**CHAPTER 1**

**INTRODUCTION**

Docker Container is a standardized unit which can be created on the fly to deploy a particular application or environment. It could be an Ubuntu container, CentOs container, etc. to full-fill the requirement from an operating system point of view. Also, it could be an application oriented container like CakePHP container or a Tomcat-Ubuntu container etc. 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.

**CHAPTER 2**

**WORK DONE IN PROJECT PART- 1**

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.

2.1. Test docker version

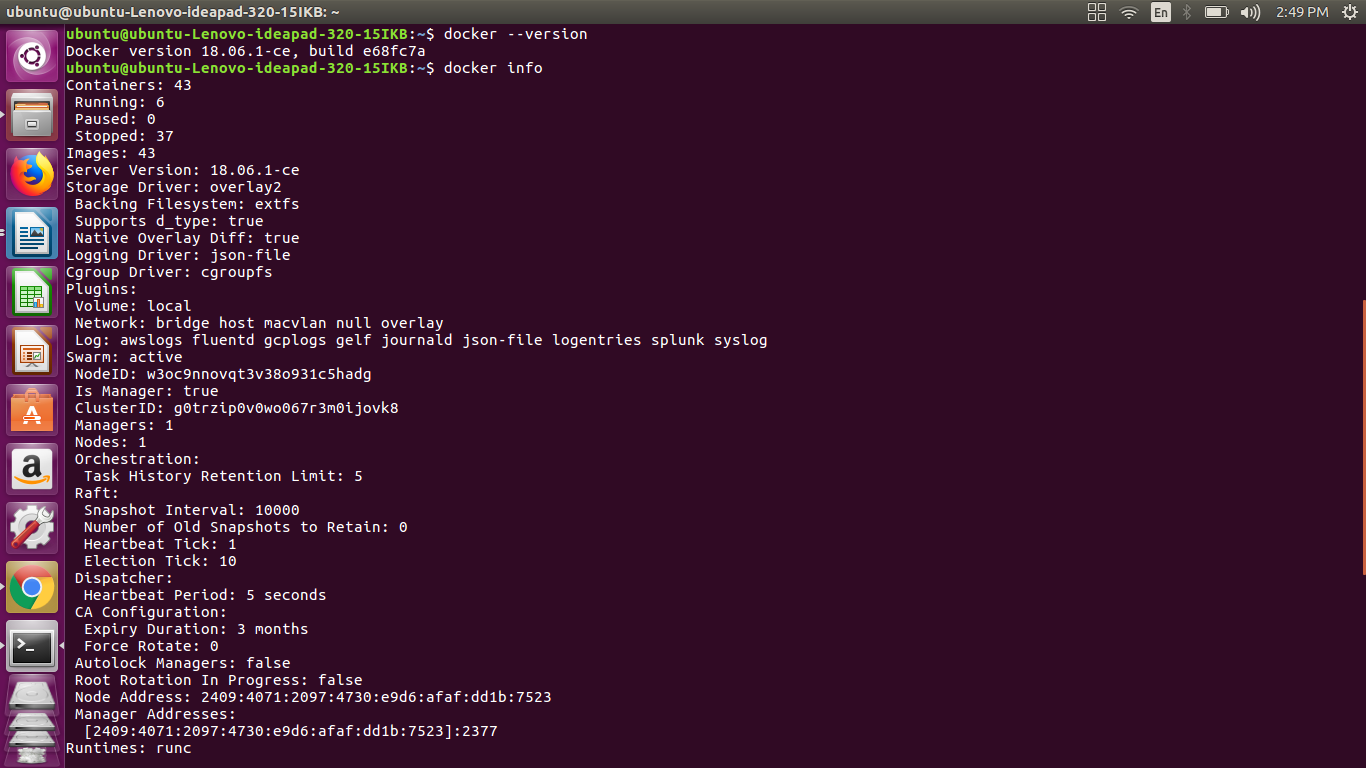
* We run *docker –version* and ensure that we have a supported version of Docker:
* ****Run *docker info*  to view even more details about your docker installation

Figure 2.1.1: Checking for docker version.

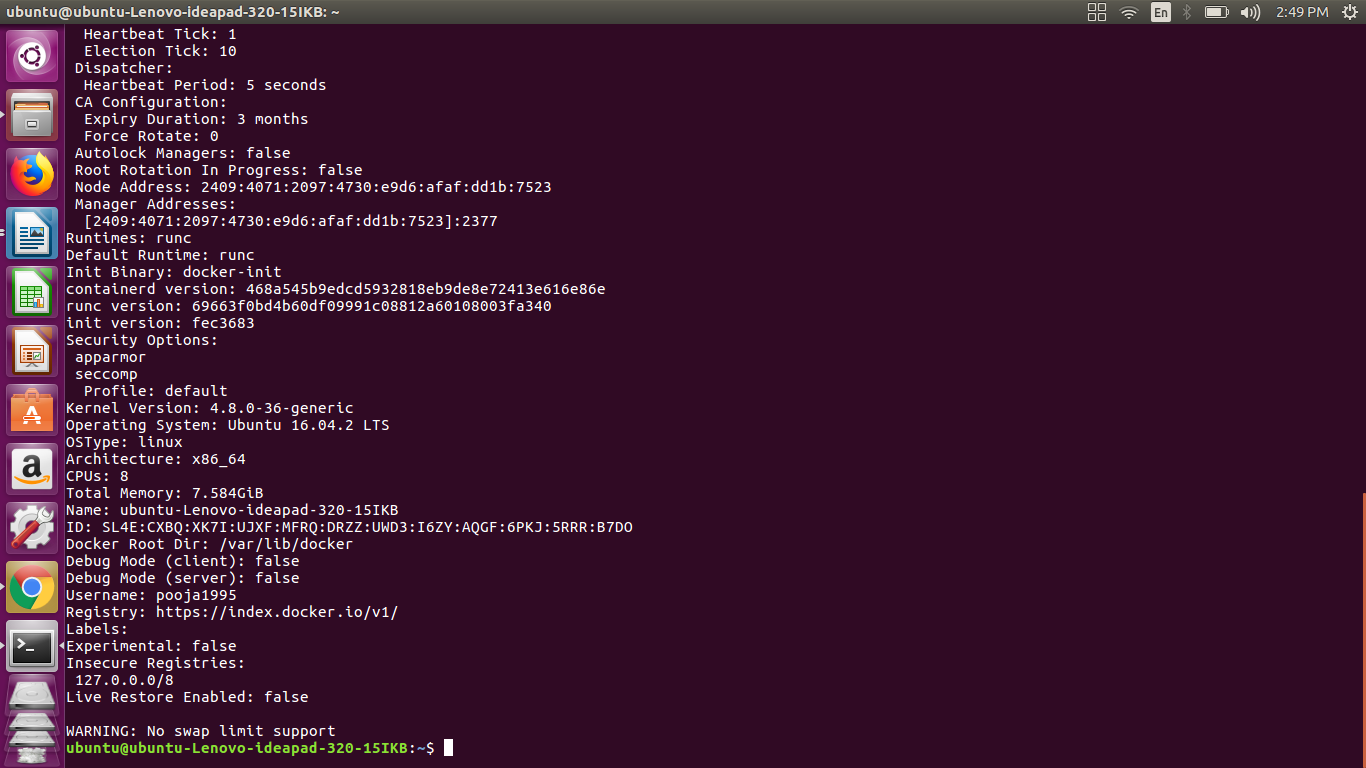


Figure 2.1.2: Dockerinfo

2.2. Test Docker installation

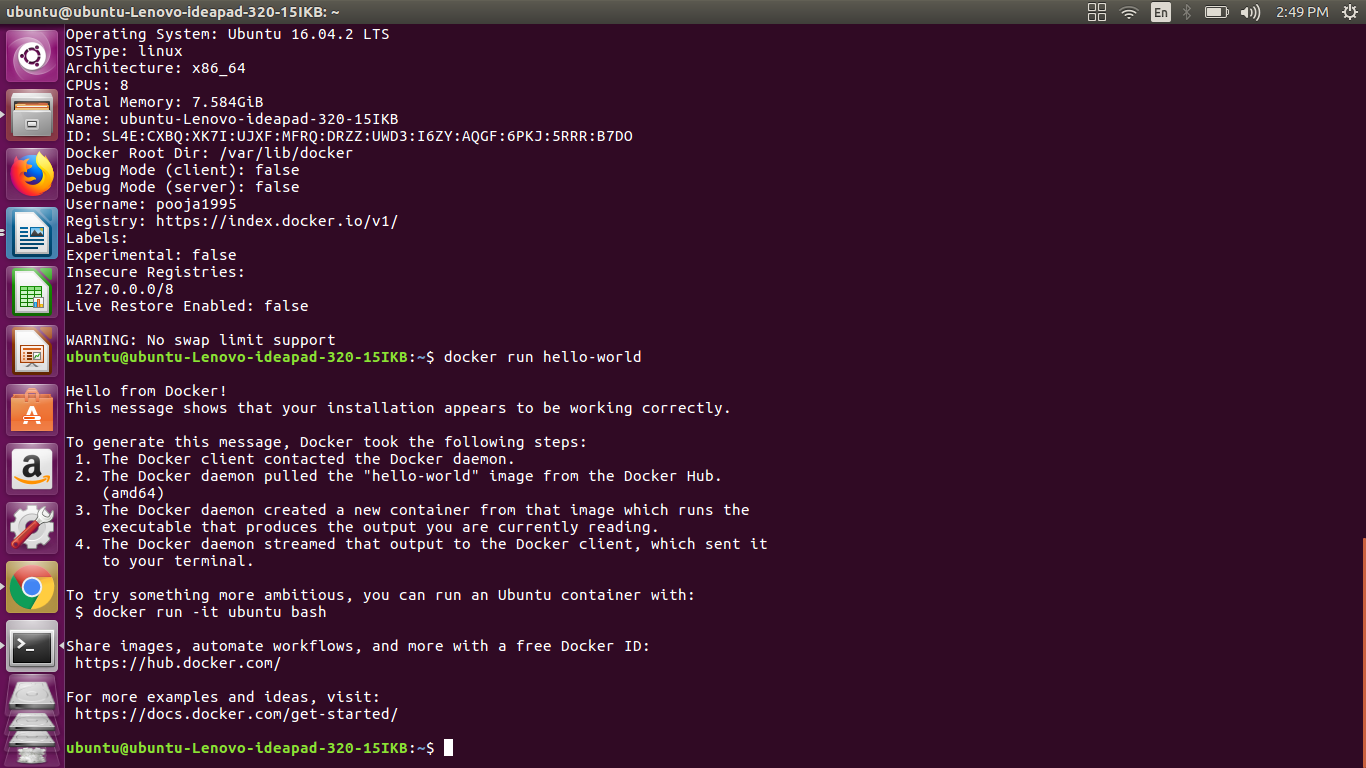
Test that your installation works by running the simple Docker image, [hello-world](https://hub.docker.com/_/hello-world/)

Figure 2.2.1: checking for successful installation

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

docker image ls

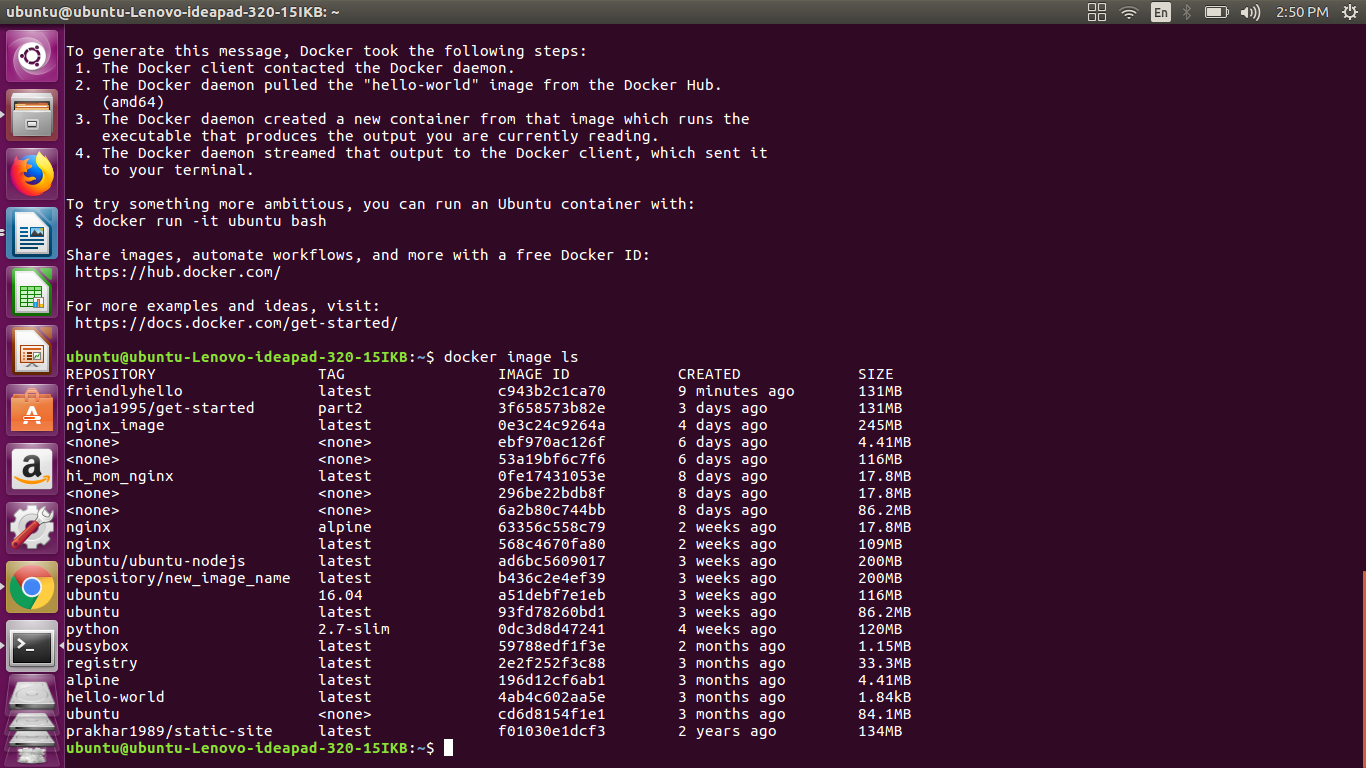


Figure 2.2.2: listing the downloaded images.

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

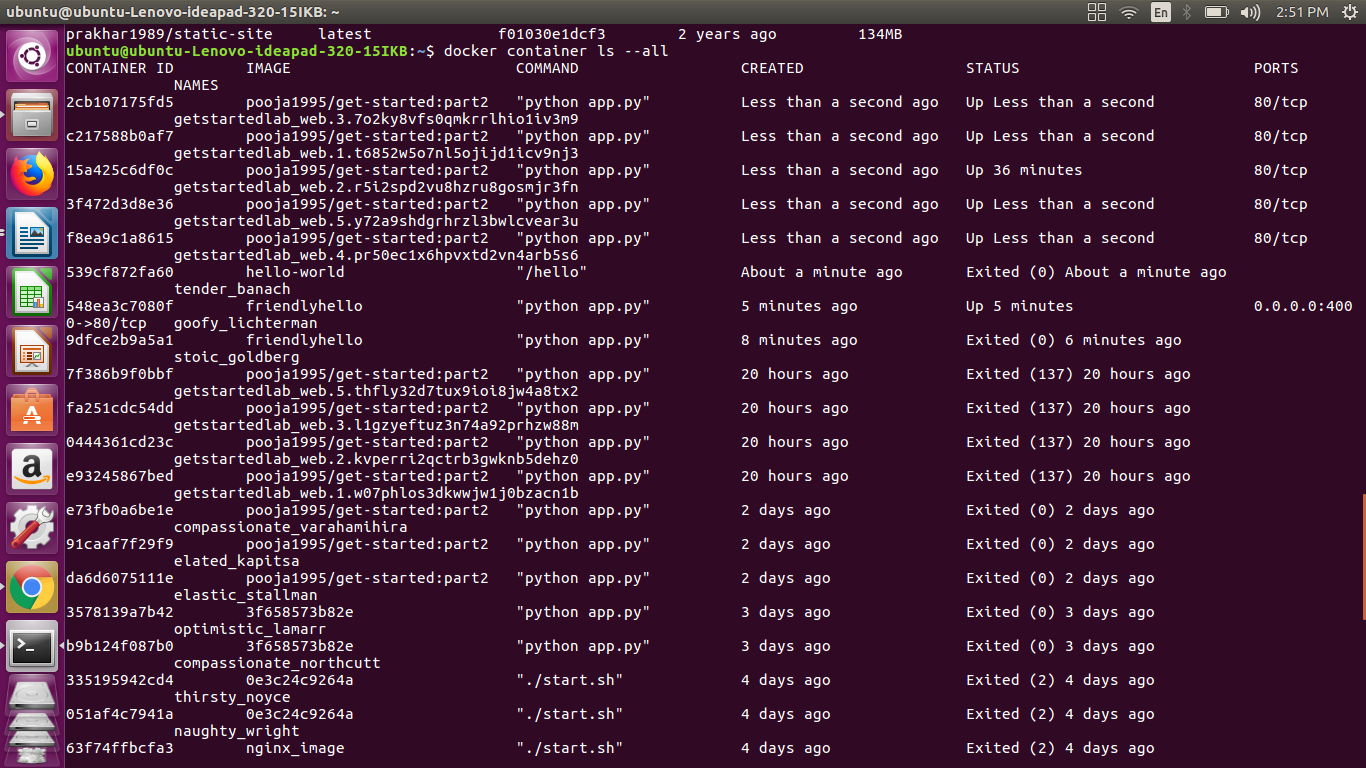
docker container ls –all

Figure 2.2.3: listing containers

2.3. 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.

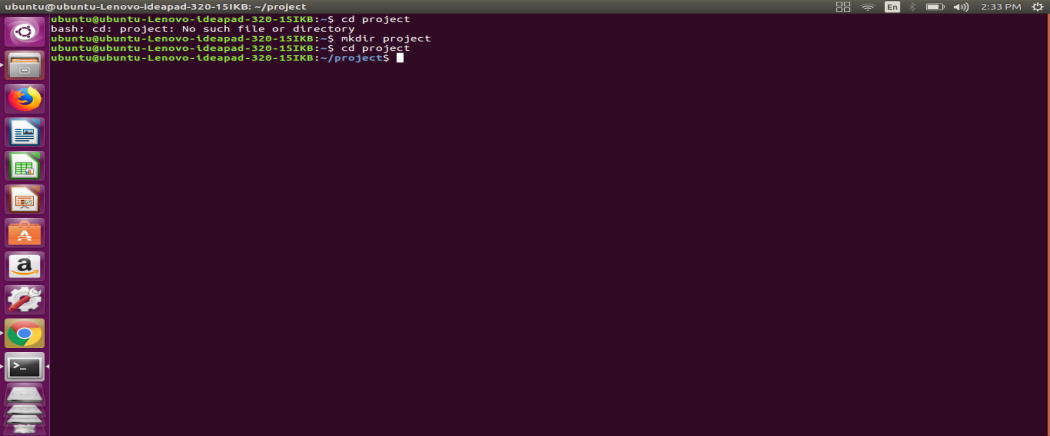


Figure 2.3.1: creating a directory

* # 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:

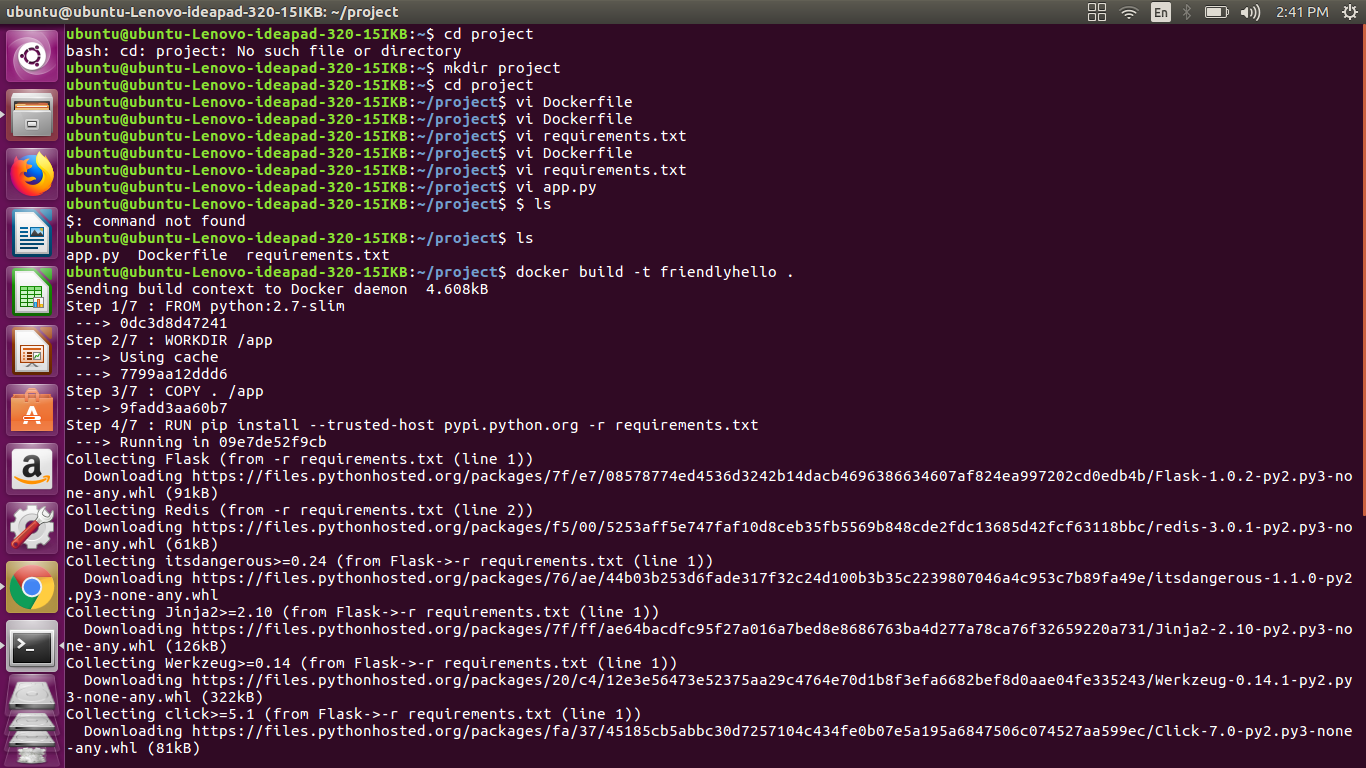
$ ls

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

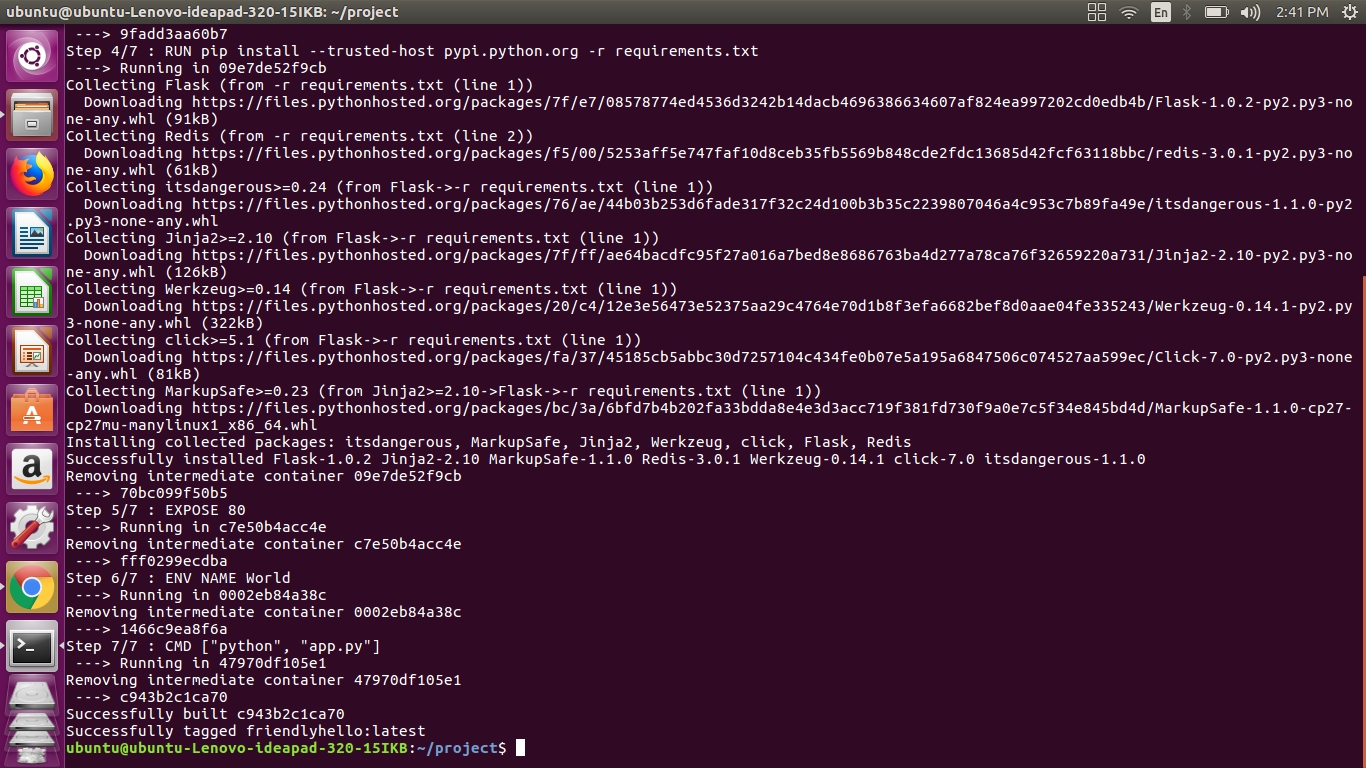


Figure 3.3.3: creating docker image

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

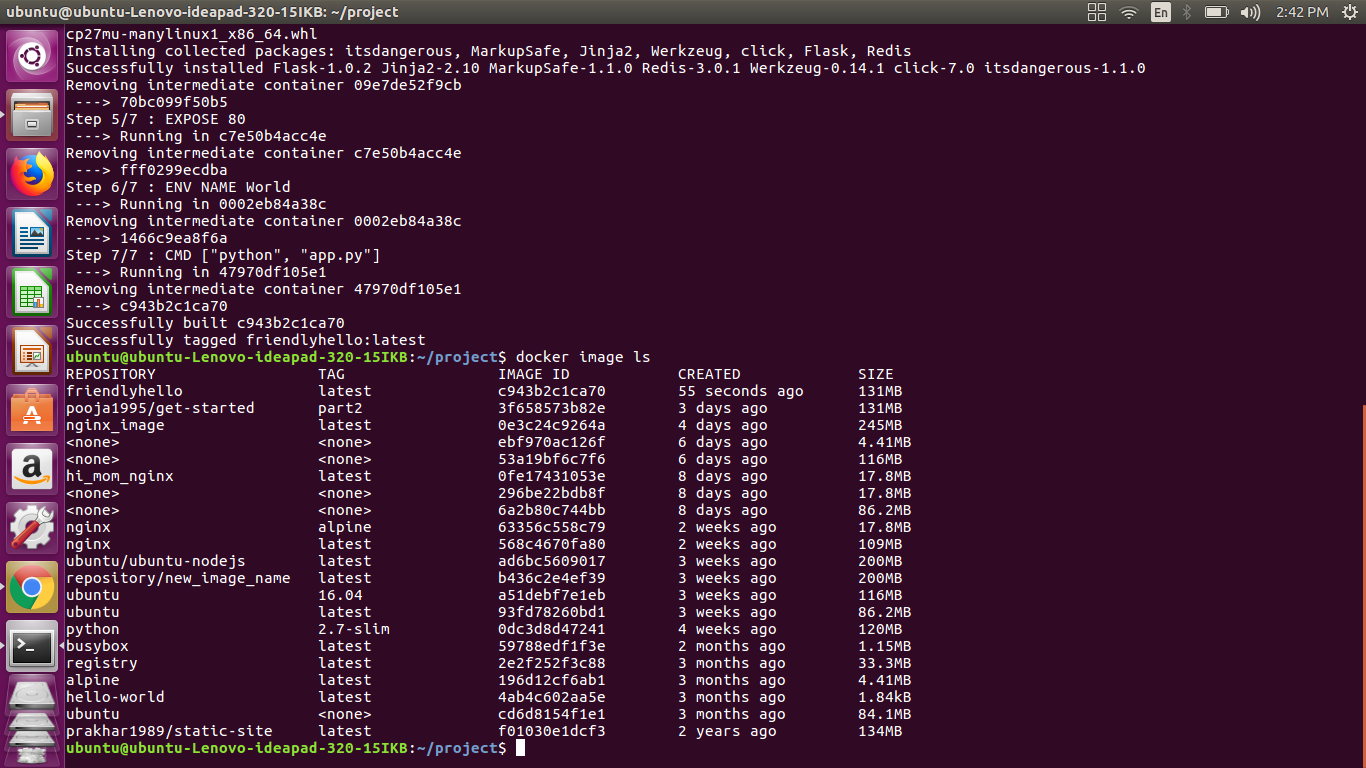


Figure 3.3.4: list of docker images

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

docker run –p 4000:80 friendlyhello

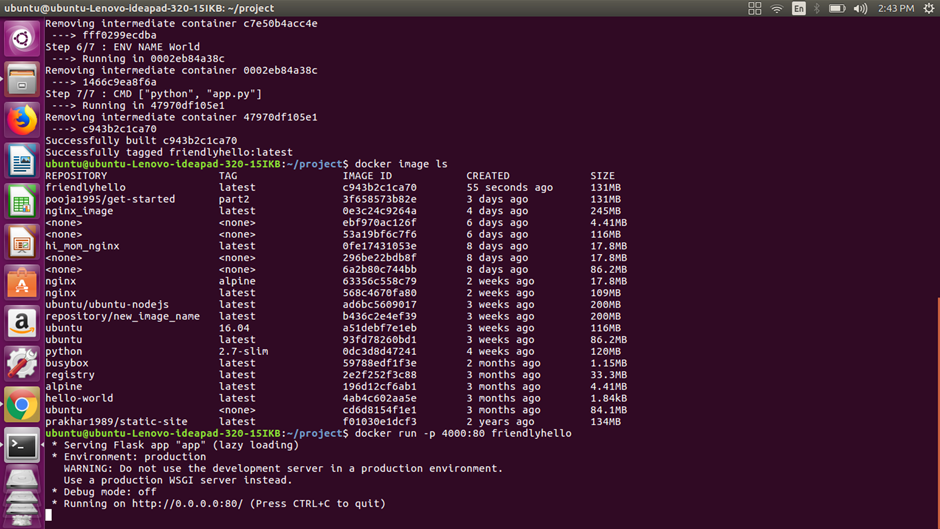


Figure 3.3.5: running the image

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.

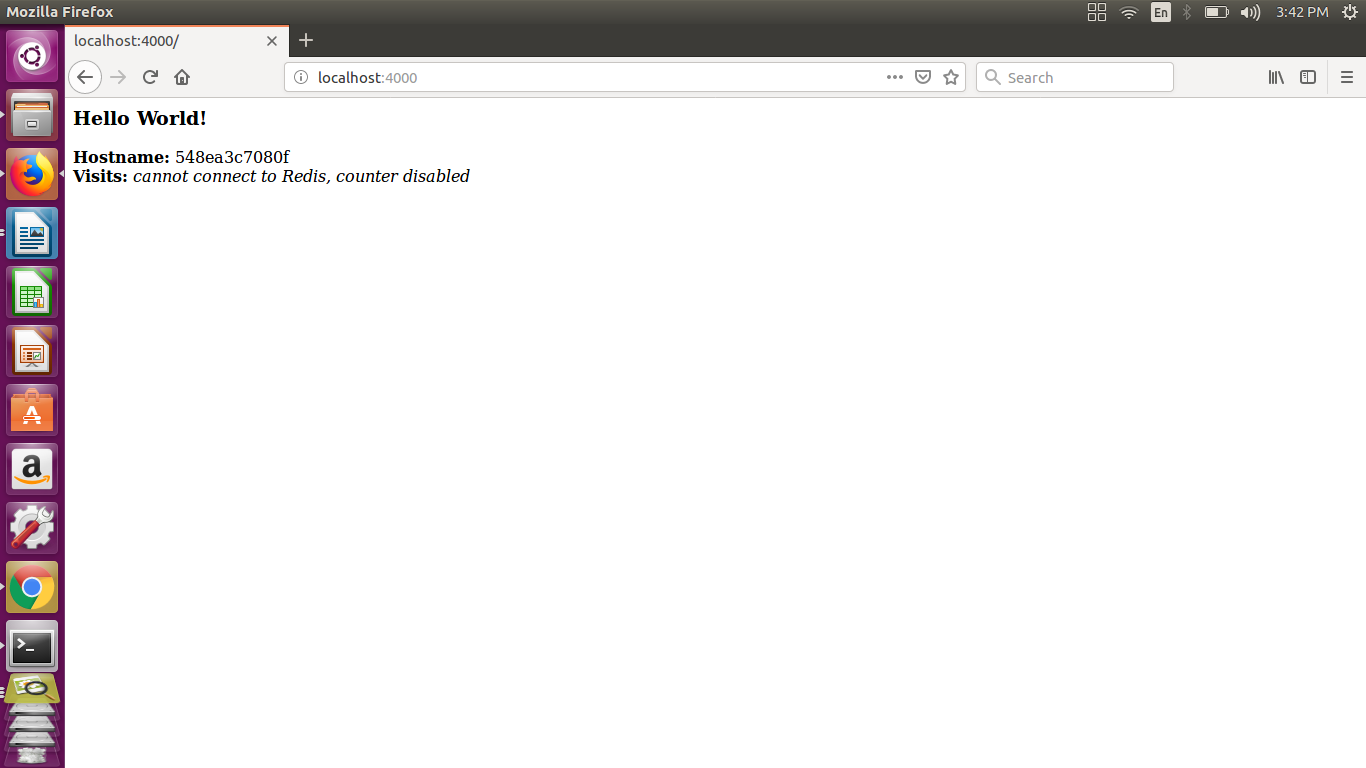


Figure 3.3.6: Display content served up on a web page.

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

**2.4.1. 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

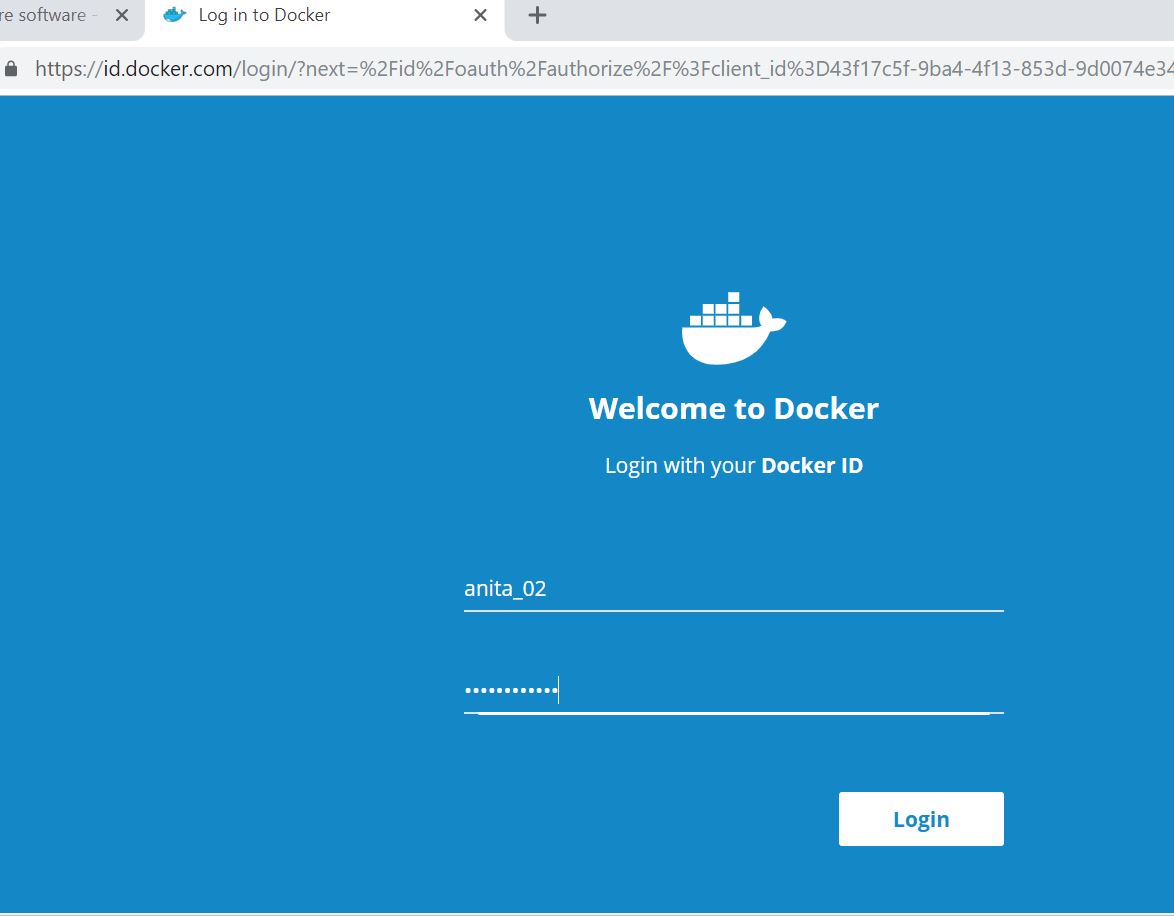


Figure 2.4.1.1: docker registry login page

2.4.2. 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.

2.4.3. 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.

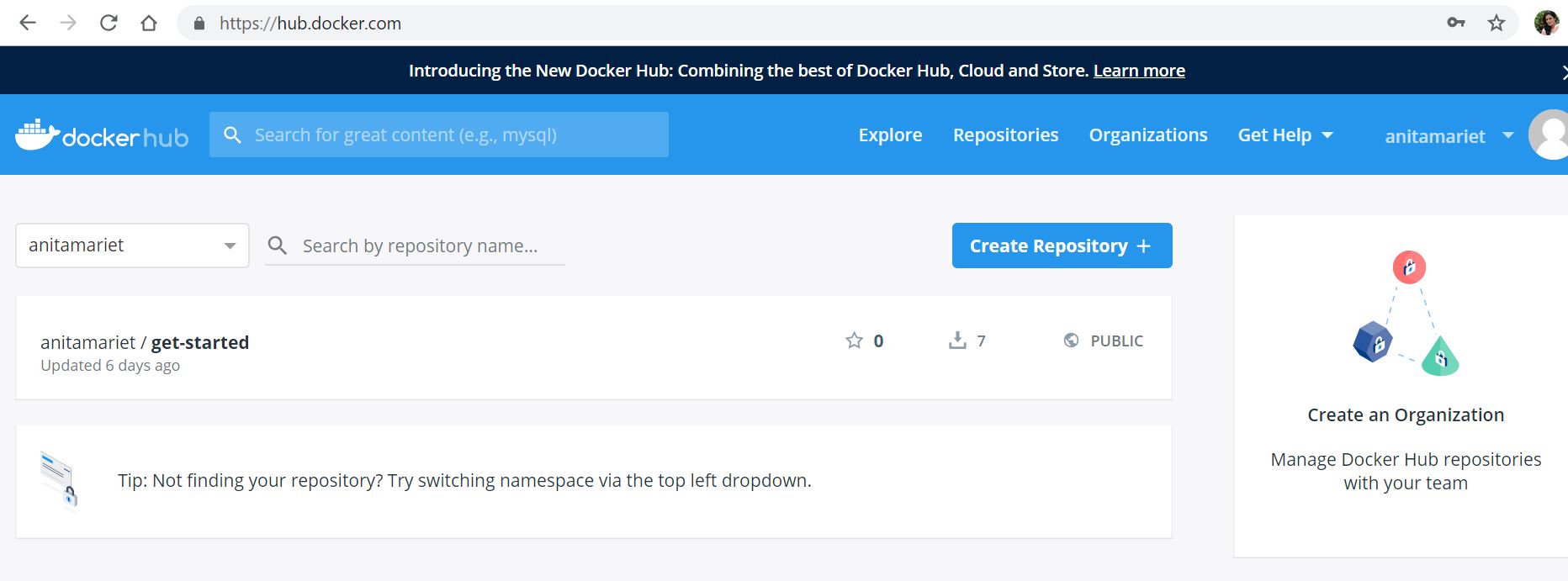


Figure 2.4.3: docker hub repository

**CHAPTER 3**

**WORK CARRIED OUT IN PROJECT PART- 2**

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.

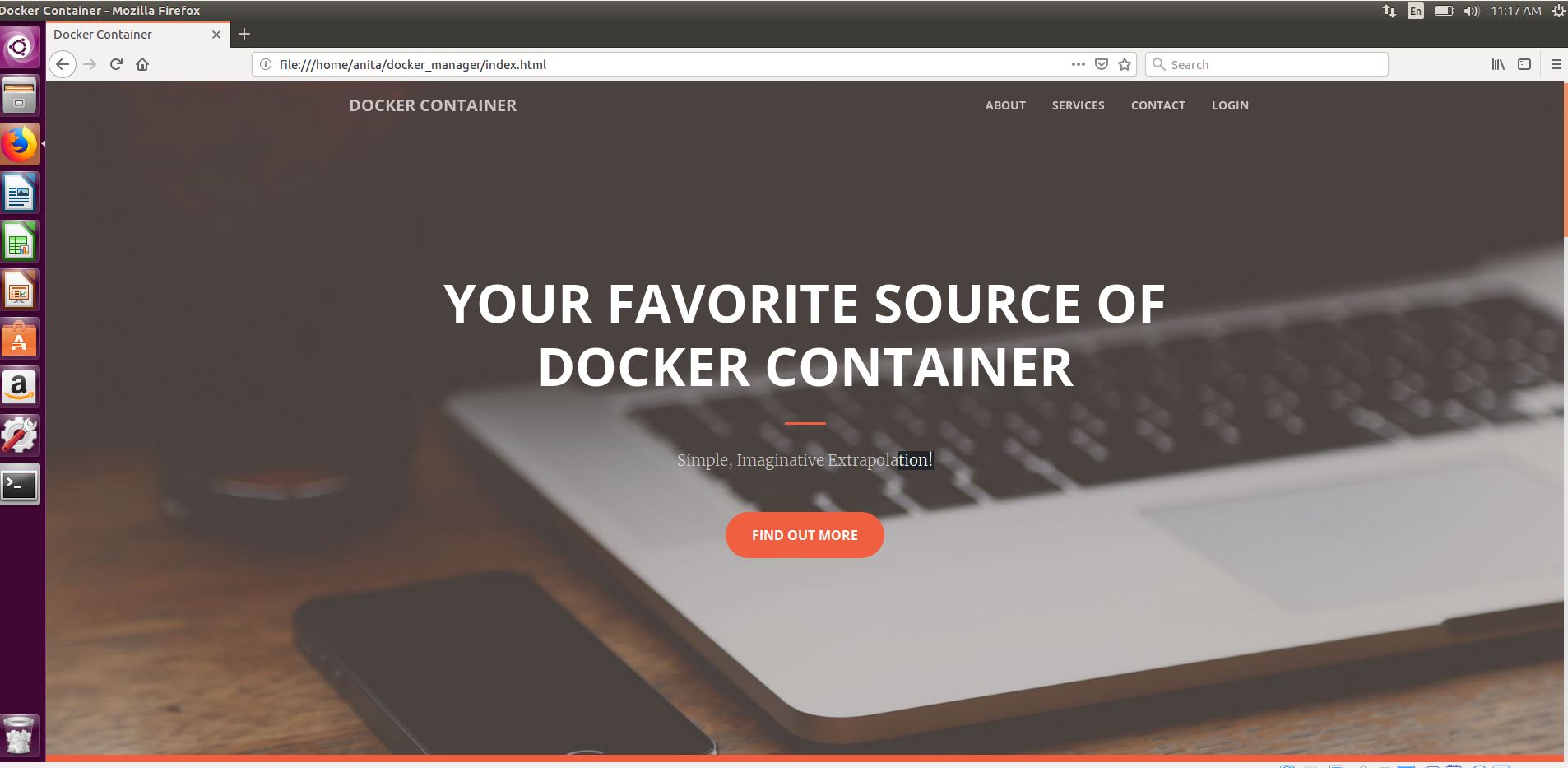


Fig 3.1.1: Initial page of our website.

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.

This will contain some information about the company providing containers

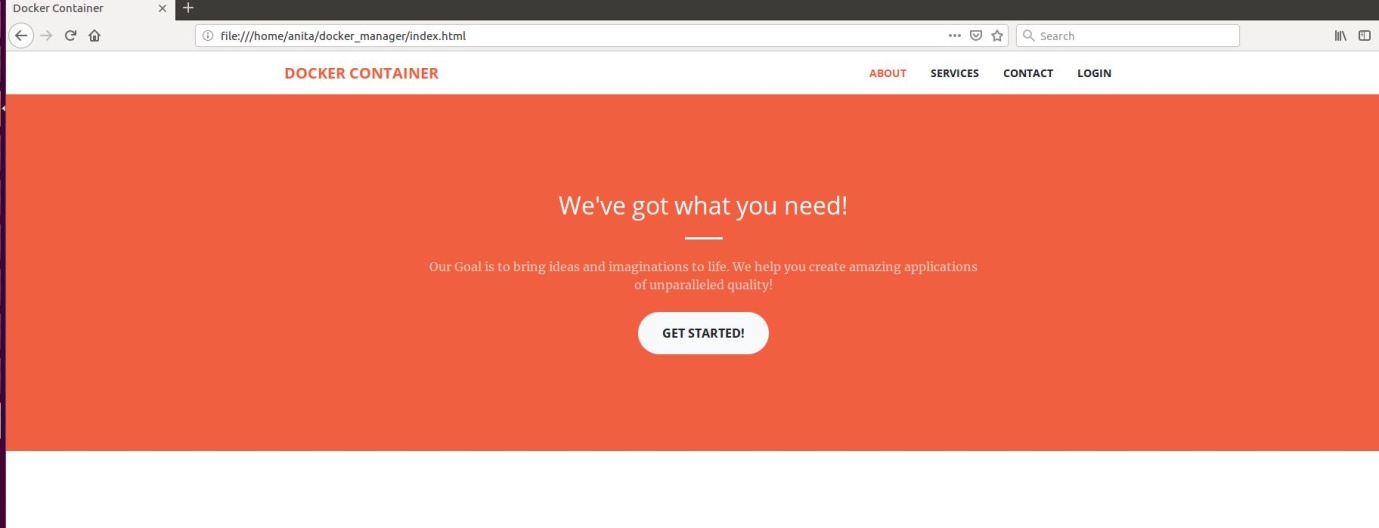


Fig 3.2.1 About page of the website

**3.3 SERVICE**

This contains the services provided by the company. The main service we provide is the distribution of the containers.

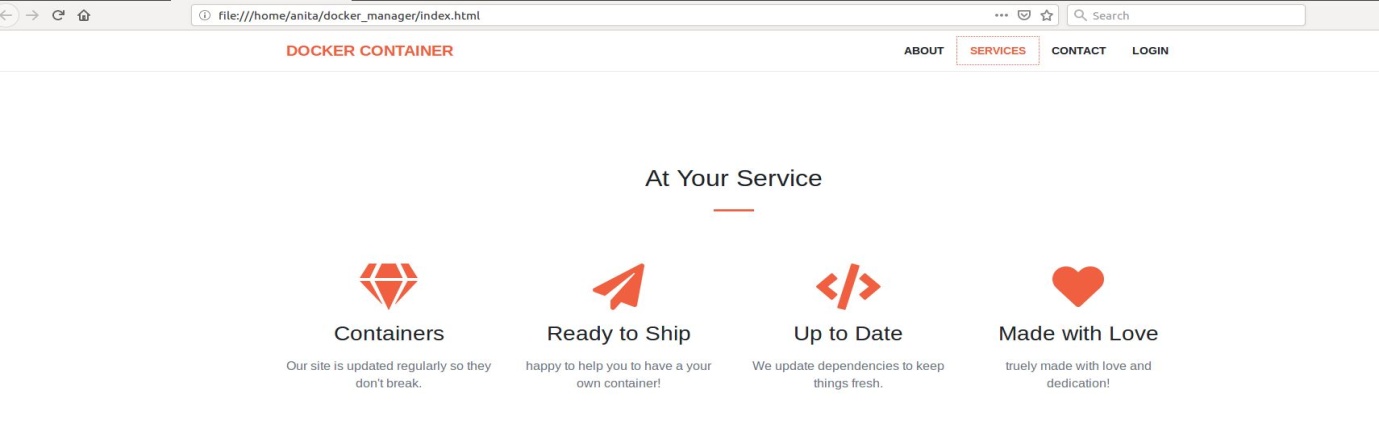


Fig 3.3.1 Service section of the website

**3.4 Contact**

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

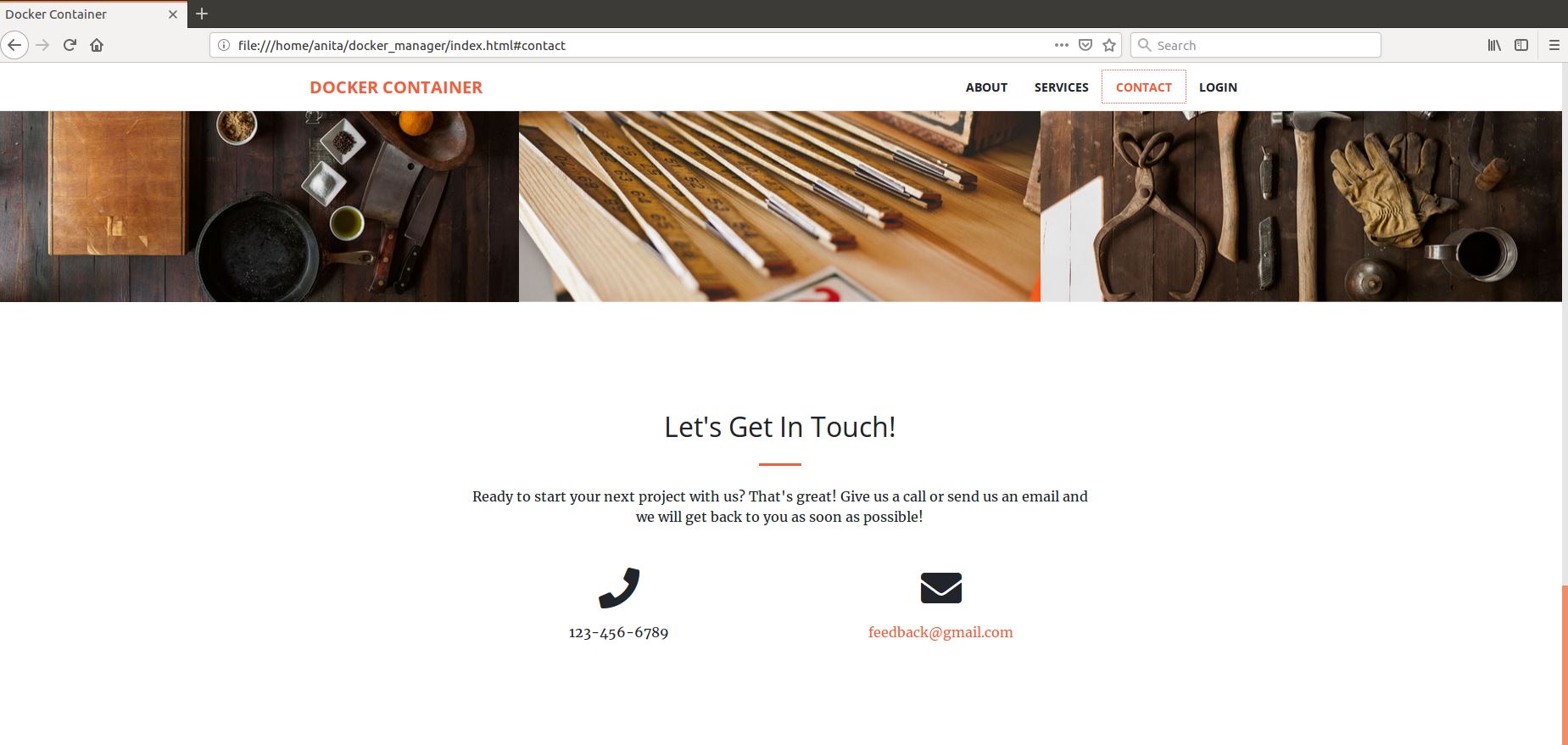


Fig 3.4.1 Contact section of the website

**3.5 LOGIN**

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

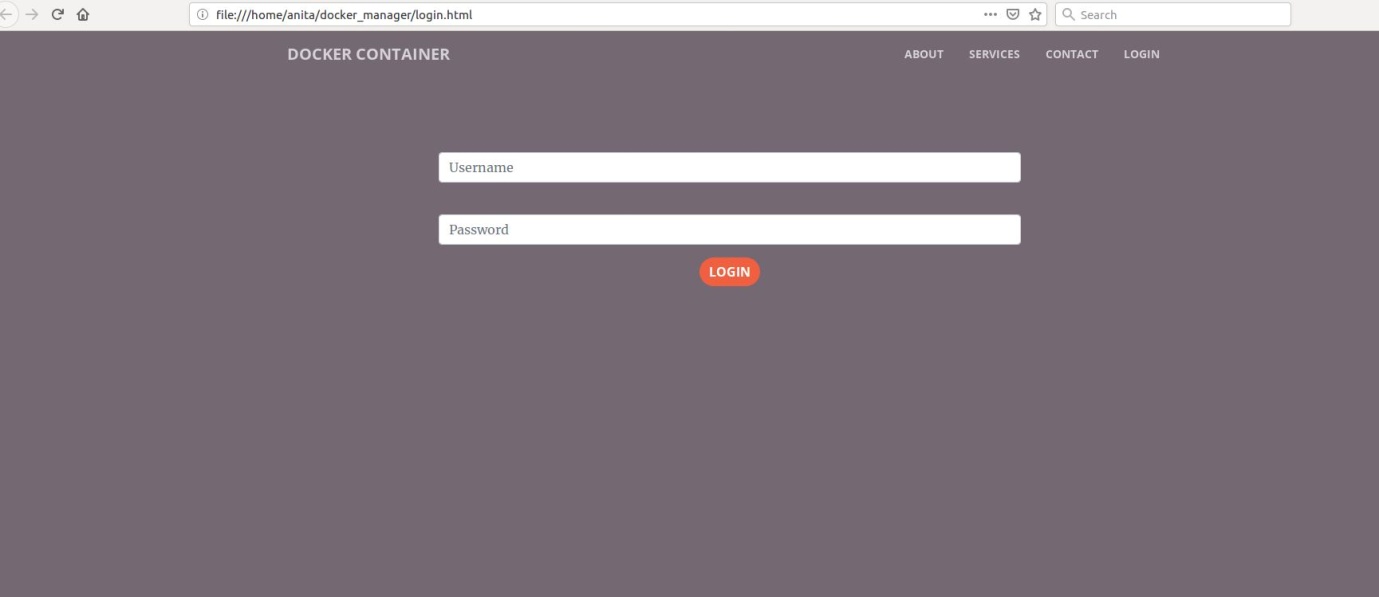


Fig 3.5.1 login page of the website

**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.

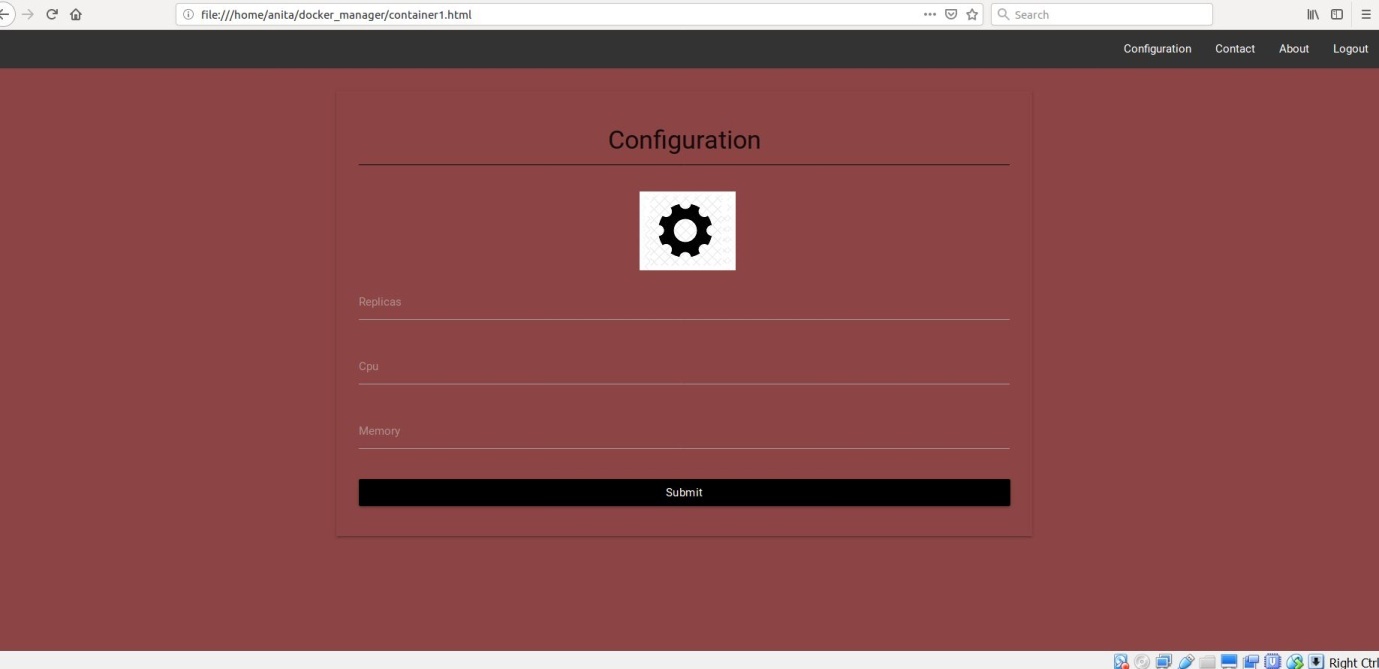


Fig 3.6.1 Configuration page of the website.

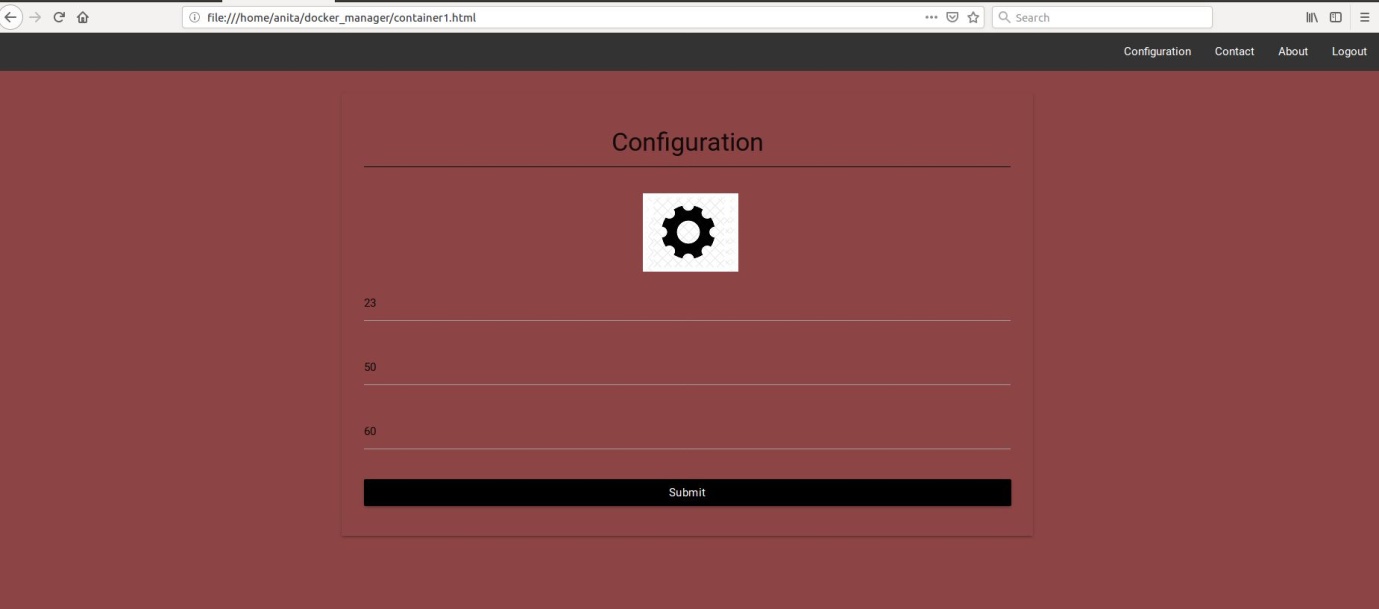


Fig 3.6.2 inserted data of the configuration fields

**3.7 SAVED DATA**

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

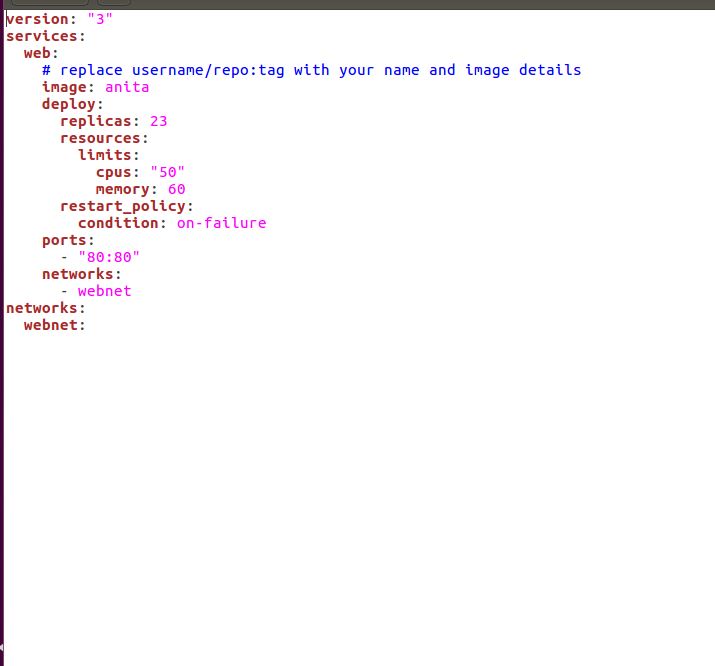


Fig 3.7.1 saved data on the file

**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 |
|  |  |
|  |
|  |  |
|  |  |