RED ACRE

TECHNICAL ASSIGNMENT

S. SUJIN NIHAR | 27TH OCTOBER 2021

TABLE OF CONTENTS

FOREWORD

To begin with, I would like to thank you for giving me the opportunity to show my motivation and eagerness to provide additional value as a DevOps Lead engineer in RedAcre.

In this document I'll showcase you how I would design an AWS/Docker/Kubernetes environment based on the provided requirements. In doing so I will try to not only repeat what I've learned/implemented in my experience but extend my knowledge by doing research on different approaches and log these attempts to give you a clear insight in my *modus operandi*

TASK 1 – DOCKERIZE THE APPLICATION

The first task is to dockerise this application - as part of this task you will have to get the application to work with Docker and Docker Compose. You can exposé the frontend using NGINX or HaProxy. The React container should also perform npm build every time it is built. Hint/Optional - Create 3 separate containers. 1 for the backend, 2nd for the proxy and 3rd for the react frontend. It is expected that you create another small document/walkthrough or readme which helps us understand your thinking process behind all the decisions you made. The only strict requirement is that the application should spin up with docker compose up --build command

DELIVERABLES

- `FRONTEND USING REACT APP
- BACKEND USING PYTHON FLASK
- PROXY SERVER -NGINX
- BUILD ONLY USING DOCKER-COMPOSE

APPROACH

Before I start writing even a single line of code, I want to create a clear overview of what is needed to complete this assignment and how the later questions could impact the design I make in the first question. There are 3 parts in this task: 1) Dockerize Python-Flask and 2) Dockerize React app 3) Spin up both the containers using Docker compose

To prevent myself from chasing a red herring and spending a lot of hours (paid by the customer) on a design I started collating the requirements and decided to begin Ubuntu machine as I am much familiar with linux/unix CLI environments

REQUIREMENT GATHERING:

- PYTHON VERSION (LATEST)
- DOCKER
- FLASK VERSION 1.1.2
- NPM FOR REACT
- NODE JS
- DOCKER COMPOSE 1.25.0

My first thoughts are to create 3 different containers, one for the simple backend python-flask, other for react app and later focus on proxy. Then creating some sort of spinning process assisted by docker-compose. But before being the hammer and that only sees nails, I want to verify whether all the images are present in docker hub to make applications containerized only from a secure environment.

GOOGLING "DOCKER IMAGES FOR RESPECTIVE REQUIREMENT" QUICKLY BROUGHT ME TO THE FOLLOWING WEBSITES: HTTPS://HUB.DOCKER.COM/ WHERE I LOGGED IN & VERIFIED MY EXISTING DOCKER ACCOUNT

Implementing a quick driven solution could force us straight into the pickle and considering

Security is a major challenge when running applications in a virtual environment. Therefore, securing docker containers is vital. It requires securing everywhere from the host to the network.

Docker containers lay out a safer environment for your systems than traditional virtual machines. They provide a way to split up applications into smaller components.

The components are isolated from one another, thus helping to reduce attacks.

Quickly installed docker on my ubuntu machine and ensured running containers as a non-root which helps to mitigate security vulnerabilities. Running your containers on rootless mode will verify that your application environment is safe.

```
$ sudo usermod -aG docker $USER
```

Started jotting down Dockerfile for python-flask, will not mention here the commands used or how I have written as you can verify them in github repo.

https://github.com/Sujin451/RedAcre.git

However, while writing down the dockerfiles certain measures are kept in mind that downloading container images from untrusted sources and vendors can introduce security vulnerabilities in containers. Make sure that images downloaded from online platforms are from trusted and secure sources.

As mentioned during the beginning of my approach, To avoid security vulnerabilities:

- Used container images that are authentic. Docker Hub. As it is the largest Docker registry with multiple container images.
- Made use of images that are verified by the Docker Content Trust.

Below is the directory structure which I have used for python flask backend, Quickly added the requirements like flask, cors as given in app.py python script.

Created an image- docker build -t pythonapp

```
flask-backend      latest    45cc2caa92dd
```

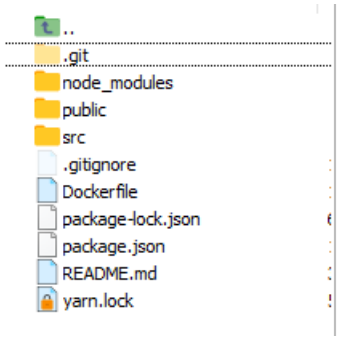
Added to a container for test purpose

```
├── Dockerfile
├── app.py
└── requirements.txt
```

Checked the output in the browser -boom it worked as expected as written in the Dockerfile


```
{
  "cpu": 2.7,
  "ram": 58.8
}
```


Similarly, created directory structure for React front end as below and written



Spined up react backend container image and added to the container.

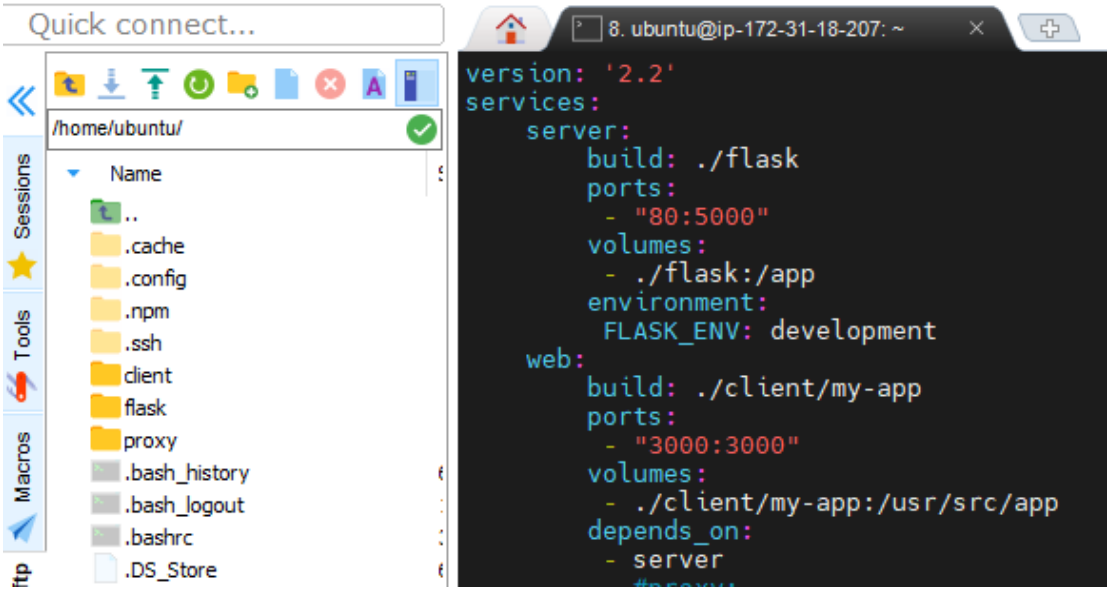
```
react-frontend     latest    db8c86839489
```

However, the result dint come up as expected as npm install did not run properly, so tried to verify docker logs container and re-ran it by aligning the indentation and commands

Also copied the given content by the RedAcre and added to my React-backend folder structure. Boom it worked like charm!

However **STRICT requirement** is to spin up using DOCKER-COMPOSE.

Hence started drafting docker-compose.yml. Thankfully as the frontend and backend were working like expected I had to just combine both the services together in docker compose file so that containers along with images will be created automatically



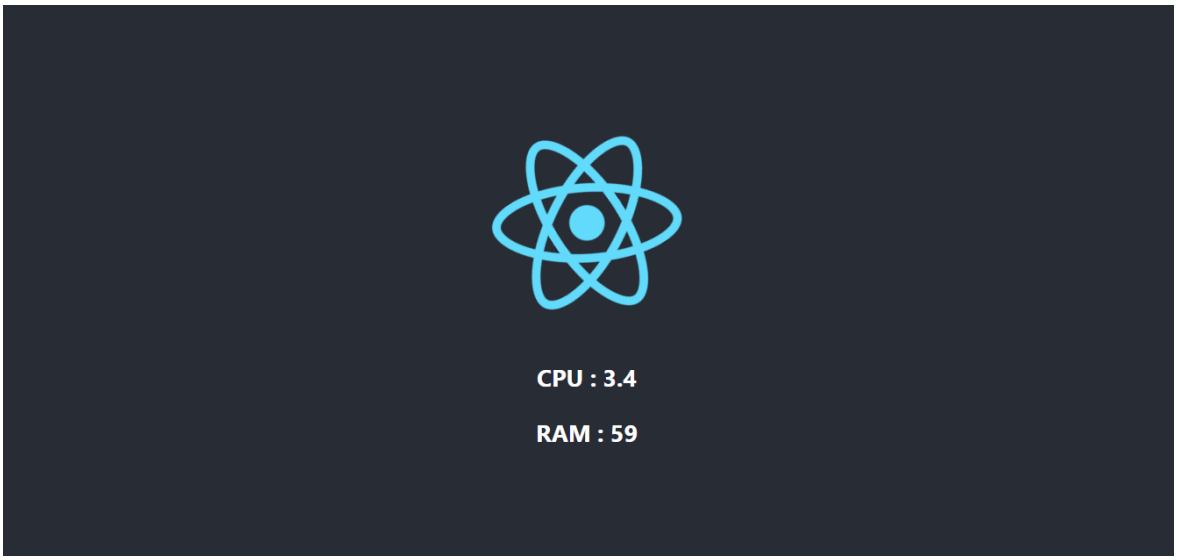
We can further grain down to specific docker-network solely for our container, however, didn't want to convolute the process.

Post verifying the docker-compose file, realized that no network has been setup specifically for it, however issued docker-compose to verify

$ docker-compose up -→ Gave the below output.

```
Creating ubuntu_server_1 ... done
Creating ubuntu_web_1    ... done
Attaching to ubuntu_server_1, ubuntu_web_1
server_1  |  * Serving Flask app "app" (lazy loading)
server_1  |  * Environment: development
server_1  |  * Debug mode: on
server_1  |  * Running on all addresses.
server_1  |    WARNING: This is a development server. Do not use it in a production deployment.
server_1  |  * Running on http://172.18.0.2:5000/ (Press CTRL+C to quit)
server_1  |  * Restarting with stat
server_1  |  * Debugger is active!
server_1  |  * Debugger PIN: 126-904-389
web_1     |
web_1     | > sys-stats@0.1.0 start /usr/src/app
web_1     | > react-scripts start
web_1     |
web_1     | [wds]: Project is running at http://172.18.0.3/
web_1     | [wds]: webpack output is served from
web_1     | [wds]: Content not from webpack is served from /usr/src/app/public
web_1     | [wds]: 404s will fallback to /
web_1     | Starting the development server...
web_1     |
web_1     | Compiled successfully!
web_1     |
web_1     | You can now view sys-stats in the browser.
web_1     |
web_1     |   Local:            http://localhost:3000
web_1     |   On Your Network:  http://172.18.0.3:3000
web_1     |
web_1     | Note that the development build is not optimized.
web_1     | To create a production build, use yarn build.
web_1     |
```

Post successful run of docker-compose verified if containers are created and immediately checked in browser for the output. **Hurray……!** Got the expected result



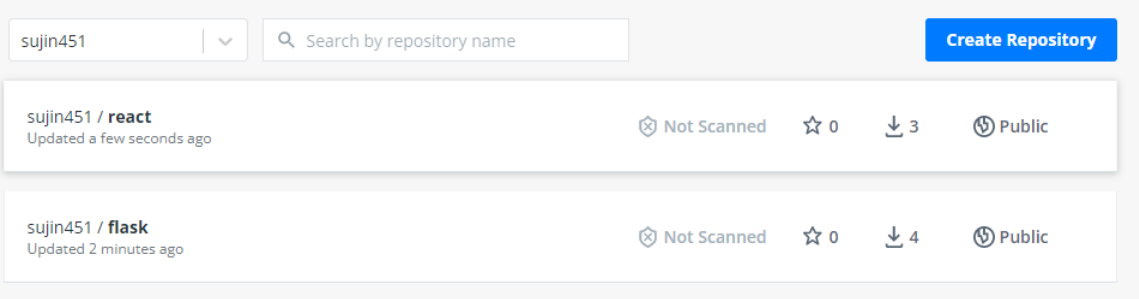
Now time to save the IMAGES and push to docker hub repo. From the below snippet sujin451 is my

username of dockerhub /react is the repo and tag as updated

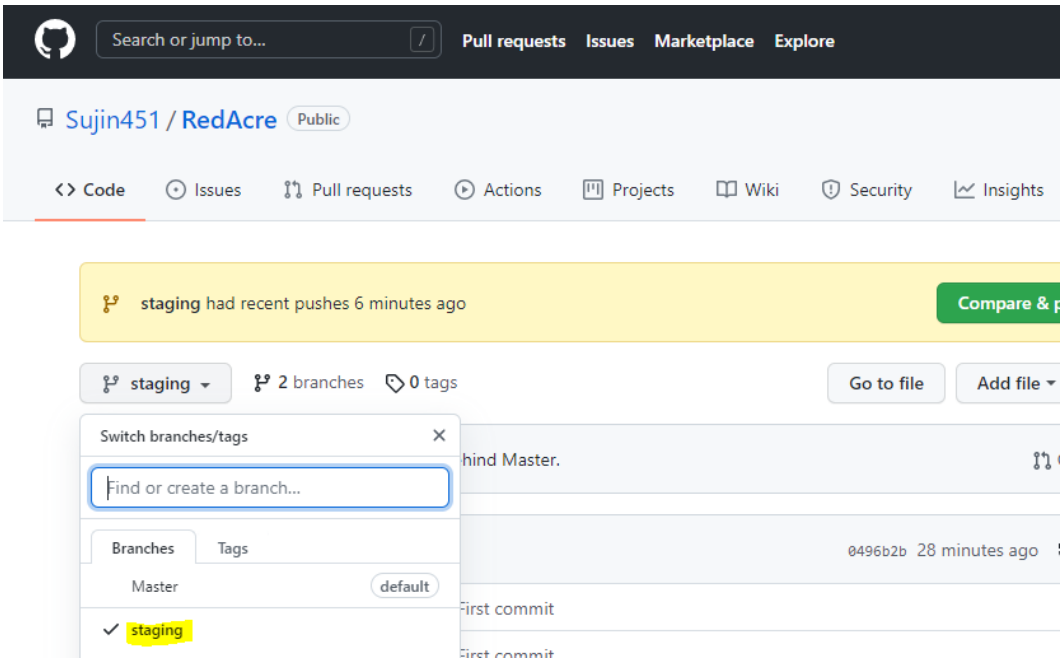
```
ubuntu@ip-172-31-18-207:~$ docker commit 3cd0b96c424b sujin451/react:updated
sha256:227962f17911edf960d6ad7374b1ab13d0225b90e12855a20f151759d7299e7c
```

```
ubuntu@ip-172-31-18-207:~$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
sujin451/react      updated   227962f17911   10 seconds ago   582MB
```

Similarly created 1 for python-flask backend as well and push to my docker hub repo. Ideally, we use organization repository to place the images and save them in private



Also, as the requirement is not to keep in master branch – created new branch ‘Staging’ and pushed remotely in GITHUB



Moving on to second task, to host the application on cloud.

TASK 2 – DEPLOY ON CLOUD

Deploy on Cloud Next step is to deploy this application to absolutely any cloud of your choice. It's important to remember here that the application is already containerize, maybe you could deploy it to services which take an advantage of that fact. (example, AWS EBS or ECS?) You could use any other cloud service provider of your choice too. Use the smallest instance size available to save up on the cloud costs. The React App should be accessible on a public URL, that's the only hard requirement. Use the best practices for exposing the cloud VM to the internet, block access to unused ports, add a static IP (elastic IP for AWS), create proper IAM users and configure the app exactly how you would in production. Write a small document to explain your approach. Usage of tools such as terraform or ansible will be appreciated

DELIVERABLES

- APPLICATION (DOCKERIZED) TO BE HOSTED ON CLOUD
- REACT APP SHOULD BE ACCESSIBLE ON A PUBLIC URL
- FIXED IP

APPROACH

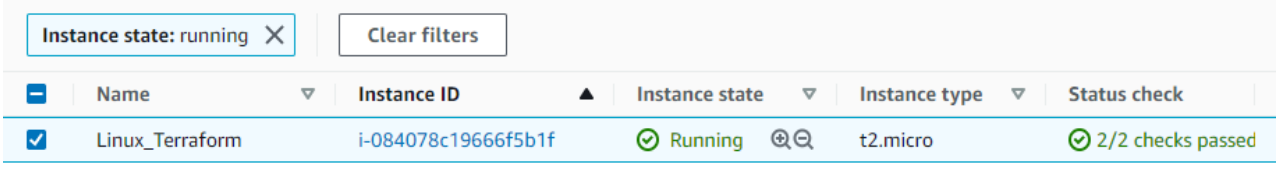
After giving it some thought I've decided to start off with the qualities required by the customer. These

are clearly defined in the assignment. The environment needs to have the following qualities:

- Accessed by public
- Can be Cost effective- low instance type -t2 micro
- Should have elastic IP

One line which attracted me the most is usage of Ansible or Terraform will be appreciated- As I like automating the process, I have decided to use Infrastructure as a code tool – Terraform.

Hence without much due I quickly procured Linux server and installed Terraform in it as it is a simple & easy to install.



Verified if it installed properly by ssh into instance and as per below snippet

```
[root@ip-172-31-1-89 terraform]# terraform -v
Terraform v0.15.5
on linux_amd64
```

REQUIREMENT GATHERING:

IN REAL-TIME SCENARIO WE NEED TO BE MORE INQUISITIVE ON THE REQUIREMENTS WITH THE CLIENT OR THE VENDOR TO FOCUS ON PERFORMANCE /CAPACITY/AVAILABILITY MANAGEMENT.

- ELASTIC IP – IN ORDER TO HAVE STATIC IP SO THAT IT DOESN'T CHANGE WHENEVER SERVER REBOOTS
- VPC
- SUBNET
- IGW
- IAM USERS/GROUPS
- SECURITY GROUP
- KEY PAIR
- IAM USER FOR TERRAFORM

Primary focus was to create IAM Groups & users. Hence created like below:

Roles & Responsibilities:

L1 Team – Cloud Engineering – Monitoring – Read only access

L2 Team- Cloud Admin – SME -Full access-specific services

L3 Team – Cloud Architect – AWS infra – End to end

Based on the above usability, we create groups and users accordingly.

Also, we attach custom policies, managed policies, permission boundary and inline policies as per the requirement by creating simple json syntax policy. In real time we use AWS organizations to create OU's organizations units like production/development/ staging and add service control policies. For time being I did not utilize it as primary focus was to deploy app on cloud environment.

DESIGN

After reviewing the current specifications and requirements I've concluded to start off with a basic design in two availability zones in the eu-west-1 region. The AWS CLI tells me there are three AZ's in this region so depending on the number of application servers we're going to deploy; we have the possibility to expand to three AZ's.

$aws ec2 describe-availability-zones --region eu-west-1

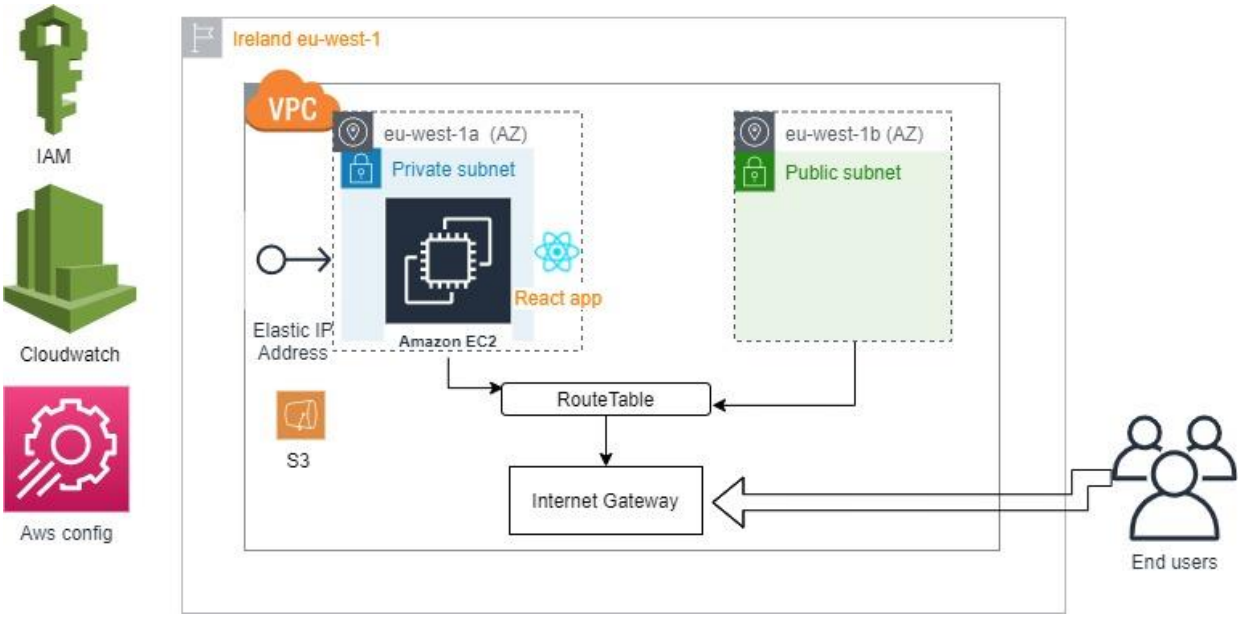
A contributing factor is to choose this region is that it offers all major AWS services we need for this assignment. I've read this article prior to this assignment and bookmarked it:

https://www.concurrencylabs.com/blog/choose-your-aws-region-wisely/ - Ireland looks to be the leader of the pack regarding the introduction of new services. Let's stick with **Ireland** first

In a real-life situation I would consult the customer on all requirements and advice on the impact it has on costs. I would most definitely talk to the third-party developer to better understand their current way of working and how our design would fit in their needs. If we can save them many hours of manual deployments, this indirectly would be in the advantage of the customer and could fuel an intensification of the partnership with this third party.

Let's assume, since we're talking about a fictive customer, the customer wants the website to be highly available but isn't willing to spend the money on a fault-tolerant environment. Disaster Recovery is a nice to have, but not something we should focus on. This customer always wants his/her website to be fast and available (just not when disaster strikes). Lacking any of both could result in some serious SEO penalties and subsequent lost revenue. DDoS mitigation is a serious concern as competition is fierce and our customer is afraid to fall victim to other companies with nefarious intents as I mentioned earlier I am not focusing on RTO or RPO's here however depending the customer SOW.

Quickly made this architect design from draw.io – Also saved the io file and image in github repo



SKELETON

I've decided to use the two AZ's in eu-west-1. A CDN for DDoS mitigation and caching of static content can be added later- In real life we use AWS config and AWS WAF services anyway due to time constraints, let's focus on what's inside the VPC first.

To start off with I was thinking if we need a load balancer. AWS offers three kinds of load balancers, classic, application and network ELBs. However as we are not using any active-passive or active -active I've not chosen for the ELB even though it offers some nice features like the integration with WAF (which could be useful when the customer grows), the ability for path-based routing (which we e.g. could use for the admin server), the integration with ECS (which we might use now or in the future) and the ability to use the unique trace identifier in debugging performance issues.

So, before we even start with the big picture and the rest, let's start with a Mobaxterm (I personally recommend using this rather than putty) so we can SSH into any EC2 instance and see what's going on. The VPC has to be created and IGW needs to attached.
Nothing much to play with the VPC peering as I see one host in one vpc and not much to play with route tables.
Of course, we can still use VPC peering if the application has different domains hosted in other region but our assessment I didn't think it was necessary atleast at this juncture.

We need to select an AMI for the application to be hosted – I preferred ubuntu ami. so let's use an AMI of the shelf. Obviously, we first need to create a keypair to be able to login to it. (Safely keep the key pairs-private/public) Ideal scenario only we use vaults .Later needs to create a security group and subnet

to launch the EC2 instance and add EIP to it to prevent the loss of IP when it is rebooted. Of course, we need to add a security group and NACL to secure it from the base up.

For this assessment I've skipped rewriting this part of the template considering I've spent countless hours creating the exact same skeleton template. I do want to show the reader how I would evaluate every component to match it with the set- out requirements. We now have a highly-available secure entry point.

Hence in order to avoid the manual longtime efforts in creating these in console, I preferred using TERRAFORM

CONSIDERATION

In real time I would present this design to the customer and the third-party developer. If all parties consent, and only then, we start building the environment. I need to make the templates as re-usable as possible, so we only have to maintain one template. It assists in maximizing environment parity as well which will save the customer and RedAcre a lot of unplanned work and time on debugging

SCOPE OF IMPROVEMENTS - AUTO-SCALING

The promise of auto-scaling is that you can quickly and automatically react when a VM gets corrupted or there's a sudden spike in visitors. I plan on leveraging target tracking with the ALB and tweak the settings after the migration. We shouldn't forget the AMI has to download the whole application so every second counts!

Depending on the application s criticality and SLA we can implement Autoscaling /ELB additionally

LOGS AND BACKUP

Logging is a key component in this setup. EC2 instances are to be regarded ephemeral so all logging needs to be stored in one highly available data store. For this we can use CloudWatch. The CloudWatch agent will run on the application servers and send the system logs as well as the application logs to CloudWatch. The logs can then be stored in an S3 bucket which resided in a separate account, to ensure the integrity of the log files.

Backups of the database are performed daily and can be retained between 0 and 35 days. In that time period daily snapshots are complemented with transaction logs, allowing us to restore the database to the second within the given retention period. After that period, we will automatically store the daily backups on S3 and install a lifecycle policy. The retention periods are set to the requirements of the customer.

The S3 bucket is redundant within the region and offers 9 nines durability. If required, we can configure versioning and cross-region replication.

THE CUSTOMERS TIME IS VALUABLE, MINE IS PRICELESS

Hence without much due instead of creating manually the instances I have jotted down the simple Terraform .tf file and initialized it. Ideally as a best practice, we created separate modules for each in order to segregate VPC.tf /Subnet.tf however due to time constraint and it is only 1 minor environment, created a simple directory structure.

Note : Hardcoded currently with the default inputs- However we can also create Variables.tf so that values can be inputted by the other team members whenever they issue terraform plan or apply commands.

Below is the snippet of the terraform VPC file which I have created:

```
#This terraform file is to spin up the VPC for React app to be hosted on Cloud

#Provider - AWS cloud

provider "aws"  {
        region="eu-west-1"
}

#VPC Creation
# Added /28 to use only 16 ip addrresses to prevent ip exhaustion we are hardly using few  EC2 instances
# Used Class C ip address

resource "aws_vpc" "React_VPC" {
  cidr_block       = "192.168.0.0/28"
  instance_tenancy = "default"

  tags = {
    Name = "React_VPC"
  }
}

#Subnet 1 creation
#Can procure  multiple subnets however as only we exposing it to public lets consider this as public subnet
# Always good practice is create private/public subnets seperately
# 8 ip addresses only will be used maximum as we have defined /29 notation
resource "aws_subnet" "React_subnet" {
        vpc_id=aws_vpc.React_VPC.id
        cidr_block="192.168.0.0/29"

        tags={
         Name="React_Subnet"
        }
}

#Internet Gateway
# NO need to manually attach IGW as it would automatically attached to VPC defined
resource "aws_internet_gateway" "React_IGW" {
```

Before I jump into deep dive explaining the above tf file , I have added comments in brief so elucidating them again here is repetitive however to sum it up , as mentioned in the design section , all the requirements especially related to the VPC/Networking have been taken care in the tf file. Of course we can even grain them in minor enhancements like availability-zones or security group etc. However, I used all the mandatory elements only required to spin up the VPC. Also created a separate terraform file to spin up EC2 and attach EIP to it

Basic commands used:

Terraform init

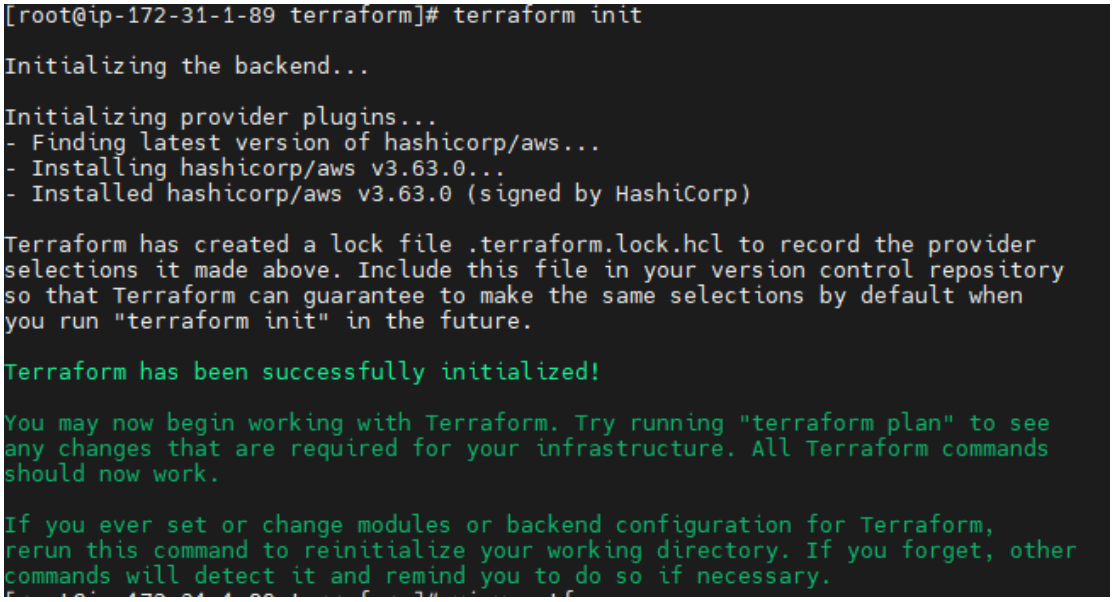
Terraform plan

Terraform apply

Of course, in real time we take care of below aspects too:

- Manage S3 backend for tfstate files.
- Turn on debug when you need do troubleshooting.
- Minimum AWS permissions necessary for a Terraform run
- Usage of remote ends to retrieve meta data
- Enable version control on terraform state files bucket

In order to expedite the things, I started initializing the terraform files



Alas! Could see the below error while issuing plan command, quickly realized and logged in using aws configure which prompted for Access Key ID and secret key, for which I created a separate IAM user for Terraform user and saved the credentials safely in Vault

```
[root@ip-172-31-1-89 terraform]# terraform plan

Error: error configuring Terraform AWS Provider: no valid credential sources for Terraform AWS Provider found.

Please see https://registry.terraform.io/providers/hashicorp/aws
for more information about providing credentials.

Error: NoCredentialProviders: no valid providers in chain
caused by: EnvAccessKeyNotFound: failed to find credentials in the environment.
SharedCredsLoad: failed to load profile, .
EC2RoleRequestError: no EC2 instance role found
caused by: EC2MetadataError: failed to make EC2Metadata request
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
  <title>404 - Not Found</title>
 </head>
 <body>
  <h1>404 - Not Found</h1>
 </body>
```

Post fixing up the IAM role, I reinitiated the commands terraform init and terraform plan & apply

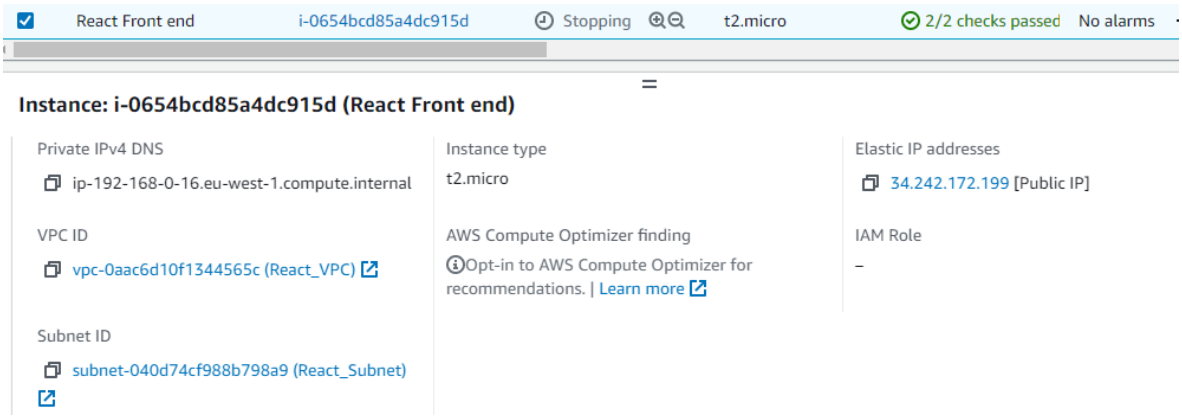
Bingooo….!!! Everything created at once

```
  Enter a value: yes

aws_vpc.React_VPC: Creating...
aws_vpc.React_VPC: Creation complete after 1s [id=vpc-0aac6d10f1344565c]
aws_subnet.React_subnet: Creating...
aws_internet_gateway.React_IGW: Creating...
aws_internet_gateway.React_IGW: Creation complete after 0s [id=igw-0492a360a6c586ef1]
aws_subnet.React_subnet: Creation complete after 0s [id=subnet-040d74cf988b798a9]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

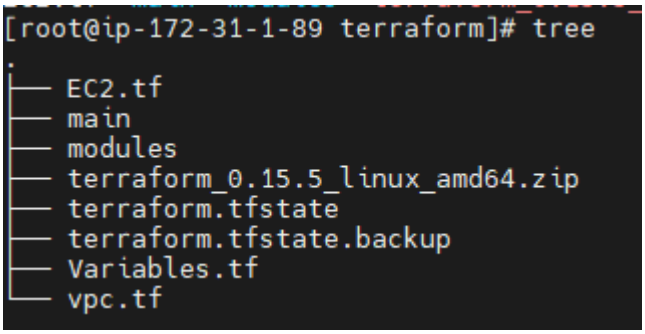
Created separate EC2.tf terraform file and executed it. I will share in git repo all the terraform files created

```
aws_instance.React_Stats: Creating...
aws_eip.React_EIP: Creating...
aws_eip.React_EIP: Creation complete after 0s [id=eipalloc-032f1f804ec262535]
aws_instance.React_Stats: Still creating... [10s elapsed]
aws_instance.React_Stats: Still creating... [20s elapsed]
aws_instance.React_Stats: Still creating... [30s elapsed]
aws_instance.React_Stats: Creation complete after 31s [id=i-0654bcd85a4dc915d]
aws_eip_association.eip_assoc: Creating...
aws_eip_association.eip_assoc: Creation complete after 1s [id=eipassoc-0568b43e8cc509cf0]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

Let’s check the console



Final directory structure

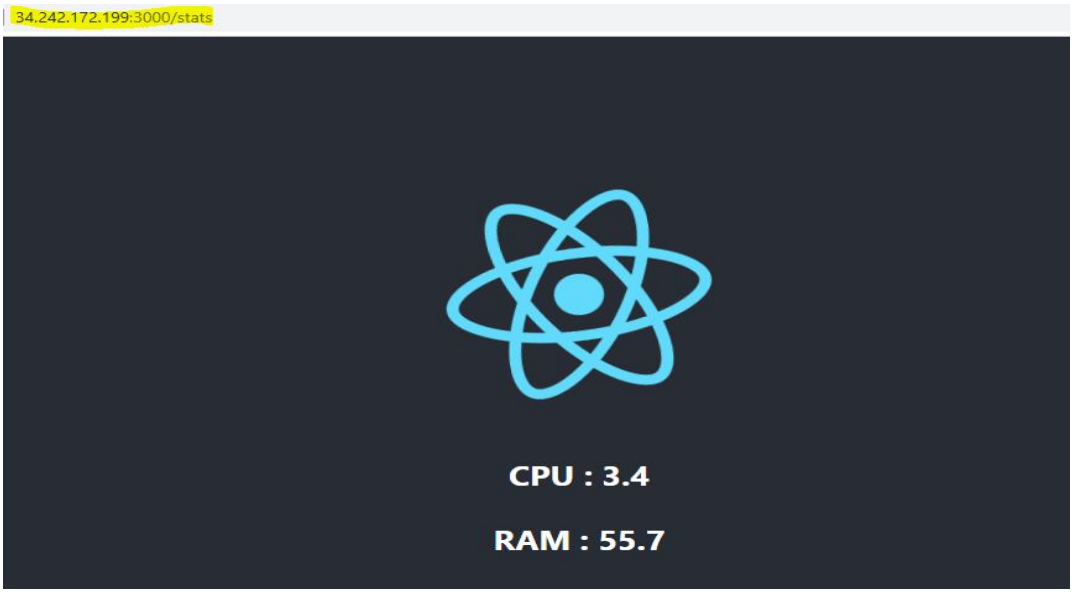


Exposed ports: only 80 for http and ssh 22 in Security Group, Realtime scenario https (443) and SSL certifications will be procured from certification manager and installed too

Elastic IP has been associated – 34.242.172.199

Now type the URL in the browser: http://34.242.172.199:3000/stats

REACT APP is hosted on AWS CLOUD….!!! TASK 2 completed. Below is the screenshot 😊



TASK 3 – GET IT TO WORK ON KUBERNETES

Next step is separate from step 2. Go back to the application you built in Stage 1 and get it to work with Kubernetes. Separate out the two containers into separate pods, communicate between the two containers, add a load balancer (or equivalent), expose the final App over port 80 to the final user (and any other tertiary tasks you might need to do) Add all the deployments, services and volume (if any) yaml files in the repo. The only hard-requirement is to get the app to work with minikube

DELIVERABLES

- Load balancer
- Enable version control on terraform state files bucket
- Expose the final app over :80 (http) port
- Minikube

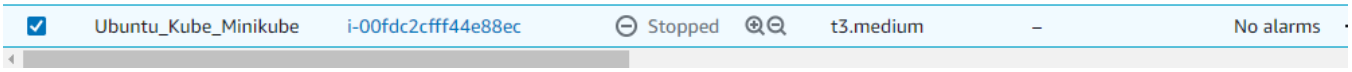
APPROACH

As partial work is already done, I just need to push the images to docker repo so the pods /containers can be easily procured via deployment.yaml files. The environment needs to be

- Accessed by public

- 2 pods in 2 containers
- Load balancer

As my laptop is my client laptop- I cannot use or install minikube due to strict organizational policies, I have procured an new EC2 instance (ubuntu) , installed minikube in it. Minimum 2 Vcpus are required hence used t3.medium.



Installed minikube as well, wouldn't mention the process of installing it as it is not the key concern here.

REQUIREMENTS GATHERING:

- Ubuntu EC2 instance
- minikube
- Kuberenetes/kubectl
- SG – 80/22 ports needs to be opened
- Key pair
- Namespace
- Deployment/services/ingress yml. In real time we use Helm charts, however the requirement is minimal hence preferred to jot down yml files

FULFILLING REQUIREMENTS

Installed minikube/kubectl and started them:

```
ubuntu@ip-172-31-43-86:~$ minikube version
minikube version: v1.23.2
commit: 0a0ad764652082477c00d51d2475284b5d39ceed
```

Started minikube

```
root@ip-172-31-43-86:~# minikube start --vm-driver=none
* minikube v1.23.2 on Ubuntu 18.04
* Using the none driver based on user configuration
* Starting control plane node minikube in cluster minikube
* Running on localhost (CPUs=2, Memory=3876MB, Disk=15817MB) ...
* OS release is Ubuntu 18.04.5 LTS
* Preparing Kubernetes v1.22.2 on Docker 20.10.7 ...
  - kubelet.resolv-conf=/run/systemd/resolve/resolv.conf
    > kubectl.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubelet.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubeadm.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubectl: 44.73 MiB / 44.73 MiB [-----------] 100.00% 642.25 MiB p/s 300ms
    > kubeadm: 43.71 MiB / 43.71 MiB [-----------] 100.00% 159.59 MiB p/s 500ms
    > kubelet: 146.25 MiB / 146.25 MiB [---------] 100.00% 218.99 MiB p/s 900ms
  - Generating certificates and keys ...
  - Booting up control plane ...
  - Configuring RBAC rules ...
* Configuring local host environment ...
*
! The 'none' driver is designed for experts who need to integrate with an existing VM
* Most users should use the newer 'docker' driver instead, which does not require root!
* For more information, see: https://minikube.sigs.k8s.io/docs/reference/drivers/none/
*
! kubectl and minikube configuration will be stored in /root
! To use kubectl or minikube commands as your own user, you may need to relocate them. F
*
  - sudo mv /root/.kube /root/.minikube $HOME
```

Verified if cluster is created.

```
root@ip-172-31-43-86:~# kubectl cluster-info
Kubernetes control plane is running at https://172.31.43.86:8443
CoreDNS is running at https://172.31.43.86:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

In order to have good practice - Created a separate namespace so that our project work can kept under it and for the access purpose of the team members we typically create namespace

```
root@ip-172-31-43-86:~# kubectl create namespace sujin
namespace/sujin created
```

Working as expected. Lets spin up pods/containers accordingly

```
root@ip-172-31-43-86:~# kubectl get pods --namespace sujin
No resources found in sujin namespace.
```

As time is running up, quickly started drafted deployment.yml files. Later thought let me just do a dry-run so that we can yml format automatically. Easy peasy

Kubectl create pods –dry-run=client -o deployment.yaml

Kubectl apply -f deployment.yaml.

Quicker methods leads to Quicker solutions too !!

Created two separate pods from our images which we created earlier for React/Flask front end/backend application and created separate container using yml files.I will share them in repo

Ofcoures there are various ways of doing this be it helm charts or jotting them manually, but I am

going for this solution.

Another important file in order to authenticate login is ‘Secrets’ which as of now I created it separately but real time scenarios it is either used as interpolation function or stored in vaults.
Many methods are revolving in my mind at this juncture.

However, moving on to check the ip address of minikube

```
root@ip-172-31-43-86:~# minikube ip
172.31.43.86
```

Eventually wanted to expose this ip to port 80 to get the application running
Below is the snippet how I started off writing down the script for spinning up 2 different pods.
We don't specify a hostPort for a Pod unless it is absolutely necessary. When you bind a Pod to a hostPort, it limits the number of places the Pod can be scheduled, because each <hostIP, hostPort, protocol> combination must be unique. If you don't specify the hostIP and protocol explicitly, Kubernetes will use 0.0.0.0 as the default hostIP and TCP as the default protocol.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    name: React
spec:
    containers:
      - name: react-frontend #Naming convention very imp (should be lowercase only)
        image: sujin451/react:updated
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 3000
      - name: flask-frontend
        image: sujin451/flask:updated
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 8000
~
```

Similarly created another pod- Pod2, now the focus is on to connect the port, where my mind is focusing on creating a service -Loadbalancer , of course by default there is ClusterIP which is enabled by default in Kubernetes however the connection between pods needs to be established.

```
root@ip-172-31-43-86:~/kubernetes# kubectl apply -f pod.yaml --namespace sujin
```

Created resource under namespace Sujin

```
root@ip-172-31-43-86:~/kubernetes# kubectl get pods --namespace sujin
NAME    READY   STATUS    RESTARTS   AGE
pod1    2/2     Running   0          4m4s
pod2    2/2     Running   0          61s
```

Similarly created deployments and services.yml which will be uploaded in repo later, as time is running out, I prefer to focus on executing the output rather than taking individual screenshots.
However now realized instead of creating pods.yml I have created deployments itself separately this time along with replica sets
My bad I was overthinking perhaps…Never mind, let me quickly create new deployment.yml.
Deleted the pod1 and pod2

```
root@ip-172-31-43-86:~/kubernetes# kubectl get pods --namespace sujin
NAME                                READY   STATUS    RESTARTS   AGE
pod1                                2/2     Running   0          49m
pod2                                2/2     Running   0          45m
react-deployment-5b4c849b7-djvpk    2/2     Running   0          3m55s
react-deployment-5b4c849b7-tv5bc    2/2     Running   0          3m55s
root@ip-172-31-43-86:~/kubernetes# kubectl get deployments --namespace sujin
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
react-deployment   2/2     2            2           4m30s
```

Resource Quota will be added too in the deployment.yml files to prevent the exhaustion of the resources

```
root@ip-172-31-43-86:~/kubernetes# kubectl describe deployments --namespace sujin
Name:                   react-deployment
Namespace:              sujin
CreationTimestamp:      Thu, 04 Nov 2021 19:26:53 +0000
Labels:                 app=react
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=react
Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=react
  Containers:
   react-sys-stats:
    Image:        sujin451/react:updated
    Port:         3000/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
   flask-sys-stats:
    Image:        sujin451/flask:updated
    Port:         8000/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   react-deployment-5b4c849b7 (2/2 replicas created)
Events:
```

Created new pods/deployments 😊

```
root@ip-172-31-43-86:~/kubernetes# kubectl get pods -o wide --namespace sujin
NAME                                READY   STATUS    RESTARTS   AGE   IP           NODE
react-deployment-5b4c849b7-djvpk    2/2     Running   0          11m   172.17.0.6   ip-172-31-43-86
react-deployment-5b4c849b7-tv5bc    2/2     Running   0          11m   172.17.0.5   ip-172-31-43-86
```

Verified if replicasets are created or not

```
root@ip-172-31-43-86:~/kubernetes# kubectl get rs --namespace sujin
NAME                          DESIRED   CURRENT   READY   AGE
react-deployment-5b4c849b7    2         2         2       17m
```

Next part is on communication establishment between the pods

A Pod can communicate with another Pod by directly addressing its IP address, but the recommended way is to use Services. A Service is a set of Pods, which can be reached by a single, fixed DNS name or IP address. In reality, most applications on Kubernetes use Services as a way to communicate with each other.
Added the extension to the deployment yaml files & added a simple service

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: react-service
spec:
  selector:
    app: react
  ports:
    - port: 8000
      targetPort: 80
  type: LoadBalancer
```

Created service & utilized loadbalancer type

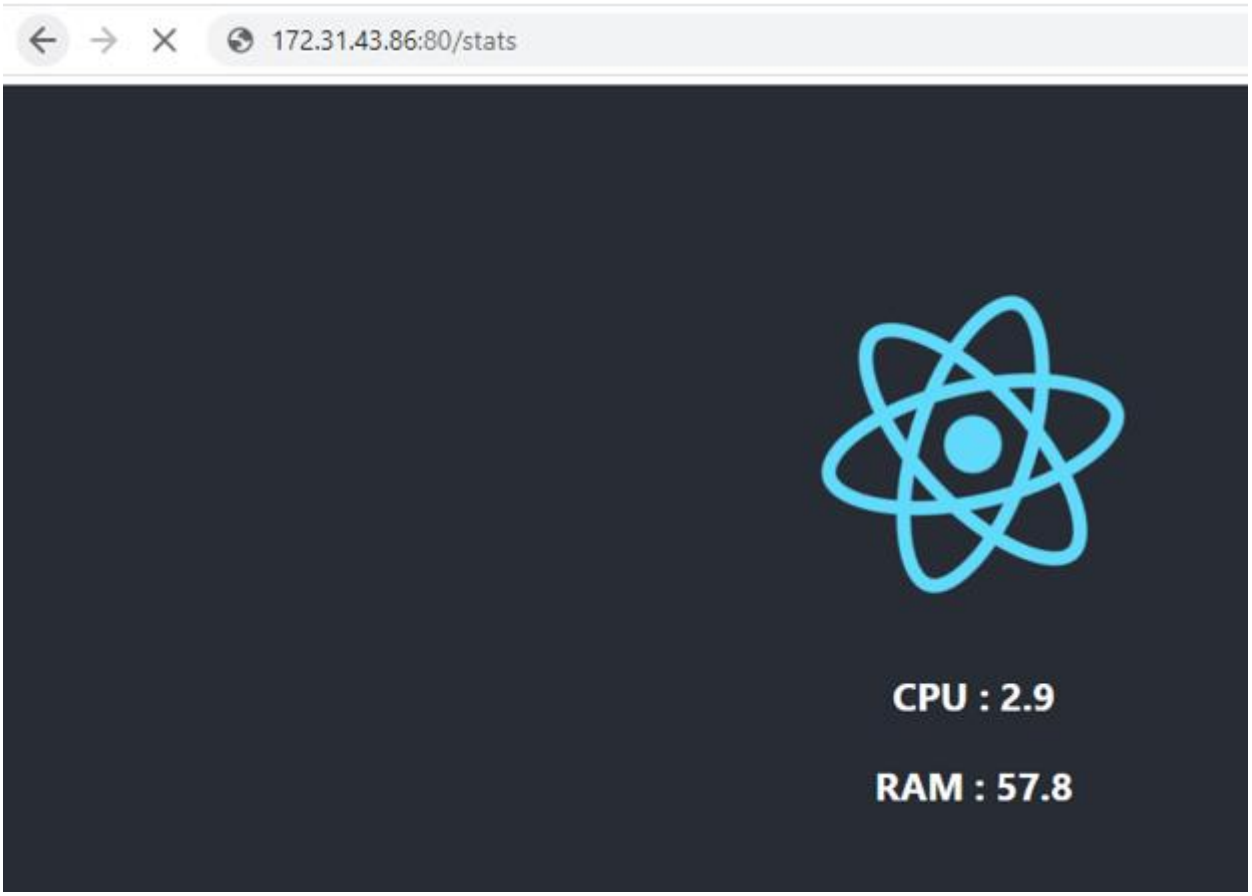
```
root@ip-172-31-43-86:~/kubernetes# kubectl get services --namespace sujin
NAME               TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
react-deployment   LoadBalancer   10.103.71.196   <pending>     80:31574/TCP   22s
```

Later added Ingress rule and verified if routing aspect can be utilized and external ip as minikube ip

Deploying an application can be done in many ways. This basic application sys-stats comes with its own particulars.we need to decide how we want to deploy. Canary? Blue-green? Or a rolling deployment? Or Jenkins. There are many options and we luckily use this simple command rollout.

Upon kubectl rollout and few modifications like exposing port 80 and setting target ports.

HURRAY…..!! Expose the minikube ip address and got the expected result



Definitely, there are many ways to get this application hosted in more secure way like using se readinessProbe & livenessProbes or node affinities or Deploy RBAC for security.

Like they say more power more responsibilities, similarly more application complexities more complaince rules needs to be taken care of.

LAST WORDS

I've had a great time working on this assessment and I'm looking forward to more riddles to solve. Thank you for giving me the opportunity to work on this assessment and I'm looking forward to hearing your feedback!

HAPPY to show you the demo/walk through the application spinned up and hosted on Docker/AWS/Kubernetes environments

Kind regards
Sujin Nihar
+31-682655056
sujin.nihar451@gmail.com