CHAPTER 1

INTRODUCTION

Linux containers are often considered as something in the middle between a chroot and a full-fledged virtual machine. The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel.

Linux containers, in short, contain applications in a way that keep them isolated from the host system that they run on. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. And they are designed to make it easier to provide a consistent experience as developers and system administrators move code from development environments into production in a fast and replicable way.

Containers behave like a virtual machine. To the outside world, they can look like their own complete system. But unlike a virtual machine, rather than creating a whole virtual operating system, containers don't need to replicate an entire operating system, only the individual components they need in order to operate. This gives a significant performance boost and reduces the size of the application. They also operate much faster, as unlike traditional virtualization the process is essentially running natively on its host, just with an additional layer of protection around it.

1.1 Why containers?

Undoubtedly, one of the biggest reasons for recent interest in container technology has been the Docker open source project, a command line tool that made creating and working with containers easy for developers and sysadmins alike, similar to the way Vagrant made it easier for developers to explore virtual machines easily.

Docker is a command-line tool for programmatically defining the contents of a Linux container in code, which can then be versioned, reproduced, shared, and modified easily just as if it were the source code to a program.

Containers have also sparked an interest in micro service architecture, a design pattern for developing applications in which complex applications 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. Each piece, or service, can live inside of a container, and can be scaled independently of the rest of the application as the need arises.

1.2 Architecture of Linux containers

Several components are needed for Linux Containers to function correctly, most of them are provided by the Linux kernel. Kernel *namespaces* ensure process isolation and *cgroups* are employed to control the system resources. *SELinux* is used to assure separation between the host and the container and also between the individual containers. *Management interface* forms a higher layer that interacts with the above mentioned kernel components and provides tools for construction and management of containers.

The following scheme illustrates the architecture of Linux Containers:

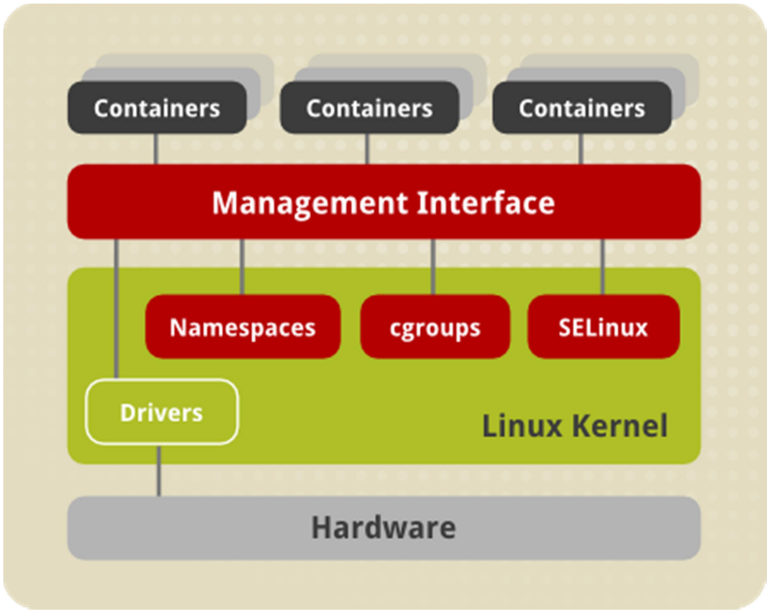

Fig 1.1: Architecture of Linux Containers

1.3 Docker

Docker is gaining popularity and its usage is spreading like wildfire. The reason for Docker's growing popularity is the extent to which it can be used in an IT organization. Very few tools out there have the functionality to find itself useful to

both developers and as well as system administrators. Docker is one such tool that truly lives up to its promise of Build, Ship and Run.

In simple words, Docker is a software containerization platform, meaning you can build your application, package them along with their dependencies into a container and then these containers can be easily shipped to run on other machines.

For example: Let's consider a linux based application which has been written both in Ruby and Python. This application requires a specific version of linux, Ruby and Python. In order to avoid any version conflicts on user's end, a linux docker container can be created with the required versions of Ruby and Python installed along with the application. Now the end users can use the application easily by running this container without worrying about the dependencies or any version conflicts.

These containers use Containerization which can be considered as an evolved version of Virtualization. The same task can also be achieved using Virtual Machines, however it is not very efficient.

1.3.1 What is Virtualization?

Virtualization is the technique of importing a Guest operating system on top of a Host operating system. This technique was a revelation at the beginning because it allowed developers to run multiple operating systems in different virtual machines all running on the same host. This eliminated the need for extra hardware resource. The advantages of Virtual Machines or Virtualization are:

- Multiple operating systems can run on the same machine
- Maintenance and Recovery were easy in case of failure conditions
- Total cost of ownership was also less due to the reduced need for infrastructure

As you know nothing is perfect, Virtualization also has some shortcomings. Running multiple Virtual Machines in the same host operating system leads to performance degradation. This is because of the guest OS running on top of the host OS, which will have its own kernel and set of libraries and dependencies. This takes up a large chunk of system resources, i.e. hard disk, processor and especially RAM.

Another problem with Virtual Machines which uses virtualization is that it takes almost a minute to boot-up. This is very critical in case of real-time applications.

Following are the disadvantages of Virtualization:

- Running multiple Virtual Machines leads to unstable performance
- Hypervisors are not as efficient as the host operating system
- Boot up process is long and takes time

These drawbacks led to the emergence of a new technique called Containerization.

1.3.2 What is Containerization?

Containerization is the technique of bringing virtualization to the operating system level. While Virtualization brings abstraction to the hardware, Containerization brings abstraction to the operating system. Do note that Containerization is also a type of Virtualization. Containerization is however more efficient because there is no guest OS here and utilizes a host's operating system, share relevant libraries & resources as and when needed unlike virtual machines. Application specific binaries and libraries of containers run on the host kernel, which makes processing and execution very fast. Even booting-up a container takes only a fraction of a second. Because all the containers share, host operating system and holds only the application related binaries & libraries. They are lightweight and faster than Virtual Machines.

1.3.3 Advantages of Containerization over Virtualization:

- Containers on the same OS kernel are lighter and smaller
- Better resource utilization compared to VMs
- Boot-up process is short and takes few seconds

Containers only contain application specific libraries which are separate for each container and they are faster and do not waste any resources.

All these containers are handled by the containerization layer which is not native to the host operating system. Hence software is needed, which can enable you to create & run containers on your host operating system.

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

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.

1.3.4 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.

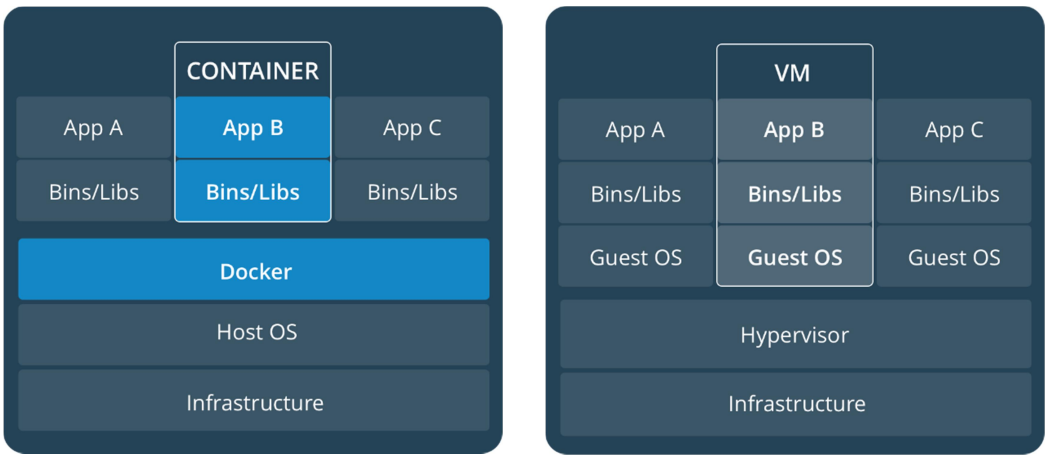

Fig 1.2 Virtualization versus containerization

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.

1.4 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.

1.5 Docker concepts

Docker is a platform for developers and sysadmins to **develop, deploy, and run** applications with containers. The use of Linux containers to deploy applications is called *containerization*. Containers are not new, but their use for easily deploying applications is.

Containerization is increasingly popular because containers are:

- Flexible: Even the most complex applications can be containerized.
- Lightweight: Containers leverage and share the host kernel.
- Interchangeable: You can deploy updates and upgrades on-the-fly.
- Portable: You can build locally, deploy to the cloud, and run anywhere.
- Scalable: You can increase and automatically distribute container replicas.
- Stackable: You can stack services vertically and on-the-fly.

1.6 Images and containers

A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.

A **container** is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, `docker ps`, just as you would in Linux.

CHAPTER 2

WORK DONE TILL PROJECT PART -1

To start working with docker, the maintained version of a docker community edition (CE) has been installed. The platform we are working is Ubuntu 16.04 version. After the installation we perform the following steps to ensure that the docker has been installed correctly.

2.1 Testing docker version

- Run docker --version and ensuring that we have a supported version of Docker:

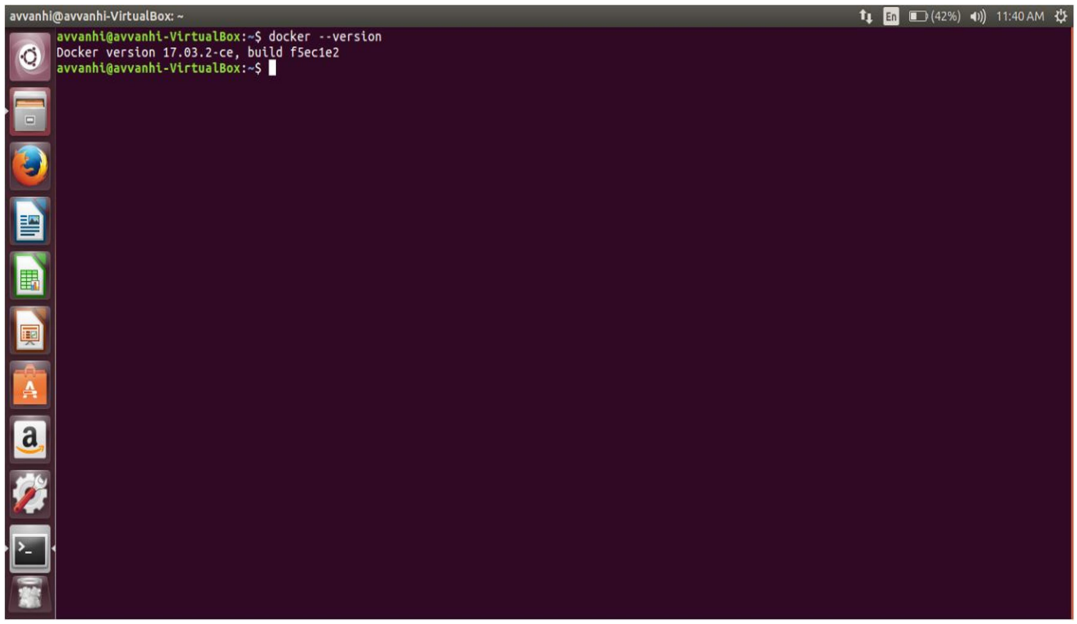

Fig 2.1: Testing docker version

- Run docker info to view even more details about docker installation:

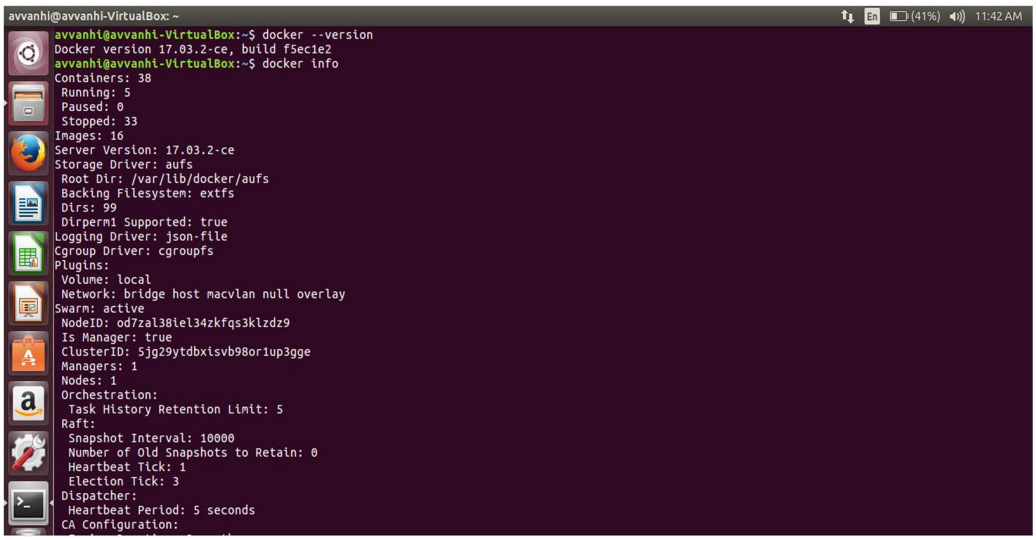

Fig 2.2 Docker info

2.2 Test Docker installation

Test that the installation works by running the simple Docker image, hello-world:

- docker run hello-world

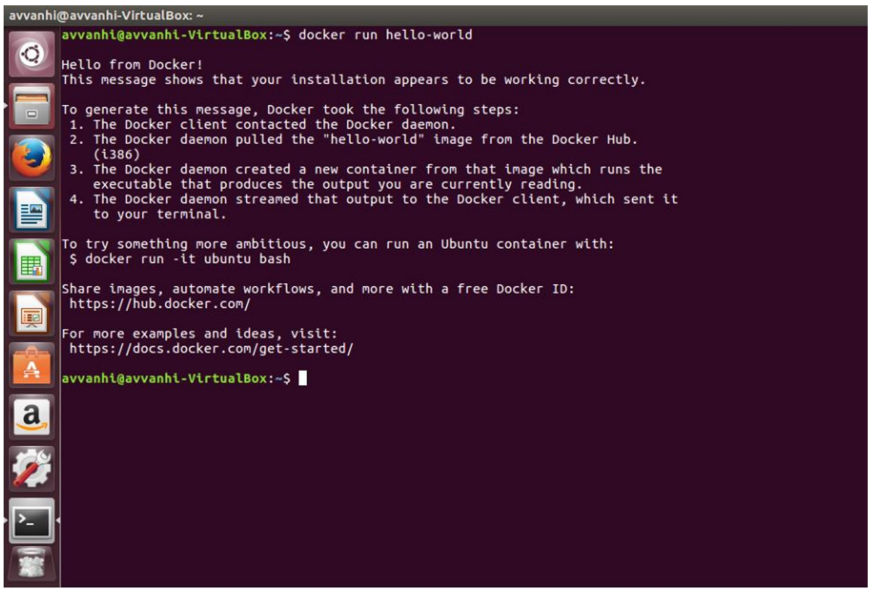

Fig 2.3: Testing docker installation

- List the hello-world image that was downloaded to the machine using: docker image ls

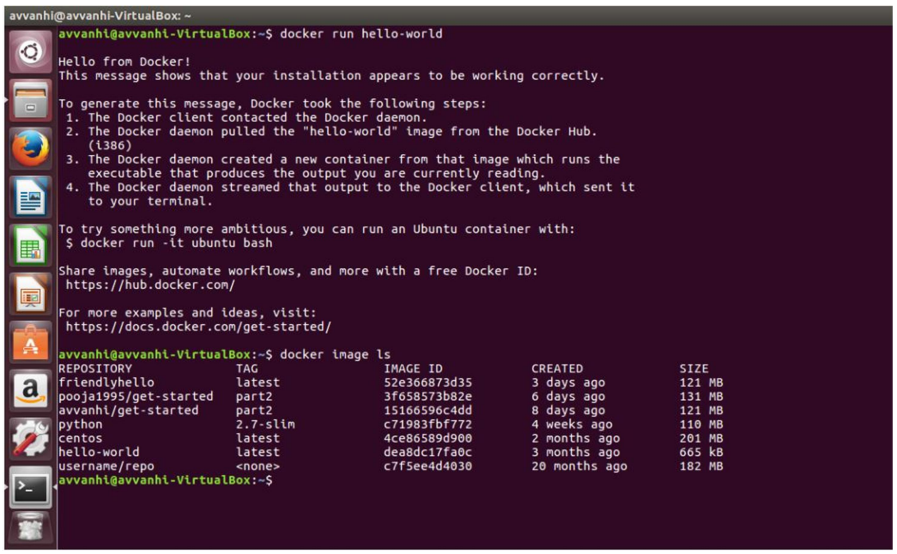

Fig 2.4: Listing of docker images

- List the hello-world container using the command

  docker container ls –all

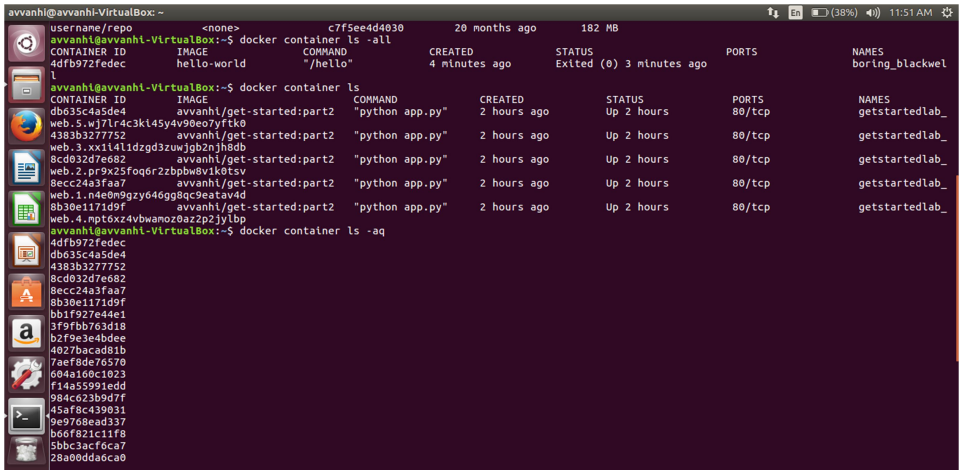

Fig 2.5: Listing of docker containers

2.3 Development environment

If we have to start writing a Python app, we first need to install a Python runtime onto the machine. But, that creates a situation where the environment on the 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.

2.4 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 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.

2.4.1 Working with Dockerfile

- Create an empty directory.
- Change directories (cd) into the new directory.
- Create a file called Dockerfile.
- The Dockerfile consists of following code snippets:

```
# 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 namely app.py and requirements.txt.
- 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

```
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.

2.5 Build the app

We are ready to build the app. Make sure you are still at the top level of your new directory. Here's what ls should show:

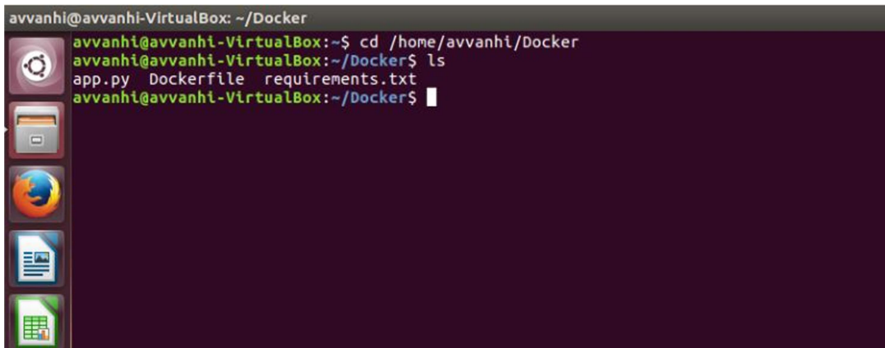

Fig 2.6: Showing the level of directory

- Now 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 .

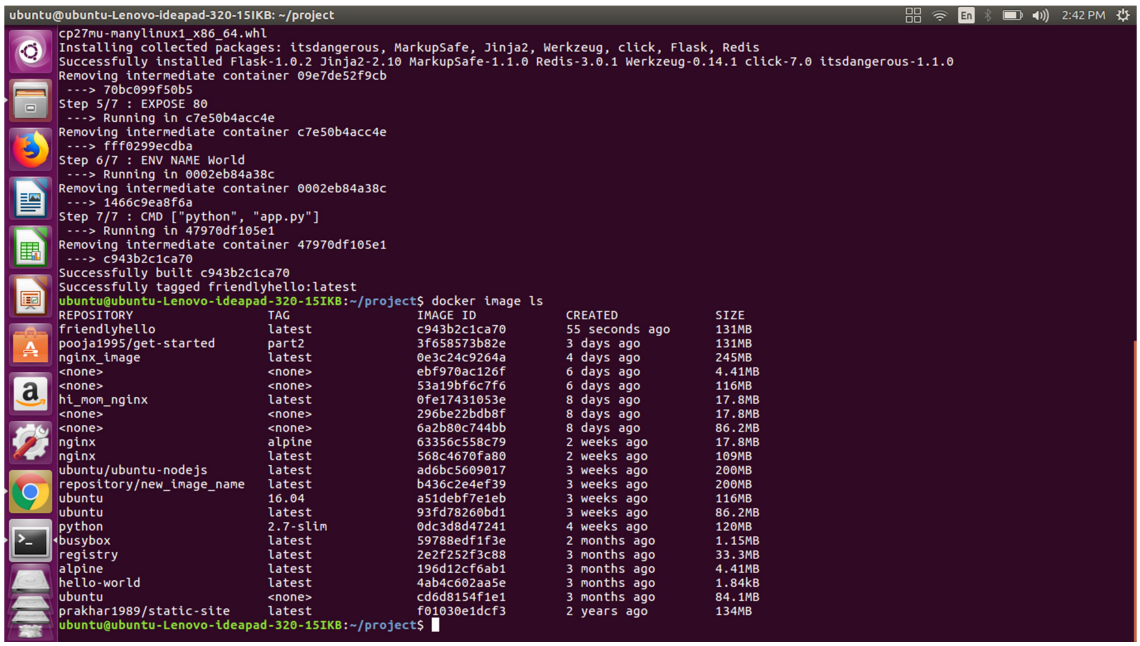

Fig 2.7: Listing of docker images

2.6 Run the app

Run the app, mapping our machine's port 4000 to the container's published port 80 using -p:

```
docker run -p 4000:80 friendlyhello
```

We should 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 that we mapped port 80 of that container to 4000, making the correct URL http://localhost:4000.

If we go to that URL in a web browser we can see the display content served up on a web page.

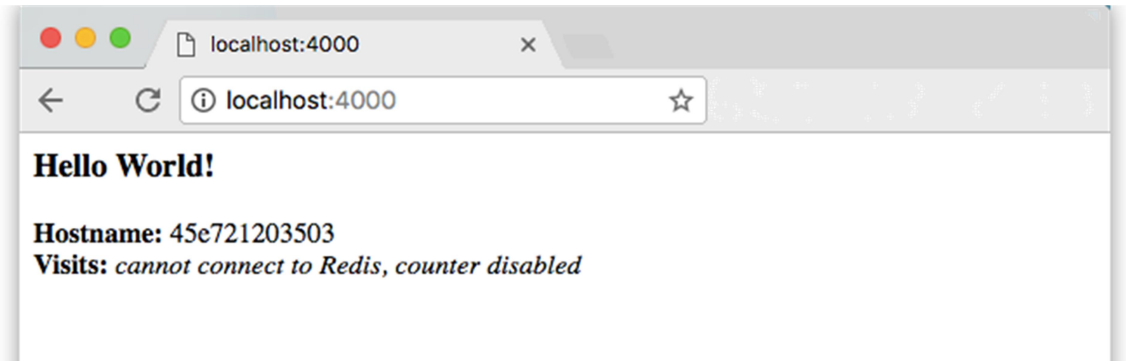

Fig 2.8: Content in a web browser

This port remapping of 4000:80 demonstrates the difference between EXPOSE within the Dockerfile and what the publish value is set to when running docker run -p. In later steps, map port 4000 on the host to port 80 in the container and use http://localhost.

To run the app in the background, in detached mode we use:

```
docker run -d -p 4000:80 friendlyhello
```

When we execute this we get the long container ID for our app and then are kicked back to our terminal. Our container is running in the background. We can also see the abbreviated container ID with docker container ls(and both work interchangeably when running commands):

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED
1fa4ab2cf395        friendlyhello       "python app.py"     28 seconds ago
```

If we notice CONTAINER ID that matches what's on http://localhost:4000. We use docker container stop to end the process, using the CONTAINER ID, like:

```
docker container stop 1fa4ab2cf395
```

2.7 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, you need to know how to push to registries when you want to deploy containers to production.

A registry is a collection of repositories, and a repository is a collection of images sort 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 to the Docker public registry on our local machine.

```
$ docker login
```

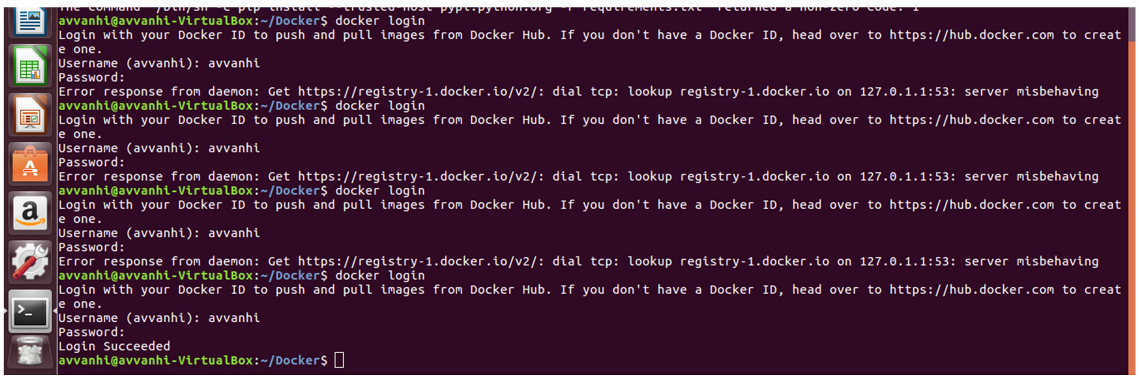

Fig 2.9: Docker login to docker public registry on local machine

2.8 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 avvanhi/get-started:part2
```

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

```
docker tag image username/repository:tag
```

Run docker image ls to see your newly tagged image.

```
$ docker image ls
```

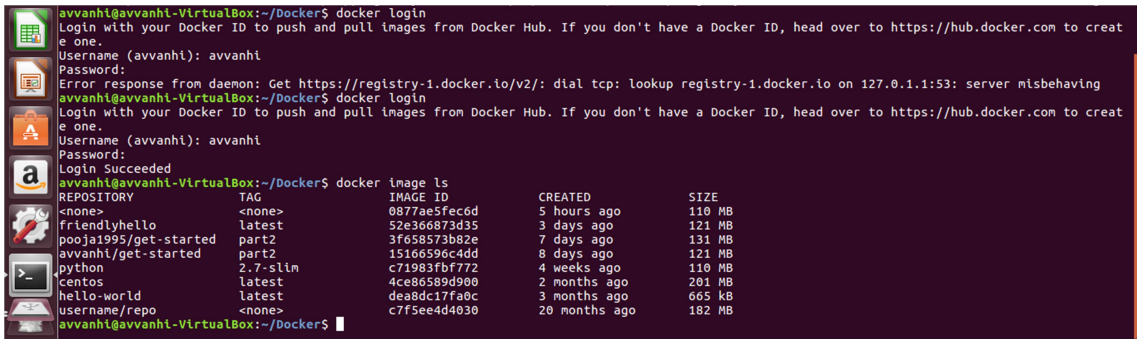

Fig 2.10: Listing of docker images

- Publish the image

Here we upload our 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, we can see the new image there, with its pull command.

- Pull and run the image from the remote repository

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

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

If the image isn't available locally on the machine, Docker pulls it from the repository.

No matter where docker run executes, it pulls image, along with Python and all the dependencies from requirements.txt, and runs the code. It all travels together in a neat little package, and we don't need to install anything on the host machine for Docker to run it.

CHAPTER 3

WORK CARRIED OUT IN PHASE II

The main aim of the project is to provide containers to the customers, in order to do this we need a certain medium to communicate with the customers. For this purpose we are creating our own simple user interface which can be used by our customers to communicate with us. Our website contains different sections such as home, about, services, contact and login page. Below is the brief explanation of each section.

3.1 Index

The index page contains different sections like home, about, services, contact and login. Whenever a customer enters into our website he will be able to view the different sections. Whenever he clicks on to the menus, the customer will be directed to the particular section. The simplicity of our webpage which consumes less time, because of the single page view.

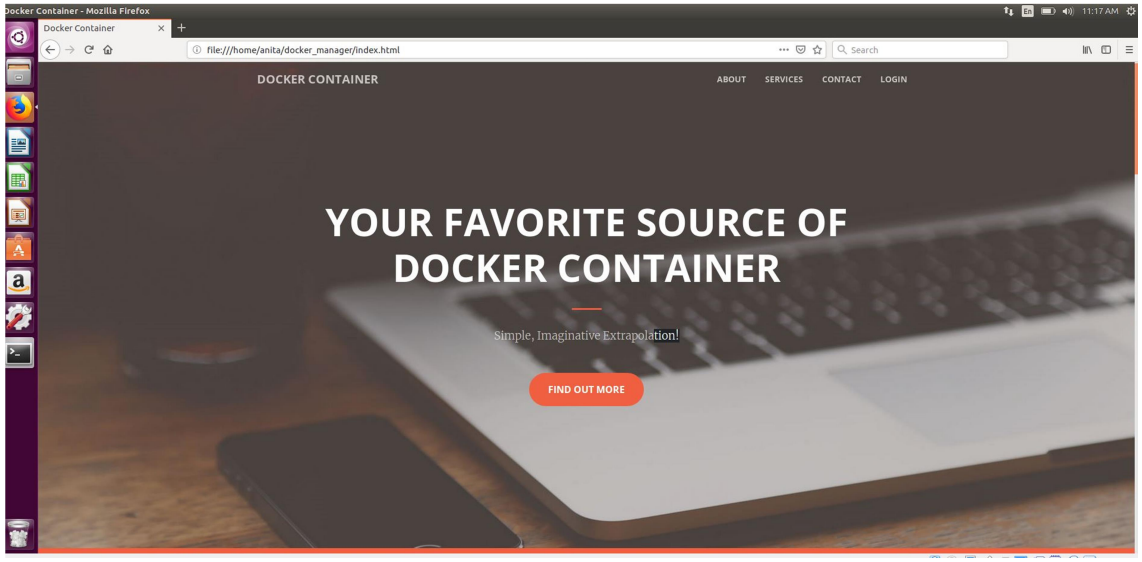

Fig 3.1: Initial page of our website.

3.2 About

As all company website contains the introductory part of the company, our website also contains the same. We will also include the benefits that the customers will receive from our company if they buy containers from us and also why our company is a better container provider when compared to the other companies.

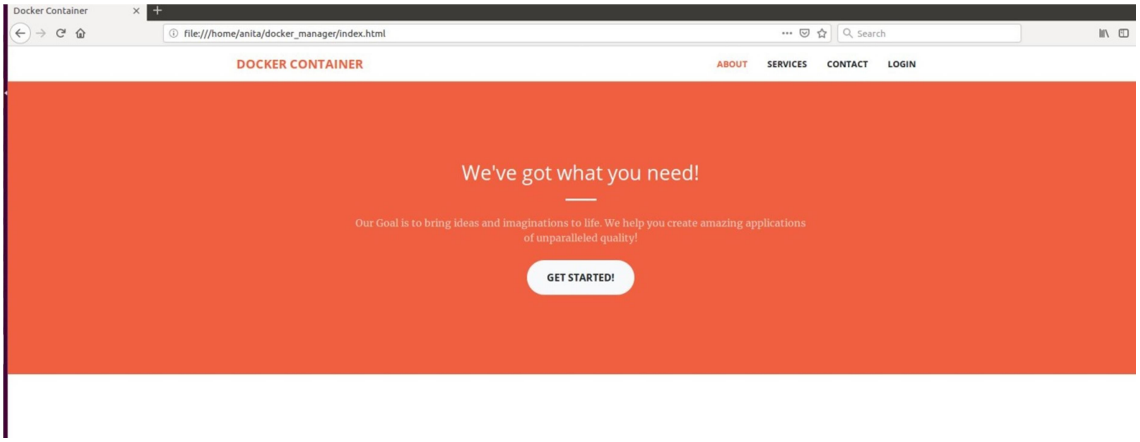

Fig 3.2: About Section

3.3 Service

This section will contain the services provided by our company. The main service we provide is the distribution of the containers.

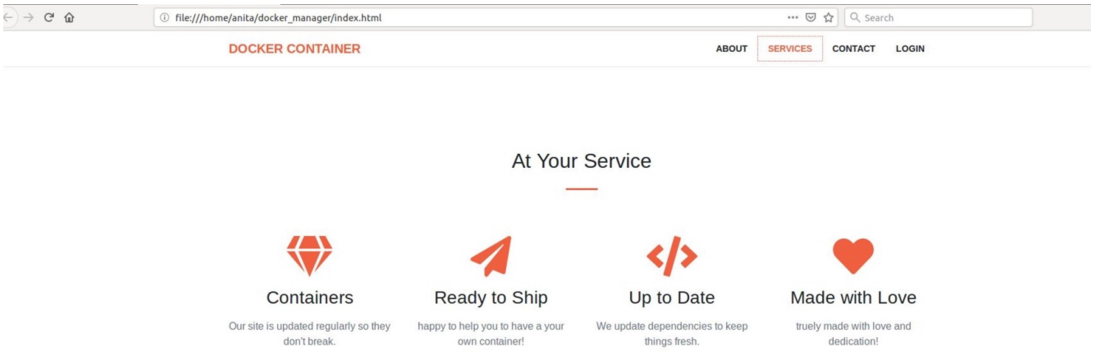

Fig 3.3 Service section

3.4 Contact

This section contains the contact details of the company to get in touch with us for the end users.

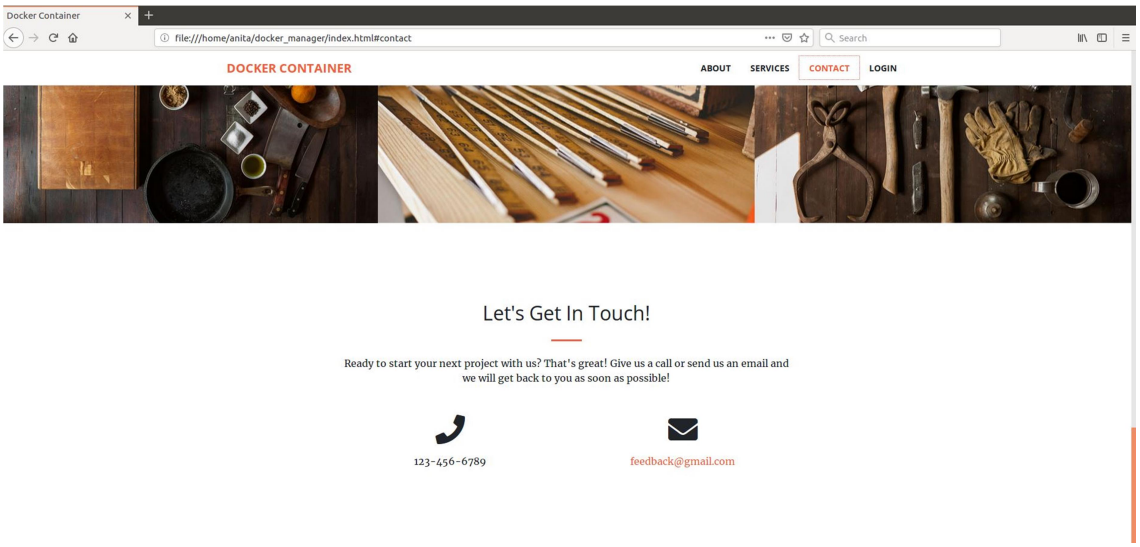

Fig 3.4 Contact section

3.5 Login

The login page will allow the customer to request his needs.

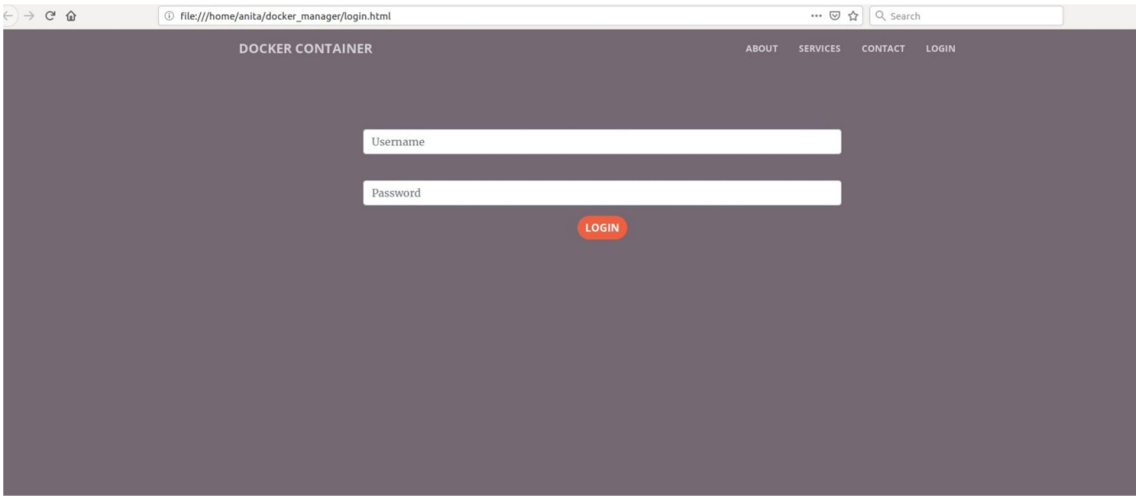

Fig 3.5 Login page

3.6 Configuration

After the customer logs into his account he will be directed to the configuration page. Where he can submit his desired requirements. Such as number of replicas he needs to create, cpu and memory usage.

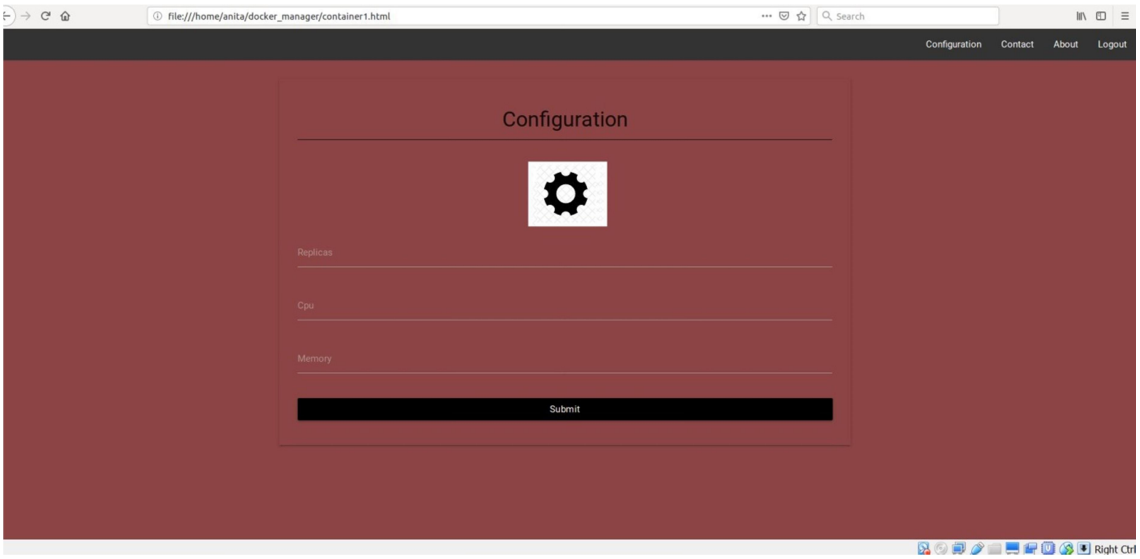

Fig 3.6 Configuration

3.7 Saved data

The data submitted by the customers about their container requirements will be saved in the following file.

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: anita
    deploy:
      replicas: 23
      resources:
        limits:
          cpus: "50"
          memory: 60
      restart_policy:
        condition: on-failure
    ports:
      - "80:80"
    networks:
      - webnet
networks:
  webnet:
```

Fig 3.7 Saved data

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 is by far the most popular containerisation platform, but to successfully adopt containers we will also need to implement a container orchestration system. Kubernetes 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] David Bernstein, Containers and Cloud: From LXC to Docker to Kubernetes, http://www.computer.org/cloudcomputing, (September 2014).

[2] Docker Introduction, https://www.docker.com.

[3] OpenVZ. Openvz virtuozzo containers wiki. https://openvz.org/MainPage, November 2016. (Accessed on 11/16/2016).

[4] Parallels, An introduction to operating system virtualization and parallels cloud server. https://virtuozzo.com/wp-content/uploads/2016/03/VirtuozzoIntro OSVirtualizationWPLtr2013March2016.pdf, 2016. (Accessed on 11/16/2016).

[5] Virtuozzo. Virtuozzo – containers, vms, storage virtualization. https://virtuozzo.com/, 2016. (Accessed on 11/16/2016).

[6] PH. Kamp, R. Watson. Jails: Confining the omnipotent root. In Proceedings of the 2nd International SANE Conference, volume 43, page 116, 2000.

[7] A. Gholami, E. Laure. Security and privacy of sensitive data in cloud computing: a survey of recent developments. ArXiv preprint arXiv:1601.01498, 2016.

[8] E. Reshetova, J. Karhunen, T. Nyman, N Asokan. Security of oslevel virtualization technologies. In Nordic Conference on Secure IT Systems, pages 77–93. Springer, 2014.

[9] DevOps.com, ClusterHQ, State-of-container-usage, june-2016.pdf.https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf, 2016. (Accessed on 11/25/2016).

[10] Linux-Vserver. Linux-vserver. http://www.linux-vserver.org, June 2013. (Accessed on 11/16/2016).

[11] Canonical Ltd. Linux containers – lxc – documentation. https: //linuxcontainers.org/lxc/documentation/,2016. (Accessedon 11/24/2016).

[12] S. Graber. Lxc 1.0 release. https://lists.linuxcontainers.org/pipermail/lxc-devel/2014-February/008165.html, 2014. (Accessed on 11/25/2016).

[13] Canonical Ltd. Linux containers – lxd – introduction. https://linuxcontainers.org/lxd/introduction/, 2016. (Accessed on 11/24/2016).

[14] D. Price, A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In LISA, volume 4, pages 241–254, 2004.

[15] J.Victor, SolarisTM containers technology architecture guide. Sun Microsystems, pages 91–115, 2006.

[16] ClusterHQ. The current state of container usage. https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2015.Pdf, 2015. (Accessed on 11/30/2016).

[17] Docker. Docker engine frequently asked questions (faq) docker. https://docs.docker.com/engine/faq/, 2016. (Accessed on 11/24/2016).

[18] Z. Kozhirbayev, R.O. Sinnott. A performance comparison of container-based technologies for the cloud. Future Generation Computer Systems, 68:175–182, 2016.