07/07/24

Application Life Cycle Management
Rolling Update and Rollbacks - Part 2

Doing this way will result in the deployment definition file a different configuration.

The difference between the recreate and rolling-update strategies can also be seen when the deployments are viewed in detail.

"kubectl describe deployment 'deployment name'"

When a deployment is created, the old deployment is scaled down to '0'. See in the message column of the 'event' section of the command mentioned above. Then the replica set scale up to the mentioned number.

However, when the rolling update is used, the old replica set was scaled down one at a time, simultaneously scaling up the new replica set one at a time.

07/07/24

Upgrade

When a deployment is created, say to deploy five replicas, it first creates a replica set automatically, which in turn creates the number of pods required to meet the number of replicas. When the app is upgraded, the Kubernetes deployment object creates a new replica-set under the hood and starts deploying the containers there. At the same time, taking down the pods in the old replica set, following a rolling update strategy. This can be seen ~~when~~ using the following command-

```
kubectl get replicasets
```

The old replica-set with 0 pods and the new replica-set with 5 pods.

Kubernetes deployment allows to rollback to a previous ~~version~~ revision. To undo a change-

07/07/24

kubectl rollout undo 'deployment name'

The deployment will then destroy the pods in the new replica set, and bring the older ones up in the old replica-set.

Again "kubectl get deployments 'deployment name'" will show the old deployment has now the desired number of pods.

Rolling Update

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
```

(Same level as strategy)

```
selector:
  matchLabels:
    app: my-app
template:
  metadata:
    labels:
      app: my-app
  spec:
    containers:
    - name: my-container
      image: my-image:latest
      ports:
      - containerPort: 80
```

07/07/24

~~Rolling~~ Recreate

~~kind: Deployment~~
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: my-app

  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: my-image:latest
        ports:
        - containerPort: 80

07/07/24

Commands and Arguments in kubernetes

The dockerfile has 'ENTRYPOINT' and 'CMD' command. The entrypoint is the command that is run at startup. and the 'CMD' is the default parameter passed the command. With the "args" option in the ~~'pod~~ pod definition file, it overrides the 'CMD' instruction in the dockerfile.

To override the entrypoint command use 'command' option in the pod definition file. The 'command' field corresponds to entrypoint instruction in the dockerfile.

Ubuntu dockerfile

```
FROM Ubuntu
ENTRYPOINT ~~["sete~~ ["sleep"]
CMD ["5"]
```

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      command: ["sleep2.0"]
      args: ["10"]
```

07/07/24

```
apiVersion: v1
kind: Pod
metadata:
  name: custom-command-pod

spec:
  containers:
  - name: my-container
    image: Ubuntu
    command: ["sh", "-c"]
    args: ["echo Hello, Kubernetes! && sleep 3600"]
```

Environment variables in Kubernetes

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
  - name: ~~ubuntu~~ simple-webapp-color
    image: simple-webapp-color
    ports:
    - containerPort: 8080
    (same level as 'ports')
    env:
    - name: APP_COLOR
      value: pink
```

07/09/24

Config Maps

When there is a lot of pod definition files it becomes difficult to manage the environment data stored within the query files. These information can be taken out of the pod definition files and manage it centrally using configuration Maps.

Configmaps are used to pass configuration data in the form of key-value pairs in Kubernetes.

```
kubectl create configmap \ app-config --from-literal=APP_COLOR=blue
```

```
kubectl create configmap <config-name> --from-file=<path-to-file>
```

config-map.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: app-config
data:
  APP_COLOR: blue
  APP_MODE: prod
```

07/07/24

Use different config maps for different apps.

```
kubectl get configmaps
```

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
    - containerPort: 8080

    envFrom:
    - configMapRef:
        name: app-config
```

08/07/24

Secrets

Secrets are used to store sensitive information like passwords or keys. They are stored in a encoded or hashed format. As with configMap, there are two steps involved in working with secrets. First, create the secret and second inject them into pod.

There are two ways to create a secret - the imperative way without using a secret definition file and the declarative way by using a secret definition file.

kubectl create secret generic <secret-name> --from-literal=<key>=<value>

kubectl create secret generic app-secret --from-literal=DB_HOST=mysql

For additional key-value pairs, use '--from-literal' multiple times. However this could get complicated when there is too many secrets to pass in. Another

06/07/24

through
way to input the secret data is through a file.

```
kubectl create secret generic app-secret
  --from-file = app_secret.properties
```

For the declarative approach, create a definition file.

secret-data.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_HOST : mysql
  DB_User : root
  DB_PASSWORD : password
```

But the values are in plain text, so they need to be in encoded format.
To convert into encoded format

```
echo -n 'mysql' | base64
```

Output: bXlzcWw=

08/07/24

kubectl get secrets

kubectl describe secrets

To see the secrets with their values

kubectl get secret app-secret -o yaml

To decode the 'base64' encode

echo -n 'bXlzcWw=' | base64 --decode

Output: mysql

Inject the encoded secret into the pod definition file-

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-weapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
    - containerPort: 8080
    envFrom
    - secretRef:
        name: app-secret
```

(Same level as ~~envFrom~~ name)

08/07/24

All the secret files can be injected in a single environment variables or inject the whole secret as files in a volume.

secrets in a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-pod
spec:
  containers:
  - name: mycontainer
    image: nginx
    volumeMounts:
    - name: mysecretvolume
      mountPath: /etc/secret-volume
  volumes:
  - name: mysecretvolume
    secret:
      secretName: mysecret
```

08/07/24

Note on Secrets

- Secrets are not encrypted. Only encoded.
→ Don't check-in secret objects to SCM along with code.
→ secretes are not encrypted in 'etcd'.
  ↳ Enable encryption at rest
→ Anyone able to create pods/deployments in the same namespace can access the secrets.
  ↳ Configure least-privilege access to Secrets-RBAC
→ Consider third-party secrets store providers
  AWS provider, Azure provider, GCP provider, Vault provider.

~~08/07/24~~

Additional Resource

youtube.com/watch?v=MTnQWgMxnRI

The way kubernetes handles secrets, such as:
→ A secret is only sent to a node if a pod on that node requires it.
→ Kubelet stores the secret into a tmpfs so that the secret is not written to disk storage
→ Once the pod that depends on the secret is deleted,

08/07/24

kubelet will delete its local copy of the secret data as well.

There are other better ways of handling sensitive data like passwords in Kubernetes, such as using tools like Helm secrets, and HashiCorp Vault.

Multi Micro-container Pods

The idea of decoupling a large monolithic application into sub-components known as microservices enables to develop and deploy a set of independent, small, and reusable code. This architecture can then helps to scale up, down as well as modify each service as required - as opposed to modifying the entire app.

However, at times it may be necessary two services to work together, such as a web server and a logging service. One agent instance per web server instance needs to be paired together that can scale up and down together. Don't merge and bloat the code of the two services,

08/07/24

as each of them target different functionalities. The requirement here is the two functionality to work together.

For these requirements, multi-container pods are necessary that share the same life cycle. These share the same network space which means they can refer to each other as localhost, and they have access to the same storage volumes.

This way, no need to establish volume sharing or services between the pods to enable communication between them.

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
  image: simple-webapp
  ports:
  - containerPo
```

```
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
    - containerPort: 8080
  - name: log-agent
    image: log-agent
```

08/07/24

Init Containers

In a multi-container pod, each container is expected to run a process that stays alive as long as the pod's lifecycle.

For example, in the multi-container pod, there is a web app and logging agent, both the containers are expected to stay alive at all times.

The process running in the log agent container, is expected to stay alive as long as the web-application is running. If any of them fail, the pod restarts. But, a process runs in a container must be completed. For example, a process that pulls a code or binary from a repository that will be used by the main web app.

That is a task that will be run only one time when the pod is first created, or a process that waits for an external service or db to be up before the actual app starts.

08/07/24

Thats where initContainer is configured in a pod like all other containers, except that is specified inside a 'initContainers' section.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp

spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'git clone ;']
```

When a pod is first created the initContainer is run, and the process in the initContainers must run to a completion before the real ~~conta~~

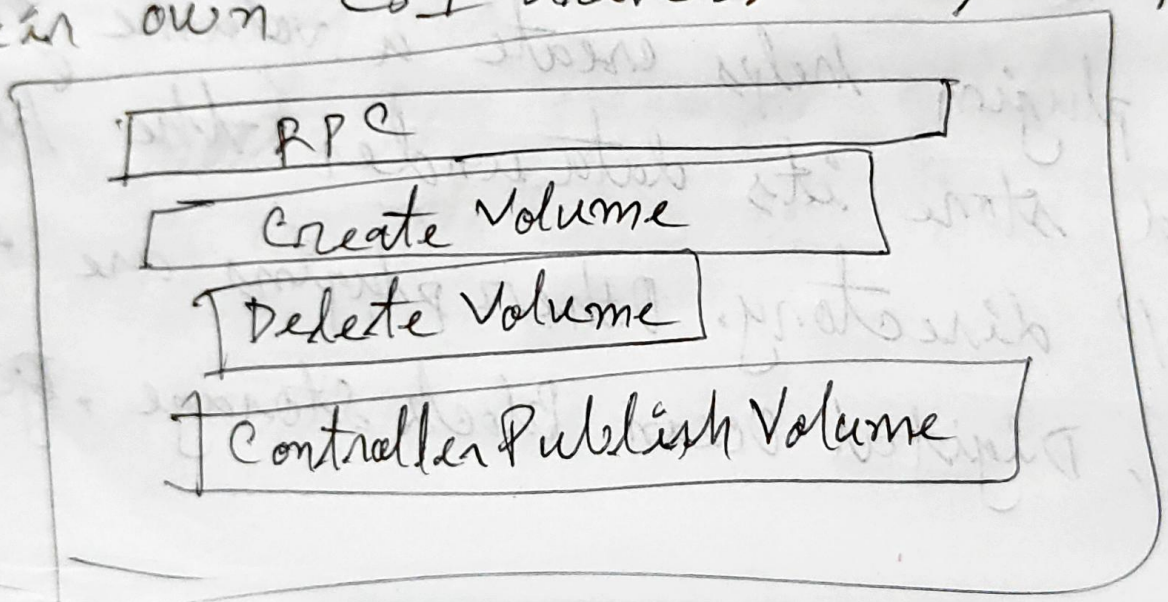
08/07/24

container hosting the app starts.

Such ~~a~~ multiple 'initContainers' can be configured. In that case, each init container is run one at a time in sequential order. If any of the initContainers fail to complete, Kubernetes restarts the pod repeatedly until the init-container succeeds.

Container Storage Interface

The container runtime interface (CRI) is a standard that defines how an orchestration solution like Kubernetes would communicate with container runtimes like docker. To extend support for different docker networking solution, the CNI was introduced. Every vendor has their own CSI driver.



08/07/24

Volumes

The pods created in Kubernetes are transient in nature. When a ~~pods~~ pod is created to process data and then deleted, the data processed by it gets deleted as well. For this, a volume to the pod is attached. The data generated by the pod is now stored in the volume, even after the pod is deleted, the data remains.

There is a single node Kubernetes cluster, create a pod that generates a random between 1 and 100 and writes that to a file, then gets deleted along with the random number.

```
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh", "-c"]
    args: ["shuf -i 0-100 -n 1 >>/opt/number.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume
```

08/07/24

To retain the number generated by the pod - create a volume and a volume needs storage. Once the volume is created, to access it from a container, the volume is mounted to a directory inside the container.

Use the ~~volume mou~~ 'volumeMounts' field in each container to mount the 'data-volume' to the directory '/opt' within the container. The random number will now be written to '/opt' mounted inside the container which happens to be 'data-volume' which is infact '/data' directory on the host. When the pod gets deleted, the file with the random number still lives on the host.

(same level as containers)

```
volumes:
- name: data-volume
  hostpath:
    path: /data
    type: Directory
```

08/07/24

The way the volume is configured here is not recommended for use in a multinode cluster. This is because the pods would use the '/data' directory on all the nodes, and expect all of them to be the same and have the same data since they are on different servers. They are ~~not~~ infact not the same unless configured some kind of ~~replicated~~ external replicated cluster storage solution. Kubernetes supports several types of standard storage solutions such as NFS, glusterFS, cephFS, AWS EBS, Azure DiskV etc.

09/07/24

Persistent volumes

~~Usha~~ As per the previous method, in a large environment with a lot of users deploying a lot of pods the users would have to configure storage everytime for each pod. Everytime, any change to be made, the users would have to make them on all of his pods. Persistent volume provides centralized solution.

A persistent volume is a cluster wide pool of storage volumes configured by an administrator

09/07/24

to be used by users deploying apps on the cluster. The users can now select storage from this pool using persistent volume claims.

pv-definition.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 1Gi
  hostPath:
    path: /tmp/data
```

The supported 'accessModes' are 'ReadOnlyMany', 'ReadWriteOnce' and 'ReadWriteMany'.

```
kubectl create -f pv-definition.yaml
```

```
kubectl get persistentvolume
```

09/07/24

Persistent Volume Claims

Persistent volume and persistent volume claims are two separate objects in Kubernetes namespace. An admin creates a set of persistent volumes and a user creates persistent volume claims to use storage. Once the persistent volume claims are created, Kubernetes binds the persistent volumes to 'claims' based on the request and properties set on the volume. Every persistent volume claims is bound to single persistent. During the binding process Kubernetes tries to find a persistent volume that has sufficient capacity as requested by the 'claim' and anyother properties such as access modes, volume modes, storage class etc. However, if there are multiple possible matches for a single claim and I would like to specifically use a ~~partial~~ particular volume & labels and selectors still could be used to bind to the right volumes. A smaller claim may get bound to a larger volume if all other

09/07/24

criteria matches and there are no better options. There is one-to-one relationship between claims and volumes, so no other claims can utilize the remaining capacity in the volume.

If there are no volumes available, the persistent volume claim will remain in a pending state until newer volumes are made available to the cluster once newer volumes are available. Once newer volumes are available the 'claim' would automatically be bound to the newly available volume.

pvc-definition.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    request:
      storage: 500Mi
```

09/07/24

```
kubectl apply -f pvc-definition.yaml
```

```
kubectl get persistentvolumeclaim
```

When the claim is created, Kubernetes looks at the volume created previously. The 'accessMode' match. The capacity requested is 500MB but the volume is configured with 1GB of storage. Since there are no others volumes available, the persistent volume claim is bound to persistent volume.

Delete PVCs

```
kubectl delete persistentvolumeclaim myclaim
```

If the admin can choose what will happen when the 'claim' is deleted. By default it is set to retain meaning the persistent volume will remain until it is manually deleted by the admin. It is not available for reuse by any other 'claim' on it can be deleted automatically. This way as soon as the claim is deleted, the volume will be deleted as well, thus freeing up storage on the end storage

09/07/24

device, ~~or~~ A third option to recycle. In this case, in the data volume will be scrubbed before making it available to other claims.

we in pod definition file

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers
  - name: myfrontend
    image: nginx
    VolumeMounts:
    - mountPath: "/var/www/html"
      name: mypd
  Volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim
```

09/07/24

deployment definition file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myfrontend
        image: nginx
        volumeMounts:
        - mountPath: "/var/www/html"
          name: mypd
      volumes:
      - name: mypd
        persistentVolumeClaim:
          claimName: myclaim
```

09/07/24

Storage class

With storage classes, provisioner can be defined that can automatically provision storage and attach them to a pod when a claim is made. That's called dynamic provisioning of volumes. Create a storage class definition file, the pod is in the ~~goo~~ google cloud

sc-definition.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: google-storage
provisioner: kubernetes.io/gce-pod
```

(same level as provisioner)

```
parameters:
  type: pd-standard
  replication-type: none
```

So now the pv definition is not needed, because the pv and any associated storage is going to be created automatically when the storage class is created. The pvc definition file -

09/07/24

pvc-definition.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: google-storage
  resources:
    requests:
      storage: 500Mi
```

That's how the pvc knows which storage class to use. Next time, if pvc is created, the storage class associated with it uses the defined provisioner to provision a new disk with the required size. Then creates a persistent volume and then binds the pvc to that volume. With different types of provisioner, pass different types of parameters.

—o—