Running and Managing POD

What is A POD?

Provide higher level of abstraction over a container. POD is a wrapper around the container that allows Kubernetes to manage access to the container, allocating required resources for the service/application running inside the container.

PODs are the smallest, most basic deployable objects in Kubernetes. A Pod represents a single instance of a running process in your cluster.

PODs contain one or more *containers*, such as Docker containers.

POD can be seen as a process running somewhere in the cluster. PODs are associated with resources like storage, memory or CPU.

Scheduler on cluster helps to deploy a POD with required number of containers inside the POD.

Controller helps to keep the state of the PODs with the responsibility to start and stop the POD. Also helps in horizontal scaling of PODs.

Most often PODs will not be created naked / bare which means PODs will be created under the controller so in case of failure of a POD the same can be recreated automatically.

Static PODs:

Static PODs are the PODs that are managed by kubelet service on the nodes. The PODs are maintained in the state that is defined in the manifest file located at `/etc/kubernetes/manifests`. Example of static PODs are the Kubernetes system PODs that area created at time of booting the Kubernetes cluster.

The manifest file path can be altered in the kubelet service Config file located at `/var/lib/kubelet/config.yaml.`

The **`staticPodPath`** is *watched by Kubelet service* on the Nodes. So the default Kubernetes system PODs are started and maintained the required state by the kubelet service and not by using `APIserver.` But these PODs can be seen via APIservice but are not controllable using APIservice. So if the kubelet service isn't running the Kubernetes cluster components may not start at boot time.

If we want a service to start every time the cluster is booted without a Replication controller, we can create a manifest and define the POD definitions in it and place the file in `/etc/kubernetes/manifests` path.

Working with a POD using kubectl:

Examples,

```
# execute a command inside a running POD. If there is more than one
container inside a POD, then we will have specify the container name
in the command.

$ kubectl exec –it <pod_name> --container <container_name> -- /bin/bash
```


```
# fetch logs for a container in a POD.

$ kubectl logs <pod_name> --container <container_name>

# port forward an application running inside a container in a POD

$ kubectl port-forward pod <pod_name> <local_port>:<container_port>

# start up kubectl and get events option and run it in background

$ kubectl get events  --watch &

# These events are the same that we see when we run kubectl describe
on a POD and see events that occur during the process of POD creation.

To understand this in detail, we can run a deployment imperatively
from command line using below request. Make sure you run the above
command '--watch' prior to running the deployment, so we can see the
events.

$ kubectl create deployment hello-deploy –-image=nginx:latest --replicas=2

On the screen we should be able to see the events for the above
deployment. We can make changes to the deployment object by running
below command.

$ kubectl scale deployment hello-deploy --replicas=1

Now, with above command the replicas are scaled down to 1 from the
count of 2.
```

Multi-Container POD:

What are the reasons to have a multi container POD?

1) Usually, the tightly coupled application running in container will be deployed inside one POD. And thus such application will be scheduled together.
2) Requirement to have some shared resourced between the processes running inside containers running inside one POD., like
3) One service in a container is generating data while other service in other container consumes that data. Such containers run in the same POD.

YAML for multi-container POD:

```
apiVersion: v1
kind: Pod
metadata:
  name: multicontainer-pod
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
  ...
  - name: alpine
    image: alpine
```

In case of multi container POD, we wouldn't want to have a web service and a Database container in one POD. This will limit the scalability of the application, as in most cases the requirement to have scaling for web service and database would always be different.

```
apiVersion: v1
kind: Pod
metadata:
   name: multicontainer-pod
spec:
   containers:
 -  name: producer
    image: Ubuntu
    command: [“/bin/bash”]
    args: [“-c”, “while true, do echo $(hostname) $(date) >>
    /var/log/index.html; sleep 10; done”]
    volumeMounts:
```

```
      - name: webcontent
        mountPath: /var/log
  - name: consumer
    image: nginx
    ports:
      - containerPort: 80
    volumeMounts:
      - name: webcontent
        mountPath: /usr/share/nginx/html
volumes:
  - name: webcontent
    emptyDir: {}
```

Here `emptyDir` is a volume type that enables to right data on the local node.

Data written to the `emptyDir` volume is not persistent. So if the POD is removed, data written to the volume will also get removed.

Init Container in a Multi-container POD:

- Init containers are deployed / started inside a POD to configure required setup before the main container starts.
- Init containers must complete the run before the main container starts. The Init container state must be changed to *complete* before the main container.
- There can be more than one Init containers inside a POD before the main container starts. All Init containers would start as per the order, sequentially specified inside the specification defined in manifest.
- If the Init container fails to start and complete the run, the *restartPolicy* applies which can control restarting of Init container.

Example:

```
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! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for myservice; sleep 2; done"]
```

```
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for mydb; sleep 2; done"]
```

Container Restart Policy:

- A container can restart independent of a POD
- Restart policy applies to containers inside a POD and is defined in specification of POD.
- A POD is the environment, a container run inside.
- If a container fails, it is not rescheduled on another node but restarted on that same node.
- **The restart policy can have below values,**
  1) Always (default) – will restart all container inside a POD
  2) OnFailure – on non-graceful termination
  3) Never – will never restart container in any condition

For demonstration purpose, we can create a POD with a container inside it. Then we can kill the running container inside the POD manually and see if the container gets restarted automatically or depending upon the restartPolicy, as *never, onFailure, always.*

Defining POD health:

- A POD is considered Ready when all containers inside that are Ready.
- While the POD is running we do not know the exact status of the Application running inside the container.
- We can add additional intelligence to our POD's health.
- Container probes
  - livenessProbe
  - readinessProbe
  - startupProbe

livenessProbe:

- A ***livenessProbe*** continuously runs a diagnostic check on the container.
- The livenessProbe is set in the specification of a container. So if there are multiple containers in a POD, then will have to set livenessProbe on every container if we want to know about the status of the container running App.
- On failure the kubelet will try to restart the container as per the restartPolicy.
- This gives Kubernetes a better understanding of applications inside containers.

readinessProbe:

- A ***readinessProbe*** continuous runs diagnostic check on the container.
- This setting is again the per container setting
- The container will not receive traffic until the readinessProbe succeeds.
- On failure of the container inside a POD, the POD is removed from the Load Balancing.
- This makes sure that the application will not get into errors.

startupProb:

- ***startupProbe*** continuously runs a diagnostic check on the container is the POD
- Ensures all containers in the POD are ready to receive traffic.
- Setting are per container
- On startup all probes area disabled until startProb succeeds.
- Some applications have a long start time and waiting for the application to be up is achieved using ***startupProbe.***
- On failure of startupProbe the container is restarted as per set restart policy.

Types of diagnostic checks for available with Probes:

- exec : executes a code inside container and checks the exit code. Depending upon the exit code, we derive status of Probe.
- tcpSocket: try to access a TCP port and derive Probe status as per result.
- httpget: try to access a URL and check the return code, if it is greater than or equal to 200 and less than 400,then derive the probe as success.

We can either get ***success, failure*** or ***unknown*** status for the above diagnostic checks

Configuring container probes:

- **initialDelaySeconds:** number of seconds to wait after the container has started before running the Probe, default 0.
- **periodSeconds:** time interval in seconds between running a Probe, default 10 seconds.
- **timeoutSeconds:** Probe time seconds, default 1.
- **FailureThreshold:** number of missed checks before reporting a failure, default 3.
- **successThreshold:** number probes to be considered successful and live, default 1.

```
spec:
  containers:
    ...
    livenessProbe:
      tcpSocket:
          port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

```
readinessProbe:
   tcpSocket:
     port: 8080
   initialDelaySeconds: 5
   periodSeconds: 10
```

For ***startupProbe*** below is a simple example.

```
spec:
  containers:
    ...
    startupProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 10
      periodSeconds: 5
```

While we start the demo, always make sure to start the ***event watch.***

```
$ kubectl get events --watch &
```

```
$ kubectl create –f container-probes.yaml
```

Now, above command will deploy defined resource (deployment) and also create a POD with container as defined in file with probes deployed as well.

Let's look at the YAML file.

```yaml
apiVersion: apps/v1
kind: deployment
metadata:
  name: hello-world
spec:
  replicas: 1
  selector:
    matchLables:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-world
        image: gcr.io/google-samples/hello-app:1.0
        ports:
        - containerPort: 8080
        livenessProbe:
          tcpSocket:
            port: 8081
          initialDelaySeconds: 10
```

```
    periodSeconds: 5
  readinessProbe:
    httpGet:
      path: /
      port: 8081
    initialDelaySeconds: 10
    periodSeconds: 5
```

In above file we have intentionally mentioned the port number 8081, which will cause the probe to fail. We will then make change to the port number and recheck the probe status.