



KUBERNETES

(Part 3)

Prepared by: Aamir Pinger



fb.com/AamirPingerOfficial



linkedin.com/in/AamirPinger



github.com/AamirPinger

SERVICE



Service

- When we create pod, our application is not accessible to outer world
- We then used `kubectl port-forward` to forward the pod
- Port-forward command solve our problem by making single specified pod to outer world
- Think of the situation where you have hundreds of pods and each pod have multiple copies
- In the situation like that port-forward is not the best option to use
- Instead we create a resource provided by kubernetes called Service

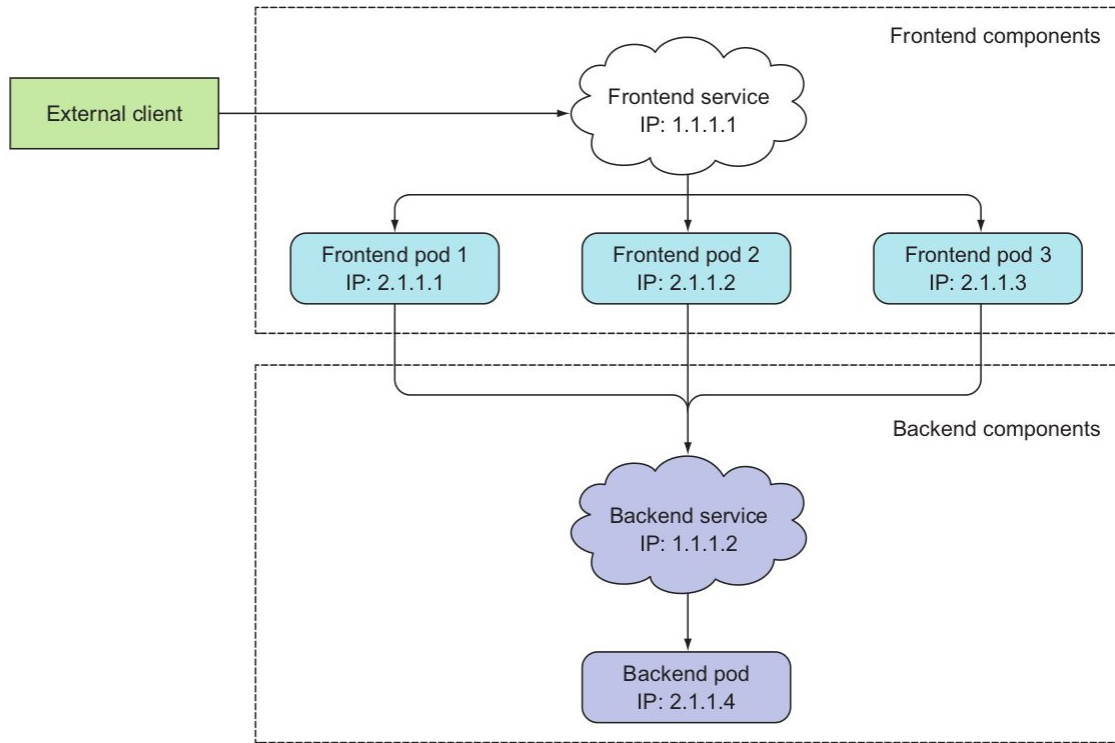


Service

- We have learnt that every pod has its own IP address
- We have learnt that Kubernetes can group pods providing at a single static IP address
- Service resource is the one which is used to create a single, constant point of entry to a group of pods
- Each service resource has an IP address and port that never change while the service resource exists
- By using that IP and Port provided by service resource, we can access our application
- Even if pod moves around the cluster, service IP doesn't change and you get diverted to the new location where the pod is rescheduled



Service



- Both Frontend and backend app components are exposed with Kubernetes Service resource
- That means, even frontend and backend Pods IP address changes in case of relocation or recreation, user can externally or app can internally communicate with each other without any hazzel



Service

- Service resource has many types, few of them are
 - Cluster IP
 - Node Port
 - Load Balancer
 - External Name



Service

- **ClusterIP (default)**
 - Exposes the Service on an internal IP in the cluster
 - This type makes the Service only reachable from within the cluster
 - We can check it by `minikube ssh` and then `clusterIP:port`
- **NodePort**
 - Exposes the Service on the same port of each selected Node in the cluster
 - Port range is 30000 to 32767
 - Makes a Service accessible from outside the cluster using `<NodeIP>:<NodePort>`

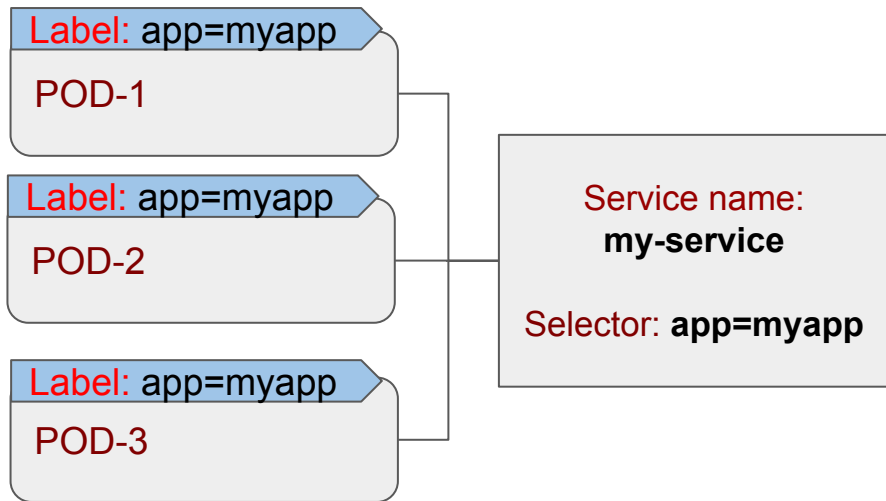


Service

- **LoadBalancer**
 - Creates an external load balancer for traffic
 - Assigns a fixed, external IP to the Service
- **ExternalName**
 - To create a service that serves as an alias for an external service
 - Let's say your database is on AWS and it has the following URL `test.database.aws.com`
 - By create externalName you can have let's say my-db diverted to `test.database.aws.com`



Service



my-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - port: 8080
      targetPort: 80
  selector:
    app: myapp
  type: LoadBalancer
```



Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-ext-svc
spec:
  type: ExternalName
  externalName: ap12.mlab.com
```

- For example you want to make a service of a mongodb database which is hosted at mlab.com
- By creating external service you don't need to use your mlab path to use your database, instead you will use service name
- Now we can use following

```
mongodb://<dbuser>:<dbpassword>@my-ext-svc:<port>/dev
```



Service

- Another way to create service

```
aamir@ap-linux:~$ kubectl expose rs myrs --port=8000 --target-port=80
--type=LoadBalancer --name my-svc-lb --selector=app=rsexample
service/my-svc-lb exposed
```

```
aamir@ap-linux:~$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-svc-lb	LoadBalancer	10.111.102.249	<pending>	8000:32117/TCP	2m
...					

HEALTH CHECK



HEALTH CHECK

- One of the main benefits of using Kubernetes is to keep our containers running somewhere in the cluster
- But what if one of those containers dies? What if all containers of a pod die?
- If an app container crashes because of a bug in your app, Kubernetes will restart your app container automatically
- But what about those situations when your app stops responding because it falls into an infinite loop or a deadlock?
- Kubernetes provides a way to check the health of your application

LIVENESS PROBES



Liveness Probes

- Pods can be configured to periodically check an application's health from the outside and not depend on the app doing it internally
- You can specify a liveness probe for each container in the pod's specification
- Kubernetes will periodically execute the probe and restart the container if the probe fails
- **IMPORTANT POINT:** "Container is restarted" means old one is killed and a completely new container is created — it's not the same container being restarted again



Liveness Probes

- There are three types of probes

1. HTTP GET

- This type of probe send request on the container's IP address, a port and path you specify
- Probe is considered a failure and Container will be automatically restarted if
 - Probe receives error response code
 - Container app doesn't respond at all



Liveness Probes

2. TCP SOCKET

- TCP Socket probe tries to open a TCP connection to the specified port of the container
- If the connection is established successfully, the probe is successful
- Otherwise, the container is restarted.



Liveness Probes

3. EXEC Probe

- An Exec probe executes some commands you provide inside the container and checks the command's exit status code
- If the status code is 0, the probe is successful
- All other codes are considered failures



Liveness Probes

my-ln-exec.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: myapp-ln-exec
spec:
  containers:
  - name: myapp
    image: aamirpinger/hi
    ports:
    - containerPort: 80
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
```

my-ln-tcp.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: myapp-ln-tcp
spec:
  containers:
  - name: myapp
    image: aamirpinger/hi
    ports:
    - containerPort: 80
    livenessProbe:
      tcpSocket:
        port: 8080
```

my-ln-http.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: myapp-ln-http
spec:
  containers:
  - name: myapp
    image: aamirpinger/hi
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        port: 80
        path: /
```



Liveness Probes

- There are additional properties which can be defined with any numbers in any of liveness probe. For example:

This tells to restart the container in case of 3 consecutive failure

```
livenessProbe:
  httpGet:
    path: /
    port: 80
  failureThreshold: 3
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 1
  initialDelaySeconds: 15
```



Liveness Probes

- There are additional properties which can be defined with any numbers in any of liveness probe. For example:

Run this liveness Probe every 10 seconds

```
livenessProbe:
  httpGet:
    path: /
    port: 80
  failureThreshold: 3
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 1
  initialDelaySeconds: 15
```



Liveness Probes

- There are additional properties which can be defined with any numbers in any of liveness probe. For example:

Reset the failureThreshold counter on 1 successful response

```
livenessProbe:
  httpGet:
    path: /
    port: 80
  failureThreshold: 3
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 1
  initialDelaySeconds: 15
```



Liveness Probes

- There are additional properties which can be defined with any numbers in any of liveness probe. For example:

Response must come within 1 second. Mark failure even successful response comes after specified time in seconds

```
livenessProbe:  
  httpGet:  
    path: /  
    port: 80  
  failureThreshold: 3  
  periodSeconds: 10  
  successThreshold: 1  
  timeoutSeconds: 1  
  initialDelaySeconds: 15
```



Liveness Probes

- There are additional properties which can be defined with any numbers in any of liveness probe. For example:

Kubernetes will wait for 15 seconds to start first liveness probe

```
livenessProbe:  
  httpGet:  
    path: /  
    port: 80  
  failureThreshold: 3  
  periodSeconds: 10  
  successThreshold: 1  
  timeoutSeconds: 1  
  initialDelaySeconds: 15
```


READINESS PROBES



Readiness Probes

- We have just learned about liveness probes and how they help keep your apps healthy by ensuring unhealthy containers are restarted automatically
- Similar to liveness probes, Kubernetes allows you to also define a readiness probe for your pod
- The readiness probe is invoked periodically and determines whether the specific pod should receive client requests or not
- When a container's readiness probe returns success, it's signaling that the container is ready to accept requests



Readiness Probes

- This notion of being ready is obviously something that's specific to each container
- Same as liveness probe Kubernetes sends request to container and based on the result either successful or unsuccessful response it decides container is ready to take traffic or still getting ready for that
- Unlike liveness probes, if a container fails the readiness check, it won't be killed or restarted
- It is good practice to always add readiness probe even its a simplest app in the container



Readiness Probes

- There are three types of probes

1. HTTP GET

- This type of probe send request on the container's IP address, a port and path you specify
- Probe is considered a failure and Container will be treated as not ready and no traffic will get diverted to it



Readiness Probes

2. TCP SOCKET

- TCP Socket probe tries to open a TCP connection to the specified port of the container
- If the connection is established successfully, container will be marked as ready and it will receive traffic
- Otherwise, Kubernetes will wait and rerun the probe to check the status again



Readiness Probes

3. EXEC Probe

- An Exec probe executes some commands you provide inside the container and checks the command's exit status code
- If the status code is 0, the probe is successful
- All other codes are considered failures



Readiness Probes

my-rn-exec.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: myapp-rn-exec
spec:
  containers:
  - name: myapp
    image: aamirpinger/hi
    ports:
    - containerPort: 80
    readinessProbe:
      exec:
        command:
        - ls
        - /tmp/ready
```

my-rn-tcp.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: myapp-rn-tcp
spec:
  containers:
  - name: myapp
    image: aamirpinger/hi
    ports:
    - containerPort: 80
    readinessProbe:
      tcpSocket:
        port: 8080
```

my-rn-http.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: myapp-rn-http
spec:
  containers:
  - name: myapp
    image: aamirpinger/hi
    ports:
    - containerPort: 80
    readinessProbe:
      httpGet:
        port: 80
        path: /
```

VOLUMES



Volumes

- Container contains its own directories and files
- If for any reason kubernetes restarts any container, all files will be lost that we might created at runtime (e.g. log files)
- Volumes in kubernetes can be thought of shared directory for the containers in a Pod at a Pod level
- Pod level mean the life of that volume is dependent on Pod's life, If Pod restarts all the files will be lost
- Shared directory means all the containers of that pod can share that directory and the files in it

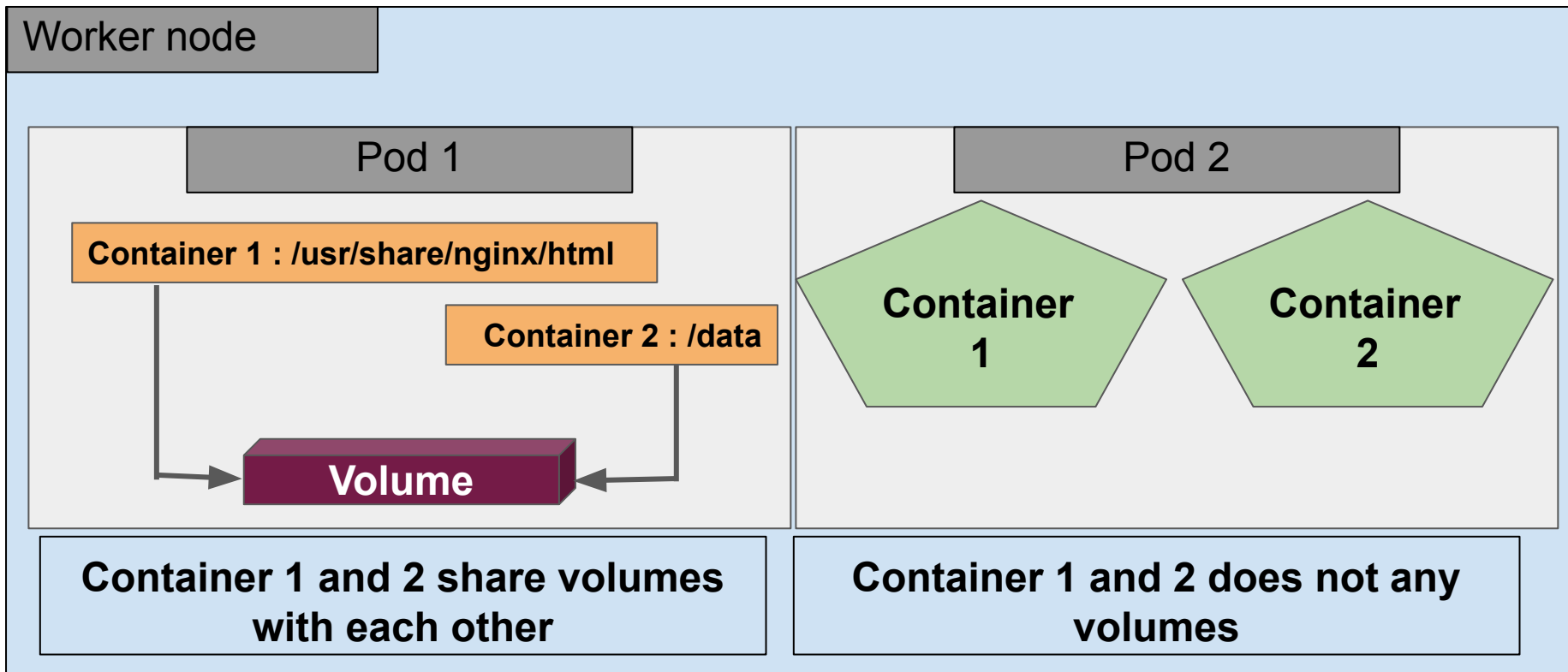


Volumes

- Volumes aren't a standalone Kubernetes object and cannot be created or deleted on their own
- Kubernetes volumes are a component of a pod and are thus defined in the pod's specification—much like containers
- A volume is available to all containers in the pod, but it must be mounted in each container that needs to access it



Volumes





Volumes

- There are many types of volumes
 - emptyDir
 - configMap , secret , downwardAPI
 - persistentVolumeClaim
 - gitRepo
 - gcePersistentDisk
 - awsElasticBlockStore
 - azureDisk



Volumes

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pod-with-vol
spec:
  volumes:
    - name: share-dir
      emptyDir: {}
  containers:
    - name: container-one
      image:
aamirpinger/logfile_nodejs
```

```
    ports:
      - containerPort: 80
    volumeMounts:
      - name: share-dir
        mountPath: /data
  - name: container-two
    image: nginx
    ports:
      - containerPort: 80
    volumeMounts:
      - name: share-dir
        mountPath: /var/c-two
```

PERSISTENT VOLUMES



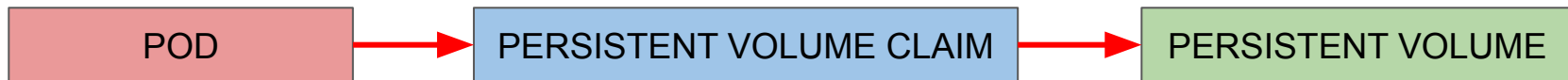
Persistent Volume

- Volumes were great as they saves us from data lost incase of container restart
- Volumes hold data at a pod level but question may be asked what if for any reason kubernetes terminates the pod e.g. rescheduling the pod
- In the case of pod termination data in the volumes will be lost
- To solve this issue Kubernetes provides us option of Persistent Volume
- Persistent Volume add a volume at a cluster level instead pod level



Persistent Volume

- We create a Persistent Volume resource in which we offer cluster level volume that can be used by any pod
- Any pod can use that Persistent Volume by using another resource Persistent Volume Claims
- Kubernetes Persistent Volumes remains available outside of the pod lifecycle
- That means volume will remain even after the pod is deleted
- This volume will be available to claim by another pod if required, and the data is retained





Persistent Volume Claim

- It is a kind of formal request from user for claiming a persistent volume
- A Persistent Volume Claim describes the amount and characteristics of the storage required by the pod
- Based on requirement from user PVC finds any matching persistent volumes and claims it
- Depending on the configuration options used for Persistent Volume resource, these PV resource can later be used/claim by other pods

PERSISTENT VOLUMES IN ACTION



Persistent Volume in Action

- We will be now creating PV, PVC, and POD
- We will be using minikube ssh to check files PV saving on cluster
- SSH, or Secure Shell, is a protocol used to securely log onto remote systems
- It is the most common way to access remote Linux servers
- For any resource yaml file there are 4 part which we write. Kind, apiVersion, metadata, and spec
- Spec part of PV carries few special things link accessModes and persistentReclaimPolicy
- Let's discuss these two before we go further writing yaml file for PV and PVC



Persistent Volume Access Modes

- Type of accessModes
 - ReadWriteOnce
 - Only a single node can mount the volume for reading and writing
 - ReadOnlyMany
 - Multiple nodes can mount the volume for reading
 - ReadWriteMany
 - Multiple nodes can mount the volume for both reading and writing
- RWO , ROX , and RWX pertain to the number of worker nodes that can use the volume at the same time, not to the number of pods!



Persistent Volume Reclaim Policy

- We have learned that depending on the configuration options used for Persistent Volume resource, these PV resource can later be used/claim by other pods
- The lifetime of a Persistent Volume is determined by its reclaim policy
- Reclaim Policy controls the action the cluster will take when a pod releases its ownership of the storage
- `persistentVolumeReclaimPolicy` tag can be used in YAML configuration file at the time of creating PV



Persistent Volume Reclaim Policy

- Reclaim Policy can be set to
 - Delete
 - Recycle
 - Retain (Default)
- If `persistentVolumeReclaimPolicy` is **Delete**
 - PersistentVolume will be deleted when the PVC is deleted but data will persist
- If `persistentVolumeReclaimPolicy` is **Recycle**
 - Volume's contents will be deleted
 - Persistent Volume will be available to be claimed again



Persistent Volume Reclaim Policy

- If `persistentVolumeReclaimPolicy` is **Retain**
 - If `persistentVolumeReclaimPolicy` not provided, Retain is default
 - Kubernetes will retain the volume and its contents after it's released from its claim
 - To make PersistentVolume available again for claims can be done by delete and recreate the PersistentVolume resource manually
 - Underlying storage can either delete or left to be reused by the next pod



Persistent Volume in Action

my-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 100M
  hostPath:
    path: /tmp/pvexmaple
persistentVolumeReclaimPolicy: Delete
```

my-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100M
  storageClassName: ""
```

my-pv-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-pod
spec:
  volumes:
    - name: pvol
      persistentVolumeClaim:
        claimName: pvc
  containers:
    - image: aamirpinger/logfile_nodejs
      name: container1
      volumeMounts:
        - name: pvol
          mountPath: /data
```


ConfigMap



ConfigMap

- ConfigMaps allow you to separate your application configurations from your application code, which helps keep your containerized applications portable
- ConfigMaps are useful for storing and sharing non-sensitive, unencrypted configuration information which you can change at runtime
- It help makes their configurations easier to change and manage, and prevents hardcoding configuration data to Pod specifications



ConfigMap

- Creating ConfigMaps from literal values on the command line

```
aamir@ap-linux:~$ kubectl create configmap cm-lit --from-literal=fname=aamir  
--from-literal=lname=pinger  
configmap/cm-lit created
```

```
aamir@ap-linux:~$ kubectl get configmap
```

NAME	DATA	AGE
cm-lit	2	59s

- You also can use **kubectl get cm**



ConfigMap

- To get configMap output in YAML file

```
aamir@ap-linux:~$ kubectl get configmap cm-lit -o yaml
```

```
apiVersion: v1
data:
  fname: aamir
  lname: pinger
kind: ConfigMap
metadata:
  creationTimestamp: "2019-07-19T08:14:20Z"
  name: cm-lit
  namespace: default
  resourceVersion: "399691"
  selfLink: /api/v1/namespaces/default/configmaps/cm-lit
  uid: a1c98d84-be6f-4a85-9868-d774d760103d
```



ConfigMap

- To get insight of configMap

```
aamir@ap-linux:~$ kubectl describe configmap cm-lit
```

```
Name:      cm-lit
Namespace: default
Labels:    <none>
Annotations: <none>
```

```
Data
```

```
====
```

```
Iname:
```

```
----
```

```
pinger
```

```
fname:
```

```
----
```

```
aamir
```

```
Events: <none>
```



ConfigMap

- To create a configMap and populate key/value data from text file

```
aamir@ap-linux:~$ kubectl create cm cm-from-file
--from-file=user.txt
configmap/cm-from-file created
aamir@ap-linux:~$ kubectl get cm cm-from-file -o yaml
apiVersion: v1
data:
  user.txt: |
    name=aamir
    surname=pinger
kind: ConfigMap
metadata:
  name: cm-from-file
...
```

user.txt

name=aamir
surname=pinger



ConfigMap

- You can add custom key instead of file name as a key

```
aamir@ap-linux:~$ kubectl create cm cm-from-file
--from-file=myuserkey=user.txt
configmap/cm-from-file created
aamir@ap-linux:~$ kubectl get cm cm-from-file -o yaml
apiVersion: v1
data:
  myuserkey: |
    name=aamir
    surname=pinger
kind: ConfigMap
metadata:
  name: cm-from-file
...
```

user.txt

name=aamir
surname=pinger



ConfigMap

- If you want to get both key/value from file

```
aamir@ap-linux:~$ kubectl create cm cm-from-env-file
--from-env-file=cm.env
configmap/cm-from-env-file created
aamir@ap-linux:~$ kubectl get cm cm-from-env-file -o yaml
apiVersion: v1
data:
  CREATEDBY: Aamir Pinger!!!
kind: ConfigMap
metadata:
  creationTimestamp: "2019-07-19T10:00:07Z"
  name: cm-from-env-file
...
```

cm.env

CREATEDBY=Aamir
PINGER!!!



ConfigMap

- ConfigMaps can also be created from a directory of files on the command line
kubectl create configmap cm-lit --from-file=/path/directory_name
- Each file name will be key and text inside that file will be value

ConfigMap's DATA AS VOLUMES



ConfigMap Data as Volumes

cm-pod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-cm
spec:
  volumes:
  - name: cm-vol
    configMap:
      name: cm-from-env-file
```

```
containers:
- name: configmap-container
  image: nginx
  volumeMounts:
  - name: cm-vol
    mountPath: /etc/config
```

```
aamir@ap-linux:~$ kubectl create -f cm-pod.yaml
pod/pod-cm created
aamir@ap-linux:~$ kubectl exec pod-cm -it -- sh
# cd /etc/config
```

```
# ls
CREATEDBY
# cat name
Aamir Pinger!!!#
```

ConfigMap's DATA AS ENVIRONMENTAL VARIABLE



ConfigMap Data as Environmental Variable

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-cm-env
  labels:
    app: cmexample
spec:
  containers:
    - name: configmap-container
      image: aamirpinger/node-app-image
      envFrom:
        - configMapRef:
            name: cm-from-env-file
```

cm-epod.yaml

```
aamir@ap-linux:~$ kubectl create -f cm-epod.yaml
pod/pod-cm-env created
aamir@ap-linux:~$ kubectl expose pod pod-cm-env
--target-port 8080 --type=LoadBalancer
service/pod-cm-env exposed
aamir@ap-linux:~$ kubectl get svc
http://192.168.99.100:<svc_port_no>
```

SECRET



Secret

- Secrets are also used to pass key/value data to application dynamically like configMaps
- Secrets are secure objects which stores sensitive data, such as passwords, OAuth tokens, and SSH keys, in your clusters
- Storing sensitive data in Secrets is more secure than plain text ConfigMaps or in Pod specifications
- Using Secrets gives you control over how sensitive data is used, and reduces the risk of exposing the data to unauthorized users
- The Secret values are Base64 encoded in Kubernetes



Secrets

- Secrets can be create from literal values on the command line

```
kubectl create secret generic secret-lit --from-literal=name=aamir  
--from-literal=surname=pinger
```

- To get list of Secret

```
kubectl get secret
```

- To describe Secret

```
kubectl describe secret <Secret name>
```




Secrets

- To get yaml of any Secret

```
kubectl get secret <Secret name> -o yaml
```

- To decode secret base64 value

```
echo YWFtaXI= | base64 -d
```

- Secret creation from single file on the command line

```
kubectl create secret generic sec-from-file --from-file=data.txt
```

```
kubectl create secret generic sec-key-from-file --from-file=<customkey>=data.txt
```

```
kubectl create secret generic sec-from-env-file --from-env-file=data.txt
```



Secret Data as Environmental Variable

sec-epod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-scr-env
spec:
  containers:
  - name: secret-container-env
    image: nginx
    envFrom:
    - secretRef:
        name: secret-lit
```

```
aamir@ap-linux:~$ kubectl create -f sec-pod.yaml
pod/pod-scr-env created
aamir@ap-linux:~$ kubectl exec pod-scr-env -it -- sh
```

```
# env
# echo $name
```



Secret Data as Volumes

scr-pod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-scr
spec:
  volumes:
  - name: scr-vol
    secret:
      secretName: secret-lit
```

```
containers:
- name: secret-container
  image: nginx
  volumeMounts:
  - name: scr-vol
    mountPath: /etc/secret
```

```
aamir@ap-linux:~$ kubectl create -f scr-pod.yaml
pod/pod-scr created
aamir@ap-linux:~$ kubectl exec pod-scr -it -- sh
# cd /etc/secret
```

```
# ls
name surname
# cat name
aamir#
```

ENVIRONMENTAL VARIABLE



Environmental Variable

- Uptil now we have seen how we can pass environmental variables to containers using envFrom Tag using configMap and Secrets
- We also can pass environmental variables hardcoded in pod specification

```
aamir@ap-linux:~$ kubectl create -f
scr-pod.yaml
pod/pod-scr created
aamir@ap-linux:~$ kubectl exec pod-scr -it -- sh
# cd /etc/secret
```

```
kind: Pod
apiVersion: v1
metadata:
  name: myapp-env
spec:
  containers:
  - name: myapp
    image: nginx
    ports:
    - containerPort: 80
    env:
    - name: AUTHOR_FIRST_NAME
      value: "Aamir"
    - name: FULL_NAME
      value:
        "$(AUTHOR_FIRST_NAME)
```

Pinger"



Environmental Variable

env-pod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: pod-env
spec:
  containers:
  - name: myapp
    image: nginx
    ports:
    - containerPort: 80
    env:
    - name: AUTHOR_FIRST_NAME
      value: "Aamir"
    - name: FULL_NAME
      value: "${AUTHOR_FIRST_NAME}
Pingr"
```

```
aamir@ap-linux:~$ kubectl create -f env-pod.yaml
pod/pod-env created
```

```
aamir@ap-linux:~$ kubectl exec pod-env -it -- sh
```

```
# echo $FULL_NAME
```

```
Aamir Pingr
```

```
# env
```

```
...
```

DEPLOYMENT



Deployment

- Till the moment we have learned
 - How to group our containerized app into pods
 - Ways to provide them with temporary or permanent storage
 - How to pass both secret and non-secret config data to them
 - Allowed pods to find and talk to each other
 - How to run a full-fledged system composed of independently running smaller components (microservices)

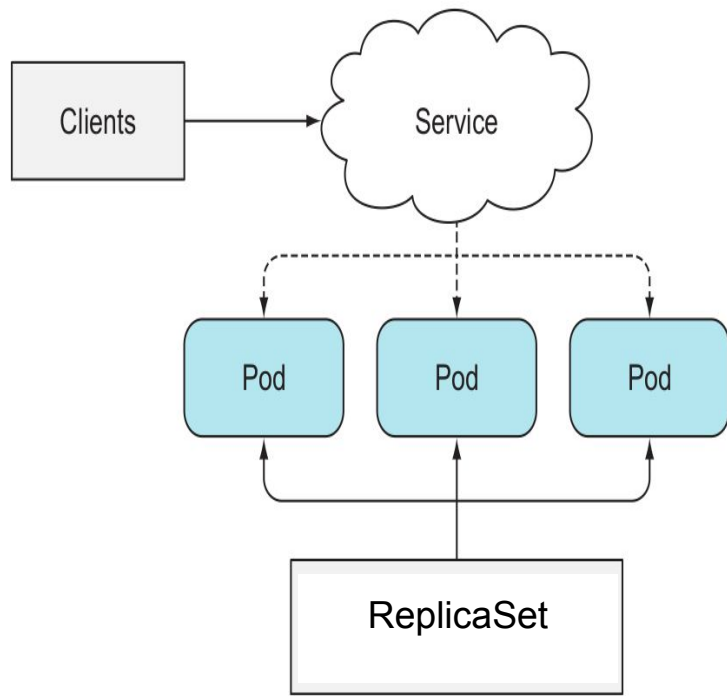
?

Is there
anything
else?

Eventually,
you're going to
want to update
your app



Deployment



- Let's say you created your pod with a container having any image e.g. `aamirpinger/helloworld:v1`
- After sometime you want you running pods to update the image with `aamirpinger/helloworld:v2`
- Because you can't change an existing pod's image after the pod is created, you need to remove the old pods and replace them with new ones running the new image



Deployment

- We have two ways of updating all those pods
 1. Delete all existing pods first and then start the new ones
 2. Start new ones and, once they're up, delete the old ones
 - a. This could be done either by adding all the new pods and then deleting all the old pods at once
 - b. Sequentially, by adding new pods and removing old ones gradually

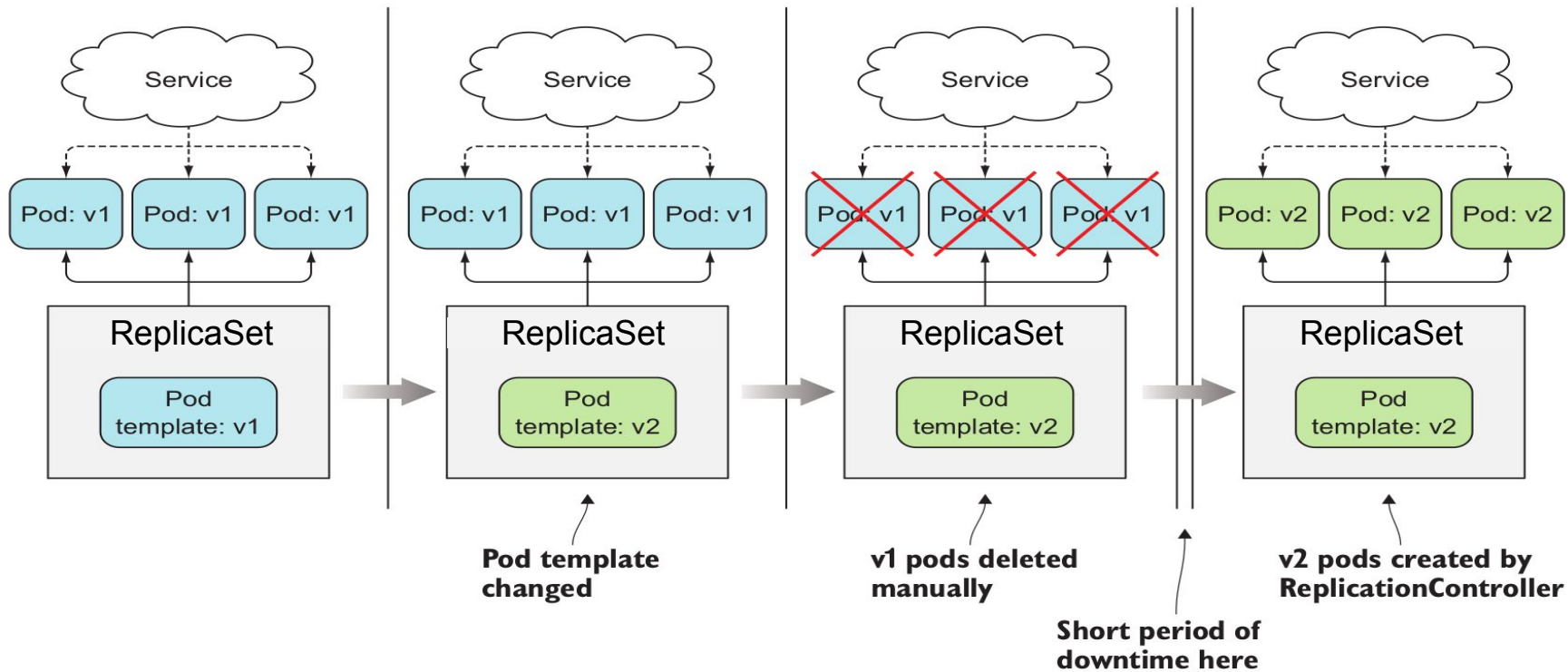


Deployment

- Both these strategies have their benefits and drawbacks
 1. The first option would lead to a short period of time when your application is unavailable
 2. The second option requires your app to handle running two versions of the app at the same time but sometimes it's not possible
 - What if you added a mandatory field in database and updates your image with new API and API in currently running pods having older version of image does not sends data for that mandatory field

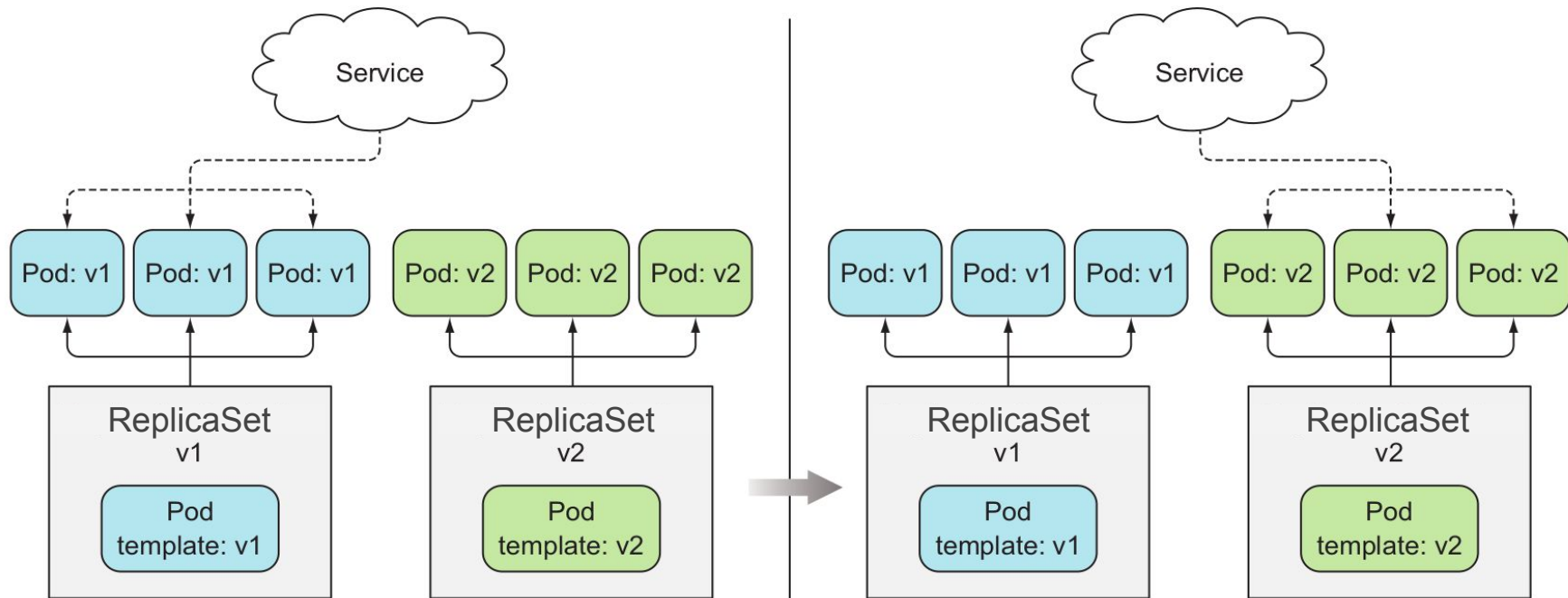


Deployment





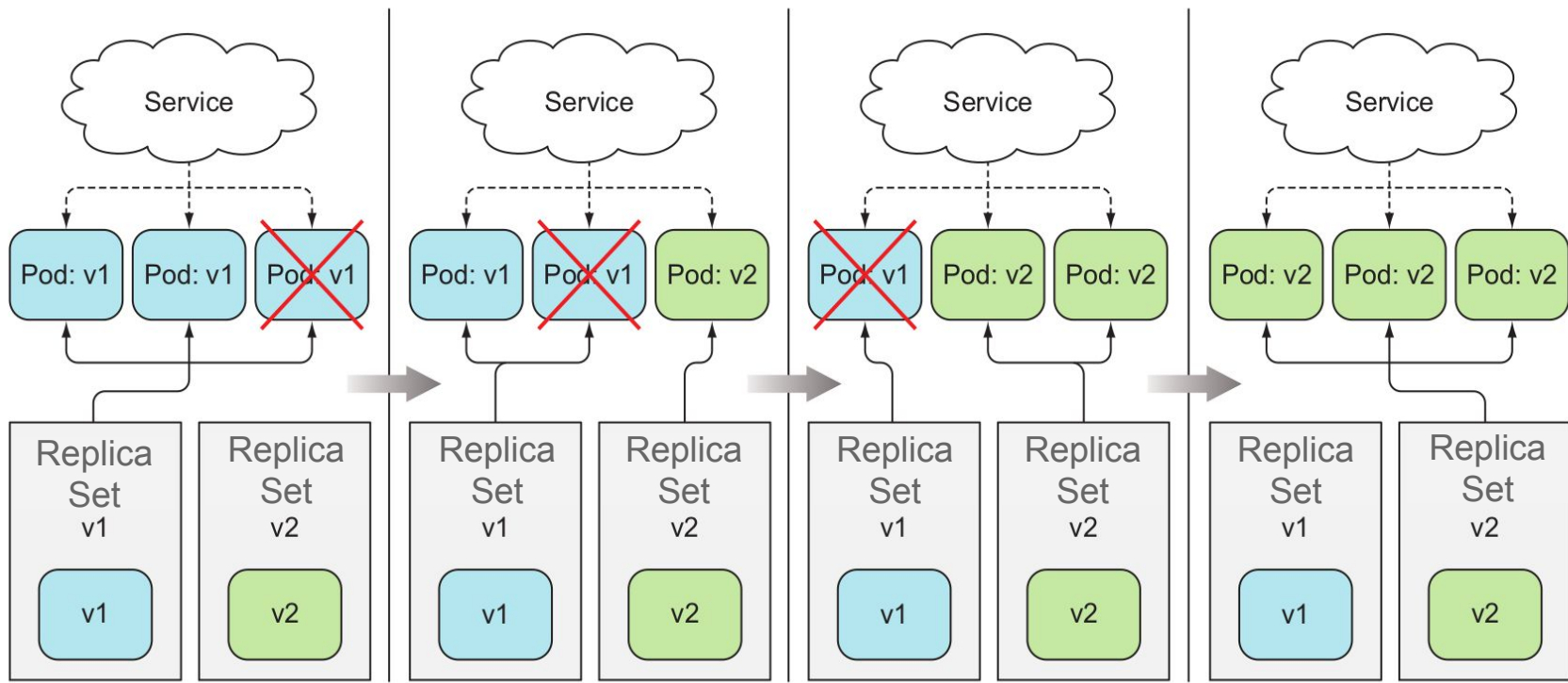
Deployment



SWITCHING A SERVICE FROM THE OLD PODS TO THE NEW ONES



Deployment





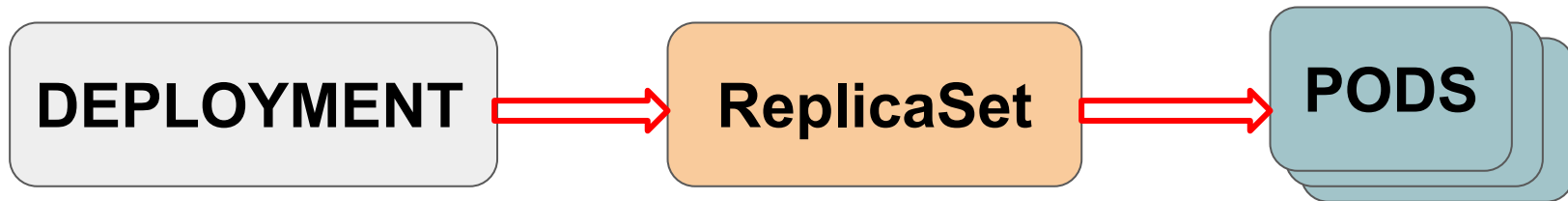
Deployment

- Last way of updating your pods app can be done by slowly scaling down the previous RS and scaling up the new one, this is called **rolling-update**
- Service's pod selector to include both the old and the new pods, so it directs requests toward both sets of pods (Problem: database example we just discussed)
- Doing a rolling update manually is laborious and error-prone
- Depending on the number of replicas, you'd need to run a dozen or more commands in the proper order to perform the update process



Deployment

- Luckily kubernetes provide us a resource which do all the update on behalf of us automatically
- A Deployment is a resource meant for deploying applications and updating them declaratively
- When you create a Deployment, a ReplicaSet resource is created underneath
- When using a Deployment, the actual pods are created and managed by the Deployment's ReplicaSets, not by the Deployment directly





Deployment

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-deploy
spec:
  replicas: 4
  template:
    metadata:
      name: deploy-pod
      labels:
        app: deploy-app
    spec:
      containers:
      - image: aamirpinger/helloworld
        name: container1
        imagePullPolicy: IfNotPresent
```

my-deploy.yaml



Deployment

```
aamir@ap-linux:~$ kubectl create -f my-deploy.yaml --record
deployment.apps/my-deploy created
aamir@ap-linux:~$ kubectl expose deploy my-deploy --type=LoadBalancer --port=80
service/my-deploy exposed
aamir@ap-linux:~$ kubectl get deploy,rs,pod,svc
...
aamir@ap-linux:~$ minikube ip
192.168.99.100
aamir@ap-linux:~$ curl 192.168.99.100:<port>
...
aamir@ap-linux:~$ kubectl set image deploy my-deploy container1=aamirpinger/hi
deployment.extensions/my-deploy image updated
aamir@ap-linux:~$ while true; do curl 192.168.99.100:port ; done
...
```



Deployment

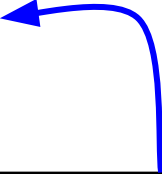
- Deployment can be set with two rolling update strategies
 - **Recreate**
 - In this strategy, all the old pods get deleted at once and then new pods get created
 - Problem: Downtime
 - **RollingUpdate (Default)**
 - Removes old pods one by one, while adding new ones at the same time, keeping the application available throughout the whole process
 - maxSurge and maxUnavailable, these two properties affect how many pods are replaced at once during a Deployment rolling update



Deployment

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-deploy-ru
spec:
  replicas: 4
  template:
    metadata:
      name: deploy-pod-ru
      labels:
        app: deploy-app-ru
    spec:
      containers:
```

```
    - image: aamirpinger/helloworld
      name: container
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
```



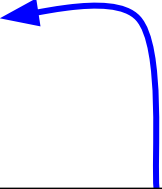
- **maxUnavailable: 0** will make sure pods unavailability cannot be less than desired replica count. In our case desired replica count is 4



Deployment

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-deploy-ru
spec:
  replicas: 4
  template:
    metadata:
      name: deploy-pod-ru
      labels:
        app: deploy-app-ru
    spec:
      containers:
```

```
    - image: aamirpinger/helloworld
      name: container
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
```




- **maxUnavailable** determines how many pod can be unavailable during the update process. Default is number nearest round off to 25% of replica count. I.e. availability of Pod will be minimum 75% of desired replica count



Deployment

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-deploy-ru
spec:
  replicas: 4
  template:
    metadata:
      name: deploy-pod-ru
      labels:
        app: deploy-app-ru
    spec:
      containers:
```

```
    - image: aamirpinger/helloworld
      name: container
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
```



- **maxSurge** determines how many pod can be created with a updated image before deleting the old pod. Default is number nearest round off to 25% of replica count. In our case total new/old all together 5 pods can be up at any particular time



Deployment

- `kubectl set image deploy my-deploy --image=aamirpinger/flag`
- `kubectl rollout pause deployment my-deploy-ru`
- `kubectl rollout resume deployment my-deploy-ru`
- `kubectl rollout status deployment my-deploy-ru`
- `kubectl rollout history deployment my-deploy-ru`
- `kubectl rollout undo deployment my-deploy-ru`
- `kubectl rollout undo deployment my-deploy-ru --to-revision=1`

KUBERNETES BEST PRACTICES



Kubernetes Best Practices

- Try avoid using latest tag instead us proper tags
- `imagePullPolicy` should be used wisely
- If the `imagePullPolicy` is `IfNotPresent` and you push updated image with the same previous tag, container will not updated as it will find image already present so won't pull again
- If the `imagePullPolicy` is `Always` it will pull image everytime pod instance will created, this will slow down the initialization phase of container



Kubernetes Best Practices

- Using multi-dimensional instead of single-dimensional labels
 - Don't forget to label all your resources, not only Pods. Make sure you add multiple labels to each resource, so they can be selected across each individual dimension
- Labels may include things like
 - The name of the application (or perhaps microservice) the resource belongs to
 - Application tier (front-end, back-end, and so on)
 - Environment (development, QA, staging, production, and so on)
 - Version
 - etc



Kubernetes Best Practices

- Making manageable container images
 - Only need files and dependencies to add in image file should be added
- Describing each resource through annotations
 - To add additional information to your resources use annotations.
 - At the least include annotation describing the resource and with contact information of the person responsible for it



Kubernetes Best Practices

- Handling application logs
 - Your apps should write to the standard output instead of files e.g `console.log("message")` in javascript
 - This makes it easy to view logs with the `kubectl logs` command
 - If a container crashes and is replaced with a new one, you'll see the new container's log. To see the previous container's logs, use the `--previous` option with `kubectl logs`

Thank you and God bless you all!



fb.com/AamirPingerOfficial



linkedin.com/in/AamirPinger



github.com/AamirPinger