RHEL 7/6 and Fedora 28-23**

<https://tecadmin.net/install-java-8-on-centos-rhel-and-fedora/>

**How to enable proxy in Maven:**

<https://www.mkyong.com/maven/how-to-enable-proxy-setting-in-maven/>

**Creating a Jar file and running the Jar file from Docker Image**

**Step1**: Create a Directory

**Step2**: Go to that Directory

**Step3**: Create a Docker file

FROM openjdk:jdk-alpine

COPY myapp/target/myapp-1.0-SNAPSHOT.jar /deployments/

CMD java -jar /deployments/myapp-1.0-SNAPSHOT.jar

Where myapp contains a single java file.

**Step4**: Install maven and create the jar file inside myapp/target directory

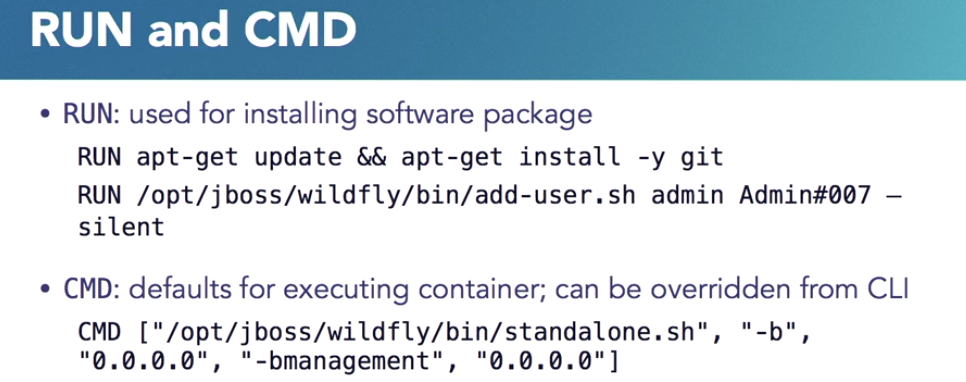
mvn -f myapp/pom.xml clean package

**Step5**: Create the image with name helloweb version 2

docker image build -t helloweb:2 .

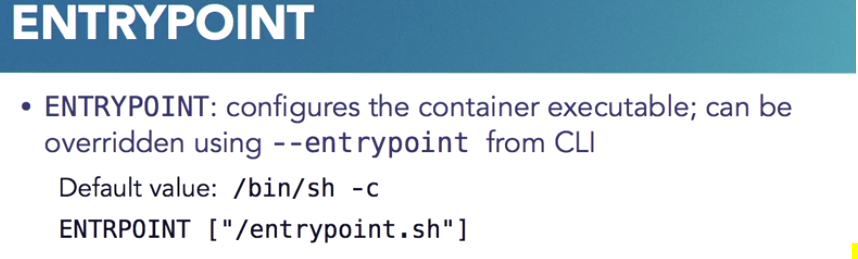
**Step6**: docker container run helloweb:2

It will print the content of main class

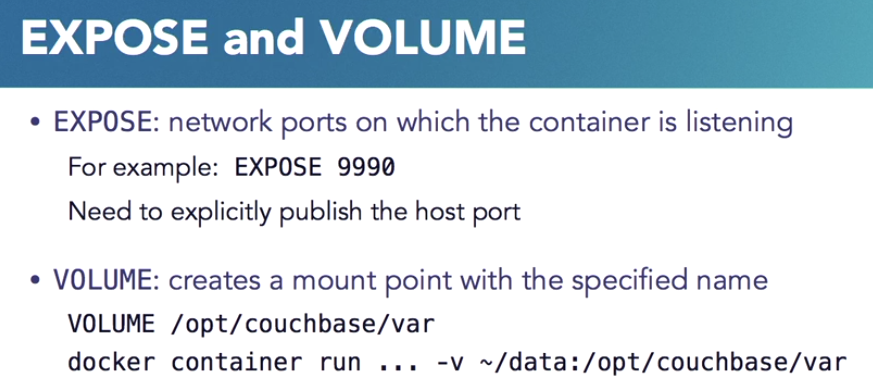


RUN is typically used for installing software package or running a script from your container file system. So typically, you'd say apt-get update or apt-get install and then you will update your operating system.

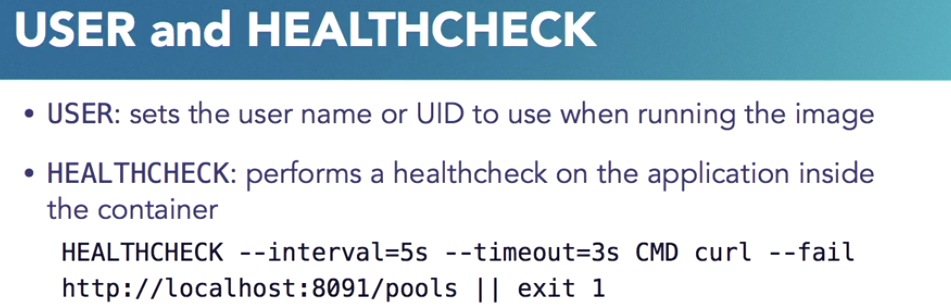
CMD, as we talked about is the default for executing a container. And, this is the default command. We talked about how an image can have multiple CMD but only the last CMD command is the one that is into effect.



Docker container, when it starts, it uses an ENTRYPOINT. The default ENTRYPOINT is, /bin/sh, it runs out of Shell basically. You can override that, using the ENTRYPOINT instruction and I'll, say, give it a script, and it'll run that script, and in that script, you can do whatever you want to do. And if you want to override this from the CLI, then you have a --entrypoint option which allows you to override as well



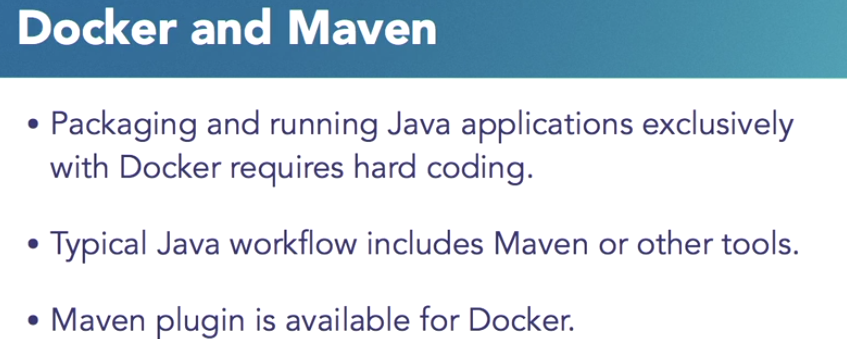
When we are building an application, and, in that application, you are publishing arrest API on a certain port, or on a network port. Then, you need to expose that port. So, for example, in our WildFly image, let's say we want to expose an administrative port as well, then we'll have to add, EXPOSE 9990. Expose only exposes the port from the container. But, the important part is to understand, when you are running the container, at that point of time you still need to say, Docker run either -P or -p to publish the host port.



In this case what it's saying is, perform the health check every five seconds, which is --interval, then we are saying the timeout is three seconds. That, if in three seconds it does not return a result, mark the container as unhealthy and then we are saying, what is the command that I need to run? And in this case, what I'm saying is, just ping localhost:8091 which is the administrative port of Couchbase and Couchbase also has a rest API.

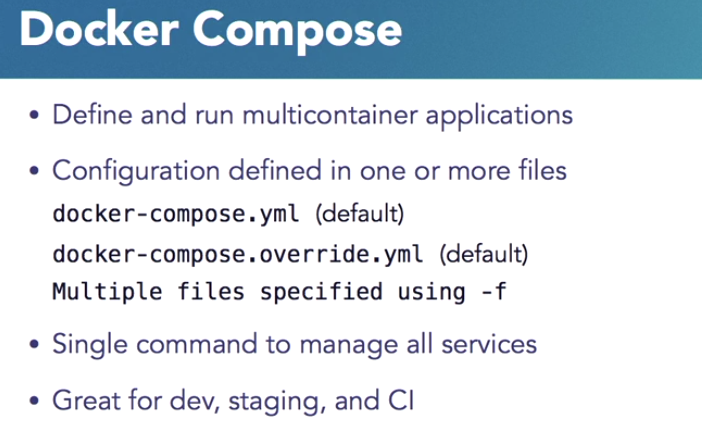
So, I'm saying, /pools, ping that rest API and make sure that the database pools are available, and if it fails, then you exit with status 1. So, essentially what will happen is, every five seconds the Docker container, or the Docker engine will make a ping to the container. If the pool exists, then it's good. If the pool doesn't exist, or if for some reason, the pool is not accessible within three seconds, it'll mark the container as unhealthy. And, if it's unhealthy, then the Docker scheduler might kick in and reschedule the container somewhere else.

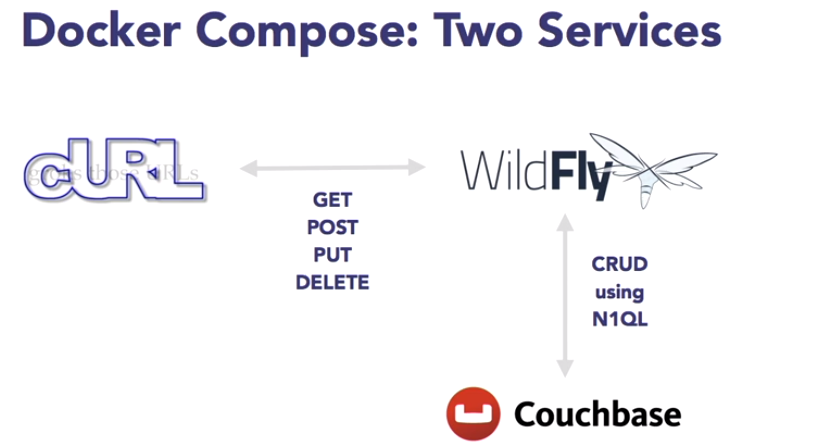
**Docker and Maven:**

****

**Running a Spring Boot Jar in Docker:**

[**https://www.callicoder.com/spring-boot-docker-example/**](https://www.callicoder.com/spring-boot-docker-example/)

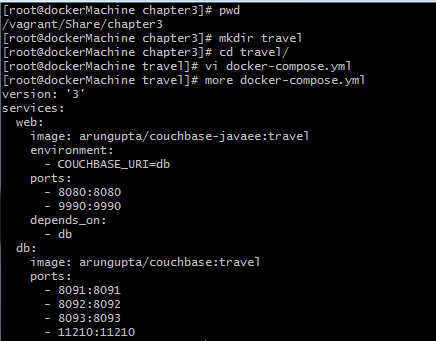




 Let's look at how we can create a multi-container application in this case. And for our use case, what we're going to do is we're going to use WildFly, which we used in a previous application as the main application. In that WildFly, we have an application to deploy. That deployed application will be talking to Couchbase, which is a new SQL document database.

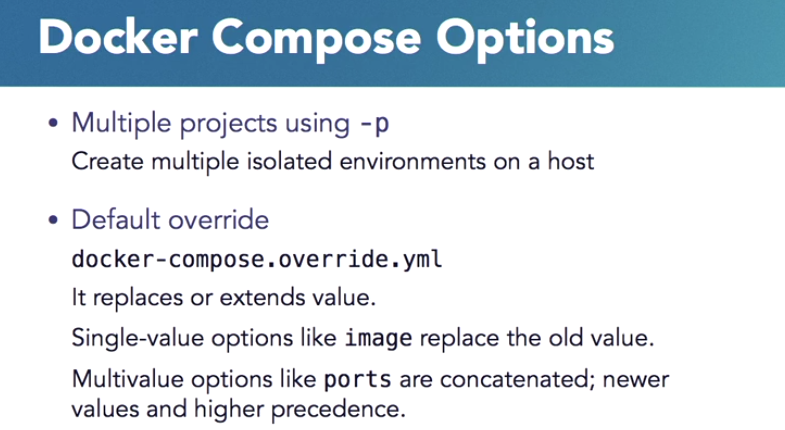
And it'll be storing that data into Couch space, reading the data from Couchspace, using N1QL. N1QL is a Couchbase programming language, which allows you to run SQL queries on JSON. So that's a typical business logic we're going to do. So, we're going to do CRUD on Couchbase document using N1QL from WildFly to Couchbase. And then because WildFly is our front end, we're going to expose a REST API that can then be invoked using CURL. I know we're using CURL as an example, but it could very well be any application that can consume a REST API.

Ex:



**Content of yml file: This is just an example. We need to have all the images defined in the YML file to run the file**

****

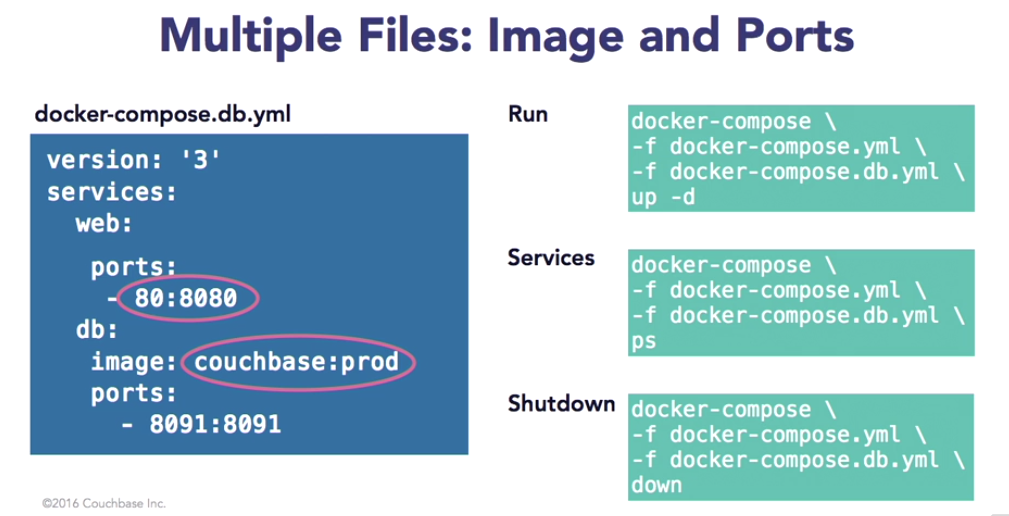


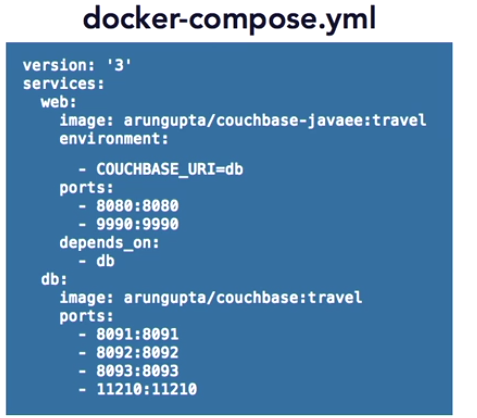
**To run the docker compose file:**

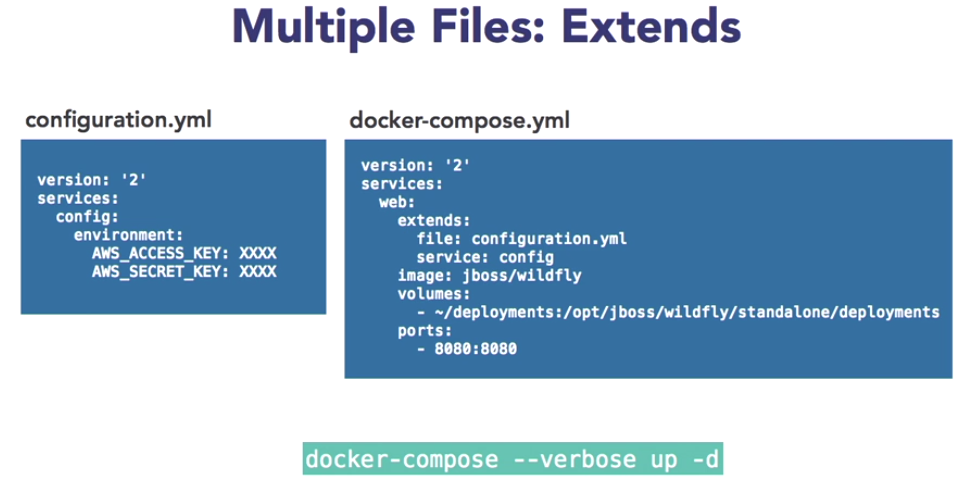
docker-compose up -d

**To stop:**

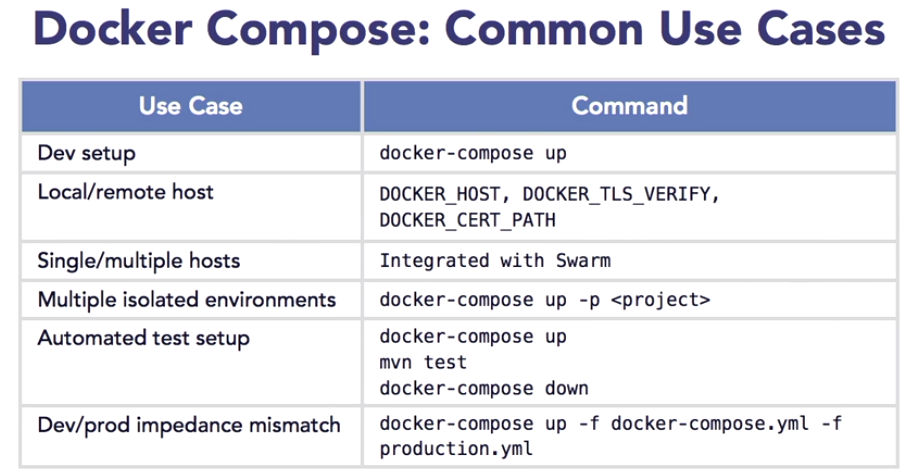
docker-compose down



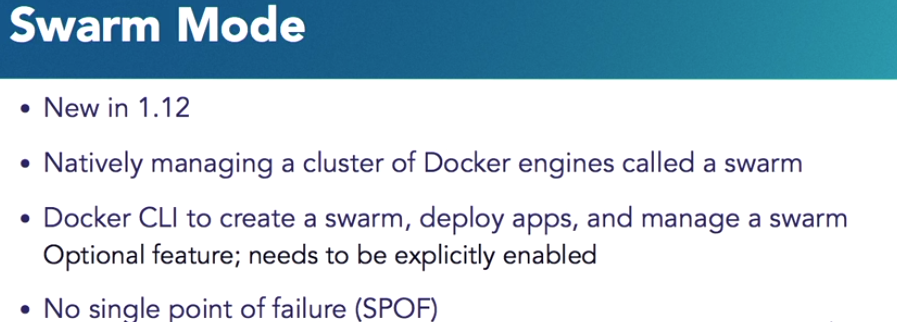


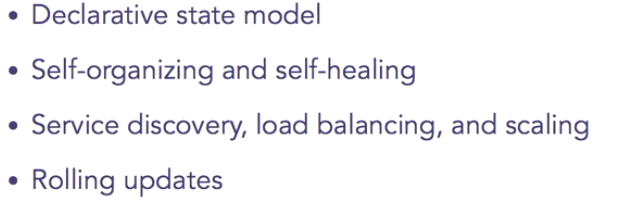


Imagine a new developer joining your team and you can't ask them to say install a JDK, install a WildFly server, install these libraries, do this Maven pull, do this checkout. Instead of doing all that setup, you can say, here is my docker-compose.yml, bring up, and your container's alive, and your application is fully functional. No downloading, installing, all of that. One of the options in Docker Compose is you can point it either to a host or to a cluster of host.



**Swarm Mode:**

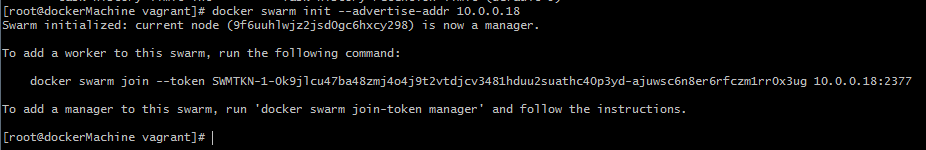




In any distributed architecture, it's fundamentally important to avoid a single point of failure. Swarm mode is a new mode that is introduced in Docker 1.12 and exactly allows you to do that.

Swarm Mode is natively managing a cluster of Docker engines together and a bunch of engines sitting around together is called a swarm.

**To initialize Swarm mode:**

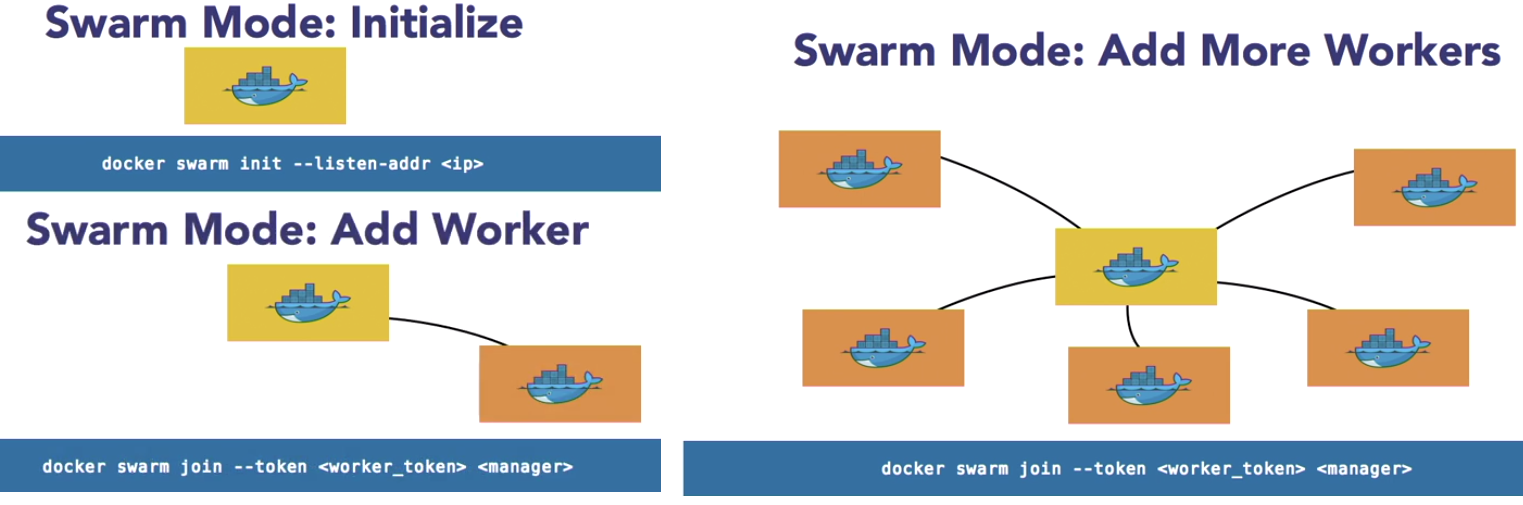


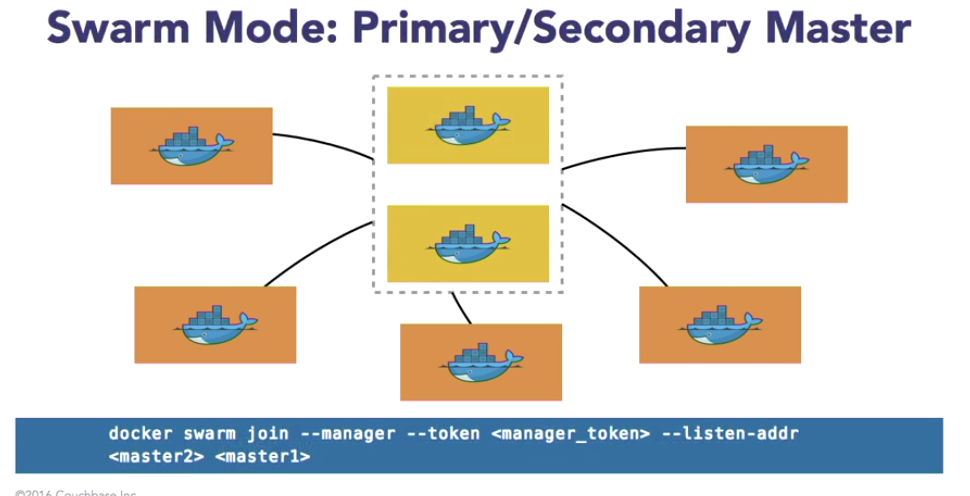
1. docker swarm init --advertise-addr 10.0.0.18
2. To know more details:

docker info

1. To leave the docker mode:

docker swarm leave -f

****



**To Create 3 manager and 3 workers node:**

****

**Run the above script**

**Before running the script:**

1. **Install docker-machine:**
2. **Install virtual box:**

yum install virtualbox

1. Installing Docker Engine:

wget [https://download.docker.com/linux/centos/docker-ce.repo -O /etc/yum.repos.d/docker.repo](https://download.docker.com/linux/centos/docker-ce.repo%20-O%20/etc/yum.repos.d/docker.repo)