Cloud Computing 2022/23 spring

Practice 12 - Working with Kubernetes

Kubernetes, also known as K8s, is an open-source system for automating the deployment, scaling, and management of containerized applications. In this practice session, you will explore setting up a Kubernetes cluster using K3s. K3s, is an open-source lightweight Kubernetes used to set up for development environments wherever resources are limited. Further, you will learn to create, deploy and scale the containers in a pod and manage the message board application.

References

- Kubernetes documentation: https://kubernetes.io/docs/home/
- K3s guide : https://docs.k3s.io/

Exercise 12.1. Setting up of Kubernetes Cluster!

In this task, you're going to set up two Kubernetes cluster using k3s. Further, you will configure **kubectl** to interact with the Kubernetes cluster.

- Create two VMs with the names `Lab12_LASTNAME_Master` and `Lab12_LASTNAME_Worker`
  - Source: Use Image `ubuntu22.04`
    - Enable "Delete Volume on Instance Delete"
  - Flavour: `m2.tiny`
  - Security Groups: `K8S cluster`
- Connect to **Master** VM, In this VM we will configure a k3s server
- Installing the k3s on Master VM:
  - K3s provides an installation script that is a convenient way to install it as a service on systemd or openrc-based systems. This script is available at https://get.k3s.io.
  - To install K3s, just run `curl -sfL https://get.k3s.io | sh -`
  - To access k3s cluster, *kubectl* is needed and its installed during k3s setup.
  - After installation, A *kubeconfig* file will be written to /etc/rancher/k3s/k3s.yaml and its used by *kubectl* to access the k3s cluster.
  - Notedown the *K3S_TOKEN* in notepad, located `/var/lib/rancher/k3s/server/node-token` using `cat` command. This token is used to register the agents or extra nodes in to the cluster.
  - To confirm that K3s has been set up successfully, run the command `sudo kubectl get nodes`. You should see the output with one node and k3s version etc.
- Installing the k3s agent on Worker VM:
  - Login to Worker VM, and run the below command to add the node to the Kubernetes cluster
    - Change the *myserver* to IP of your Master Node and *mynodetoken* to token noted down in the previous step `curl -sfL https://get.k3s.io | K3S_URL=https://myserver:6443 K3S_TOKEN=mynodetoken sh -`
- Check the nodes added to the cluster using `sudo kubectl get nodes`
- You can display the detailed information about the nodes using `sudo kubectl get nodes -o wide`
- To avoid using sudo when running kubectl, you can change the permissions of kubeconfig `sudo chmod 644 /etc/rancher/k3s/k3s.yaml`
- Deliverable: Take a screenshot of the output of the command `kubectl get nodes -o wide`.

Exercise 12.2. Working with Pods!

Pods are the smallest, most basic deployable objects in Kubernetes. They can hold one or a group of containers which share the storage, network, and other resources. In this task, you will work with kubectl to create pods and deploy the services.

Kubernetes objects are persistent entities in the Kubernetes system. To interact with the Kubernetes API of the cluster, **kubectl** uses the `.yaml` file format to specify the Kubernetes objects. You can read up on writing and reading YAML for Kubernetes Guide

Task 12.2.1: Running a container in a Pod

- We will write the YAML file to create the first pod (Which runs with a single busy box container).
- Create a new YAML file named `first_pod.yaml`
  - The First statement starts with apiVersion; this indicates the version of the Kubernetes API you're using to create this object.
    - Add `apiVersion: v1` into the file
  - Next, let's specify what kind of object we want to configure/create. For example **Pod** or **Deployment**.
    - Add `kind: Pod` line
  - Next, we will define the metadata block. It helps to uniquely identify the object, including a name string, label, UID, and optionally the name of the namespace.
    - Name should be unique; a label can match a set of pods with the same functionality/task/goal (for example, all pods of the same backend API implementation).
    - Let's assign **first** as the name of the pod and **myfirst** as the label.

```
metadata:
 name: first
 labels:
  app: myfirst
```

- Finally, mention the specification of the object itself, the Pod
  - Let's use busybox container image (from Docker Hub by default)
  - We will also specify the name for the container and what command-line command to run inside the container:

```
spec:
 containers:
 - name: myapp-container
   image: busybox
   command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 180']
```

- The final YAML file looks something like this: first_pod.txt

- To ask Kubernetes to create the pod: `kubectl create -f first_pod.yaml`
- Check the status of the pods: `kubectl get pods` or `kubectl get pods -o wide` (Check the difference between these two commands)
- To check the logs of the pod: `kubectl logs first`
- To get a more detailed description of the pod: `kubectl describe pod first`
  - Deliverable: Take a screenshot of the output of the above command.
- You can also open a command line inside the pod using
  - `kubectl exec -it first -- /bin/sh`
  - type the command `cat /etc/resolv.conf` inside the pod command line
    - This will print out information that pods use to resolve service names inside the local Kubernetes, VM, and university network
- Delete the pod: `kubectl delete -f first_pod.yaml`

PS! The command we specified inside the pod sleeps for 180 seconds. After that, the pod shuts down and is restarted by Kubernetes. So you might be "kicked" out of the container when this happens.

Task 12.2.2: Scaling the Pods with ReplicaSets

ReplicaSets are Kubernetes controllers that are used to maintain the number and running state of pods. This is mainly to keep a predefined number of pods running. If more pods are up, the additional ones are terminated. Similarly, if one or more pods failed, new pods are activated until the desired count is reached. The ReplicaSet uses labels to match the pods that it will manage.

- Creating ReplicaSet
  - Create a file with the name `replicaset.yml` and it contains:
  - The ReplicaSet definition file will be like this: replicaset.txt
    - The apiVersion should be: **v1**
    - The kind of this object is: **ReplicaSet**
    - In metadata, we define the name by which we can refer to this ReplicaSet: **web**. Further, we also define two types of labels:
      - `env: dev` (development environment)
      - `role: web` (serving web pages)
    - The spec part is mandatory and is used to specify:
      - The number of replicas (of Pod) that should be maintained.
      - The selection criteria by which the ReplicaSet will choose its pods (find pods that match the same labels).
      - The (Pod) template is used to define the specification of Pods, just like in the previous task.
- Create the ReplicaSet: `kubectl apply -f replicaset.yml`
- Check the status of the created ReplicaSet: `kubectl get rs`. Here, you see the 4 replicas of the Pod.
- Check the list of the Pods `kubectl get pods`. Take note of one pod name and remove the pod from the replica set using:
  - Removing a Pod from ReplicaSet `kubectl edit pods web-<SOME_SEQUENCE>`.

        - Copy the name of one of the pods from an earlier command (replacing SOME_SEQUENCE with an actual generated string of any one of the pods).
        - This command opens a (vi -based) editor with the deployment definition of that pod
        - Change the line `role:web` to `role:isolated`.
            - If you are not familiar with `vi`, then read some `vi` editor tutorial or check: https://www.atmos.albany.edu/daes/atmclasses/atm350/vi_cheat_sheet.pdf
        - After this, the pod still will be in a running state but will no longer be managed by the ReplicaSet Controller.
- Scale up the replicaset `kubectl scale --replicas=10 rs/web`
- Check the created replicas `kubectl get rs`
- Scale down the replica set to 1.
- Check the replica set.

Exercise 12.3. Working Kubernetes deployments

A Deployment resource uses a ReplicaSet to manage the pods. However, it handles updating them in a controlled way. **Deployments** are used to describe the desired state of Kubernetes. They dictate how Pods are created, deployed, and replicated.

ReplicaSet has one major drawback that once you select the pods that are managed by a ReplicaSet, you cannot change their pod templates. For example, if you want to change the image from `message-board.v1` to `message-board.v2` in the pod template, then you need to delete the ReplicaSet and create a new replica set. This increases the downtime.

Task 12.3.1: Creating the first deployment

In this task, we are going to work with deployment using kubectl. We will create a deployment for the Flask-based message board application. We also work with a rolling update to change the template image without downtime in the application deployment.

- Create a file with the name `first_deployment.yml`
    - Provide API version `apiVersion: apps/v1`
    - Mention the kind, here its `kind: Deployment`
    - Mention the `metadata:` object keys
        - Add `name: flask-deployment`
    - Provide specifications of the deployment as shown below

```
spec:
  selector:
    matchLabels:
      app: flask
  replicas: 2
  template:
    metadata:
      labels:
        app: flask
    spec:
      containers:
      - name: flask
        image: shivupoojar/message-board.v1
        ports:
        - containerPort: 5000
```

- Deployment will be chosen based on the matchable labels under the selector key during the deployment.
- Replicas will describe the number of pods that needs to be created.
- Template that includes the app name, list of containers, and associated objects.

- The final deployment file should be like Attach:first_deployment.txt.

- Deploy the definition file using `kubectl apply -f first_deployment.yml`
- Check your deployment `kubectl get deployments`. You should see the flask deployment with two replicas.
- Get the description of the deployment `kubectl describe deploy`
- List the pods `kubectl get po`. Here, you should see two pods.
- Let us use a rolling update to change the Pod's template image to a new version:
    - `kubectl set image deployments/flask-deployment flask=shivupoojar/message-board.v2`
- You should be able to see the change in the image name after running:
    - `kubectl describe pods`
- Kubernetes will replace Pods one by one, and after a while, you should only see Pods of the second version
- Delete the deployment `kubectl delete -f first_deployment.yml`.

Task 12.3.2: Creating deployment for Message Board Flask application and Postgresql database.

- Create a deployment definition file with the name `message-board.yml`
    - Use the deployment definition as in `first_deployment.yml` file and update `image` value like `image: <YOUR_DOCKER_HUB_USERNAME/IMAGE_NAME>` with your flask application developed and pushed into the docker hub in the Practice Session 2 in Task 2.4 (PS!! Messageboard application with PostgreSQL not with data.json)
- Similarly, we will create a deployment definition for postgress in `message-board.yml` as follows:
    - Append the following code modifications to `message-board.yml` file.
        - Add `---` at the end of the file.
        - You can copy the flask deployment block and modify the object as necessary
            - Change the metadata `name` key's value, `lables` `app` key values to `postgres`. Also `replicas: 1`.

        - In container spec, change the name to `postgresql`, image to `postgres:14.1-alpine`, and containerPort to `5432`.
- Just hold on in editing `message-board.yml` file; in the next section, we will add the definition for secrets.

Task 12.3.3: Creating Secrets to Store the Postgres Username and Password

In this task, we are going to create a secret to store the Postgres credentials.

Secret is an object that contains a small amount of sensitive data, such as a password, a token, or a key. It's not good practice to mention the definition of secrets directly in the deployment file - e.g., `message-board.yml`. In this task, we will create Kubernetes secrets through kubectl.

- Create secret using `kubectl create secret generic postgres-secret-config --from-literal=username=postgres --from-literal=password=<CHOOSE_FREELY_PASSWORD>`
    - You can choose the password freely.
- Check for the created secrets `kubectl get secret postgres-secret-config`
- Check the secrets in yml `kubectl get secret postgres-secret-config -o yaml`

Task 12.3.4: Updating deployment file to use secrets

Now, let us update the `message-board.yml` file to set environment variables for Postgres and Flask deployment.

- In `message-board.yml` file and move on to the Postgres deployment block.
- add the following after `Ports` object

```
        env:
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: postgres-secret-config
              key: username
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret-config
              key: password
```

- Move on to the flask deployment block and add the following block after `Ports`

```
        env:
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: postgres-secret-config
              key: username
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret-config
              key: password
        - name: DATABASE_URL
          value: postgresql://$(POSTGRES_USER):$(POSTGRES_PASSWORD)@postgresql:5432/postgres
```

Task 12.3.5: Testing the deployment

- Now, create the deployment `kubectl apply -f message-board.yml`
- Check the created deployments `kubectl get deployments` and secrets `kubectl get secrets`
- Deliverable: Take the screenshots of the above command - deployments.
- Check the list of pods. If you see errors like CrashLooplack, and can check using `kubectl logs <pod_name>`.
    - **But need not worry about errors** in flask application deployment. This is due to the Flask application cannot connect to the Postgres service since we did not expose the Postgres service to access outside the pod.
- Check the logs of Postgres and Flask Pods.
- Delete the deployment `kubectl delete -f message-board.yml`
- Deployment definition is not complete until we need to add service definition (We will add in Exercise 12.5) to expose the flask application to the outside world. Further, we need to expose the Postgres service to be accessible outside the pod.

Exercise 12.4. Working with Kubernetes Services

Service enables network access to a set of Pods in Kubernetes. Services select Pods based on their labels. Since the Flask-based Message-board application is only accessible within the cluster of pods. So Service is an object that makes a collection of Pods with appropriate labels available outside the cluster and provides a network load balancer to distribute the load evenly between the Pods. We will create a service that routes the network traffic to the pods with the label **flask**.

- Append to deployment file `message-board.yml`
    - Add `---` at the end of the file.
    - Add the API Version `apiVersion: v1`
    - Mention `kind: Service`
    - Add `metadata:` with `name: flask-service`
    - Add specification object `spec:`
        - Add ports and targetPort
        - Finally selector, the label associated with the pod
- The final service definition block should look like

```
---
apiVersion: v1
kind: Service
metadata:
  name: service-flask
spec:
  type: LoadBalancer
  selector:
    app: flask
  ports:
  - protocol: TCP
    port: 5000
    targetPort: 5000
    name: tcp-5000
```

Similarly, we need to create a service for Postgres deployment since it is accessed by the Flask application.

- Append to deployment file `message-board.yml`
    - Add `---` at the end of the file.
    - Change the metadata object `name: postgresql` and labels `name: postgresql`
    - In spec object, change `port: 5432` and remove the targetPort
    - In the selector block, change the app to `postgresql`

```
---
apiVersion: v1
kind: Service
metadata:
  name: postgresql
  labels:
    name: postgresql
spec:
  selector:
    app: postgres
  ports:
  - port: 5432
```

- Finally, now it is time to test the complete deployment of `message-board.yml`.
    - Create the deployment. After this, services and deployments are created.
    - Check the created pods `kubectl get po -o wide`
    - Check the deployments `kubectl get deployments`
    - Check for the services `kubectl get svc` and note down node port address <NODEPORT_ADDRESS>(Starts with 3**) to access in your brouser.
        - Open your message board application at http://MASTER_IP:<NODEPORT_ADDRESS> and enter a few messages.
        - Deliverable: Take a screenshot of the web page showing the message board (Your IP should be visible).
    - If you see the errors in the pods running status, then check the logs of the pods.

Scale the Flask application deployment

- Use the following commands to scale and check the flask deployment
    - Scale to 2 replicas `kubectl scale deployment flask-deployment --replicas=2`
    - Check the scaling of the pods `kubectl get pods`
    - Get the deployment `kubectl get deployment`
        - Deliverable: Take a screenshot of the output of the above commands.

Exercise 12.5. Working with Data Volume in Kubernetes

In this task, you're working with data volume management in Kubernetes application deployment. In the previous tasks, the message in the Postgres database will be deleted once you delete the pod. So need to make the data persist and accessible to newly created pods.

- In `message-board.yml` inside spec of containers in postgress deployment definition block
    - Add on more environment variable with name `name: PGDATA` and with value `value: /var/lib/postgresql/data/pgdata`
    - You can refer to the example here
    - Add `volumeMounts` with `name` to `data-storage-volume` and `mountPath` to `/var/lib/postgresql/data`
    - The final volume mounts should be inside the `containers` block like this:

        -
          ```
                  volumeMounts:
                    - name: data-storage-volume
                      mountPath: /var/lib/postgresql/data
          ```
        - It should be at the same level as `env` block under/inside the `containers` block!
    - Add the following under `spec` after the `containers` block. Same level as `containers` block!

        -
          ```
                volumes:
                  - name: data-storage-volume
                    persistentVolumeClaim:
                      claimName: postgres-db-claim
          ```
- You also create PersistentVolumeClaim, so create `data-volume.yml` and add the following

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-db-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
  storageClassName: manual
```

- Create the data volume `kubectl apply -f data-volume.yml`
- Re-create the deployment
  - NB! You may sometimes get an error if your Flask application creates database tables only when the container starts. When the database is re-created (as a result of recreating the Pod, database messages table might no longer exist)
    - Simply recreate the flask deployment or scale it to 0 and back to 2.
- Check for the mounts `kubectl describe pod <POSTGRES_POD>`
- Deliverable: Take a screenshot of the output of this command showing mounts.
- Open your application in the browser and add a few messages
  - PS!! Please check that, you using the correct version of the messageboard application. You should use the PostgreSQL-based messageboard application, not the data.json
  - You can do that by exec the pod and checking the app.py code that it uses Postgresql libraries.
- Delete and re-create the deployment (or scale individual deployments to 0 and then back to correct replica values again)
- Open the application in the browser, and you should see that the previous messages have not disappeared - as they are now stored inside a volume and not inside the Pod.

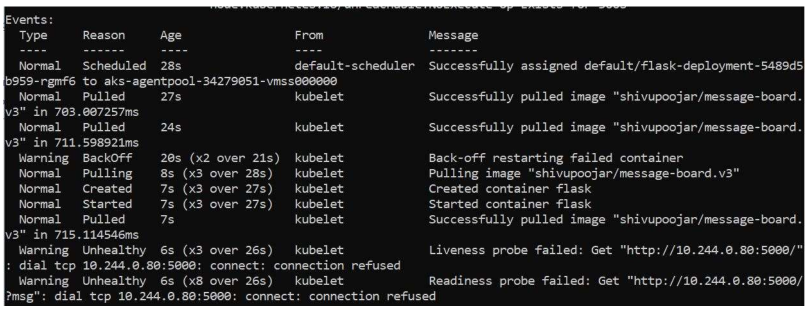
Bonus exercise: Checking the readiness and liveliness of services

The goal of the bonus tasks is to experiment with how service liveliness and lifecycle checks can be configured in Kubernetes deployments. Liveness and Readiness probes are used to control the health of an application running inside a Pod's container.

- Liveness probes: Kubernetes wants to know if your app is alive or dead.
- Readiness probes: Kubernetes wants to know when your app is ready to serve traffic.

In this, you need to add a definition for readiness and liveness checking for flask application.

- Use the `message-board.yml`
  - Add `livenessProbe:` and `readinessProbe:` in `containers` object inside the flask deployment definition block. For more information, You can check for documentation here
    - Set `initialDelaySeconds` to `0` seconds.
    - Here the livenessProbe check method should use httpGet and `Path` should be `/` and and readiness check method should be httpGet and `Path` should be `/?msg=Hello`
    - You can use `kubectl describe pod <pod_name>` to check the life cycle events of the pod. The output should look like



```
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  28s                default-scheduler  Successfully assigned default/flask-deployment-5489d5
b959-rgmf6 to aks-agentpool-34279051-vmss000000
  Normal   Pulled     27s                kubelet            Successfully pulled image "shivupoojar/message-board.
v3" in 703.007257ms
  Normal   Pulled     24s                kubelet            Successfully pulled image "shivupoojar/message-board.
v3" in 711.598921ms
  Warning  BackOff    20s (x2 over 21s)  kubelet            Back-off restarting failed container
  Normal   Pulling    8s (x3 over 28s)   kubelet            Pulling image "shivupoojar/message-board.v3"
  Normal   Created    7s (x3 over 27s)   kubelet            Created container flask
  Normal   Started    7s (x3 over 27s)   kubelet            Started container flask
  Normal   Pulled     7s                 kubelet            Successfully pulled image "shivupoojar/message-board.
v3" in 715.114546ms
  Warning  Unhealthy  6s (x3 over 26s)   kubelet            Liveness probe failed: Get "http://10.244.0.80:5000/"
: dial tcp 10.244.0.80:5000: connect: connection refused
  Warning  Unhealthy  6s (x8 over 26s)   kubelet            Readiness probe failed: Get "http://10.244.0.80:5000/
?msg": dial tcp 10.244.0.80:5000: connect: connection refused
```

- Deliverable: Take the screen shot of the **Events** section from the command output.

- Increase the `initialDelaySeconds` to `8` seconds. Answer the question:
  - What happens after increasing the `initialDelaySeconds` time limit, and why did liveliness and readiness errors occur during the first scenario?

- You can refer Flask example here: https://github.com/sebinxavi/kubernetes-readiness

Deliverables:

- 6 screenshots from:
    1. Task 12.1
    2. Task 12.2.1
    3. Task 12.3.5
    4. Task 12.4 (two screenshots)
    5. Task 12.5
- 5 .yml files:
    1. first_pod.yaml
    2. first_deployment.yml
    3. replicaset.yml
    4. message-board.yml
    5. data-volume.yml

Task	Lab 12 - Kubernetes
Current submission	ZIP (11:21 27.04.2023) If your homework consists of multiple files (for example HTML + CSS + JS) it should be archived before submitting.
File	Choose File No file chosen
Comment	
	Submit

Potential issues and solutions

- NB! You may sometimes get an error if your Flask application creates database tables only when the container starts. When the database is re-created (as a result of recreating the Pod, database messages table might no longer exist)
    - Simply recreate the flask deployment or scale it to 0 and back to 2.