PROJECT: Implementation and Demonstration of Load Balancer for HTTP Requests in MS Azure using Virtualization Techniques

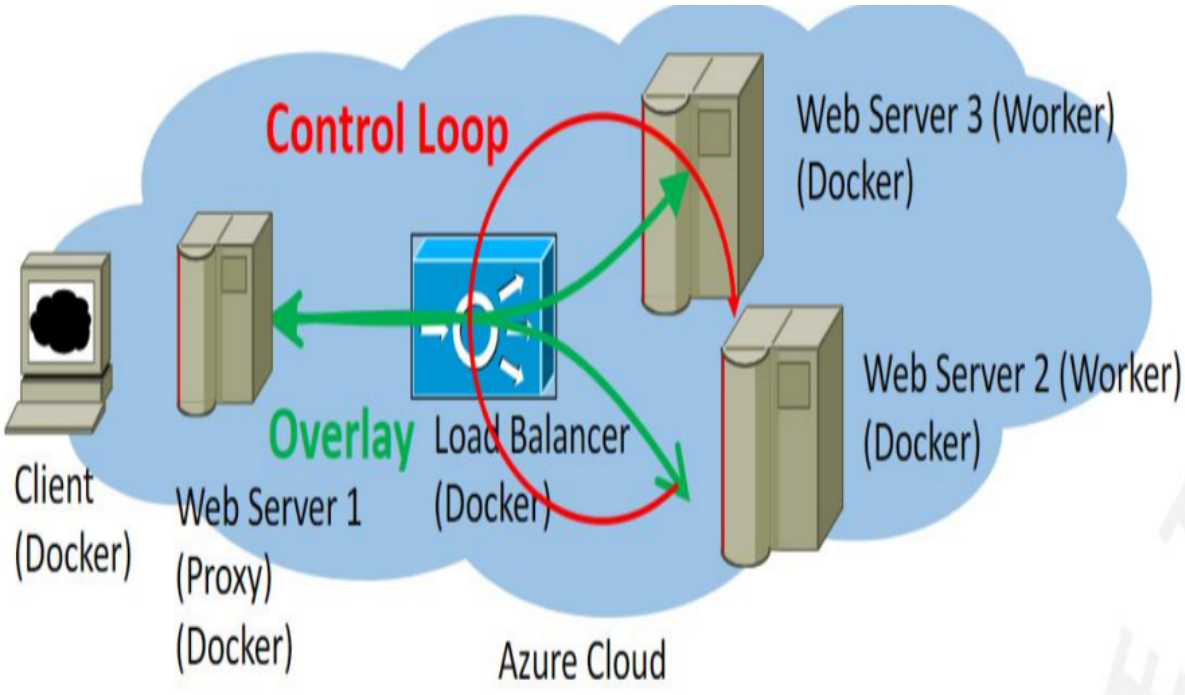
Prajna Bala Sai Potnuri
BTH ID : 20001026-4711
prpo21@student.bth.se

Sri Phani Deverakonda
BTH ID: 20010604-1254
srdv21@student.bth.se

AIM: To implement and demonstrate a Load Balancer for HTTP Requests in MS Azure using Virtualization Techniques.

Tools used: Ubuntu machine, Azure virtual machine.

Procedure: Three virtual machines (VM4, VM5, and VM6) on Microsoft Azure were provided to us. The other two virtual machines are our web servers, and VM9 is our load-balancing server. We utilized the Flask module in Python to configure the webservers and the load-balancing server.



Considering the situation above, we chose VMs 4 and 5 to function as web servers 2 and 3.

Additionally, VM 6 serves as a load-balancing server (proxy server) for Webserver 1.

Setting up web servers:

We put in place a web service that shows a webpage when a customer requests it. We responded to this using a little Python script called app.py. This script sets up a web server that is listening on port 8080 and sends out a brief HTML page that includes the hostname of the container and a greeting.

Setting up Virtual Machines:

VM4:

```
vm4@et2599vm4:~/prajna$ cat app.py
from flask import Flask, render_template, request, send_file, redirect, url_for, session
import os, psutil, json, time, subprocess, sys
import socket
app = Flask (__name__)

@app.route("/")

def hello():

    html = "<h3>Hello {name}!</h3> <b>Hostname:</b> {hostname} <h3> This is from VM4 </h3> <br/>"

    return html.format(name=os.getenv("NAME", "Kurt"), hostname=socket.gethostname())

@app.route("/load")

def load():

    load=psutil.cpu_percent()

    return str(load)

if __name__ == "__main__":

    app.run(host='0.0.0.0', port=8080)
```

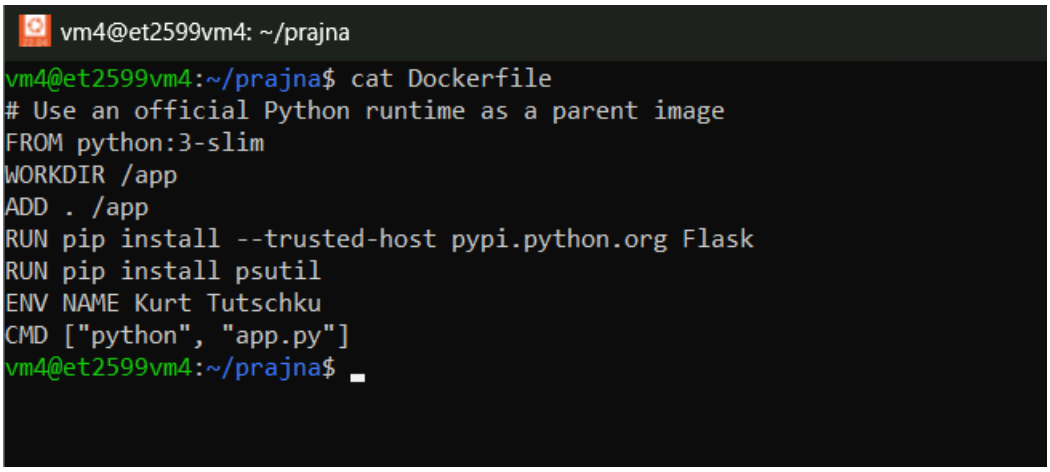
VM5:

```
vm5@et2599vm5:~$ cat app.py
from flask import Flask, render_template, request, send_file, redirect, url_for, session
import os, psutil, json, time, subprocess, sys
import socket
app = Flask (__name__)

@app.route("/")

def hello():

    html = "<h3>Hello {name}!</h3> <b>Hostname:</b> {hostname} <h3> This is from VM5 </h3> <br/>"

    return html.format(name=os.getenv("NAME", "Kurt"), hostname=socket.gethostname())

@app.route("/load")

def load():

    load=psutil.cpu_percent()

    return str(load)

if __name__ == "__main__":

    app.run(host='0.0.0.0', port=8080)
```

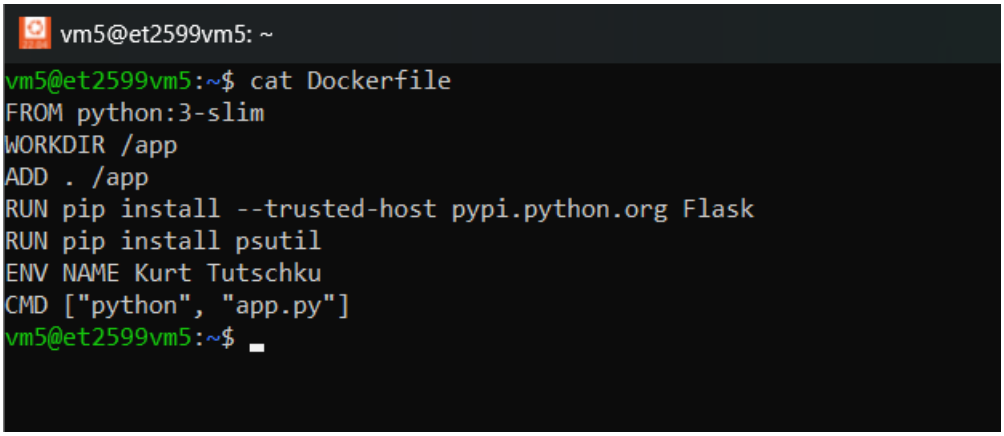
Now we created the docker files.

Docker file for webservers:

VM4:



VM5:



The Docker files start with a base image containing the Python runtime. Subsequently, the current working directory is added to the working directory (/app) using the ADD instruction. After adding the script to the image, we proceed to install the Flask Python package, a library utilized for the web server. Notably, the installation of Flask is carried out using the RUN instruction, allowing the execution of commands during the image-building process.

In the HTML page generated by app.py, there is a reference to the environment variable NAME. For this purpose, the variable is set to (Kurt Tutschku).

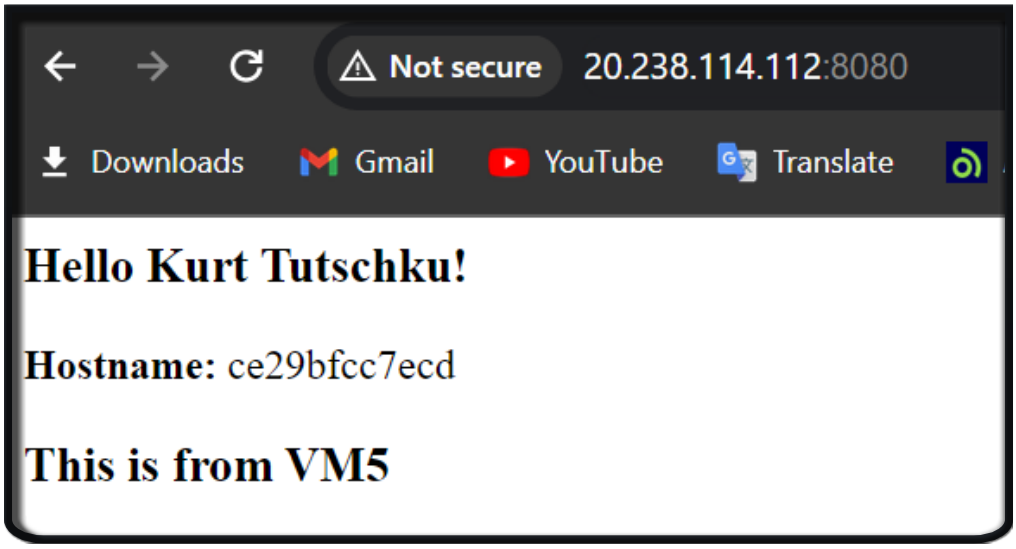
Finally, the Docker file specifies the command to execute when the image is run using the CMD instruction. It accommodates both the command and a list of parameters to be passed to the command.

So,

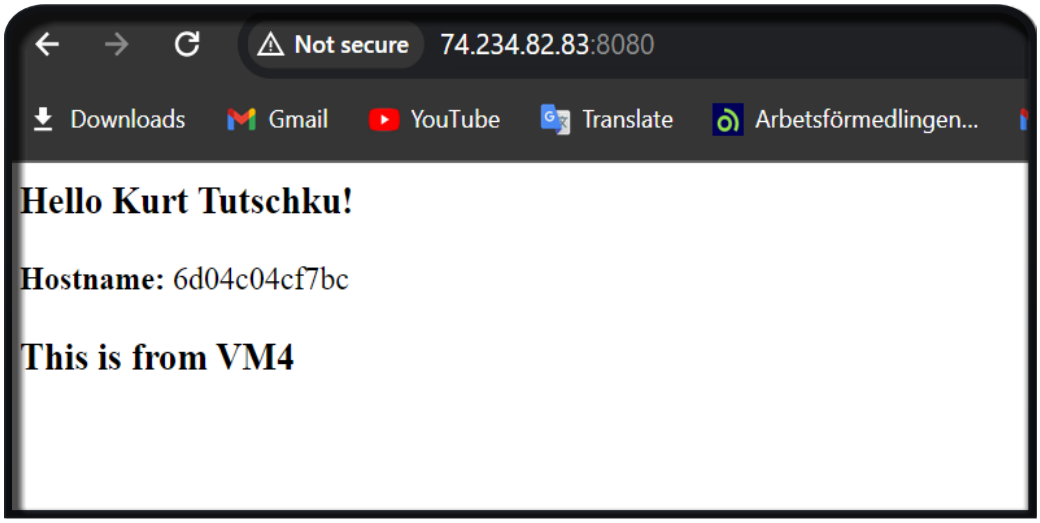
- Initially, we verify if our web servers function as intended.
- After that, we launch the Docker file and create an image.
- Following the command (docker build -t name.) we construct an image.
- Next, we launch our Docker by running the command " sudo docker run --name vm4 -p 8080:8080 image01"

The following are the responses to the VMs found on each website:

VM4:



VM5:



Configuring the Load Balancer with Round Robin:

In the Round Robin algorithm, each request to the load balancer is sequentially distributed among available servers. To configure this load balancing approach, create a new folder. Within this folder, create a Dockerfile and a Python file following the provided code below:

Python code:

```
vm6@et2599vm6:~$ cat rr.py
from flask import Flask, jsonify
import requests

app = Flask(__name__)

# Define the IP addresses and ports of servers to be load balanced
server_configs = [
    {"ip": "74.234.82.83:8080", "name": "Server 4"},
    {"ip": "20.238.114.112:8080", "name": "Server 5"}
]

current_server_index = 0

@app.route('/')
def load_balance():
    global current_server_index

    # Get the current server configuration
    current_server = server_configs[current_server_index]

    # Increment the index for the next request (round-robin)
    current_server_index = (current_server_index + 1) % len(server_configs)

    # Get the content from the selected server
    selected_server_content = get_server_content(current_server["ip"])

    # Prepare response
    response = {
        'selected_server': current_server["name"],
        'selected_server_content': selected_server_content
    }

    return selected_server_content  # Return the entire HTML content

def get_server_content(server_ip):
    try:
        response = requests.get(f"http://{server_ip}")
        return response.content.decode('utf-8')
    except requests.exceptions.RequestException as e:
        # Handle the case where a server is unreachable
        print(f"Warning: Unable to retrieve content from {server_ip}. Error: {e}")
        return f"Unable to retrieve content from {server_ip}"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

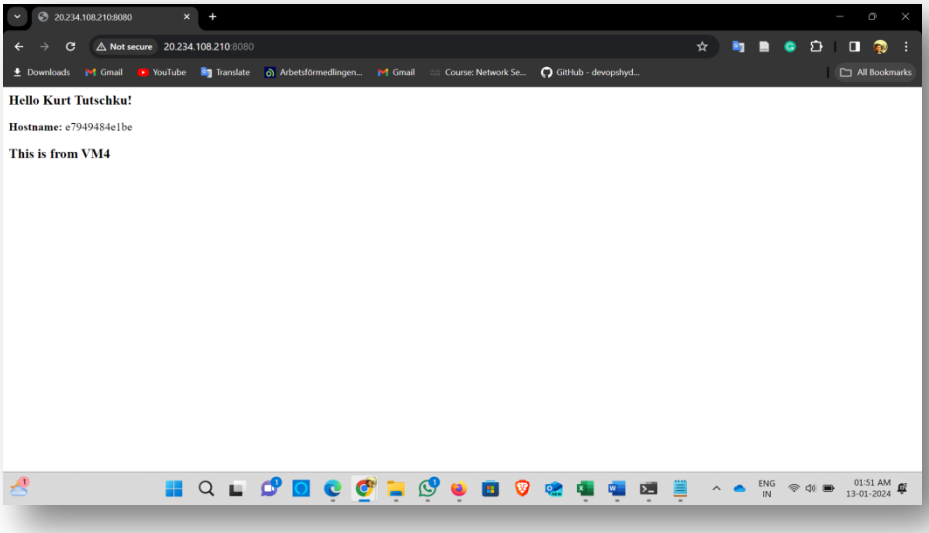
In the implemented load balancing logic, a list of server configurations, denoted by **server_configs**, has been defined, comprising IP addresses and ports of servers to be balanced. The variable **current_server_index** is initialized to zero, signifying the starting point in the round-robin sequence. Upon receiving a client request, the load balancer updates the **current_server_index** to point to the next server in the list. This cyclic incrementation ensures a sequential distribution of requests among the available servers. Subsequently, the load balancer extracts the content from the selected server using the **get_server_content** function. The retrieved content is then encapsulated in a JSON response, including details such as the selected server's name and its content. This approach allows for an efficient and straightforward Round Robin load balancing strategy, ensuring each server handles requests in a sequential manner.

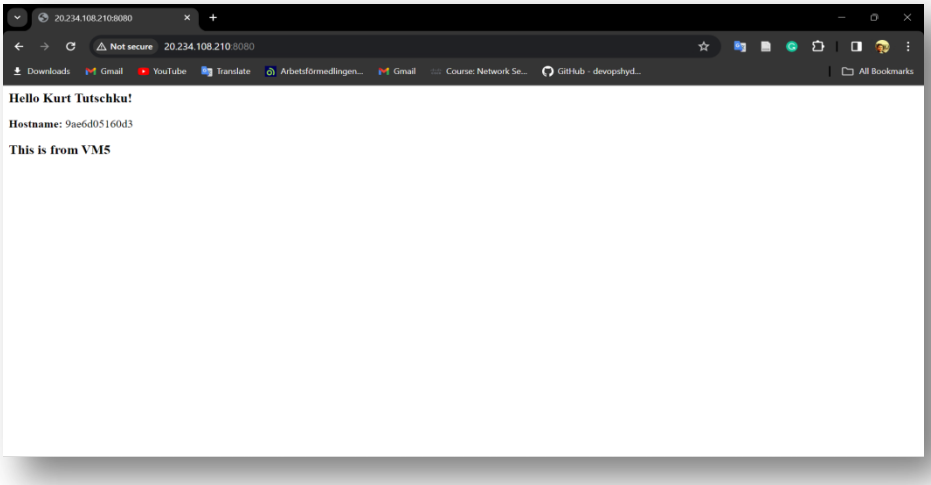
Docker File:

```
vm6@et2599vm6:~$ cat Dockerfile
FROM python:3-slim
WORKDIR /app
ADD . /app
RUN pip install --trusted-host pypi.python.org Flask
RUN pip install psutil
RUN pip install requests
ENV NAME Kurt Tutschku
CMD ["python", "rr.py"]
```

In crafting the Dockerfile, I began by selecting the official Python 3 slim image as the foundational base image. The working directory within the container was established at /app, providing an organized space for subsequent operations. The entirety of the local directory was then transposed into the container's working directory. To fortify the environment, I executed several pip install commands, introducing Flask, psutil, and requests as essential dependencies. An environmental variable named NAME was defined, set to the value "Kurt Tutschku" to facilitate configuration. The final touch involved specifying the command for container execution, opting for the Python script rr.py over the original app.py for Round Robin functionality.

Result : VM4->VM5->VM4->VM5……





Configuring the Load Balancer with Weighted Round Robin:

In the Weighted Round Robin algorithm, incoming requests to the load balancer are distributed among available servers in a sequential manner, but with a consideration for predefined weights assigned to each server. To set up this load balancing approach, initiate by creating a new folder. Within this folder, craft a Dockerfile and a Python file using the provided code below:

Python code:

```
vm6@et2599vm6:~$ cat wrr.py
from flask import Flask, render_template
import requests

app = Flask(__name__)

# Servers' configurations (as per your original code)
servers = [
           {"host": "74.234.82.83:8080", "identifier": "Alpha", "load_factor": 3},
               {"host": "20.238.114.112:8080", "identifier": "Beta", "load_factor": 5}
               ]

# List for managing server rotation (as per your original code)
rotation_list = []
for index, server in enumerate(servers):
        rotation_list += [index] * server["load_factor"]

        rotation_counter = 0

@app.route('/')
def balance_load():
    global rotation_counter
    chosen_server = rotation_list[rotation_counter]
    rotation_counter = (rotation_counter + 1) % len(rotation_list)
    content = get_content(servers[chosen_server]["host"])
    return render_template('index.html', chosen_server=servers[chosen_server]["identifier"], content_from_server=content)

def get_content(host):
    try:
        resp = requests.get(f"http://{host}")
        resp.raise_for_status()
        return resp.text
    except requests.exceptions.RequestException as e:
        print(f"Error: {host} unreachable. Details: {e}")
        return f"Error: {host} unreachable"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

In this Flask application, the code implements a basic load balancing mechanism using a round-robin algorithm among two backend servers, denoted as "Alpha" and "Beta." The `servers` list holds configurations for each server, including their host addresses, identifiers, and load factors, where the load factor signifies the weight of each server in the load balancing process. The `rotation_list` is a sequence of indices that determines the order in which servers are selected based on their load factors.

The `rotation_counter` variable is initialized to zero and tracks the current position in the rotation list. The `/` route, triggered by a client request, selects a server from the rotation list using the `rotation_counter`, fetches content from that server using the `get_content` function, and renders an HTML template ('index.html') with information about the chosen server and its content.

The `get_content` function attempts to fetch content from the specified server's root endpoint using the `requests.get` method. If successful, the content is returned; otherwise, an exception is caught, and an error message is printed, indicating that the server is unreachable.

Overall, this application dynamically distributes incoming requests among servers in a round-robin fashion, considering each server's load factor for a balanced distribution of traffic. The rotation list and counter facilitate a sequential selection of servers, ensuring an equitable distribution of requests based on their respective loads.

Docker File:

```
vm6@et2599vm6:~$ cat Dockerfile
FROM python:3-slim
WORKDIR /app
ADD . /app
RUN pip install --trusted-host pypi.python.org Flask
RUN pip install psutil
RUN pip install requests
ENV NAME Kurt Tutschku
CMD ["python", "wrr.py"]
```

This Dockerfile configures a Python 3 slim image, setting the working directory to '/app' and copying local content. Dependencies for the Flask app—Flask, psutil, and requests—are installed. An environment variable 'NAME' is set to "Kurt Tutschku". The CMD instruction launches the Flask app specified in "wrr.py".

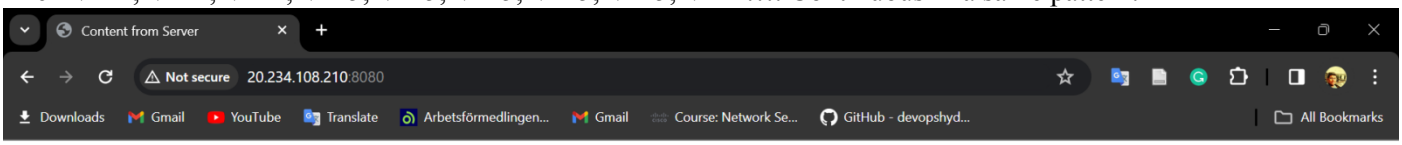
index.html:

```
vm6@et2599vm6:~/templates$ cat index.html
<!DOCTYPE html>
<html>
        <head>
                <title>Content from Server</title>

        </head>
        <body>
                <h1>Content Received from Server</h1>
                    <p><b>Chosen Server:</b> {{ chosen_server }}</p>
                        <div>
                                        <p>Content from Server:</p>
                                                <div>{{ content_from_server | safe }}</div>
                                                    </div>

        </body>
</html>
```

Results:

Since Loads Are set for 3 and 5 for VM4 and VM5 respectively.

Then VM4, VM4, VM4, VM5, VM5, VM5, VM5, VM5, VM4…. Continuous in a same pattern.



Content Received from Server

Chosen Server: Alpha

Content from Server:

Hello Kurt Tutschku!

Hostname: e87874605396

This is from VM4

Configuring the Load Balancer with Workload-Sensitive:

In a workload-sensitive load balancing algorithm, the distribution of incoming requests to web servers is intricately linked to the workload sensitivity of each server. Unlike the traditional round-robin approach, workload-sensitive algorithms adapt dynamically to the varying capacities and capabilities of individual servers based on their real-time workloads. To configure the load balancer with a workload-sensitive algorithm, the process begins by creating a new folder. Within this folder, a Dockerfile and a Python file should be established, following the code provided below. This tailored approach ensures that the load balancer intelligently directs traffic, considering the current workload of each web server and optimizing resource utilization.

Python code:

```
vm6@et2599vm6:~$ cat wls.py
from flask import Flask, render_template, request
import requests

app = Flask(__name__)

ip_server1 = "74.234.82.83:8080"  # Replace with your server 1 IP and port
ip_server2 = "20.238.114.112:8080"  # Replace with your server 2 IP and port

@app.route('/')
def index():
        return render_template('index1.html')

@app.route('/weight')
def output():
    load_server1 = float(requests.get("http://" + ip_server1 + "/load").content)
    load_server2 = float(requests.get("http://" + ip_server2 + "/load").content)

    if load_server1 <= load_server2:
        load_result = requests.get("http://" + ip_server1 + "/")
        chosen_server = "Server 1"
    else:
        load_result = requests.get("http://" + ip_server2 + "/")
        chosen_server = "Server 2"
    return render_template('result.html', content=load_result.content, load_server1=load_server1, load_server2=load_server2,
 chosen_server=chosen_server)
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=8080)
```

In this Flask application, designed for a simple load balancing scenario, two backend servers, denoted as Server 1 and Server 2, are configured with their respective IP addresses and ports. The application features two routes - the '/' route, rendering an HTML template ('index1.html'), and the '/weight' route, orchestrating a load balancing strategy. In the '/weight' route, the application dynamically calculates the load of each server by fetching the '/load' endpoint and comparing the results. The load balancing logic determines whether to select Server 1 or Server 2 based on their respective loads. If Server 1 has a lower or equal load compared to Server 2, a request is made to its root ('/') endpoint; otherwise, a request is directed to Server 2. The result is presented in an HTML template ('result.html'), displaying the content from the chosen server, the loads of both servers, and the identifier of the selected server. This implementation provides a straightforward yet effective demonstration of load balancing, ensuring that the server with the lower load handles incoming requests.

Docker file:

```
vm6@et2599vm6:~$ cat Dockerfile
FROM python:3-slim
WORKDIR /app
ADD . /app
RUN pip install --trusted-host pypi.python.org Flask
RUN pip install psutil
RUN pip install requests
ENV NAME Kurt Tutschku
CMD ["python", "wls.py"]
```

In this Dockerfile, based on Python 3 slim, the working directory is set to '/app', and local contents are added. Essential dependencies for the Flask app—Flask, psutil, and requests—are installed. An environmental variable 'NAME' is assigned. Notably, the CMD instruction is adjusted to ["python", "wls.py"], signifying the execution of the Flask application encapsulated in 'wls.py' upon container initiation.

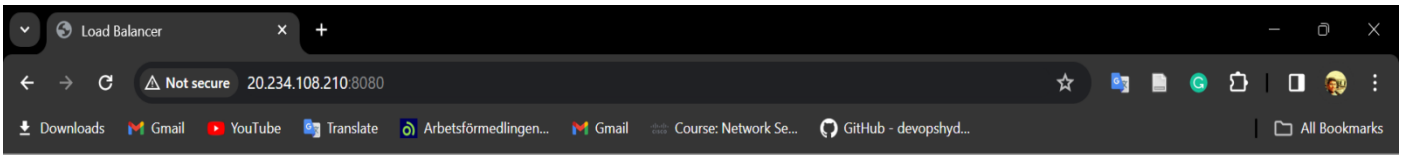
index1.html:

```
vm6@et2599vm6:~/templates$ cat index1.html
<!DOCTYPE html>
<html>
        <head>
                <title>Load Balancer</title>
        </head>
        <body>
                <h1>Welcome to Load Balancer</h1>
                    <p>Click <a href="/weight">here</a> to check load balancing.</p>
        </body>
</html>
```

result.html:

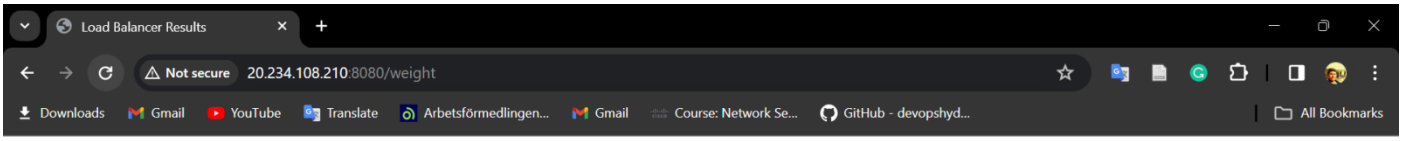
```
vm6@et2599vm6:~/templates$ cat result.html
<!DOCTYPE html>
<html>
<head>
    <title>Load Balancer Results</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 20px;
        }
        .content-container {
            border: 1px solid #ccc;
            padding: 10px;
            margin-bottom: 20px;
            white-space: pre-line;
        }
    </style>
</head>
<body>
        <h1>Load Balancer Results</h1>
        <p>Chosen Server: {{ chosen_server }}</p>
        <div class="content-container">
            <h2>Content from Chosen Server:</h2>
            <div>{{ content | safe }}</div>
        </div>
        <p>Load of Server 1: {{ load_server1 }}</p>
        <p>Load of Server 2: {{ load_server2 }}</p>
</body>
</html>
```

Result:



Welcome to Load Balancer

Click here to check load balancing.



Load Balancer Results

Chosen Server: Server 1

Content from Chosen Server:

b'

Hello Kurt Tutschku!

Hostname: e87874605396

This is from VM4

'

Load of Server 1: 0.3

Load of Server 2: 2.8