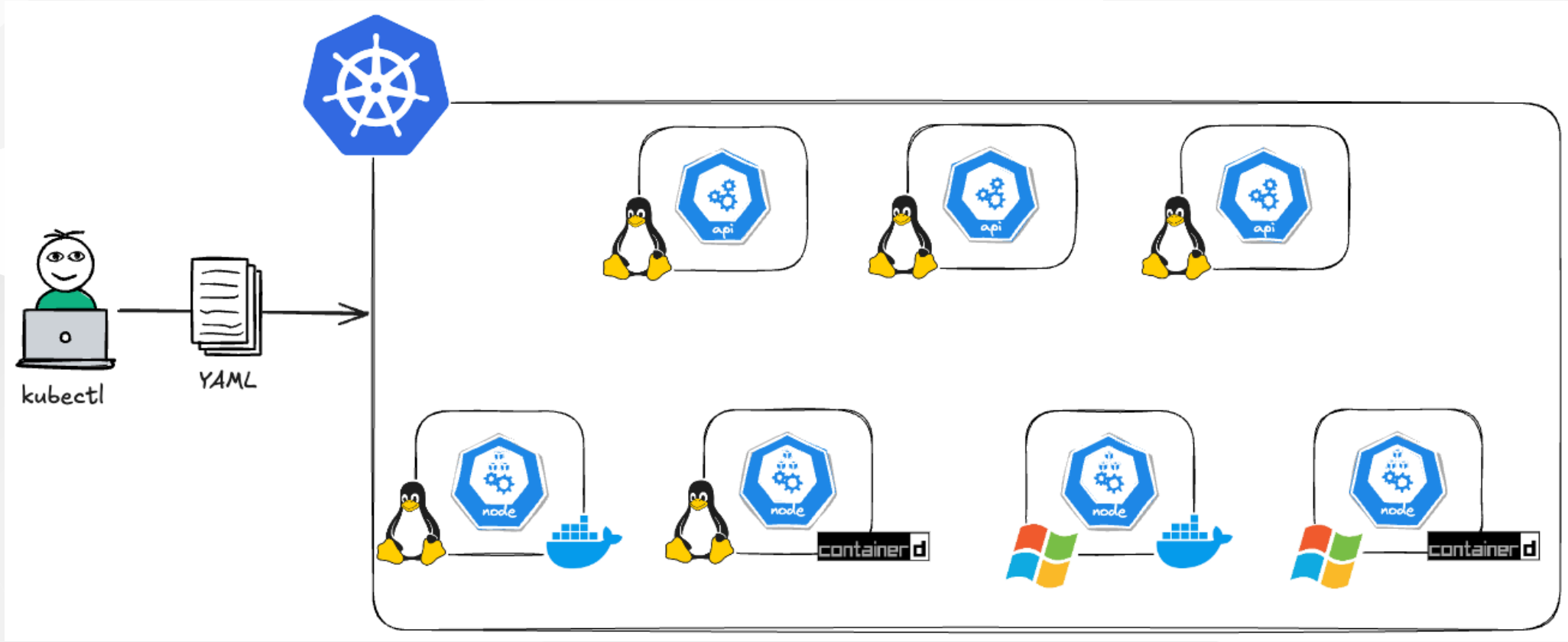
k8s Workshop

Part 1
Running containers with Pods and Deployments

Why k8s is so popular ?

API and Cluster

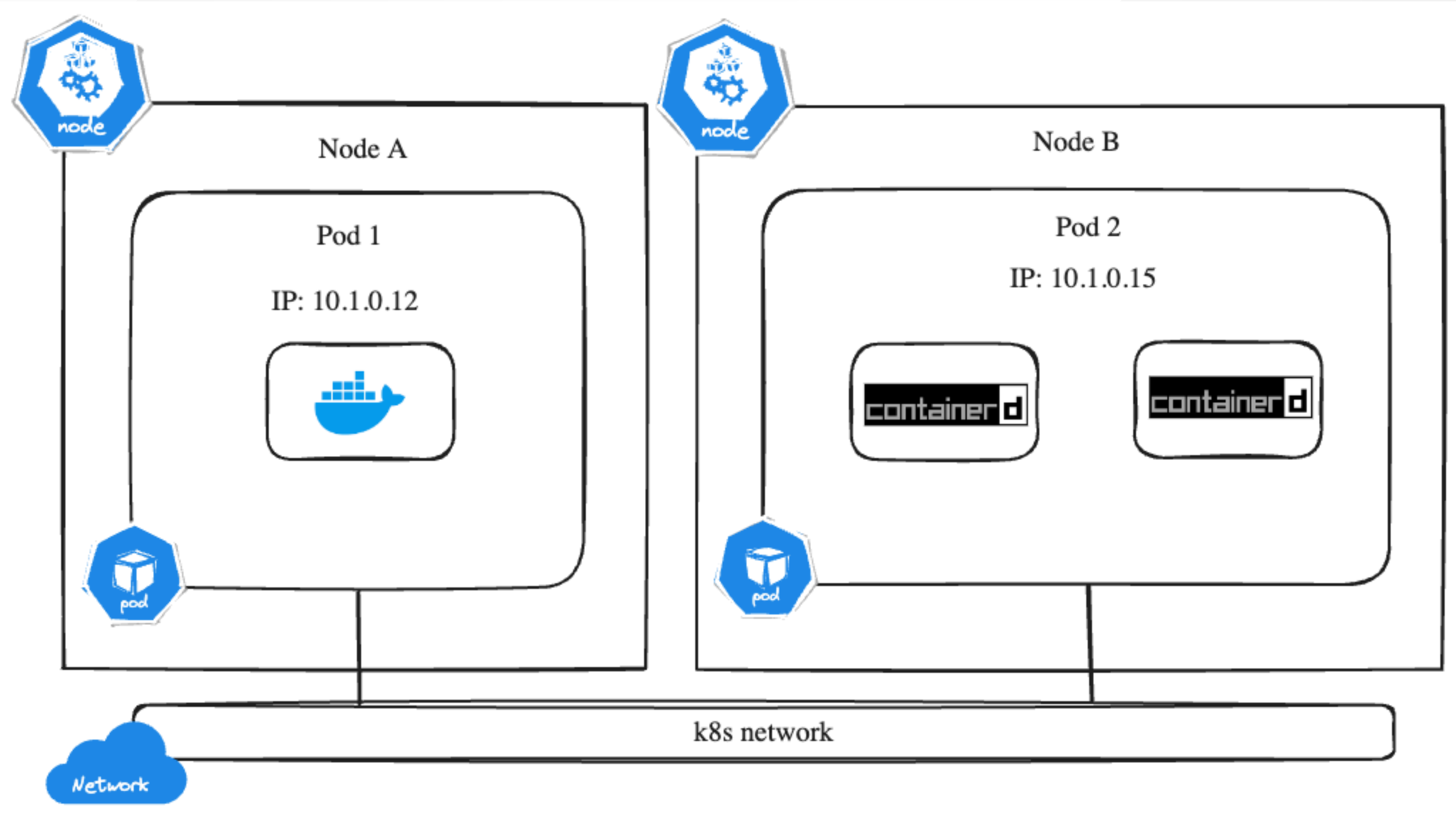


Containers & Pods

- Container is virtualized environment
- Pod wraps one or more containers
- Pod is a unit of compute
- Pod has a unique virtual IP
- Pods communicate over the k8s network
- Containers in a Pod share the same network address

Why a Pod is the smallest deployable unit in k8s ?

- k8s runs Pods not Containers
- Containers are managed by the container runtime of the node (Docker, containerd, etc.)
- Nodes manages containers using the CRI API
- The CRI API is standard for all container runtimes


node
Node A
Pod 1
IP: 10.1.0.12
pod
node
Node B
Pod 2
IP: 10.1.0.15
containerd
containerd
pod
k8s network
Network

kubectl

kubectl is cli tool for communicating with the k8s cluster's control plane using the k8s API. Some of the main kubectl commands:

- get: get information about resources
- describe: get detailed information about a resource
- run: run a resource
- create: create a resource
- apply: create or update a resource
- delete: delete a resource


HANDS-ON
with Pods

Let's create a Pod

```
kubectl run hello-world-1 --image=salvovitale/k8s-ws-app-hello-world:latest --restart=Never
```

check the pod

```
kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
hello-world-1   1/1     Running   0          39s
```

Use custom-columns

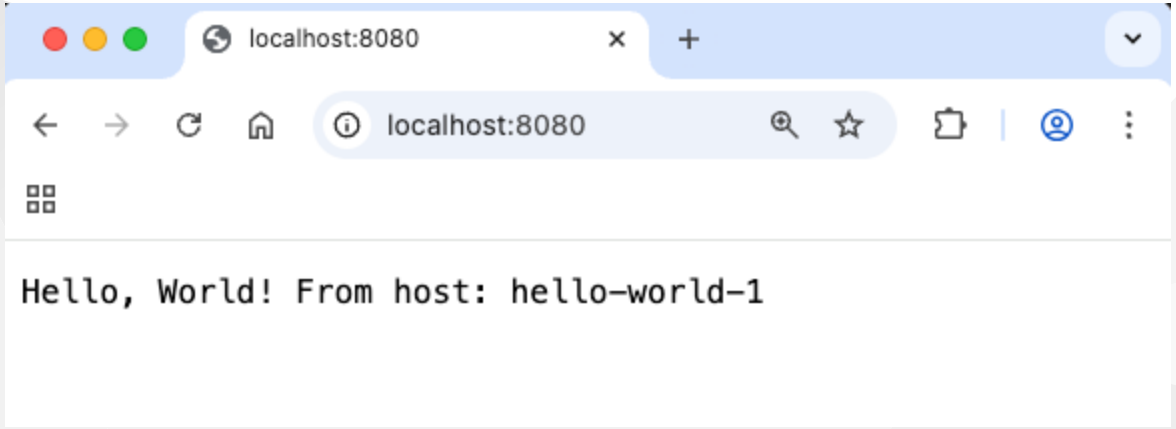
```
kubectl get pod hello-world-1 --output \
custom-columns=NAME:metadata.name,NODE_IP:status.hostIP,POD_IP:status.podIP
NAME            NODE_IP      POD_IP
hello-world-1   172.19.0.3   10.244.1.4
```

Search for specific information

```
kubectl get pod hello-world-1 -o jsonpath='{.status.containerStatuses[0].containerID}'
containerd://b4c566898b5e875a95cd475a71126da585e6f99afebb79f51dd88ca62eb5e53c%
```

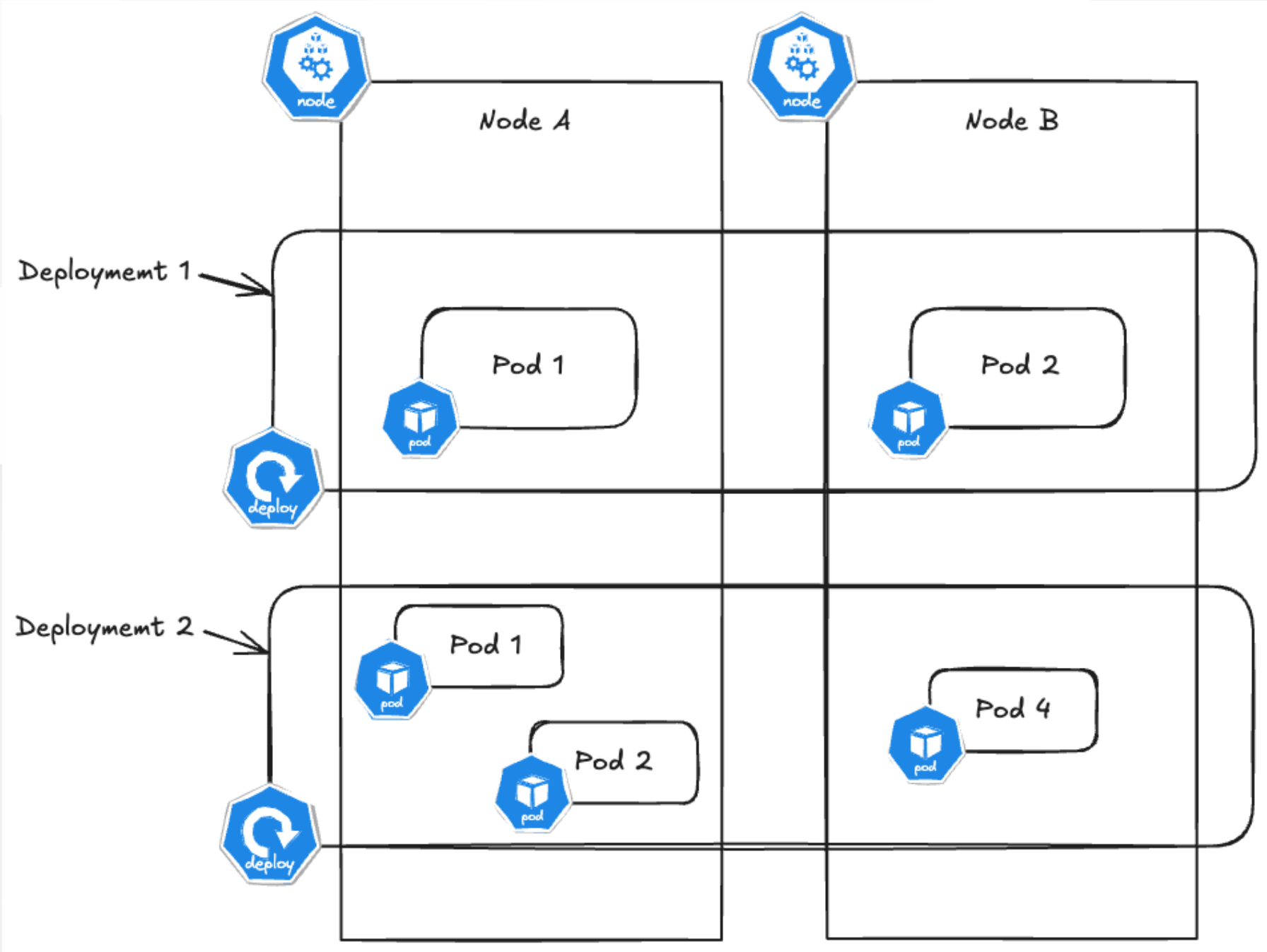
Port-forward

```
kubectl port-forward pod/hello-world-1 8080:8080
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```



Deployments

- Pods are too simple
- Deployments is a controller that manages Pods
- Controller ensures that its managed resources are running and healthy
- Controller uses the k8s API to watch the state of its managed resources


node
Node A
node
Node B
Deployment 1
Pod 1
pod
Pod 2
pod
deploy
Deployment 2
Pod 1
pod
Pod 2
pod
Pod 4
pod
deploy


HANDS-ON
with Deployments

Create a deployment

```
kubectl create deployment  hello-world-2 --image=salvovitale/k8s-ws-app-hello-world:latest
```

check the deployment

```
kubectl get deploy hello-world-2
```

```
kubectl describe deploy hello-world-2
```

Resources Relationship and Labels

- k8s resources can have labels
- Labels are key-value pairs attached to k8s resources
- k8s uses labels to loosely couple resources (Deployments <--> Pods)
- controllers use labels to identify the resources they manage
- controllers don't need to maintain the list of resources they manage

Get the labels for the deployment

```
kubectl get deploy hello-world-2 -o jsonpath='{.spec.template.metadata.labels}'
```

we can use the labels to identify the pods

```
kubectl get pod -l app=hello-world-2
```

lets inspect the current labels situation in pods

```
kubectl get pods -o custom-columns=NAME:metadata.name,LABELS:metadata.labels
NAME                              LABELS
hello-world-1                      map[run:hello-world-1]
hello-world-2-69c988857-bdj5w   map[app:hello-world-2 pod-template-hash:69c988857]
```

lets override the label of a pod linked to the deployment

```
kubectl label pods -l app=hello-world-2 --overwrite app=hello-world-x
```

lets inspect the new labels situation in pods

```
kubectl get pods -o custom-columns=NAME:metadata.name,LABELS:metadata.labels
NAME                            LABELS
hello-world-1                    map[run:hello-world-1]
hello-world-2-69c988857-9fkxf   map[app:hello-world-x pod-template-hash:69c988857]
hello-world-2-69c988857-bdj5w   map[app:hello-world-2 pod-template-hash:69c988857]
```

lets revert back the situation

```
kubectl label pods -l app=hello-world-x --overwrite app=hello-world-2
```

Manifests

- complete description of your app
- written in JSON or YAML
- versioned and tracked in source control
- create the same deployment on any k8s cluster
- declarative not imperative
- kubectl apply -f <manifest>

Pod manifest

```yaml
# Manifests always specify the version of the Kubernetes API and the type of resource.
apiVersion: v1
kind: Pod
# Metadata for the resource includes the name (mandatory) and labels (optional).
metadata:
  creationTimestamp: null
  labels:
    run: hello-world-3
  name: hello-world-3
# The spec is the actual specification for the resource.
# For a Pod the minimum is the container(s) to run,
# with the container name and image.
spec:
  containers:
  - image: app-hello-world:latest
    name: hello-world-3
```

Deployment manifest

```yaml
# Deployments are part of the apps version 1 API spec.
apiVersion: apps/v1
kind: Deployment
# Metadata for the resource includes the name (mandatory) and labels (optional).
metadata:
  labels:
    app: hello-world-4
  name: hello-world-4
# The spec includes the label selector the Deployment uses it to find its own managed resources. I'm using the app label,
# but this could be any combination of key-value pairs.
spec:
  selector:
    matchLabels:
      app: hello-world-4
# The template is used when the Deployment creates a Pod
  template:
  # Pods in a Deployment don't have a name, but they need to specify labels that match the selector in the spec.
    metadata:
      labels:
        app: hello-world-4
    # The Pod spec lists the container name and image
    spec:
      containers:
      - image: app-hello-world:latest
        name: app-hello-world
```


HANDS-ON
with Manifests

Lets use the cli to generate a manifest for the pod we created before

```
kubectl run hello-world-3 \
--image=salvovitale/k8s-ws-app-hello-world:latest \
--image-pull-policy=IfNotPresent \
-o yaml \
--dry-run=client > pod.yaml
```

and apply it

```
kubectl apply -f pod.yaml
```

lets now generate the manifest for a a deployment

```
kubectl create deployment hello-world-4 \
--image="app-hello-world:latest" \
-o yaml \
--dry-run=client > deployment.yaml
```

and apply it

```
kubectl apply -f deployment.yaml
```

Interactive shell

We can run an interactive shell in the pod container

```
kubectl exec -it hello-world-3 -- sh
```

we can also target a deployment

```
kubectl exec -it deploy/hello-world-4 -- sh
```

Inspect Logs

We can check the logs of the application running in the pod

```
kubectl logs pod/hello-world-3 -f
```

again also in this case we can refer directly to the deployment

```
kubectl logs deploy/hello-world-4 -f
```

Filesystem

Interacting with the filesystem of pod

```
kubectl cp temp/text.txt hello-world-3:/app/text.txt
```

also works the other way around to copy a file from the pod to the local filesystem

Delete Resources

Lets delete all the pods

```
kubectl delete pods --all
```

The pod managed by the deployment will be re-created. To delete that resource we need to delete the controlling resource. In this case the deployment

```
kubectl delete deploy --all
```

Some final info

- ohmyzsh kubectl alias

```
alias | grep "kubectl"
```

Lab

- Use the "kubectl run" to create a pod with the official Nginx image
- Port-forward to the pod
- Use the "kubectl create" to create a deployment with the official Nginx image
- Use kubectl to generate basic manifests for both pod and deployment
- Interact with the filesystem to modify/replace the nginx landing page
- Repeat with an hello-world app in your technology of choice