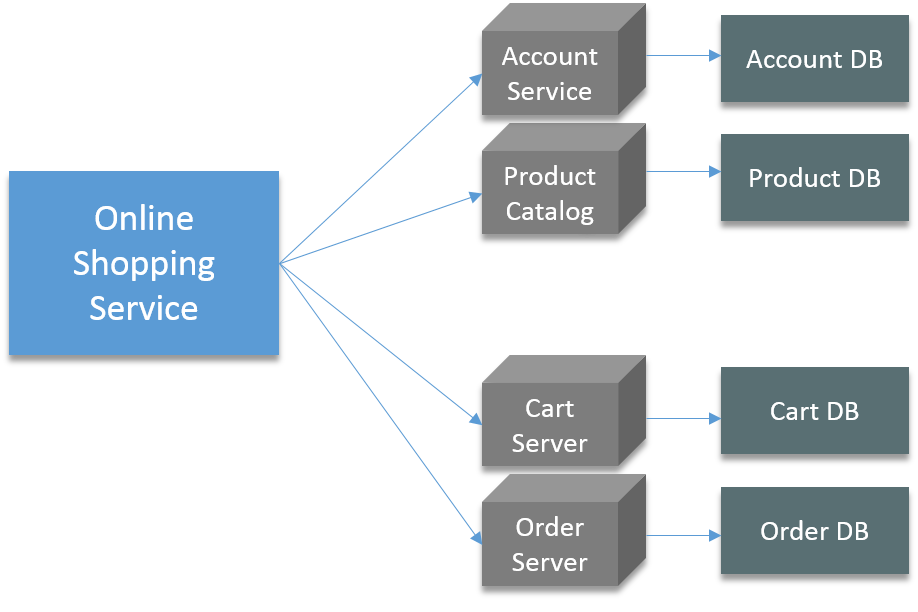
Docker :

microservices is that certain types of applications become easier to build and maintain when they are broken down into smaller, composable pieces which work together. Each component is developed separately, and the application is then simply the sum of its constituent components.

Consider the example below:



In the above diagram there is an online shop with separate microservices for user-account, product catalog, order processing and shopping carts.

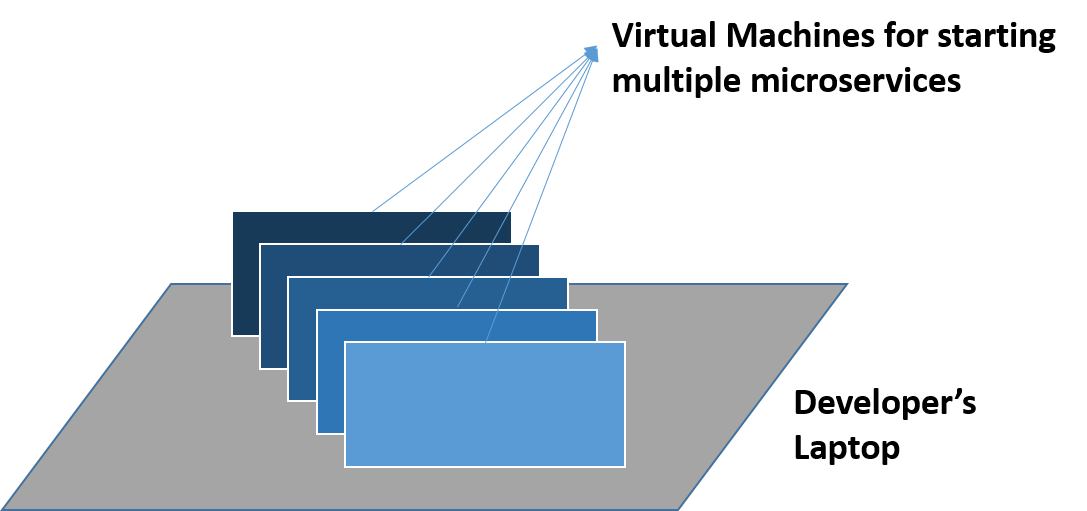
Well, this architecture has a lot of benefits:

* Even if one of your microservice fails, your entire application is largely unaffected.
* It is easier to manage

There are many other benefits as well, I won’t go into much detail about microservices in this post. But, soon I will be coming up with a couple of blogs on microservices as well.

In this architecture, we were using CentOS Virtual Machines. Those Virtual Machines were configured by writing long scripts. Well, configuring those VMs was not the only problem.

Developing such applications requires starting of several of microservices in one machine. So if you are starting five of those services you require five VMs on that machine. Consider the diagram below:



The other problem is pretty common, I know a lot of you can relate to it. The application works in a developer’s laptop but not in testing or production. This can be because of not keeping a consistent computing environment. Consider the diagram below:

There were many other problems apart from this as well, but I feel, these problems are enough for me to explain you the need of Docker Containers.

[***Learn How Docker Containers Are Better Than Virtual Machines***](https://www.edureka.co/blog/docker-tutorial)

So, imagine if I am giving 8 GB of RAM to all my VMs, and I have 5 microservices running on different Virtual Machines. In that case, these VMs will require 40 GB of RAM. Well, now I require the configurations of my host machine to be very high, almost 44 GB of RAM should be there in my host machine. Obviously, this is not a sustainable solution for such an architecture because, I am wasting a lot of resources here.

Fine, I have a lot of resources to waste, but still I have a problem of inconsistency in my software delivery life-cycle (SDLC). I have to configure these VMs in test as well as in prod environment. Somewhere in that process, some software was not updated in the test server, and the Dev team is using the updated version of the software. This leads to conflicts.

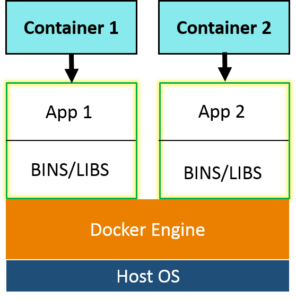
What if I am using 100 VMs, then configuring each VM will take a lot of time, and at the same time it is prone to error as well.

Now, let us understand what is Docker Container and how it works, and how it solved my problem.

**What is a Docker Container?**

Docker is a tool designed to make it easier to create, deploy and run applications by using containers.

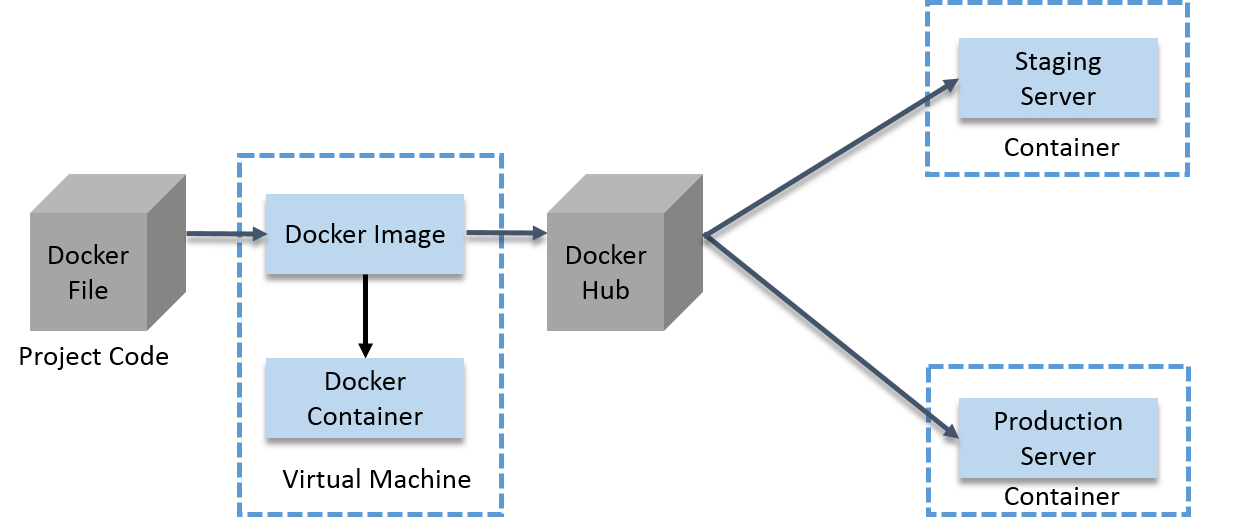
You can create Docker Containers, these containers will contain all the binaries and libraries required for your application or microservice in my case. So your application is present in a container, or you have containerized your application. Now, that same container can be used in the Test and Prod environment.

Docker Containers are a lightweight solution to Virtual Machines, and it uses the host OS. The best part, you don’t have to pre-allocate any RAM to the Docker Container, it will take it as and when required. So, with Docker Container I don’t have to worry about wastage of resources.

Let’s understand now, how  a Docker Container works.

**How a Docker Container Works?**

The below diagram is basically, a way to use Docker. And I am assuming that, you have an idea about Docker Image and Dockerfile.

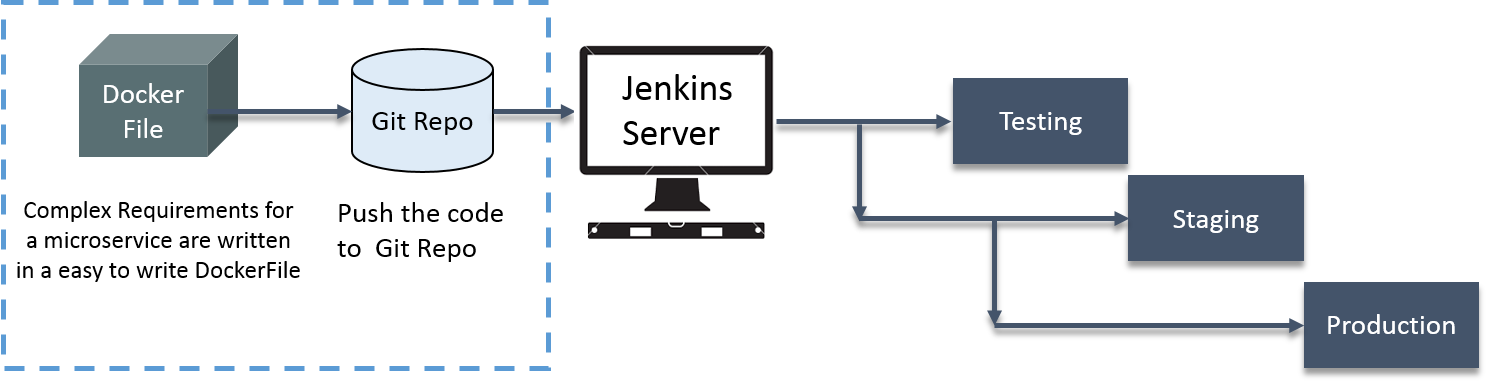


Guys, I know the diagram looks a bit complex, but trust me it ain’t that complex. Below is the explanation of the diagram, even after that you feel it is tough to understand, you can comment your doubt, I will address those questions ASAP.

* A developer will first write the project code in a Docker file and then build an image from that file.
* This image will contain the entire project code.
* Now, you can run this Docker Image to create as many containers as you want.
* This Docker Image can be uploaded on Docker hub (It is basically a cloud repository for your Docker Images, you can keep it public or private).
* This Docker Image on the Docker hub, can be pulled by other teams such as QA or Prod.

This not only prevents the wastage of resources, but also makes sure that the computing environment that is there in a Developer’s laptop is replicated in other teams as well. I feel now, I don’t have to tell you why we need Docker.

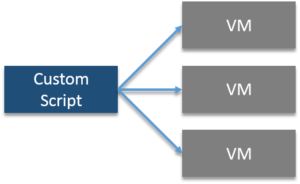
This was one way to use it, I am guessing you guys must be curious to know how I used Docker to solve my problem of microservices. Let me give you an overview on the same.



Below is the explanation of the diagram:

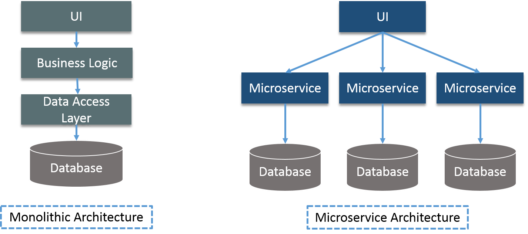
* Firstly, we wrote the complex requirements within a Dockerfile.
* Then, we pushed it on GitHub.
* After that we used a CI server (Jenkins).
* This Jenkins server will pull it down from Git, and the build the exact environment. This will be used in Production servers as well as in Test servers.
* We deployed it out to staging (It refers to deploying your software onto servers for testing purposes, prior to deploying them fully into production.) environments for Testers.
* Basically, we rolled exactly what we had in Development, Testing and Staging into Production.

They were using custom scripts to deploy the applications in the VM.



Their environment was optimized for their legacy Java-based applications. Their growing environment involves new products that aren’t solely java based. In order to give their students the best experience possible, the University needed to begin modernizing the applications.

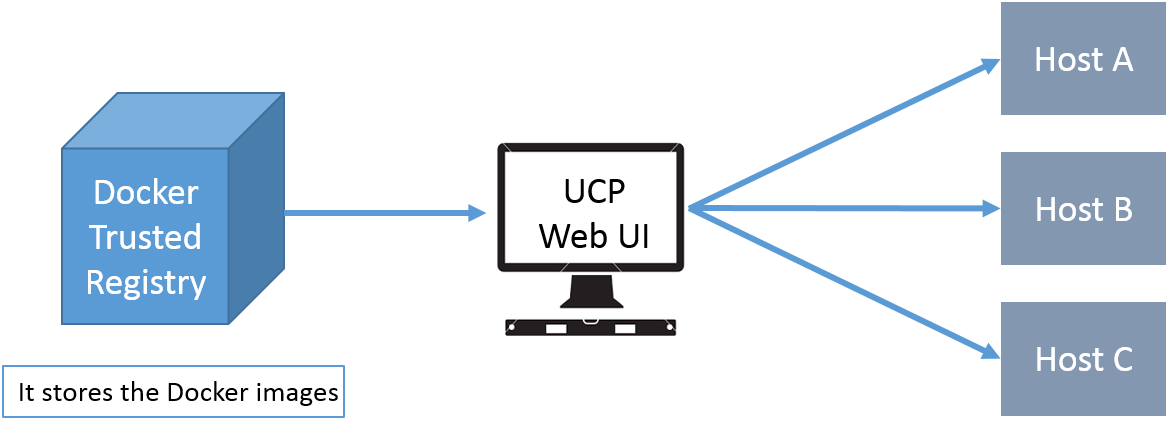
The University wanted to improve the way they architect applications, by moving to a microservices based architecture for their applications.



Security was needed for student’s data such as SSNs and student health data.

### ****Solution:****

All the problems were addressed by Docker Data Center (DDC), consider the diagram below:



**Docker Trusted Registry**– It stores the Docker Images.

***UCP (Universal Control Plane) Web UI*** – Helps in managing whole cluster from a single place. Services are deployed using UCP web UI, using Docker images that are stored in DTR (Docker Trusted Registry).

IT ops teams leverages Universal Control Plane to provision Docker installed software on hosts, and then deploy their applications without having to do a bunch of manual steps to set up all their infrastructure.

UCP and DTR integrates with their LDAP server to quickly provision access to their applications.

I am hoping you guys have read the previous blogs to learn the basics of Docker.

Now, I will explain you how we can use Docker Compose for multi container application.

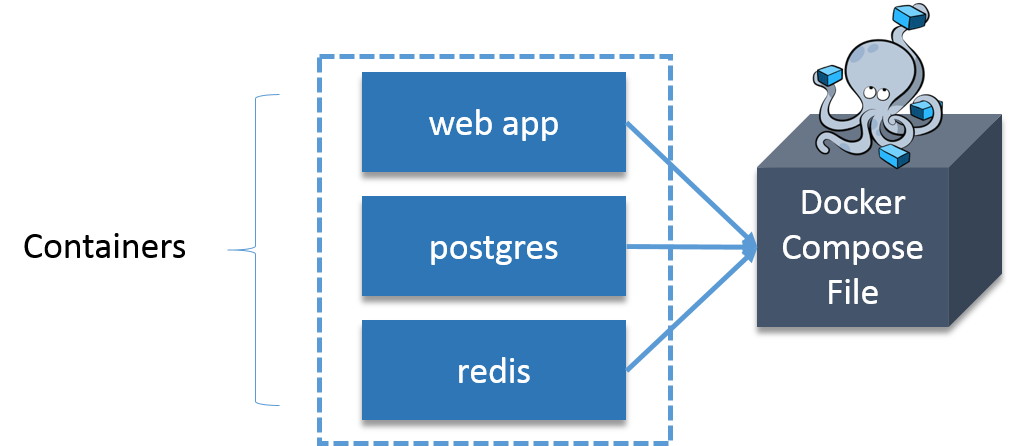
## ****Docker Hands-On:****

I am assuming you have installed Docker. I will be using Docker Compose in this post, below I have given a small introduction to Docker Compose.

**Docker Compose:** It is a tool for defining and running multi-container Docker applications. With Docker Compose, you can use a Compose file to configure your application’s services. Then, using a single command, you can create and start all the services from your configuration.

Suppose you have multiple applications in various containers and all those containers are linked together. So, you don’t want to execute each of those containers one by one. But, you want to run those containers with a single command. That’s where Docker Compose comes in to the picture. With it you can run multiple applications in various containers with a single command. i.e. docker-compose up.

Example: Imagine you have different containers, one running a web app, another running a postgres and another running redis, in a YAML file. That is called docker compose file, from there you can run these containers with a single command.

Let us take one more example:

Suppose you want to publish a blog, for that you will use CMS (Content Management System), and wordpress is the most widely used CMS. Basically, you need one container for WordPress and you need one more container as MySQL for back end, that MySQL container should be linked to the wordpress container. We also need one more container for Php Myadmin that will be linked to MySQL database, basically, it is used to access MySQL database.

How about I execute the above stated example practically.

### ****Steps involved:****

1. **Install Docker Compose:**
2. **Install WordPress:**We’ll be using the official [WordPress](https://hub.docker.com/_/wordpress/) and [MariaDB](https://hub.docker.com/_/mariadb/) Docker images.
3. **Install MariaDB:** It is one of the most popular database servers in the world. It’s made by the original developers of MySQL. MariaDB is developed as open source software and as a relational database it provides an SQL interface for accessing data.
4. **Install PhpMyAdmin:**It is a free software tool written in PHP, intended to handle the administration of MySQL over the Web.
5. **Create The WordPress Site:**

Let’s get started!

### ****Install Docker Compose:****

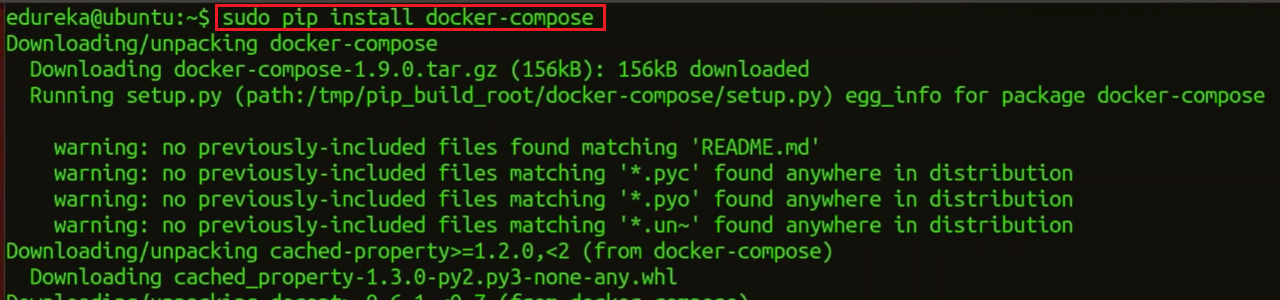
Install Python Pip first:

sudo apt-get install python-pip



Now, you can install Docker Compose:

sudo pip install docker-compose



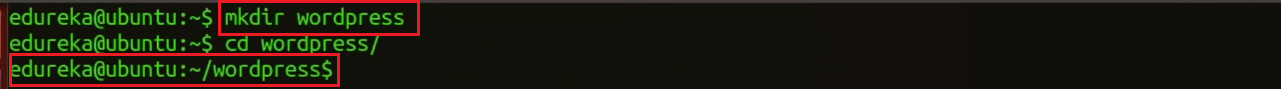
### ****Install WordPress:****

Create a wordpress directory:

mkdir wordpress

Enter this wordpress directory:

cd wordpress/



In this directory create a Docker Compose YAML file, then edit it using gedit:

sudo gedit docker-compose.yml

Docker Compose File - Docker Container - Edureka

Paste the below lines of code in that yaml file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | wordpress:    image: wordpress    links:    - wordpress\_db:mysql    ports:    - 8080:80    wordpress\_db:    image: mariadb    environment:    MYSQL\_ROOT\_PASSWORD: edureka    phpmyadmin:    image: corbinu/docker-phpmyadmin    links:    - wordpress\_db:mysql    ports:    - 8181:80    environment:    MYSQL\_USERNAME: root    MYSQL\_ROOT\_PASSWORD: edureka |

I know you want me to explain this code, so what I will do, I will take small sections of this code and explain you what’s happening.

|  |  |
| --- | --- |
| 1  2  3  4  5 | wordpress\_db:  ...   environment:      MYSQL\_ROOT\_PASSWORD: edureka  ... |

This will set an environment variable inside the wordpress\_db container called MYSQL\_ROOT\_PASSWORD with your desired password. The MariaDB Docker image is configured to check for this environment variable when it starts up and will take care of setting up the DB with a root account with the password defined as MYSQL\_ROOT\_PASSWORD.

|  |  |
| --- | --- |
| 1  2  3  4  5 | wordpress:  ...   ports:      - 8080:80  ... |

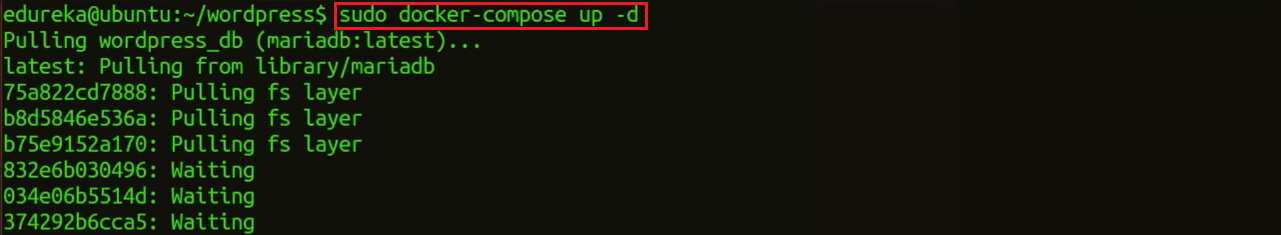
The first port number is the port number on the host, and the second port number is the port inside the container. So, this configuration forwards requests on port 8080 of the host to the default web server port 80 inside the container.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | phpmyadmin:    image: corbinu/docker-phpmyadmin    links:      - wordpress\_db:mysql    ports:      - 8181:80    environment:      MYSQL\_USERNAME: root      MYSQL\_ROOT\_PASSWORD: edureka |

This grabs docker-phpmyadmin by community member corbinu, links it to our wordpress\_db container with the name mysql (meaning from inside the phpmyadmin container references to the hostname mysql will be forwarded to our wordpress\_db container), exposes its port 80 on port 8181 of the host system, and finally sets a couple of environment variables with our MariaDB username and password. This image does not automatically grab the MYSQL\_ROOT\_PASSWORD environment variable from the wordpress\_dbcontainer’s environment, the way the wordpress image does. We actually have to copy the MYSQL\_ROOT\_PASSWORD: edureka line from the wordpress\_db container, and set the username to root.

Now start the application group:

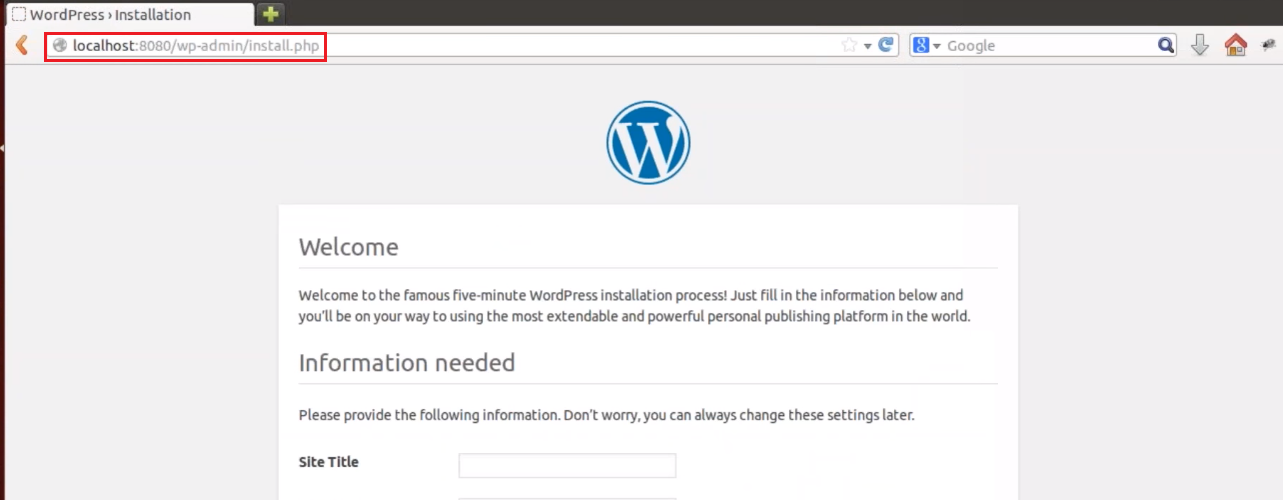
docker-compose up -d



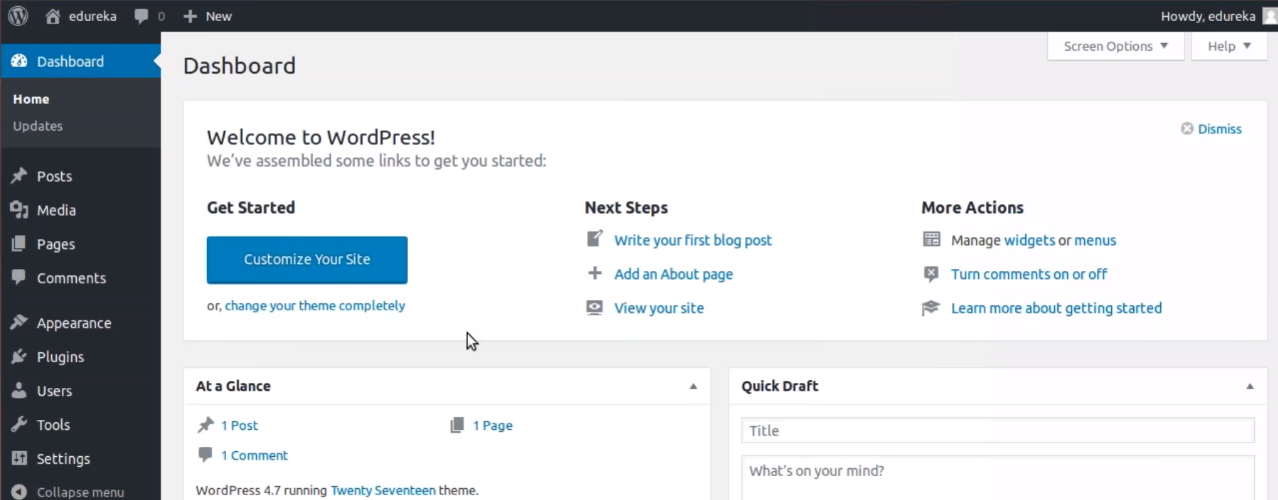
That’s all you have to do. You can add as many containers as you like this way, and link them all up in any way you please.

Now, in the browser go to port 8080, using your public IP or host name, as shown below:

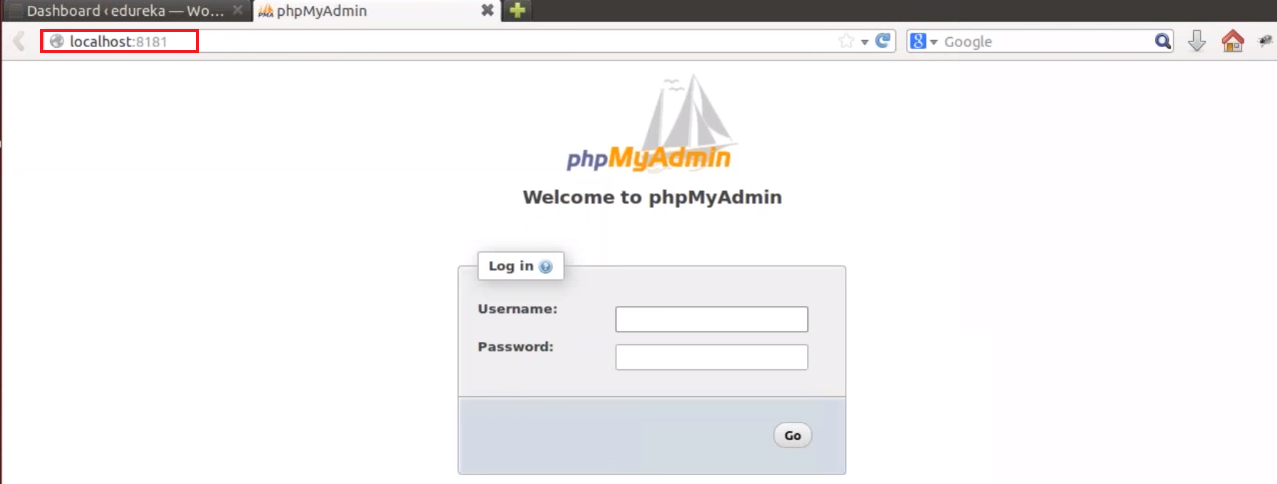
localhost:8080



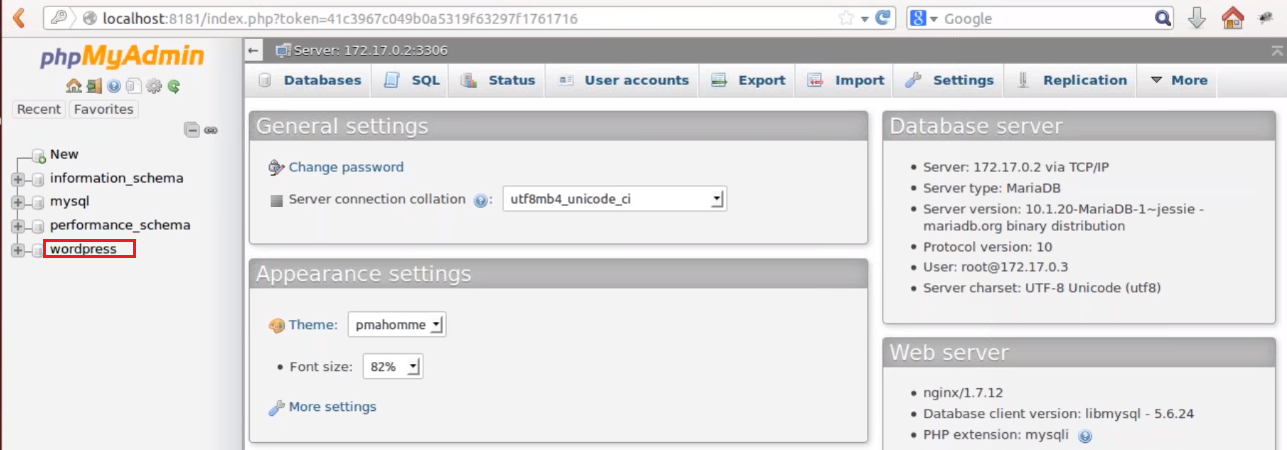
Fill this form and click on install WordPress.



Once it is finished, visit your server’s IP address again (this time using port 8181, e.g. localhost:8181). You’ll be greeted by the phpMyAdmin login screen:



Go ahead and login using username root and password you set in the YAML file, and you’ll be able to browse your database. You’ll notice that the server includes a wordpress database, which contains all the data from your WordPress install.



Following are the commands which are being covered:

* **[docker –version](https://www.edureka.co/blog/docker-commands/" \l "version)**
* **[docker pull](https://www.edureka.co/blog/docker-commands/" \l "pull)**
* **[docker run](https://www.edureka.co/blog/docker-commands/" \l "run)**
* **[docker ps](https://www.edureka.co/blog/docker-commands/" \l "ps)**
* **[docker ps -a](https://www.edureka.co/blog/docker-commands/" \l "psa)**
* **[docker exec](https://www.edureka.co/blog/docker-commands/" \l "exec)**
* **[docker stop](https://www.edureka.co/blog/docker-commands/" \l "stop)**
* **[docker kill](https://www.edureka.co/blog/docker-commands/" \l "kill)**
* **[docker commit](https://www.edureka.co/blog/docker-commands/" \l "commit)**
* **[docker login](https://www.edureka.co/blog/docker-commands/" \l "login)**
* **[docker push](https://www.edureka.co/blog/docker-commands/" \l "push)**
* **[docker images](https://www.edureka.co/blog/docker-commands/" \l "images)**
* **[docker rm](https://www.edureka.co/blog/docker-commands/" \l "rm)**
* **[docker rmi](https://www.edureka.co/blog/docker-commands/" \l "rmi)**
* **[docker build](https://www.edureka.co/blog/docker-commands/" \l "build)**

So, let’s get started:

**Docker Commands**

1.**docker –version**

This command is used to get the currently installed version of docker

2.**docker pull**

**Usage: docker pull <image name>**

This command is used to pull images from the **docker repository**(hub.docker.com)

3. **docker run**

**Usage: docker run -it -d <image name>**

This command is used to create a container from an image

4. **docker ps**

This command is used to list the running containers

5. **docker ps -a**

This command is used to show all the running and exited containers

6. **docker exec**

**Usage: docker exec -it <container id> bash**

This command is used to access the running container

7. **docker stop**

**Usage: docker stop <container id>**

This command stops a running container

8. **docker kill**

**Usage: docker kill <container id>**

This command kills the container by stopping its execution immediately. The difference between ‘docker kill’ and ‘docker stop’ is that ‘docker stop’ gives the container time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to kill it

9. **docker commit**

**Usage: docker commit <conatainer id> <username/imagename>**

This command creates a new image of an edited container on the local system

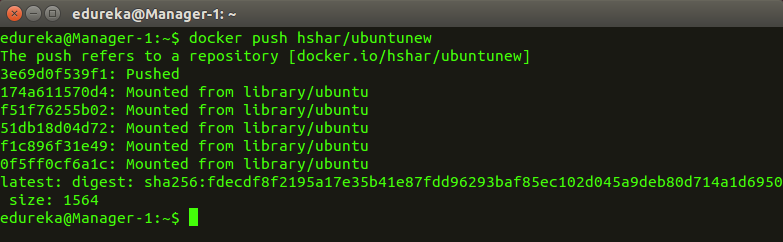
10. **docker login**

This command is used to login to the docker hub repository

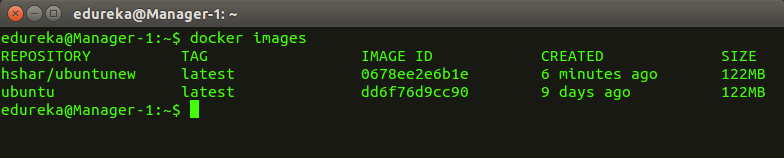
11.**docker push**

**Usage: docker push <username/image name>**

This command is used to push an image to the docker hub repository

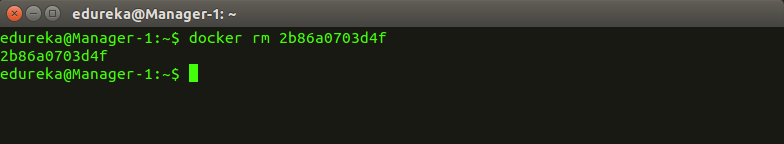
  
  
12. **docker images**

This command lists all the locally stored docker images

  
  
13. **docker rm**

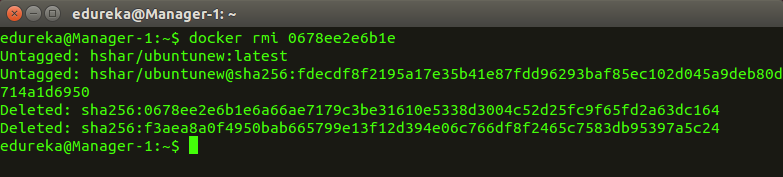
**Usage: docker rm <container id>**

This command is used to delete a stopped container

  
  
14. **docker rmi**

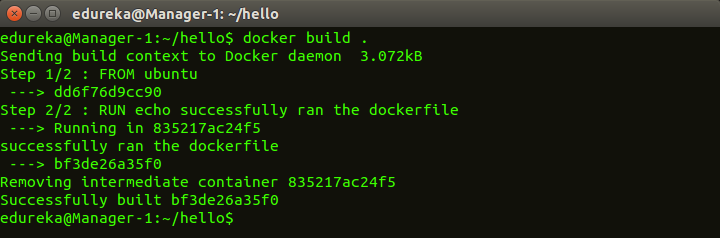
**Usage: docker rmi <image-id>**

This command is used to delete an image from local storage

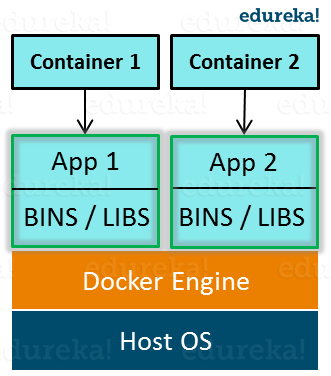
  
  
15. **docker build**

**Usage: docker build <path to docker file>**

This command is used to build an image from a specified docker file



Docker is a containerization platform that packages your application and all its dependencies together in the form of Containers to ensure that your application works seamlessly in any environment.



As you can see in the diagram on the right, each application will run on a separate container and will have its own set of libraries and dependencies. This also ensures that there is process level isolation, meaning each application is independent of other applications, giving developers surety that they can build applications that will not interfere with one another.

As a developer, I can build a container which has different applications installed on it and give it to my QA team who will only need to run the container to replicate the developer environment.

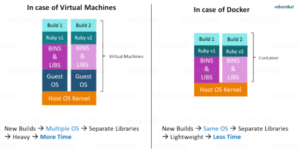
## ****Benefits of Docker****

Now, the QA team need not install all the dependent software and applications to test the code and this helps them save lots of time and energy. This also ensures that the working environment is consistent across all the individuals involved in the process, starting from development to deployment. The number of systems can be scaled up easily and the code can be deployed on them effortlessly.

## ****Virtualization vs Containerization****

Virtualization and Containerization both let you run multiple operating systems inside a host machine.

Virtualization deals with creating many operating systems in a single host machine. Containerization on the other hand will create multiple containers for every type of application as required.

**Figure:** What Is Big Data Analytics – Virtualization versus Containerization

As we can see from the image, the major difference is that there are multiple Guest Operating Systems in Virtualization which are absent in Containerization. The best part of Containerization is that it is very light weight as compared to the heavy virtualization.