Hands-on Kubernetes Interview Questions

The Kubernetes interview questions that follow go beyond the typical rudimentary inquiries, such as “What is a pod?” and “What is a service?” This article assumes that you can answer those questions. Instead, these technical questions are designed for developers applying for positions that require a more advanced understanding of Kubernetes, to demonstrate their hands-on working knowledge of Kubernetes and container orchestration.

These questions just scratch the surface of working with Kubernetes beyond rote usage. But, being able to answer them is a good way to move onto the next step in your career as a Kubernetes application developer.

Why won’t the service defined in the manifest file below bind to the pod defined in the same file?

Below is a manifest file that declares a Kubernetes deployment and a Kubernetes service. What’s going wrong with it?

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my_deployment
spec:
  selector:
    matchLabels:
      type: example
      color: red
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 10%
  template:
    metadata:
      labels:
        type: example
        color: red
    spec:
      containers:
      - name: echocolor
        image: reselbob/echocolor:v0.1
        ports:
        - containerPort: 3000
        env:
        - name: COLOR_ECHO_COLOR
          value: RED
        - name: COLOR_ECHO_VERSION
          value: V1
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: my_service
spec:
  selector:
    type: example
    color: blue
  ports:
  -
    protocol: TCP
    port: 3000
    targetPort: 3000

  type: NodePort
```

The reason that the service cannot bind to the pods in the deployment is because the labels in the deployment's pods do not match the service's selector field values.

A service binds to a pod through a label match between a service selector and pod. When a service starts up, it "looks for" pods in the cluster with labels that are declared in the service's selector field.

Figure 1 below illustrates the problem that exists in the manifest file above. Notice that the service in the figure below shows a Kubernetes service that has the selector values *type=example* and *color=blue*. Yet, the labels field of each of the pods in the Kubernetes deployment have the values *type=example* and *color=red*. The values in the service's selector field and the pods' labels fields do not match up. Hence, no binding.

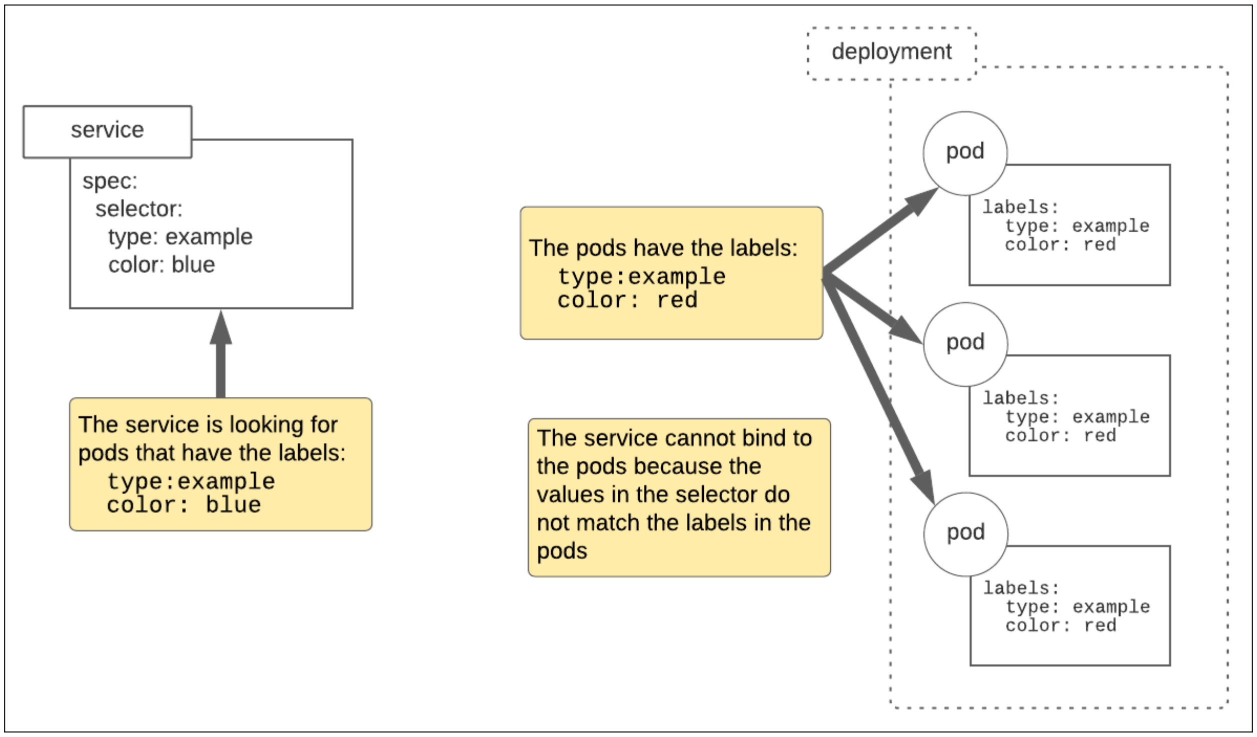

Figure 1: For a Kubernetes service to bind to a pod, the values in the service's *selector* field must match the values in the pods' *labels* fields.

The values in the service's selector field and the pods' labels fields do NOT match up. Hence, no binding.

To bind Kubernetes to the existing pods, change the values in the service's selector field to *type=example* and *color=red*.

What is an init container and when would you use one?

An init container is a container that Kubernetes runs before any other containers in the pod are created. You can use an init container to implement initialization behavior that other pods will use.

The example below shows how to create an init container that waits for a database service to come online before it creates the container that will use the database.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.31
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.31

    command: ['sh', '-c', 'until nslookup redis-master; do echo waiting
for redis-master; sleep 2; done;']
```

What’s the difference between a Kubernetes ConfigMap and a Kubernetes secret?

A ConfigMap is a Kubernetes API object used to store data that is not confidential. ConfigMap information is stored in a variety of information formats, such as key-value pairs and JSON. (The sample Kubernetes ConfigMap below contains both name-value pairs and a JSON object.) Typically a developer uses a ConfigMap to provide information to the cluster that gets consumed by other API objects.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: db-configmap
data:
  # Configuration data as key-value properties
  database: mongodb
  database_uri: mongodb://localhost:27017
  # Other data in JSON
  keys: |
    maximum.connections=5

    timeout=10000
```

A Kubernetes secret is a Kubernetes object that stores information in the cluster in an encrypted format. Typically a secret is used to store confidential information. The example below is a manifest file that describes a Kubernetes secret for a username/password pair. Notice the Base64 encrypted values.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  # An encrypted username/password pair
  username: YWRtaW4=

  password: MWYyZDFlMmU2N2Rm
```

What is a Kubernetes operator?

A Kubernetes operator is a design pattern to package, deploy, and manage a Kubernetes application. Think of an operator as a way to create and deploy all the Kubernetes resources that go with an application at once using automation.

For example, an operator creates a Kubernetes service, the backing pods, the storage volumes as well as the Roles, and RoleBindngs, and all the configurations that are part of an application that runs on a Kubernetes cluster.

There are Kubernetes operators for MySQL, Oracle and Redis to name a few

How do you implement service discovery internally in a Kubernetes cluster?

Kubernetes comes with an internal DNS server that automatically assigns DNS names to the services it creates. These DNS names are accessible from pods within the cluster.

The format of the DNS name is:

`<service>.<namepace>`

However, if both the service and the pod calling the service are in the same namespace, the calling container within the pod can use just the service name as the DNS name (see **Figure 2** below).

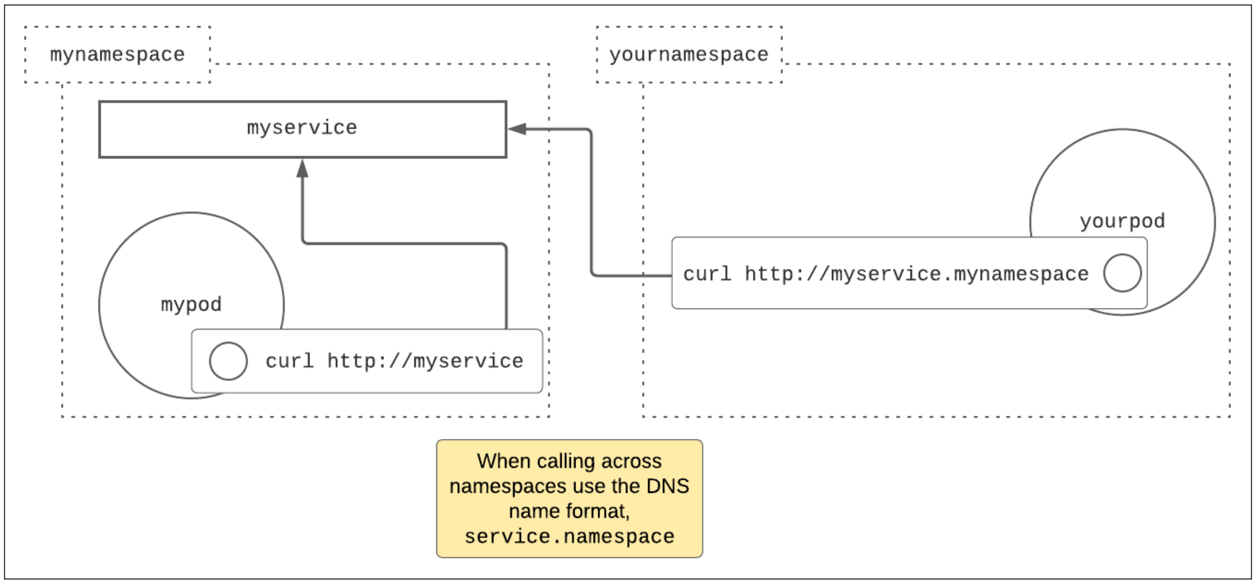
mynamespace
myservice
mypod
curl http://myservice
yournamespace
yourpod
curl http://myservice.mynamespace
When calling across namespaces use the DNS name format, service.namespace

Figure 2: Kubernetes has a DNS server that creates DNS names accessible within a cluster.

For example, imagine you have a service, called *myservice*, in the namespace called *mynamespace*. The service *myservice* represents an HTTP web server.

Now, imagine you have a pod named *mypod* also in the namespace *mynamespace*, and you have a pod named *yourpod* in the namespace *yournamespace*.

To call the service *myservice* from a container in the pod *yourpod*, using curl for example, you would use the URL http://myservice.mynamespace. Why? Because *yourpod* is in a different namespace from *myservice*.

However, to call *myservice* from *mypod* using curl, the DNS name to use is http://myservice because *mypod* and *myservice* are in the same namespace, *mynamespace*.

What is the purpose of kubelet?

The purpose of kubelet is to realize containers on a given Kubernetes node. Every node in a Kubernetes cluster has a running instance of kubelet.

When Kubernetes needs to create a pod(s), it identifies the node where the pod(s) can be created using Scheduler. A message then gets sent to the kubelet instance on the identified node. Kubelet in turn works with the container runtime interface (CRI) to create the container(s) for the pod(s). Figure 3 below outlines how these processes work together.

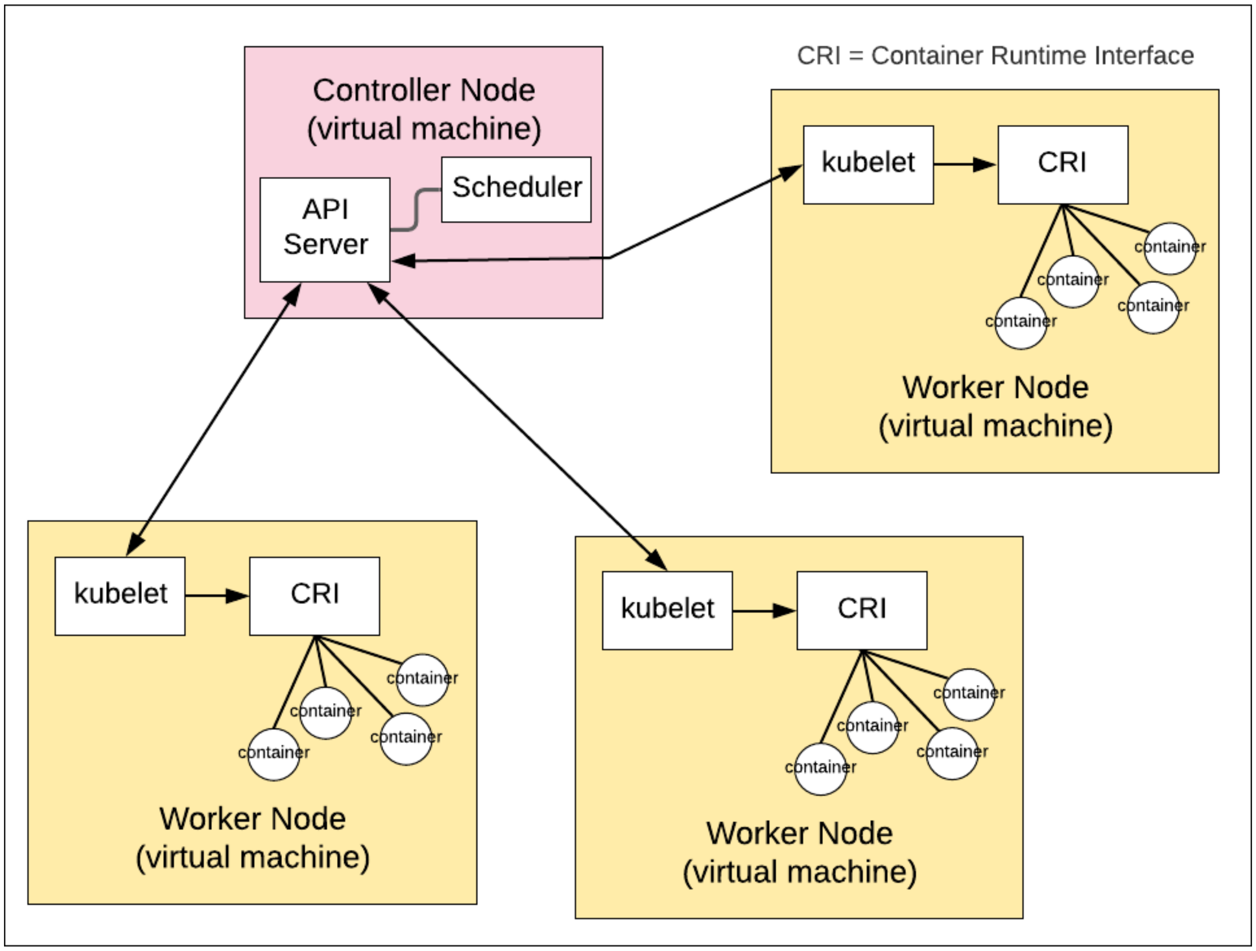
CRI = Container Runtime Interface
Controller Node (virtual machine)
API Server
Scheduler
kubelet
CRI
container
Worker Node (virtual machine)

Figure 3: How kubelet realizes containers on a Kubernetes node

What is RBAC and how does it relate to Kubernetes?

Role-based access control (RBAC) is the technique by which one creates Kubernetes Roles and RoleBindings. A Kubernetes Role is an abstraction that defines a relationship between a Kubernetes resource and action — for example, “can create a pod” where *pod* is the Kubernetes resource and *create* is the action. A RoleBinding maps a particular user or service to a particular Role.

What is the difference between a StatefulSet and a DaemonSet?

A StatefulSet is a Kubernetes API object that supports data persistence. A StatefulSet is similar to a deployment, in that it's a collection of pods, However, in a deployment the given pod loses its data when it is destroyed. A StatefulSet enables a pod's information to be preserved and reattached to a newly created pod.

A DaemonSet is an API object in which a copy of a given pod is run on each node in a Kubernetes cluster. Logging is a good example of the usefulness of a DaemonSet. A logging system typically uses an application called a collector to gather all the log information generated on each machine, and then forwards that information to a central location where it's aggregated. Configuring a log collector as a Kubernetes DaemonSet ensures that the collector is run installed on each node.

What is a sidecar and when is it best to use one?

In a sidecar pattern, one container is created in a pod to provide service to another container in that pod. The sidecar container runs on the same machine under the same IP address as all the other containers that run in the pod. Thus, it's very efficient to move traffic between the main container and the sidecar container. The sidecar provides extended features to the main container, yet it is independent so it allows updates without stopping the main container.

When you need to provide additional features or services to a container in a versatile, low-latency manner, the sidecar pattern is a good way to go.

How can you make it so that a pod runs on a specific node?

Use node affinity. Node affinity is a Kubernetes deployment technique in which a node is assigned an arbitrary label, and then pods are configured to be assigned to that node according to the label created.

For example, this following code snippet creates an arbitrary label, *nodelocation*, and assigns the value *usa* to the node named *worker-01*:

```
kubectl label nodes worker-01 nodelocation=usa
```

The manifest file shown below describes a Kubernetes deployment in which all the pods created for the deployment are assigned to any node that has a label with key-value pair *nodelocation=usa*.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
          # bind the pods in the deployment to nodes that have the
key-value
      # pair, nodelocation=usa
      affinity:
         nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: nodelocation
                operator: In
                values:
                - usa
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
```

```
        - containerPort: 80
```

Credit: Bob Reselman