**Docker as a container service**

* Docker is a container management service. The keywords of Docker are **develop, ship** and **run** anywhere.
* The initial release of Docker was in March 2013.
* The good thing about the Docker engine is that it is designed to work on various operating systems.

uname -a

Linux kernel version is 4.2.0-27 which is higher than version 3.8, so we are good to go

sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates

echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main”

| sudo tee /etc/apt/sources.list.d/docker.list

apt-cache policy docker-engine

apt-get update

sudo apt-get install linux-image-extra-$(uname -r)

linux-image-extra-virtual

sudo apt-get install –y docker-engine

sudo docker version

sudo docker info

Return Value

The output will provide the various details of the Docker installed on the system such as −

* Number of containers
* Number of images
* The storage driver used by Docker
* The root directory used by Docker
* The execution driver used by Docker

service docker status

service docker restart

service docker start --start docker service

service docker stop --stop docker service

Docker Image: is application or set of command/source code/instructions which can be executed on demand

--show list of images

$sudo docker images

--remove existing images

$sudo docker rmi imageid or name

--add image /download image (docker hub)

$sudo docker pull Jenkins

--run or execute images

$sudo docker image run <imagename>

**Test Hello-world**

$sudo docker run hello-world

The **run** command is used to mention that we want to create an instance of an image, which is then called a **container**.

**Test Centos**

sudo docker run centos –it /bin/bash

## Displaying Docker Images

To see the list of Docker images on the system, you can issue the following command.

docker images

sudo docker images -q

-return only image id

sudo docker inspect jenkins --inspect particular images

## Removing Docker Images

docker rmi ImageID

## Running a Container

* Running instance of image is called container

--show list of running container

$sudo docker ps : show all running container

$sudo docker ps –a : show all running (active) and inactive container

Container life cycle:

Start -> running -> Stop

* Kill
* Pause -> Unpause

Running of containers is managed with the Docker **run** command. To run a container in an interactive mode, first launch the Docker container.

sudo docker run –it centos /bin/bash

Then hit Crtl+p and you will return to your OS shell.

Used the Ctrl+P+Q command to exit out of the container.

## Listing of Containers

docker ps --running container

docker ps -a –all container

sudo docker history centos --Docker history

docker stop ContainerID --Docker stop

sudo docker rm ContainerID

docker stats ContainerID

docker pause ContainerID

docker unpause ContainerID

docker kill ContainerID

## Recap and cheat sheet

## List Docker CLI commands

docker

docker container --help

## Display Docker version and info

docker --version

docker version

docker info

## Execute Docker image

docker run hello-world

## List Docker images

docker image ls

## List Docker containers (running, all, all in quiet mode)

docker container ls

docker container ls --all

docker container ls -aq

**Custom Image**

## Define a container with Dockerfile

Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs.

### Dockerfile

Create an empty directory. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r reqs.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

This Dockerfile refers to a couple of files we haven’t created yet, namely app.py and requirements.txt. Let’s create those next.

## The app itself

Create two more files, requirements.txt and app.py, and put them in the same folder with the Dockerfile. This completes our app, which as you can see is quite simple. When the above Dockerfile is built into an image, app.py andrequirements.txt is present because of that Dockerfile’s COPY command, and the output from app.py is accessible over HTTP thanks to the EXPOSE command.

### requirements.txt

Flask

Redis

### app.py

from flask import Flask

from redis import Redis, RedisError

import os

import socket

# Connect to Redis

redis = Redis(host="redis", db=0, socket\_connect\_timeout=2, socket\_timeout=2)

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

try:

visits = redis.incr("counter")

except RedisError:

visits = "<i>cannot connect to Redis, counter disabled</i>"

html = "<h3>Hello {name}!</h3>" \

"<b>Hostname:</b> {hostname}<br/>" \

"<b>Visits:</b> {visits}"

return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=80)

Now we see that pip install -r requirements.txt installs the Flask and Redis libraries for Python, and the app prints the environment variable NAME, as well as the output of a call to socket.gethostname(). Finally, because Redis isn’t running (as we’ve only installed the Python library, and not Redis itself), we should expect that the attempt to use it here fails and produces the error message.

docker build -t friendlyhello .

docker run -p 4000:80 friendlyhello

Now let’s run the app in the background, in detached mode:

docker run -d -p 4000:80 friendlyhello

### Pull and run the image from the remote repository

From now on, you can use docker run and run your app on any machine with this command:

docker run -p 4000:80 username/repository:tag

**Dockerfile**

#Custom Imae

FROM ubuntu

MAINTAINER infotechvision@gmail.com

RUN apt-get update

RUN apt-get install –y nginx

CMD [“echo”,”Test Images”]

* The first line "#This is a sample Image" is a comment. You can add comments to the Docker File with the help of the **#** command
* The next line has to start with the **FROM** keyword. It tells docker, from which base image you want to base your image from. In our example, we are creating an image from the **ubuntu** image.
* The next command is the person who is going to maintain this image. Here you specify the **MAINTAINER** keyword and just mention the email ID.
* The **RUN** command is used to run instructions against the image. In our case, we first update our Ubuntu system and then install the nginx server on our **ubuntu** image.
* The last command is used to display a message to the user.

## docker build

This method allows the users to build their own Docker images.

### Syntax

docker build -t ImageName:TagName dir

### Options

* **-t** − is to mention a tag to the image
* **ImageName** − This is the name you want to give to your image.
* **TagName** − This is the tag you want to give to your image.
* **Dir** − The directory where the Docker File is present.

**Push to Hub**

[https://hub.docker.com](https://hub.docker.com/)

1. In a terminal window, set the environment variable **DOCKER\_ID\_USER** as *your username* in Docker Cloud.

This allows you to copy and paste the commands directly from this tutorial.

$ export DOCKER\_ID\_USER="username"

If you don’t want to set this environment variable, change the examples in this tutorial to replace DOCKER\_ID\_USER with your Docker Cloud username.

1. Log in to Docker Cloud using the docker login command.
   1. $ docker login

This logs you in using your Docker ID, which is shared between both Docker Hub and Docker Cloud.

If you have never logged in to Docker Hub or Docker Cloud and do not have a Docker ID, running this command prompts you to create a Docker ID.

1. Tag your image using docker tag.

In the example below replace my\_image with your image’s name, and DOCKER\_ID\_USER with your Docker Cloud username if needed.

$ docker tag my\_image $DOCKER\_ID\_USER/my\_image

1. Push your image to Docker Hub using docker push (making the same replacements as in the previous step).
2. $ docker push $DOCKER\_ID\_USER/my\_image
3. Check that the image you just pushed appears in Docker Cloud.

Go to Docker Cloud and navigate to the **Repositories** tab and confirm that your image appears in this list.

## docker tag

This method allows one to tag an image to the relevant repository.

### Syntax

docker tag imageID Repositoryname

### Options

* **imageID** − This is the ImageID which needs to be tagged to the repository.
* **Repositoryname** − This is the repository name to which the ImageID needs to be tagged to.

### Return Value

None

### Example

sudo docker tag ab0c1d3744dd demousr/demorep:1.0

## docker push

This method allows one to push images to the Docker Hub.

### Syntax

docker push Repositoryname

### Options

* **Repositoryname** − This is the repository name which needs to be pushed to the Docker Hub.

### Return Value

The long ID of the repository pushed to Docker Hub.

### Example

sudo docker push demousr/demorep:1.0

# Services

## About services

In a distributed application, different pieces of the app are called “services.” For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Luckily it’s very easy to define, run, and scale services with the Docker platform -- just write a docker-compose.yml file.

## Your first docker-compose.yml file

A docker-compose.yml file is a YAML file that defines how Docker containers should behave in production.

### docker-compose.yml

Save this file as docker-compose.yml wherever you want. Be sure you have [pushed the image](https://docs.docker.com/get-started/part2/#share-your-image) you created in [Part 2](https://docs.docker.com/get-started/part2/) to a registry, and update this .yml by replacing username/repo:tag with your image details.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repo:tag

deploy:

replicas: 5

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "4000:80"

networks:

- webnet

networks:

webnet:

This docker-compose.yml file tells Docker to do the following:

* Pull [the image we uploaded in step 2](https://docs.docker.com/get-started/part2/) from the registry.
* Run 5 instances of that image as a service called web, limiting each one to use, at most, 10% of the CPU (across all cores), and 50MB of RAM.
* Immediately restart containers if one fails.
* Map port 4000 on the host to web’s port 80.
* Instruct web’s containers to share port 80 via a load-balanced network called webnet. (Internally, the containers themselves publish to web’s port 80 at an ephemeral port.)
* Define the webnet network with the default settings (which is a load-balanced overlay network).

## Run your new load-balanced app

Before we can use the docker stack deploy command we first run:

docker swarm init

**Note**: We get into the meaning of that command in [part 4](https://docs.docker.com/get-started/part4/). If you don’t run docker swarm init you get an error that “this node is not a swarm manager.”

Now let’s run it. You need to give your app a name. Here, it is set to getstartedlab:

docker stack deploy -c docker-compose.yml getstartedlab

Our single service stack is running 5 container instances of our deployed image on one host. Let’s investigate.

Get the service ID for the one service in our application:

docker service ls

Look for output for the web service, prepended with your app name. If you named it the same as shown in this example, the name is getstartedlab\_web. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

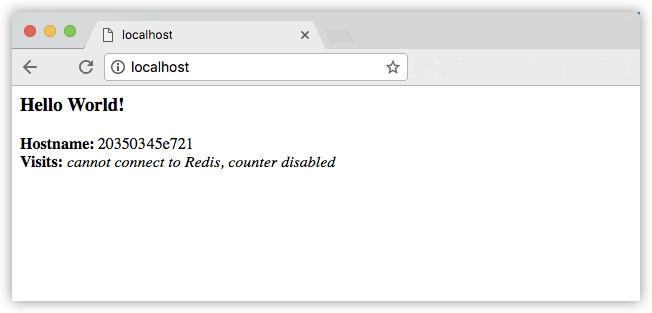
A single container running in a service is called a **task**. Tasks are given unique IDs that numerically increment, up to the number of replicas you defined in docker-compose.yml. List the tasks for your service:

docker service ps getstartedlab\_web

Tasks also show up if you just list all the containers on your system, though that is not filtered by service:

docker container ls -q

You can run curl -4 http://localhost:4000 several times in a row, or go to that URL in your browser and hit refresh a few times.



Either way, the container ID changes, demonstrating the load-balancing; with each request, one of the 5 tasks is chosen, in a round-robin fashion, to respond. The container IDs match your output from the previous command (docker container ls -q).

**Running Windows 10?**

Windows 10 PowerShell should already have curl available, but if not you can grab a Linux terminal emulator like [Git BASH](https://git-for-windows.github.io/" \t "_blank), or download [wget for Windows](http://gnuwin32.sourceforge.net/packages/wget.htm) which is very similar.

**Slow response times?**

Depending on your environment’s networking configuration, it may take up to 30 seconds for the containers to respond to HTTP requests. This is not indicative of Docker or swarm performance, but rather an unmet Redis dependency that we address later in the tutorial. For now, the visitor counter isn’t working for the same reason; we haven’t yet added a service to persist data.

## Scale the app

You can scale the app by changing the replicas value in docker-compose.yml, saving the change, and re-running the docker stack deploy command:

docker stack deploy -c docker-compose.yml getstartedlab

Docker performs an in-place update, no need to tear the stack down first or kill any containers.

Now, re-run docker container ls -q to see the deployed instances reconfigured. If you scaled up the replicas, more tasks, and hence, more containers, are started.

### Take down the app and the swarm

* Take the app down with docker stack rm:
* docker stack rm getstartedlab
* Take down the swarm.

docker swarm leave --force

**Docker Hub: Setup Jenkins**

sudo docker pull jenkins

sudo docker run -p 8080:8080 -p 50000:50000 jenkins

Note the following points about the above **sudo** command −

* We are using the **sudo** command to ensure it runs with root access.
* Here, **jenkins** is the name of the image we want to download from Docker hub and install on our Ubuntu machine.
* **-p** is used to map the port number of the internal Docker image to our main Ubuntu server so that we can access the container accordingly.

In Docker, the containers themselves can have applications running on ports. When you run a container, if you want to access the application in the container via a port number, you need to map the port number of the container to the port number of the Docker host.

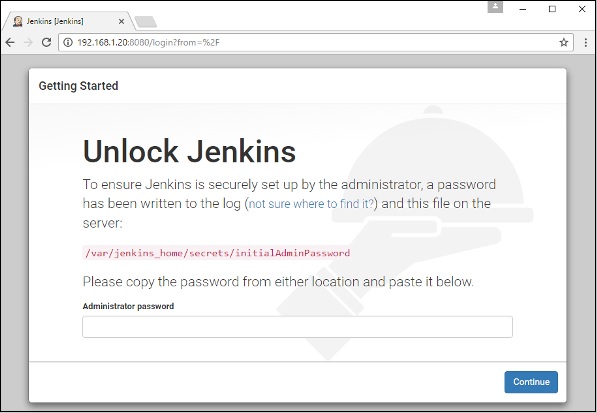
sudo docker pull jenkins

docker inspect Container/Image

sudo docker inspect jenkins

sudo docker inspect Jenkins > tmp.txt

The output of the **inspect** command gives a JSON output. If we observe the output, we can see that there is a section of "ExposedPorts" and see that there are two ports mentioned. One is the **data port** of 8080 and the other is the **control port** of 50000.



docker run -p 8080:8080 -p 50000:50000 -v /home/ec2-user/jenkins:/var/jenkins\_home Jenkins

docker exec jenkins cat /var/jenkins\_home/secrets/initialAdminPassword

# Private Registries

Use the Docker **run** command to download the private registry. This can be done using the following command.

sudo docker run –d –p 5000:5000 –-name registry registry:2

The following points need to be noted about the above command −

* **Registry** is the container managed by Docker which can be used to host private repositories.
* The port number exposed by the container is 5000. Hence with the **–p command**, we are mapping the same port number to the 5000 port number on our localhost.
* We are just tagging the registry container as “2”, to differentiate it on the Docker host.
* The **–d** option is used to run the container in detached mode. This is so that the container can run in the background

Now let’s tag one of our existing images so that we can push it to our local repository. In our example, since we have the **centos** image available locally, we are going to tag it to our private repository and add a tag name of **centos**.

sudo docker tag 67591570dd29 localhost:5000/centos

The following points need to be noted about the above command −

* **67591570dd29** refers to the Image ID for the **centos** image.
* **localhost:5000** is the location of our private repository.
* We are tagging the repository name as **centos** in our private repository.

Now let’s use the Docker **push** command to push the repository to our private repository.

sudo docker push localhost:5000/centos

Now that we don’t have any **centos** images on our local machine, we can now use the following Docker **pull** command to pull the **centos** image from our private repository.

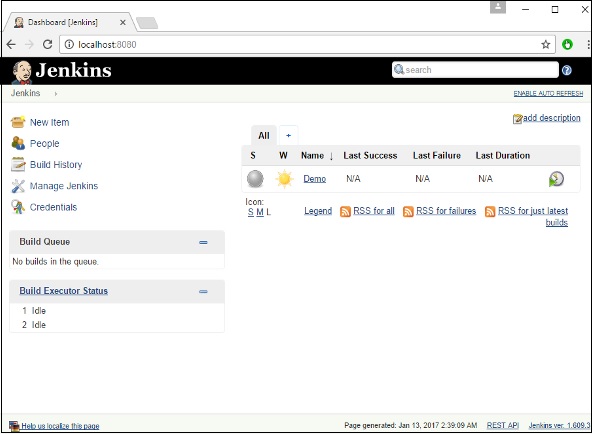
sudo docker pull localhost:5000/centos

# Continuous Integration

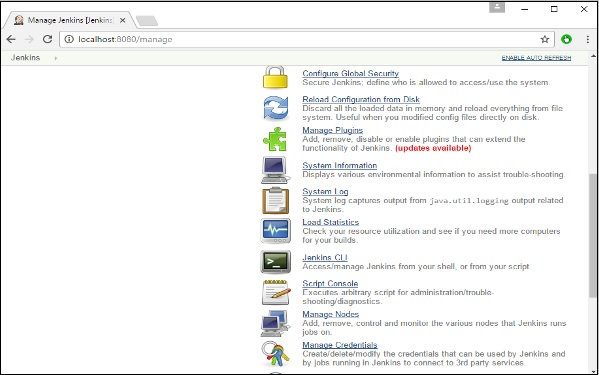
Docker has integrations with many Continuous Integrations tools, which also includes the popular CI tool known as **Jenkins**. Within Jenkins, you have plugins available which can be used to work with containers. So let’s quickly look at a Docker plugin available for the Jenkins tool.

Let’s go step by step and see what’s available in Jenkins for Docker containers.

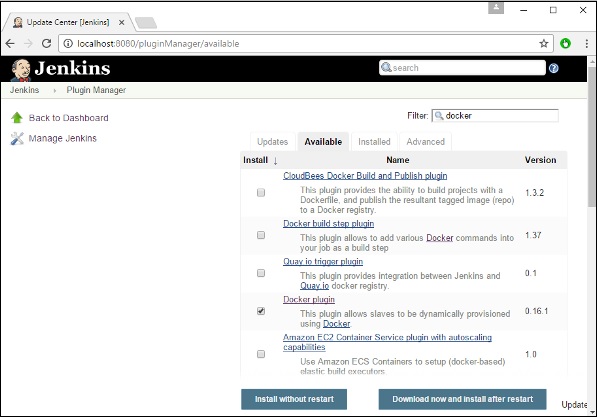
**Step 1** − Go to your Jenkins dashboard and click **Manage Jenkins**.



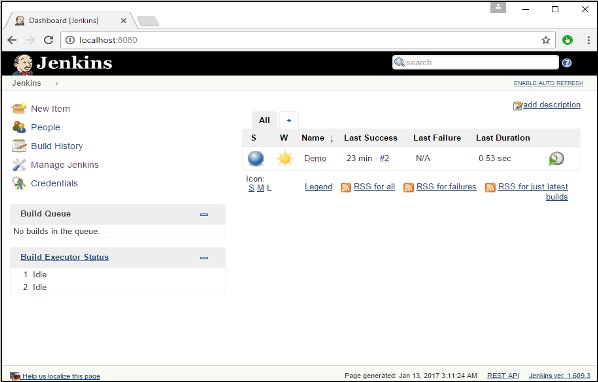
**Step 2** − Go to **Manage Plugins**.



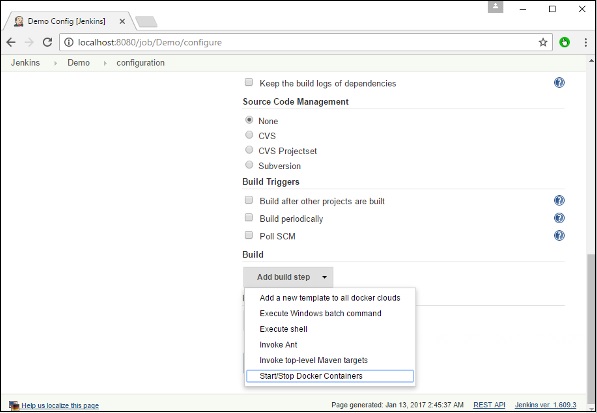
**Step 3** − Search for Docker plugins. Choose the Docker plugin and click the **Install** **without restart** button.



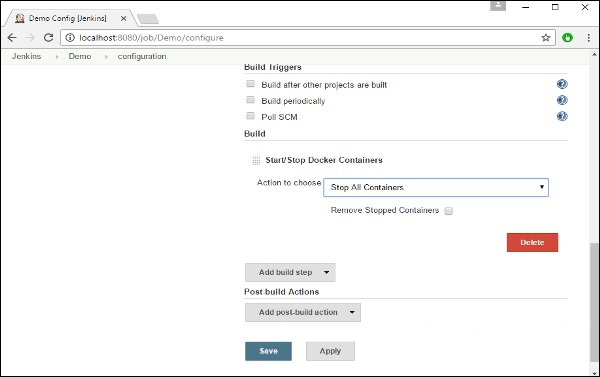
**Step 4** − Once the installation is completed, go to your job in the Jenkins dashboard. In our example, we have a job called **Demo**.



**Step 5** − In the job, when you go to the Build step, you can now see the option to start and stop containers.



**Step 6** − As a simple example, you can choose the further option to stop containers when the build is completed. Then, click the **Save** button.



Now, just run your job in Jenkins. In the Console output, you will now be able to see that the command to Stop All containers has run.

