
Running containers on K8S

Pods in Kubernetes

- Kubernetes groups multiple containers into a single, atomic unit called a *Pod*

- A Pod represents a collection of application containers and volumes running in the same execution environment.

- Applications running in the same Pod share the same:
  - IP-address (network port & namespace)
  - Hostname
  - Can communicate over System V IPC or POSIX message queues (IPC namespace)

Pods

- Pods are grouped containers that run on the same machine
- Different Pods can be considered as running on different machines (no-sharing-whatsoever)

- “Will these containers work correctly if they land on different machines?”
  - If No => they should be in the same Pod
  - Otherwise: different pods

- i.e. Wordpress with MySql containers in same Pod?

Running a simple pod

- `kubectl run my-pod --image nginx`
- `kubectl get pods --selector=run=my-pod`
- `kubectl delete deployments/my-pod`

Debugging

- `kubectl exec -it <pod-name> -- bash`
- `kubectl logs <pod-name>`
- `kubectl cp <pod-name>:/path/to/remote/file /path/to/local/file`

The Pod manifest

- Text file containing *declarative configuration*
  - Describing the desired state when this configuration is applied
- Imperative configuration, where you simply take a series of actions (e.g., apt-get install foo) to modify the world isn’t used in Kubernetes
- Declarative configuration is why Kubernetes can have self-healing behavior without user action.

Pod manifest

When we look back at the run command:

`kubectl run my-pod --image nginx`

Actually that command doesn’t just run the pod.

It creates a declarative manifest with a deployment

Pod manifest

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    name: my-pod
spec:
  containers:
  - name: my-pod
    image: nginx
    ports:
    - containerPort: 80
```

Pod manifest

```
kubectl apply –f nginx.yml
```

```
PS >kubectl apply -f .\nginx.yml
pod/my-pod created
```

```
kubectl get pods
```

```
kubectl delete –f nginx.yml / kubectl delete pods/my-pod
```

```
PS >kubectl delete -f .\nginx.yml
pod "my-pod" deleted
```

```
kubectl describe pods/my-pod
```

Port forwarding

Creating a secure tunnel from your localmachine -> K8S master -> pod

```
kubectl port-forward my-pod 8080:80
```

Connect via http://localhost:8080

```
kubectl logs my-pod
kubectl logs my-pod --previous
```

Healthchecks

- Kubernetes keeps your pod alive with *process health checks*
- Liveness probes
- Readiness probes

Liveness probe

- Simple process check is not enough
- Kubernetes can check if your application inside the container is alive by doing a liveness check.
- These need to be added manually into the manifest

```
livenessProbe:
  httpGet:
    path: /
    port: "80"
  initialDelaySeconds: 5
  timeoutSeconds: 1
  periodSeconds: 10
  failureTreshold: 3
  successThreshold: 1
```

Liveness probe

- HTTP type:
- Checks periodically by polling at the given path and portnumber
- Result should be statuscode between 200 and 400
- If it fails <failureTreshold> times the container will be restarted

Readiness probe

- Describes when a container is ready to handle user requests
- Configured just like liveness probes
- Combination of readiness and liveness probes makes sure only healthy containers are running
- Types
  - HTTP check
  - tcpSocket (tests if port can be opened)
  - Exec (runs a command, 0 resultcode is success)

Resource management

- In addition to simplified distributed application deployment we want to increase overall utilization of our compute nodes
- Basic cost of operating a machine is constant, regardless of being idle or fully loaded
- With Kubernetes managing resource packing we can go up well over 50%
- To do this Kubernetes needs to know what can be expected from your application in terms of resource requirements, so packing can be optimal across the cluster

Resource management

- Based on 2 different resource metrics
  - Requested resources
    - Minimum required:
    - CPU (i.e. 500m is half a cpu)
    - Memory (i.e. 128Mi is 128MB of memory)
    - Expressed in container section(s) of manifest file
  - Resource limits
    - Same settings, but maximizing/capping to the configured level

Resource management

```
containers:
  - name: my-pod
    image: nginx
    resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 500m
          memory: 256Mi
```

Persisting data with Volumes

- Containers are stateless by default
- During runtime we will get a temporary write-layer on top of our image
- After removing (or restarting!) a container all stored data will be lost
- Sometimes we need to store data that survives this.
- Kubernetes has the concept of volumes to support that.

Persistent volumes

- Add a `spec.volumes` section to the manifest
- This contains an array of volume definitions
- Not all containers in the pod are required to mount these volumes
- We can control which containers mount by adding the `volumeMounts` section to the declaration

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    name: my-pod
spec:
  volumes:
    - name: "my-pod-data"
      hostPath:
        path: "/var/lib/data"
  containers:
  - name: my-pod
    image: nginx
    volumeMounts:
      - mountPath: "/var/lib/data"
        name: "my-pod-data"
```

Volumes

- Communication / synchronization
- Cache
- Persistent data
- Mounting host file system (i.e. /dev)
  - hostPath variable in manifest
- Binding remote disks
  - NFS
  - iSCSI
  - Cloud provider based solutions (Azure Files,