**Docker**

Docker is an open-source tool for containerization that streamlines application creation and deployment using **containers.** Containers enable us to bundle all parts of an application into a single package for deployment.

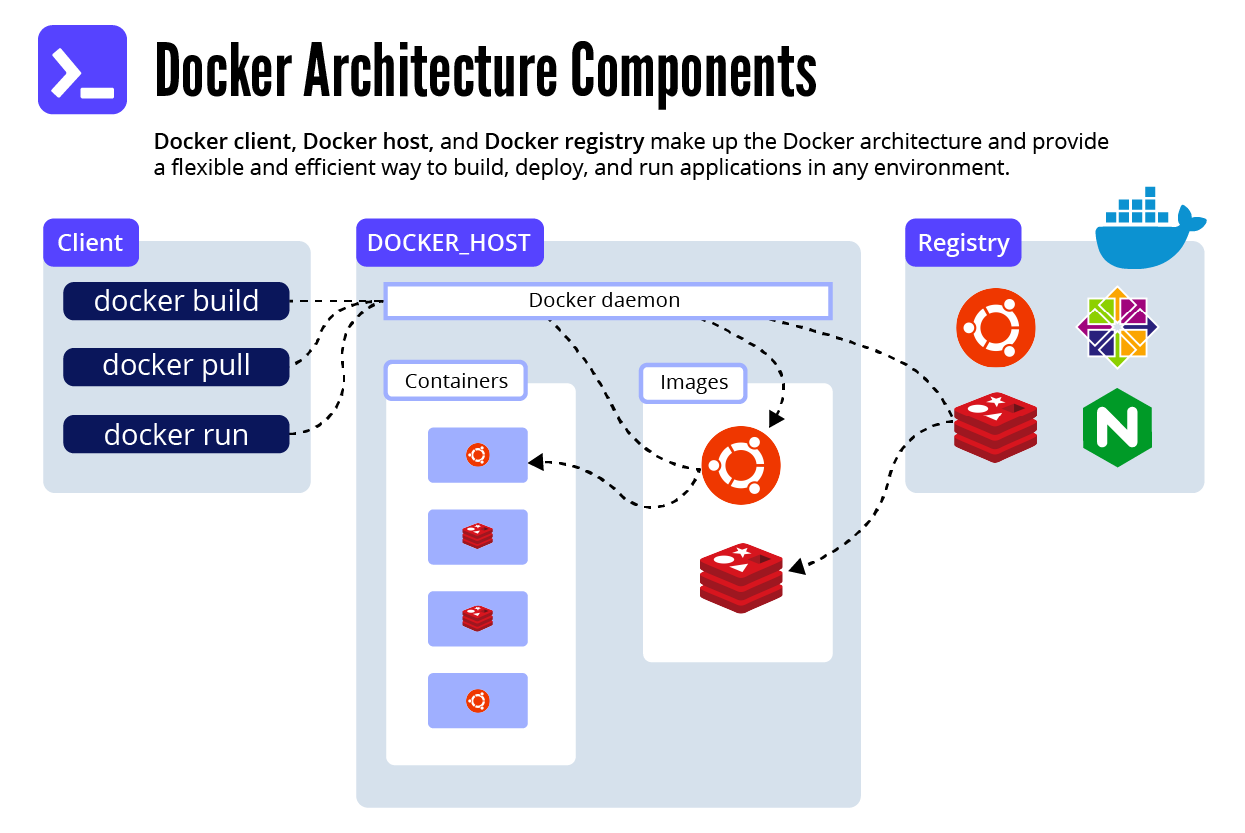
This tool makes it easy for different developers to work on the same project in the same environment without any dependencies or OS issues. Docker functions similarly to a virtual machine, however, it enables applications to share the same Linux kernel.

Docker offers many advantages for developers and DevOps teams. Some of these advantages include:

* Being highly demanded by both large and small companies.
* Providing isolation from the main system.
* Streamlining configuration.
* Providing access to thousands of pre-configured images through Docker Hub.
* Supporting many CI tools like Travis and Jenkins.
* Allowing developers to concentrate solely on writing code.
* Simplifying deployment management for operations teams.

Docker is frequently utilized in conjunction with **Kubernetes,** a robust container management tool that automates the deployment of Docker containers. While Docker is utilized to package, isolate, and distribute applications in containers, Kubernetes acts as the container scheduler responsible for deploying and scaling the application.

These two technologies complement each other, making application deployment effortless.



**Fundamentals of Docker**

Before diving into advanced Docker concepts, like Docker Compose, we want to make sure to refresh the fundamentals of Docker as a whole. Let’s define and explore the basics of Docker.

**Docker Client**

A Docker Client is a component used by a Docker user to interact with the Docker daemon and issue commands. These commands are based on the Docker API.

**Docker Architecture**

The Docker Architecture is made of layers, as we will discuss below. The bottom layer is the physical server that we use to host virtual machines. This is the same as a traditional virtualization architecture. The second layer is the Host OS, which is the base machine (i.e., Windows or Linux). Next, is the Docker Engine, which we use to run the operating system. Above that are the Apps which run as Docker containers. Those Docker Objects are made up of images and containers.

**Containers and images**

The basic structure of Docker relies on images and containers. We can think of a container as an object and an image as its class.

A container is an isolated system that holds everything required to run a specific application. It is a specific instance of an image that simulates the necessary environment. The following is an example command for running an Ubuntu Docker container and accessing the bash shell:

**docker run -i -t ubuntu /bin/bash**

**Images,** on the other hand, are used to start up containers. From running containers, we can get images, which can be composed together to form a system-agnostic way of packaging applications.

Images can be pre-built, retrieved from registries, created from already existing ones, or combined together via a common network.

**Dockerfiles**

Dockerfiles are how we containerize our application, or how we build a new container from an already pre-built image and add custom logic to start our application. From a Dockerfile, we use the Docker build command to create an image.

Think of a Dockerfile as a text document that contains the commands, we call on the command line to build an image.

Below is an example of a Dockerfile:

FROM python:3  
WORKDIR /usr/src/app  
COPY requirements.txt ./  
RUN pip install --no-cache-dir -r requirements.txt  
COPY . .  
CMD [ "python", "./your-daemon-or-script.py" ]

**Layers**

A Dockerfile works in layers. These are the building blocks of Docker. The first layer starts with the FROMkeyword and defines which pre-built image we will use to build an image. We can then define user permissions and startup scripts.

In Docker, a container is an image with a readable layer built on top of a read-only layer. These layers are called intermediate images, and they are generated when we execute the commands in our Dockerfile during the build stage.

**Docker Registry**

Docker Registry is a centralized location for storing and distributing Docker images. The most used public registry is Docker Hub, but you can also create your own private registry.

**Docker Daemon**

Docker Daemon runs on a host machine and manages containers, images, networks, and volumes. It receives commands from the Docker client and executes them. The Docker daemon uses Docker images to create containers.

**Docker Hub**

Docker Hub is a Docker Registry that provides unlimited storage for public images and offers paid plans for hosting private images. Anybody can access a public image. But to publish and access images on Docker Hub, you must create an account first.

Here are some common commands for using Docker Hub:

* docker login: Login to your Docker Hub account from the command line.
* docker pull: Download an image from Docker Hub to your local machine. For example, docker pulls alpine.
* docker push: Upload a local image to Docker Hub. For example, docker pushes username/image-name.
* docker search: Search for an image on Docker Hub. For example, docker search alpine.
* docker tag: Tag an image with a new repository name and/or tag. For example, docker tag image-id username/repository:tag.
* docker images: List all images on the local machine. docker rmi: Remove an image from the local machine. For example, docker rmi 4535, where 4535 is an ID of an existing image on your machine.

**Dockerfile vs Docker Compose**

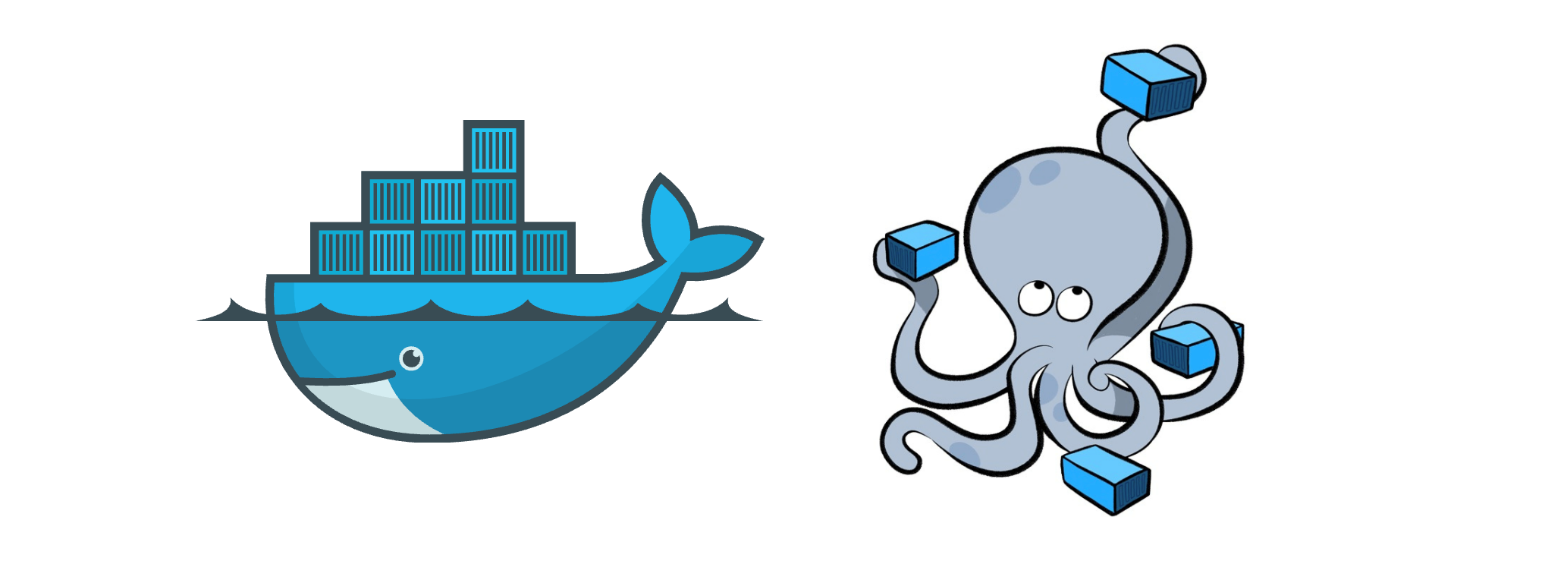
As we discussed, a Dockerfile is a script that embodies instructions for building a Docker image. It stipulates the base image to use, the commands to run, and the files to copy into the image. After creating a Dockerfile, you can utilize the docker build command to build an image, which is then run as a container.

Meanwhile, Docker Compose is the tool we use for defining and running multi-container Docker applications. You can use it to define the services that make up your application, along with their configurations and dependencies, in a single file called docker-compose.yml.

**Dockerfiles** and **Docker Compose** play separate roles but work in harmony to help streamline your DevOps workflow.

The Docker methodology is to write an appropriate Dockerfile for each image you need to create, then use Docker Compose to group the images together using the build command.

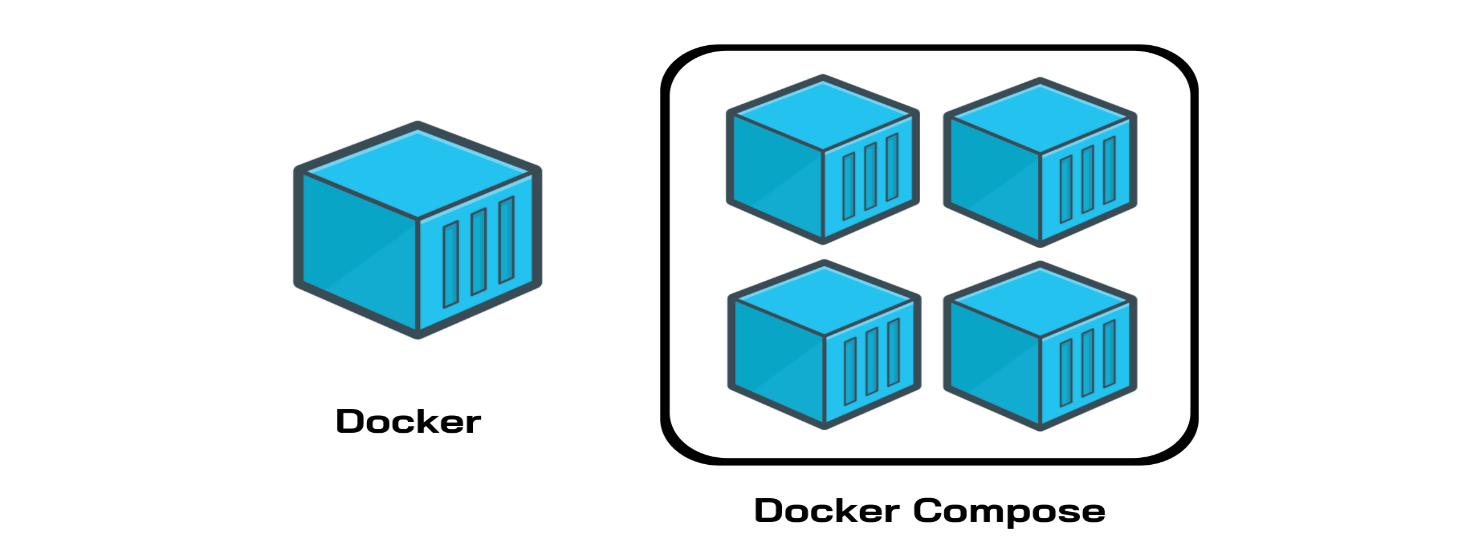
This was a brief overview of the basics of Docker before we delve into more advanced concepts. It’s important to note that there is much more to learn about Docker beyond what we have discussed here.

**Getting started with Docker Compose**

Now for the advanced stuff. **Docker Compose** is a Docker tool used to define and run multi-container applications. With Compose, you use a YAML file to configure your application’s services and create all the app’s services from that configuration.

Think of docker-compose as an **automated multi-container workflow.** Compose is an excellent tool for development, testing, CI workflows, and staging environments. According to the Docker documentation, the most popular features of Docker Compose are:

* Multiple isolated environments on a single host
* Preserve volume data when containers are created
* Only recreate containers that have changed
* Variables and moving a composition between environments
* Orchestrate multiple containers that work together.



**How to use and install Docker Compose**

Compose uses the Docker Engine, so you’ll need to have the Docker Engine installed on your device. You can run Compose on Windows, Mac, and 64-bit Linux. Installing Docker Compose is actually quite easy.

On desktop systems, such as Docker Desktop for Mac and Windows, Docker Compose is already included. No additional steps are needed. On Linux systems, you’ll need to:

1. Install the Docker Engine
2. Run the following command to download Docker Compose

sudo curl -L "https://github.com/docker/compose/releases/download/1.26.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

1. Apply permissions to the binary, like so:

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

1. Test the installation to check it worked properly

$ docker-compose --versiondocker-compose version 1.26.2, build 1110ad01

Regardless of how you chose to install it, once you have Docker Compose downloaded and running properly, you can start using it with your Dockerfiles. This process requires three basic steps:

1. Define your app’s environment using a Dockerfile. This way, it can be reproduced.
2. Define the services for your app in a docker-compose.yml file. This way, they can run in an isolated environment.
3. Run docker-compose to start your app.

You can easily add Docker Compose to a pre-existing project. If you already have some Dockerfiles, add Docker Compose files by opening the Command Palette. Use the **Docker: Docker Compose Files to the Workspace** command, and, when prompted, choose the Dockerfiles you want to include.

You can also add Docker Compose files to your workspace when you add a Dockerfile. Similarly, open the Command Palette and use the **Docker: Add Docker Files to Workspace** command.

You’ll then be asked if you want to add any Docker Compose files. In both cases, compose extension will add the docker-compose.yml file to your workspace.

**Docker Compose file structure.**

Now that we know how to download Docker Compose, we need to understand how Compose files work. It’s actually simpler than it seems. In short, Docker Compose files work by applying multiple commands that are declared within a single docker-compose.yml configuration file.

The basic structure of a Docker Compose YAML file looks like this:

version: 'X'

services:

web:

build: .

ports:

- "5000:5000"

volumes:

- .:/code

redis:

image: redis

Now, let’s look at a real-world example of a Docker Compose file and break it down step-by-step to understand all this better. Note that all the clauses and keywords in this example are industry-standard and commonly used.

With just these, you can start a development workflow. There are some more advanced keywords that you can use in production, but for now, let’s just get started with the necessary clauses.

ports:

- "5000:5000"

# Mount volume

volumes:

- "/usercode/:/code"

# Link database container to app container

# for reachability.

links:

- "database:backenddb"

database:

# image to fetch from docker hub

image: mysql/mysql-server:5.7

# Environment variables for startup script

# container will use these variables

# to start the container with these, define variables.

environment:

- "MYSQL\_ROOT\_PASSWORD=root"

- "MYSQL\_USER=testuser"

- "MYSQL\_PASSWORD=admin123"

- "MYSQL\_DATABASE=backend"

# Mount init.sql file to automatically run

# and create tables for us.

# everything in docker-entrypoint-initdb.d folder

# is executed as soon as container is up nd running.

build: .

# Mapping of container port to host

* **version ‘3’:** This denotes that we are using version 3 of Docker Compose, and Docker will provide the appropriate features. At the time of writing this article, version 3.7 is latest version of Compose.
* **services:** This section defines all the different containers we will create. In our example, we have two services, web and database.
* **web:** This is the name of our Flask app service. Docker Compose will create containers with the name we provide.
* **build:** This specifies the location of our Dockerfile, and. represents the directory where the docker-compose.yml file is located.
* **ports:** This is used to map the container’s ports to the host machine.
* **volumes:** This is just like the -v option for mounting disks in Docker. In this example, we attach our code files directory to the containers’ ./code directory. This way, we won’t have to rebuild the images if changes are made.
* **links:** This will link one service to another. For the bridge network, we must specify which content should be accessible to which container using links.
* **image:** If we don’t have a Dockerfile and want to run a service using a pre-built image, we specify the image location using the image clause. Compose will fork a container from that image.
* **environment:** The clause allows us to set up an environment variable in the container. This is the same as the -e argument in Docker when running a container.

Congrats! Now you know a bit about Docker Compose and the necessary parts you’ll need to get started with your workflow.

**Docker Compose commands**

Now that we know how to create a docker-compose file, let’s go over the most common Docker Compose commands that we can use with our files. Keep in mind that we will only be discussing the most frequently used commands.

**docker-compose:** Every Compose command starts with this command. You can also use docker-compose <command> --help to provide additional information about arguments and implementation details.

**$ docker-compose --help**Define and run multi-container applications with Docker.

**docker-compose build:** The build command builds or rebuilds images in the docker-compose.yml file. This file contains all the necessary configurations for all the services that make up the application.

The job of the build command prepares images to create containers. If a service is using the pre-built image, it will skip this command. The docker-compose build command reads the Dockerfile for each service, including the instructions to build the image. The built images can then be used to create containers for each service using the docker-compose-up command. Furthermore, we use the docker-compose build command for building the images for the services in a consistent and reproducible way, making deployment in different environments easier.

**$ docker-compose build**  
database uses an image, skipping  
Building web  
Step 1/11 : FROM python:3.9-rc-buster  
 ---> 2e0edf7d3a8a  
Step 2/11 : RUN apt-get update && apt-get install -y docker.io

**docker-compose images:** This command will list the images you’ve built using the current docker-compose file.

**$ docker-compose images**

Container Repository Tag Image Id Size

---------------------------------------------------------------------------------------------------------------------------------------

7001788f31a9\_docker\_database\_1 mysql/mysql-server 5.7 2a6c84ecfcb2 333.9 MB

docker\_database\_1 mysql/mysql-server 5.7 2a6c84ecfcb2 333.9 MB

docker\_web\_1 <none> <none> d986d824dae4 953MB

**docker-compose stop:** This command stops the running containers of specified services.

**$ docker-compose stop**Stopping docker\_web\_1      ... done  
Stopping docker\_database\_1 ... done

**docker-compose run:** This is similar to the docker run command. It will create containers from images built for the services mentioned in the compose file.

**$ docker-compose run web**Starting 7001788f31a9\_docker\_database\_1 ... done  
 \* Serving Flask app "app.py" (lazy loading)  
 \* Environment: development  
 \* Debug mode: on  
 \* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)  
 \* Restarting with stat  
 \* Debugger is active!  
 \* Debugger PIN: 116-917-688

**docker-compose up:** This command does the work of the docker-compose build and docker-compose run commands. It builds the images if they are not located locally and starts the containers. If images are already built, it will fork the container directly.

**$ docker-compose up**Creating docker\_database\_1 ... done  
Creating docker\_web\_1      ... done  
Attaching to docker\_database\_1, docker\_web\_1

**docker-compose ps:** This command list all the containers in the current docker-compose file. They can then either be running or stopped.

**$ docker-compose ps**  
 Name Command              State                Ports           
---------------------------------------------------------------------------------------------------------------------------------------  
docker\_database\_1   /entrypoint.sh mysqld    Up (healthy)    3306/tcp, 33060/tcp     
docker\_web\_1         flask run                Up              0.0.0.0:5000->5000/tcp

$ docker-compose ps  
Name                  Command           State     Ports  
------------------------------------------------------------------------------------------------------------------  
docker\_database\_1    /entrypoint.sh mysqld    Exit 0          
docker\_web\_1         flask run                Exit 0

**docker-compose down:** This command is like the docker system prune command. However, in Compose, it stops all the services and cleans up the containers, networks, and images.

**$ docker-compose down**  
Removing docker\_web\_1      ... done  
Removing docker\_database\_1 ... done  
Removing network docker\_default  
(django-tuts) Venkateshs-MacBook-Air:Docker venkateshachintalwar$ docker-compose images  
Container   Repository   Tag   Image Id   Size  
----------------------------------------------  
(django-tuts) Venkateshs-MacBook-Air:Docker venkateshachintalwar$ docker-compose ps  
Name   Command   State   Ports  
------------------------------

**Practice Notes**

**Docker Compose**

This is a feature of docker using which we can create multi-container architecture using YAML files. This YAML file contains information about the containers that we want to launch and how they must be linked with each other. YAML is a file format. It is not a scripting language.

YAML will store the data in key-value pairs.

Lefthand side - Key

Righthand side - Value

The Yaml file is space indented.

Sample Yaml file

---

durgasoft: root element

trainers: child element

shiva: Devops

raj: Python

Coordinators:

lakshmi: Devops

rani: AWS

...

durgasoft --> root element

* To validate the above Yaml file - Open - <http://www.yamllint.com/> -->Paste the above code --> Go button

**Installing Docker compose**

1) Open https://docs.docker.com/compose/install/

2) Go to linux section

Copy and pase the below two commands

# sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

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

How to check whether docker-compose is installed or not?

# docker-compose --version

1. **Create a docker-compose file for setting up the dev environment. The MySQL container is linked with the WordPress container.**

# vim docker-compose.yml ( Name of the file should be docker-compose.yml)

---

version: '3'

services:

mydb:

image: mysql:5

environment:

MYSQL\_ROOT\_PASSWORD: sunilsunil

mysite:

image: wordpress

ports:

- 5050:80

links:

- mydb:mysql

...

:wq

Let’s remove all the running container

# docker rm -f $(docker ps -aq)

How to start the above services from dockerfile

# docker-compose up

We got a lot of logs coming on the screen. to avoid it we use the -d option. Before adding -d stop the container.

# docker-compose stop

# docker-compose up -d

To check WordPress

public\_ip:5050

To stop both the containers

# docker-compose stop

Remove the container

# docker rm -f $(docker ps -aq)

1. **Create a docker-compose file for setting up LAMP architecture.**

# vim docker-compose.yml

---

version: '3'

services:

mydb:

image: mysql:5

environment:

MYSQL\_ROOT\_PASSWORD: shiva

apache:

image: tomee

ports:

- 6060:8080

links:

- mydb:mysql

php:

image: php

links:

- mydb:mysql

- apache:tomcat

...

:wq

How to start the above services from dockerfile

# docker-compose up -d

To see the list of the containers

# docker container ls (Observation - we are unable to see the php container)

# docker ps -a

1. **Docker-compose file for setting up CI-CD Environment. The Jenkins container is linked with two Tomcat containers.**

# vim docker-compose.yml

---

version: '3'

services:

devserver:

image: jenkins/jenkins

ports:

- 7070:8080

qaserver:

image: tomee

ports:

- 8899:8080

links:

- devserver:jenkins

prodserver:

image: tomee

ports:

- 9090:8080

links:

- devserver:jenkins

...

:wq

# docker rm -f $(docker ps -aq)

# docker-compose up -d

# docker container ls

To check

public\_ip:7070 (To check Jenkins) ---------> 13.126.58.183:7070

public\_ip:8899 (Tomcat qa server) ---------> 13.126.58.183:8899

public\_ip:9090 (Tomcat prod server) ---------> 13.126.58.183:9090

1. **Docker-compose file to set up a testing environment. Selenium hub container is linked with two node containers.**

# vim docker-compose.yml

---

version: '3'

services:

hub:

image: selenium/hub

ports:

- 4444:4444

chrome:

image: selenium/node-chrome-debug

ports:

- 5901:5900

links:

- hub:selenium

firefox:

image: selenium/node-firefox-debug

ports:

- 5902:5900

links:

- hub:selenium

...

:wq

Let’s delete all the running containers.

# docker rm -f $(docker ps -aq)

# docker-compose up -d

# docker container ls

As it is a GUI container, we can access using the VNC viewer.

Open VNC viewer

52.77.219.115:5901

password: secret