Assignment 2 - A simple service the Docker way

The task is to extend the simple web service from assignment one. This time you are going to use containers instead of virtual machine images for deploying your applications to the cloud (the containers, however, will still run in virtual machines.). Instead of configuring your image by snapshotting it or installing packages via Ansible, you are going to use Docker. Essentially, this means creating a Dockerfile that explains how to package your application and all its dependencies into a Docker image instead. In contrast to OpenStack Images, Docker images can be deployed exactly the same way on your own computer or on any Linux machine (virtual or otherwise) running in the cloud. There are number of great tutorials that will get you started. I suggest you start here.

Once you have successfully created a Docker image locally and tried to run it in a Docker container, you should push it to our own Docker registry that is hosted at https://gitlab.datahub.erdc.net. Start by signing up for an account and then you can find info on the container registry functionality here. All GitLab projects have their own Docker registries provided by GitLab. You need to authenticate to both push and pull Docker images. Pulling should be made using a Deploy Key, so that you do not need to make your personal password known on the server.

Once you've got a grip on Docker, it's time to design your application. We will build upon the visitor counter service from last time, but this time you will use OpenStack block storage (Volumes) to save the persistent storage. Instead of storing your data in a Swift storage container, you should setup a Redis database and make sure to configure it to put its files on an OpenStack Volume you create and attach to the database server instance. Info on Redis is found here.

One of the key strengths of Docker is that the community provides and supports Docker images via Docker Hub. You should use the official redis image for your database. Integration with Docker Hub is very easy, and trivially running the redis database is as easy as the following (note, though, that it does not specify where data will be stored, so this would not be sufficient for this assignment):

```
docker run -d redis
```

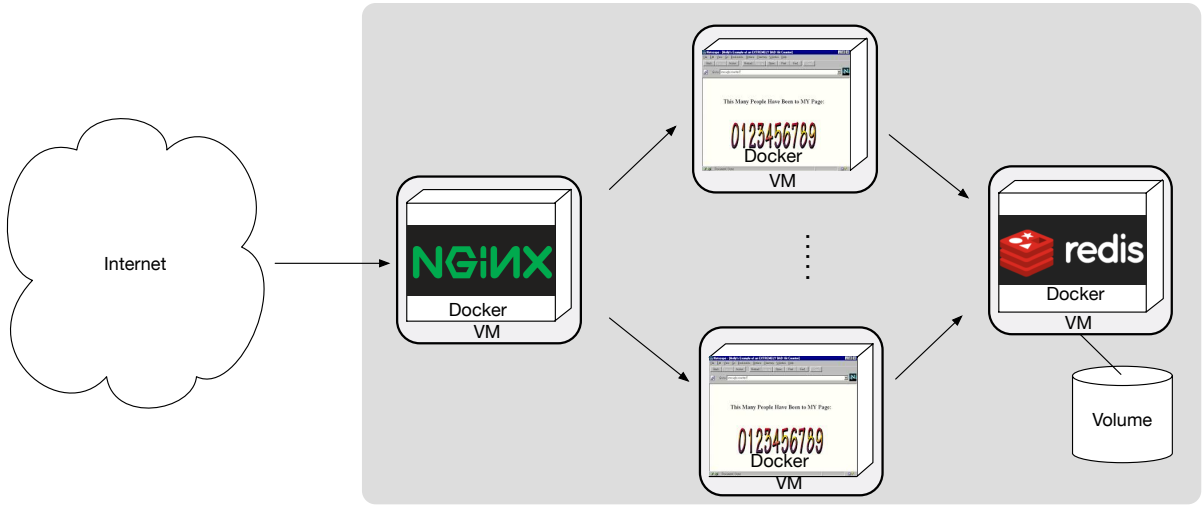
More info on Docker and Redis is found below and as always, Google is your friend!

https://docs.docker.com/samples/library/redis/
https://markheath.net/post/exploring-redis-with-docker

Please note that Redis database should reside on an OpenStack Volume, which will provide persistent storage via the OpenStack service Cinder.

You shall also modify your Flask-based frontend to make use of the redis database instead of Swift containers. There are great tutorials on how to store and atomically update redis values from Python, so just Google for it.

When you have configured your Flask frontend to connect to redis, you should now add an instance of NGINX to act as a load balancer/reverse proxy. The whole setup should look something like this:



Just like with Redis, there are ready-made Nginx containers to download from Docker-hub, e.g.

```
docker run --name mynginx1 -P -d nginx
```

The full assignment is therefore that you should define three kinds of Docker images that upon launch configure themselves correctly. Letting your various server instances download the correct Docker images upon booting up is a nice way of distributing software of the right version to each of them.

You choose how you want to deploy these services. Terraform or Ansible are recommended approaches, but you could of course manage this just fine with a bash script of epic proportions, as well. Do note the next section on service discovery, as there are some runtime configurations that you will need to do.

When you have completed this assignment, you have essentially deployed a three-tier web application, which is hardly trivial work. Good job!

A note about service discovery

One of the more difficult things about this assignment is the subtle detail of how the services find each other. How does the Flask app know what IP address the redis database has, and how does the NGINX load balancer know where to find the Flask app instances? The answer to these questions depends on your technology of choice!

In Ansible, you can obtain the IP addresses of the servers that have specific roles. You can then feed that into configuration files (or environment files) and restart the applications so that they pick up the appropriate values.

In Terraform, you can make use of the “attributes” (roughly “output variables”, if you recall Lecture #2) from each created server and feed them into the other servers’ user-data. Changes to a given server’s user-data causes the server to be replaced, so you know that changes to e.g. the redis database IP will cause the Flask server instances to be re-created,

which will then also in turn cause the NGINX server to be replaced. Terraform figures out these dependencies and does the appropriate thing. It will of course also attach the floating IP to the new NGINX instance for you.

In upcoming assignments and in your project, you will use Kubernetes, which has service discovery built in via DNS. With DNS, you will be able to just say “connect to the name redis” and leave the hairy details to the provided Kubernetes DNS service. If you want to configure something like this yourself for bragging rights, you can look into Consul. It is however not at all required, and will likely be quite a time investment (but as always, you learn a lot, too).