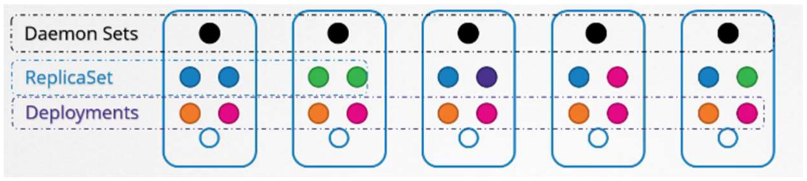
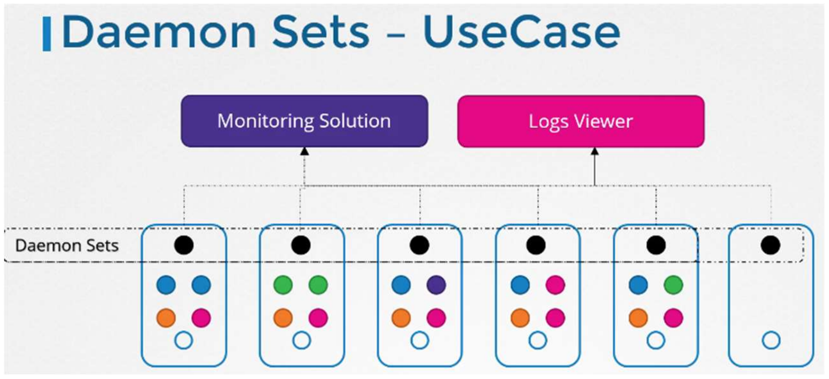
DaemonSets

In this lecture, we look at DaemonSets in Kubernetes. So far we have deployed various pods on different nodes in our cluster. With the help of ReplicaSets and deployments, we made sure multiple copies of our applications are made available across various different worker nodes. DaemonSets are like ReplicaSets, as in it helps you deploy multiple instances of pods. But it runs one copy of your pod on each node in your cluster. Whenever a new node is added to the cluster, a replica of the pod is automatically added to that node. And when a node is removed, the pod is automatically removed. The DaemonSet ensures that one copy of the pod is always present on all nodes in the cluster.



So what are some use cases of DaemonSets? Say you would like to deploy a monitoring agent or log collector on each of your nodes in the cluster, so you can monitor your cluster better. A DaemonSet is perfect for that as it can deploy your monitoring agent in the form of a pod on all the nodes in your cluster. Then you don't have to worry about adding or removing monitoring agents from these nodes when there are changes in your cluster, as the DaemonSet will take care of that for you.

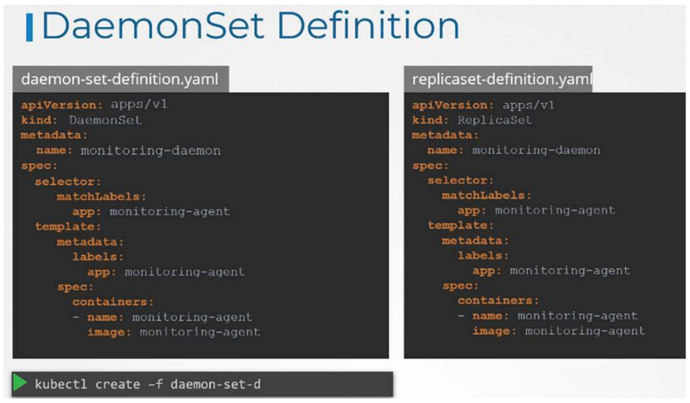


Earlier, while discussing the Kubernetes architecture, we learned that one of the worker node components required on every node in the cluster is a kube-proxy. That is one good use case of DaemonSets. The kube-proxy component can be deployed as a DaemonSet in the cluster. Another use case is for networking. Networking solutions like Vivenet require an agent to be deployed on each node in the cluster. We will discuss networking concepts in much more detail later during this course, but I just wanted to point it out here for now.

Creating a DaemonSet is similar to the ReplicaSet creation process. It has a nested pod specification under the template section and selectors to link the DaemonSet to the pods. A DaemonSet definition file has a similar structure. We start with the API version, kind, metadata, and spec.

The API version is "apps" we want, kind is "DaemonSet" instead of "ReplicaSet." We will set the name to "Monitoring Daemon." Under spec, you have a selector and a pod specification template. It's almost exactly like the ReplicaSet definition, except that the kind is a DaemonSet. Ensure the labels in the selector match the ones in the pod template.

Once ready, create the DaemonSet using the kubectl Create DaemonSet command.



DaemonSet Definition

daemon-set-definition.yaml

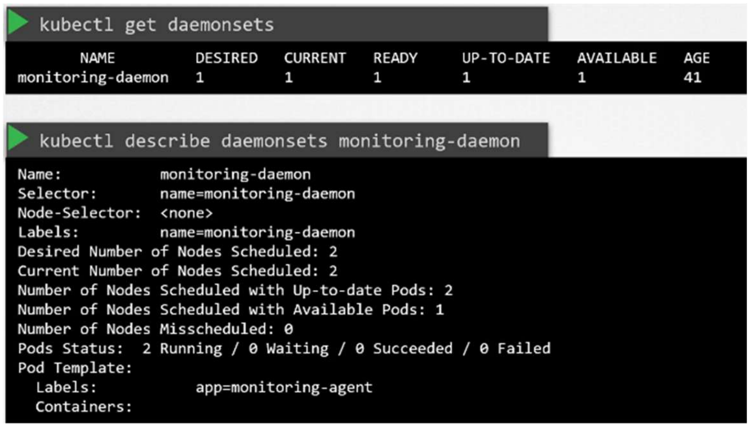
```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon
spec:
  selector:
     matchLabels:
       app: monitoring-agent
  template:
     metadata:
       labels:
         app: monitoring-agent
     spec:
       containers:
       - name: monitoring-agent
         image: monitoring-agent
```

replicaset-definition.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: monitoring-daemon
spec:
  selector:
     matchLabels:
       app: monitoring-agent
  template:
     metadata:
       labels:
         app: monitoring-agent
     spec:
       containers:
       - name: monitoring-agent
         image: monitoring-agent
```

```
kubectl create -f daemon-set-d
```

To view the create DaemonSet run the kube control, Get DaemonSet command. And of course, to view more details, run the kube control, Describe DaemonSet command.



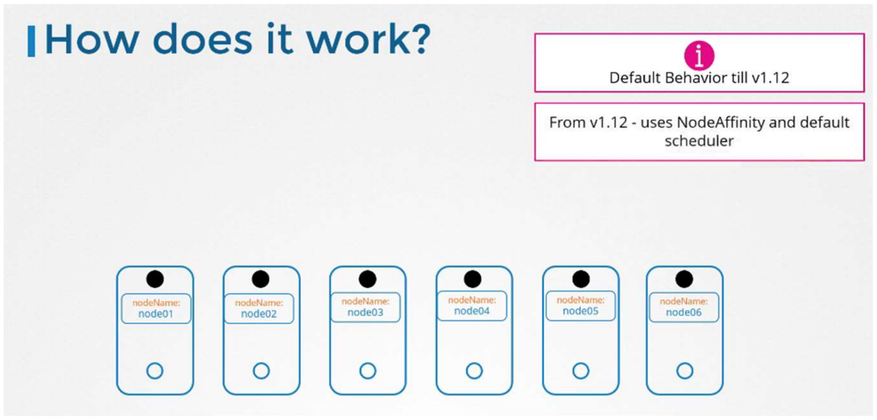
```
kubectl get daemonsets
      NAME          DESIRED   CURRENT   READY    UP-TO-DATE   AVAILABLE   AGE
monitoring-daemon   1         1         1        1            1           41
```

```
kubectl describe daemonsets monitoring-daemon
Name:           monitoring-daemon
Selector:       name=monitoring-daemon
Node-Selector:  <none>
Labels:         name=monitoring-daemon
Desired Number of Nodes Scheduled: 2
Current Number of Nodes Scheduled: 2
Number of Nodes Scheduled with Up-to-date Pods: 2
Number of Nodes Scheduled with Available Pods: 1
Number of Nodes Misscheduled: 0
Pods Status:  2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:           app=monitoring-agent
  Containers:
```

So how does a DaemonSet work? How does it schedule pods on each node and how does it ensure that every node has a pod? If you were asked to schedule a pod on each node in the cluster, how would you do it?

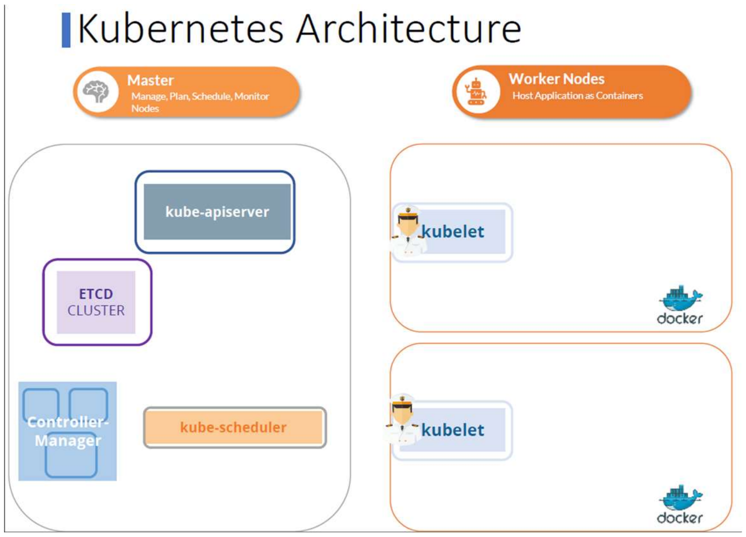
In one of the previous lectures in this section, we discussed that we could set the node name property on the pod to bypass the scheduler and get the pod placed on a node directly. So that's one approach. On each pod, set the node name property in its specification before it is created, and when they are created, they automatically land on the respective nodes.

So that's how it used to be until Kubernetes version 1.12. From version 1.12 onwards, the DaemonSet uses the default scheduler and node affinity rules that we learned in one of the previous lectures to schedule pods on nodes.



Static Pod

In this lecture, we discuss static pods in Kubernetes. Earlier in this course, we talked about the architecture and how the kubelet functions as one of the many control plane components in Kubernetes. The kubelet relies on the kube API server for instructions on what pods to load on its node, which are based on decisions made by the kube scheduler and stored in the etcd data store.



But what if there were no kube API server, kube scheduler, controllers, or etcd cluster? What if there was no master at all? What if there were no other nodes? What if you're all alone at sea by yourself, not part of any cluster? Is there anything that the kubelet can do as the captain of the ship? Can it operate as an independent node? If so, who would provide the instructions required to create those pods?

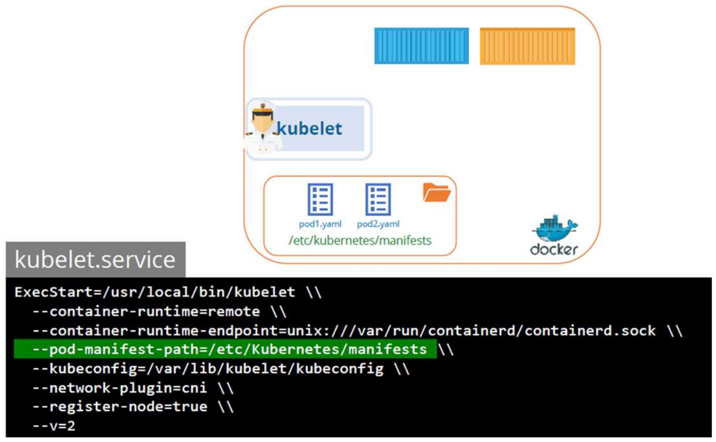
Well, the kubelet can manage a node independently. On the ship host, we have the kubelet installed, and of course, we have Docker as well to run containers. There is no Kubernetes cluster, so there are no kube API server or anything like that. The one thing that the kubelet knows how to do is create pods. But we don't have an API server here to provide pod details. As we've learned, to create a pod, you need the details of the pod in a pod definition file. But how do you provide the pod definition file to the kubelet without a kube API server? You can configure the kubelet to read the pod definition files from a directory on the server designated to store information about pods. Simply place the pod definition files in this directory. The kubelet periodically checks this directory for files, reads these files, and creates pods on the host.

Not only does it create the pod, but it can also ensure that the pod stays alive. If the application crashes, the kubelet attempts to restart it. If you make a change to any of the files within this directory, the kubelet recreates the pod for those changes to take effect. If you remove a file from this directory, the pod is deleted automatically. These pods that are created by the kubelet on its own, without the intervention from the API server or the rest of the Kubernetes cluster components, are known as **static pods**.

Remember, you can only create pods this way. You cannot create replica sets, deployments, or services by placing a definition file in the designated directory. These are all concepts that are part of the entire Kubernetes architecture, which requires other control plane components like replication and deployment controllers, and so on. The kubelet operates at the pod level and can only understand pods, which is why it's able to create static pods in this manner.

So what is that designated folder, and how do you configure it? It could be any directory on the host, and the location of that directory is passed to the kubelet as an option when running the service. The option is named "pod manifest path," and here it is set to /etc/kubernetes/manifests folder.

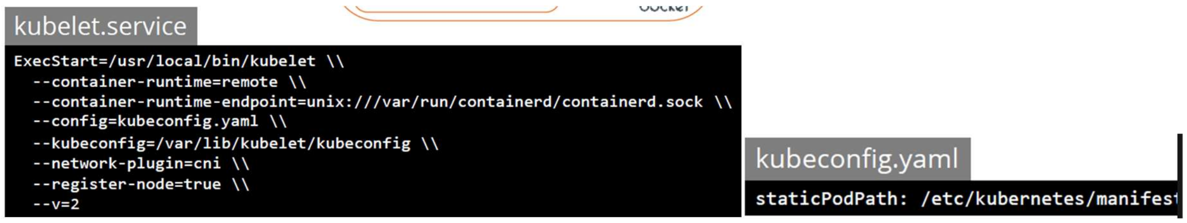
Static PODs



```
ExecStart=/usr/local/bin/kubelet \\
  --container-runtime=remote \\
  --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
  --pod-manifest-path=/etc/Kubernetes/manifests \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --register-node=true \\
  --v=2
```

There's also another way to configure this. Instead of specifying the option directly in the kubelet.service file, you could provide a path to another config file using the **--config** option and define the directory path as the static pod path in that file. Clusters set up by the kube admin tool use this approach.

If you're inspecting an existing cluster, you should inspect this option of the kubelet to identify the path to the directory. You will then know where to place the definition file for your static pods. So keep this in mind when you go through the labs. You should know how to view and configure this option, irrespective of the method used to set up the cluster. First, check the option pod manifest path in the kubelet service file. If it's not there, then look for the config option and identify the file used as the config file, and then within the config file, look for the static pod path option. Either of these should give you the right path.



kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  --container-runtime=remote \\
  --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
  --config=kubeconfig.yaml \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --register-node=true \\
  --v=2
```

kubeconfig.yaml

```
staticPodPath: /etc/kubernetes/manifest
```

Once the static pods are created, you can view them by running the docker ps command. So why not use the kubectl command as we have been doing so far? Remember, we don't have the rest of the Kubernetes cluster yet. So the kubectl utility works with the kube API server. Since we don't have an API server now, we can't use the kubectl command, which is why we are using the docker command.

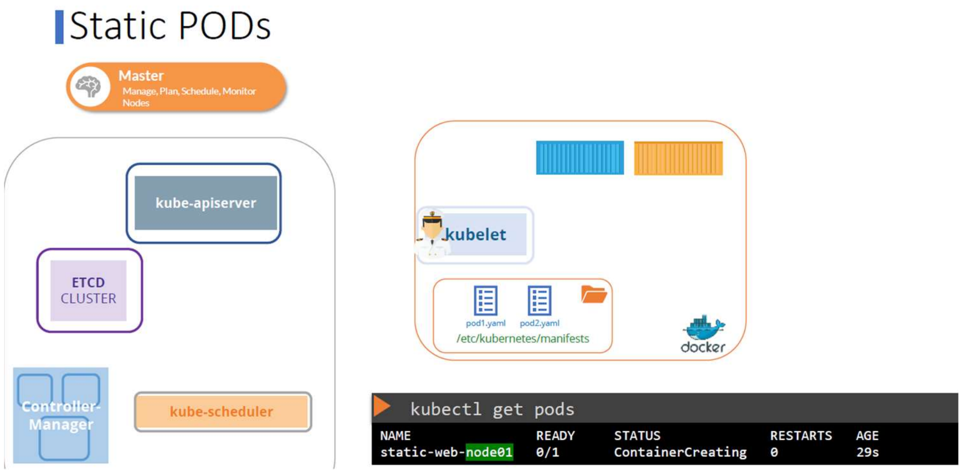
So then how does it work when the node is part of a cluster, when there is an API server requesting the kubelet to create pods? Can the kubelet create both kinds of pods at the same time?

Well, the way the kubelet works is it can take in requests for creating pods from different inputs. **The first is through the pod definition files from the static pods folder, as we just saw**. **The second is through an HTTP API endpoint. And that is how the kube API server provides input to kubelet**. The kubelet can create both kinds of pods, the static pods and the ones from the API server at the same time.

Well, in that case, is the API server aware of the static pods created by the kubelet?

Yes, it is. If you run the kubectl get pods command on the master node, the static pods will be listed as any other pod.

Well, how is that happening? When the kubelet creates a static pod, if it is a part of a cluster, it also creates a mirror object in the kube API server. What you see from the kube API server is just a read-only mirror of the pod. You can view details about the pod but you cannot edit or delete it like the usual pods. You can only delete them by modifying the files from the node's manifest folder. Note that the name of the pod is automatically appended with the node name, in this case, node 01.



```
kubectl get pods
NAME               READY   STATUS              RESTARTS   AGE
static-web-node01  0/1     ContainerCreating   0          29s
```

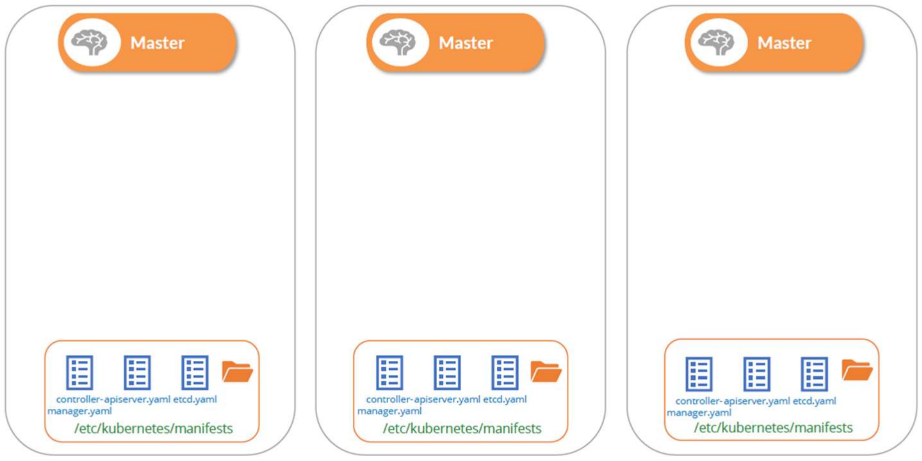
So then why would you want to use static pods? Since static pods are not dependent on the Kubernetes control plane, you can use static pods to deploy the control plane components itself as pods on a node.

Well, start by installing kubelet on all the master nodes then create pod definition files that use Docker images of the various control plane components such as the API server controller, etc. Place the definition files in the designated manifest folder, and the kubelet takes care of deploying the control plane components themselves as pods on the cluster.

This way, you don't have to download the binaries, configure services, or worry about the services crashing. If any of these services were to crash, since it's a static pod, it will automatically be restarted by the kubelet as needed.

That's how the kube admin tool sets up a Kubernetes cluster. Which is why when you list the pods in the kube-system namespace, you see the control plane components as pods in a cluster setup by the kube admin tool.

Use Case


Master
controller-manager.yaml
apiserver.yaml
etcd.yaml
/etc/kubernetes/manifests
Master
controller-manager.yaml
apiserver.yaml
etcd.yaml
/etc/kubernetes/manifests
Master
controller-manager.yaml
apiserver.yaml
etcd.yaml
/etc/kubernetes/manifests