Cloud Computing
Project Deliverable:
IaaS - Virtualization and Containerization

Jan 25 2022

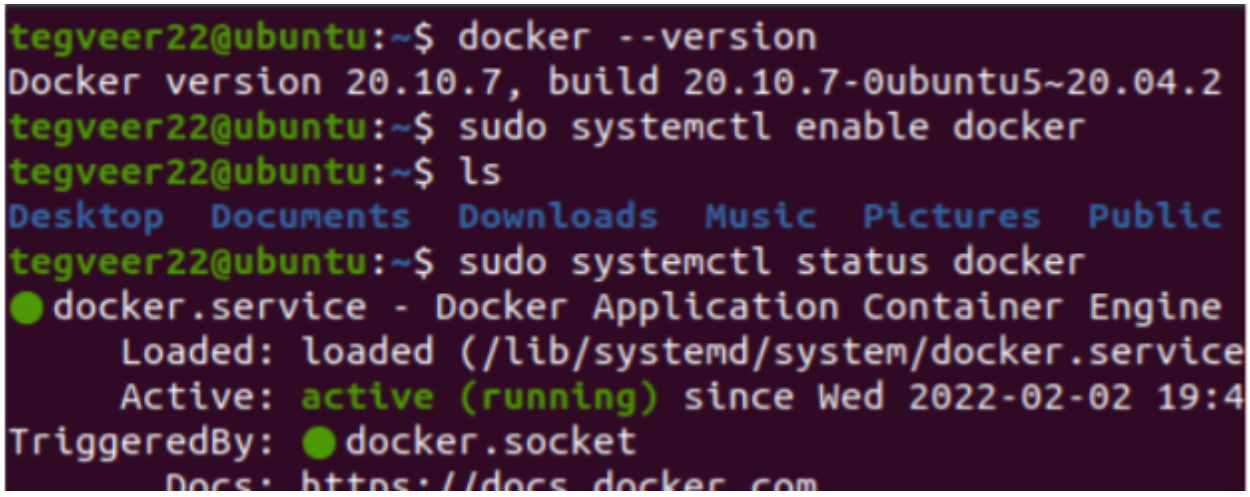
Group 7
Owais Quadri 100697281
Tegveer Singh 100730432
Danial Asghar 100671850
Shayan Sepasdar 100722542

GitHub Link
https://github.com/danialasghar/Cloud-Group-7

The following table compares Containers vs Virtual Machines:

Containers	Virtual Machines
Provides lightweight isolation from the host and other containers but not a clear boundary as a VM	Provides complete isolation from host and other VMs
Namespaces create the illusion for each container OS and Cgroups limit the resource usage so as to not tax the OS	Hardware virtualized by Hypervisor for Virtual Machines
High portability due to use of Dockerfile	VMs aren't as portable as they require full transfer of resources and OS
An orchestrator can easily manager containers starting/stopping depending on load changes	VM load balancing is done by running them in other servers within a failover cluster
Hardware interfaced to Host OS by the Kernel	Hardware layer followed by Hypervisor to create virtualized instances of processors, RAM, network cards
Runs on top of the Docker engine without needing a full OS, saving resources	Requires a complete OS and kernel which requires more hardware resources
Has to run on the system OS only	Can run any different type of OS without having any constraints

Docker installation on a Linux OS:



The following section explains the basic Docker terminology:

Docker image: The Docker image is created from a Dockerfile which defines the instructions to

build a Docker image; it is an immutable file containing source code, libraries, dependencies and all files required for the execution of the application.

Docker container: A container is the executing instance of an image; it is a virtualized and sandboxed runtime environment useful in application development. The containers are portable and compact, allowing them to be started and stopped quickly. A container requires and depends upon a runnable Docker image as the container provides the runtime environment and the image provides the application to execute.

Docker registry: A registry contains named Docker images that can be stored and distributed. The Docker Hub, Docker's registry, allows users to store and distribute Docker images.

Video 1: Intro to Docker
The following lists the commands and their respective explanations from Video 1:

- `docker build -t helloworld:1.0 .`
    - The Docker build command is used to build an image with the specified tag of "helloworld", version "1.0".
- `docker images`
    - Returns a list of all Docker images within the system. It also includes their repository, tags and size information.
- `docker run helloworld:1.0`
    - This command instantiates a container and runs the image "helloworld:1.0" using the specified entry point command.
- `docker ps`
    - Returns a list of all running containers.
- `docker ps -a`
    - Returns a list of all containers, running or stopped.
- `docker run -d helloworld:2.0`
    - Runs the Docker container from "helloworld" image, version "2.0". The -d flag ensures the container runs in detached mode, which allows the container to execute in the background and doesn't lock the terminal during execution.
- `docker logs container_id`
    - Retrieves a batch log of the container identified by the container id or app name.

The following commands can be used to stop and delete containers:

- `docker stop container_id`
    - Stops the container referenced by "container_id".
- `docker kill container container_id`
    - Kills the container referenced.
- `docker rm container_id`
    - Deletes a stopped container.
- `docker rm -f container_id`
    - Stops and deletes a container when the -force tag is used.
- `docker container prune`
    - Deletes all stopped containers within the system.

The following video shows a container being created, displaying the container's logs, stopping them and deleting them:
https://drive.google.com/file/d/1b0rbgssuKb2wf6NXrY1dTY6uKRf_x5ID/view

What are multi-container applications?
A multi-container Docker application runs more than one container providing different features as microservices. These features provide low coupling and lightweight processes. Modern, modular applications do not run in a single monolithic component. Within Docker each container should do one thing. With respect to the provided web app, one container was for MySQL and another for the Web App.

How do containers communicate among themselves?
Containers in a multi-container application communicate amongst themselves using network bridges. Docker uses a software bridge for bridge networks, allowing containers connected to the same bridge to communicate with each other, while still isolating from containers not connected to the bridge. In our application, the app-db is connected with the web application using network bridges.

Commands to stop and delete a container:

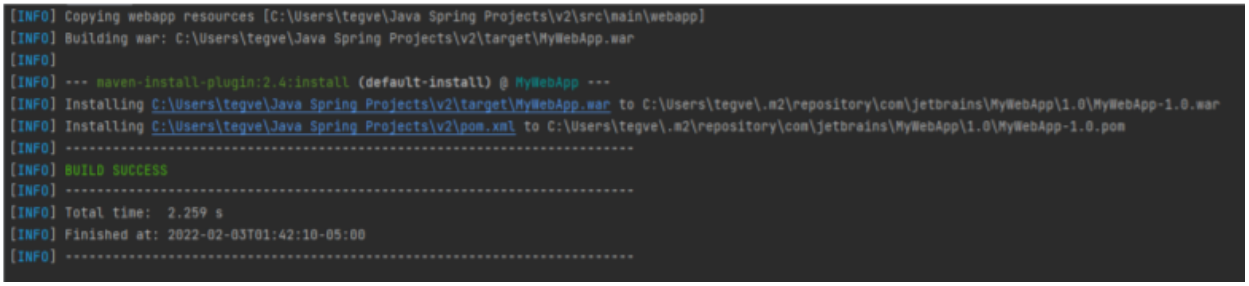
- `docker rm -f container_ID`
  - Force stops and deletes a container.
- `docker stop container_ID`
  - Only stops a running container.
- `docker rm container_ID`
  - Deletes a stopped container.

Video 2: Intro to Docker (Networking, Docker Compose)
The following list contains the commands used within Video 2 and their respective explanations:

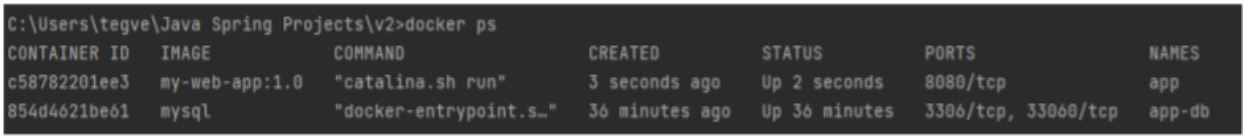
- `docker pull mysql`
  - Retrieves the official MySQL image from Docker Hub.
- `docker run --name app-db -d -e MYSQL_ROOT_PASSWORD=password -e MYSQL_DATABASE=myDB mysql`
  - Runs a MySQL container from the officially pulled image.
  - The --name option tags the container by a specific name.
  - The -d option is specified to run the container in detached mode.
  - The -e option specifies that it will be followed by a configuration environment variable. The MySQL root password and database name are passed in as environment variables.
- `mvn install`
  - Maven command used to download dependencies, build and create a .war file which is used within the container image.
- `docker run --name app -d -p 8080:8080 my-web-app:1.0`
  - Start running a container from the "my-web-app" image, version 1.0.
  - Creates the name label "app".
  - Binds the container port 8080 to the host port 8080 to allow communication.
- `docker network create app-network`

    - Create a dedicated bridge network between two containers and specify its name as "app-network"
- `docker network ls`
    - Lists all the running networks within Docker.
- `docker network connect app-network app-db`
    - Connects the "app-db" container to the bridge network "app-network"
- `docker network connect app-network app`
    - Connects the "app" container to the bridge network "app-network"
- `docker compose up -d`
    - Builds, recreates, starts, and attaches to containers for a service. The instructions are present in the docker-compose.yml file.

Screenshots for Video 2:
The following figure shows the output when using *mvn install*:



The following image shows both containers for the web application and the MySQL db running:



The following image shows the web-application running and accessible from localhost:8080:

The following video shows creating the web application, running the full web-app with DB communication, stopping and deleting the containers:
https://drive.google.com/file/d/1PdVGW4W1pMrYg6MuzQTtYK_h_sxk63V1/view

The following video displaying deploying Docker and Kubernetes on Google Cloud Platform:
https://drive.google.com/file/d/175W-xvg7DtmmsE1UwCSSjxlXnKfyYl_t/view

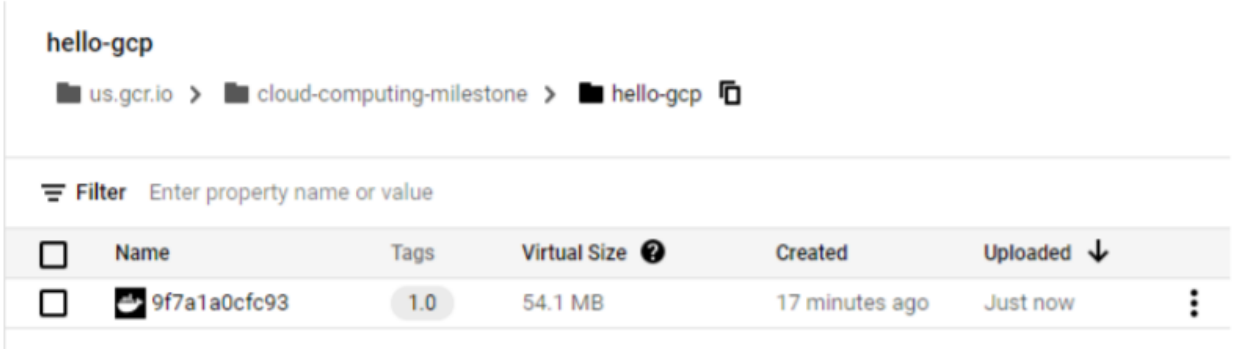
Video 3: Google Kubernetes engine: Create GKE Cluster and Deploy sample website
The following list contains the commands used within video 3 and their respective explanations:

- `gcloud config set project project_name`
    - Set the current project in the cloud shell to be a newly created project for your website cloud computing milestone
- `gcloud config set compute/zone us-central1-a`
    - Set a default compute zone for GK cluster to enable the kubectl command and related GCP services
- `docker run -d -p 8080:80 nginx:latest`
    - Run the nginx server (in detached version using -d) by exposing port 8080 of the cloud shell port 80 for the nginx web server using the following command
- `docker cp index.html container_id:/usr/share/nginx/html`
    - Copy a sample index.html file onto the nginx server file system using the command
- `docker commit container_id image_name:tag`
    - Commit the newly added file changes to a newly created docker image using the following command
- `docker tag image_name:version_tag us.gcr.io/project_name/image_name:version`
    - Tag the source image such that it refers to a target image in the GCP container registry
- `docker push us.gcr.io/project_name/image_name:version`
    - Push the image onto the registry
- `gcloud container clusters create gk-cluster --num-nodes=1`
    - Create a Google Kubernetes cluster and name that cluster gk-cluster. The number of nodes have been specified as 1
- `gcloud container clusters get-credentials gk-cluster`
    - Get the authentication credentials to deploy the container by provisioning kubectl which communicates with the API
- `kubectl create deployment web-server --image=us.gcr.io/project_name/image_name:version_tag`
    - Create a deployment service to the cluster and name it web-server. --image specifies the container image to use
- `kubectl expose deployment web-server --type LoadBalancer --port 80 --target-port 80`
    - Expose your service running within a cluster over an internet port.
      --port specifies public internet port

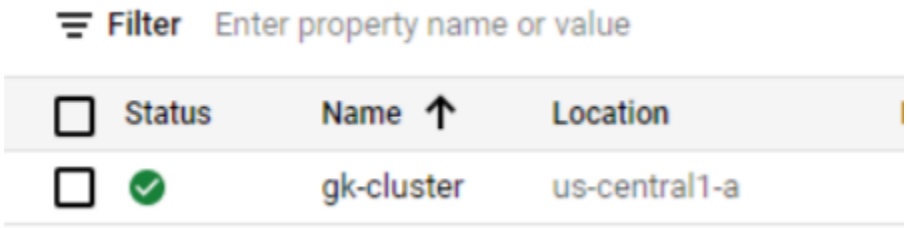
--target-port specifies the nginx server port
--type signifies the compute engine Kubernetes service

- `kubectl get service web-server`
    - This commands provides a public IP that can be used to access the site

Screenshots for Video 3:
The following image shows the container pushed within the GC registry:



The following image shows the Kubernetes cluster in ready status within the GCP console:



The following video displays deploying the web application using a YAML file on GCP:
https://drive.google.com/file/d/1VB4g99POmPbrVuiy5SCF3oTtqpVyfQoJ/view

The following section explains the basic Kubernetes terminology:
Kubernetes Pods: Pods are the smallest deployable units of execution which can contain one or more containers. Containers within a pod share storage, network resources and include a specification for how the containers should be run.

Kubernetes Service: Services provide a way to expose pods to the internet on the port specified by the kubectl command. It defines a logical set of Pods and a policy by how they are accessible.

Kubernetes Node: A node is a working entity to manage pods in Kubernetes. Nodes are what Pods run upon, they can be virtual or physical machines managed by a control plane.

Kubernetes Deployment: Instruction set given to Kubernetes on how to deploy the pods. The Deployment runs multiple replicas of the application and monitors health so that any Pods which fail get replaced.

Replicas: A program that runs multiple pods so as to provide backup and increase scalability in the programs. Kubernetes uses the ReplicaSet to ensure a specified number of pod replicas are running at a given time to match the minimum availability requirement.

Types of Kubernetes Services: There are four types of Kubernetes services: ClusterIP, NodePort, LoadBalancer and ExternalName. The following section describes each service:

- ClusterIP: Exposes a service within the cluster; this is the default type of service.
- NodePort: Exposes service via static ports present on all nodes' IPs. Kubernetes routes traffic automatically from the static IP port to the service.
- LoadBalancer: Uses the cloud provider's load balancer to expose the service.
- ExternalName: Maps a service to a predefined externalName field by returning a value for the CNAME record.