Individual Project

Development of a Virtualised Infrastructure Manager for a WMN-based Disaster Network

Submitted by: Nazowa Tanim (1224746)
First examiner: Dr. Professor Ulrich Trick
Date of start: 25.05.2018
Date of submission: 24.08.2018

Statement

I confirm that I have written this project on my own. No other sources were used except those referenced. Content which is taken literally or analogously from published or unpublished sources is identified as such. The drawings or figures of this work have been created by myself or are provided with an appropriate reference. This work has not been submitted in the same or similar form or to any other examination board.

_____

Date, signature of the student

Content

1 Introduction

The main purpose of this Student project is to implement a Virtual Infrastructure Manager (VIM) for Wireless Meshed Network (WMN) based disaster Network. In this project the functionalities of Docker have been developed which can be implemented on disaster network in future. In the theoretical part of this student project will cover the NFV, VIM, Docker Container Technology, REST API and some more topics. The implementation part will cover the REST API based implementation of Docker functionalities in Python. This paper will cover the following chapters:

1. Introduction
2. Theoretical Background
3. Requirement Analysis
4. Realization
5. Summary and Future Scope
6. References

2 Theoretical Background

This chapter will explain all the theoretical background of all technologies which have been used in this student project. This discussion will help to understand the implementation of this research work. Therefore, it is necessary to understand the theories, architectures, protocols and software which have been used in this project.

This chapter contains following major sections:

- Network Function Virtualization (NFV)
- Virtual Infrastructure Manager (VIM)
- Implementation in Python
- Docker Container technology
- WMN based Disaster Network

2.1 Network Function Virtualisation

The concept of Network Function Virtualisation (NFV) originated from the requirement of telecommunication service providers worldwide, to accelerate deployment of new network services and to support their revenue and future growth objectives. Present day telecommunication networks are over populated with a large and increasing variety of proprietary hardware appliances. Some examples of typical telecom equipment are routers, switches, base stations, firewalls, voice gateways, and IMS and Mobile Packet Core. These types of equipment are typically monolithic in design; that is, they consist of hardware, software, and associated management systems. Network Function Virtualization (NFV) is a new way to design, deploy, and manage networking services by decoupling the physical network equipment from the functions that run on them, which replaces hardware centric, dedicated network devices with software running on general-purpose CPUs or virtual machines, operating on standard servers. [1][2]

2.1.1 Origins of Network Function Virtualisation (NFV)

Capitalizing on the technology advances that emerged from the software-defined networking (SDN) movement, and the move to virtualisation and the cloud in data centers, the NFV concept emerged as a call to action by a global group of telecom industry operators in 2012. The Industry Specification Group on NFV (NFV ISG) forum was formed under the European Telecommunications Standards Institute (ETSI) to address this issue. The group published the first formal definition of NFV — the “NFV Architectural Framework” — in 2013. The NFV industry movement began with the goal of saving capital equipment costs by transferring network functions from expensive proprietary platforms to commodity servers. [1]

2.1.2 ETSI NFV Framework

The European Telecommunications Standards Institute (ETSI) is a recognized regional standards body made up of CSPs working together to define frameworks for solutions used in their networks. The ETSI NFV Industry Specification Group (ISG) has defined a high-level functional architectural framework for NFV [1] (fig 2.1).

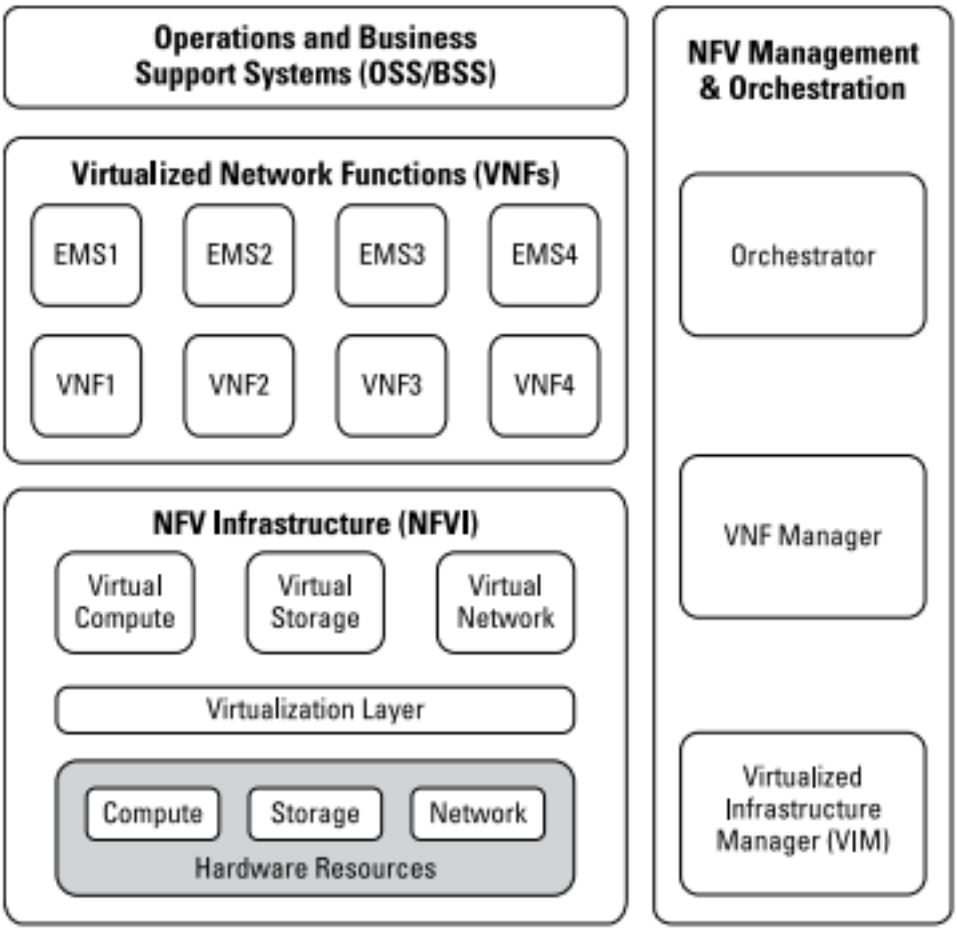

Fig 2.1 ETSI NFV Framework [1]

This framework consists of three major parts [1]:

1. **Network Functions Virtualization Infrastructure (NFVI):** A subsystem that consists of all the hardware (servers, storage, and networking) and software components on which Virtual Network Functions (VNFs) are deployed. This includes the compute, storage, and networking resources, and the associated virtualization layer (hypervisor).
2. **Management and Orchestration (MANO):** A subsystem that includes the Network Functions Virtualization Orchestrator (NFVO), the virtualized infrastructure manager (VIM), and the Virtual Network Functions Manager (VNFM).
3. **Virtual Network Functions (VNFs):** VNFs are the software implementation of network functions that are instantiated as one or more virtual machines (VMs) on the NFVI.

2.2 Virtual Infrastructure Manager (VIM)

The Virtualised Infrastructure Manager (VIM) is responsible for controlling and managing the NFVI compute, storage and network resources, usually within one operator's Infrastructure Domain (e.g. all resources within an NFVI-PoP, resources across multiple NFVI-POPs, or a subset of resources within an NFVI-PoP). There are so many fucntions performed by VIM and it can be discussed as following [3]:

- Orchestrating the allocation/upgrade/release/reclamation of NFVI resources (including the optimization of such resources usage), and managing the association of the virtualised resources to the physical compute, storage, networking resources. Therefore, the VIM keeps an inventory of the allocation of virtual resources to physical resources, e.g. to a server pool.
- Supporting the management of VNF Forwarding Graphs (create, query, update, delete), e.g. by creating and maintaining virtual Links, virtual networks, sub-nets, and ports, as well as the management of security group policies to ensure network/traffic access control.
- Managing in a repository inventory related information of NFVI hardware resources (compute, storage, networking) and software resources (e.g. hypervisors), and discovery of the capabilities and features (e.g. related to usage optimization) of such resources.
- Management of the virtualised resource capacity (e.g. density of virtualised resources to physical resources), and forwarding of information related to NFVI resources capacity and usage reporting.
- Management of software images (add, delete, update, query, copy) as requested by other NFV-MANO functional blocks (e.g. NFVO). While not explicitly shown in the NFV-MANO architectural framework, the VIM maintains repositories for software images, in order to streamline the allocation of virtualised computing resources. A validation step, performed by VIM, is required for software images before storing the image (e.g. VNF package on-boarding and update). Image validation operations during run-time, e.g. during instantiation or scaling, are outside the scope of the current version of the present document.
- Collection of performance and fault information (e.g. via notifications) of hardware resources (compute, storage, and networking) software resources (e.g. hypervisors), and virtualised resources (e.g. VMs); and forwarding of performance measurement results and faults/events information relative to virtualised resources.
- • Management of catalogues of virtualised resources that can be consumed from the NFVI. The elements in the catalogue may be in the form of virtualised resource configurations (virtual CPU configurations, types of network connectivity (e.g. L2, L3), etc.), and/or templates (e.g. a virtual machine with 2 virtual CPUs and 2 GB of virtual memory).

2.3 Implementation in Python

Python has been used for the implementation of this project. Flask framework of Python has been used to develop RESTful HTTP client-server.

2.3.1 REST API

In this project RESTful API has been used to control docker aspects. REST is a web standards-based architecture and uses HTTP Protocol for data communication. It revolves around resources where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. Some methods are as following [4]:

- GET − Provides a read only access to a resource.
- PUT − Used to create a new resource.
- DELETE − Used to remove a resource.
- POST − Used to update an existing resource or create a new resource

2.3.2 Python Flask Framework

Flask is a micro web framework written in Python. It is classified as a microframework because it does not require tools or libraries. REST API can be implemented easily with less codes and but doesn't limit in any feature. As an example following is a 'Hello-World' web application that can be regarded as a simple HTTP client-server app with Flask [5]:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == '__main__':
    app.run()
```

this program will return 'Hello, World!' with GET method and URI http://localhost:5000/ sent from the client side.

2.4 Docker

In this project one task was to implement Docker functionalities. Docker is an open-source engine that automates the deployment of applications into containers. Docker adds an application deployment engine on top of a virtualised container execution environment. It is designed to provide a lightweight and fast environment in which code can be run as well as an efficient workflow to get that code from laptop to our test environment and into production. An application can be dockerized in minutes, then container can be created running the applications [8]

Docker has four components

- The Docker client and server
- Docker Images
- Registries
- Docker Containers

Docker can be used in various aspects like following [8]:

- Helps to make local development and build workflow faster, more efficient, and more lightweight. Local developers can build, run, and share Docker containers. Containers can be built in development and promoted to testing environments and, in turn, to production.
- Running stand-alone services and applications consistently across multiple environments, a concept especially useful in service-oriented architectures and deployments that rely heavily on micro-services.
- Building and testing complex applications and architectures on a local host prior to deployment into a production environment.
- Building a multi-user Platform-as-a-Service (PAAS) infrastructure.

Providing lightweight stand-alone sandbox environments for developing, testing, and teaching technologies, such as the Unix shell or a programming language.

2.4.1 Container

A container is a packaging format for a unit of software that ships together. A container is a format that encapsulates a set of software and its dependencies, the minimal set of runtime resources the software needs to do its function. The key difference between VMs and containers is that each VM has its own full-sized OS, while containers typically have a more minimal OS. [4]

2.4.2 Docker SDK(Docker-py)

A Python library for the Docker Engine API. It provides most of the functionalities of docker within Python apps like run containers, manage containers, manage Swarms, etc. [9].

As an example image can be pulled from Docker hub with Docker-py built in syntax as following:

```
client.images.pull(imagename)
```

2.5 WMN based Disaster Network

In disaster situations, an operative communication infrastructure is essential, in order to rescue victims and organize, coordinate, and support rescue teams. Existing communication infrastructures are often affected in case of disaster, so that the infrastructure is damaged in whole or part. Consequently, the necessity occurs to develop a proper communication system. This system should be enabled to be established rapidly, easily, and cost-effectively in order to share information inside and with the disaster area constantly and robust. Moreover, a desirable system provides not only voice communication but also multimedia communication to support the rescue teams and helpers sufficiently. Wireless Mesh Network (WMN) is considered as one of the most suitable solutions because it can easily configure a network without any wired infrastructure. WMNs are comprised of two types of nodes: mesh routers and mesh clients. Other than conventional wireless router, the capability for gateway/bridge functions differs in additional routing functions to support mesh network. The architecture of WMNs can be classified into three types (Fig 2.2) [6]:

1. **Backbone** – The mesh routers/gateways form an infrastructure for clients and can be connected to various networks, e.g. Internet. Furthermore, the mesh routers have minimal mobility.
2. **Client** – Client meshing provides peer-to-peer networks among client devices. In this type of architecture, client nodes constitute the actual network to perform end-user applications to customers.
3. **Hybrid –** This architecture is the combination of backbone and client meshing, as shown in Fig. 2.2 Mesh clients can access the network through mesh routers and directly connect to other mesh clients. The backbone provides connectivity to other networks.

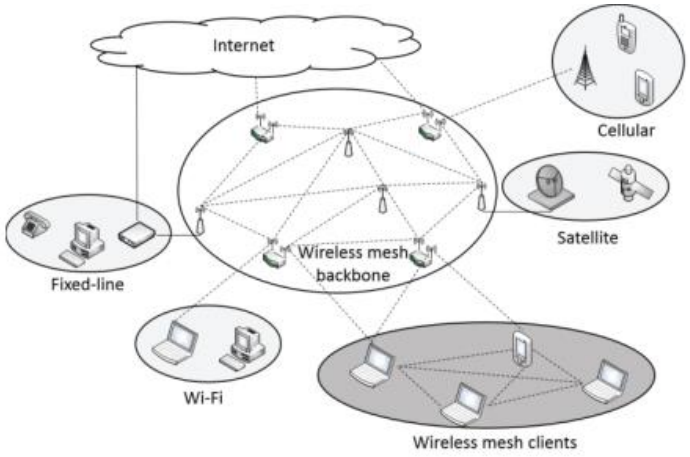

Fig 2.2: Hybrid WMN architecture[6]

3. Requirements Analysis

The goal of this student project to develop the Virtualized Infrastructure Manager (VIM) for Wireless mesh Network(WMN) based Disaster Network.

3.1 General Objectives

In this project an application has to be developed which will provide the docker functionalities like create and manage containers and these docker aspects can be controlled by REST API. The application shall therefore consist of a HTTP client and a server. With the client component REST methods can be sent to the server to retrieve the functionalities. The application shall support to develop the LXC/LXD functionalities and later deploy in WMN based disaster network.

3.2 Clarifying the Requirements

The developed application shall allow user to control docker functionalities through REST API. The application consists of a HTTP server and client. The user manages the RESTful methods with a RESTful API feature provider (e.g: POSTMAN). This shall be developed based on Python and Linux.

The application shall support the development of LXC/LXD functionalities, creating tunnel points and later deploy in disaster network. The application shall provide the user with feature to control docker aspects on WMN based disaster network. Some requirements that have been given for this project are as following:

- Researching required functionalities of VIM
- Implementation in Python and Linux.
- REST based interface for the orchestrator and HTTP based communication protocol.
- Interfacing hosts functionalities like Docker create, start, stop with Docker py.

Some requirements of docker functionalities can be discussed with following figures:

1. The application shall consist of HTTP client and server. Therefore it shall be possible for client to control the server with RESTful API.

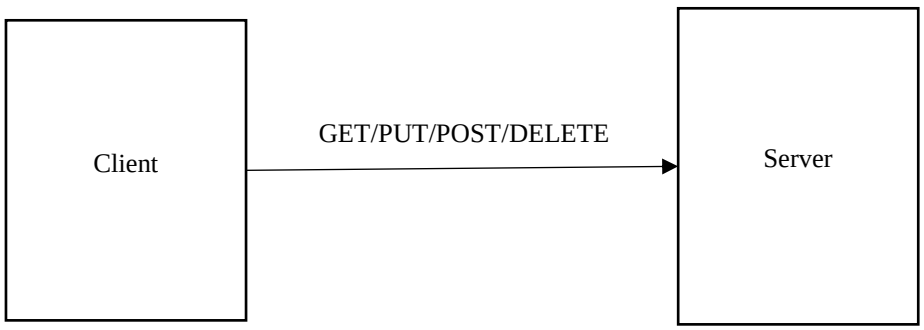

Fig 3.1:Client Server Architecture

2. It shall possible to send input from client side with RESTful methods like POST,PUT as JSON format to the server.

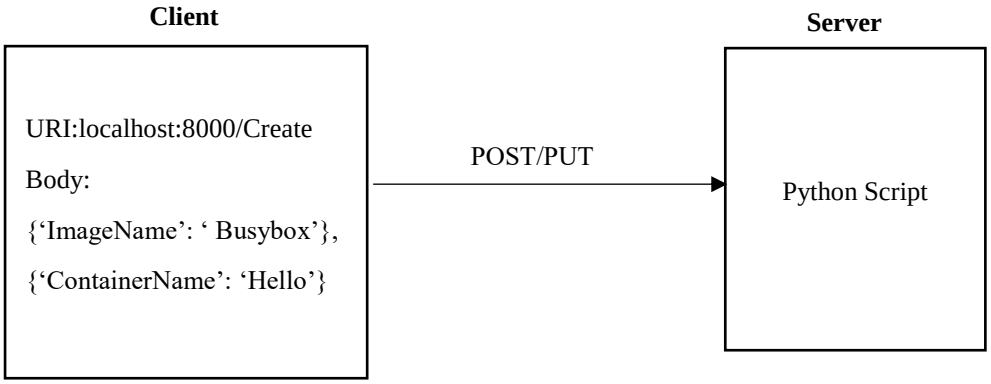

Fig 3.3: Sending Input from Client with JSON format

1. Docker-py shall be used to develop the docker functionalities. Python Subprocess module also provides the opportunity to develop docker functionalities on Linux but docker-py would be a better choice.
2. Response of the server shall be in JSON format. As example if ‘docker info’ functionality need to be developed response shall be in JSON format like following:


```
{
    "ID": "CNKX:KBUK:TM4J:C4RT:HC3E:BLSG:2E7Z:GYX7:EICB:ZS5V:R2GU:7QTE",
    "Containers": 0,
    "ContainersRunning": 0,
    "ContainersPaused": 0,
    "ContainersStopped": 0,
    "Images": 64,
    "Driver": "overlay2",
    "DriverStatus": [
        [
            "Backing Filesystem",
            "extfs"
        ],
        [
            "Supports d_type",
            "true"
        ],
        [
            "Native Overlay Diff",
            "true"
        ]
    ],
    "SystemStatus": null,
    "Plugins": {
        "Volume": [
            "local"
        ],
        "Network": [
```


Fig 3.4 Required output at Client Side

3.3 Time Frames

Milestones:

25.05.2018-8.06.2018	Analysis of the Requirements.
09.05.2018	Submission of Requirement Analysis
10.06.2018-18.06.2018	Literature review and learning theoretical background.
19.06.2018-30.06.2018	VM installation, HTTP server implementation.
07.07.2018 -12.07.2018	Collect the logs to check REST API response, writing report.
13.07.2018-14.07.2018	Installing docker environment on VM.
15.07.2018-25.07.2018	implementation of docker functionalities with docker-py.
26.07.2018-02.08.2018	Implement Docker functionalities with REST API, writing report.
03.08.2018-15.08.2018	Writing report
16.08.2018-23.082018	Editing report as suggested and implementing additional functionalities.
24.08.2018	Submission of Report

3.4 Target State

For the implementation of the prototype, the final state shown in Figure 3.5 is expected. Client can use the methods of RESTful services and communicate to the server. By using this application client can retrieve docker functionalities information from the server and can manage container aspects on server side. As an example Fig 3.5 can be shown:

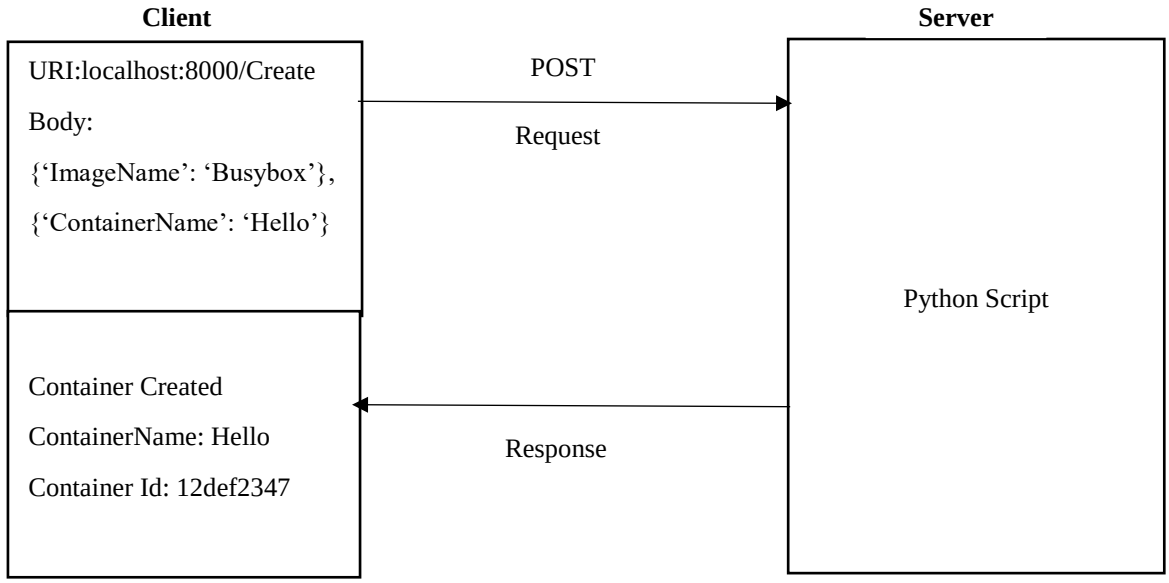

Fig 3.5: An overview of target state

4 Realization

This chapter will describe the implementation and testing of this project work. Following are the steps of the implementation of this student project.

1. Preparation of Environment
2. Setting up Docker-py and Python
3. Setting up Postman App
4. Detail of Python Implementation

4.1 Prepare Environment

For this project virtual box is used and Ubuntu 18.04 has been installed on that. Python 3 is installed for implementation and Docker SDK (Docker-py) has been installed to provide docker functionalities. To act as REST client POSTMAN APP is used. To do implementation task following steps have been followed for the necessary environment.

4.1.1 Installation of Virtual Box

Virtual box is a cross platform virtualization application. It allows us to run multiple operating systems on the same host. In this project implementation and testing tasks have been completed on Linux and Python 3Virtual box can be downloaded from their official website(virtualbox.org) [10].

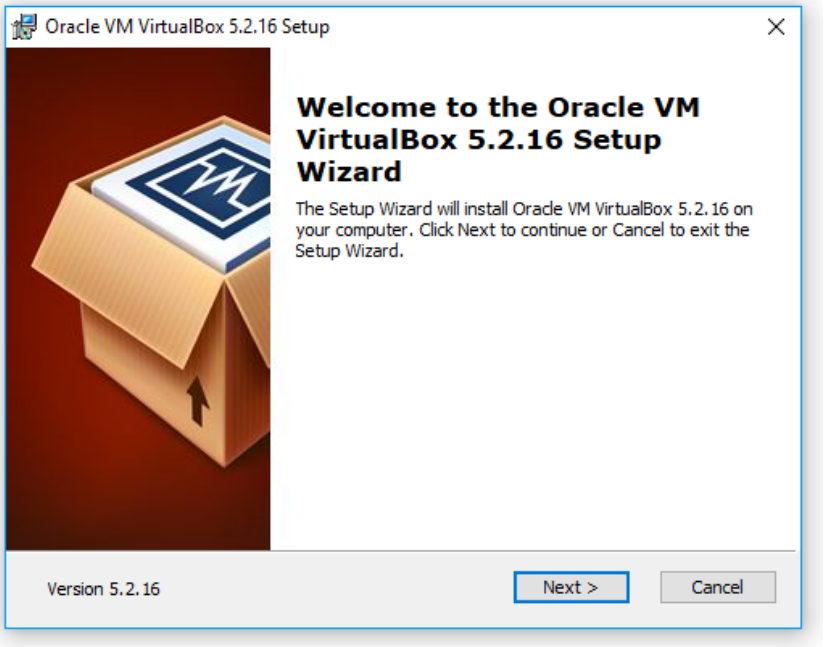

Fig 4.1 Virtual Box Installation Prompt

4.1.2 Create Virtual Machine

Virtual machine has been used to carry out this task. Ubuntu 18.04 has been installed for this task. To create a virtual machine at first the name must be defined and need to select the type of Operating System. We can see figure 4.2

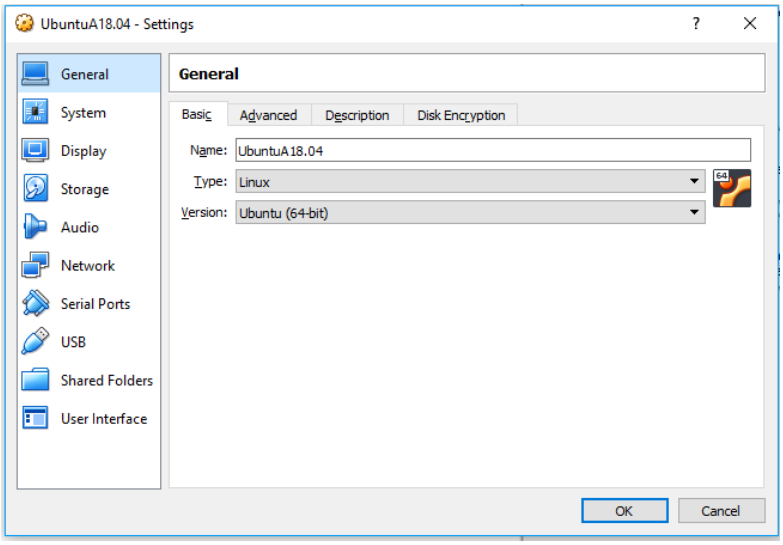

Fig 4.2 Name of Operating System

For this project one VM is installed and credentials of this VM is as following:

VM Name: Ubuntu 18.04

User Name. nazowa

Password: 123

It is important to select the memory size of the VM. Here it is 2GB

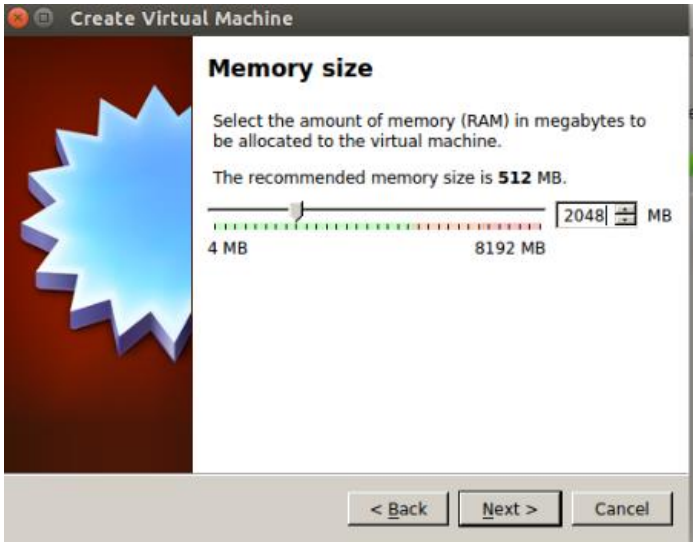

Fig 4.3 Memory Size

Then virtual hard disk should be selected as hard disk type

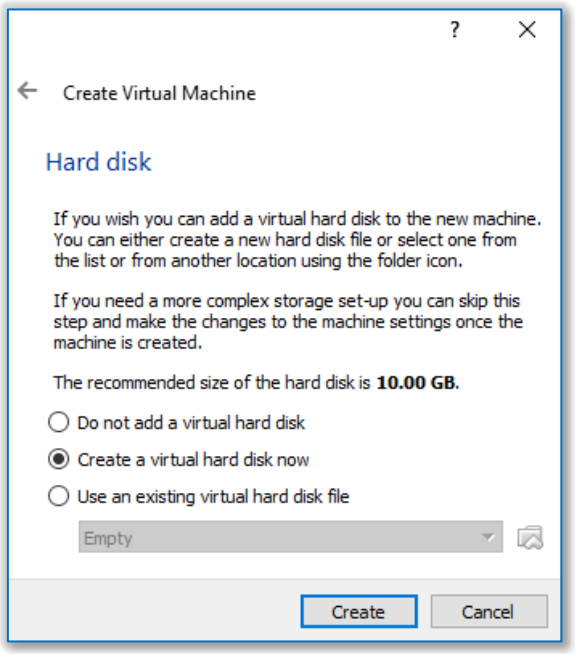

Fig 4.4 Defining Hard Disk option

Then it will ask for type of virtual disk need to be used. For this project VirtualBox Disk Image(VDI) has been used

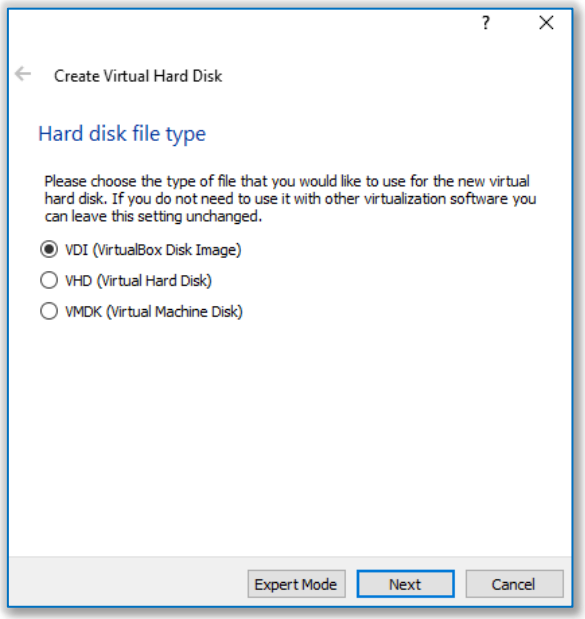

Fig 4.5 Defining Hard Disk File type

Then it will ask how the new virtual disk should expand memory when it is used. For this project ‘Dynamically ‘allocated’ option should be used.

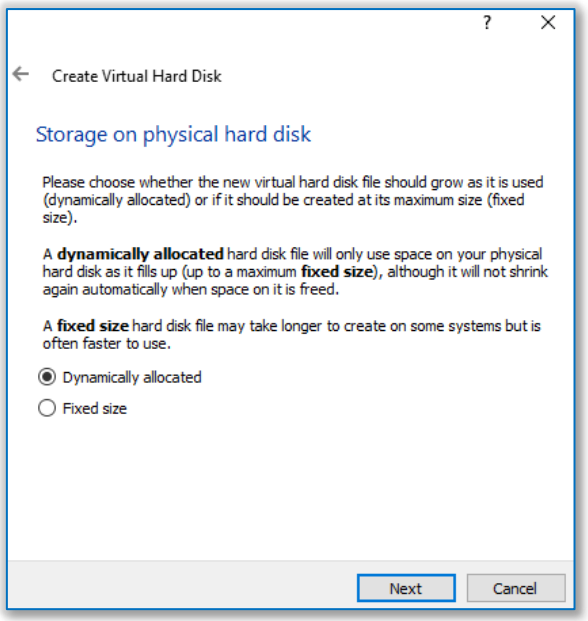

Fig 4.6 Defining physical hard disk type

With above steps virtual hard disk can be created.

4.2 Configuring Ubuntu 18.04 LTS

Ubuntu 18.04 LTS has been used in this project and it can be downloaded from ubuntu official website(ubuntu.com)[11]. Before installing Docker we have carried out some configuration on the ubuntu host.

Ubuntu 18.04 (LTS)

```
$ sudo apt-get update
```

```
$ sudo apt-get upgrade
```

4.2.1 Installing Docker on Ubuntu 16.04 LTS

As we have implemented Docker functionalities we need to install Docker on the Host. We have used Docker v18.03.0-ce. It can be done by following steps (Docker, 2018f).

Prerequisites:

To work with Docker Linux installation should be on 64-bit and Docker version should be 3.10 or higher of the Linux kernel. Kernels older than 3.10 lack some of the features required to run Docker containers and contain known bugs which can cause data loss. We can check the version by following command:

```
$ uname -r
```

```
4.15.0-29-generic
```

Uninstall the old version

Older version must be uninstalled. were called Docker or Docker-Engine.

```
$ sudo apt-get remove docker docker-engine docker.io

$ sudo apt-get update
```

Installing Packages

```
$ sudo apt-get install apt-transport-https ca-certificates curl software-
properties-common
```

Update apt Sources

To work with the http method package information should be updated and CA certificates are installed

```
$ sudo apt-get update

$ sudo apt-get install apt-transport-https ca-certificates curl software-
properties-common
```

Install Docker's official GPG key

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add –
```

OK

Then it should be checked whether the key is with fingerprint or not:

Setting up the repository

```
$ sudo add-apt-repository \

   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \

   $(lsb_release -cs) \

   stable"
```

Need to update apt package index and install the latest version of Docker CE:

```
$ sudo apt-get update

$ sudo apt-get install docker-ce
```

Verifying that Docker CE is installed correctly by running the hello-world image:

```
$ sudo docker run hello-world
```

Above command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits. Following figure can show the result:

```
nazowa@nazowa-VirtualBox:~$  sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
9db2ca6ccae0: Pull complete
Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/
```

Fig: 4.7 testing successful docker installation

Python Version

For this project Python version 3 has been used and ubuntu 18.04 LTS has the version by default. We can check it by following command:

```
$ python3 –version
```

```
Python 3.6.5
```

And python packages need to be installed to work with the Docker API with following command:

```
$ pip install docker
```

And for Flask framework we must also install flask with following command:

```
$ pip install Flask
```

4.3 Installing Postman APP

For testing the HTTP RESTful functionalities, we are using Postman APP. Any input can be sent from the client side with this app. To use JSON format this app is an easy option. This app can be downloaded from Postman official website(getpostman.com) [12].

4.4 Implementation on Python

This project aims to develop functionalities of Docker with Python script which can be used in Disaster Network in future. Functionalities of Docker that have been deployed on python are as following:

1. Docker info
2. Image list
3. Running container list
4. Inspect container
5. Pull image
6. Create container
7. Start container with container id
8. Run container
9. Docker network creation
10. Docker exec command
11. Stop container
12. Remove Container

These twelve functionalities have been deployed in this project. WMN based WLAN interface command has also been deployed with REST API. For different functionalities different methods have been used based on the requirement. POST and PUT methods are both used to create new resources on the system. However, POST method has been used in this project to send the JSON requests from the client. GET and POST method scenario can be shown as following figures:

GET Method functionalities in Brief

For GET method no input will be sent from client side. It will simply retrieve the information needed from the server. Client will just receive the response which is already store on the system. It does not manipulate any system configuration or cannot create any new resource.

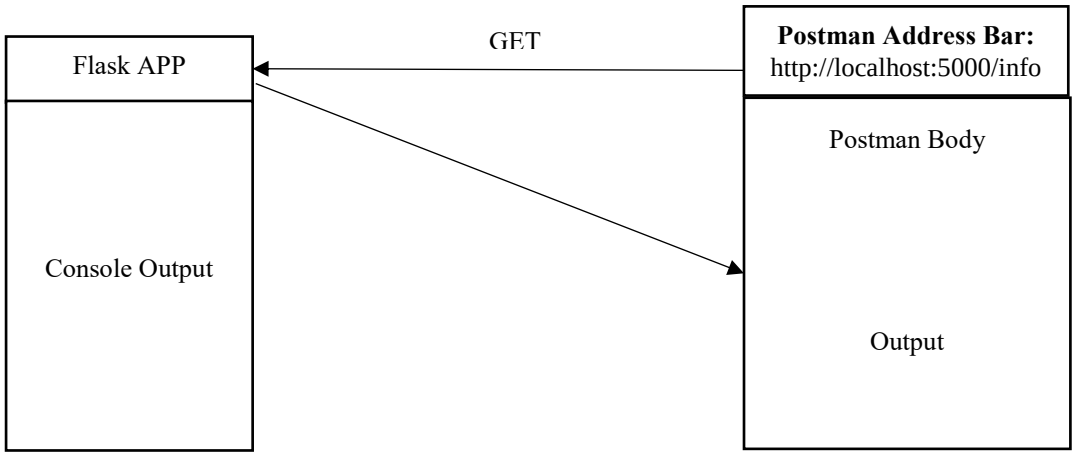

Fig: 4.8 GET method scenario

POST Method functionalities in Brief

This method is used to send the request required input as JSON format. This has been used where input needs to be sent to the server and creating new resources on the system.

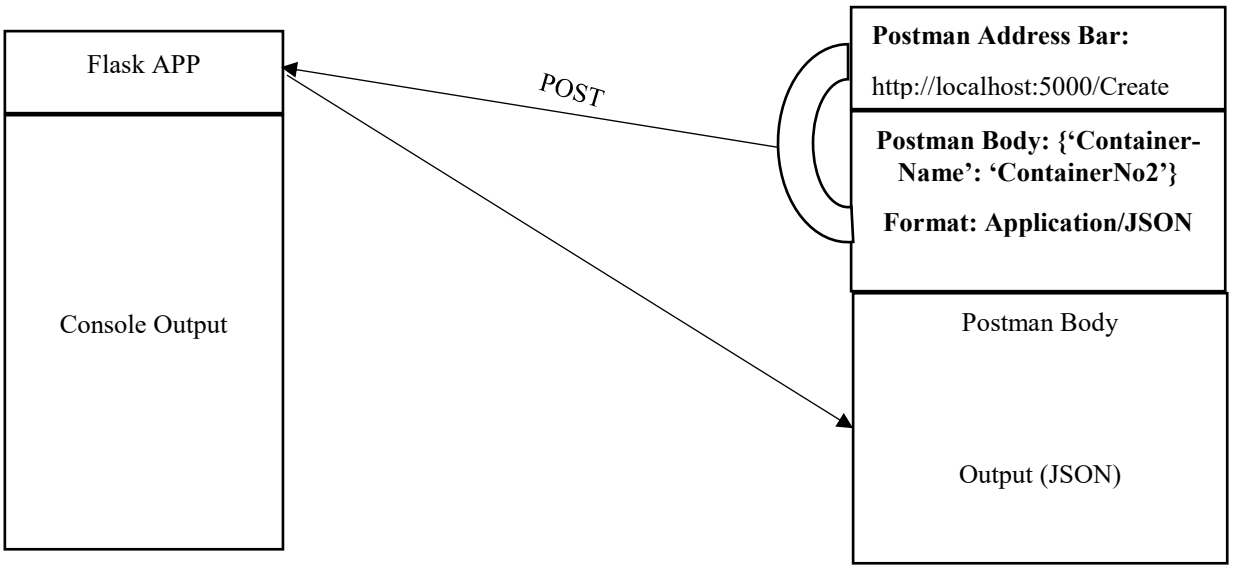

Fig 4.9: POST method scenario

4.5 Detail of Implementation and Testing

1. HTTP client server with Flask Framework

In this part HTTP client-server has been created with Flask framework and POSTMAN app has been used to send the REST API methods from client to the server. It can be created with following source code:

Method: GET

URI: http://localhost:5000/

Code

```
import docker
from flask import Flask, request
import json
import requests
import subprocess #executes terminal commands
app = Flask(__name__)
@app.route("/", methods=['GET'])
def test():
    # global variable to connect to docker environment
```

```
    client = docker.from_env()
    return 'server has been created'
if __name__ == "__main__":
    app.run(debug=True)
```

When this code is executed it is started calling the 'run()' function and as debug is set True so any inconvenience can be handled and also to track the errors with debugger. Here app = Flask(__name__) function acts as an constructor and app.route() function is a decorator in flask to bind URL to a function. As example here, URL is http://localhost:5000/ and GET method is used to retrieve the information from server. Any port can be used instead of 5000. The libraries have been used to work with Flask framework and docker functionalities. As an example, 'import docker' will be responsible to retrieve the docker-py functionalities, 'import json' is used to handle JSON format and 'import requests' to handle HTTP requests. Here 'docker.from_env()' is used as a global variable to connect with docker environment and get the functionalities of docker. This function is used in all functionalities to connect to the docker environment. Subprocess is used in the WLAN information dumping, it is a Python 3 module to execute terminal commands.

2. Retrieving Docker Information (Docker info)

With docker info functionalities docker information can be retrieved as JSON format. Information like existing images, running containers, paused containers and system information as it can be retrieved with the following docker equivalent command on Linux:

```
$ docker info
```

Method: GET

URI: http://localhost:5000/info

Code:

```
@app.route("/info", methods=['GET'])
def info():
    try:
         docker_info = client.info()
        """load ouput in json format"""
        docker_info_string = (json.dumps(docker_info))
        return (docker_info_string)
    except requests.exceptions.HTTPError:
        pass
```

Given URI can be sent from POSTMAN with GET method. With this REST method it will connect to the server. In server side 'client' will connect to docker environment docker.from_env() and then retrieve 'info' functionality of the docker environment. First the 'try' clause is executed and if no exception occurs then 'exception' clause is skipped. If any exception occurs during the 'try' clause the rest of the clause is skipped and 'except' clause is executed and it will show the HTTP errors like 400 Bad Request, 404 not found and so on. Response of 'docker_info' functionalities retrieving by client will be dumped as JSON with 'json.dumps' function of Flask. Here, 'pass' means it will not do anything if exception occurs but will pass the HTTP errors to the client.

Screenshot of the POSTMAN output

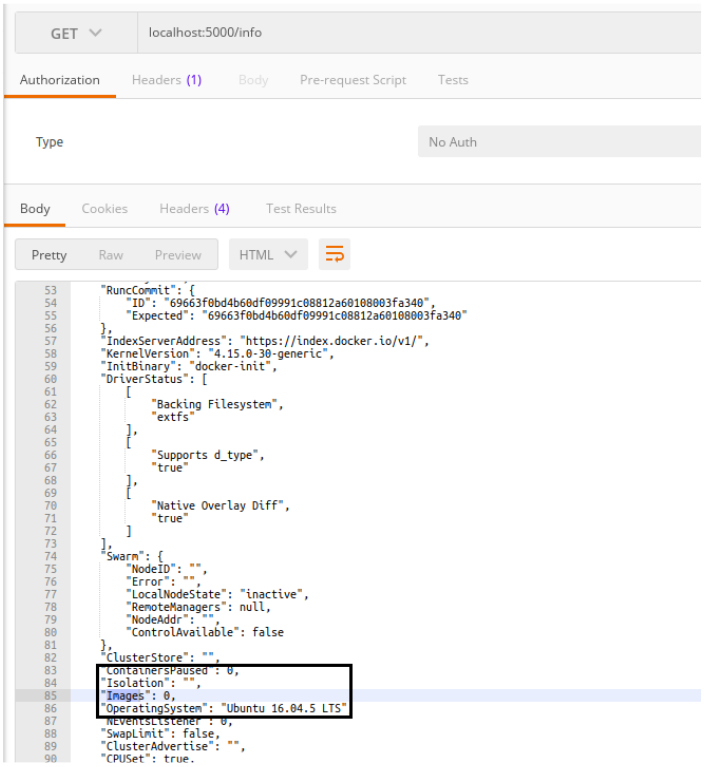

Fig 4.11 Docker info in JSON format.

In above screenshot mentioned information is shown and it shows the system information as it was tested in 'ubuntu':16.04 LTS system.

3 Pull Image

With this functionality image can be pulled from the Docker Hub[10]. A set of images (a repository) or any particular image with tag can be pulled with this functionality. It works like following command of Docker on Linux:

```
$ docker pull IMAGENAME:TAG
```

Method: POST

URI: http://localhost:5000/pull

Code

```
@app.route("/pull", methods=['POST'])
def pull_image_from_hub():
    try:
        data = request.get_json()
```

```
        imagename = data['imagename']
        pulledimage = client.images.pull(imagename)
        return ("pulled image:" +str(pulledimage))
    except requests.exceptions.HTTPError:
        pass
```

Here, from the client side image name will be given as JSON format and it will store on 'imagename' variable as given above. 'request.get_json()' is the function to get input from client as JSON format. After getting the request from client with image name 'try' clause will be executed. If no exception occurs client will be connected to docker environment and then retrieve the functionality of pull image with 'images.pull(imagename)' and will return the image name to the client. It should be returned as 'string' to get the output as it is in docker-py.

Screenshot of the pulled image in POSTMAN and Terminal

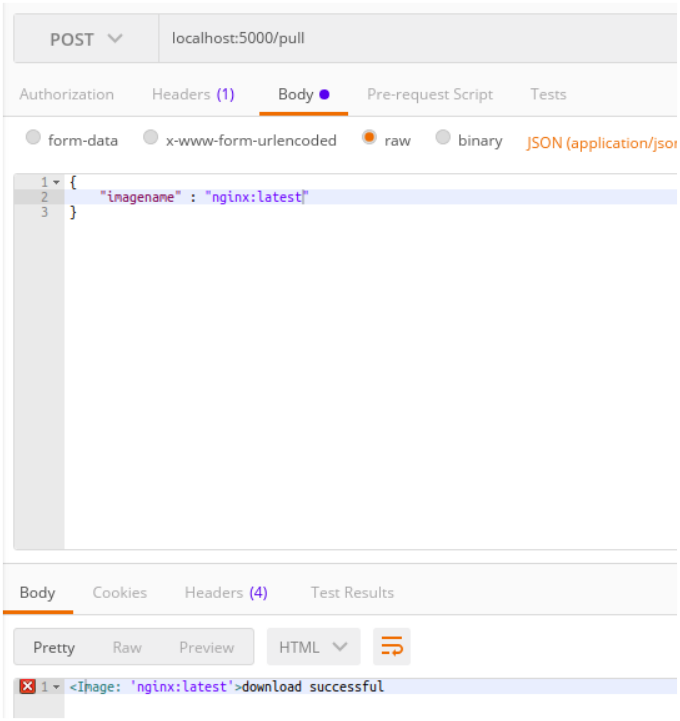

Fig 4.12: Pulling image from Docker Hub

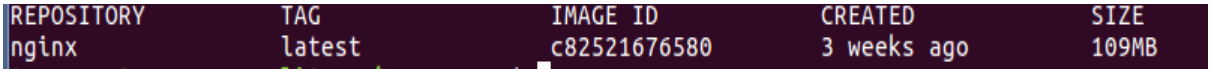

Fig 4.13 Terminal output after pulling image (Terminal command: `docker pull IMAGE:TAG`)

4. Image List

Existing image list can be shown with docker-py. Before pulling there was no image as shown above with Docker info functionality. It works as the docker equivalent command on Linux as following:

```
$ docker images
```

Method: GET

URI: http://localhost:5000/Ilist

Code:

```
@app.route("/Ilist", methods=['GET'])
def show_image_list():
    try:
        li = client.images.list()
        """iterate through the imagelist and print it in server side"""
        iterator = li.__iter__()
        for i in iterator:
            print (i)
        return str(li)
    except requests.exceptions.HTTPError:
        pass
```

Image list can be retrieved by GET method sent from client and after getting connected to the docker environment 'try' clause will be executed. Information will retrieved by client with the function 'images.list()'. To print in console need to iterate through all the elements of image list if it is more than one image. But to return the response to client side it is not needed to iterate as 'return' function can manipulate the whole response in 'string' format.

Screenshots of Image List before and after pulling

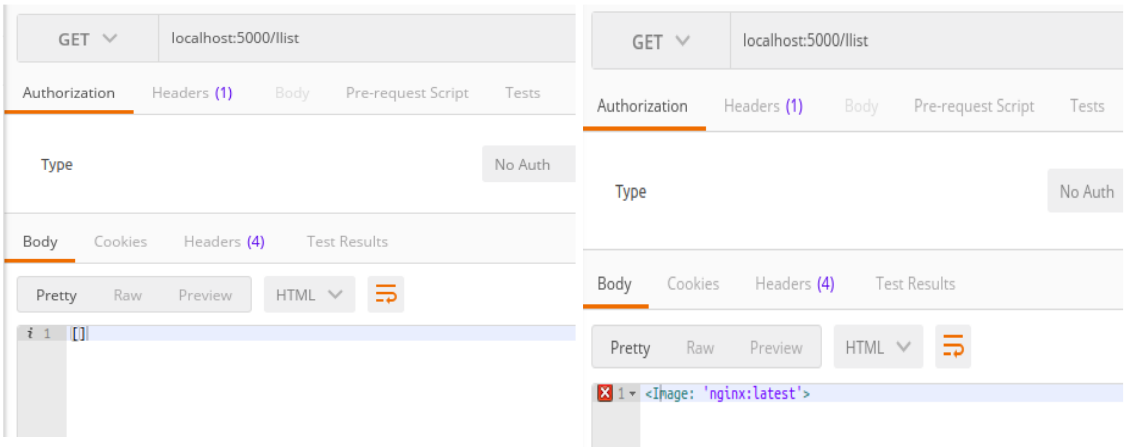

Fig 4.14: Image list before(left) and after pulling(right)

5. Create Container

To create the container POST method has been used as it creates a new resource in the system. From user two inputs will be sent to the server side.'imagename' and 'containername' will be taken from the client side. 'containername' can be defined as client's wish and it will be taken as the given name of the container. Image should be pulled before creating container.It works as the following command:

```
$ docker create IMAGE
```

Method: POST

URI: http://localhost:5000/create

Code:

```
@app.route("/create", methods=['POST'])
def create_container_from_image():
    try:
        data = request.get_json()
        imagename = data['imagename']#defined by user as JSON
        containername = data['containername']#defined by user as JSON
        container = client.containers.create(imagename,detach=True,
                            name=containername)
        return (str(container))
         except requests.exceptions.HTTPError:
            pass
```

Two inputs as JSON format will be taken from the client and will be stored in 'imagename' and 'containername' variable. 'Client' function of the docker-py will create the container with 'conatainers.create()' function with defined 'imagename' and 'containername'. It should be mentioned 'containername' should be different from existing containers name.Here 'detach'=True means it will create and immediately return it's object like container 'id' as shown below.

Screenshots of creating container

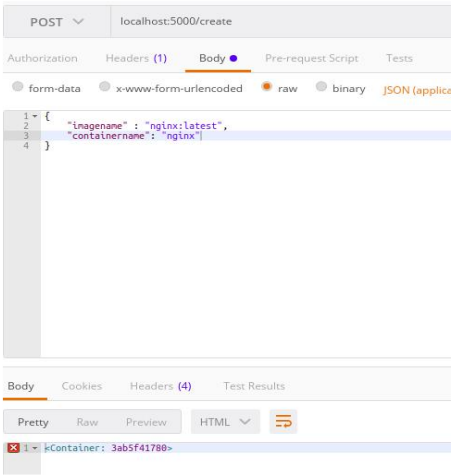

Fig: 4.15 POSTMAN response after creating container

Fig 4.16:Terminal output after creating container(Terminal command: `docker ps -a`)

Above container is created with the given 'imagename' and defined name 'nginx'. From terminal response can be shown with much detail like created time, status but with docker-py only container-id can be shown. So above response is only that what we get from docker-py built in function 'containers.create()'. And docker-py can not print last two digit of container 'id' but it does not limit anything as it takes the 'id' input as shown in terminal. It can referred to below screenshots.

Container can be created only when image is already pulled or already on the system. Docker-py cannot create container directly with any image. So image should be pulled before creating container

6. Start Container

This functionality offers client to start container with container 'id'. Container can be started after creating the container.

```
$ docker start CONTAINER
```

Method: POST

URI: http://localhost:5000/start

Code

```
@app.route("/start", methods=['POST'])
def start_container_from_image():
    try:
        data = request.get_json()
        containerid = str(data['containerid'])
        container = client.containers.get(containerid)
        container.start()
        return ('started container:' + str(container))
    except requests.exceptions.HTTPError:
        pass
```

After getting the request from client with 'containerid' it retrieves the given id with 'containers.get()' function and 'container' variable can start the container with 'start()' function. It will return the container 'id' to the client after starting.

Screenshots of starting container and terminal

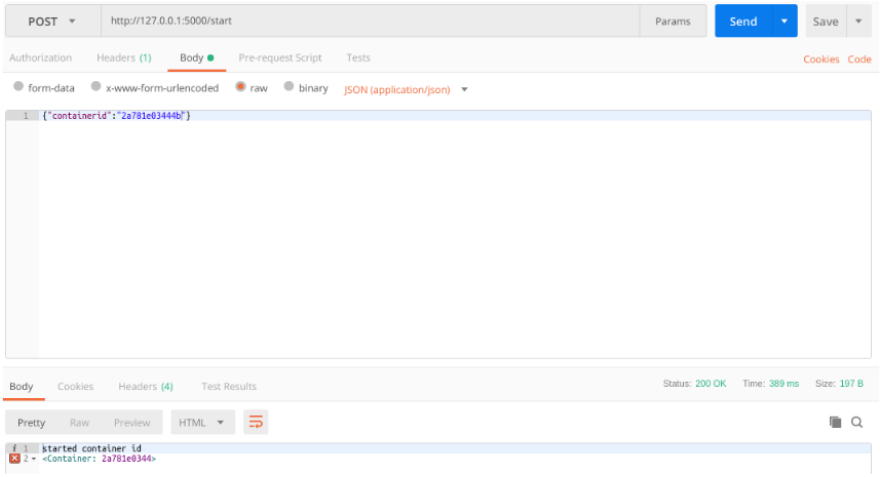

Fig 4.17: Start container with container 'id'

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
        NAMES
2a781e03444b        nginx:alpine        "nginx -g 'daemon ..."   About an hour ago   Up 15 seconds       0.0.0.0:10007->
```

Fig 4.18: Terminal output after(Terminal command: `docker ps`)

Docker-py only shows the container 'id' after starting.

7. Run Container

With POST method container can be run directly with given image. It pulls the image from docker hub directly, then create and run the container. It can take a bit more time than 'start container' functionality as it pulls the image from the docker hub directly.

It works as the following docker command:

```
$ docker run IMAGE
```

Method: POST

URI: http://localhost:5000/run

Code

```
@app.route("/run", methods=['POST'])
def start_container_from_image():
    try:
        data = request.get_json()
        imagename = data['imagename']
        containername = str(data['containername'])
        client.containers.run(imagename, detach=True, name=containername)
```

```
        containerstate = client.containers.get(containername)
        return ('run container id'+ str(containerstate))
    except requests.exceptions.HTTPError:
        pass
```

It also works as create container functionality, takes two input from user ‘imagename’ and ‘containername’. During creating container ‘conatainarname’ is also defined user’s choice and here it can also be done on running containers with user given name that can be shown below as screenshot. ‘client’ function of docker-py runs the container with ‘containers.run()’ function. It takes three parameters. Here, ‘detach=True’ means it will start container immediately and return container object like container ‘id’ immediately. With ‘containers.get()’ function container ‘id’ can be returned to the user as response.

Screenshots of running container

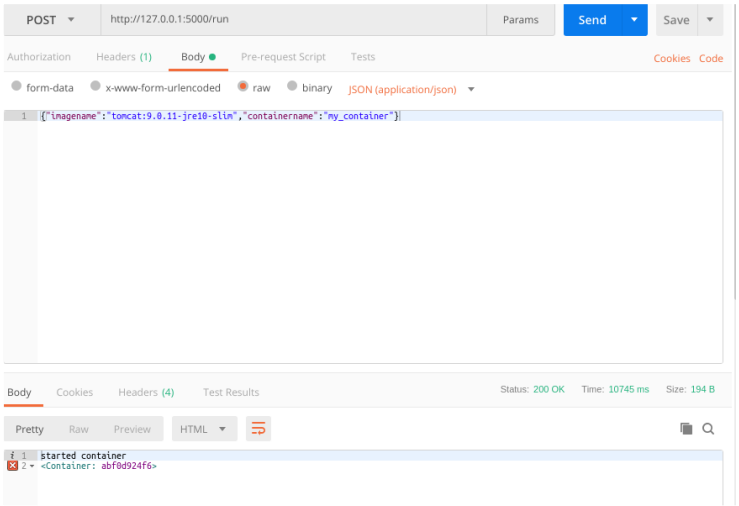

Fig 4.19: Run Container with defined container name

Here container has been run with given image and has been named as ‘my_container’. This name should be unique if any name matches with this name it will conflict and show error in the program.

8. Container List

Docker-py can show only list of the running containers, not all the containers that have been created. It works as following docker equivalent command:

```
$ docker ps
```

Method: GET

URI: http://localhost:5000/Clist

Code:

```
@app.route("/Clist", methods=['GET'])
def show_container_list():
    try:
        container_list = client.containers.list()
```

```
        return ('\n'+'online containers \n'+str(container_list))
    except requests.exceptions.HTTPError:
        pass
```

Container list can be shown with docker-py function ‘containers.list()’ and it will return container list to the user side(POSTMAN).

Screenshot of the list of running containers

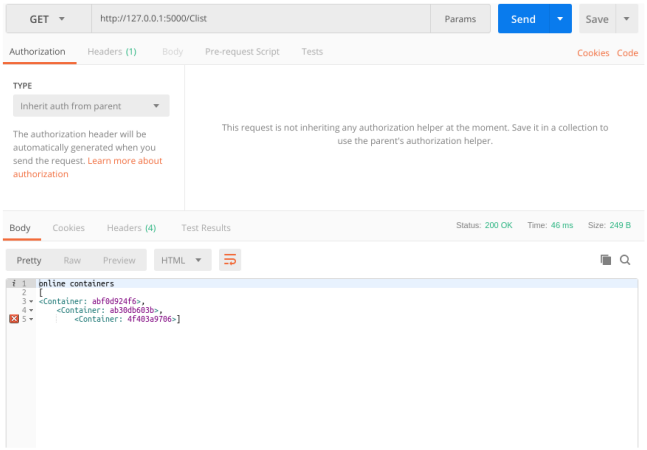

Fig 4.20: List of running containers

9. Inspect Container

With this functionality container can be inspected with its name and ‘id’. ‘Id’ or ‘name’ of the container must be defined by the user that has to be inspected. It works as following docker command on terminal:

```
$ docker inspect NAME/ID
```

Method: POST

URI: http://localhost:5000/inspectCont

Code:

```
@app.route("/inspectCont", methods=['POST'])
def inspect_running_container():
    try:
        data = request.get_json()
        containerID = data['containerID']
        inspect = client.api.inspect_container(containerID)
        """Return in json format"""
        inspect_info = (json.dumps(inspect, indent=4))
        return (inspect_info)
    except requests.exceptions.HTTPError:
        pass
```

to inspect container ‘api.inspect_container()’ function has been used. This function provides default response in JSON format. This response is dumped into JSON with ‘json.dumps()’ function. This response will be retuned to POSTMAN with ‘return’ function.

Screenshot of docker inspect

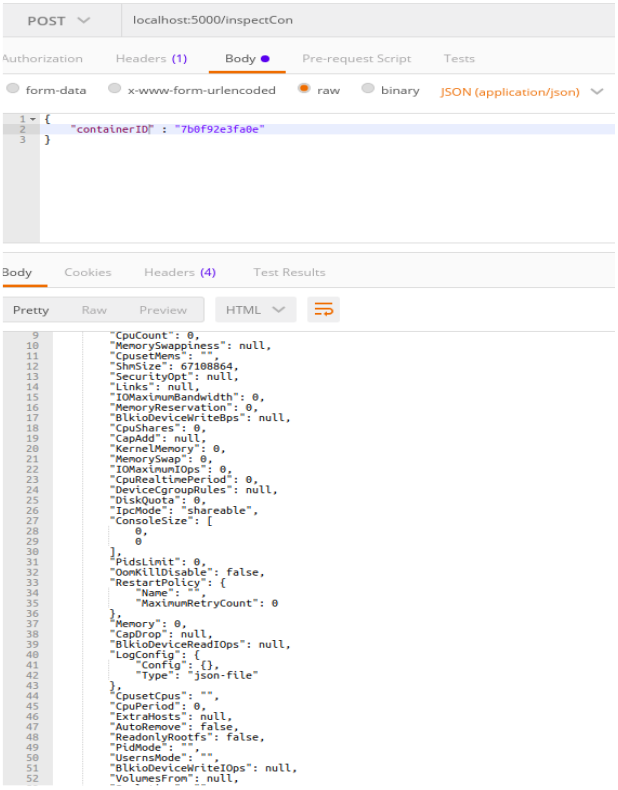

Fig 4.21: Docker inspect

Here, Docker inspect functionality is shown with container ‘id’. Container name can be used also instead of ‘id’.

10. Docker Network Creation

In docker defined network can be created with given network name, subnet, IP range, driver type and many more options. This functionality can be provided also with docker-py. This section shows the detail of docker network creation with docker-py. Docker equivalent command is following:

```
$ docker network create [OPTIONS] NETWORK
```

These OPTIONS can be bridge name, IP range, driver typre and so on.

Method: POST

URI: http://localhost:5000/networkcreate

Code

```
@app.route("/networkcreate", methods=['POST'])

def network_create():
```

```
try:
    data = request.get_json()
    networkname = data['networkname']
    subnet = data['subnet']
    iprange = data['iprange']
    gateway = data['gateway']
    bridgename = str(data['bridgename'])
    ipam_pool = docker.types.IPAMPool(subnet, iprange, gateway)
    ipam_config = docker.types.IPAMConfig(pool_configs=[ipam_pool]
    networkcreate = client.networks.create(networkname, driver =
  "bridge", ipam = ipam_config, options={
        "com.docker.network.bridge.name": bridgename })
    return str(networkcreate) + 'created successfully'
except requests.exceptions.HTTPError:
    pass
```

There are four parameters which has to be given by client and docker-py recognize and configure these network attributes with 'IPAMPool()' and 'IPAMConfig()' function. In docker driver type can be bridge, overlay or MACVLAN.It has to be defined by client. This bridge name can be made dynamic with 'options' field of docker-py as shown above.

Screenshot of network creation

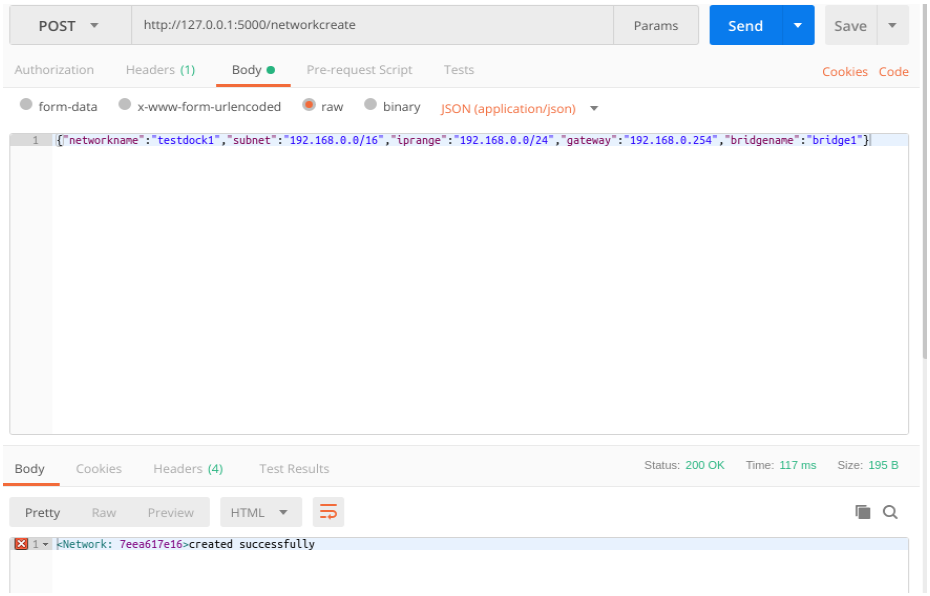

Fig 4.22: Docker network creation

11. Docker Exec Container

With docker exec functionality a command like ifconfig, pwd can be run on running container. This is executed with following command:

```
$ docker exec [OPTIONS] CONTAINER COMMAND
```

Method: POST

URI: http://localhost:5000/containerexec

```
@app.route("/containerexec", methods=['POST'])
def container_exec():
    try:
        data = request.get_json()
        imagename = data['imagename']
        containername = str(data['containername'])
        command = str(data['command'])
        container=client.containers.run(imagename, detach=True, name=con-
tainername, command=command)
        log = str(container.logs())
        runningcontainer = client.containers.get(containername)
        return "Container is: " + str(runningcontainer) + "and log is " +
str(log)
    except requests.exceptions.HTTPError:
        pass
```

With given parameters ‘imagename’ and ‘containername’ docker-py will run the container with ‘containers.run()’ function. Here ‘detach=True` means it will run the container in the background. To print and return the result ‘containers.logs()’ function is used. It will return log of the given command to the client. Following screenshot can be referred to understand better.

Screenshot of Exec Functionality

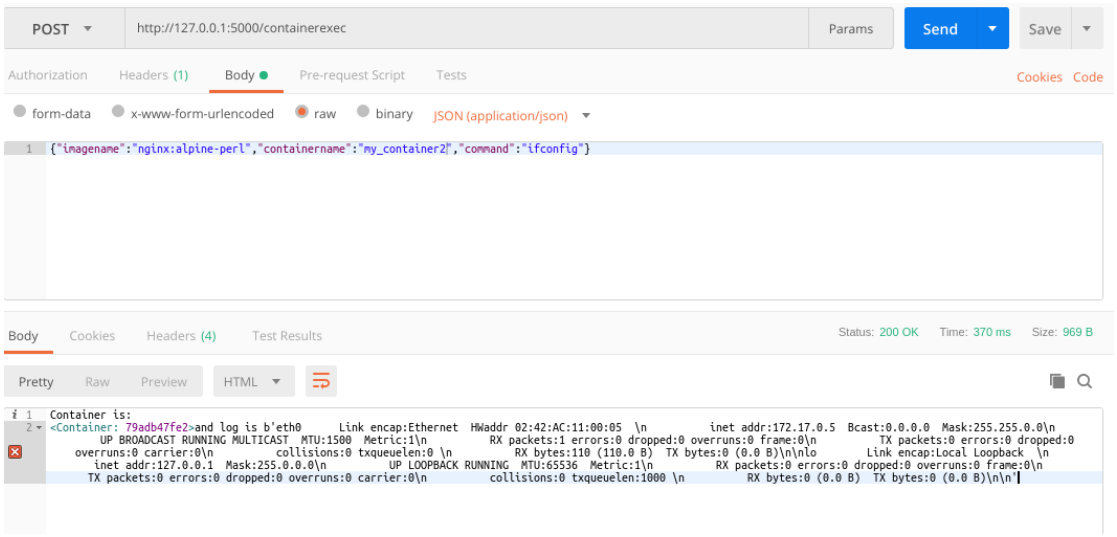

Fig 4.24: Screenshot of POSTMAN of Exec functionality

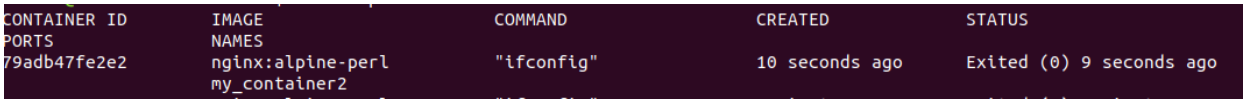

Fig 4.25: Terminal Output of Docker Exec Response (Terminal Command: `docker ps -a`)

In POSTMAN 'ifconfig' command information is shown. It is not that much pretty printing as docker-py doesn't provide those. It can be regarded as future scope to convert the response in JSON format.

In terminal screenshot that can be shown that container exited automatically after running. So it does not show as a running container. What it does is it runs the container based on request from the container and runs the given command inside running container and then exited automatically. As it is controlled with REST API, so it cannot handle the running container to stay online. As a result it stops automatically.

Fig 4.23: Bridge name defined by Client (Terminal command: docker network inspect NETWORKNAME)

Here it can be shown that network is created with defined input from client. To create network with same parameter values created network has to be removed before creating new one.

12. Stop Container

Stop Single Container
This function can be used to stop one running container and all running containers. To stop one container 'POST' method should be used to define the name of container from the user side. It works like following docker command:

$ `docker stop CONTAINERNAME`

Method: POST

URI: http://localhost:5000/stoponecont

Code:

```
@app.route("/stoponecont", methods=['POST'])
def stop_one_cont():
    try:
        data = request.get_json()
        containerid = data['containerid']
        container=client.containers.get(containerid)
        container.stop()
        return ('stopped containenr id'+str(container))
    except requests.exceptions.HTTPError:
        pass
```

Container 'id' will be stored in the variable 'containerid' and it will be passed as parameter into 'containers.get()' function. It will also store the id in a variable 'container' and this variable will use 'stop()' function of docker-py to stop container.

Screenshot of stopping single container

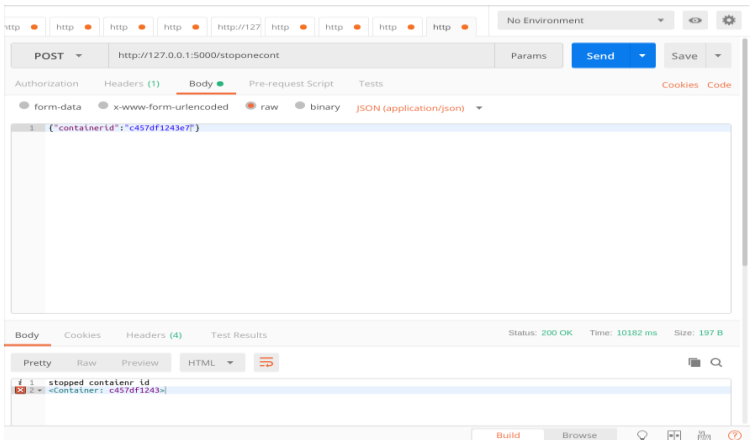

Fig 4.26: Stopped Container

Stop all running containers
Docker-py provides a functionality to stop all the containers at a time. It woks as following docker command:

```
$ docker stop $(docker ps -a -q)
```

Method: GET

URI: http://localhost:5000/stop

```
@app.route("/stop", methods=['GET'])
def stop():
    try:
        container_list = client.containers.list()
        for container in container_list:
            container.stop()
        return "All container stopped"
    except requests.exceptions.HTTPError:
        pass
```

To stop all the containers, it needs to be iterated through all the containers. In above code 'container' variable will iterate through all the running containers in the list and every element will be stopped by 'container.stop()' function.

Screenshot of stopping all the containers

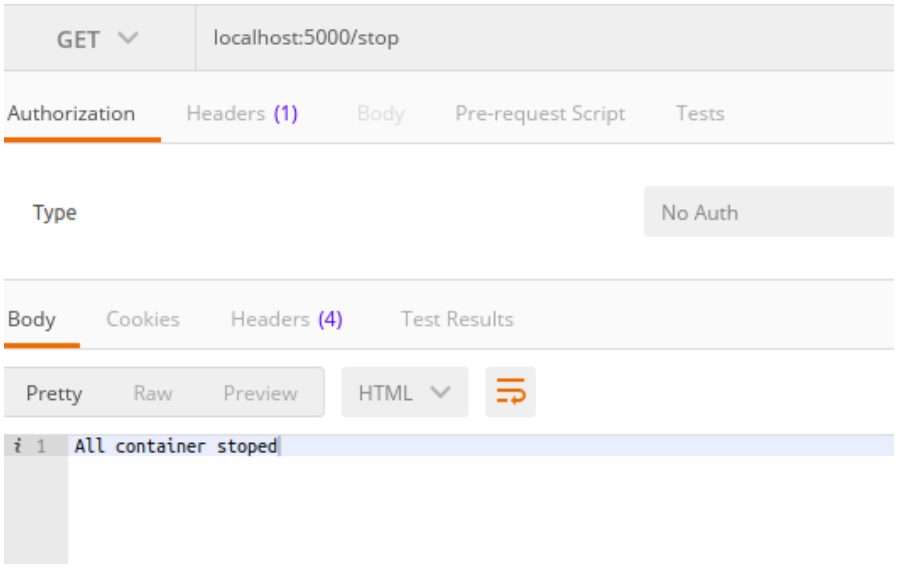

Fig 4.27: Stopped all running containers

13. Remove Containers

All stopped containers can be remove with the DELETE method sent from the POSTMAN app. It will work as following:

```
$ docker rm $(docker ps -a -q)
```

Method: DELETE

URI: http://localhost:5000/remove

Code:

```
@app.route("/remove", methods=['DELETE'])
def remove_existing_container():
    try:
        delete = client.containers.prune()
        return (str(delete))
    except requests.exceptions.HTTPError:
        pass
```

with 'containers.prune()' docker-py can delete all the containers.

Screenshot of removing Containers

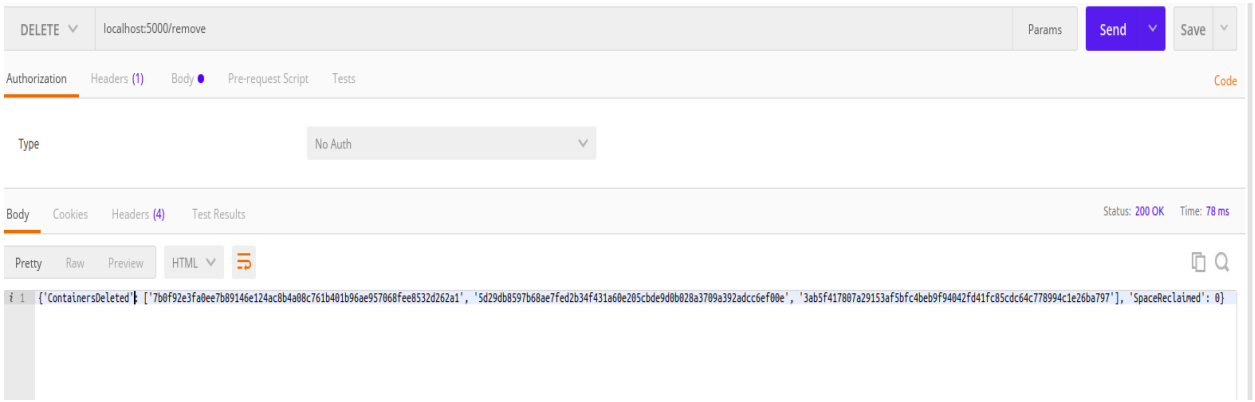

Fig 4.28: Deleted Containers

After executing the code it will show all the deleted containers id in POSTMAN app.

Above functionalities are the docker functionalities created based on docker-py python SDK. As this program aims to be deployed in WMN based disaster network one demo scenario of the WMN is created on linux terminal. In practical WMN scenario is difficult to deploy in the lab, so a network with three WLAN interface is created on linux terminal.This implementation is given below with linux terminal commands.

14. Creating WLAN interface command (station dump)

This project aims to be deployed in WMN scenario. As in practical it is hard to realize the network a dummy network is created on Linux. Some commands have to be run on Linux terminal to create the wireless network. Comamands are given below:

To create the network all radios should be stopped first. It can be done with following command:

```
$ sudo modprobe -r mac80211_hwsim
```

Sudo has to be used if it is not root user,

```
$ sudo modprobe  mac80211_hwsim radios =X(X=number of radios)
```

To check the radios are created,

```
$ ifconfig
```

It will show the wlan0,wlan1,wlan2

```
$ sudo ifconfig <wlan interface> down
```

It will make the wlan down.It shouldbe down to set mesh network type.

```
$ sudo ifconfig <wlan interface> set type mesh
```

It will set wlan interface (e.g: wlan0)type mesh

```
$ sudo ifconfig <wlan interface> up
```

Then interface should be up before joining network.

```
$ sudo ifconfig <wlan interface> up
```

```
$ sudo iw dev <wlan interface>mesh join testnet
```

It will join wlan interface with ‘testnet’ network

```
$ iw dev <wlan interface>station dump
```

It is the required Command with REST API. It shows the information about the defined station (e.g:wlan0).

It can be implemented with below code:

Method: POST

URI: http://localhost:5000/meshnet

```
@app.route("/meshnet", methods=['POST'])
def station_dump():
    try:
        data = request.get_json()
        interface = data['interface']
        i=subprocess.call(["iw", "dev",interface, "station", "dump"])
        return str(i)
    except requests.exceptions.HTTPError:
        pass
```

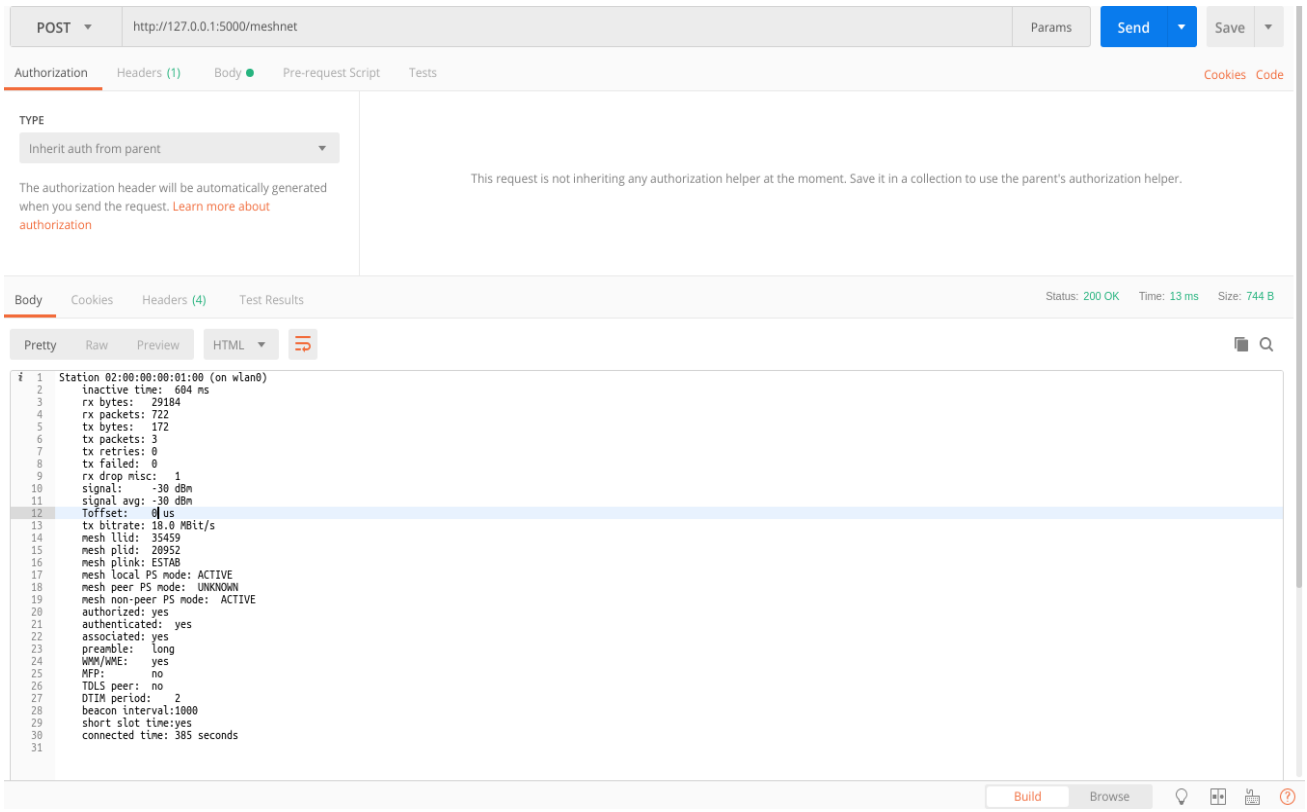

Fig: 4.29 Station Dump Command output with REST API

For this implementation subprocess module of python 3 is used. User has to give input of wlan interface name (e.g:{“interface”:”wlan0”}) from the POSTMAN body. This is shown as the format of terminal output.

4.5 Example of Integrating Implementation Code

With above details one can implement the code with desired functionalities. Here an example is shown for convenience. With following code one can implement the functionality of docker pull. Similarly any functionality can be implemented integrating the functions as following:

```
import docker

from flask import Flask, request

import json

import requests

import subprocess

app = Flask(__name__)

@app.route("/", methods=['GET'])

def test():

    # global variable to connect to docker environment

    client = docker.from_env()
```

```
    return 'server has been created'
@app.route("/pull", methods=['POST'])
def pull_image_from_hub():
    try:
        data = request.get_json()
        imagename = data['imagename']
        pulledimage = client.images.pull(imagename)
        return ("pulled image:" +str(pulledimage))
    except requests.exceptions.HTTPError:
        pass
if __name__ == "__main__":
    app.run(debug=True)
```

Like above all functionalities can be implemented and controlled docker aspects with REST API from POSTMAN.

5 Summary and Future Scope

Twelve functionalities of docker have been implemented in this project. It was implemented using a VM as Linux based system is needed to run this implementation. It should be mentioned that Python 3 version is used in this project and to work Python version 2 there will be changes as docker-py has different syntax depending on the version. Flask framework has been chosen to implement REST API as it is an easiest option to use. All input given from client end must be in JSON format otherwise this program will not work. Some notes should be taken in mind that some images that is used to run container stops immediately after running, as an example "hello-world:linux" image. It is because that it is as set in the docker hub and it is not because of this project implementation. Therefore, all docker functionalities can be controlled by REST API. Docker functionalities have been provided with docker-py module. Though this can also be provided with Subprocess module of Python which we used in 'WLAN interface command creation' but docker-py is a better choice as it offers more functionalities than subprocess.

Though twelve functionalities of docker have been created in this project but much more can be done in future, as example creating containers with more options like CPU/RAM usage, making container interactive, defining MAC address and so on. Functionalities like 'docker stats', 'docker commit' can also be done in future. This implementation can be extended for providing LXC/LXD functionalities. Some functionalities lack the response in JSON format, so it can be also a field to work with. And it provides a great possibility to deploy in Disaster Network and docker aspects of the disaster network can be controlled by the REST API of this program.

6 Abbreviations

A

API	Application program interface

L

LXC	Linux Container
LXD	Linux Container Daemon

R

REST	Representational State Transfer

V

VM	Virtual machine

7 References

1. Balamurali Thekkedath(2016): *Network Functions Virtualization For Dummies,* John Wiley & Sons, Inc.
2. FN Division Telecommunication engineering centre: *Network Function Virtualization (NFV) & Its impact on Future Telecom Networks,* http://tec.gov.in/pdf/Studypaper/Network_Function_Virtualization%20.pdf, [accessed 27th July, 2018].
3. ETSI Group Specification(2014): *"Network functions virtualization (NFV*); management and orchestration,"https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf [accessed 21st August, 2018]
4. *Restful Web Services,*https://www.tutorialspoint.com/restful/restful_introduction.htm[accessed 1st July, 2018]
5. *Full Stack Python,*https://www.fullstackpython.com/flask.html[accessed 4th July,2018]
6. Research Group for Telecommunication Networks Frankfurt University of Applied Sciences:2016: *Optimization of Wireless Disaster Network Through Network Virtualization,*

   https://e-technik.org/frame_aufsaetze_vortraege.htm [accessed 21st August, 2018].
7. *Docker SDK for Python,* https://docker-py.readthedocs.io/en/stable/ [accessed 7th July, 2018]
8. Turnball(2014): *The Docker Book,*http://osgp88fat.bkt.clouddn.com/books/The%20Docker%20Book.pdf[accessed 2nd July, 2018]
9. *Docker SDK for Python,* https://docker-py.readthedocs.io/en/stable/,[2nd June, 2018].
10. *VirtualBox,* https://www.virtualbox.org/ [accessed 26th May, 2018]
11. *Download Ubuntu desktop,* https://www.ubuntu.com/download/desktop, [accessed 26th May, 2018].
12. *Postman Makes API Development Simple,* https://www.getpostman.com/, [accessed 14th June, 2018].