**CHAPTER 1**

**INTRODUCTION**

Containers are a type of virtualization technology that enables the reuse of certain system resources that are not required to be duplicated. This can add a few different benefits depending on your needs. The most significant difference is that you are going to being able to spin up a higher quantity of containers on the same hardware or be able to assign more resources to your containers versus traditional virtual machines.

There are a number of components that are important for an OS to function properly. The parts we are going to focus on are the ones that are handled differently between traditional Virtual Machine (VM) configurations and container implementations. When power is run to a computer system (typically when the power button is turned on, or a piece of equipment is plugged in) the very first thing that happens is power runs to the NVRAM (Non-Volatile RAM), which is a particular kind of chip that stores a dedicated section of code called the BIOS (Basic Input/Output System). The BIOS is a set of code that loads a basic set of drivers that don’t need to be changed often. The BIOS is the reason you are able to use your keyboard and anything else that interfaces with the computer before the actual operating system is loaded. The BIOS is configured to look for an MBR (Master Boot Record) in the first section of a formatted drive.Despite not technically being part of the operating system,we can think of the bootloader as the first part of the OS. The bootloader sits on a special section of the boot disk called the MBR (Master Boot Record) and contains executable code that kicks off the booting process of the OS code. Some popular bootloaders that you may have seen are LILO (LInux LOader), LOADLIN (LOAD LINux), GRUB (GRand Unified Bootloader) and of course, the Windows bootloader. Bootloaders can point to multiple operating systems, or can even be chained together in some cases. This is most commonly done when dual-booting an OS where modifying the bootloader can cause problems. For example, running GRUB to run Linux, but passing to the Windows bootloader if the Windows OS is selected from the GRUB menu).The function of the bootloader is to load the OS Kernel. The Kernel is the core set of code that handles interfacing with the lower levels of the system and allows all of the abstraction that modern operating systems offer, such as graphical windowing systems (commonly referred to as GUI’s, Desktop Environments, or Window Managers, depending on which part is being referenced).The kernel of an operating system contains the core group of libraries and binaries required to run the OS. This set of code includes all of the underlying tools used by every other application running under the OS. This combination of code and resources is commonly referred to as Bins/Libs, for binaries and libraries.

In order to execute the bootloader and operating system in a traditional virtual machine environment, a Virtualization layer sits on top of the Host OS (the Operating system that you have installed onto the physical device). This layer then emulates hardware, presenting a virtualized version of a set of hardware to the Guest OS (the OS being installed inside of the Virtualization Layer). The virtualization layer essentially presents a fake version of physical hardware, copying how the hardware would usually function as closely as possible. This means you are presenting an emulated storage device for the boot loader, kernel, the OS bins/libs, and any other default applications installed by the Guest OS. Below is a simple diagram, but there will be differences depending on the type of virtualization. Notably Para virtualization, which is a specialized kernel that is configured to send commands to a hypervisor rather than directly to hardware. The nuanced differences are difficult to explain, but the core differences come down to the following. If the guest is aware that it is running in a virtualized environment, it purposefully routes requests to a hypervisor. If instead, the Host OS makes special allocations to run a higher execution ring, it allows the guest to execute in ring 0. This is really where containers shine, when we load a container layer, such as Docker Engine or Google Container Engine, the engine does not provide a typical virtualization. Instead, it provides the guest OS a wrapper to access the Host OS’s existing kernel, scheduler, and memory manager.

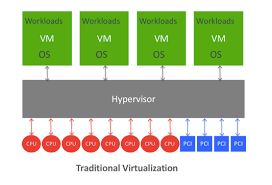
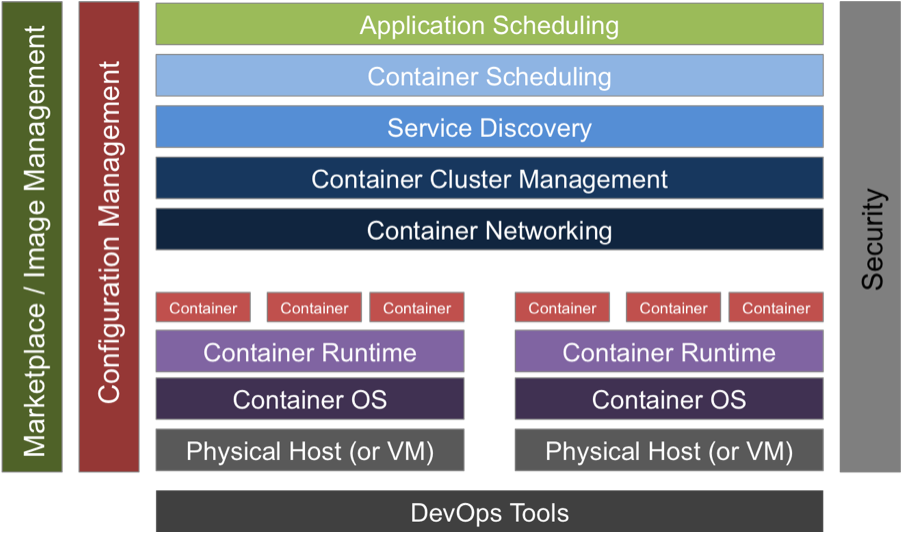


Figure 1.1: Traditional Virtualization

In reality, containers are not running a virtualized OS at all. Instead, they are allowed limited access to the existing kernel, binaries, and libraries that exist on the host operating system. The engine loads any additional bins/libs that are required by the application and groups the running processes of the container together, very similarly to the way that chroot jails function. The reason containers can run in this manner (as opposed to traditional styles of virtualization) is in large part due to cgroups and namespaces.

Figure 1.2: Container Architecture

Docker is an open-source project based on Linux containers. It uses Linux Kernel features like namespaces and control groups to create containers on top of an operating system.

Containers are far from new; Google has been using their own container technology for years. Others Linux container technologies include Solaris Zones, BSD jails, and LXC, which have been around for many years.

Docker is used for many reasons

1. **Ease of use:** Docker has made it much easier for anyone developers, systems admins, architects and others   to take advantage of containers in order to quickly build and test portable applications. It allows anyone to package an application on their laptop, which in turn can run unmodified on any public cloud, private cloud, or even bare metal. The mantra is: “build once, run anywhere.”

2. **Speed:** Docker containers are very lightweight and fast. Since containers are just sandboxed environments running on the kernel, they take up fewer resources. You can create and run a Docker container in seconds, compared to VMs which might take longer because they have to boot up a full virtual operating system every time.

3. **Docker Hub:** Docker users also benefit from the increasingly rich ecosystem of Docker Hub, which you can think of as an “app store for Docker images.” Docker Hub has tens of thousands of public images created by the community that are readily available for use. It’s incredibly easy to search for images that meet your needs, ready to pull down and use with little-to-no modification.

4. **Modularity and Scalability:** Docker makes it easy to break out your application’s functionality into individual containers. For example, you might have your Postgres database running in one container and your Redis server in another while your Node.js app is in another. With Docker, it’s become easier to link these containers together to create your application, making it easy to scale or update components independently in the future.

**1.1. Fundamentals of Docker**

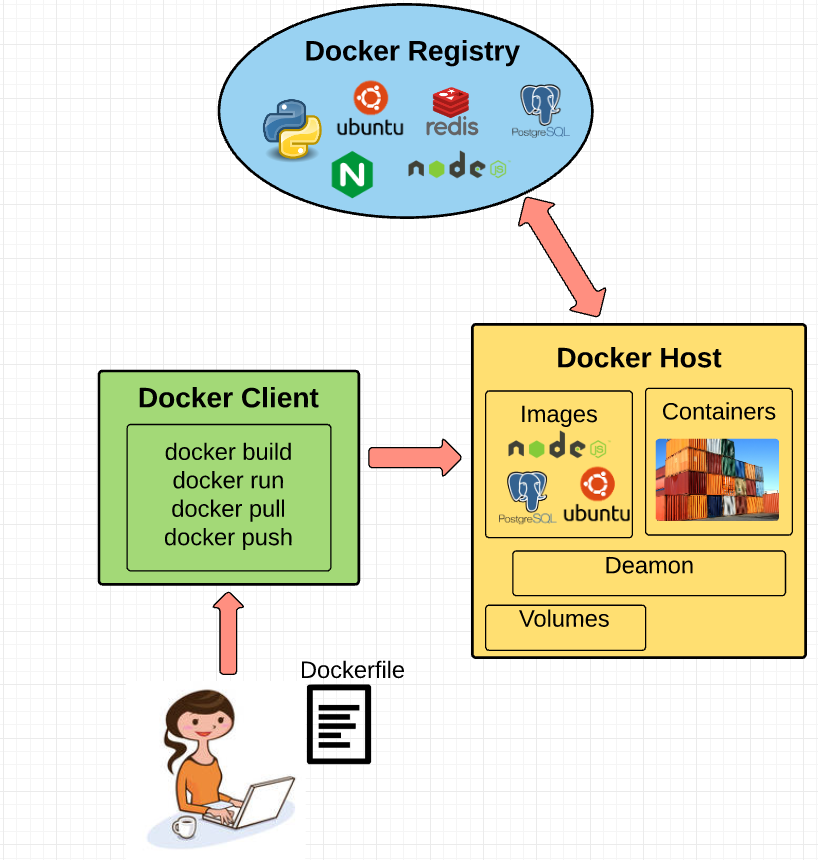


Figure.1.1.1: Fundamentals of Docker

Docker engine is the layer on which Docker runs. It’s a lightweight runtime and tooling that manages containers, images, builds, and more. It runs natively on Linux systems and is made up of:

1. A Docker Daemon that runs in the host computer.  
2. A Docker Client that then communicates with the Docker Daemon to execute commands.  
3. A REST API for interacting with the Docker Daemon remotely.

**1.1.1. Docker Client**

The Docker Client is what you, as the end-user of Docker, communicate with. Think of it as the UI for Docker. For example, when you do docker build iampeekay/someImage  you are communicating to the Docker Client, which then communicates your instructions to the Docker Daemon. Docker can be explained as a client and server based application, as depicted in Figure. The docker server gets the request from the docker client and then processes it accordingly. The complete RESTful (Representational state transfer) API and a command line client binary are shipped by docker. Docker daemon/server and docker client can be run on the same machine or a local docker client can be connected with a remote server or daemon, which is running on another machine

**1.1.2. Docker Daemon**

The Docker daemon is what actually executes commands sent to the Docker Client — like building, running, and distributing your containers. The Docker Daemon runs on the host machine, but as a user, you never communicate directly with the Daemon. The Docker Client can run on the host machine as well, but it’s not required to. It can run on a different machine and communicate with the Docker Daemon that’s running on the host machine.

**1.1.3. Docker Images**

There are two methods to build an image. The first one is to build an image by using a read-only template. The foundation of every image is a base image. Operating system images are basically the base images, such as Ubuntu 14.04 LTS, or Fedora 20. The images of operating system create a container with an ability of complete running OS. Base image can also be created from the scratch. Required applications can be added to the base image by modifying it, but it is necessary to build a new image. The process of building a new image is called “committing a change”. The second method is to create a docker file. The docker file contains a list of instructions when “Docker build” command is run from the bash terminal it follows all the instructions given in the docker file and builds an image. This is an automated way of building an image.

**1.1.4. Dockerfile**

A Dockerfile is where you write the instructions to build a Docker image. These instructions can be:

* **RUN apt-get y install some-package**: to install a software package
* **EXPOSE 8000:**to expose a port
* **ENV ANT\_HOME /usr/local/apache-ant** to pass an environment variable

and so forth. Once you’ve got your Dockerfile set up, you can use the **docker build** command to build an image from it.

**1.1.5. Docker Registries**

Docker images are placed in docker registries. It works correspondingly to source code repositories where images can be pushed or pulled from a single source. There are two types of registries, public and private. Docker Hub is called a public registry where everyone can pull available images and push their own images without creating an image from the scratch. Images can be distributed to a particular area (public or private) by using docker hub feature.

**1.1.6. Docker Containers**

Docker image creates a docker container. Containers hold the whole kit required for an application, so the application can be run in an isolated way. For example, suppose there is an image of Ubuntu OS with SQL SERVER, when this image is run with docker run command, then a container will be created and SQL SERVER will be running on Ubuntu OS.

**CHAPTER 2**

**A STUDY ON DOCKER CONTAINER**

Docker is an open source platform that runs applications and makes the process easier to develop, distribute. The applications that are built in the docker are packaged with all the supporting dependencies into a standard form called a container. These containers keep running in an isolated way on top of the operating system’s kernel. The extra layer of abstraction might effect in terms of performance [1]

**Babak Bashari Rad et.al [2]** proposedan Introduction to Docker and Analysis of its Performance in 2017. This concept explains about the Docker, which provides some facilities that are useful for developers and administrators. It is an open platform can be used for building, distributing, and running applications in a portable, lightweight runtime and packaging tool, known as Docker Engine. It also provides Docker Hub, which is a cloud service for sharing applications. Costs can be reduced by replacing traditional virtual machine with docker container. It excellently reduces the cost of re-building the cloud development platform.

**R. R. Yadav et.al [3]** proposedaperformance comparison between virtual Machines and docker containers. The main goal of this paper is to conduct a comparative study of the performance evaluation of virtual machines and containers. In order to accomplish this, a methodology is proposed to evaluate the performance of Docker containers and virtual machines. Additionally, a real-world case study is proposed to illustrate the applicability of the methodology approach. For all the experiments, the docker container showed lower execution times for the requests against the virtual machine.

**Ilias Mavridis and Helen Karatza [4]** introduced performance and overhead study of containers running on top of virtual machines. In this work, they have conducted several experiments to study how the container performance is affected by the additional virtualization layer of the Virtual Machine. Docker is been used to deploy and run the containers, KVM hypervisor for the Virtual Machines and executed a series of well-known benchmarks on different operating systems.

**Enrico Bacis et.al [5]** proposedDocker Policy Modules: Mandatory Access Control for Docker Containers, In order to increase docker security and flexibility, they have proposed an extension to the Dockerfile format to let image maintainers ship a specific SELinux policy for the processes that run in a Docker image, enhancing the security of containers.

**2.1 Docker vs. other Container Technology**

Seo et al. [2014] summarize that there is no guest OS of docker in the cloud, so the storage and the wastage of CPU resources are less [6]. The images are not disturbed; boot time is faster and the time of generating the images is short. These are the benefits of docker cloud in comparison with VM Cloud. They used two similar servers with the same configuration in the cloud environment. One server was used for docker and the other one was for an Open Stack platform for KVM by means of a virtualization tool. Ubuntu Server was used as a base platform.

To calculate the approximate boot-time, 20 images were generated on each server and boot time was checked. Figure 4 shows that the boot time of docker is lesser than the boot time of KVM. Docker uses the Host OS, whereas KVM uses Guest OS. Thus, the boot time of docker is shorter than the boot time of KVM.

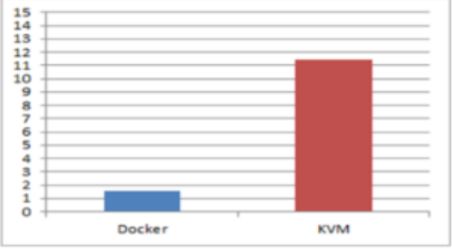
****

Fig 2.1.1 docker vs KVM average boot time

To calculate the operational speed, python language was used. Figure 2.2 shows that operation speed of 100,000 is averagely around 4.5s. To measure the operation speed, they obtain the average process time and standard deviation, by repeating the same process 100 times on docker and VM.

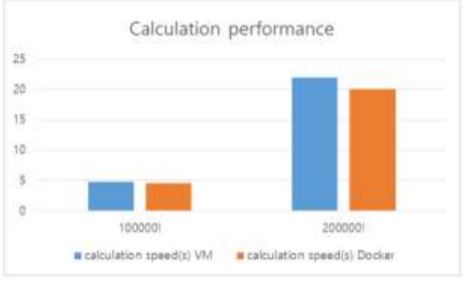
****

Fig 2.1.2 CPU calculation performance

Figure 2.2 shows the calculation speed of docker is slightly faster than the calculation speed of the VM. Seo et al. (2014) concluded that VM works independently. This is one of the reasons that it is easy to apply and manage the policy of network, security, user, and the system. However, docker does not contain a guest operating System. Therefore, it takes very less time in distributing and gathering images. Its boot time is also very short. These are the main advantages of utilizing docker cloud as compared with VM Cloud.

**2.2 Virtual Machines vs. Containers**

Table 1 compare features of different containerized and virtual machine technologies. Virtual machine uses an extra layer between the host operating system and guest operating system. This layer is known as a Hypervisor. Whereas docker adds up an extra layer between host operating systems and where the applications are virtualized and executed, which is known as a Docker Engine. As docker does not use any guest operating system that makes a big difference in performance between a docker container and a virtual machine technology. In Table 2.2.1, the performances of applications running in different containers and virtual machines are also briefly compared. As it is given in the table above, according to Seo et al. (2014) the docker performance is better than KVM, in terms of boot time and calculation speed [6], whereas Felter et al. (2014) proves that there is no difference of wastage of resources (overhead) between Docker and KVM but there is a noticeable difference in execution, as KVM is faster than Docker [7]. Scheepers (2014) found out that LXC takes a longer time to accomplish tasks, whereas Xen Server takes less time [8]. LXC is better in the sense of fewer wasted resources while Xen is better in the sense of equally distributing resources.

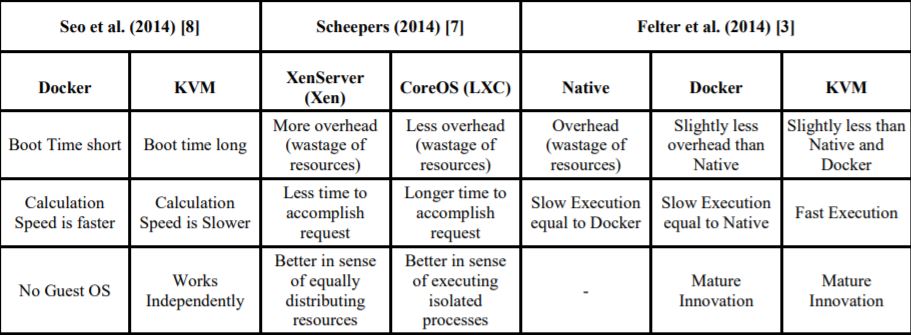
****

Table 2.2.1: Comparison Table based on Different Virtual Machines and Containerized Technology

**CHAPTER 3**

**GETTING STARTED WITH DOCKER**

To start working with docker the docker community version has been installed. The platform we are working is Ubuntu 16.04 version. After the installation we perform the following steps to ensure that our docker has been installed correctly.

3.1. Test docker version

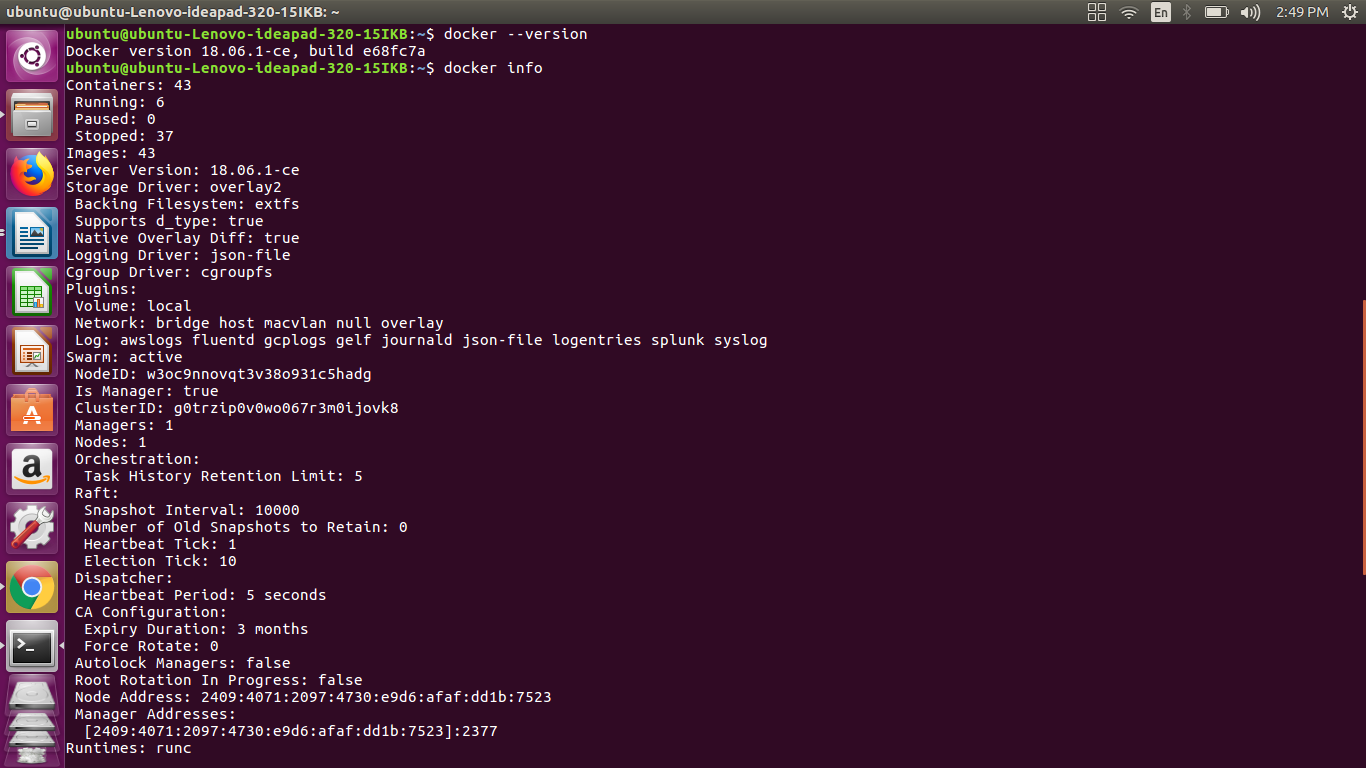
* We run *docker –version* and ensure that we have a supported version of Docker:
* ****Run *docker info*  to view even more details about your docker installation

Figure 3.1.1: Checking for docker version.

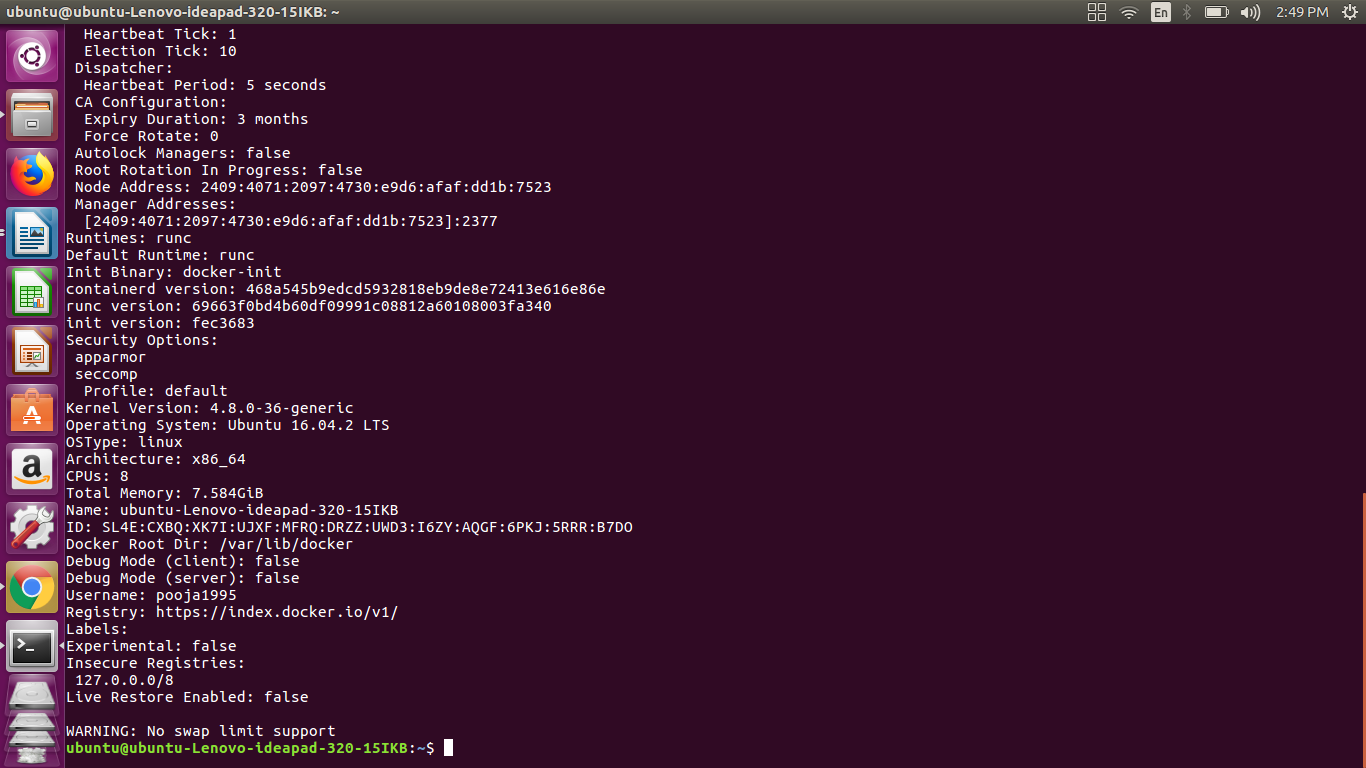


Figure 3.1.2: Dockerinfo

3.2. Test Docker installation

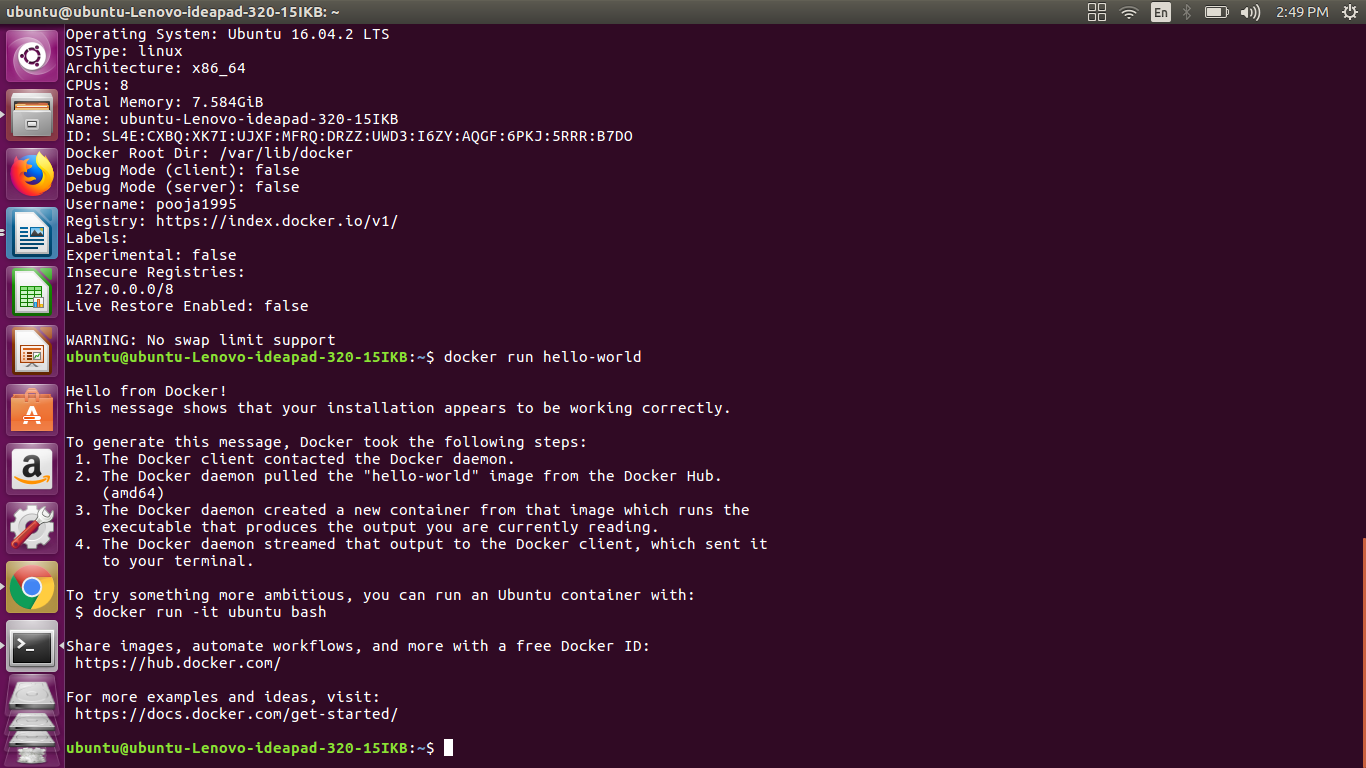
Test that your installation works by running the simple Docker image, [hello-world](https://hub.docker.com/_/hello-world/)

Figure 3.2.1: checking for successful installation

Listing the hello-world image that was downloaded to the machine

docker image ls

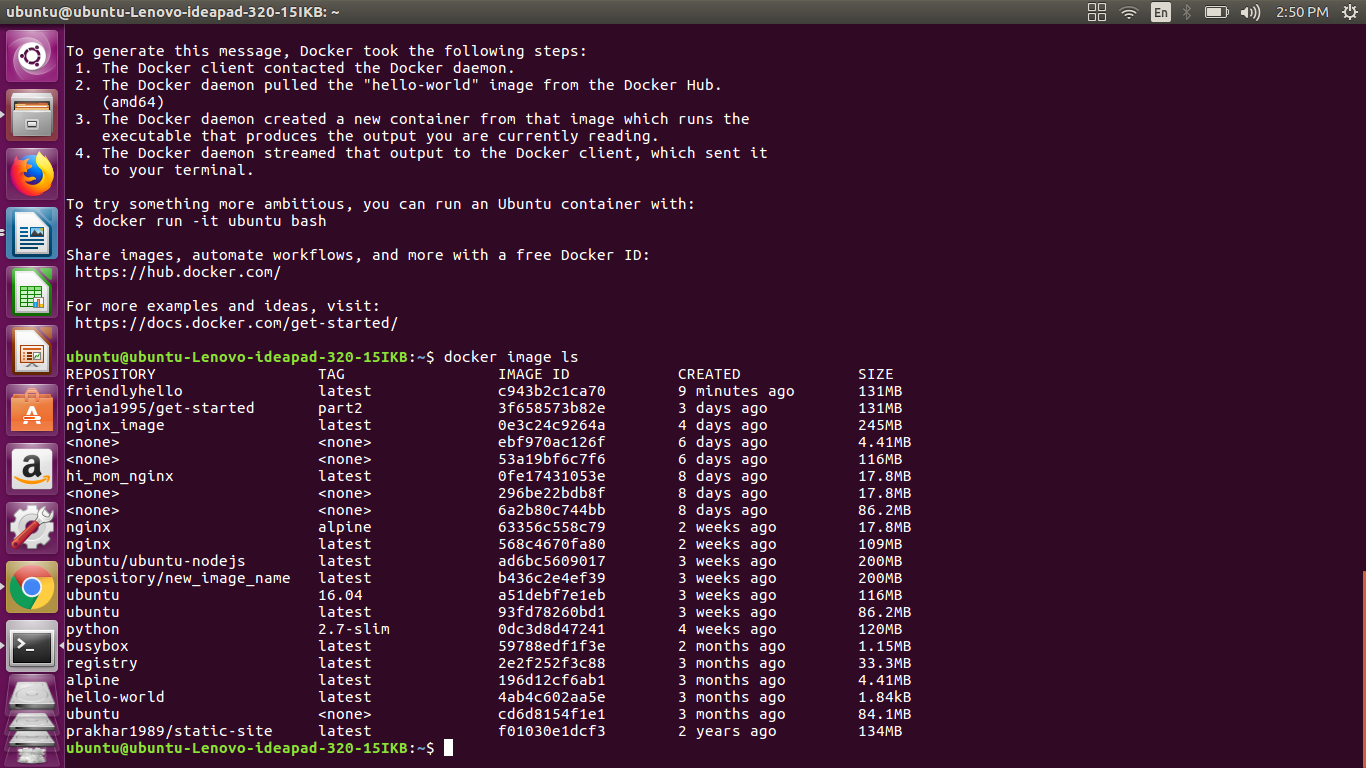


Figure 3.2.2: listing the downloaded images.

Listing the hello-world container (spawned by the image) which exits after displaying its message. If it were still running, we would not need the –all option

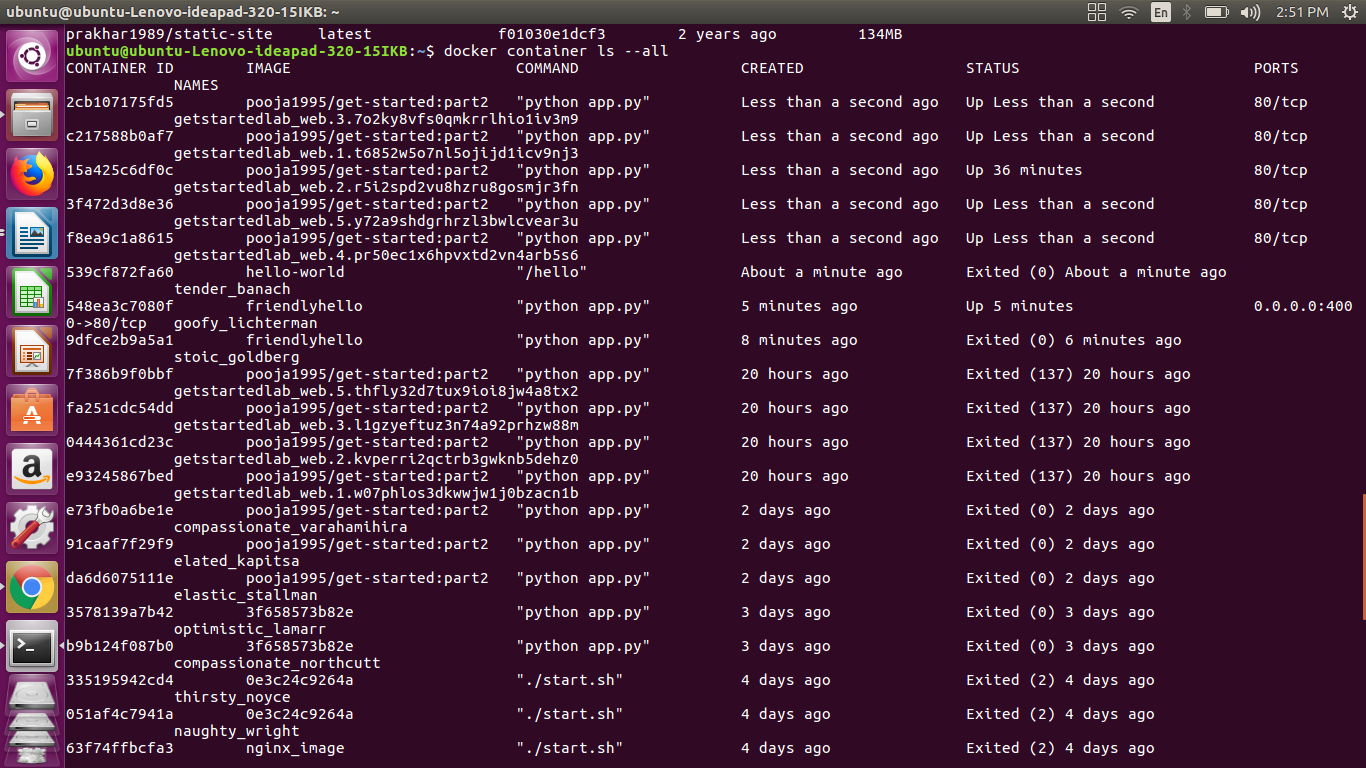
docker container ls –all

Figure 3.2.3: listing containers

3.3. Dockerfile

In the past, if we were to start writing a Python app, our first order of business was to install a Python runtime onto our machine. But, that creates a situation where the environment on our machine needs to be perfect for our app to run as expected, and also needs to match our production environment.

With Docker, we can just grab a portable Python runtime as an image, no installation necessary. Then, our build can include the base Python image right alongside our app code, ensuring that our app, its dependencies, and the runtime, all travel together. These portable images are defined by something called a Dockerfile.

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

Creating a Dockerfile

* We 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
* Save it.

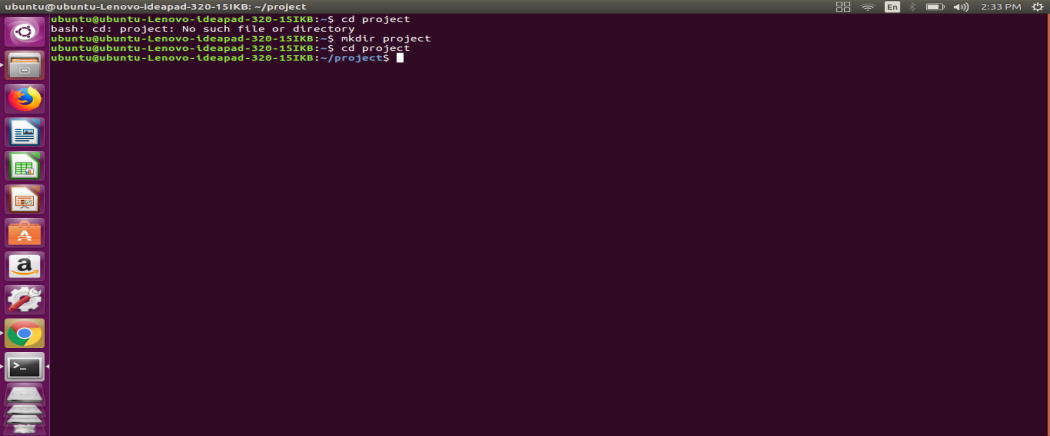


Figure 3.3.1: creating a directory

* # 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 requirements.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. we will Create two more files, requirements.txt and app.py, and put them in the same folder with the Dockerfile. This completes our app. When the above Dockerfile is built into an image, app.py and requirements.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

fromflaskimportFlask

fromredisimportRedis,RedisError

importos

importsocket

# Connect to Redis

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

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

@app.route(“/”)

defhello():

try:

visits=redis.incr(“counter”)

exceptRedisError:

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

html=“<h3>Hello {name}!</h3>” \

“<b>Hostname:</b> {hostname}<br/>” \

“<b>Visits:</b> {visits}”

returnhtml.format(name=os.getenv(“NAME”,“world”),hostname=socket.gethostname(),visits=visits)

if\_\_name\_\_==“\_\_main\_\_”:

app.run(host=’0.0.0.0’,port=80)

**Build the app**

We are ready to build the app. Here’s what ls should show:

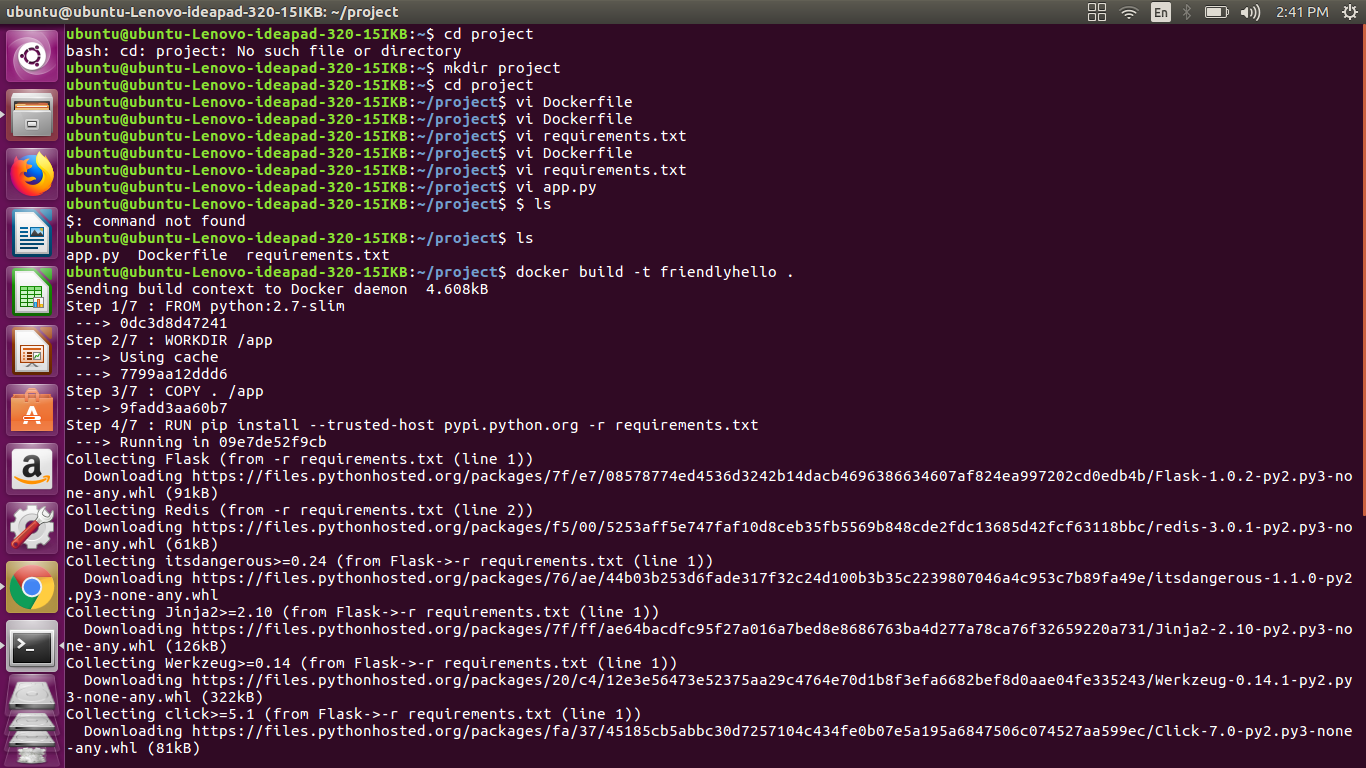
$ ls

Dockerfile app.py requirements.txt

Now we will run the build command. This creates a Docker image, which we’re going to tag using –t so it has a friendly name.

docker build –t friendlyhello .

The execution of the build command is shown below.

Figure 3.3.2: creating docker image

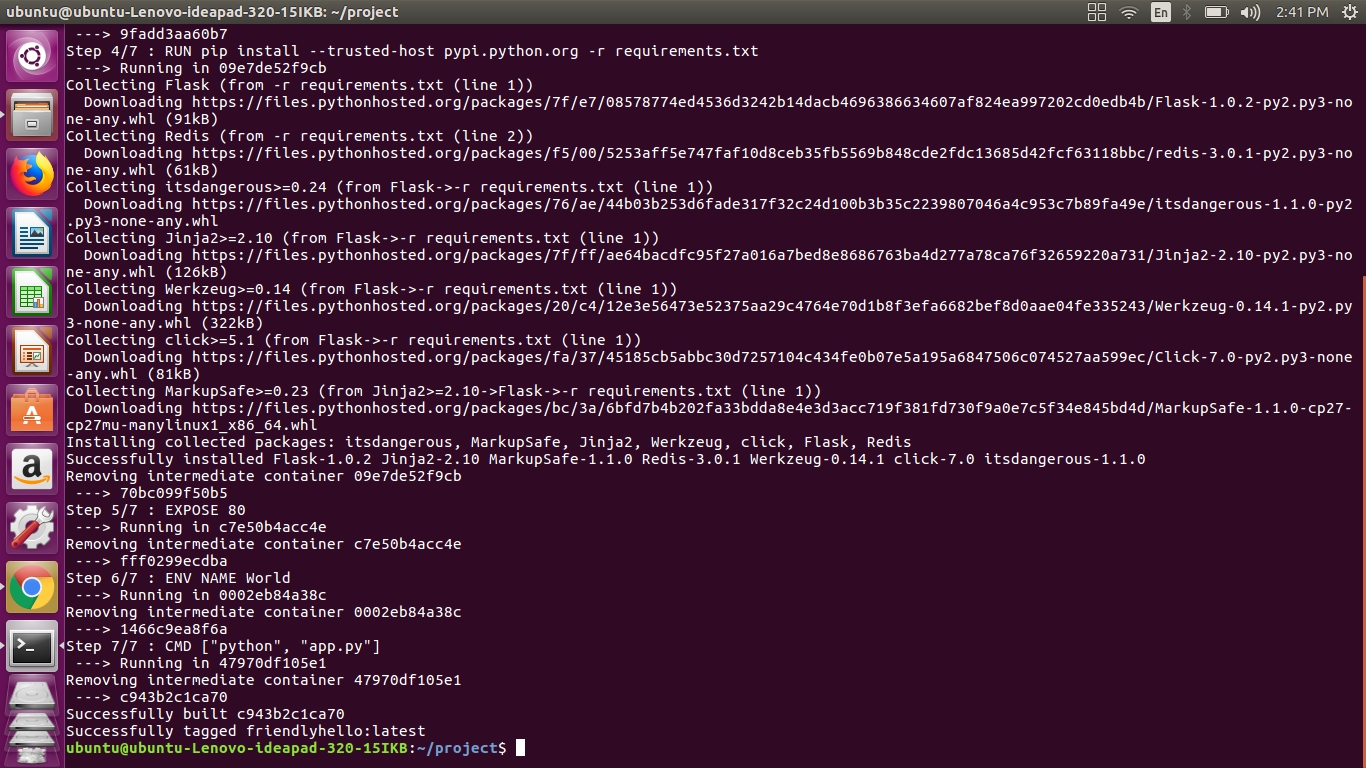


Figure 3.3.3: creating docker image

Where is our built image? It’s in your machine’s local Docker image registry:Run the app

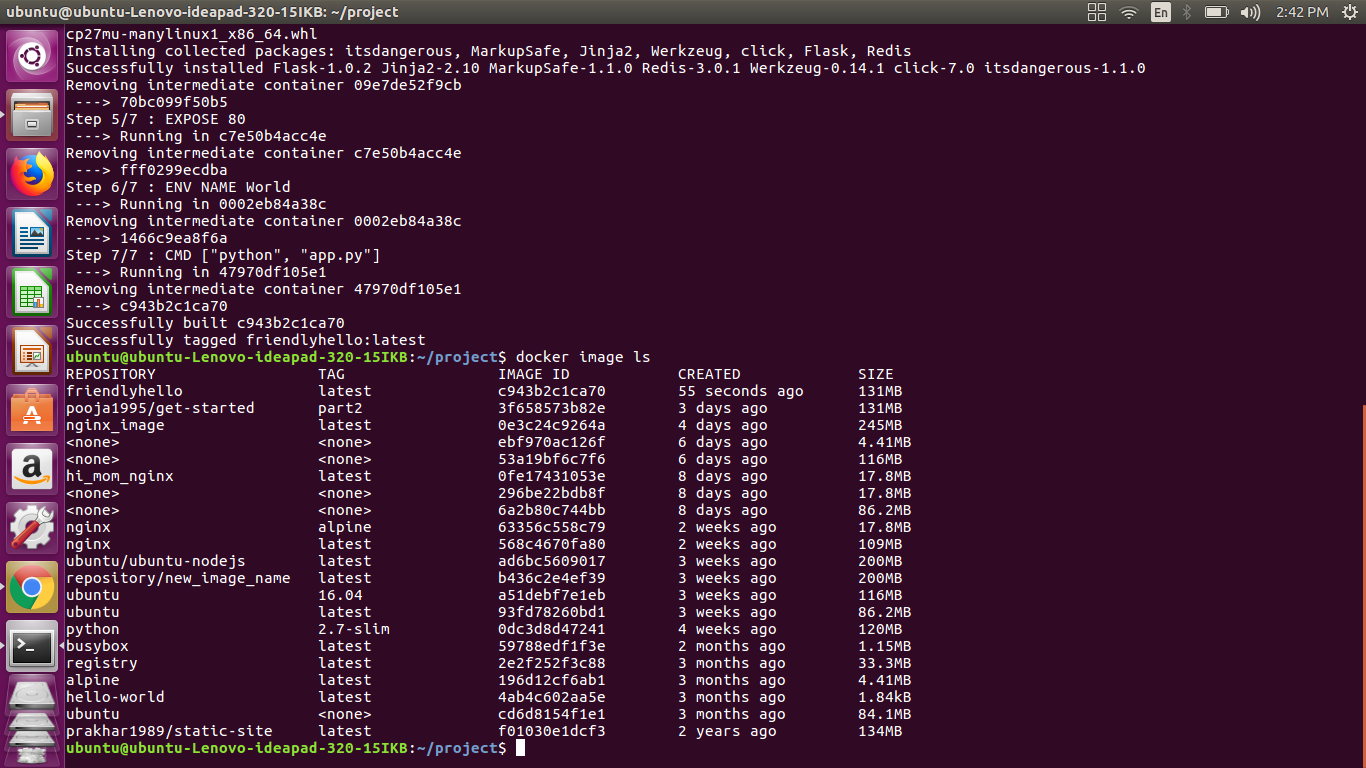


Figure 3.3.4: list of docker images

Running the app, mapping our machine’s port 4000 to the container’s published port 80 using –p:

docker run –p 4000:80 friendlyhello

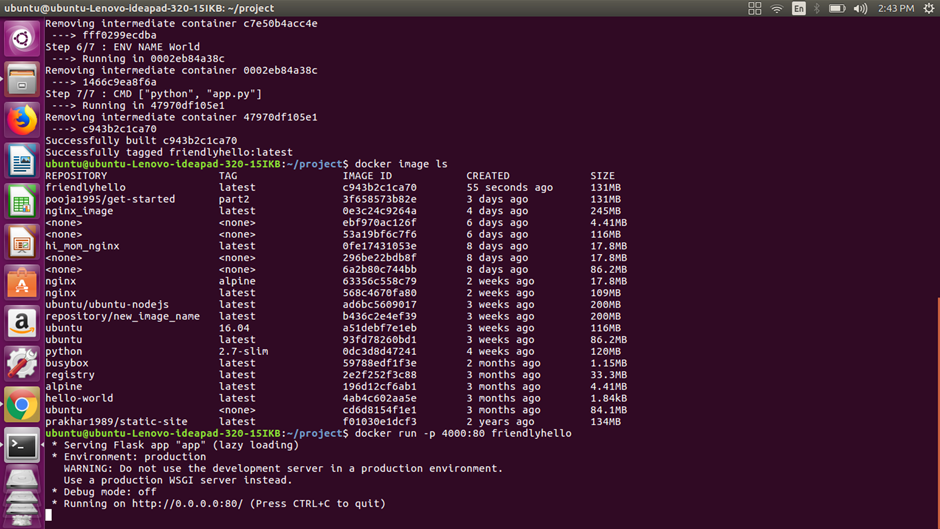


Figure 3.3.5: running the image

We will get to see a message that Python is serving our app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn’t know we have mapped port 80 of that container to 4000, making the correct URL [http://localhost:4000](http://localhost:4000/).

We will Go to that URL in a web browser to see the display content served up on a web page.

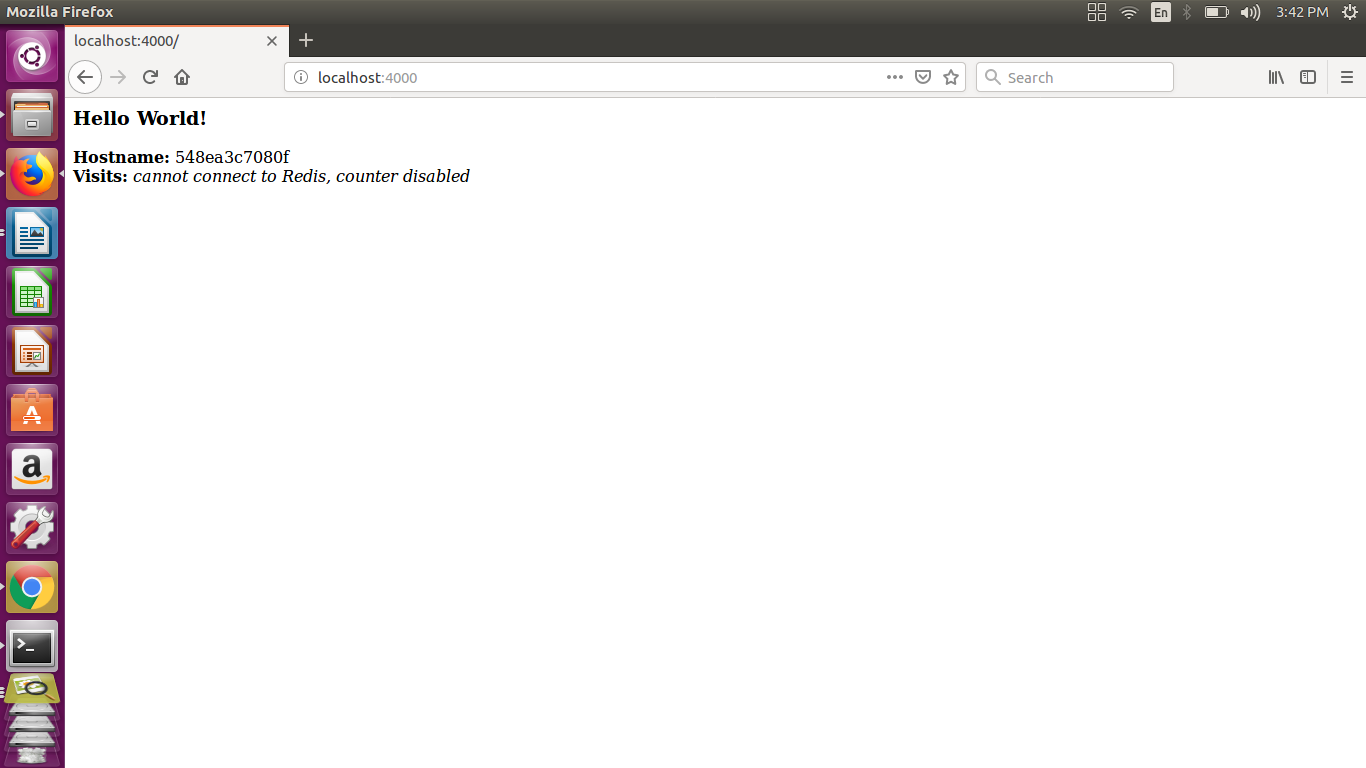


Figure 3.3.6: Display content served up on a web page.

We can also use the curl command in a shell to view the same content.

$ curl <http://localhost:4000>

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b><i>cannot connect to Redis, counter disabled</i>

Now we use docker container stop to end the process, using the CONTAINER ID, like so:

docker container stop 1fa4ab2cf395

**3.3.1. Sharing the image**

To demonstrate the portability of what we just created, let’s upload our built image and run it somewhere else. After all, we need to know how to push to registries when we want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of imagessort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker’s public registry by default.

We will log in with our Docker ID

We will log in to the Docker public registry on our local machine.

$ docker login

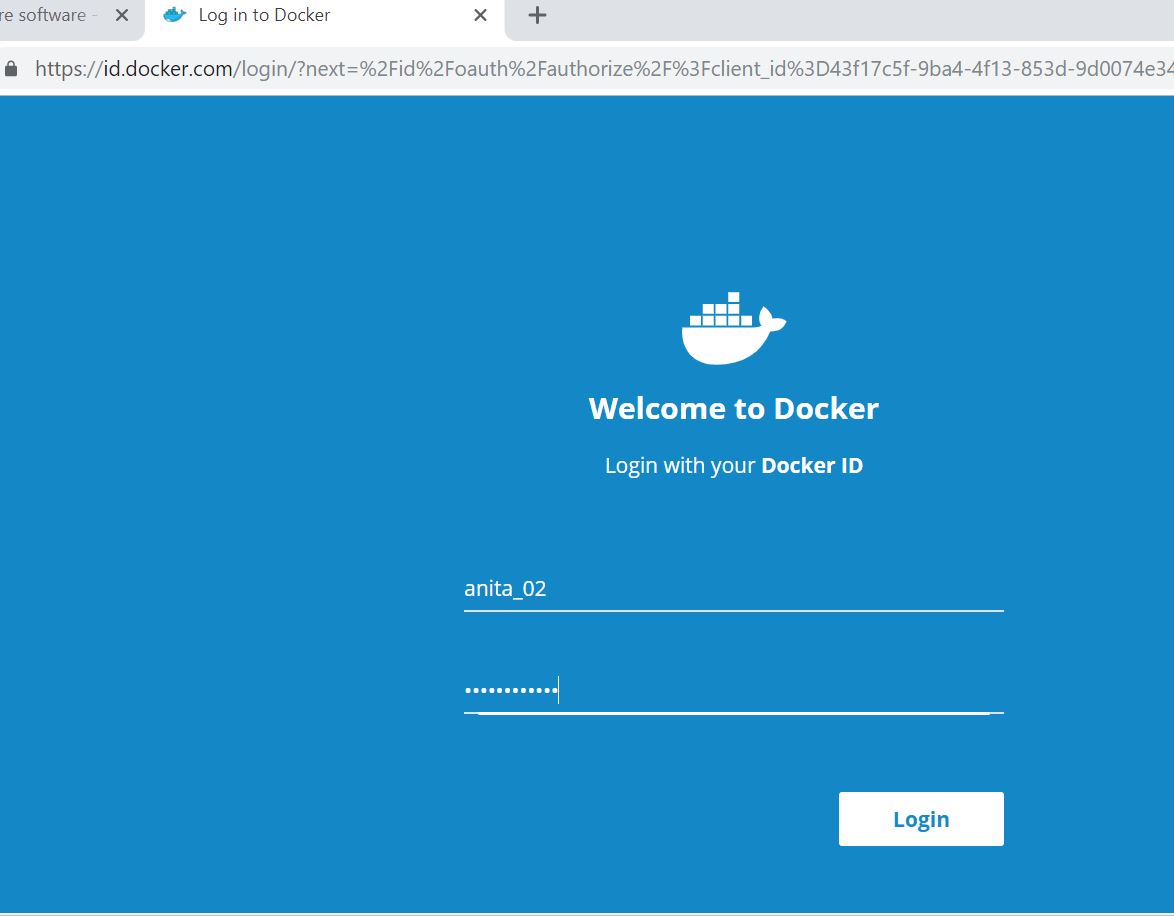


Figure 3.3.1.1: docker registry login page

3.3.2. Tag the image

The notation for associating a local image with a repository on a registry is username/repository:tag. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. We will give the repository and tag meaningful names for the context, such asget-started:part2. This puts the image in the get-started repository and tags it as part2.

Now, we put it all together to tag the image. Run docker tag image with our username, repository, and tag names so that the image uploads to our desired destination. The syntax of the command is:

docker tag image username/repository:tag

For example:

docker tag friendlyhello anitamg/get-started:part2

Here anitamariet will the id used to login to the docker hub and get-strarted:part2 will be the repository name we are tagging to.

3.3.3. Publish the image

Upload your tagged image to the repository:

docker push username/repository: tag

Once complete, the results of this upload are publicly available. If we log in to [Docker Hub](https://hub.docker.com/), we see the new image there, with its pull command as shown in below image.

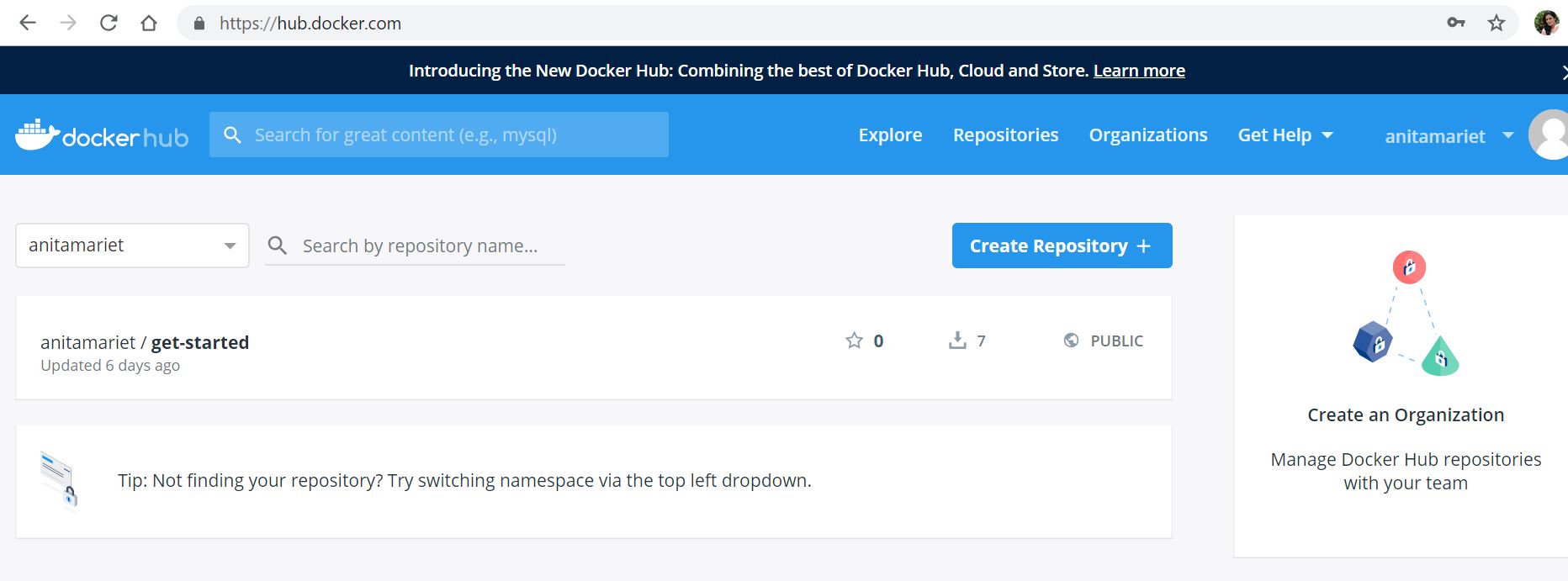


Figure 3.3.3.1: docker hub repository

**CHAPTER 4**

**CONCLUSION**

A major benefit of containers is their portability. A container wraps up an application with everything it needs to run, like configuration files and dependencies.Since containers do not require a separate operating system, they use up less resources. While a VM often measures several gigabytes in size, a container usually measures only a few dozen megabytes, making it possible to run many more containers than VMs on a single server. Since containers have a higher utilisation level with regard to the underlying hardware, we require less hardware, resulting in a reduction of bare metal costs as well as datacentre costs.Although containers run on the same server and use the same resources, they do not interact with each other. If one application crashes, other containers with the same application will keep running flawlessly and won’t experience any technical problems. This isolation also decreases security risks: If one application should be hacked or breached by malware, any resulting negative effects won’t spread to the other running containers.As mentioned before, containers are lightweight and start in less than a second since they do not require an operating system boot. Creating, replicating or destroying containers is also just a matter of seconds, thus greatly speeding up the development process, the time to market and the operational speed. Releasing new software or versions has never been so easy and quick. But the increased speed also offers great opportunities for improving customer experience, since it enables organisations and developers to act quickly, for example when it comes to fixing bugs or adding new features.

[Docker](https://www.docker.com/) is by far the most popular containerisation platform, but to successfully adopt containers we will also need to implement a container orchestration system. [Kubernetes](http://kubernetes.io/), based on over 10 years of experience in running containerised workloads at Google, is the clear market leader in container orchestration. In the next work of the project kubernetes will be used and implemented

**REFERENCES**

|  |  |
| --- | --- |
| [1] | PH. Kamp, R. Watson. Jails: Conﬁning the omnipotent root. In Proceedings of the 2nd International SANE Conference, volume 43, page 116, 2000. |
| [2] | Rad, Babak Bashari, Harrison John Bhatti, and Mohammad Ahmadi. "An Introduction to Docker and Analysis of its Performance." *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017): 228. |
| [3] | Yadav, R. R., E. T. G. Sousa, and G. R. A. Callou. "Performance Comparison Between Virtual Machines and Docker Containers." *IEEE Latin America Transactions* 16.8 (2018): 2282-2288. |
| [4] | Mavridis, Ilias, and Helen Karatza. "Performance and Overhead Study of Containers Running on Top of Virtual Machines." *Business Informatics (CBI), 2017 IEEE 19th Conference on*. Vol. 2. IEEE, 2017. |
| [5] | Bacis, Enrico, et al. "DockerPolicyModules: mandatory access control for docker containers." *Communications and Network Security (CNS), 2015 IEEE Conference on*. IEEE, 2015. |
| [6] | Seo, K.-T., Hwang, H.-S., Moon, I.-Y., Kwon, O.-Y., & Kim, B.-J. (2014). Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud. |
| [7] | Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2014). An updated performance comparison of virtual machines and linux containers. technology, 28, 32. |
| [8] | Scheepers, M. J. (2014). Virtualization and containerization of application infrastructure: A comparison. |
| [9] | OpenVZ. Openvz virtuozzo containers wiki. https://openvz. Org/Main Page, November 2016. (Accessed on 11/16/2016). |
| [10] | PARALLELS. An introduction to operating system virtualization and parallels cloud server. https: //virtuozzo.com/wp-content/uploads/2016/03/Virtuozzo Intro OS Virtualization WP Ltr 2013 March2016.pdf, 2016. (Accessed on 11/16/2016). |
| [11] | Virtuozzo. Virtuozzo – containers, vms, storage virtualization. https://virtuozzo.com/, 2016. (Accessed on 11/16/2016). |
| [12] | Z. Kozhirbayev, R.O. Sinnott. A performance comparison of container-based technologies for the cloud. Future Generation Computer Systems, 68:175–182, 2016. |
| [13] | Canonical Ltd. Linux containers – lxc – documentation. https: //linuxcontainers.org/lxc/documentation/,2016. (Accessedon 11/24/2016). |
| [14] | S. Graber. Lxc 1.0 release. https://lists.linuxcontainers.org/ pipermail/lxc-devel/2014-February/008165.html, 2014. (Accessed on 11/25/2016). |
| [15] | Canonical Ltd. Linux containers – lxd – introduction. https: //linuxcontainers.org/lxd/introduction/, 2016. (Accessed on 11/24/2016 |
| [16] | Docker. Docker engine frequently asked questions (faq) docker. https://docs.docker.com/engine/faq/, 2016. (Accessed on 11/24/2016). |