Cloud assignment

Q1. Create and run a docker container that has a NodeJS app inside it

To create and run a Docker container with a Node.js app inside it, you'll need to follow these steps:

Create a Node.js application.

Create a Dockerfile to specify how the Docker image should be built.

Build the Docker image.

Run the Docker container.

Here's a simple example to demonstrate this process:

Step 1: Create a Node.js Application

Create a directory for your Node.js application and navigate into it. Then create a file named app.js with the following content:

javascript

Copy code

// app.js

const http = require('http');

const hostname = '0.0.0.0';

const port = 3000;

const server = http.createServer((req, res) => {

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Hello World\n');

});

server.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`);

});

Step 2: Create a Dockerfile

Create a file named Dockerfile in the same directory as your app.js file with the following content:

Dockerfile

Copy code

# Use the official Node.js 14 image as a base

FROM node:14

# Set the working directory in the container

WORKDIR /usr/src/app

# Copy package.json and package-lock.json to the working directory

COPY package\*.json ./

# Install the dependencies

RUN npm install

# Copy the rest of the application code to the working directory

COPY . .

# Expose port 3000 to the outside world

EXPOSE 3000

# Command to run the application

CMD [ "node", "app.js" ]

Step 3: Build the Docker Image

Open a terminal, navigate to the directory containing your Dockerfile and app.js file, and run the following command to build the Docker image:

perl

Copy code

docker build -t my-node-app .

Step 4: Run the Docker Container

Once the image is built, you can run a container using the following command:

arduino

Copy code

docker run -p 3000:3000 my-node-app

This command will start a container based on the my-node-app image and map port 3000 from the container to port 3000 on your host machine. You should now be able to access your Node.js application by visiting http://localhost:3000 in your web browser.

That's it! You've created and run a Docker container with a Node.js app inside it.

Q2. Create and run a NodeJS app along with a DB container using docker networks

To create and run a Node.js app along with a database container (e.g., MongoDB) using Docker networks, follow these steps:

Step 1: Create a Node.js Application

Create your Node.js application as you normally would. For this example, let's assume your Node.js application code is in a directory named node-app.

Step 2: Dockerize the Node.js Application

Create a Dockerfile in your node-app directory:

Dockerfile

Copy code

# Use the official Node.js 14 image as a base

FROM node:14

# Set the working directory in the container

WORKDIR /usr/src/app

# Copy package.json and package-lock.json to the working directory

COPY package\*.json ./

# Install the dependencies

RUN npm install

# Copy the rest of the application code to the working directory

COPY . .

# Expose port 3000 to the outside world

EXPOSE 3000

# Command to run the application

CMD [ "node", "app.js" ]

Step 3: Create a Docker Compose File

Create a docker-compose.yml file in the same directory as your node-app directory:

yaml

Copy code

version: '3.8'

services:

node-app:

build: ./node-app

ports:

- "3000:3000"

depends\_on:

- db

networks:

- app-network

db:

image: mongo:latest

restart: always

environment:

MONGO\_INITDB\_ROOT\_USERNAME: root

MONGO\_INITDB\_ROOT\_PASSWORD: example

ports:

- "27017:27017"

networks:

- app-network

networks:

app-network:

driver: bridge

Step 4: Run Docker Compose

Open a terminal, navigate to the directory containing your docker-compose.yml file, and run the following command:

Copy code

docker-compose up

This command will start both the Node.js application container and the MongoDB container. They will be able to communicate with each other within the same Docker network (app-network), which is created by Docker Compose automatically.

Step 5: Access the Node.js Application

Once both containers are running, you can access your Node.js application at http://localhost:3000.

That's it! You've created and run a Node.js application along with a MongoDB container using Docker networks.

Q3. Create and run a NodeJS app along with a persistently stored DB container using docker-compose

To create and run a Node.js app along with a persistently stored database container (e.g., MongoDB) using Docker Compose, you can follow these steps:

Step 1: Set up the Node.js Application

Assuming you have a Node.js application in a directory named node-app, with a Dockerfile as described in the previous examples.

Step 2: Set Up Docker Compose

Create a docker-compose.yml file in the root directory of your project:

yaml

Copy code

version: '3.8'

services:

node-app:

build: ./node-app

ports:

- "3000:3000"

depends\_on:

- db

networks:

- app-network

db:

image: mongo:latest

restart: always

environment:

MONGO\_INITDB\_ROOT\_USERNAME: root

MONGO\_INITDB\_ROOT\_PASSWORD: example

ports:

- "27017:27017"

volumes:

- db-data:/data/db

networks:

- app-network

networks:

app-network:

driver: bridge

volumes:

db-data:

driver: local

Step 3: Explanation

We define two services: node-app for our Node.js application and db for MongoDB.

For the node-app service:

We specify the build context as ./node-app, where our Node.js application resides.

We expose port 3000 to access our Node.js application from the host.

It depends on the db service to ensure that MongoDB is running before the Node.js app starts.

For the db service:

We use the official MongoDB image.

We set environment variables for the MongoDB root username and password.

We expose port 27017 to access MongoDB from the host.

We use a named volume (db-data) to persistently store MongoDB data.

Step 4: Run Docker Compose

Open a terminal, navigate to the directory containing your docker-compose.yml file, and run the following command:

Copy code

docker-compose up

This command will start both the Node.js application container and the MongoDB container, creating a persistent volume for MongoDB data.

Step 5: Access the Node.js Application

Once both containers are running, you can access your Node.js application at http://localhost:3000.

That's it! You've created and run a Node.js application along with a persistently stored MongoDB container using Docker Compose.

Q4. Create and run a complex application that has multiple services

To create and run a complex application that consists of multiple services using Docker Compose, you can follow these steps:

Define the Services:

Determine the services that your application will consist of. For example, let's create a simple application with two services: a frontend web server using Node.js and a backend database using MongoDB.

Create Dockerfiles:

For each service, create a Dockerfile to define its environment and dependencies. Below are sample Dockerfiles for the frontend and backend services:

Frontend Service Dockerfile (Node.js):

Dockerfile

Copy code

FROM node:14

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD [ "node", "app.js" ]

Backend Service Dockerfile (MongoDB):

Dockerfile

Copy code

FROM mongo

COPY init-db.js /docker-entrypoint-initdb.d/

Create Docker Compose Configuration:

Create a docker-compose.yml file in your project directory to define the services and their configurations. Below is a sample docker-compose.yml for our application:

yaml

Copy code

version: '3'

services:

frontend:

build: ./frontend

ports:

- "3000:3000"

depends\_on:

- backend

backend:

image: mongo

volumes:

- ./data:/data/db

This configuration defines two services: frontend and backend. The frontend service builds from the Dockerfile located in the frontend directory and exposes port 3000. The backend service uses the official MongoDB image.

Set Up Application Code:

Create the application code for each service and place it in separate directories (frontend and backend in this example). Make sure the code corresponds to the Dockerfiles and Docker Compose configuration.

Run Docker Compose:

Open a terminal in your project directory.

Run the following command to start the services defined in the docker-compose.yml file:

bash

Copy code

docker-compose up

Docker Compose will build the necessary images (if not already built) and start the services.

Access the Application:

Once the services are up and running, you can access the frontend application in your web browser by navigating to http://localhost:3000.

Stopping the Services:

To stop the services and remove the containers, press Ctrl + C in the terminal where docker-compose up is running.

Optionally, you can run the following command to stop and remove the containers:

bash

Copy code

docker-compose down

That's it! You've created and run a complex application with multiple services using Docker Compose. You can add more services or modify the existing ones to suit your application's requirements.

Q5. Create a configuration file to setup and run a simple Kubernetes cluster

To create a configuration file to set up and run a simple Kubernetes cluster, you need to define several resources including the cluster itself, nodes, deployments, services, etc. Below is an example configuration file for setting up a simple Kubernetes cluster using Minikube, a lightweight Kubernetes implementation:

yaml

Copy code

# cluster.yaml

# API Version

apiVersion: v1

# Define the kind of object

kind: Pod

# Metadata for the pod

metadata:

name: my-pod

# Specification for the pod

spec:

containers:

- name: my-container

image: nginx:latest

ports:

- containerPort: 80

This YAML configuration file defines a simple Pod named my-pod running an NGINX container.

To run this configuration, follow these steps:

Install Minikube: Install Minikube on your local machine. You can find installation instructions on the Minikube documentation: https://minikube.sigs.k8s.io/docs/start/

Start Minikube: Start Minikube using the following command:

bash

Copy code

minikube start

Apply Configuration: Apply the configuration file using the following command:

bash

Copy code

kubectl apply -f cluster.yaml

Verify Pods: Verify that the Pod is running using the following command:

bash

Copy code

kubectl get pods

You should see the my-pod Pod in the list of running Pods.

This is a basic example to get you started with a Kubernetes cluster using Minikube. You can explore more advanced configurations and Kubernetes features as you become more familiar with the platform.

Q6. Create a configuration file to setup and run an application with multiple services running in a cluster

To create a configuration file to set up and run a simple Kubernetes cluster, you need to define several resources including the cluster itself, nodes, deployments, services, etc. Below is an example configuration file for setting up a simple Kubernetes cluster using Minikube, a lightweight Kubernetes implementation:

yaml

Copy code

# cluster.yaml

# API Version

apiVersion: v1

# Define the kind of object

kind: Pod

# Metadata for the pod

metadata:

name: my-pod

# Specification for the pod

spec:

containers:

- name: my-container

image: nginx:latest

ports:

- containerPort: 80

This YAML configuration file defines a simple Pod named my-pod running an NGINX container.

To run this configuration, follow these steps:

Install Minikube: Install Minikube on your local machine. You can find installation instructions on the Minikube documentation: https://minikube.sigs.k8s.io/docs/start/

Start Minikube: Start Minikube using the following command:

bash

Copy code

minikube start

Apply Configuration: Apply the configuration file using the following command:

bash

Copy code

kubectl apply -f cluster.yaml

Verify Pods: Verify that the Pod is running using the following command:

bash

Copy code

kubectl get pods

You should see the my-pod Pod in the list of running Pods.

This is a basic example to get you started with a Kubernetes cluster using Minikube. You can explore more advanced configurations and Kubernetes features as you become more familiar with the platform.

Q6. Create a configuration file to setup and run a replicaset for an nginx pod

To create a configuration file to set up and run a ReplicaSet for an NGINX Pod in Kubernetes, you can use the following YAML configuration:

yaml

Copy code

# nginx-replicaset.yaml

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-replicaset

spec:

replicas: 3 # Number of NGINX replicas you want to run

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

This configuration file defines a ReplicaSet named nginx-replicaset that ensures there are three replicas of an NGINX Pod running at all times.

To apply this configuration to your Kubernetes cluster, save it to a file (e.g., nginx-replicaset.yaml) and use the kubectl apply command:

bash

Copy code

kubectl apply -f nginx-replicaset.yaml

Kubernetes will create and manage the NGINX Pods according to the specified ReplicaSet configuration. You can adjust the number of replicas by modifying the replicas field in the YAML file.

Q7. Create a configuration file to setup and run an application with multiple services and persistent storage running in a cluster

To set up an application with multiple services and persistent storage running in a Kubernetes cluster, you'll need to define various Kubernetes resources such as Deployments, Services, PersistentVolumeClaims (PVCs), and PersistentVolumes (PVs). Below is an example YAML configuration file for such a setup:

yaml

Copy code

# app-with-storage.yaml

# Define ConfigMap for application configuration

apiVersion: v1

kind: ConfigMap

metadata:

name: my-app-config

data:

APP\_NAME: "My Application"

APP\_VERSION: "1.0"

# Define Secrets for sensitive information

apiVersion: v1

kind: Secret

metadata:

name: my-app-secrets

type: Opaque

data:

DB\_USERNAME: "base64\_encoded\_username"

DB\_PASSWORD: "base64\_encoded\_password"

# Define PersistentVolumeClaim for database

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: db-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

# Define Deployment for database service

apiVersion: apps/v1

kind: Deployment

metadata:

name: db-deployment

spec:

replicas: 1

selector:

matchLabels:

app: db

template:

metadata:

labels:

app: db

spec:

containers:

- name: db

image: mysql:latest

ports:

- containerPort: 3306

env:

- name: MYSQL\_ROOT\_PASSWORD

valueFrom:

secretKeyRef:

name: my-app-secrets

key: DB\_PASSWORD

volumes:

- name: db-storage

persistentVolumeClaim:

claimName: db-pvc

# Define Service for database service

apiVersion: v1

kind: Service

metadata:

name: db-service

spec:

selector:

app: db

ports:

- protocol: TCP

port: 3306

targetPort: 3306

# Define Deployment for frontend service

apiVersion: apps/v1

kind: Deployment

metadata:

name: frontend-deployment

spec:

replicas: 2

selector:

matchLabels:

app: frontend

template:

metadata:

labels:

app: frontend

spec:

containers:

- name: frontend

image: my-frontend-image:latest

ports:

- containerPort: 3000

env:

- name: DB\_HOST

value: "db-service"

# Define Service for frontend service

apiVersion: v1

kind: Service

metadata:

name: frontend-service

spec:

selector:

app: frontend

ports:

- protocol: TCP

port: 80

targetPort: 3000

This YAML configuration file defines an application with two services: a frontend service and a database service (MySQL), along with PersistentVolumeClaims for persistent storage for the database.

To apply this configuration to your Kubernetes cluster, save it to a file (e.g., app-with-storage.yaml) and use the kubectl apply command:

bash

Copy code

kubectl apply -f app-with-storage.yaml

Kubernetes will create and manage the necessary resources to run the application with persistent storage in your cluster. Adjust the configuration according to your application's requirements and desired storage size.

Q8. Write a simple Ansible playbook

A simple Ansible playbook typically includes a list of tasks that describe the desired state of remote hosts. Here's an example of a simple Ansible playbook that installs Nginx on Ubuntu servers:

yaml

Copy code

# playbook.yml

---

- name: Install Nginx

hosts: web\_servers

become: yes # Run tasks with sudo

tasks:

- name: Update apt package cache

apt:

update\_cache: yes

- name: Install Nginx

apt:

name: nginx

state: present

- name: Start Nginx service

service:

name: nginx

state: started

Explanation:

name: The name of the playbook.

hosts: The target hosts where the tasks will be executed. In this example, the tasks will be applied to hosts belonging to the group web\_servers. You need to define this group in your Ansible inventory file (/etc/ansible/hosts by default).

become: Specifies that the tasks should be run with sudo privileges.

tasks: List of tasks to be executed on the target hosts.

apt: Ansible's built-in module for managing packages on Debian-based systems like Ubuntu. The first task updates the apt package cache.

service: Ansible's module for managing services. The second task installs Nginx using the apt module.

The third task starts the Nginx service.

To execute this playbook, save it to a file (e.g., playbook.yml) and run the following command:

bash

Copy code

ansible-playbook -i inventory\_file playbook.yml

Replace inventory\_file with the path to your Ansible inventory file. Ensure that your inventory file contains the group web\_servers with the appropriate IP addresses or hostnames of your Ubuntu servers.

This playbook will install Nginx on the specified servers and start the Nginx service. You can modify the playbook to include additional tasks based on your requirements.

Q9. Write an Ansible playbook to configure an application server

Sure, here's an example Ansible playbook to configure an application server:

yaml

Copy code

# playbook.yml

---

- name: Configure Application Server

hosts: application\_servers

become: yes # Run tasks with sudo

tasks:

- name: Update apt package cache

apt:

update\_cache: yes

- name: Install required packages

apt:

name:

- python3

- python3-pip

- nginx

state: present

- name: Ensure Nginx is running and enabled

service:

name: nginx

state: started

enabled: yes

- name: Copy application files

copy:

src: /path/to/application

dest: /var/www/html/application

remote\_src: yes

- name: Configure Nginx for the application

template:

src: nginx.conf.j2

dest: /etc/nginx/sites-available/application

notify: Restart Nginx

handlers:

- name: Restart Nginx

service:

name: nginx

state: restarted

Explanation:

name: The name of the playbook.

hosts: The target hosts where the tasks will be executed. In this example, the tasks will be applied to hosts belonging to the group application\_servers. You need to define this group in your Ansible inventory file (/etc/ansible/hosts by default).

become: Specifies that the tasks should be run with sudo privileges.

tasks: List of tasks to be executed on the target hosts.

apt: Ansible's built-in module for managing packages on Debian-based systems like Ubuntu. The first task updates the apt package cache and installs required packages (python3, python3-pip, and nginx).

service: Ansible's module for managing services. The second task ensures Nginx is running and enabled.

copy: Copies the application files from the local machine to the application server.

template: Copies a Jinja2 template file (nginx.conf.j2) to the application server and configures Nginx accordingly.

handlers: List of handlers, which are tasks that are triggered by other tasks. In this case, the Restart Nginx handler is triggered whenever the Configure Nginx for the application task is completed.

Make sure to replace /path/to/application with the actual path to your application files and provide the appropriate content for the nginx.conf.j2 template file.

To execute this playbook, save it to a file (e.g., playbook.yml) and run the following command:

bash

Copy code

ansible-playbook -i inventory\_file playbook.yml

Replace inventory\_file with the path to your Ansible inventory file. Ensure that your inventory file contains the group application\_servers with the appropriate IP addresses or hostnames of your application servers.