In this scenario you'll learn how to bootstrap a Kubernetes cluster using Kubeadm.

Kubeadm solves the problem of handling TLS encryption configuration, deploying the core Kubernetes components and ensuring that additional nodes can easily join the cluster. The resulting cluster is secured out of the box via mechanisms such as RBAC.

"bootstrap" refers to the process of starting up a system or software application by loading and executing the minimal set of components necessary to get the system or application up and running.

Kubeadm is a tool provided by Kubernetes to simplify the process of creating and managing a Kubernetes cluster. It is used to bootstrap a new Kubernetes cluster by initializing the control plane, generating the necessary certificates, and configuring the cluster networking. Kubeadm automates many of the manual tasks involved in setting up a cluster, and provides a standardized way of creating a new cluster that is both reliable and repeatable.

Kubeadm is used primarily for setting up new clusters, but it can also be used to add or remove nodes from an existing cluster. It is a command-line tool that provides a simple and straightforward way to configure a Kubernetes cluster without having to manually edit configuration files or write custom scripts.

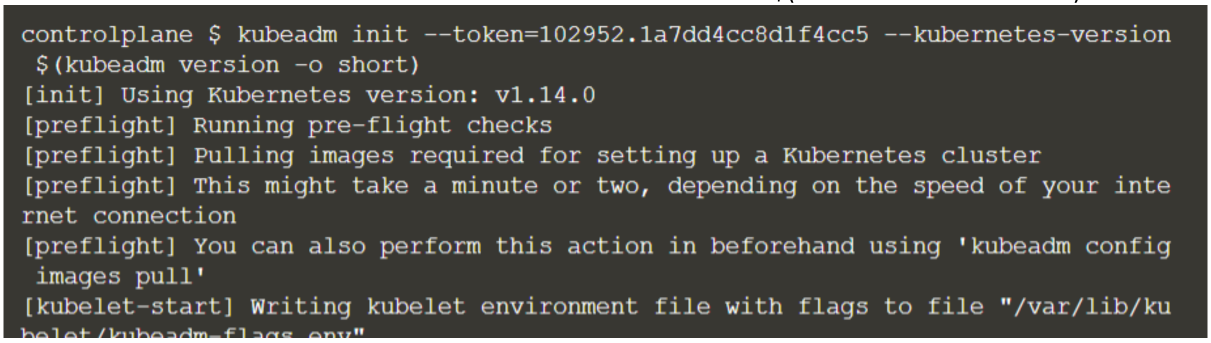
Kubeadm is a core component of Kubernetes and is included in the official Kubernetes distribution. It is designed to be used in conjunction with other Kubernetes components like Kubelet and kubectl, and is often used in combination with other tools like Helm and Terraform to automate the deployment and management of Kubernetes clusters.

Step 1 - Initialise Master
Kubeadm has been installed on the nodes. Packages are available for Ubuntu 16.04+, CentOS 7 or HypriotOS v1.0.1+.

The first stage of initialising the cluster is to launch the master node. The master is responsible for running the control plane components, etcd and the API server. Clients will communicate to the API to schedule workloads and manage the state of the cluster.

Task
The command below will initialise the cluster with a known token to simplify the following steps

kubeadm init --token=102952.1a7dd4cc8d1f4cc5 --kubernetes-version $(kubeadm version -o short)



```
controlplane $ kubeadm init --token=102952.1a7dd4cc8d1f4cc5 --kubernetes-version
 $(kubeadm version -o short)
[init] Using Kubernetes version: v1.14.0
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your inte
rnet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config
 images pull'
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/ku
belet/kubeadm-flags.env"
```

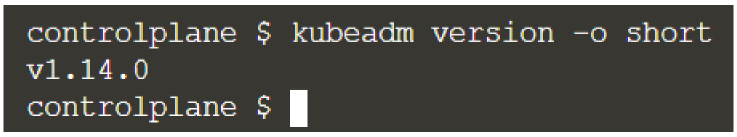
In production, it's recommend to exclude the token causing kubeadm to generate one on your behalf.

Yes, it is recommended to exclude the token causing kubeadm to generate one on your behalf for security reasons. By default, when you run kubeadm init to initialize a Kubernetes cluster, kubeadm generates a bootstrap token that is used by nodes to authenticate with the control plane. However, this token is highly sensitive and should not be shared or exposed.

To exclude the token causing kubeadm to generate one on your behalf, you can use the --token and --certificate-key flags to manually generate a token and certificate key for your cluster. This allows you to control the security of your cluster and limit access to authorized parties.

For example, you can run the following command to manually generate a bootstrap token and certificate key:
kubeadm init --token=<token> --certificate-key=<certificate-key>

Replace <token> and <certificate-key> with your own values. It is recommended to use a strong, random value for the token and certificate key. Additionally, make sure to securely store these values in a safe location and not share them with unauthorized parties.



```
controlplane $ kubeadm version -o short
v1.14.0
controlplane $ 
```

The command kubeadm init --token=102952.1a7dd4cc8d1f4cc5 --kubernetes-version $(kubeadm version -o short) initializes a new Kubernetes control plane using kubeadm, with a specified token and the same version of Kubernetes that is currently installed on the machine.

Here's a breakdown of the flags used in this command:

--token=102952.1a7dd4cc8d1f4cc5: This flag specifies the token that will be used by the joining nodes to authenticate with the control plane. In this case, the token is 102952.1a7dd4cc8d1f4cc5.

--kubernetes-version $(kubeadm version -o short): This flag specifies the version of Kubernetes that will be used to initialize the control plane. The $(kubeadm version -o short) command dynamically retrieves the version of kubeadm that is currently installed on the machine and passes it to the --kubernetes-version flag.

Note that this command will initialize a new Kubernetes control plane on the machine and generate new certificates and keys for the control plane components. Additionally, it will create a kubeconfig file that you can use to access the Kubernetes API server.

The Kubernetes control plane is a collection of components that together make up the "brain" of a Kubernetes cluster. It is responsible for managing the overall state of the cluster, as well as for making decisions about scheduling, scaling, and other important operational tasks.

The control plane components include:

kube-apiserver: This component provides a RESTful API that serves as the primary interface for managing the Kubernetes cluster. It validates and processes API requests, and stores the state of the cluster in etcd.

etcd: This is a distributed key-value store that is used to store the configuration and state of the Kubernetes cluster.

kube-scheduler: This component is responsible for scheduling new workloads on the nodes in the cluster. It makes decisions based on resource availability, workload requirements, and other factors.

kube-controller-manager: This component is responsible for monitoring the state of the cluster and making adjustments as needed. It includes a number of different controllers that are responsible for managing things like nodes, pods, services, and replication.

cloud-controller-manager: This optional component includes a set of controllers that interact with cloud provider APIs to manage cloud-specific resources like load balancers, persistent volumes, and security groups.

These components work together to provide a highly available and fault-tolerant control plane that can scale to support large, complex Kubernetes clusters.

To manage the Kubernetes cluster, the client configuration and certificates are required. This configuration is created when kubeadm initialises the cluster. The command copies the configuration to the users home directory and sets the environment variable for use with the CLI.

perform the following actions:

The first command sudo cp /etc/kubernetes/admin.conf $HOME/ makes a copy of the Kubernetes admin.conf file located at /etc/kubernetes/admin.conf and places it in the user's home directory ($HOME).

The second command sudo chown $(id -u):$(id -g) $HOME/admin.conf changes the ownership of the copied admin.conf file to the current user who executed the command, making the file owned by the user.

The third command export KUBECONFIG=$HOME/admin.conf sets the KUBECONFIG environment variable to the path of the copied admin.conf file. This allows you to use kubectl commands without specifying the path to the config file each time.

By executing these commands, you are creating a copy of the Kubernetes admin.conf file in your home directory and setting the KUBECONFIG environment variable to point to this file. This allows you to access and manage your Kubernetes cluster using the kubectl command-line tool.

The command sudo chown $(id -u):$(id -g) $HOME/admin.conf changes the ownership of the admin.conf file located in the user's home directory to the current user who executed the command.

Here's a breakdown of the command:

sudo: This command runs the following command with superuser privileges.

chown: This command changes the ownership of a file or directory.

$(id -u) and $(id -g): These are command substitutions that return the current user ID and group ID of the user who executed the command, respectively. These values are used to set the ownership of the file.

$HOME/admin.conf: This is the path to the admin.conf file located in the user's home directory.

By changing the ownership of the admin.conf file to the current user, you ensure that the user has the necessary permissions to access and modify the file as needed. This is important for managing your Kubernetes cluster using the kubectl command-line tool, which relies on this file to authenticate and communicate with the Kubernetes API server.

Step 2 - Deploy Container Networking Interface (CNI)
The Container Network Interface (CNI) defines how the different nodes and their workloads should communicate. There are multiple network providers available, some are listed here.

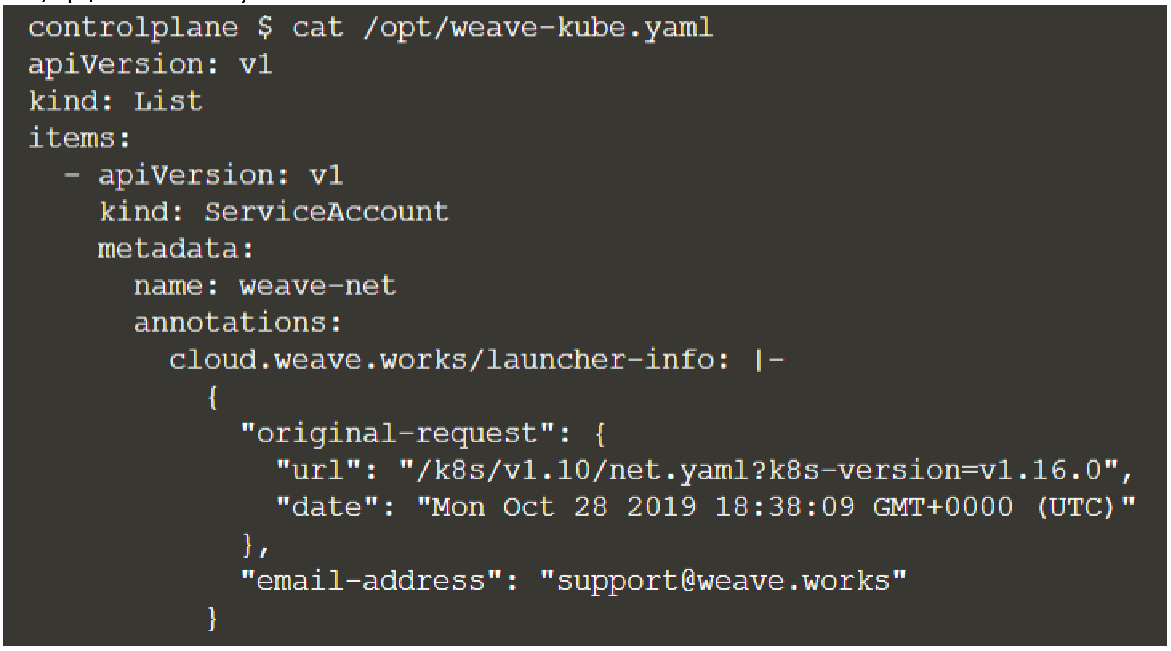
The Container Networking Interface (CNI) is a specification that defines a standard interface between container runtimes and network plugins. It allows container runtimes like Docker and Kubernetes to use different network plugins to create and manage network connections for containers.

CNI plugins are responsible for configuring network interfaces, setting up virtual networks, and managing IP addresses for containers. They provide a wide range of networking options, such as overlay networks, virtual private networks (VPNs), and software-defined networks (SDNs), among others.

CNI plugins can be implemented as stand-alone binaries, or as container images that are deployed as sidecar containers alongside the main container. They can be installed and configured independently of the container runtime, allowing for greater flexibility and modularity in the network stack.

Kubernetes uses CNI as its default networking model, and provides support for a number of CNI plugins, including Calico, Flannel, Weave Net, and others. This allows Kubernetes users to choose the network plugin that best meets their needs, and to configure and manage their network infrastructure in a way that is consistent with their existing toolchains and workflows.

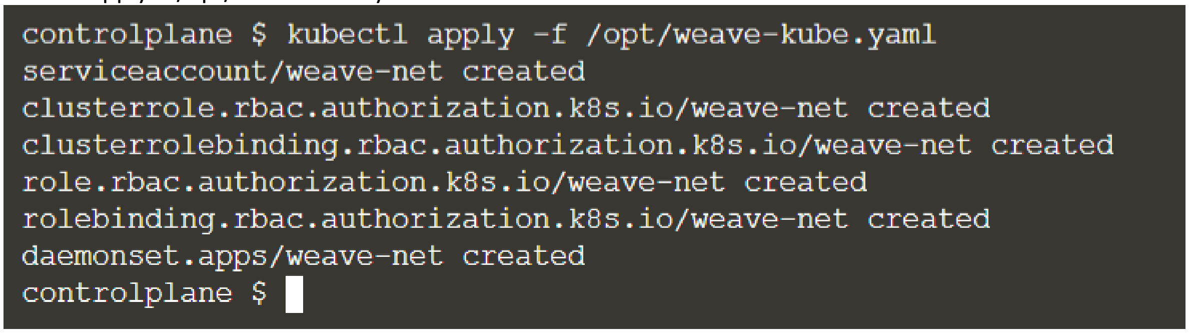
Task
In this scenario we'll use WeaveWorks. The deployment definition can be viewed at
cat /opt/weave-kube.yaml



```
controlplane $ cat /opt/weave-kube.yaml
apiVersion: v1
kind: List
items:
  - apiVersion: v1
    kind: ServiceAccount
    metadata:
      name: weave-net
      annotations:
        cloud.weave.works/launcher-info: |-
          {
            "original-request": {
              "url": "/k8s/v1.10/net.yaml?k8s-version=v1.16.0",
              "date": "Mon Oct 28 2019 18:38:09 GMT+0000 (UTC)"
            },
            "email-address": "support@weave.works"
          }
```

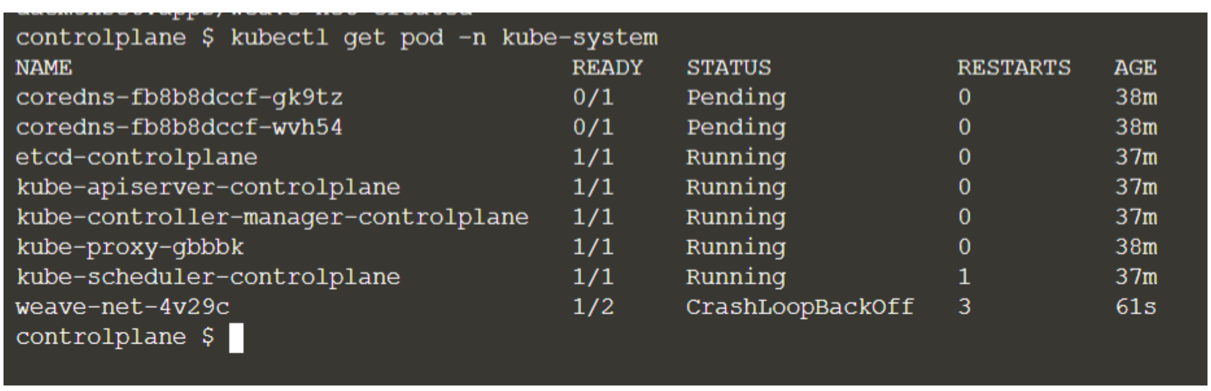
This can be deployed using kubectl apply.

kubectl apply -f /opt/weave-kube.yaml



```
controlplane $ kubectl apply -f /opt/weave-kube.yaml
serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created
controlplane $
```

Weave will now deploy as a series of Pods on the cluster.
The status of this can be viewed using the command
kubectl get pod -n kube-system

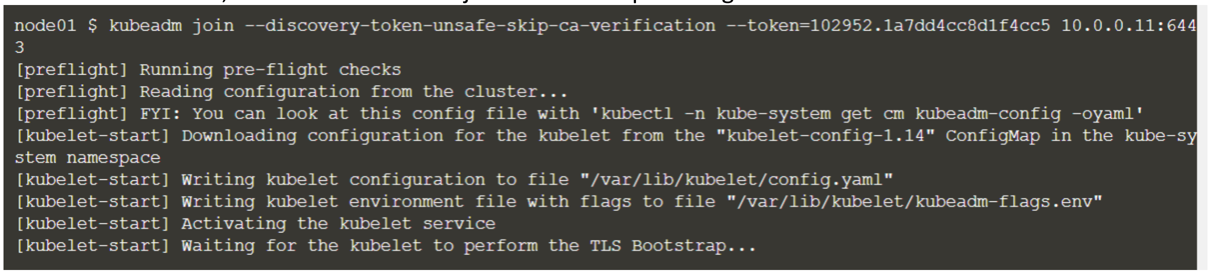


```
controlplane $ kubectl get pod -n kube-system
NAME                                   READY   STATUS             RESTARTS   AGE
coredns-fb8b8dccf-gk9tz                0/1     Pending            0          38m
coredns-fb8b8dccf-wvh54                0/1     Pending            0          38m
etcd-controlplane                      1/1     Running            0          37m
kube-apiserver-controlplane            1/1     Running            0          37m
kube-controller-manager-controlplane   1/1     Running            0          37m
kube-proxy-gbbbk                       1/1     Running            0          38m
kube-scheduler-controlplane            1/1     Running            1          37m
weave-net-4v29c                        1/2     CrashLoopBackOff   3          61s
controlplane $
```

Step 3 - Join Cluster

Once the Master and CNI has initialised, additional nodes can join the cluster as long as they have the correct token. The tokens can be managed via kubeadm token, for example kubeadm token list

Task
On the second node, run the command to join the cluster providing the IP address of the Master node.



```
node01 $ kubeadm join --discovery-token-unsafe-skip-ca-verification --token=102952.1a7dd4cc8d1f4cc5 10.0.0.11:644
3
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
[kubelet-start] Downloading configuration for the kubelet from the "kubelet-config-1.14" ConfigMap in the kube-sy
stem namespace
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Activating the kubelet service
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...
```

kubeadm join --discovery-token-unsafe-skip-ca-verification --token=102952.1a7dd4cc8d1f4cc5 10.0.0.11:6443

The kubeadm join command is used to add a new worker node to an existing Kubernetes cluster. Here is what each argument in the command you provided does:

--discovery-token-unsafe-skip-ca-verification: This flag skips verification of the TLS certificate presented by the API server during node registration. This is not recommended for production use, but can be used in cases where the CA certificate is not yet available or in other testing scenarios.

--token=102952.1a7dd4cc8d1f4cc5: This specifies the token that the worker node should use to authenticate with the cluster during the registration process.

10.0.0.11:6443: This is the IP address and port number of the Kubernetes control plane that the worker node should connect to for registration.

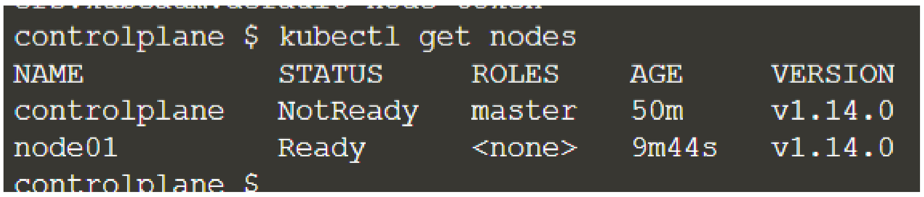
When executed, this command will initiate the node registration process, and the worker node will join the existing Kubernetes cluster. Note that you should replace the IP address and token in the command with the values appropriate for your specific Kubernetes cluster.

This is the same command provided after the Master has been initialised.

The --discovery-token-unsafe-skip-ca-verification tag is used to bypass the Discovery Token verification. As this token is generated dynamically, we couldn't include it within the steps. When in production, use the token provided by kubeadm init.

Step 4 - View Nodes
The cluster has now been initialised. The Master node will manage the cluster, while our one worker node will run our container workloads

Task
The Kubernetes CLI, known as kubectl, can now use the configuration to access the cluster. For example, the command below will return the two nodes in our cluster.

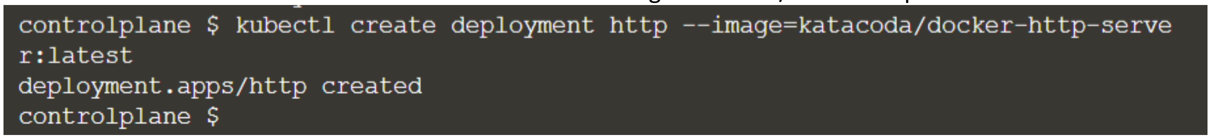


```
controlplane $ kubectl get nodes
NAME           STATUS     ROLES    AGE     VERSION
controlplane   NotReady   master   50m     v1.14.0
node01         Ready      <none>   9m44s   v1.14.0
controlplane $
```

Step 5 - Deploy Pod
The state of the two nodes in the cluster should now be Ready. This means that our deployments can be scheduled and launched.

Using Kubectl, it's possible to deploy pods. Commands are always issued for the Master with each node only responsible for executing the workloads.

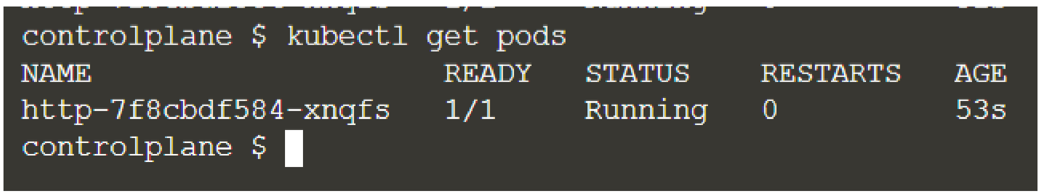
The command below create a Pod based on the Docker Image katacoda/docker-http-server.



```
controlplane $ kubectl create deployment http --image=katacoda/docker-http-serve
r:latest
deployment.apps/http created
controlplane $
```

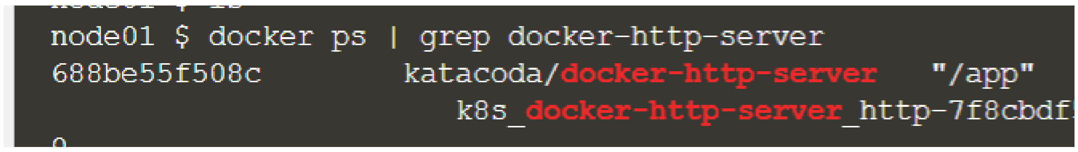
The status of the Pod creation can be viewed using

kubectl get pods



```
controlplane $ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
http-7f8cbdf584-xnqfs   1/1     Running   0          53s
controlplane $
```

Once running, you can see the Docker Container running on the node.



```
node01 $ docker ps | grep docker-http-server
688be55f508c        katacoda/docker-http-server    "/app"
                       k8s_docker-http-server_http-7f8cbdf
0
```

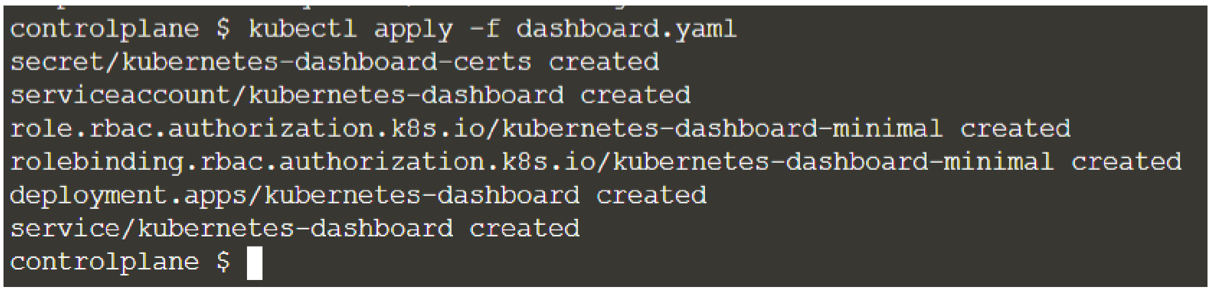
Step 6 - Deploy Dashboard

Kubernetes has a web-based dashboard UI giving visibility into the Kubernetes cluster.

Task

Deploy the dashboard yaml with the command

kubectl apply -f dashboard.yaml



```
controlplane $ kubectl apply -f dashboard.yaml
secret/kubernetes-dashboard-certs created
serviceaccount/kubernetes-dashboard created
role.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
deployment.apps/kubernetes-dashboard created
service/kubernetes-dashboard created
controlplane $
```

controlplane $ cat dashboard.yaml

# Copyright 2017 The Kubernetes Authors.

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

```
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# ------------------- Dashboard Secret ------------------- #

apiVersion: v1
kind: Secret
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard-certs
  namespace: kube-system
type: Opaque

---
# ------------------- Dashboard Service Account ------------------- #

apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system

---
# ------------------- Dashboard Role & Role Binding ------------------- #
```

```
kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

  name: kubernetes-dashboard-minimal

  namespace: kube-system

rules:

  # Allow Dashboard to create 'kubernetes-dashboard-key-holder' secret.

- apiGroups: [""]

  resources: ["secrets"]

  verbs: ["create"]

  # Allow Dashboard to create 'kubernetes-dashboard-settings' config map.

- apiGroups: [""]

  resources: ["configmaps"]

  verbs: ["create"]

  # Allow Dashboard to get, update and delete Dashboard exclusive secrets.

- apiGroups: [""]

  resources: ["secrets"]

  resourceNames: ["kubernetes-dashboard-key-holder", "kubernetes-dashboard-certs"]

  verbs: ["get", "update", "delete"]

  # Allow Dashboard to get and update 'kubernetes-dashboard-settings' config map.

- apiGroups: [""]

  resources: ["configmaps"]

  resourceNames: ["kubernetes-dashboard-settings"]

  verbs: ["get", "update"]

  # Allow Dashboard to get metrics from heapster.

- apiGroups: [""]

  resources: ["services"]

  resourceNames: ["heapster"]

  verbs: ["proxy"]

- apiGroups: [""]

  resources: ["services/proxy"]

  resourceNames: ["heapster", "http:heapster:", "https:heapster:"]
  verbs: ["get"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kubernetes-dashboard-minimal
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: kubernetes-dashboard-minimal
subjects:
- kind: ServiceAccount
  name: kubernetes-dashboard
  namespace: kube-system

---
# ------------------- Dashboard Deployment ------------------- #

kind: Deployment
apiVersion: apps/v1beta2
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
spec:
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      k8s-app: kubernetes-dashboard
```

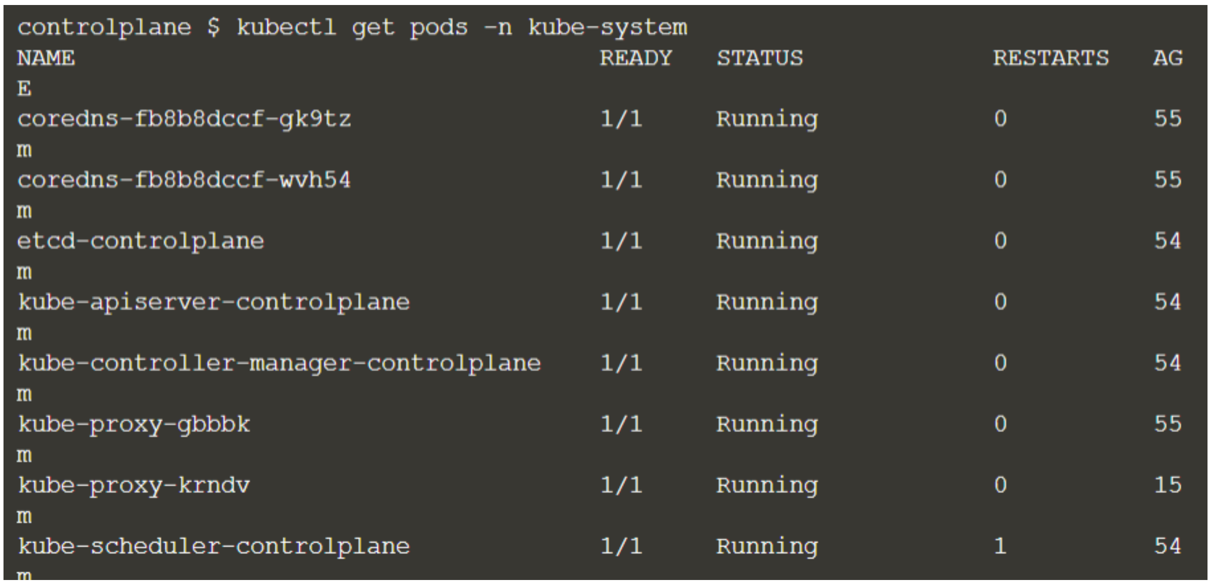
```
template:
  metadata:
    labels:
      k8s-app: kubernetes-dashboard
  spec:
    containers:
    - name: kubernetes-dashboard
      image: k8s.gcr.io/kubernetes-dashboard-amd64:v1.10.0
      ports:
      - containerPort: 8443
        protocol: TCP
      args:
        - --auto-generate-certificates
        # Uncomment the following line to manually specify Kubernetes API server Host
        # If not specified, Dashboard will attempt to auto discover the API server and connect
        # to it. Uncomment only if the default does not work.
        # - --apiserver-host=http://my-address:port
      volumeMounts:
      - name: kubernetes-dashboard-certs
        mountPath: /certs
        # Create on-disk volume to store exec logs
      - mountPath: /tmp
        name: tmp-volume
      livenessProbe:
        httpGet:
          scheme: HTTPS
          path: /
          port: 8443
        initialDelaySeconds: 30
        timeoutSeconds: 30
    volumes:
    - name: kubernetes-dashboard-certs
      secret:
          secretName: kubernetes-dashboard-certs
      - name: tmp-volume
        emptyDir: {}
      serviceAccountName: kubernetes-dashboard
      # Comment the following tolerations if Dashboard must not be deployed on master
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule

---
# ------------------- Dashboard Service ------------------- #

kind: Service
apiVersion: v1
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
spec:
  externalIPs:
  - 10.0.0.12
  ports:
    - port: 8443
      targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboardcontrolplane $
```

The dashboard is deployed into the kube-system namespace. View the status of the deployment with

```
kubectl get pods -n kube-system
```



```
controlplane $ kubectl get pods -n kube-system
NAME                                    READY   STATUS             RESTARTS   AG
E
coredns-fb8b8dccf-gk9tz                 1/1     Running            0          55
m
coredns-fb8b8dccf-wvh54                 1/1     Running            0          55
m
etcd-controlplane                       1/1     Running            0          54
m
kube-apiserver-controlplane             1/1     Running            0          54
m
kube-controller-manager-controlplane    1/1     Running            0          54
m
kube-proxy-gbbbk                        1/1     Running            0          55
m
kube-proxy-krndv                        1/1     Running            0          15
m
kube-scheduler-controlplane             1/1     Running            1          54
m
```