Red Hat Enterprise Linux Atomic Host 7
Getting Started with Kubernetes

Getting Started with Kubernetes

Red Hat Atomic Host Documentation Team

Getting Started with Kubernetes

Legal Notice

Abstract

Set up Kubernetes on RHEL or RHEL Atomic Host and learn to manage pods

Table of Contents

CHAPTER 1. GET STARTED ORCHESTRATING CONTAINERS WITH KUBERNETES

1.1. OVERVIEW

Kubernetes is a tool for orchestrating and managing Docker containers. Red Hat provides several ways you can use Kubernetes that include:

- **OpenShift Container Platform**: Kubernetes is built into OpenShift, allowing you to configure Kubernetes, assign host computers as Kubernetes nodes, deploy containers to those nodes in pods, and manage containers across multiple systems. The OpenShift Container Platform web console provides a browser-based interface to using Kubernetes.
- **Container Development Kit (CDK)**: The CDK provides Vagrantfiles to launch the CDK with either OpenShift (which includes Kubernetes) or a bare-bones Kubernetes configuration. This gives you the choice of using the OpenShift tools or Kubernetes commands (such as **kubectl**) to manage Kubernetes.
- **Kubernetes in Red Hat Enterprise Linux**: To try out Kubernetes on a standard Red Hat Enterprise Linux server system, you can install a combination of RPM packages and container images to manually set up your own Kubernetes configuration.

The procedures in this section let describes how to set up Kubernetes using that last option (Kubernetes on Red Hat Enterprise Linux or Red Hat Enterprise Linux Atomic Host). Specifically, in this chapter you set up a single-system Kubernetes sandbox so you can:

- Deploy and run two containers with Kubernetes on a single system.
- Manage containers in pods with Kubernetes.

This procedure results in a setup that provides an all-in-one Kubernetes configuration in which you can begin trying out Kubernetes and exploring how it works. In this procedure, services that are typically on a separate Kubernetes master system and two or more Kubernetes node systems are all running on a single system.

Note

The Kubernetes software described in this chapter is packaged and configured differently than the Kubernetes included in OpenShift. We recommend you use the OpenShift version of Kubernetes for permanent setups and production use. The procedure described in this chapter should only be used as a convenient way to try out Kubernetes on an all-in-one RHEL or RHEL Atomic Host system. As of RHEL 7.3, support for the procedure for configuring a Kubernetes cluster (separate master and multiple nodes) directly on RHEL and RHEL Atomic Host has ended. For further details on Red Hat support for Kubernetes, see How are container orchestration tools supported with Red Hat Enterprise Linux?

1.2. UNDERSTANDING KUBERNETES

While the Docker project defines a container format and builds and manages individual containers, an orchestration tool is needed to deploy and manage sets of containers. Kubernetes is a tool designed to orchestrate Docker containers. After building the container images you want, you can use a Kubernetes Master to deploy one or more containers in what is referred to as a pod. The

Master tells each Kubernetes Node to pull the needed the containers to that Node, where the containers run.

Kubernetes can manage the interconnections between a set of containers by defining Kubernetes Services. As demand for individual container pods increases or decreases, Kubernetes can run or stop container pods as needed using its replication controller feature.

For this example, both the Kubernetes Master and Node are on the same computer, which can be either a RHEL 7 Server or RHEL 7 Atomic Host. Kubernetes relies on a set of service daemons to implement features of the Kubernetes Master and Node. Some of those run as systemd services while others run from containers. You need to understand the following about Kubernetes Masters and Node:

- **Master**: A Kubernetes Master is where you direct API calls to services that control the activities of the pods, replications controllers, services, nodes and other components of a Kubernetes cluster. Typically, those calls are made by running **kubectl** commands. From the Master, containers are deployed to run on Nodes.
- **Node**: A Node is a system providing the run-time environments for the containers. A set of container pods can span multiple nodes.

Pods are defined in configuration files (in YAML or JSON formats). Using the following procedure, you will set up a single RHEL 7 or RHEL Atomic system, configure it as a Kubernetes Master and Node, use YAML files to define each container in a pod, and deploy those containers using Kubernetes (**kubectl** command).

Note

Three of the Kubernetes services that were defined run as systemd services (**`kube-apiserver`**, **`kube-controller-manager`**, and **`kube-scheduler`**) in previous versions of this procedure have been containerized. As of RHEL 7.3, only containerized versions of those services are available. So this procedure describes how to use those containerized Kubernetes services.

1.3. RUNNING CONTAINERS FROM KUBERNETES PODS

You need a RHEL 7 or RHEL Atomic system to build the Docker containers and orchestrate them with Kubernetes. There are different sets of service daemons needed on Kubernetes Master and Node systems. In this procedure, all service daemons run on the same system.

Once the containers, system and services are in place, you use the **kubectl** command to deploy those containers so they run on the Kubernetes Node (in this case, that will be the local system).

Here's how to do those steps:

1.3.1. Setting up to Deploy Docker Containers with Kubernetes

To prepare for Kubernetes, you need to install RHEL 7 or RHEL Atomic Host, disable firewalld, get two containers, and add them to a Docker Registry.

1. **Install a RHEL 7 or RHEL Atomic system**: For this Kubernetes sandbox system, install a RHEL 7 or RHEL Atomic system, subscribe the system, then install and start the docker service. Refer here for information on setting up a basic RHEL or RHEL Atomic system to use with Kubernetes:

Get Started with Docker Formatted Container Images on Red Hat Systems

2. **Install Kubernetes**: If you are on a RHEL 7 system, install the docker, etcd, and some kubernetes packages. These packages are already installed on RHEL Atomic:

```
# yum install docker kubernetes-client kubernetes-node etcd
```

3. **Disable firewalld**: If you are using a RHEL 7 host, be sure that the firewalld service is disabled (the firewalld service is not installed on an Atomic host). On RHEL 7, type the following to disable and stop the firewalld service:

```
# systemctl disable firewalld
# systemctl stop firewalld
```

4. **Get Docker Containers**: Build the following two containers using the following instructions:

   - Simple Apache Web Server in a Docker Container
   - Simple Database Server in a Docker Container

   After you build, test and stop the containers (**docker stop mydbforweb** and **docker stop mywebwithdb**), add them to a registry.

5. **Install registry**: To get the Docker Registry service (v2) on your local system, you must install the docker-distribution package. For example:

```
# yum install docker-distribution
```

6. **Start the local Docker Registry**: To start the local Docker Registry, type the following:

```
# systemctl start docker-distribution
# systemctl enable docker-distribution
# systemctl is-active docker-distribution
active
```

7. **Tag images**: Using the image ID of each image, tag the two images so they can be pushed to your local Docker Registry. Assuming the registry is running on the local system, tag the two images as follows:

```
# docker images
REPOSITORY    TAG         IMAGE ID      CREATED
VIRTUAL SIZE
dbforweb      latest      c29665465a6c  4 minutes ago    556.2
MB
webwithdb     latest      80e7af79c507  14 minutes ago   405.6
MB
# docker tag c29665465a6c localhost:5000/dbforweb
# docker push localhost:5000/dbforweb
# docker tag 80e7af79c507 localhost:5000/webwithdb
# docker push localhost:5000/webwithdb
```

The two images are now available from your local Docker Registry.

1.3.2. Starting Kubernetes

Because both Kubernetes Master and Node services are running on the local system, you don't need to change the Kubernetes configuration files. Master and Node services will point to each other on localhost and services are made available only on localhost.

1. **Pull Kubernetes containers**: To pull the Kubernetes container images, type the following:

```
# docker pull registry.access.redhat.com/rhel7/kubernetes-
apiserver
# docker pull registry.access.redhat.com/rhel7/kubernetes-
controller-mgr
# docker pull registry.access.redhat.com/rhel7/kubernetes-
scheduler
```

2. **Create manifest files**: Create the following apiserver-pod.json, controller-mgr-pod.json, and scheduler-pod.json files and put them in the **/etc/kubernetes/manifests** directory. These files identify the images representing the three Kubernetes services that are started later by the **kubelet** service:

   apiserver-pod.json

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-apiserver"
  },
  "spec": {
    "hostNetwork": true,
    "containers": [
      {
        "name": "kube-apiserver",
        "image": "rhel7/kubernetes-apiserver",
        "command": [
          "/usr/bin/kube-apiserver",
          "--v=0",
          "--address=0.0.0.0",
          "--etcd_servers=http://127.0.0.1:2379",
          "--service-cluster-ip-range=10.254.0.0/16",
          "--
admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,
SecurityContextDeny,ResourceQuota"
        ],
        "ports": [
          {
            "name": "https",
            "hostPort": 443,
            "containerPort": 443
          },
          {
            "name": "local",
            "hostPort": 8080,
            "containerPort": 8080
          }
        ],
        "volumeMounts": [
          {
```

```
            "name": "etcssl",
            "mountPath": "/etc/ssl",
            "readOnly": true
          },
          {
            "name": "config",
            "mountPath": "/etc/kubernetes",
            "readOnly": true
          }
        ],
        "livenessProbe": {
          "httpGet": {
            "path": "/healthz",
            "port": 8080
          },
          "initialDelaySeconds": 15,
          "timeoutSeconds": 15
        }
      }
    ],
    "volumes": [
      {
        "name": "etcssl",
        "hostPath": {
          "path": "/etc/ssl"
        }
      },
      {
        "name": "config",
        "hostPath": {
          "path": "/etc/kubernetes"
        }
      }
    ]
  }
}
```

controller-mgr-pod.json

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-controller-manager"
  },
  "spec": {
    "hostNetwork": true,
    "containers": [
      {
        "name": "kube-controller-manager",
        "image": "rhel7/kubernetes-controller-mgr",
        "volumeMounts": [
          {
            "name": "etcssl",
            "mountPath": "/etc/ssl",
            "readOnly": true
```

```
          },
          {
            "name": "config",
            "mountPath": "/etc/kubernetes",
            "readOnly": true
          }
        ],
        "livenessProbe": {
          "httpGet": {
            "path": "/healthz",
            "port": 10252
          },
          "initialDelaySeconds": 15,
          "timeoutSeconds": 15
        }
      }
    ],
    "volumes": [
      {
        "name": "etcssl",
        "hostPath": {
          "path": "/etc/ssl"
        }
      },
      {
        "name": "config",
        "hostPath": {
          "path": "/etc/kubernetes"
        }
      }
    ]
  }
}
```

scheduler-pod.json

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-scheduler"
  },
  "spec": {
    "hostNetwork": true,
    "containers": [
      {
        "name": "kube-scheduler",
        "image": "rhel7/kubernetes-scheduler",
        "volumeMounts": [
          {
            "name": "config",
            "mountPath": "/etc/kubernetes",
            "readOnly": true
          }
        ],
        "livenessProbe": {
```

```
          "httpGet": {
            "path": "/healthz",
            "port": 10251
          },
          "initialDelaySeconds": 15,
          "timeoutSeconds": 15
        }
      }
    ],
    "volumes": [
      {
        "name": "config",
        "hostPath": {
          "path": "/etc/kubernetes"
        }
      }
    ]
  }
}
```

3. **Configure the kubelet service**: Because the manifests define Kubernetes services as pods, the **kubelet** service is needed to start these containerized Kubernetes services. To configure the **kubelet** service, edit the **/etc/kubernetes/kubelet** and modify the KUBELET_ARGS line to read as follows (all other content can stay the same):

```
KUBELET_ADDRESS="--address=127.0.0.1"
KUBELET_HOSTNAME="--hostname-override=127.0.0.1"
KUBELET_API_SERVER="--api-servers=http://127.0.0.1:8080"
KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-
image=registry.access.redhat.com/rhel7/pod-infrastructure:latest"
KUBELET_ARGS="--register-node=true --
config=/etc/kubernetes/manifests/"
```

4. **Start kubelet and other Kubernetes services**: Start and enable the docker, etcd, kube-proxy and kubelet services as follows:

```
# for SERVICES in docker etcd kube-proxy kubelet; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl is-active $SERVICES
done
```

5. **Start the Kubernetes Node service daemons**: You need to start several services associated with a Kubernetes Node:

```
# for SERVICES in docker kube-proxy.service kubelet.service; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

6. **Check the services**: Run the ss command to check which ports the services are running on:

```
# ss -tulnp | grep -E "(kube)|(etcd)"
```

7. **Test the etcd service**: Use the **curl** command as follows to check the etcd service:

```
# curl -s -L http://localhost:2379/version
{"etcdserver":"2.3.7","etcdcluster":"2.3.0"}
```

1.3.3. Launching container pods with Kubernetes

With Master and Node services running on the local system and the two container images in place, you can now launch the containers using Kubernetes pods. Here are a few things you should know about that:

- **Separate pods**: Although you can launch multiple containers in a single pod, by having them in separate pods each container can replicate multiple instances as demands require, without having to launch the other container.
- **Kubernetes service**: This procedure defines Kubernetes services for the database and web server pods so containers can go through Kubernetes to find those services. In this way, the database and web server can find each other without knowing the IP address, port number, or even the node the pod providing the service is running on.

The following steps show how to launch and test the two pods:

IMPORTANT: It is critical that the indents in the YAML file be maintained. Spacing in YAML files are part of what keep the format cleaner (not requiring curly braces or other characters to maintain the structure).

1. **Create a Database Kubernetes service**: Create a **db-service.yaml** file to identify the pod providing the database service to Kubernetes.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: db
  name: db-service
  namespace: default
spec:
  ports:
  - port: 3306
  selector:
    app: db
```

2. **Create a Database server replication controller file**: Create a **db-rc.yaml** file that you will use to deploy the Database server pod. Here is what it could contain:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: db-controller
spec:
  replicas: 1
  selector:
    app: "db"
  template:
    metadata:
```

```
      name: "db"
      labels:
        app: "db"
    spec:
      containers:
      - name: "db"
        image: "localhost:5000/dbforweb"
        ports:
        - containerPort: 3306
```

3. **Create a Web server Kubernetes Service file**: Create a **webserver-service.yaml** file that you will use to deploy the Web server pod. Here is what it could contain:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: webserver
  name: webserver-service
  namespace: default
spec:
  ports:
  - port: 80
  selector:
    app: webserver
```

4. **Create a Web server replication controller file**: Create a **webserver-rc.yaml** file that you will use to deploy the Web server pod. Here is what it could contain:

```
kind: "ReplicationController"
apiVersion: "v1"
metadata:
  name: "webserver-controller"
spec:
  replicas: 1
  selector:
    app: "webserver"
  template:
    spec:
      containers:
        - name: "apache-frontend"
          image: "localhost:5000/webwithdb"
          ports:
            - containerPort: 80
    metadata:
      labels:
        app: "webserver"
        uses: db
```

5. **Orchestrate the containers with kubectl**: With the two YAML files in the current directory, run the following commands to start the pods to begin running the containers:

```
# kubectl create -f db-service.yaml
services/db-service
# kubectl create -f db-rc.yaml
replicationcontrollers/db-controller
```

```
# kubectl create -f webserver-service.yaml
services/webserver-service
# kubectl create -f webserver-rc.yaml
replicationcontrollers/webserver-controller
```

6. **Check rc, pods, and services**: Run the following commands to make sure that Kubernetes master services, the replication controllers, pods, and services are all running:

```
# kubectl cluster-info
Kubernetes master is running at http://localhost:8080
# kubectl get rc
CONTROLLER             CONTAINER(S)      IMAGE(S)    SELECTOR
REPLICAS  AGE
db-controller          db                dbforweb    name=db
1         14m
webserver-controller   apache-frontend   webwithdb
name=webserver   1         27m
# kubectl get pods --all-namespaces=true
NAME                                READY     STATUS
RESTARTS   AGE
db-controller-60wb4                 1/1       Running   0
2m
kube-apiserver-127.0.0.1            1/1       Running   2
58m
kube-controller-manager-127.0.0.1   1/1       Running   0
58m
kube-scheduler-127.0.0.1            1/1       Running   0
58m
webserver-controller-6uo45          1/1       Running   0
2m
# kubectl get service --all-namespaces=true
NAME                CLUSTER_IP       EXTERNAL_IP   PORT(S)
SELECTOR         AGE
db-service          10.254.14.227    <none>        3306/TCP
name=db          51m
kubernetes          10.254.0.1       <none>        443/TCP
<none>           59m
webserver-service   10.254.104.214   <none>        80/TCP
name=webserver   39m
```

7. **Check containers**: If both containers are running and the Web server container can see the Database server, you should be able to run the curl command to see that everything is working, as follows:

```
# curl http://localhost:80/cgi-bin/action
<html>
<head>
<title>My Application</title>
</head>
<body>
<h2>RedHat rocks</h2>
<h2>Success</h2>
</body>
</html>
```

If you have a Web browser installed on the localhost, you can open that Web browser to see a better

representation of the few lines of output. Just open the browser to this URL: **http://localhost/cgi-bin/action**.

1.4. EXPLORING KUBERNETES PODS

If something goes wrong along the way, there are several ways to determine what happened. One thing you can do is to examine services inside of the containers. To do that, you can look at the logs inside the container to see what happened. Run the following command (replacing the last argument with the pod name you want to examine).

```
# kubectl logs kube-controller-manager-127.0.0.1
```

Another problem that people have had comes from forgetting to disable firewalld. If firewalld is active, it could block access to ports when a service tries to access them between your containers. Make sure you have run **systemctl stop firewalld ; systemctl disable firewalld** on your host.

If you made a mistake creating your two-pod application, you can delete the replication controllers and the services. (The pods will just go away when the replication controllers are removed.) After that, you can fix the YAML files and create them again. Here's how you would delete the replication controllers and services:

```
# kubectl delete rc webserver-controller
replicationcontrollers/webserver-controller
# kubectl delete rc db-controller
replicationcontrollers/db-controller
# kubectl delete service webserver-service
services/webserver-service
# kubectl delete service db-service
```

Remember to not just delete the pods. If you do, without removing the replication controllers, the replication controllers will just start new pods to replace the ones you deleted.

The example you have just seen is a simple approach to getting started with Kubernetes. Because it involves only one master and one node on the same system, it is not scalable. To set up a more formal and permanent Kubernetes configuration, Red Hat recommends using OpenShift Container Platform.

CHAPTER 2. GET STARTED PROVISIONING STORAGE IN KUBERNETES

2.1. OVERVIEW

This section explains how to provision storage in Kubernetes.

Before undertaking the exercises in this topic, you must have Kubernetes set up.

If you do not have Kubernetes set up, follow the instructions in Get Started Orchestrating Containers with Kubernetes.

2.2. KUBERNETES PERSISTENT VOLUMES

This section provides an overview of Kubernetes Persistent Volumes. The example below explains how to use the **`nginx`** web server to serve content from a persistent volume.

This section assumes that you understand the basics of Kubernetes and that you have a Kubernetes cluster up and running.

A Persistent Volume (PV) in Kubernetes represents a real piece of underlying storage capacity in the infrastructure. Before using Kubernetes to mount anything, you must first create whatever storage that you plan to mount. Cluster administrators must create their GCE disks and export their NFS shares in order for Kubernetes to mount them.

Persistent volumes are intended for "network volumes" like GCE Persistent Disks, NFS shares, and AWS ElasticBlockStore volumes. HostPath was included for ease of development and testing. You'll create a local HostPath for this example.

Important

In order for HostPath to work, you will need to run a single node cluster. Kubernetes does not support local storage on the host at this time. There is no guarantee that your pod will end up on the correct node where the HostPath resides.

```
// this will be nginx's webroot
$ mkdir /tmp/data01
$ echo 'I love Kubernetes storage!' > /tmp/data01/index.html
```

Define physical volumes in a YAML file.

```
$ mkdir -p ~/examples/persistent-volumes/volumes/
$ vi ~/examples/persistent-volumes/volumes/local-01.yaml
```

Create the following content in the **`local-01.yaml`** file:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv0001
  labels:
```

```
    type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/data01"
```

Create physical volumes by posting them to the API server.

```
$ kubectl create -f ~/examples/persistent-volumes/volumes/local-01.yaml
persistentvolume "pv0001" created

$ kubectl get pv
NAME      CAPACITY   ACCESSMODES   STATUS      CLAIM     REASON    AGE
pv0001    10Gi       RWO           Available
14s
```

2.2.1. Requesting storage

Users of Kubernetes request persistent storage for their pods. The nature of the underlying provisioning need not be known by users. Users must know that they can rely on their claims to storage and that they can manage that storage's lifecycle independently of the many pods that may use it.

Claims must be created in the same namespace as the pods that use them.

Create a YAML file defining the storage claim.

```
$ mkdir -p ~/examples/persistent-volumes/claims/
$ vi ~/examples/persistent-volumes/claims/claim-01.yaml
```

Add the following content to the **`claim-01.yaml`** file:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim-1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the claim.

```
$ kubectl create -f ~/examples/persistent-volumes/claims/claim-01.yaml
persistentvolumeclaim "myclaim-1" created
```

A background process will attempt to match this claim to a volume. The state of your claim will eventually look something like this:

```
$ kubectl get pvc
NAME        STATUS    VOLUME    CAPACITY   ACCESSMODES   AGE
myclaim-1   Bound     pv0001    10Gi       RWO           7s

$ kubectl get pv
NAME      CAPACITY   ACCESSMODES   STATUS    CLAIM
REASON    AGE
pv0001    10Gi       RWO           Bound     default/myclaim-1
```

2.2.2. Using your claim as a volume

Claims are used as volumes in pods. Kubernetes uses the claim to look up its bound PV. The PV is then exposed to the pod.

Start by creating a **pod.yaml** file.

```
$ mkdir -p ~/examples/persistent-volumes/simpletest/
$ vi ~/examples/persistent-volumes/simpletest/pod.yaml
```

Add the following content to the **pod.yaml** file:

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
  labels:
    name: frontendhttp
spec:
  containers:
    - name: myfrontend
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
      - mountPath: "/usr/share/nginx/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
       claimName: myclaim-1
```

Use **pod.yaml** to create the pod and the claim, then check that it was all done properly.

Add the following content to the **pod.yaml** file:

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
  labels:
    name: frontendhttp
spec:
  containers:
```

```
    - name: myfrontend
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
      - mountPath: "/usr/share/nginx/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
       claimName: myclaim-1
```

Use **pod.yaml** to create the pod and the claim, then check that it was all done properly.

```
$ kubectl create -f ~/examples/persistent-volumes/simpletest/pod.yaml

$ kubectl describe pods mypod | less
Name:           mypod
Namespace:      default
Node:           127.0.0.1/127.0.0.1
Start Time:     Tue, 16 Aug 2016 09:42:03 -0400
Labels:         name=frontendhttp
Status:         Running
IP:             172.17.0.2
```

Page through the **kubectl describe** content until you see the IP address for the pod. Use that IP address in the next steps.

2.2.3. Check the service

Query the service using the **curl** command, with the IP address and port number, to make sure the service is running. In this example, the address is 172.17.0.2. If you get a "forbidden" errror, disable SELinux using the **setenforce 0** command.

```
# curl 172.17.0.2:80
I love Kubernetes storage!
```

If you see the output shown above, you have a successfully created a working persistent volumer, claim and pod that is using that claim.

2.3. VOLUMES

Kubernetes abstracts various storage facilities as "volumes".

Volumes are defined in the **volumes** section of a pod's definition. The source of the data in the volumes is either:

- a remote NFS share,
- an iSCSI target,
- an empty directory, or
- a local directory on the host.

It is possible to define multiple volumes in the **volumes** section of the pod's definition. Each volume must have a unique name (within the context of the pod) that is used during the mounting procedure as a unique identifier within the pod.

These volumes, once defined, can be mounted into containers that are defined in the **containers** section of the pod's definition. Each container can mount several volumes; on the other hand, a single volume can be mounted into several containers. The **volumeMounts** section of the container definition specifies where the volume should be mounted.

2.3.1. Example

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-web
spec:
  volumes:
    # List of volumes to use, i.e. *what* to mount
    - name: myvolume
      < volume details, see below >
    - name: mysecondvolume
      < volume details, see below >

  containers:
    - name: mycontainer
      volumeMounts:
        # List of mount directories, i.e. *where* to mount
        # We want to mount 'myvolume' into /usr/share/nginx/html
        - name: myvolume
          mountPath: /usr/share/nginx/html/
        # We want to mount 'mysecondvolume' into /var/log
        - name: mysecondvolume
          mountPath: /var/log/
```

2.4. KUBERNETES AND SELINUX PERMISSIONS

Kubernetes, in order to function properly, must have access to a directory that is shared between the host and the container. SELinux, by default, blocks Kubernetes from having access to that shared directory. Usually this is a good idea: no one wants a compromised container to access the host and cause damage. In this situation, though, we want the directory to be shared between the host and the pod without SELinux intervening to prevent the share.

Here's an example. If we want to share the directory **`/srv/my-data`** from the Atomic Host to a pod, we must explicitly relabel **`/srv/my-data`** with the SELinux label **`svirt_sandbox_file_t`**. The presence of this label on this directory (which is on the host) causes SELinux to permit the container to read and write to the directory. Here's the command that attaches the **`svirt_sandbox_file_t`** label to the **`/srv/my-data`** directory:

```
$ chcon -R -t svirt_sandbox_file_t /srv/my-data
```

The following example steps you through the procedure:

Define this container, which uses `/srv/my-data` from the host as the HTML root:

```
{
 "apiVersion": "v1",
 "kind": "Pod",
 "metadata": {
   "name": "host-test"
 },
 "spec": {
   "containers": [
     {
       "name": "host-test",
       "image": "nginx",
       "privileged": false,
       "volumeMounts": [
         {
           "name": "srv",
           "mountPath": "/usr/share/nginx/html",
           "readOnly": false
         }
       ]
     }
   ],
   "volumes": [
     {
       "name": "srv",
       "hostPath": {
         "path": "/srv/my-data"
       }
     }
   ]
 }
}
```

Run the following commands on the container host to confirm that SELinux denies the nginx container read access to /srv/my-data:

```
$ mkdir /srv/my-data
$ echo "Hello world" > /srv/my-data/index.html
$ curl <IP address of the container>
```

You'll get the following output:

```
<html>
<head><title>403 Forbidden</title></head>
...
```

Apply the label svirt_sandbox_file_t to the directory /srv/my-data:

```
$ chcon -R -t svirt_sandbox_file_t /srv/my-data
```

Use `curl` to access the container and to confirm that the label has taken effect:

```
$ curl <IP address of the container>
Hello world
```

If the **curl** command returned "Hello world", the SELinux label has been properly applied.

BZ#1222060 tracks this issue.

2.5. NFS

In order to test this scenario, you must already have prepared NFS shares. In this example, you will mount the NFS shares into a pod.

The following example mounts the NFS share into **/usr/share/nginx/html/** and runs the **nginx** webserver.

Create a file named `nfs-web.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-web
spec:
  volumes:
    - name: www
      nfs:
        # Use real NFS server address here.
        server: 192.168.100.1
        # Use real NFS server export directory.
        path: "/www"
        readOnly: true
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: tcp
      volumeMounts:
          # 'name' must match the volume name below.
          - name: www
            # Where to mount the volume.
            mountPath: "/usr/share/nginx/html/"
```

Start the pod:

```
$ kubectl create -f nfs-web.yaml
```

Kubernetes mounts **`192.168.100.1:/www`** into **`/usr/share/nginx/html/`**` inside the nginx container and runs it.

Confirm that the webserver receives data from the NFS share:

```
$ curl 172.17.0.6
Hello from NFS
```

Mount options in Kubernetes

As of 16 Feb 2016, the only mount options that are supported are **-ro**.

See https://github.com/kubernetes/kubernetes/issues/17226 to follow the progress of the inclusion of mount options in Kubernetes.

Troubleshooting

403 Forbidden error: if you receive a "403 Forbidden" response from the webserver, make sure that SELinux allows Docker containers to read data over NFS by running the following command:

```
$ setsebool -P virt_use_nfs 1
```

2.6. ISCSI

Make sure that the iSCSI target is properly configured. Make sure that all Kubernetes nodes have sufficient privileges to attach

a LUN from the iSCSI target.

Create a file named `iscsi-web.yaml`, containing the following pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: iscsi-web
spec:
  volumes:
    - name: www
      iscsi:
        # Address of the iSCSI target portal
        targetPortal: "192.168.100.98:3260"
        # IQN of the portal
        iqn: "iqn.2003-01.org.linux-iscsi.iscsi.x8664:sn.63b56adc495d"
        # LUN we want to mount
        lun: 0
        # Filesystem on the LUN
        fsType: ext4
        readOnly: false
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: tcp
      volumeMounts:
```

```
          # 'name' must match the volume name below.
          - name: www
            # Where to mount he volume.
            mountPath: "/usr/share/nginx/html/"
```

Create the pod:

```
$ kubectl create -f iscsi-web.yaml
```

Check that the web server uses data from the iSCSI volume:

```
$ curl 172.17.0.6
Hello from iSCSI
```

2.7. GOOGLE COMPUTE ENGINE

Google Compute Engine Persistent Disk (GCE PD)

If you are running your cluster on Google Compute Engine, you can use a Persistent Disk as your persistent storage source. In the following example, you will create a pod which serves html content from a GCE PD.

If you have the GCE SDK set up, create a persistent disk using the following command:

```
$ gcloud compute disks create --size=250GB {Persistent Disk Name}
```

Otherwise you can create the disk through the GCE web interface. If you want to set up the GCE SDK follow the instructions here.

Create a file named `gce-pd-web.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: gce-web
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: tcp
      volumeMounts:
        - name: html-pd
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: html-pd
```

```
      gcePersistentDisk:
        # Add the name of your persistent disk below
        pdName: {Persistent Disk Name}
        fsType: ext4
```

Create the pod:

```
$ kubectl create -f gce-pd-web.yaml
```

Kubernetes will create the pod and attach the disk but it will not format and mount it. This is due to a bug which will be fixed in future versions of Kubernetes. To work around this proceed to the next step.

The disk will be attached to the virtual machine and a device will appear under `/dev/disk/by-id/` with the name `scsi-0Google_PersistentDisk_{Persistent Disk Name}`. If this disk is already formatted and contains data proceed to the next step otherwise run the following command as root to format it:

```
$ mkfs.ext4 /dev/disk/by-id/scsi-0Google_PersistentDisk_{Persistent
Disk Name}
```

When the disk is formatted, mount it in the location expected by Kubernetes. Run the following commands as root:

```
# mkdir -p /var/lib/kubelet/plugins/kubernetes.io/gce-
pd/mounts/{Persistent Disk Name} && mount /dev/disk/by-id/scsi-
0Google_PersistentDisk_{Persistent Disk Name}
/var/lib/kubelet/plugins/kubernetes.io/gce-pd/mounts/{Persistent Disk
Name}
```

Note

The **`mkdir`** command and the mount command must be run in quick succession as above because Kubernetes clean up will remove the directory if it sees nothing mounted there.

Now that the disk is mounted it must be given the correct SELinux context. As root run the following:

```
$ sudo chcon -R -t svirt_sandbox_file_t
/var/lib/kubelet/plugins/kubernetes.io/gce-pd/mounts/{Persistent Disk
Name}
```

Create some data for your web server to serve:

```
$ echo "Hello world" >  /var/lib/kubelet/plugins/kubernetes.io/gce-
pd/mounts/{Persistent Disk Name}/index.html
```

You should now be able to get HTML content from the pod:

```
$ curl {IP address of the container}
Hello World!
```

CHAPTER 3. MIGRATING FROM AN EARLIER VERSION OF KUBERNETES

If you already have a Kubernetes all-in-one system up and running from an earlier release of RHEL or RHEL Atomic, you can migrate your setup to the latest release using the procedure described here. If you have a Kubernetes cluster installed from an earlier RHEL release, that configuration is no longer supported.

The main issue in migrating from an earlier RHEL Atomic Host release is that three Kubernetes services on the master (kube-scheduler, kube-apiserver, and kube-controller-manager) were dropped from the RHEL Atomic Host distribution. So, to upgrade to a new release, you need to transition to containerized versions of those services on your Kubernetes master.

Here's how you can go about upgrading your Kubernetes all-in-one master and node to a later release of RHEL or RHEL Atomic:

1. **Stop Kubernetes services**: Run these commands from the master to stop and disable Kubernetes services:

   ```
   # for SERVICES in kube-apiserver kube-controller-manager kube-
   scheduler; do
       systemctl stop $SERVICES
       systemctl disable $SERVICES
       systemctl is-active $SERVICES
       done
   # systemctl stop kubelet
   ```

2. **Upgrade each system**: Upgrading is done differently on RHEL Server and RHEL Atomic systems:

   On a RHEL Atomic Host, system type the following:

   ```
   # atomic host upgrade
   # reboot
   ```

   On a RHEL Server system, type the following:

   ```
   # yum upgrade -y
   ```

3. **Create manifest files (optional)**: You can create files named apiserver.pod.json, controller-manager.pod.json, and scheduler.pod.json files with content described earlier in this document. You may need to modify those files based on setting in your current apiserver, controller-manager, and scheduler configuration files (in **/etc/kubernetes**). Copy the new json files to the **/etc/kubernetes/manifests/** directory. If those files don't exist, however, your Kubernetes master services will use the configuration files from your previous release that are contained in the **/etc/kubernetes** directory.

4. **Reconfigure kubelet**: To have kubelet use the new manifest files, add a KUBELET_ARGS argument to the **/etc/kubernetes/kubelet** file that points to that directory, as described earlier (**--config=/etc/kubernetes/manifests/**).

5. **Pull Kubernetes containers**: On the master, before you try to start the containerized Kubernetes services, you should pull their docker images to the local system. Run the following commands to do that:

```
# docker pull rhel7/kubernetes-controller-mgr
# docker pull rhel7/kubernetes-apiserver
# docker pull rhel7/kubernetes-scheduler
```

6. **Restart etcd and kubelet**: Restarting etcd and kubelet services results in the new manifest files being used to start up the containerized versions of kube-apiserver, kube-controller-manager, and kube-scheduler.

```
# systemctl restart etcd kubelet
```

At this point, the three Kubernetes containers should have replaced the systemd versions of those services on your system.

CHAPTER 4. TROUBLESHOOTING KUBERNETES

4.1. OVERVIEW

Kubernetes is a utility that makes it possible to deploy and manage sets of docker-formatted containers that run applications. This topic explains how to troubleshoot problems that arise when creating and managing Kubernetes pods, replication controllers, services, and containers.

For the purpose of illustrating troubleshooting techniques, this topic uses the containers and configuration deployed in the Get Started Orchestrating Containers with Kubernetes chapter. Techniques described here should apply to Kubernetes running on Red Hat Enterprise Linux Server and RHEL Atomic Host systems.

4.2. UNDERSTANDING KUBERNETES TROUBLESHOOTING

Before you begin troubleshooting Kubernetes, you should have an understanding of the Kubernetes components being investigated. These include:

- **Master**: The system from which you manage your Kubernetes environment.
- **Nodes**: One or more systems on which containers are deployed by Kubernetes (nodes were previously called minions).
- **Pods**: A pod defines one or more containers to run, as well as options to the docker run command for each container and labels to define the location of services.
- **Services**: A service allows a container within a Kubernetes environment to find an application provided by another container by name (label), without knowing its IP address.
- **Replication controllers**: A replication controller lets you designate that a certain number of pods should be running. (New pods are started until the required number is reached and if a pod dies, a new pod is run to replace it.)
- **Networking (flanneld)**: The flanneld service lets you configure IP address ranges and related setting to be used by Kubernetes. This feature is optional. yaml and json files: The Kubernetes elements we'll work with are actually created from configuration files in yaml or json formats. this topic focuses primarily on yaml-formatted files.

You will troubleshoot the components just described using these commands in particular:

- **kubectl**: The **kubectl** command (run from the master) lets you create, delete, get (list information) and perform other actions on Kubernetes pods, services and replication controllers. You'll use this command to test your yaml/json files, as well as see the state of the different Kubernetes components.
- **systemctl**: Specific **systemd** services must be configured with Kubernetes to facilitate communications between master and nodes. Those services must also be active and enabled.
- **journalctl**: You can use the **journalctl** command to check Kubernetes systemd services to follow the processing of those services. You can run it on both the master and nodes to check for Kubernetes failures on those systems. All daemon logging in kubernetes uses the systemd journal.

- **etcdctl** or **curl**: The **etcd** daemon manages the storage of information for the Kubernetes cluster. This service can run on the master or on some other system. You can use the etcdctl command in RHEL or RHEL Atomic Host systems to query that information. You also can use the curl command instead to query the etcd service.

4.3. PREPARING CONTAINERIZED APPLICATIONS FOR KUBERNETES

Some of the things you should consider before deploying an application to Kubernetes are described below.

4.3.1. Networking Constraints

All Applications are not equally kuberizable, because there are limitations on the type of applications that can be run as Kubernetes services. In Kubernetes, a service is a load balanced proxy whose IP address is injected into the iptables of clients of that service. Therefore, you should verify that the application you intend to "kuberize":

- Can support network address translation or NAT-ing across its subprocesses.
- Does not require forward and reverse DNS lookup. Kubernetes does not provide forward or reverse DNS lookup to the clients of a service.

If neither of these restrictions apply, or if the user can disable these checks, you can continue on.

4.3.2. Preparing your Containers

Depending on the type of software you are running you may wish to take advantage of some predefined environment variables that are provided for clients of Kubernetes services.

For example, given a service named **db**, if you launch a Pod in Kubernetes that uses that service, Kubernetes will inject the following environment variables into the containers in that pod:

```
DB_SERVICE_PORT_3306_TCP_PORT=3306
DB_SERVICE_SERVICE_HOST=10.254.100.1
DB_SERVICE_PORT_3306_TCP_PROTO=tcp
DB_SERVICE_PORT_3306_TCP_ADDR=10.254.100.1
DB_SERVICE_PORT_3306_TCP=tcp://10.254.100.1:3306
DB_SERVICE_PORT=tcp://10.254.100.1:3306
DB_SERVICE_SERVICE_PORT=3306
```

NOTE: Notice that the service name (db) is capitalized in the variables (DB). If there were dashes (-) in the name, they would be converted to underscores (_).

To see these and other shell variables, use **docker exec** to open a shell to the active container and run **env** to see the shell variables:

```
# docker exec -it <container_ID> /bin/bash
[root@e7ea67..]# env
...
WEBSERVER_SERVICE_SERVICE_PORT=80
KUBERNETES_RO_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT=443
KUBERNETES_RO_PORT_80_TCP_PORT=80
```

```
KUBERNETES_SERVICE_HOST=10.254.255.128
DB_SERVICE_PORT_3306_TCP_PORT=3306
DB_SERVICE_SERVICE_HOST=10.254.100.1
WEBSERVER_SERVICE_PORT_80_TCP_ADDR=10.254.100.50
...
```

When starting your client applications you may want to leverage those variables. If you are debugging communications problems between containers, viewing these shell variables is a great way to see each container's view of the addresses of services and ports.

4.4. DEBUGGING KUBERNETES

Before you start debugging Kubernetes, it helps to have a high level of understanding of how Kubernetes works. When you submit an application to Kubernetes, here's generally what happens:

1. Your **kubectl** command line is sent to the kube-apiserver (on the master) where it is validated.
2. The kube-scheduler process (on the master) reads your yaml or json file and assigns pods to nodes (nodes are systems running the kubelet service).
3. The kublet service (on a node) converts the pod manifest into one or more **docker run** calls.
4. The **docker run** command tries to start up the identified containers on available nodes.

So, to debug a kubernetes deployed app, you need to confirm that:

1. The Kubernetes service daemon (systemd) processes are running.
2. The yaml or json submission is valid.
3. The kubelet service is receiving a work order from the kube-scheduler.
4. The kubelet service on each node is able to successfully launch each container with docker.

Note

The above list is missing the kube-controller-manager which is important if you do things like create a replication controller, but you see no pods being managed by it. Or you have registered nodes with the cluster, but you are not getting information about their available resources, etc.

Also note, there is a movement upstream to an all-in-one hyperkube binary, so terminology here may need to change in the near future.

4.4.1. Inspecting and Debugging Kubernetes

From the Kubernetes master, inspect the running Kubernetes configuration. We'll start by showing you how this configuration should look when everything is working. Then we'll show you how to the setup might break in various ways and how you can go about fixing it.

4.4.2. Querying the State of Kubernetes

Using **kubectl** is the simplest way to manually debug the process of application submission, service creation, and pod assignment. To see what pods, services, and replication controllers are active, run these commands on the master:

```
# kubectl get pods
POD             IP          CONTAINER(S)     IMAGE(S)  HOST
LABELS                                         STATUS
4e04dd3b-c...   10.20.29.3  apache-frontend  webwithdb
node2.example.com/  name=webserver,selectorname=webserver,uses=db
Running
5544eab2-c...   10.20.48.15 apache-frontend  webwithdb
node1.example.com/  name=webserver,selectorname=webserver,uses=db
Running
1c971a09-c...   10.20.29.2  db               dbforweb
node2.example.com   name=db,selectorname=db
Running
1c97a755-c...   10.20.48.14 db               dbforweb
node1.example.com/  name=db,selectorname=db
Running
# kubectl get services
NAME               LABELS                                    SELECTOR
IP                  PORT
webserver-service  name=webserver
name=webserver  10.254.100.50       80
db-service         name=db                                   name=db
10.254.100.1        3306
kubernetes         component=apiserver,provider=kubernetes
10.254.92.19        443
kubernetes-ro      component=apiserver,provider=kubernetes
10.254.206.141      80
# kubectl get replicationControllers
CONTROLLER             CONTAINER(S)        IMAGE(S)
SELECTOR                 REPLICAS
webserver-controller   apache-frontend     webwithdb
selectorname=webserver   2
db-controller          db                  dbforweb
selectorname=db          2
```

Here's information to help you interpret this output:

- Pods are either in Waiting or Running states. The fact that all four pods are running here is a good sign.
- The replication controller successfully started two apache-frontend and two db containers. They were distributed across node1 and node2.
- The **uses** label for apache-frontend lets that container find the db container through the db-service Kubernetes service.
- The services listing identifies the IP address and port number for each service that can be requested from pods by each service's label name.
- The kubernetes and kubernetes-ro services provide access to the kube-apiserver systemd service.

If something goes wrong in the process of getting to this state, the following sections will help you troubleshoot problems.

4.5. TROUBLESHOOTING KUBERNETES SYSTEMD SERVICES

Kubernetes is implemented using a set of service daemons that run on Kubernetes masters and nodes. If these systemd services are not working properly, you will experience failures. Things you should know about avoiding or fixing potential problems with Kubernetes systemd services are described below.

4.5.1. Checking that Kubernetes systemd Services are Up

A Kubernetes cluster that consists of a master and one or more nodes (minions) needs to initialize a particular set of systemd services. You should verify that the following services are running on the master and on each node:

- **Start Master first**: The services on the master should come before starting the services on the nodes. The nodes will not start up properly if the master is not already up.
- **Master services**: Services include: kube-controller-manager, kube-scheduler, flanneld, etcd, and kube-apiserver. The flanneld service is optional and it is possible to run the etcd services on another system.
- **Node services**: Services include: docker kube-proxy kubelet flanneld. The flanneld service is optional.

Here's how you verify those services on the master and each node:

Master: On your kubernetes master server, this will tell you if the proper services are active and enabled (flanneld may not be configured on your system):

```
# for SERVICES in etcd flanneld kube-apiserver kube-controller-manager
kube-scheduler;
    do echo --- $SERVICES --- ; systemctl is-active $SERVICES ;
    systemctl is-enabled $SERVICES ; echo "";  done
--- etcd ---
active
enabled
--- flanneld ---
active
enabled
--- kube-apiserver ---
active
enabled
--- kube-controller-manager ---
active
enabled
--- kube-scheduler ---
active
enabled
```

Nodes: On each node, make sure the proper services are active and enabled:

```
# for SERVICES in flanneld docker kube-proxy.service kubelet.service; \
do echo --- $SERVICES --- ; systemctl is-active $SERVICES ; \
```

```
systemctl is-enabled $SERVICES ; echo "";  done
--- flanneld ---
active
enabled
--- docker ---
active
enabled
--- kube-proxy.service ---
active
enabled
--- kubelet.service ---
active
enabled
```

If any of the master or node systemd services are disabled or failed, here's what to do:

- Try to enable or activate the service.
- Check the systemd journal on the system where a service is failing and look for hints on how to fix the problem. One way to do that is to use the journalctl with the command representing the service. For example:

  ```
  # journalctl -l -u kubelet
  # journalctl -l -u kube-apiserver
  ```

- If the services still don't start, check that each service's configuration file is set up properly.

4.5.2. Checking Firewall for Kubernetes

There is no iptables or firewalld service installed on RHEL Atomic Host. So, by default, there are no firewall filter rules blocking access to Kubernetes services. However, if you have a firewall running on a RHEL host or if you have added iptables firewall rules to your Kubernetes master or nodes to filter incoming ports, you need to make sure that the ports that need to be exposed on those systems are not blocked.

The following is the output of a **netstat** command, showing which ports Kubernetes and related services are listening on a Kubernetes nodes:

```
# netstat -tupln
tcp6       0      0 :::10249          :::*          LISTEN
125528/kube-proxy
tcp6       0      0 :::10250          :::*          LISTEN
125536/kubelet
```

NOTE: The kube-proxy service listens on random ports. This is not a problem on RHEL Atomic systems, since there is on filtering firewall service used by default. However, if you add a firewall to RHEL Atomic or use a default RHEL system, you can request that kube-proxy listen on specific ports in the service definition and then open those ports in the firewall.

Here is **netstat** output on a Kubernetes master:

```
tcp        0      0 192.168.122.249:7080  0.0.0.0:* LISTEN   636/kube-
apiserver
tcp6       0      0 :::8080               :::*      LISTEN
636/kube-apiserver
tcp        0      0 127.0.0.1:10252       0.0.0.0:* LISTEN
```

```
7541/kube-controller
tcp        0      0 127.0.0.1:10251       0.0.0.0:* LISTEN
7590/kube-scheduler
tcp6       0      0 :::4001               :::*      LISTEN   941/etcd
tcp6       0      0 :::7001               :::*      LISTEN   941/etcd
```

The output in the third column shows the IP addresses and port number that each service is listening on. (`:::` represents all interfaces) Open ports to each of those services.

4.5.3. Checking Kubernetes yaml or json Files

You set up your Kubernetes environment (pods, services, and replication controllers) by loading information from yaml or json files using the **kubectl create** command. Failures can result from those files being improperly formatted or missing needed information.

The following sections contain tips for fixing problems that occur from broken yaml or json files.

4.5.3.1. Troubleshooting Kubernetes Service Creation

A Kubernetes service (created with **kubectl**), attaches an IP address and port to a label. A pod that needs to use that service can refer to that service by the label, so it doesn't need to know the IP address and port numbers directly. The following is an example of a service file named db-service.yaml, followed by a list of problems that can occur when you try to create a service:

```
id: "db-service"
kind: "Service"
apiVersion: "v1"
port: 3306
portalIP: "10.254.100.1"
selector:
  name: "db"
labels:
  name: "db"
```

```
# kubectl create -f db-service
# kubectl get services
NAME          LABELS                                    SELECTOR  IP
PORT
db-service    name=db                                   name=db
10.254.100.1   3306
kubernetes-ro component=apiserver,provider=kubernetes     10.254.186.33
80
kubernetes    component=apiserver,provider=kubernetes     10.254.198.9
443
```

NOTE: If you don't see the kubernetes-ro and kubernetes services, try restarting the kube-scheduler systemd service (**systemctl restart kube-scheduler.service**).

If you don't see output similar to what was just shown, read the following:

- If the service seemed to create successfully, but the LABELS an SELECTOR were not set, the output might look as follows:

```
# kubectl get services
NAME         LABELS     SELECTOR  IP             PORT
db-service             10.254.100.1   3306
```

Check that the name: fields under selector: and labels: are each indented two spaces. In this case I deleted the two blank spaces before each **name: "db"** line and their values were not used by **kubectl**.

- If you forget that you have already created a Service and try to create it again or if some other service has already allocated an IP address you identified in your service yaml, your new attempt to create the service will result in this message:

```
create.go:75] service "webserver-service" is invalid: spec.portalIP:
    invalid value '10.254.100.50': IP 10.254.100.50 is already
allocated
```

You can either use a different IP address or stop the service that is currently consuming that port, if you don't need that service.

- The following error noting that the "Service" object isn't registered can occur for a couple of reasons:

```
7338 create.go:75] unable to recognize "db-service.yaml": no object
named "Services" is registered
```

In the above example, "Service" was misspelled as "Services". If it does correctly say "Service", then check that the apiVersion is correct. A similar error occurred when the invalid value "v99" was used as the apiVersion. Instead of saying "v99" doesn't exist, it says it can't find the object "Service".

  - Here is a list of error messages that occur if any of the fields from the above example is missing:

    - **id: missing**: service "" is invalid: name: required value ''
    - **kind: missing**: unable to recognize "db-service.yaml": no object named "" is registered
    - **apiVersion: missing**: service "" is invalid: [name: required value '', spec.port: invalid value '0']
    - **port: missing**: service "db-service" is invalid: spec.port: invalid value '0'
    - **portalIP: missing**: No error is reported because portalIP is not required
    - **selector: missing**: No error is reported, but SELECTOR field is missing and service may not work.
    - **labels: missing**: Not an error, but LABELS field is missing and service may not work.

4.5.3.2. Troubleshooting Kubernetes Replication Controller and Pod creation

A Kubernetes Pod lets you associate one or more containers together, assign run options to each container, and manage those containers together as a unit. A replication controller lets you

designate how many of the pods you identify should be running. The following is an example of a yaml file that defines a Web server pod and a replications controller that ensures that two instances of the pod are running.

```
id: "webserver-controller"
kind: "ReplicationController"
apiVersion: "v1"
metadata:
  name: "webserver-controller"
spec:
  replicas: 1
  selector:
    name: "webserver"
  template:
    spec:
      containers:
        - name: "apache-frontend"
          image: "webwithdb"
          ports:
            - containerPort: 80
    metadata:
      labels:
        name: "webserver"
        uses: db
  labels:
    name: "webserver"
```

```
# kubectl create -f webserver-service.yaml
# kubectl get pods
POD         IP          CONTAINER(S)     IMAGE(S)    HOST
       LABELS                                          STATUS
f28980d...  10.20.48.4  apache-frontend  webwithdb   node1.example.com/
       name=webserver,selectorname=webserver,uses=db      Running
f28a0a8...  10.20.29.9  apache-frontend  webwithdb   node2.example.com/
       name=webserver,selectorname=webserver,uses=db      Running
# kubectl get replicationControllers
CONTROLLER           CONTAINER(S)    IMAGE(S)   SELECTOR
REPLICAS

webserver-controller apache-frontend webwithdb  selectorname=webserver
2
```

NOTE: I truncated the pod name and wrapped the long lines in the output above.

If you don’t see output similar to what was just shown, read the following:

- **id: missing**: If a generated set of numbers and letters appears in the CONTROLLER column instead of "webserver-controller", your yaml file is probably missing the id line.
- **apiVersion set wrong**: If you see the message "unable to recognize "webserver-rc.yaml": no object named "ReplicationController" is registered", you may have an invalid apiVersion value or misspelled ReplicationController.
- **selectorname: missing**: If you see the message "replicationController "webserver-controller" is invalid: spec.selector: required value 'map[]'", there is no selectorname set after the replicaSelector line. If the selectorname is not indented properly, you will see a message like,

"unable to get type info from "webserver-rc.yaml": couldn't get version/kind: error converting YAML to JSON: yaml: line 7: did not find expected key."

4.6. TROUBLESHOOTING TECHNIQUES

If you want to look deeper into what is going on with your Kubernetes cluster, see the following techniques for investigating further.

4.6.1. Crawling and fixing the etcd database

The etcd service provides the database that Kubernetes uses to coordinate information across the cluster. There are ways to view the database directly and fix problems in it (or clear the database if it is beyond repair).

Displaying data from the etcd database: You can query most information you need from the etcd database using kubectl get commands. However, if this database seems to be inconsistent with the way you believe your configuration should be, you can directly query the etcd database using the etcdctl command.

Use the **etcdctl** command with the ls option to list the directory structure of the database. To get values, use the get option. For example, to see the root of the database, type the following:

```
# etcdctl ls /
/registry
```

To list information associated with the etcd database, type this:

```
# etcdctl ls /registry/

/registry/namespaces
/registry/ranges
/registry/serviceaccounts
/registry/services

...
```

To see the data associated with a particular entry, type the following:

```
# etcdctl get /registry/namespaces/default | python -mjson.tool
{
    "apiVersion": "v1",
    "kind": "Namespace",
    "metadata": {
        "creationTimestamp": "2016-10-24T12:05:11Z",
        "name": "default",
        "uid": "1d6efb5f-99e2-11e6-8f4b-525400585a9f"
    },
    "spec": {
        "finalizers": [
            "kubernetes"
        ]
    },
```

```
    "status": {
        "phase": "Active"
    }
}
```

The output above is piped to a python json.tool formatting module, to make it easier to read.

NOTE: Instead of the etcdctl command, you can use the **curl**. For example, to see the root of the database with **curl**, use this instead of the **etcdctl ls /** command: **curl -L http://localhost:2379/v2/keys/ | python -mjson.tool**. Use that form of the curl command to display both directory and key values. If you believe that a node is not able to connect to the etcd service on the master, you could use the following **curl** command to test that connection from the node:

```
# curl -s -L http://localhost:2379/version
{"etcdserver":"2.3.7","etcdcluster":"2.3.0"}
```

Fixing the etcd database: It is possible to correct problems with your etcd database if information gets out of sync. There is are **etcdctl update** and **etcdctl set** commands for changing the contents of a key. However, if you are not careful, changing these values can cause more problems than they fix.

However, if your etcd database become completely unuseable, you can clear it and start over again. The way to do that is to run the **etcd** daemon with the **-f** option.

WARNING: Before you clear the etcd database, try using **kubectl delete** command to try to remove the offending services, pods, replicationControllers or minions. If you still feel you need to clear the database, keep in mind that if you do so, you need to recreate everything from scratch.

To clear the etcd database, type the following:

```
# etcd -f
```

4.6.2. Deleting Kubernetes components

How you stop and delete components in Kubernetes matters. Because Kubernetes is designed to get things to a particular state, simply deleting a container or a pod will often just cause another one to be started.

If you do delete components out of order, here's what you can expect:

- **I deleted a pod, but it started up again**: If you don't stop the replication controllers first, the pods will be restarted. Stop the replication controllers (**kubectl delete replicationControllers webserver-controller**), then stop the pods.
- **I stopped and deleted a container, but it started up again**: With a Kubernetes cluster, you should not stop a container directly with **docker stop**. The replication controller will start a new container to restart the one you stopped.

4.6.3. Pods Stuck in the "WAITING" state.

PODS can be stuck in the waiting state for some time period. Here are some possible causes:

- **Pulling the Docker image is taking a while**: To confirm this, you can ssh directly into the minion which the pod is assigned, and run:

  ```
  # journalctl -f -u docker
  ```

This should show logs of docker pulling down your image. Note requests to pull dockerhub images may fail intermittently, but the kubelets will continue retrying.

- **PODs are unassigned**: If a pod remains unassigned, confirm that nodes are available to the master by running kubectl get minions. It is possible that the node may just be down or otherwise unreachable. Unassigned pods can also result from setting the replication count higher then what the cluster can provide.
- **Container Pod dies right after starting**: In some cases, if the Dockerfile you created is not written properly to start a service, or the docker CMD operation is failing, you might see the POD immediately dying after it starts. Try testing the container image with a **docker run** command, to make sure that the container itself isn't broken.
- **Check output from container**: Messages output from containers can be viewed with the kubectl log command. This can be useful for debugging problems with the applications running within the container. Here is how to list available pods and view log messages for the one you want:

```
# kubectl get pods
POD                                    IP         CONTAINER(S)
IMAGE(S)            HOST                LABELS
STATUS
e1f4b268-e87d-11e4-926b-5254001aa4ee  10.20.24.3 db
dbforweb node1.example.com/   name=db,selectorname=db
Running
# kubectl log e1f4b268-e87d-11e4-926b-5254001aa4ee
2015-04-28T16:09:36.953130209Z 150428 12:09:36 mysqld_safe Logging
to '/var/log/mariadb/mariadb.log'.
2015-04-28T16:09:37.137064742Z 150428 12:09:37 mysqld_safe Starting
mysqld daemon with databases from /var/lib/mysql
```

- **Check container output from docker**: Some errors don't percolate all the way up to the kubelet. You can look directly in the docker logs for an exited container to observe why this might be happening. Here's how:

  - Log into the node that's having trouble running a container
  - Run this command to look for an exited run:

    ```
    # docker ps -a
    61960bda2927  rhel7/rhel-tools:latest "/usr/bin/bash" 47 hours
    ago
          Exited (0) 43 hours ago      myrhel-tools4
    ```

  - Check all the output from the container with docker logs:

    ```
    # docker logs 61960bda2927
    ```

You should be able to see the entire output from the container session. So, for example, if you opened a shell in a container, you will see all the commands you ran from that shell when you run docker logs.

CHAPTER 5. YAML IN A NUTSHELL

5.1. OVERVIEW

YAML — which stands for “YAML Ain’t Markup Language” — is a human-friendly data serialization standard, similar in scope to JSON (Javascript Object Notation). Unlike JSON, there are only a handful of special characters used to represent mappings and bullet lists, the two basic types of structure, and indentation is used to represent substructure.

5.2. BASICS

The YAML format is line-oriented, with two top-level parts, *HEAD* and *BODY*, separated by a line of three hyphens.

```
HEAD
---
BODY
```

The head holds configuration information and the body holds the data. this topic does not discuss the configuration aspect; all the examples here show only the data portion. In such cases, the “---” is optional.

The most basic data element is one of:

1. A number
2. A Unicode string
3. A boolean value, spelled either true or false
4. In a key/value pair context, a missing value is parsed as nil

Comments start with a **“#”** (hash, U+23) and go to the end of the line.

Indentation is whitespace at the start of the line. You are strongly encouraged to avoid **TAB** (U+09) characters and use a series of **SPACE** (U+20) characters, instead.

5.3. LISTS

A list is a series of lines, each beginning with the same amount of indentation, followed by a hyphen, followed by a list element. Lists cannot have blank lines. For example, here is a list of three elements, the third of which has a comment:

```
- top shelf
- middle age
- bottom dweller   # stability is important
```

Note: The third element is the string “bottom dweller” and does not include the whitespace between “dweller” and the comment.

WARNING: Lists cannot normally nest directly; there should be an intervening mapping (described below). In the following example, the list’s second element seems, due to the indentation (two **SPACE** characters), to host a sub-list:

```
- top
- middle
  - highish middle
  - lowish middle
- bottom
```

In reality, the second element is actually parsed as a single string. The input is equivalent to:

```
- top
- middle - highish middle - lowish middle
- bottom
```

The newlines and indentation are normalized to a single space.

5.4. MAPPINGS

To write a mapping (also known as an associative array or hash table), use a **":"** (colon, **U+3A**) followed by one or more **SPACE** characters between the key and the value:

```
square:   4
triangle: 3
pentagon: 5
```

All keys in a mapping must be unique. For example, this is invalid YAML for two reasons: the key square is repeated, and there is no space after the colon following **triangle**:

```
square:  4
triangle:3       # invalid key/value separation
square:  5       # repeated key
```

Mappings can nest directly, by starting the sub-mapping on the next line with increased indentation. In the next example, the value for key **square** is itself a mapping (keys **sides** and **perimeter**), and likewise for the value for key **triangle**. The value for key **pentagon** is the number 5.

```
square:
  sides:     4
  perimeter: sides * side-length
triangle:
  sides:     3
  perimeter: see square
pentagon:    5
```

The following example shows a mapping with three key/value pairs. The first and third values are **nil**, while the second is a list of two elements, “highish middle” and “lowish middle”.

```
top:
middle:
  - highish middle
  - lowish middle
bottom:
```

5.5. QUOTATION

Double-quotation marks (also known as “double-quotes”) are useful for forcing non-string data to be interpreted as a string, for preserving whitespace, and for suppressing the meaning of colon. To include a double-quote in a string, escape it with `**“\”** (backslash, **U+5C**). In the following example, all keys and values are strings. The second key has a colon in it. The second value has two spaces both preceding and trailing the visible text.

```
"true" : "1"
"key the second (which has a \":\" in it)" : "  second  value  "
```

For readability when double-quoting the key, you are encouraged to add whitespace before the colon.

5.6. BLOCK CONTENT

There are two kinds of block content, typically found in the value position of a mapping element: newline-preserving and folded. If a block begins with **“|”** (pipe, **U+7C**), the newlines in that block are preserved. If it begins with **“>”** (greater-than, **U+3E**), consecutive newlines are folded into a single space. The following example shows both kinds of block content as the values for keys **good-bye** and **anyway**.

```
hello: world

good-bye: |
    first line

    third
    fourth and last


anyway: >
    nothing is guaranteed
    in life
lastly:
```

Using \n (backslash-n) to indicate newline, the values for keys **good-bye** and **anyway** are, respectively:

```
first line\n\nthird\nfourth and last\n

nothing is guaranteed in life\n
```

Note that the newlines are preserved in the **good-bye** value but folded into a single space in the **anyway** value. Also, each value ends with a single newline, even though there are two blank lines between “fourth and last” and “anyway”, and no blank lines between “in life” and “lastly”.

5.7. COMPACT REPRESENTATION

Another, more compact, way to represent lists and mappings is to begin with a start character, finish with an end character, and separate elements with **“,”** (comma, **U+2C**).

For lists, the start and end characters are **“[”** (left square brace, **U+5B**) and **“]”** (right square brace, **U+5D**), respectively. In the following example, the values in the mapping are identical:

```
one:
  - echo
  - hello, world!
two: [ echo, "hello, world!" ]
```

Note: The double-quotes around the second list element of the second value; they prevent the comma from being misinterpreted as an element separator. (If we remove them, the list would have three elements: "echo", "hello" and "world!".)

For mappings, the start and end characters are **“{”** (left curly brace, **U+7B**) and **“}”** (right curly brace, **U+7D**), respectively. In the following example, the values of both one and two are identical:

```
one:
  roses: red
  violets: blue

two: { roses: red, violets: blue }
```

5.8. ADDITIONAL INFORMATION

There is much more to YAML, not described in this topic: directives, complex mapping keys, flow styles, references, aliases, and tags. For detailed information, see the official YAML site, specifically the latest (version 1.2 at time of writing) specification.

CHAPTER 6. KUBERNETES CONFIGURATION

6.1. OVERVIEW

Kubernetes reads YAML files to configure services, pods and replication controllers. This document describes the similarities and differences between these areas and details the names and expected data types of the various files.

6.2. DESIGN STRATEGY

Kubernetes uses environment variables whose names are partially specified by the service configuration, so normally you would design the services first, followed by the pods, followed by the replication controllers. This ordering is “outside-in”, moving from the user-facing portion to the internal management portion.

Of course, you are free to design the system in any order you wish. However, you might find that you require more iterations to arrive at a good set of configuration files if you don’t start with services.

6.3. CONVENTIONS

In this document, we say **field name** and **field value** instead of **key** and **value**, respectively. For brevity and consistency with upstream Kubernetes documentation, we say **map** instead of **mapping**. As the field value can often be a complex structure, we call the combination of field name and value together a ***<field>* tree** or ***<field>* structure**, regardless of the complexity of the field value. For example, here is a map, with two top-level structures, **one** and **`two`**:

```
one:
  a: [ x, y, z ]
  b: [ q, r, s ]
two: 42
```

The **one** tree is a map with two elements, the **a** tree and the **b** tree, while the **`two`** tree is very simple: field name is **`two`** and field value is 42. The field values for both **a** and **b** are lists.

6.3.1. Extended Types

We conceptually extend the YAML type system to include some sub-types of **`string`** and some more precise sub-types of **`number`**:

`symbol`

This is a string that has no internal whitespace, comma, colon, curly-braces or square-braces. As such, it does not require double-quotes. For example, all the field names and the first two values in the following map are symbols.

```
a: one-is-a-lonely-number
b: "two-s-company"
c: 3's a crowd
```

Note that the field value for **b** is a symbol even though it is written with double-quotes. The double-quotes are unnecessary, but not invalid. The other way to think about it is: If you **need** to use double-quotes, you are not writing a symbol.

`enum`

This is a symbol taken from a pre-specified, finite, set.

`v4addr`

This is an IPv4 address in dots-and-numbers notatation (e.g., **`127.0.0.1`**).

`opt-v4addr`

This is either a **`v4addr`** or the symbol **`None`**.

`integer`

This is a number with neither fractional part nor decimal point.

`resource-quantity`

This is a number optionally followed by a scaling suffix.

Suffix	Scale	Example	Equivalence
(none)	**`1`**	**`19`**	19 (**`19 * 1`**)
`m`	**`1e-3`**	**`200m`**	0.2 (**`200 * 1e-3`**)
`K`	**`1e+3`**	**`4K`**	4000 (**`4 * 1e+3`**)
`Ki`	**`2^10`**	**`4Ki`**	4096 (**`4 * 2^10`**)
`M`	**`1e+6`**	**`6.5M`**	6500000 (**`6.5 * 1e+6`**)
`Mi`	**`2^20`**	**`6.5Mi`**	6815744 (**`6.5 * 2^20`**)
`G`	**`1e+9`**	**`0.4G`**	400000000 (**`0.4 * 1e+9`**)
`Gi`	**`2^30`**	**`0.4Gi`**	429496729 (**`0.4 * 2^30`**)
`T`	**`1e+12`**	**`37T`**	37000000000000 (**`37 * 1e+12`**)

Suffix	Scale	Example	Equivalence
Ti	**2^40**	**37Ti**	40681930227712 (**37 * 2^40**)
P	**1e+15**	**9.8P**	9800000000000000 (**9.8 * 1e+15**)
Pi	**2^50**	**9.8Pi**	11033819087057716 (**9.8 * 2^50**)
E	**1e+18**	**0.42E**	420000000000000000 (**0.42 * 1e+18**)
Ei	**2^60**	**0.42Ei**	484227031934875712 (**0.42 * 2^60**)

Note: The suffix is case-sensitive; **m** and **M** differ.

6.3.2. Full Name

The last convention relates to the ***full name*** of a field. All field names are symbols. At the top-level, the full name of a field is identical to the field name. At each sub-level, the full name of a field is the full name of the parent structure followed by a "**.**" (period, **U+2E**) followed by the name of the field.

Here is a map with two top-level items, **one** and **two**:

```
one:
  a:
    x:   9
    y:   10
    z:   11
  b:
    q:   19
    r:   20
    s:   21
two:     42
```

The value of **one** is a sub-map, while the value of **two** is a simple number. The following table shows all the field names and full names.

Name	Full Name	Depth
one	**one**	0 (top-level)
two	**two**	0

Name	Full Name	Depth
a	**one.a**	1
b	**one.b**	1
x	**one.a.x**	2
y	**one.a.y**	2
z	**one.a.z**	2
q	**one.b.q**	2
r	**one.b.r**	2
s	**one.b.s**	2

6.4. COMMON STRUCTURES

All configuration files are maps at the top-level, with a few required fields and a series of optional ones. In most cases field values are basic data elements, but sometimes the value is a list or a sub-map. In a map, the order does not matter, although it is traditional to place the required fields first.

6.4.1. Top-Level

The top-level fields are **kind**, **apiVersion**, **metadata**, and **spec**.

kind (enum, one of: Service, Pod, ReplicationController)

This specifies what the configuration file is trying to configure. Although Kubernetes can usually infer **kind** from context, the slight redundancy of specifying it in the configuration file ensures that type errors are caught early.

apiVersion (enum)

This specifies which version of the API is used in the configuration file. In this document all examples use **apiVersion: v1**.

metadata (map)

This is a top-level field for Service, Pod and ReplicationController files and additionally found as a member of the ReplicationController's **`template`** map. Common sub-fields (all optional unless otherwise indicated) are:

Field	Type	Comment
`name`	symbol	Required
`namespace`	symbol	Default is **`default`**
`labels`	map	See individual types

Strictly speaking, **`metadata`** is optional. However, we recommend including it along with the others, anyway, because **`name`** and **`labels`** facilitate later manipulation of the Service, Pod or ReplicationController.

`spec` (map)

This field is the subject of the rest of this document.

6.4.2. Elsewhere

The other fields described in this section are common, in the sense of being found in more than one context, but not at top-level.

`labels` (map)

This is often one of the fields in the **`metadata`** map. Valid **label keys** have two segements:

```
[prefix/]name
```

The **`prefix`** and "**`/`**" (slash, **`U+2F`**) portions are optional. The **`name`** portion is required and must be 1-63 characters in length. It must begin and end with with an alphanumeric character (i.e., **`[0-9A-Za-z]`**). The internal characters of **`name`** may include hyphen, dot and underscore. Here are some label keys, valid and invalid:

Label Keys	Prefix	Name	Comments
`prefix/name`	**`prefix`**	**`name`**	
`just-a-name`	(n/a)	**`just-a-name`**	

Label Keys	Prefix	Name	Comments
-simply-wrong!	(n/a)	(n/a)	beg and end not alphanumeric
example.org/service	**example.org**	**service**	looks like a domain!

In the following example, **labels** and **name** comprise the map value of field **metadata** and the map value of **labels** has only one key/value pair.

```
metadata:
  labels:
    name: rabbitmq
  name: rabbitmq-controller
```

Note that in this example **metadata.labels.name** and **metadata.name** differ.

selector (map)

This is often one of the fields in the **spec** map of a Service or ReplicationController, but is also found at top-level. The map specifies field names and values that must match in order for the configured object to receive traffic. For example, the following fragment matches the **labels** example above.

```
spec:
  selector:
    name: rabbitmq
```

protocol (enum, one of: TCP, UDP)

This specifies an IP protocol.

port (integer)

The field value is the TCP/UDP port where the service, pod or replication controller can be contacted for administration and control purposes. Similar fields are **containerPort**, **hostPort** and **targetPort**. Often, **port** is found in the same map with **name** and **protocol**. For example, here is a fragment that shows a list of two such maps as the value for field **ports**:

```
ports:
- name: dns
  port: 53
  protocol: UDP
- name: dns-tcp
  port: 53
  protocol: TCP
```

In this example, the **port** in both maps is identical, while the **name** and **protocol** differ.

`limits` (map)

The field value is a sub-map associating resource types with resource-quantity values. For **`limits`** the quantities describe maximum allowable values. A similar field is **`request`**, which describes desired values.

Valid resource types are **`cpu`** and **`memory`**. The units for **`cpu`** are Kubernetes Compute Unit seconds/second (i.e., CPU cores normalized to a canonical "Kubernetes CPU"). The units for **`memory`** are bytes.

In the following fragment, **`cpu`** is limited to 0.1 KCU and **`memory`** to 2GiB.

```
resources:
  limits:
    cpu: 100m
    memory: 2Gi
```

As shown here, the **`limits`** field is often found as part of the map value for the **`resources`** field.

6.5. SPECIFIC STRUCTURES

The following subsections list fields found in the various configuration files apart from those in Common Structures. A field value's type is either one of the elemental data types, including those listed in Conventions, **map** or **list**. Each subsection also discusses pitfalls for that particular file.

6.5.1. Service

At the most basic level, Kubernetes can be configured with one Service YAML and one Pod YAML. In the service YAML, the required field **`kind`** has value **`Service`**. The **`spec`** tree should include **`ports`**, and optionally, **`selector`** and **`type`**. The value of **`type`** is an enum, one of: **`ClusterIP`** (the default if **`type`** is unspecified), **`NodePort`**, **`LoadBalancer`**.

Here is an example of a basic Service YAML:

```
kind: Service
apiVersion: v1
metadata:
  name: blog
spec:
  ports:
    - containerPort: 4567
      targetPort: 80
  selector:
    name: blog
  type: LoadBalancer
```

Note that **`name: blog`** is indented by two columns to signify it being part of the sub-map value of both **`metadata`** and **`selector`** trees.

Warning

Omitting the indentation of **`metadata.name`** places **`name`** at top-level and gives **`metadata`** a **`nil`** value.

Each container's port 4567 is visible externally as port 80, and they are accessed in a round-robin manner because of `type: LoadBalancer'.

6.5.2. Pod

In the pod YAML, the required field **`kind`** has value **`Pod`**. The **`spec`** tree should include **`containers`** and optionally **`volumes`** fields. Their values are both a list of maps. Each element of **`containers`** specifies an **`image`**, with a **`name`** and other fields that describe how the image is to be run (e.g., **`privileged`**, **`resources`**), what ports it exposes, and what volume mounts it requires. Each element of **`volumes`** specfies a **`hostPath`**, with a **`name`**.

```
apiVersion: v1
kind: Pod
metadata:
  name: host-test
spec:
  containers:
    - image: nginx
      name: host-test
      privileged: false
      volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: srv
          readOnly: false
  volumes:
    - hostPath:
        path: /srv/my-data
      name: srv
```

This example specifies the webserver **`nginx`** to be run unprivileged and with access to the host directory **`/srv/my-data`** visible internally as **`/usr/share/nginx/html`**.

6.5.3. Replication Controller

In the replication controller YAML, the required field **`kind`** has value **`ReplicationController`**. The **`spec.replicas`** field specifies how the pod should be ***horizontally scaled***, that is, how many copies of a pod should be active simultaneously. The **`spec`** tree also has a **`template`** tree, which in turn has a sub-**`spec`** tree that resembles the **`spec`** tree from a **`Pod`** YAML.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 3                1
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:                    2
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

1

Kubernetes will try to maintain three active copies.

2

This sub-**spec** tree is essentially a Pod **spec** tree.

6.6. FIELD REFERENCE

The following table lists all fields found the files, apart from those in Common Structures. A field value's type is either one of the elemental data types (including those listed in Conventions), map, or list. For the **Context** column, the code is **s** for services, **p** for pods, **r** for replication controllers.

Field	Type	Context	Example / Comment
desiredState	map		
clusterIP	**opt-v4addr**	s	**10.254.100.50**
selector	map	s	one element, key **name**
replicas	**integer**	r	**2**

Field	Type	Context	Example / Comment
replicaSelector	map	r	one field: **selectorname**
podTemplate	map	r	two fields: **desiredState**, **labels**
manifest	map	r	
version	**string**	r	like **apiVersion**
containers	map	pr	
image	string	pr	
selectorname	string	r	
deprecatedPublicIPs	list	s	each element is an v4addr
privileged	boolean	pr	
resources	map	pr	
imagePullPolicy	enum	pr	**Always**, **Never**, **IfNotPresent**
command	list of strings	pr	for **docker run**