K8S (10) Other Topic

Annotation:

Adding key-value to annotation to add additional information. This is not for identifying or object selection.
(Usually adding Release IDs, PR numbers, git branch, Contributor,...)
Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver
  annotations:
    description: Deployment based PoC dates 2nd Mar'2022
```

The description key we add will be added into description.

```
$ kubectl describe deployment webserver
Name: webserver
Namespace: default
CreationTimestamp: Fri, 25 Mar 2022 05:10:38 +0530
Labels: app=webserver
Annotations: deployment.kubernetes.io/revision=1
description=Deployment based PoC dates 2nd Mar'2022
```

Quota and Limits Management:

When multiple users sharing a cluster, we don't want one single user use all resources
K8S admin can use ResourceQuota API to setup the resource limit for each namespace.
Example:

Compute Resource Quota: limit the computing resources (CPU, memory, etc.)
Storage Resource Quota: limit the storage resources (PersistentVolumeClaims, requests.storage, etc...)

Object Count Quota: restrict the number of objects of a given type (pods, ConfigMaps, PersistentVolumeClaims, ReplicationControllers, Services, Secrets, etc.).

K8S admin can use LimitRange to limit resources allocation to pods and containers in a namespace.

Example:

Set compute resources usage limits per Pod or Container in a namespace.

Set storage request limits per PersistentVolumeClaim in a namespace.

Set a request to limit ratio for a resource in a namespace.

Set default requests and limits and automatically inject them into Containers' environments at runtime.

AutoScaling:

Dynamic scaling, which adds or removes objects from the cluster based on resource utilization, availability, and requirements.

Example:

Horizontal Pod Autoscaler (HPA): HPA can automatically adjusts the number of replicas in a ReplicaSet, Deployment or Replication Controller based on CPU utilization.

Vertical Pod Autoscaler (VPA): VPA automatically sets Container resource requirements (CPU and memory) in a Pod and dynamically adjusts them in runtime, based on historical utilization data, current resource availability and real-time events.

Cluster Autoscaler: CA automatically re-sizes the Kubernetes cluster when there are insufficient resources available for new Pods expecting to be scheduled or when there are underutilized nodes in the cluster.

Jobs and CronJobs:

Job is also an object, job will create (and maintain) pod to make sure the given task is completed.

CronJobs can perform Jobs at scheduled times/dates.

StatefulSets:

Container is pretty fit for stateless application. If this is the case, using deployment + RS to manage pod is sufficient.

If you need to deploy stateful application, use StatefulSet to assigned unique ID to pods. Any re-scheduling will not change this pod's name/hostname.
The storage is persistent (pod can still access the same data after re-scheduling).
When you deploying or scaling, you can setup the order for pods.
Restrict:

Storage need to be PVC (PersistentVolumeClaim).
Need additional Headless Service to ensure the network ID for pods.

Custom Resources:

In k8S, a **resource** is an API endpoint which stores a collection of API objects. For example, a Pod resource contains all the Pod objects.
We can create our own **custom resources**. (Need to install **custom controller**)
Two ways to add custom resource:
Custom Resource Definitions (CRDs): Defining a CRD object creates a new custom resource with a name and schema that you specify.
API Aggregation: Write a subordinate API Service under the K8S API service.

Kubernetes Federation:

With Kubernetes Cluster Federation we can cross-cluster manage resource from other cluster.

Security Contexts and Pod Security Admission:

Security Contexts allow us to set Discretionary Access Control (define specific privileges and access control settings for Pods and Containers).
Pod Security Admission can apply security settings to multiple Pods and Containers cluster-wide

Network Policies:

Network Policies are sets of rules which define how Pods are allowed to talk to other Pods and resources inside and outside the cluster.
The Network Policy API resource specifies **podSelectors**, Ingress and/or Egress **policyTypes**, and rules based on source and destination **ipBlocks** and **ports**.

Monitoring, Logging, and Troubleshooting:

Monitoring:

Metrics Server is a cluster-wide aggregator of resource usage data.

Prometheus can also be used to scrape the resource usage.

Logging:

Kubernetes does not provide cluster-wide logging by default. 3PL

Elasticsearch + Fluentd with custom configuration as an agent on the nodes.

Troubleshooting:

```
$ kubectl logs pod-name // check the pod log
$ kubectl logs pod-name -c container-name // pod log for specific container
$ kubectl logs pod-name -c container-name -p // the log of the last failed container

// following command run custom command in running container
$ kubectl exec pod-name -- ls -la /
$ kubectl exec pod-name -c container-name -- env
$ kubectl exec pod-name -c container-name -it -- /bin/sh

// following command check the failed pod events
$ kubectl get events
$ kubectl describe pod pod-name
```

Helm:

As your K8S manifests grows, it could be difficult to manually deploy all resources (too many Deployment, Service, PV, PVC, ...)

We can templatize these resources and wrap them into Chart. Chart can be store and install through repo.

Helm is a package manager (like **yum** and **apt** for Linux) for Kubernetes, which can install/update/delete those Charts in the Kubernetes cluster.

Charts submitted for Kubernetes are available here.

Additional information about helm Charts can be found here.

Service Mesh:

Service mesh manages network traffic between services. It is another solution to expose your endpoint and manage communication between pods.

https://dev.to/thenjdevopsguy/kubernetes-ingress-vs-service-mesh-2ee2

Application Deployment Strategies:

The rolling update and rollback allow us to update application of he deployment, creating new RS and new pods. However, the Service is still exposing new and old

pods.

You can reply on following strategies to setup the Service so the Service only includes specific pods.

(1) **Canary strategy**: Runs two application releases simultaneously managed by two independent Deployment controllers, both exposed by the same Service. The users can manage the amount of traffic each Deployment is exposed to by separately scaling up or down the two Deployment controllers, thus increasing or decreasing the number of their replicas receiving traffic.

(2) **Blue/Green strategy**: Runs the same application release or two releases of the application on two isolated environments, but only one of the two environments is actively receiving traffic. This strategy requires two independent Deployment controllers, each exposed by their dedicated Services

Demo: Application Deployment Strategy—Canary:

Reuse the blue-app and green-app (app-blue-shared-vol.yaml and app-green-shared-vol.yaml)

For now, they are both exposed by their own service.

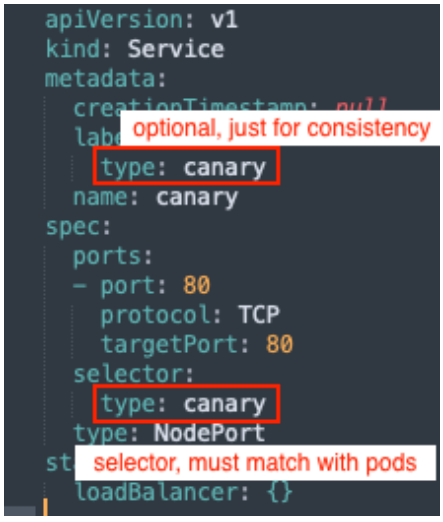
```
$ minikube service list
|---------------|-------------------------------------|--------------|---------------
-------------|
| NAMESPACE | NAME | TARGET PORT | URL |
|---------------|-------------------------------------|--------------|---------------
-------------|
| default | blue-app | 80 | http://192.168.105.6:31571 |
| default | green-app | 80 | http://192.168.105.6:30406 |
| default | kubernetes | No node port | |
| ingress-nginx | ingress-nginx-controller | http/80 | http://192.168.105.6:30479 |
| | | https/443 | http://192.168.105.6:32242 |
| ingress-nginx | ingress-nginx-controller-admission | No node port | |
| kube-system | kube-dns | No node port | |
|---------------|-------------------------------------|--------------|---------------
-------------|

$ kubectl get deploy,svc,pods,rs,ds
NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/blue-app 1/1 1 1 5m21s
deployment.apps/green-app 1/1 1 1 61s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
```

```
service/blue-app NodePort 10.100.52.49 <none> 80:31571/TCP 48s
service/green-app NodePort 10.97.156.22 <none> 80:30406/TCP 41s
service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 16h

NAME READY STATUS RESTARTS AGE
pod/blue-app-85db4f7f75-6l76q 2/2 Running 0 5m21s
pod/green-app-86dfcf4f48-nh6tk 2/2 Running 0 61s

NAME DESIRED CURRENT READY AGE
replicaset.apps/blue-app-85db4f7f75 1 1 1 5m21s
replicaset.apps/green-app-86dfcf4f48 1 1 1 61s
```

We will introduce a common service that we'll be able to route traffic to both sets of the pods of the blue application deployment and of the green application deployment.

```
// generate a template, name: canary
$ kubectl expose deploy blue-app --name canary --type=NodePort --dry-run=client -o
yaml > canary-svc.yaml

// we need some common label to select both pods
% kubectl describe pods blue-app-85db4f7f75-6l76q | grep -i labels -A2
Labels: app=blue-app
pod-template-hash=85db4f7f75
type=canary
chenyang@ChenYangs-MBP myKubernetes % kubectl describe pods green-app-86dfcf4f48 |
grep -i labels -A2
Labels: app=green-app
pod-template-hash=86dfcf4f48
type=canary
```

Update yaml file



```
$kubectl apply -f canary-svc.yaml
service/canary created

$ minikube service list
|---------------|-------------------------------------|--------------|---------------
-------------|
| NAMESPACE | NAME | TARGET PORT | URL |
|---------------|-------------------------------------|--------------|---------------
-------------|
| default | blue-app | 80 | http://192.168.105.6:31571 |
| default | canary | 80 | http://192.168.105.6:30630 |
| default | green-app | 80 | http://192.168.105.6:30406 |
|---------------|-------------------------------------|--------------|---------------
-------------|

// The canary service forward to request to blue-app and green-app, approximately
50/50
$ curl http://192.168.105.6:30630
Welcome to GREEN App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to GREEN App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to GREEN App!
$ curl http://192.168.105.6:30630
Welcome to GREEN App!
```

> Conventionally, the blue version is for table and the green is not production-ready.
> Or for some reason you need to traffic to green to be reduced.
> The the idea of canary strategy is to have multiple replicas of the first application, the stable one, blue-app in our case.
> And then the fewer replica for green-app to minimize the chances of green receiving so much traffic.

```
$ kubectl scale deploy blue-app --replicas=5
deployment.apps/blue-app scaled

$ kubectl get deploy
NAME READY UP-TO-DATE AVAILABLE AGE
blue-app 5/5 5 5 14m
```

```
green-app 1/1 1 1 9m40s

// now we hit blue more than green
$ curl http://192.168.105.6:30630
Welcome to GREEN App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to GREEN App!
$ curl http://192.168.105.6:30630
Welcome to BLUE App!
$ curl http://192.168.105.6:30630
Welcome to GREEN App!
```