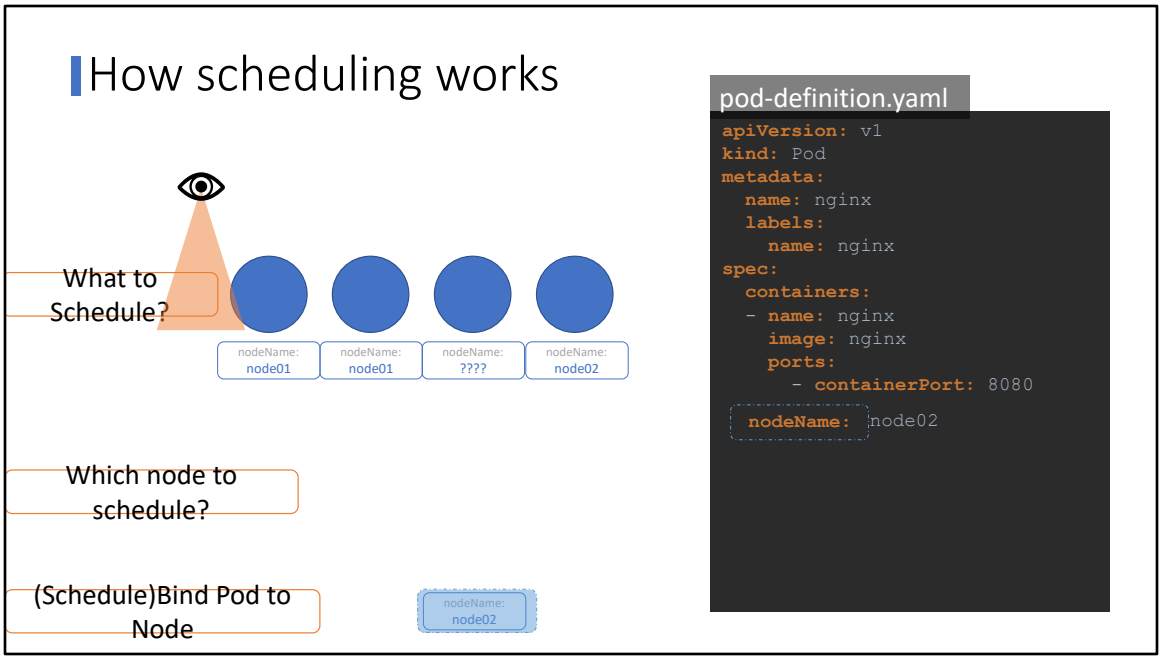
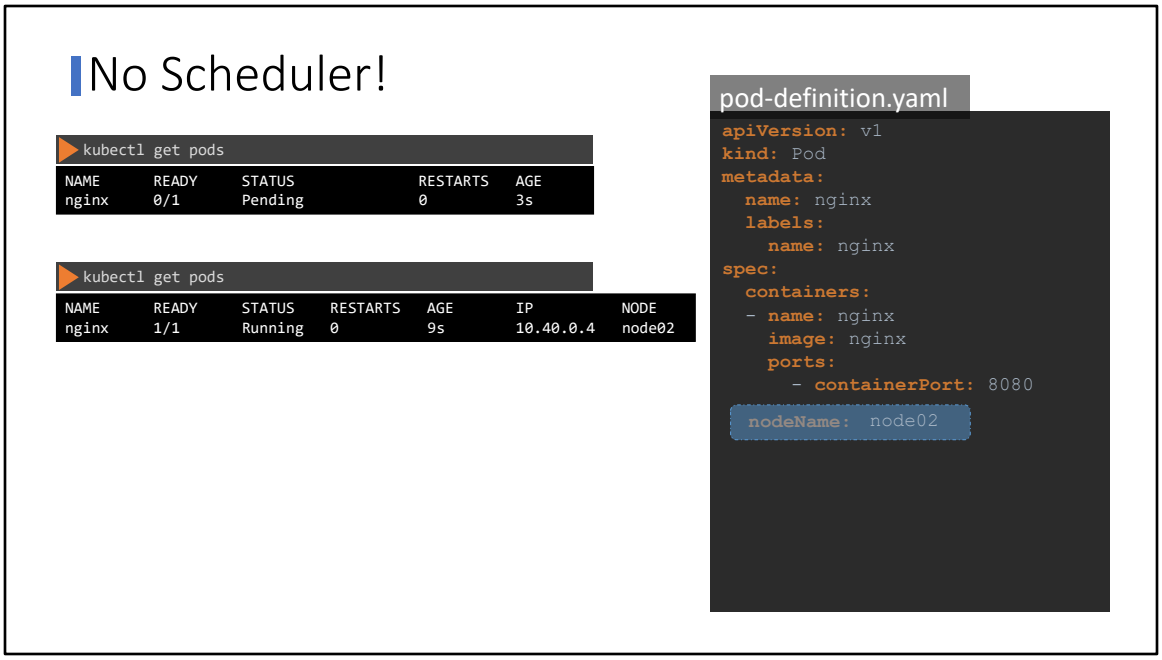

KodeKloud

Hello and welcome to this lecture. In this lecture we look at the different ways of manually scheduling a POD on a node.



What do you do when you do not have a scheduler in your cluster? You, probably do not want to rely on the built in scheduler and instead want to schedule the PODs yourself.

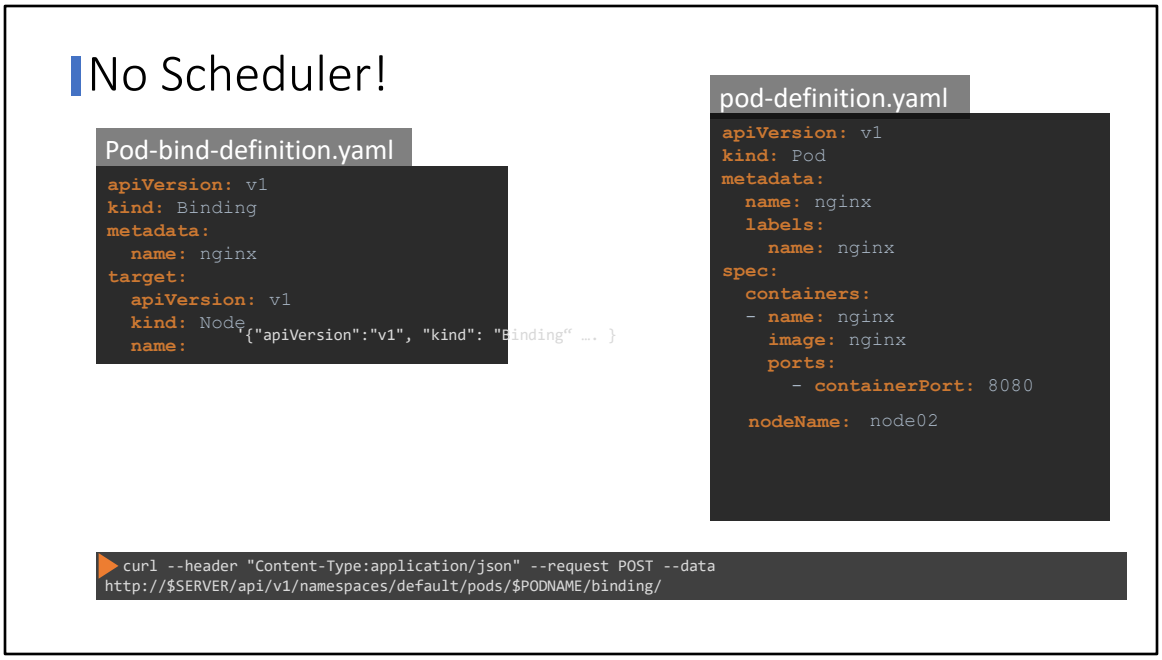
So how exactly does a scheduler work in the backend? <c> Let's start with a sample POD definition file. Every POD has a field called <c> NodeName that, by default, is not set. You don't typically specify this field when you create the manifest file, Kubernetes adds it automtically. <c> The scheduler goes through all the PODs and looks for those that do not have this property set. <c> Those are the candidates for scheduling. <c> It then identifies the right node for the POD, by running the scheduling algorithm that we discussed in the earlier lecture. Once identified, it schedules the POD on the node <c> by setting the nodeName property to the name of the node, by creating a Binding object.



So if there is no scheduler to monitor and schedule nodes, what happens?

<c> The pods continue to be in a pending state. So what can you do about it? You can manually assign pods to nodes yourself.

<c> Well without a scheduler, the easiest way to schedule a POD is to simply set the nodeName field to the name of the node in your pod specification while creating the POD. The pod then gets assigned to the specified node.

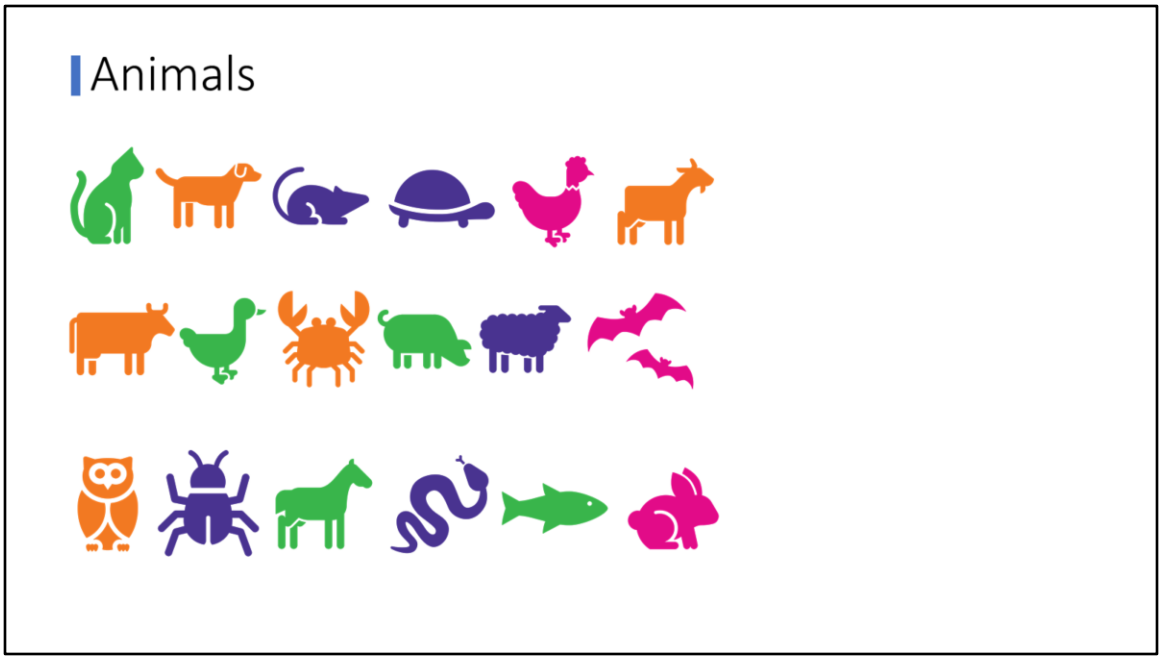


You can only specify the nodeName at creation time. What if the pod is already created and you want to assign the pod to a node? Kubernetes won't allow you to modify the nodeName property of a POD. So another way to assign a node to an existing pod, is to create a <c> binding object and send a post request to the pods binding API. Thus mimicking what the actual scheduler does. <c> In the binding object you specify a target node with the name of the node. Then send a post request to the pods binding API with the data set to the binding object in a JSON format. So you must convert the YAML file into its equivalent JSON format.

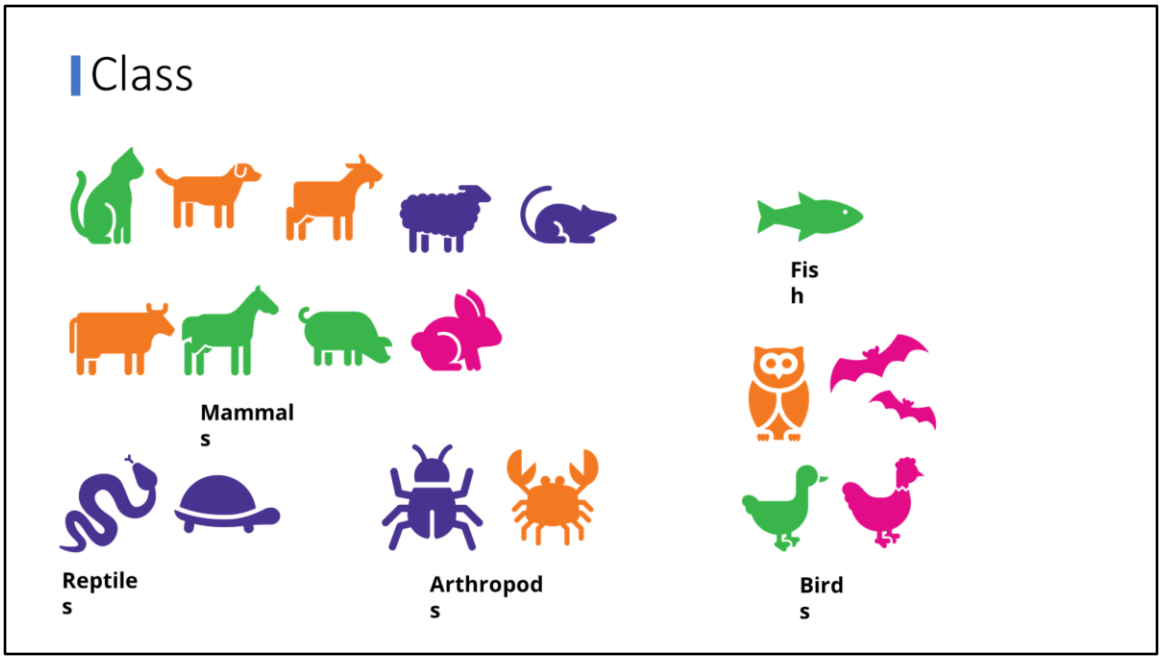
Well that's it for this lecture. Head over to the practice test and practice manually scheduling PODs to Nodes.

Labels, Selectors

Let us start with Labels and Selectors. What do we know about Labels and Selectors already?



Labels and Selectors are a standard method to group things together. Say you have a set of different species. A user wants to be able to filter them based on different criteria.



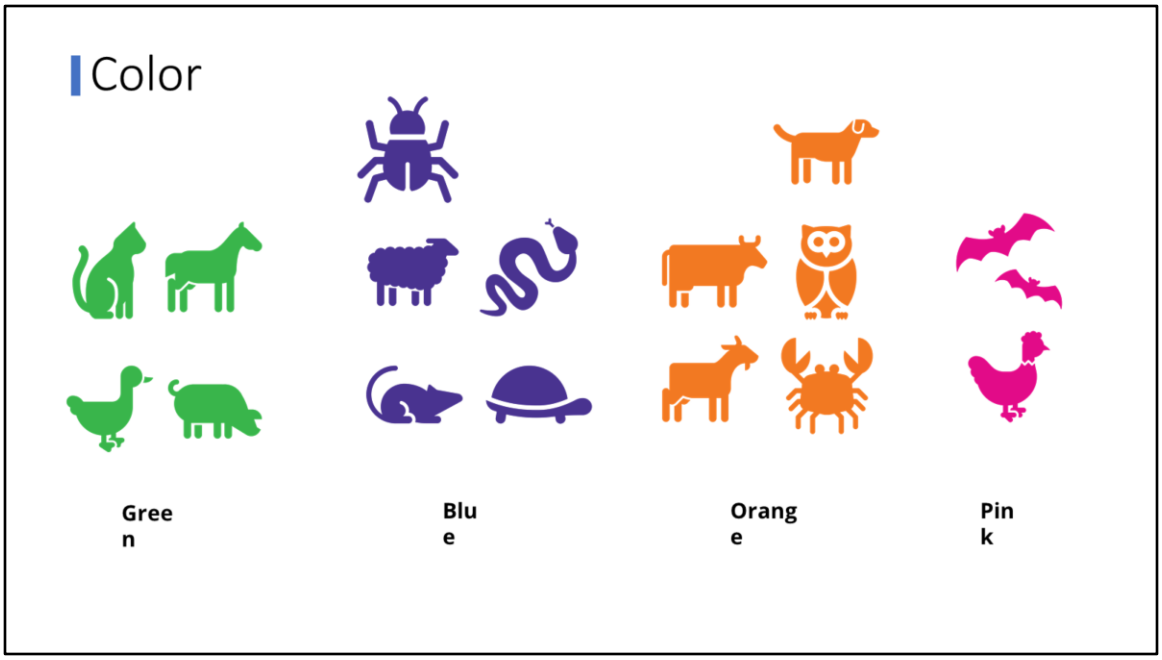
Such as based on their class.

Kind

Domesti
c

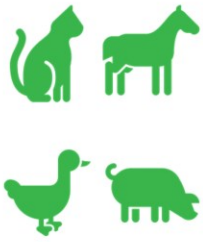
Wild

Or based on their type – domestic or wild.



Or say by color.

Color - Green



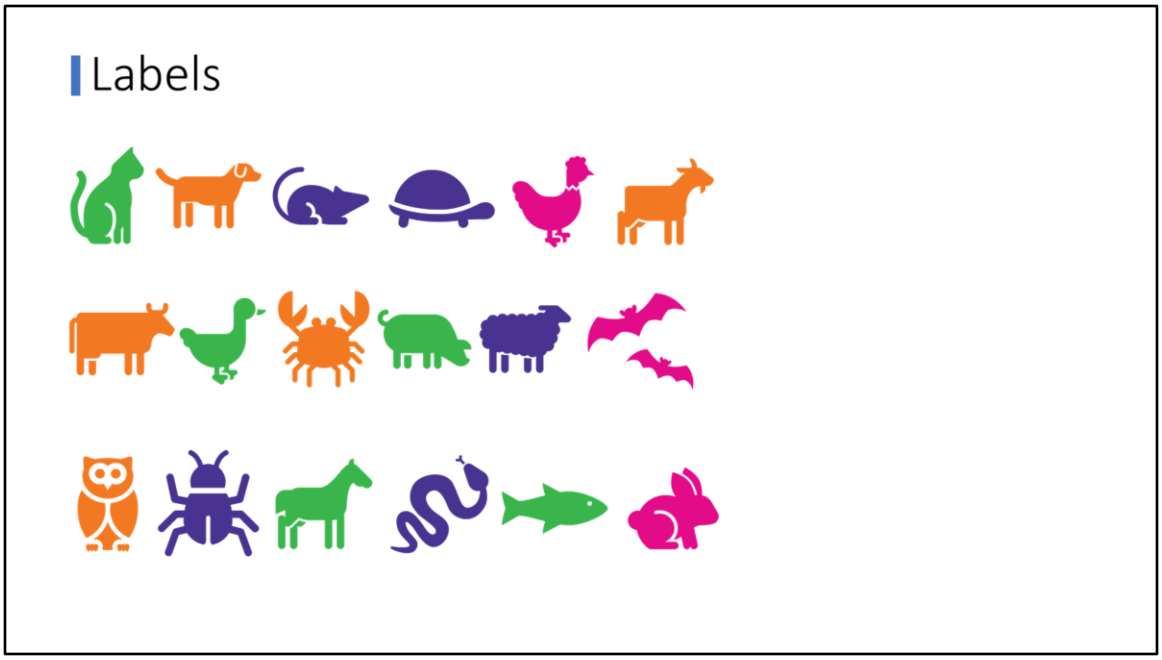
Gree
n

And not just group, you want to be able to filter them based on a criteria. Such as all green animals

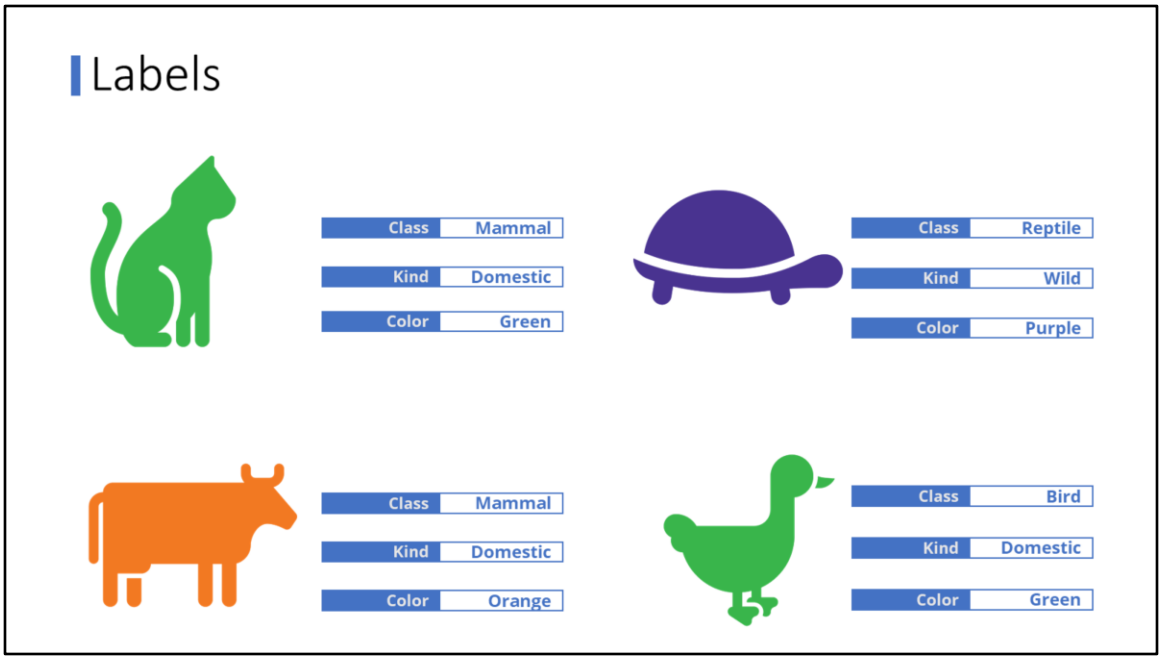
Color – Green - Bird

Green - Bird

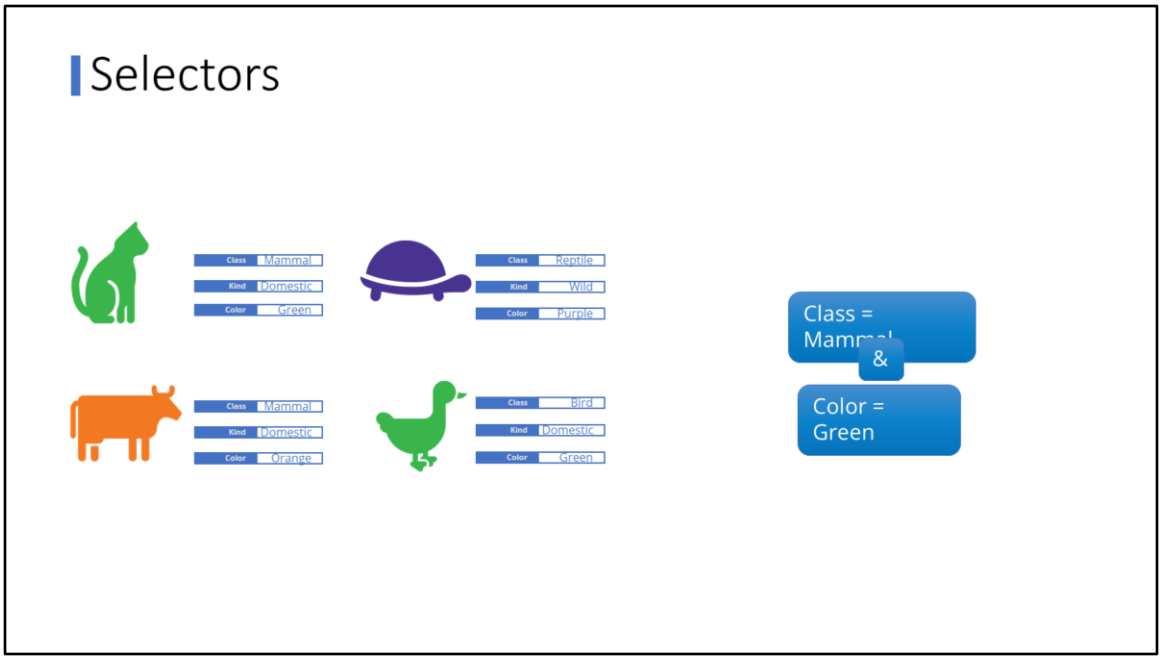
Or with multiple criteria such as everything green that is also a bird. Whatever that classification may be you need the ability to group things together and filter them based on your needs. And the best way to do that, is with labels.



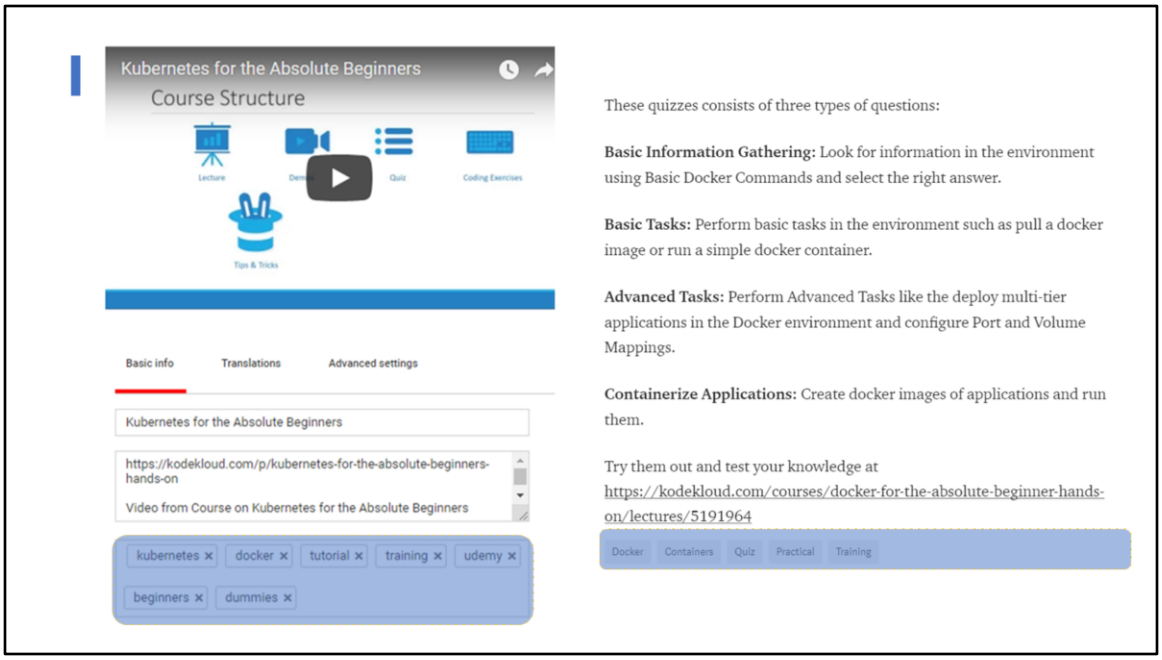
Labels are properties attached to each item.



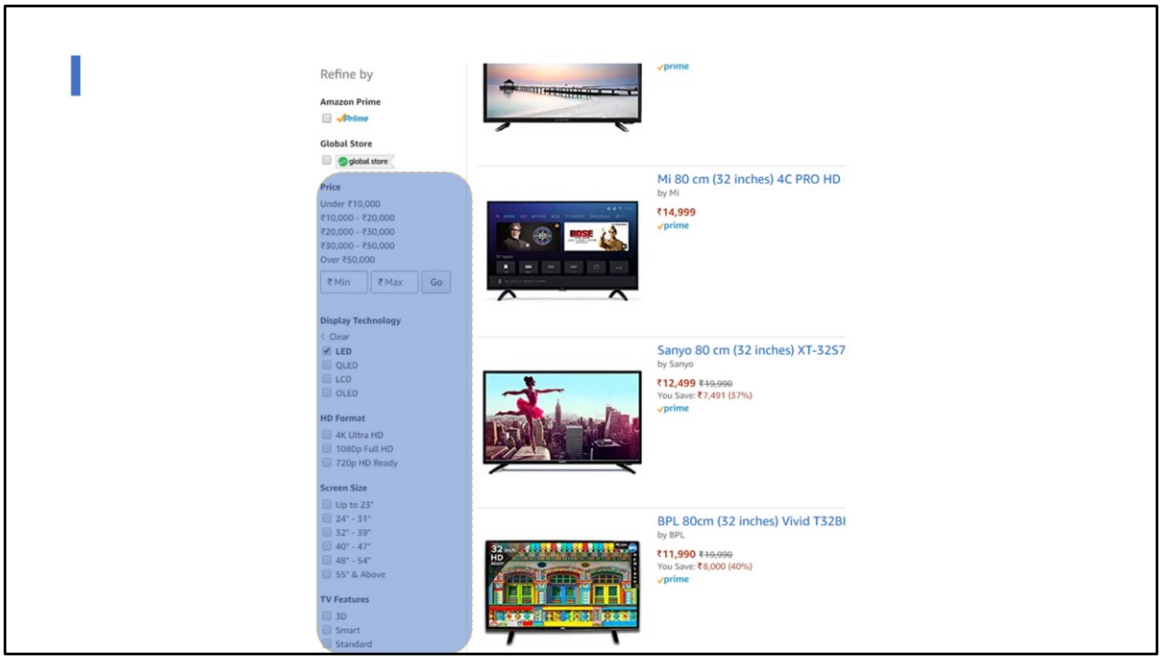
Labels are properties attached to each item. So you add properties to each item for their class, kind and color.



Selectors help you filter these items. For example, when you say class equals mammal, we get a list of mammals. <c> And when you say color equals green, we get the green mammals.

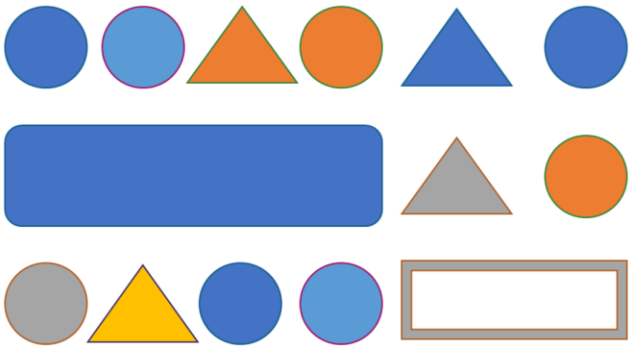


We see labels and selectors used everywhere, such as the keywords you tag to youtube videos or blogs that help users filter and find the right content.



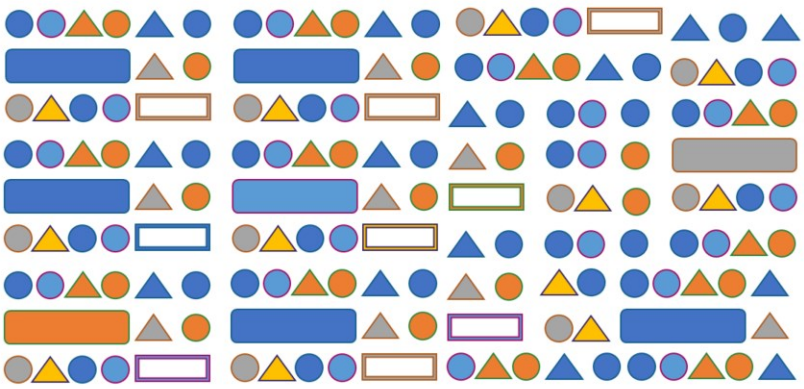
We see labels added to items in an online store that help you add different kinds of filters to view your products.

Labels & Selectors in Kubernetes

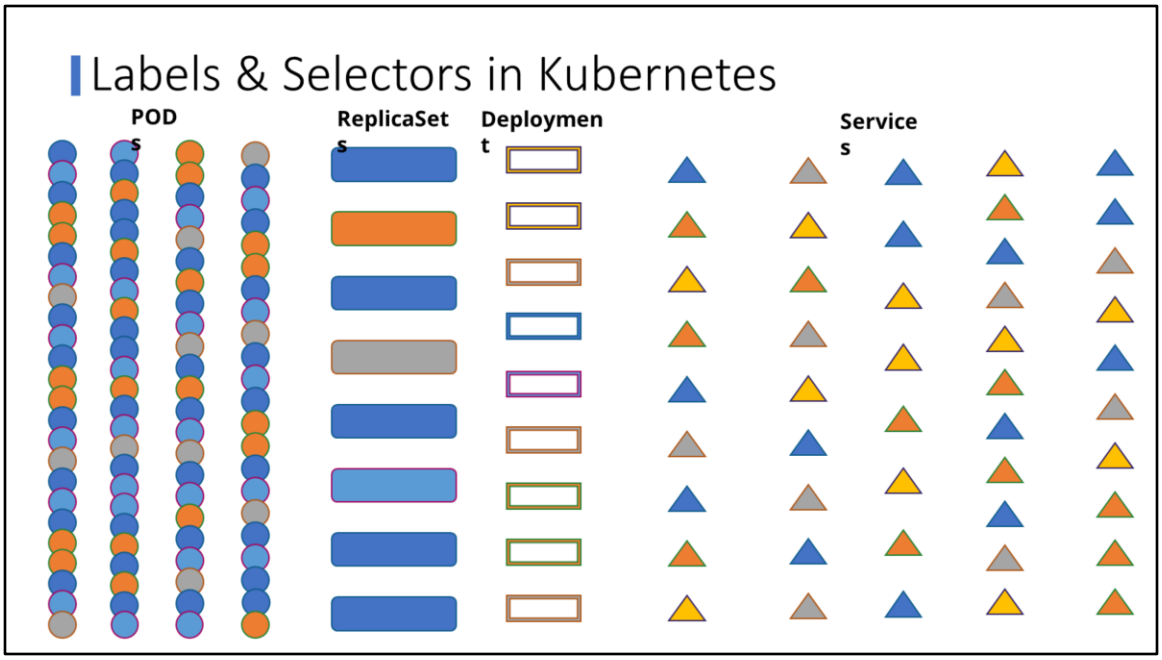


So how are labels and selectors used in Kubernetes? We have created a lot of different types of Objects in Kuberentes. Pods, Services, ReplicaSets and Deployments. For Kubernetes, all of these are different objects. Over time you may end up having 100s and 1000s of these objects in your cluster. Then you will need a way to filter and view different objects by different categories. Like

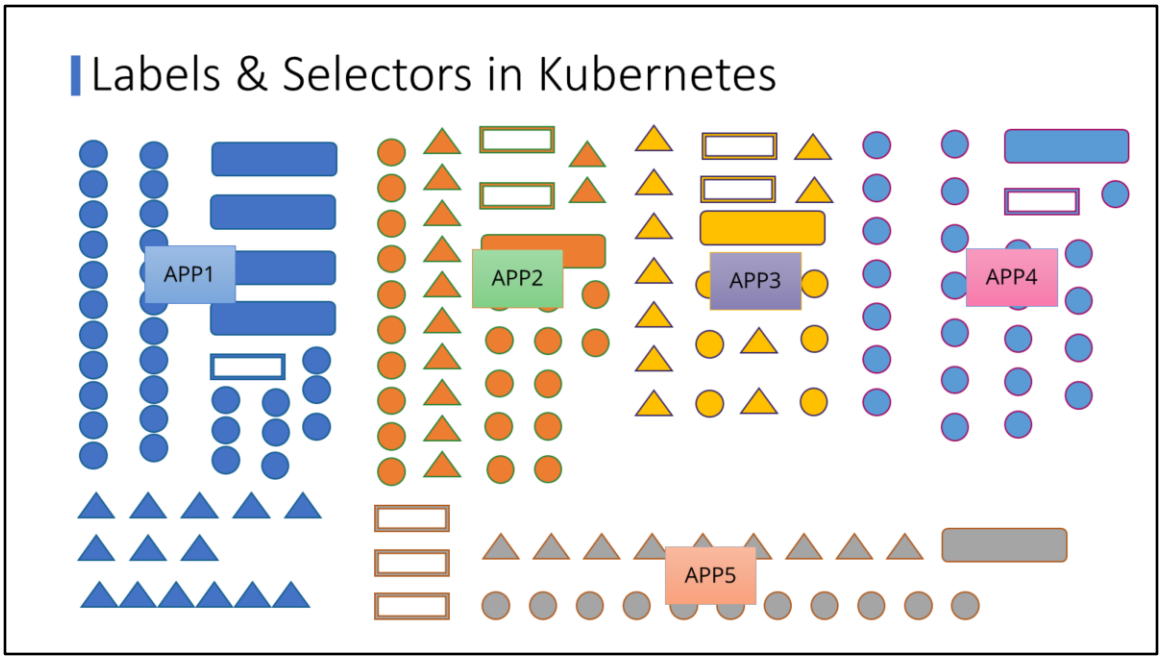
Labels & Selectors in Kubernetes



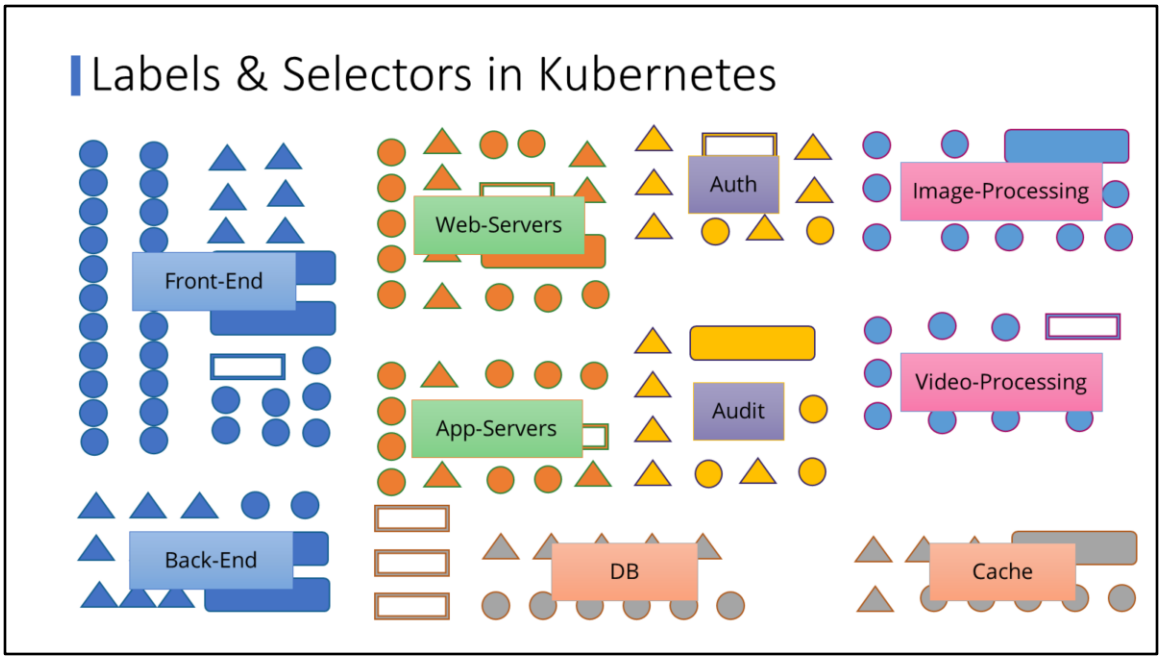
Over time you may end up having 100s and 1000s of these objects in your cluster. Then you will need a way to group, filter and view different objects by different categories.



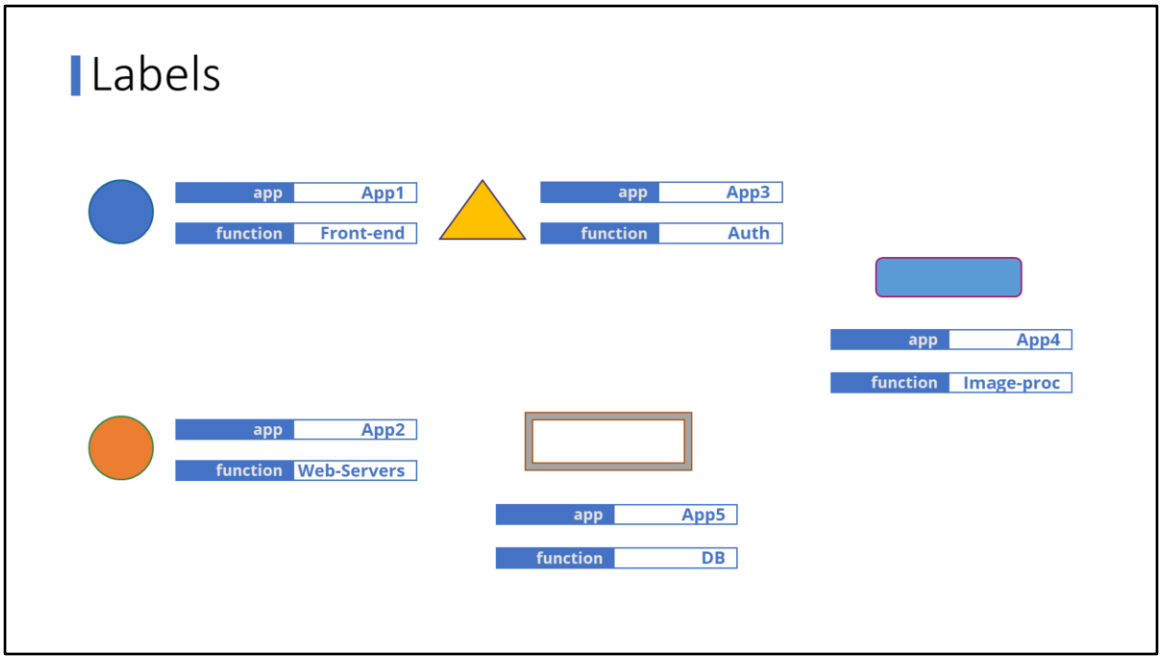
Such as to group objects by their type.



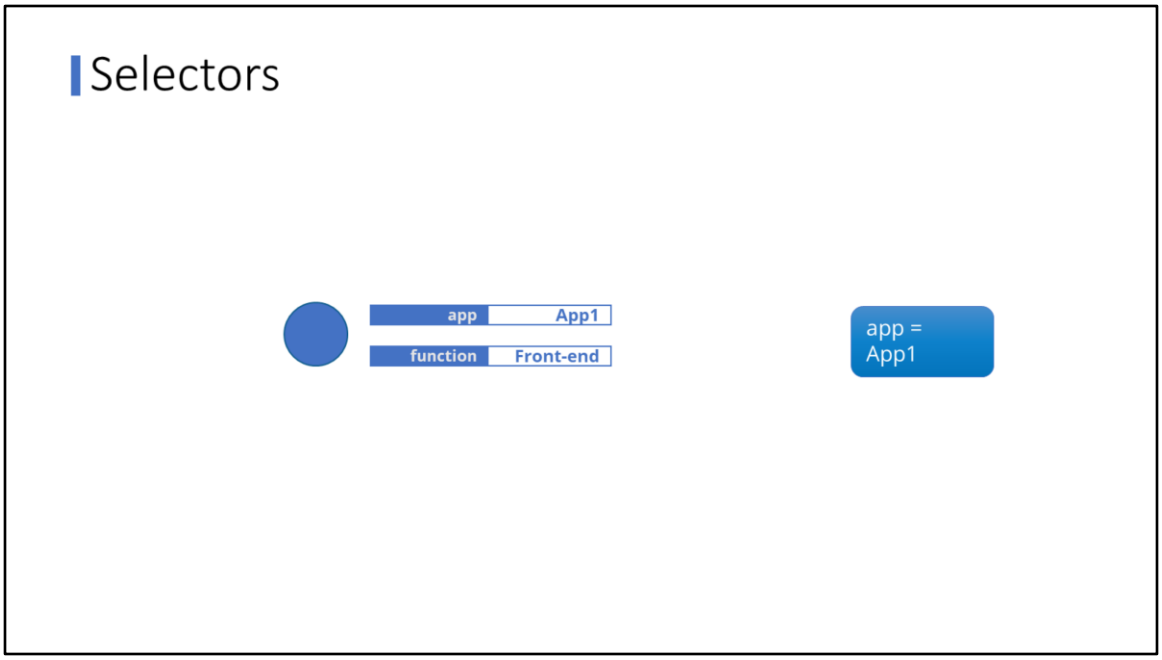
Or view objects by application.



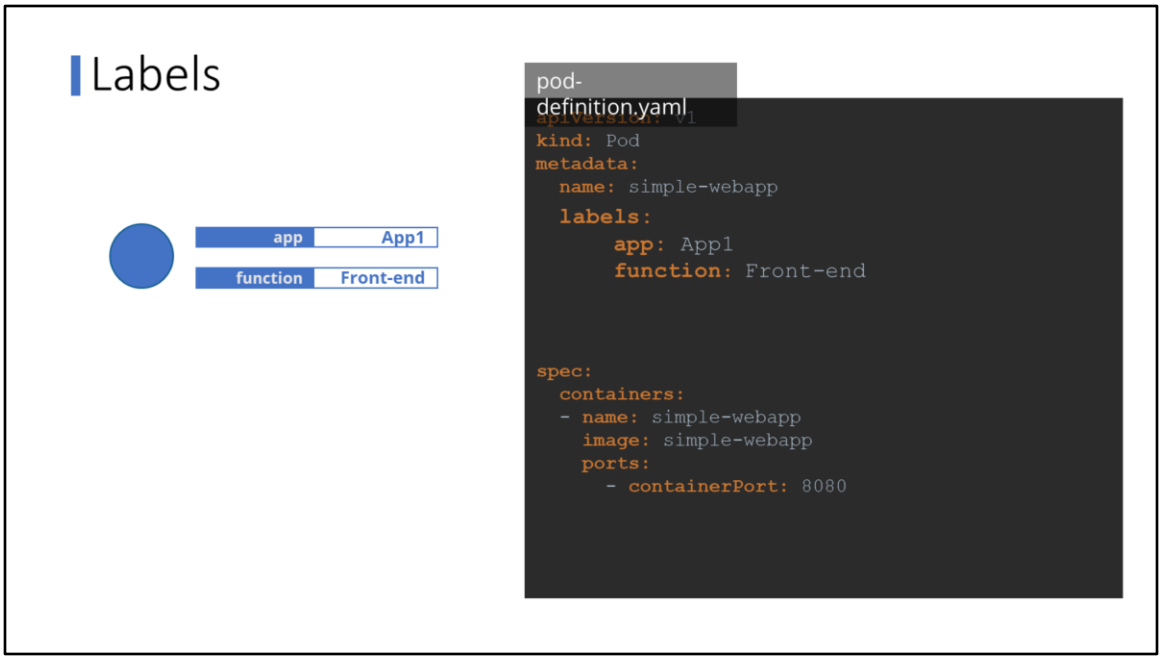
Or by their functionality. Whatever it may be, you can group and select objects using labels and selectors.



For each object attach labels as per your needs, like app, function etc.



Then while selecting, specify a condition to filter specific objects. For example app == App1.

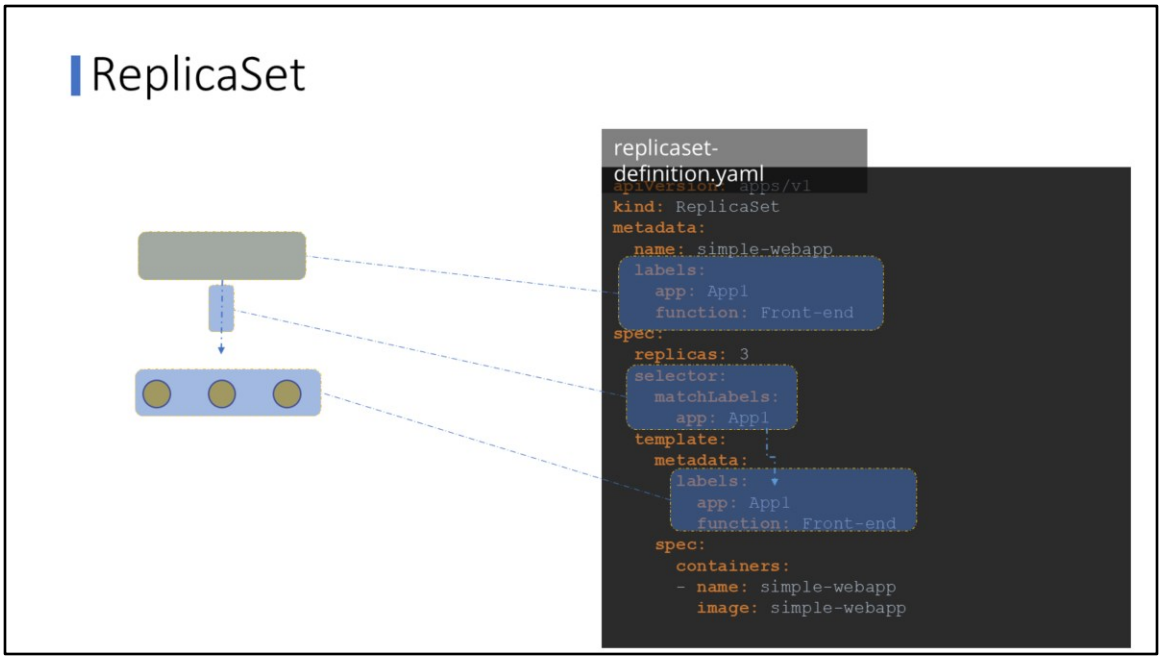


So how exactly do you specify labels in kubernetes. In a pod-definition file, under metadata, create a section called labels. Under that add the labels in a key value format like this. You can add as many labels as you like.

Select

```
kubectl get pods --selector app=App1
NAME                READY     STATUS      RESTARTS   AGE
simple-webapp        0/1    Completed   0          1d
```

Once the pod is created, to select the pod with the labels use the kubectl get pods command along with the selector option, and specify the condition like app=App1.



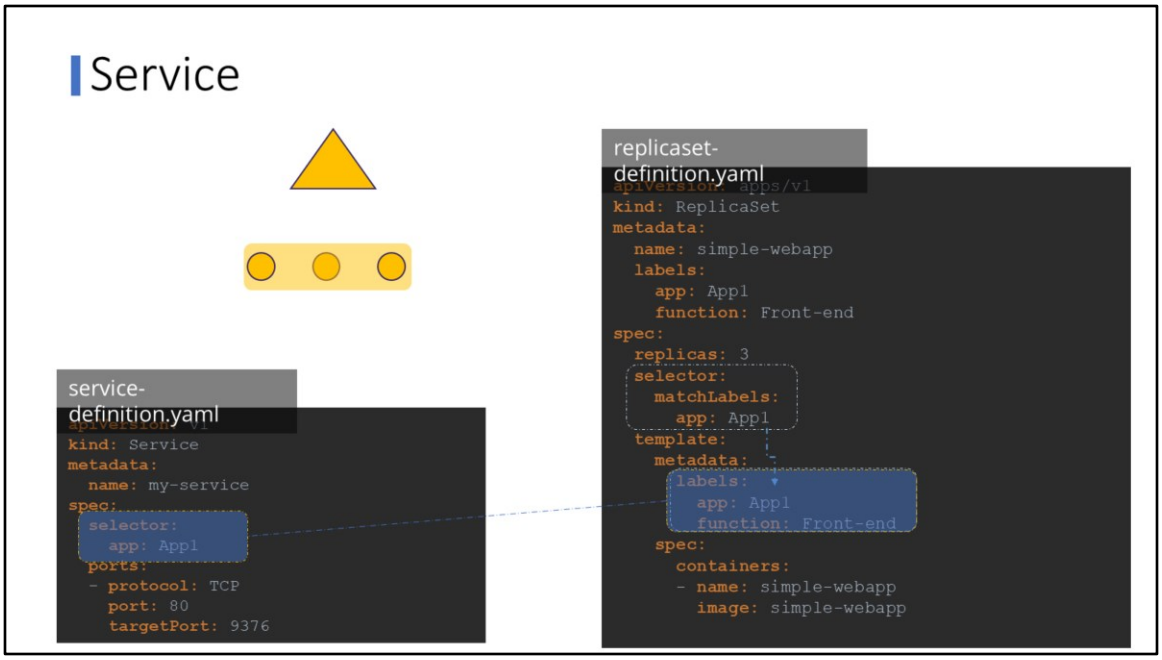
Now this is one use case of labels and selectors. <c> Kubernetes objects use labels and selectors internally to connect different objects together. For example to create a replicaset consisting of 3 different pods, we first label the pod definition and use selector in a replicaset to group the pods . In the replica-set definition file, you will see labels defined in two places. Note that this is an area where beginners tend to make a mistake. <c> The labels defined under the template section are the labels configured on the pods. The labels you see at the top <c> are the labels of the replica set. We are not really concerned about that for now, because we are trying to get the replicaset to discover the pods. The labels on the replicaset will be used if you were configuring some other object to discover the replicaset. In order to connect the replica set to the pods, <c> we configure the selector field under the replicaset specification to match the labels defined on the pod. A single label will do if it matches correctly. However if you feel there could be other pods with that same label but with a different function, then you could specify both the labels to ensure the right pods are discovered by the replicaset.

ReplicaSet

replicaset-definition.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: App1
  template:
    metadata:
      labels:
        app: App1
        function: Front-end
    spec:
      containers:
      - name: simple-webapp
        image: simple-webapp
```

On creation, if the labels match, the replicaset is created successfully.



It works the same for other objects like a service. <c> When a service is created, it uses the selector defined in the service definition file to match the labels set on the pods in the replicaset-definition file.

Annotations

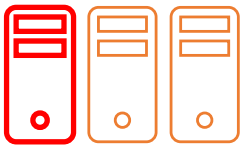
replicaset-definition.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
  annotations:
      buildversion: 1.34
spec:
  replicas: 3
  selector:
    matchLabels:
      app: App1
  template:
    metadata:
      labels:
        app: App1
        function: Front-end
    spec:
      containers:
      - name: simple-webapp
        image: simple-webapp
```

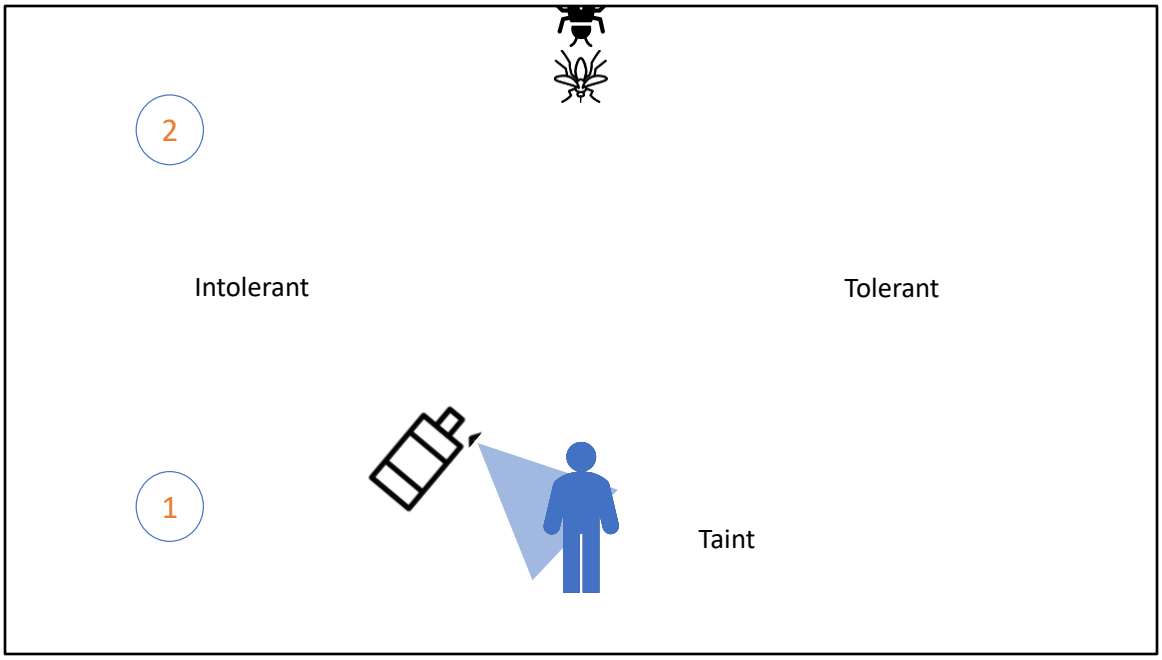
Finally let's look at annotations. While labels and selectors are used to group and select objects, annotations are used to record other details for informatory purpose. For example tool details like name, version build information etc or contact details, phone numbers, email ids etc, that may be used for some kind of integration purpose.

Well, that's it for this lecture on Labels and Selectors. Head over to the coding exercises section and practice working with labels and selectors.

Taints And Tolerations



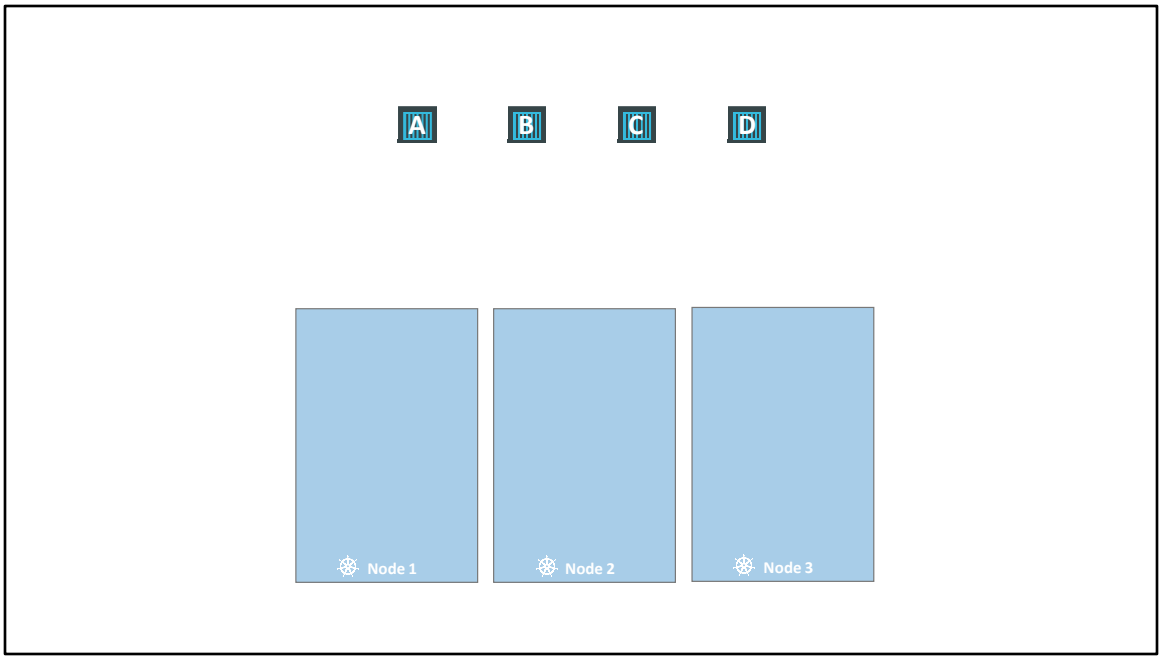
Hello and welcome to this lecture. In this lecture we will talk about Taints and Tolerations in Kubernetes.



The concept of Taints and Tolerations can be a bit confusing for beginners. So, we will try to understand what they are using an analogy of a bug approaching a person. My apologies in advance. But this is the best I could come up with. To prevent the bug from landing on the person, we spray the person with a repellent spray or a taint as we will call it here. The bug is INTOLERANT to the smell, so on approaching the person, the taint applied on the person throws the bug off.

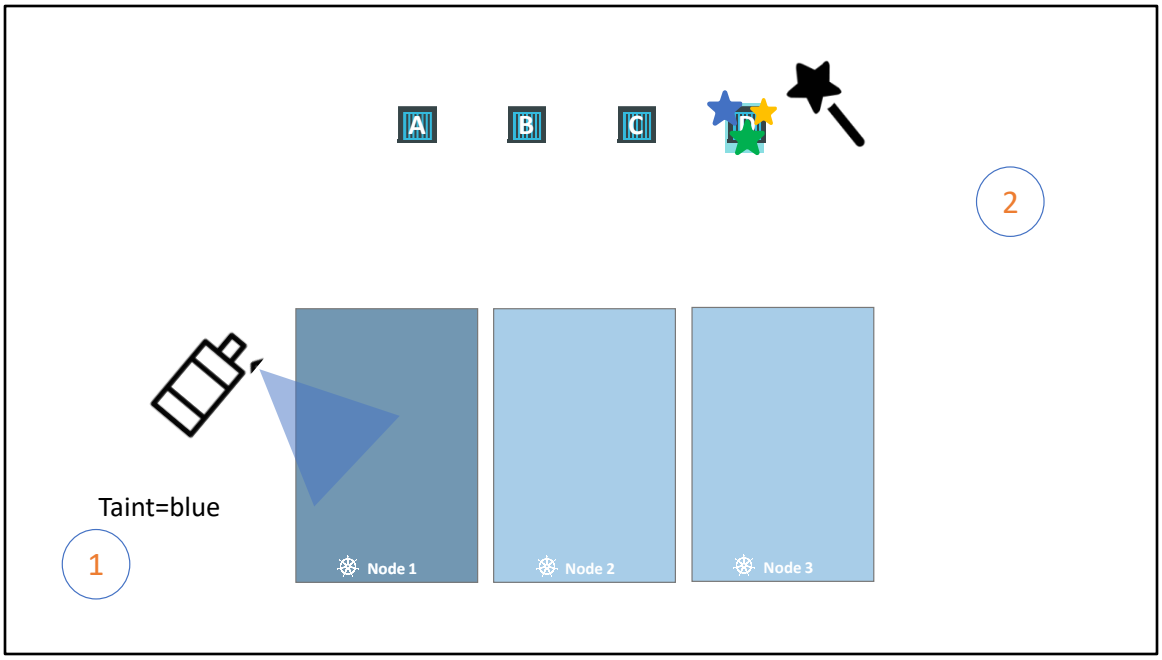
However, there could be other bugs that are TOLERANT to this smell. And so the taint doesn't really affect them, and so they end up landing on the person.

So, there are two things that decided if a bug can land on a person. First, the taint on the person. And second, the bug's toleration level to that particular taint.



Getting back to Kubernetes, the person is a Node and the Bugs are PODs. Now taints and tolerations have nothing to do with security or intrusion on the cluster. Taints and Tolerations are used to set restrictions on what PODs can be scheduled on a node.

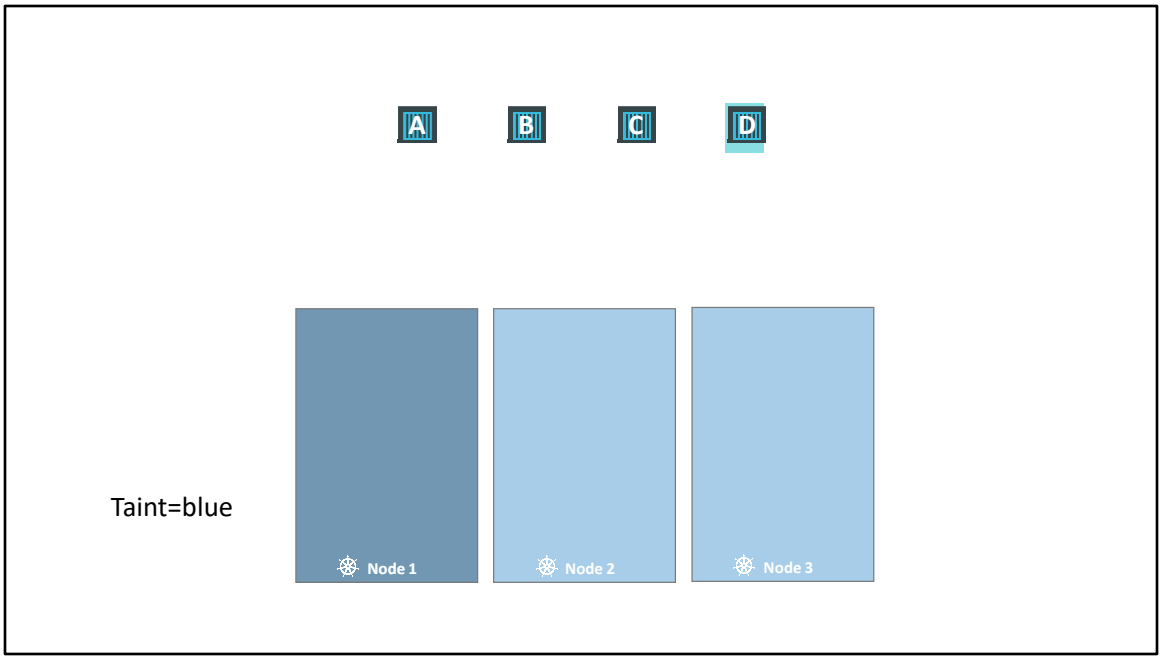
Let us start with a simple cluster with 3 worker nodes. The nodes are named 1,2 and 3. We also have a set of PODs that are to be deployed on these nodes. Let us call them A,B,C and D. When the PODs are created, Kubernetes scheduler tries to place these PODs on the available worker nodes. As of now, there are no restrictions or limitations and as such the scheduler places the PODs across all of the nodes to balance them out equally.



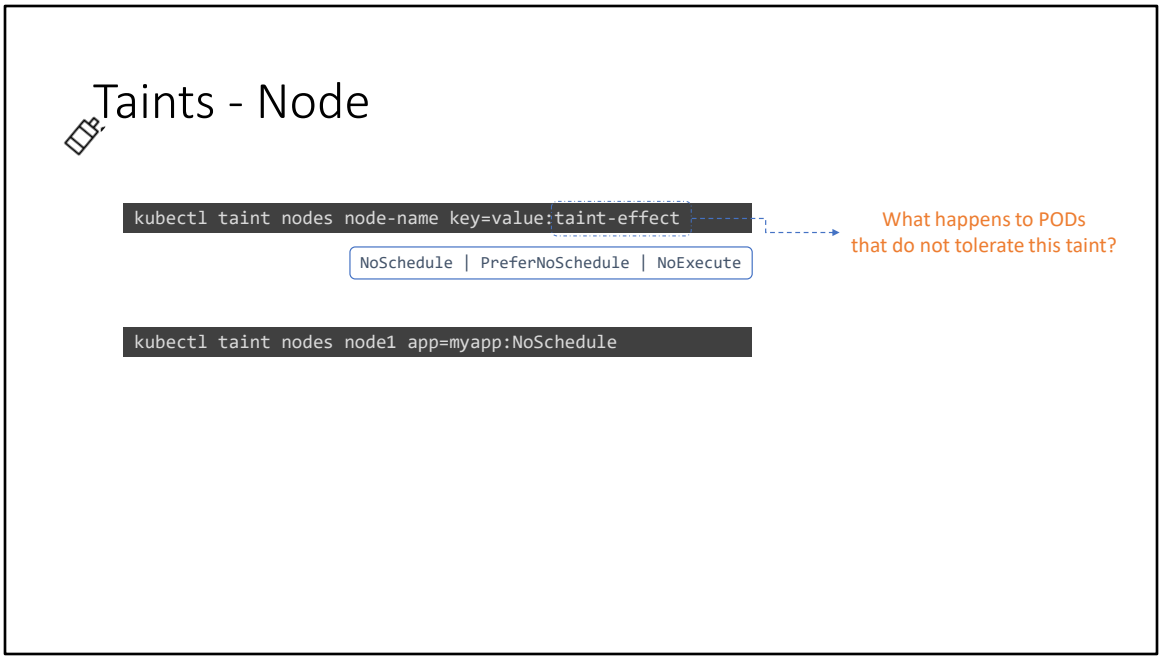
Now, let us assume that we have dedicated resources on Node1 for a particular use case or application. So we would like only those PODs that belong to this application to be placed on Node1.

First, we prevent all PODs from being placed on the Node by placing a taint on the node, let's call it blue. By default, PODs have no tolerations. Which means, unless specified otherwise, none of the PODs can tolerate any taint. So in this case none of the PODs can be placed on Node1 as none of them can tolerate the taint blue. This solves half of our requirement – no unwanted PODs are going to be placed on this Node1. The other half is to enable certain PODs to be placed on this node. For this we must specify which PODs are tolerant to this particular taint. In our case we would like to allow only POD D to be placed on this Node. And so we add a toleration to pod D.

Pod D is now tolerant to blue. So when the scheduler tries to place this pod on Node1, it goes through. Node1 can now only accept pods that can tolerate the taint blue.



So with all the taints and tolerations in place, this is how the PODs would be scheduled. The scheduler tries to place POD A on Node1, but due to the taint it is thrown off and it goes to Node2. The scheduler then tries to place POD B on Node1, but again due to the taint, it is thrown off and is placed on Node3 which happens to be the next free node. The scheduler then tries to place POD C to Node1, it is thrown off again and ends up on Node 2. Finally the scheduler tries to place POD D on Node1, since the POD is tolerant to Node1, it goes through.



So remember, Taints are set on Nodes and Tolerations are set on PODs. So how do you this? Use the kubectl taint nodes command to taint a node. Specify the name of the node to taint, followed by the taint itself which is a key value pair. For example, if you would like to dedicate the node to PODs in application myapp, then the key value pair would be app=myapp. The taint key value pair is followed by a colon and a taint effect. The taint effect defines what would happen to the PODs, if they don't tolerate the taint. There are 3 taint-effects. NoSchedule – which means the PODs will not be scheduled on the node, which is what we have been discussing. PreferNoSchedule – which means the system will try to avoid placing a POD on the Node, but that is not guaranteed. And third is NoExecute – which means that new PODs will not be scheduled on the node and existing PODs on the node, if any, will be evicted if they do not tolerate the taint. These PODs may have been scheduled on the Node before the taint was applied to the node.

An example command would be to taint node node1 with the key value pair app=myapp and an effect of NoSchedule.

Tolerations - PODs

```
kubectl taint nodes node1 app=myapp:NoSchedule
```

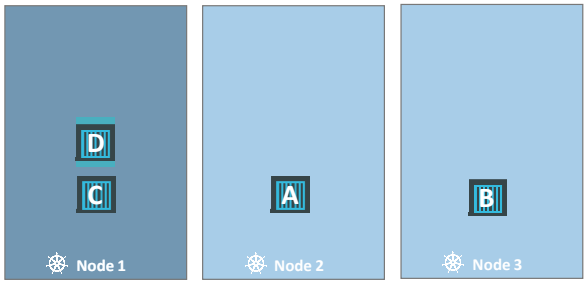
pod-definition.yml

```
apiVersion:
kind: Pod
metadata:
 name: myapp-pod
spec:
  containers:
  - name: nginx-container
    image: nginx

  tolerations:
  - key:"    "
    operator:"Equal"
    value:"    "
    effect:"          "
```

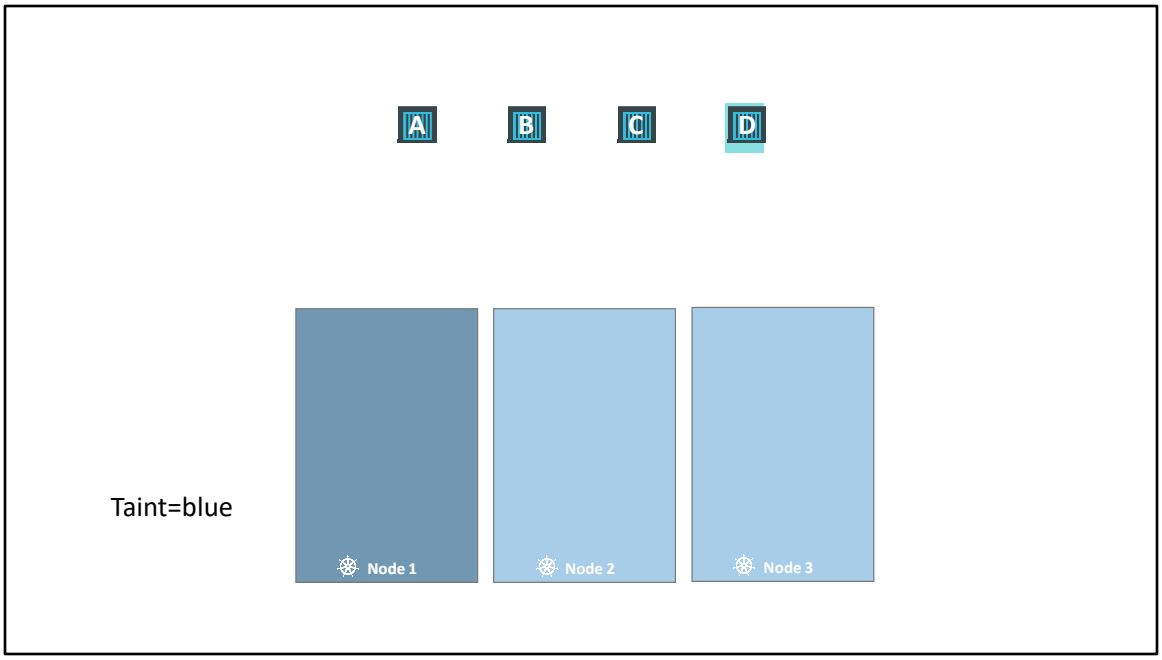
Tolerations are added to PODs. To add a toleration to a POD, first pull up the pod definition file. In the spec section of the pod-definition file add a section called tolerations. Move the same values used while creating the taint under this section. The key is app, operator is Equal, value is myapp, and the effect is NoSchedule. And remember all of these values need to be encoded in double quotes.

When the pods are now created or updated with the new tolerations, they are either not scheduled on nodes or evicted from existing nodes depending on the effect set.

Taint - NoExecute



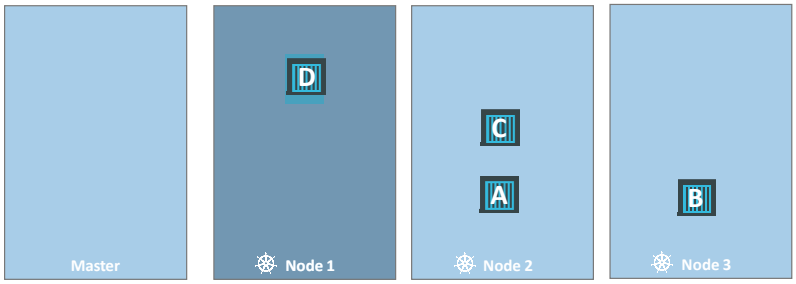
Let us try to understand the NoExecute taint effect. In this example, we have 3 nodes running some workload. We do not have any taints or tolerations at this point, so they are scheduled this way. We then decided to dedicate Node1 for a special application. And as such we taint the Node with the application name and add a toleration to the POD that belongs to the application, which happens to be POD D in this case. While tainting the node, we set the taint effect to NoExecute. As such, once the taint on the node takes effect, it evicts POD C from the node, which simply means the POD is killed. The POD D continues to run on the Node as it has a toleration to the taint.



And a final note before ending this lecture. Going back to our original scenario were we have taints and tolerations configured and the PODs are going to be placed on the Nodes. Remember, Taints and Tolerations are only meant to restrict NODEs from accepting only certain PODs. In this case, Node1 can only accept Pod D. But it does not guarantee that POD D will always be placed on Node1. Since there are no taints or restrictions applied on the other two nodes, pod D may very well be placed on any of the other two nodes. SO remember, taints and tolerations does not tell the POD to go to only a particular Node. Instead it tells the Node to only accept PODs with certain tolerations .

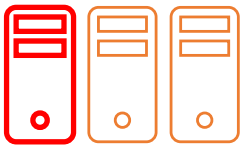
If your requirement is to restrict a POD to certain nodes, it is achieved through another concept called as Node Affinity which we will discuss in another lecture.

```
kubectl describe node kubemaster | grep Taint
Taints:             node-role.kubernetes.io/master:NoSchedule
```

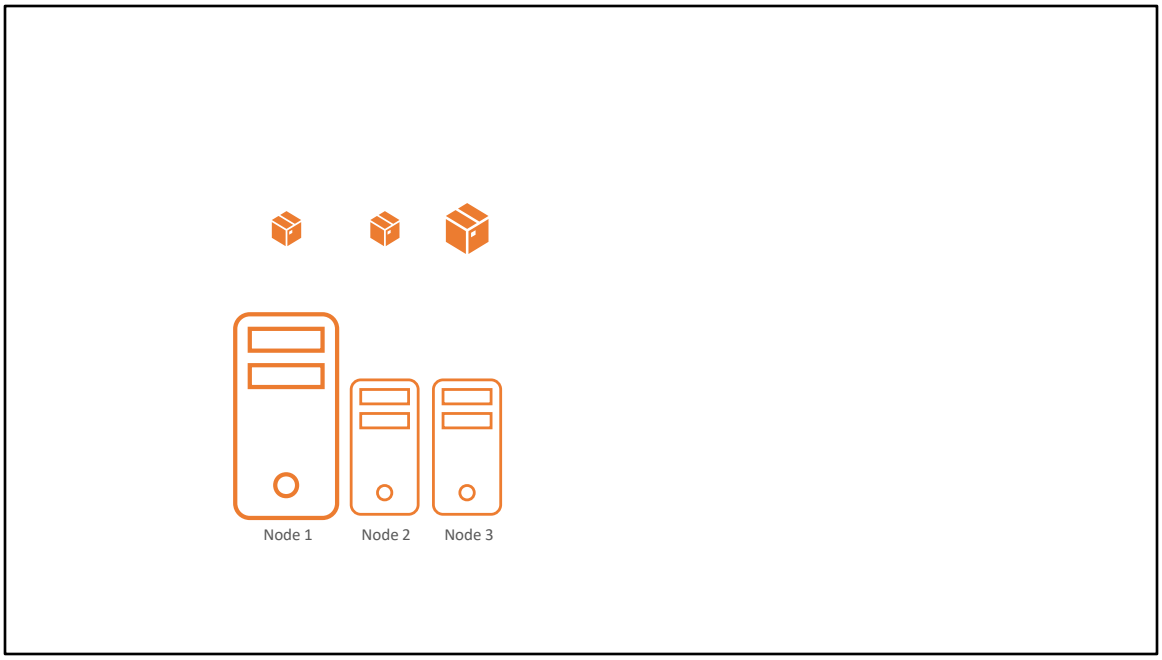


Finally, while we are on this topic, let us also take a look at an interesting fact. So far we have only been referring to the worker nodes. But we also have a master node in the cluster, which is technically just another node and has all capabilities of hosting a POD. Now I am not sure if you noticed, the scheduler does not schedule any PODs on the master node. Why is that? When the Kubernetes cluster is first setup, a taint is set on the Master node automatically, that prevents any PODs from being scheduled on this node. You can see this as well as modify this behavior if required. However, a best practice is to not deploy application workloads on a Master server. To see this taint, run a kube control describe node kubemaster command and look for the Taints section. You will see a taint set to not schedule any pods on the master node.

Node Selectors



Hello and welcome to this lecture. In this lecture we will talk about Taints and Tolerations in Kubernetes.

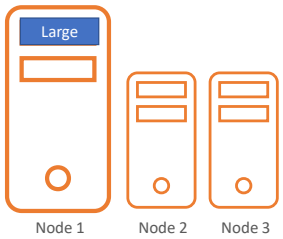


Let us start with a simple example. You have a 3 node cluster. Of which two are smaller nodes, and one of them is a larger node configured with higher resources. You have different kinds of workloads running in your cluster. You would like to dedicate the data processing workloads that require higher horse power to the larger node as that is the only node that will not run out of resources in case the job demands extra resources. However, in the current, default setup, any pods can go to any nodes, so POD C, in this case, may very well end up on Nodes 2 or 3, which is not desired.

Node Selectors

pod-definition.yml

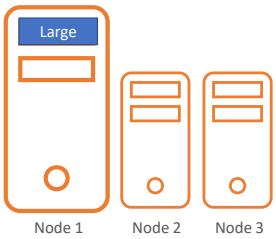
```
apiVersion:
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor

  nodeSelector:
    size: Large
```



To solve this we can set a limitation on the PODs to run on particular Nodes. There are two ways to do this. The first is using Node Selectors, which is the simple and easier method. Let us look at the pod-definition file we created earlier. This file has a simple definition to create a POD with a data processing image. To limit this POD to run on the large node, we add a new property called nodeSelector to the spec section, and specify the size as Large. But wait a minute, were did the size large come from and how does kubernetes know which is the large node? The key value pair of size and large are in fact labels assigned to the nodes. The scheduler uses these labels to match and identify the right node to place the PODs on. Labels and Selectors are a topic we have seen many times through out this Kubernetes course, such as with services, replicasets and deployments. To use labels in a nodeSelector like this, you must have first labelled your nodes prior to creating this POD.

Label Nodes

```
kubectl label nodes <node-name> <label-key>=<label-value>
kubectl label nodes node-1 size=Large
```



So let us go back and see how we can label the nodes. To label a node, use the command kubectl label nodes followed by the name of the node and the label in a key value pair format. In this case, it would be kubectl label nodes node1 followed by the label in a key value format, such as size=Large

Node Selector

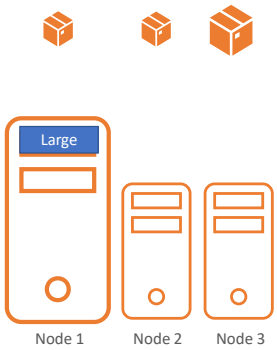
pod-definition.yml

```
apiVersion:
kind: Pod
metadata:
 name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor

  nodeSelector:
    size: Large
```

```
kubectl create -f pod-definition.yml
```

Now, that we have labelled the node, we can get back to creating the POD, this time with the node selector set to a size of large.

Node Selector



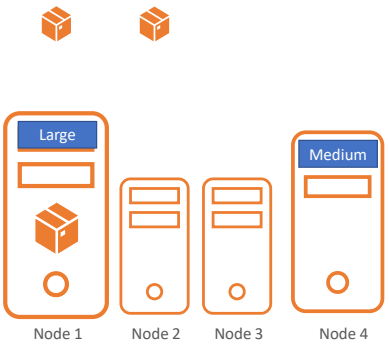
pod-definition.yml

```yaml
apiVersion:
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor

  nodeSelector:
    size: Large
```

```
kubectl create -f pod-definition.yml
```

When the pod is now created, it is placed on Node1 as desired.

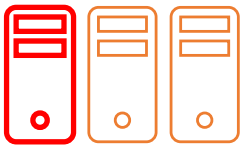
Node Selector - Limitations



- Large OR Medium?
- NOT Small

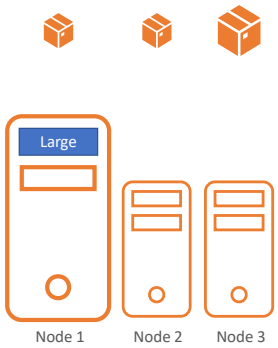
Node selector served our purpose. But it has limitations. We used a single label and selector to achieve our goal here, but what if our requirement is much more complex? For example, we would like to say something like, place the POD on a Large or Medium node? Or something like place the POD on any Nodes that are NOT small. You cannot achieve this using Node Selectors. For this Node Affinity and Anti-Affinity features were introduced.

Node Affinity



Hello and welcome to this lecture. In this lecture we will talk about Node Affinity feature in Kubernetes.

PODs to Nodes



The primary purpose of Node Affinity feature is to ensure that PODs are hosted on particular nodes. In this case to ensure the large data processing POD ends up on Node 1. In the previous lecture we did this easily using Node Selectors.

Node Selectors

pod-definition.yml

```
apiVersion:
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor
  nodeSelector:
    size: Large
```

- Large OR Medium?
- NOT Small

We discussed that you cannot provide advanced expressions like OR or NOT with Node Selectors.

Node Affinity

pod-definition.yml

```
apiVersion:
kind: Pod

metadata:
 name: myapp-pod
spec:

  containers:
  - name: data-processor
    image: data-processor

  nodeSelector:
    size: Large
```

pod-definition.yml

```
apiVersion:
kind:

metadata:
 name: myapp-pod
spec:

 containers:
  - name: data-processor
    image: data-processor

 affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: size
          operator: In
          values:
          - Large
```

The Node Affinity feature provides us with advanced capabilities to limit POD placement on specific Nodes. With great power comes great complexity. So the simple Node Selector specification will now look like this with Node Affinity although both does exactly the same thing – place the POD on the Large node.

Node Affinity

pod-definition.yml

```
apiVersion:
kind:
metadata:
 name: myapp-pod
spec:
 containers:
  - name: data-processor
    image: data-processor
 affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: size
          operator: Exists
```

Let us look at it a bit closer. Under spec you have affinity and then nodeAffinity under that. And then you have a property that looks like a sentence called requiredDuringSchedulingIgnoredDuringExection. No description needed for that. And then you have the nodeSelectorTerms that is an array and that is where you will specify the key and value pairs. The key value pairs are in the form, key, operator and value, were the operator is In. The In operator ensures that the POD will be placed on a Node whose label size has any value in the list of values specified here. In this case, it is just one called Large. If you think your POD could be placed on a Large OR a medium node, you could simply add the value to the list of values like this. You could use the <click> NotIn operator to say something like, size NotIn Small, where Node Affinity will match the nodes with a size not set to small. We know that we have only set the label size to Large and Medium nodes. The smaller nodes don't even have the label set. So we don't really have to even check the value of the label. As long as we are sure we don't set a label size to the smaller nodes, using the <click> Exists operator will give us the same result. The exists operator will simply check if the label size exists on the nodes, and you don't need the values section for that, as it does not compare the values. There are a number of other operators as well. Check the documentation for specific details.

[pause] Now, we understand all of this, and we are comfortable with creating a pod with specific affinity rules. When the PODs are created, these rules are considered and the PODs are placed onto the right nodes. But what if Node Affinity could not match a node with the given expression? In this case , what if there are no nodes with the label called size? Say we had the labels and the PODs are scheduled, what if someone changes the label on the node at a future point in time? Will the POD continue to stay on the node? All of this is answered by the long sentence-like property under nodeAffinity, which happens to be the type of nodeAffinity.

Node Affinity Types

Available:

`requiredDuringSchedulingIgnoredDuringExecution`

`preferredDuringSchedulingIgnoredDuringExecution`

Planned:

`requiredDuringSchedulingRequiredDuringExecution`

The type of node affinity defines the behavior of the scheduler with respect to Node Affinity and the stages in the lifecycle of the POD.

There are currently two types of NodeAffinity available – requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution. And there are two additional types of Node Affinity planned - requiredDuringSchedulingRequiredDuringExecution and preferredDuringSchedulingRequiredDuringExecution

We will now break this down, to understand further.

Node Affinity Types

Available:

requiredDuringSchedulingIgnoredDuringExecution

preferredDuringSchedulingIgnoredDuringExecution

	DuringScheduling	DuringExecution
Type 1	Required	Ignored
Type 2	Preferred	Ignored

We will now break this down, to understand further. We will start by looking at the two available affinity types. There are two states in the lifecycle of a POD when considering NodeAffinity. During scheduling and during execution. During scheduling is the state were a POD does not exist and is created for the first time. We have no doubt that when a POD is first created, the affinity rules specified are considered to place the PODs on the right nodes. Now what if the nodes with matching labels are not available? For example, we forgot to label the Node as Large. That is were the type of node affinity used comes into play. If you select the required type, which is the first one, the scheduler will mandate the POD to be placed on a node with the given affinity rules. If it cannot find one, the POD will not be scheduled. This type will be used in cases were the placement of the POD is crucial. If a matching node does not exist, the pod will not be scheduled. But, lets say the POD placement is less important than running the workload itself, you could set it to preferred, and in cases were a matching node is not found, the scheduler will simply ignore node affinity rules and place the pod on any available node. This is a way of telling the scheduler, hey try your best to place the POD on matching node, but if you really cannot find one, just place it anywhere.

The second part of the property, or the other state is During Execution. During

execution is the state were a POD has been running and a change is made in the environment that affects Node Affinity, such as a change in the label of a Node. For example, say an administrator removed the label we set earlier called size & Large from the node. Now what would happen to the PODs that are running on the node? As you can see the two types of Node Affinity available today has this value set to Ignored, which means the PODs will continue to run and any changes in Node Affinity will not impact them once they are scheduled.

Node Affinity Types

Planned:

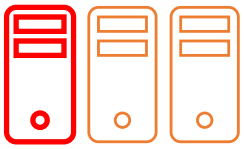
requiredDuringSchedulingRequired**DuringExecution**

	DuringScheduling	DuringExecution
Type 1	Required	Ignored
Type 2	Preferred	Ignored
Type 3	Required	Required

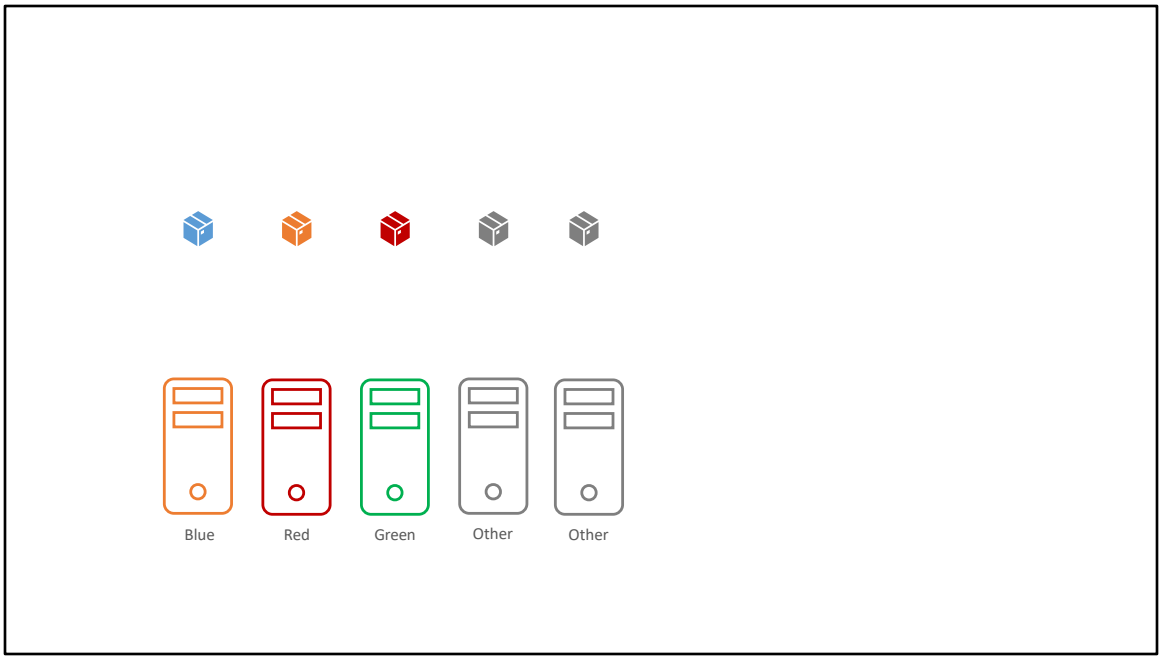
Node 1

The two new types expected in the future only have a difference in the During Execution phase. A new option called RequiredDuringExecution is introduced which will evict any PODs that are running on Nodes that do not meet affinity rules. In the earlier example, if a POD running on the large node, will be evicted if the label Large is removed from the Node.

Node Affinity
vs
Taints and
Tolerations

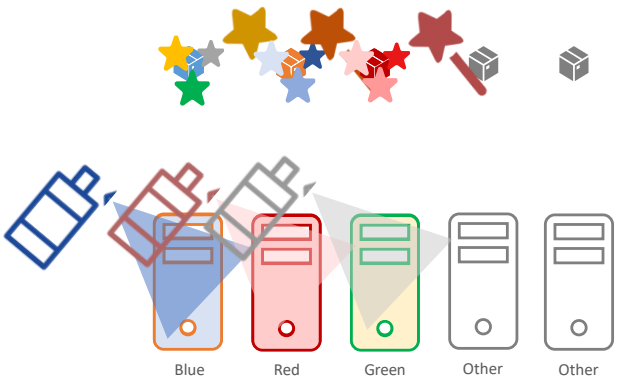


Hello and welcome to this lecture. Now that we have learned about Taints and Tolerations and Node Affinity, let us tie together the two concepts through a fun exercise.



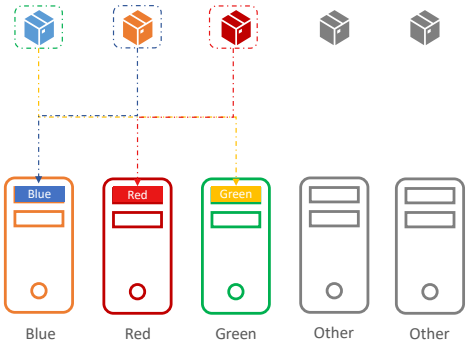
We have 3 Nodes and 3 PODs each in three colors – Blue, Red and Green. The ultimate aim is to place the blue POD in the blue Node, the red POD in the red node and likewise for green. We are sharing the same Kubernetes cluster with other teams, so there are other PODs in the cluster as well as other Nodes. We do not want any other POD to be placed on our Node, neither do we want our PODs to be placed on their nodes.

Taints and Tolerations



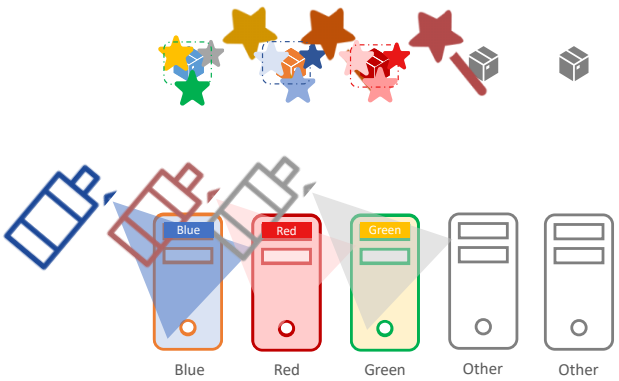
Let us first try to solve this problem using taints and tolerations. <click> We apply a Taint to the Nodes marking them with their colors. Blue, Red and Green, and <click> we then set a toleration on the PODs to tolerate the respective colors. When the PODs are now created, the NODES ensure they only accept the PODs with the right tolerations. So the Green node ends up on the Green Node and Blue Node ends up on the blue node. <click> However Taints and Tolerations does not guarantee that the PODs will only prefer these nodes, so the red node ends up on one of the other nodes that do not have a taint or a toleration set. This is not desired.

Node Affinity



Let us try to solve the same problem with Node Affinity. With Node Affinity, we first label the Nodes with their respective colors - blue, red and green. We then set Node Selectors on the PODs to tie the PODs to the Nodes. As such the PODs end up on the right nodes. However, that does not guarantee that other PODs are not placed on these nodes. This is not something we desired.

Taints/Tolerations and Node Affinity



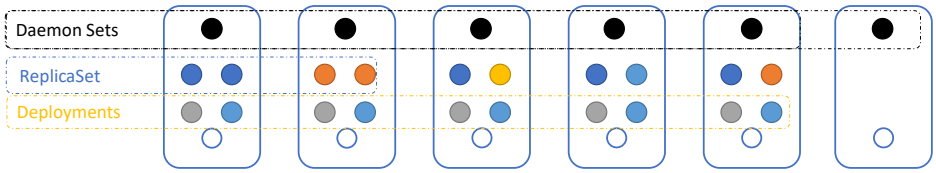
As such a combination of taints and tolerations and Node affinity rules can be used together to completely dedicate nodes for specific PODs.

We first use taints and tolerations to prevent other pods from being placed on our Nodes. And then we use node affinity to prevent our pods from being placed on their nodes.

Well that's it for this lecture. Head over to the coding exercises and play around with taints tolerations and node affinity.

Hello and welcome to this lecture. In this lecture we look at daemon Sets in Kubernetes.

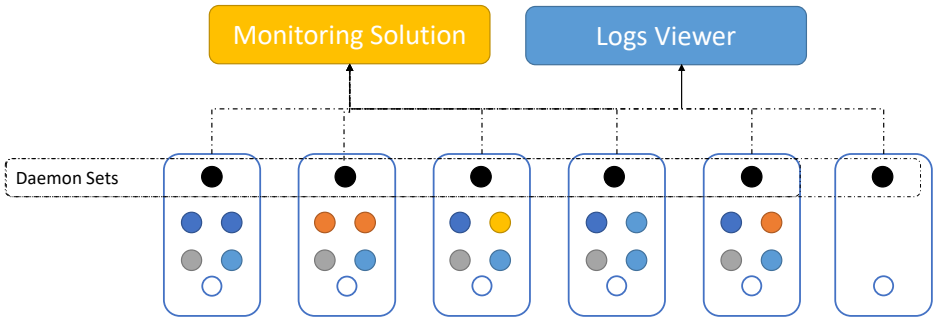
Daemon Sets



So far we have deployed various pods on different nodes in our cluster. With the help of replica sets and deployments we made sure multiple copies of our applications are made available across various different worker nodes.

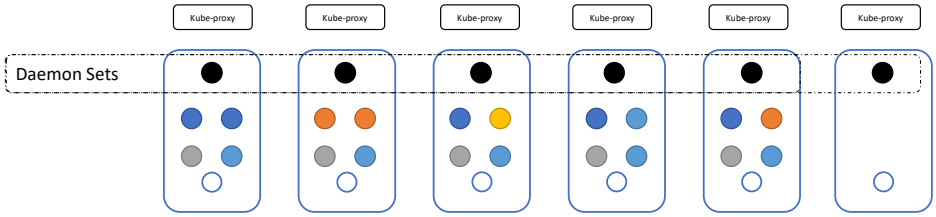
daemon set are like replica sets, as in it helps you deploy multiple instances of pod. But it runs one copy of your pod on each node in your cluster. <c> Whenever a new node is added to the cluster, a replica of the pod is automatically added to that node and when a node is removed the pod is removed. The daemon set ensures that one copy of the POD is always present on all nodes.

Daemon Sets – UseCase



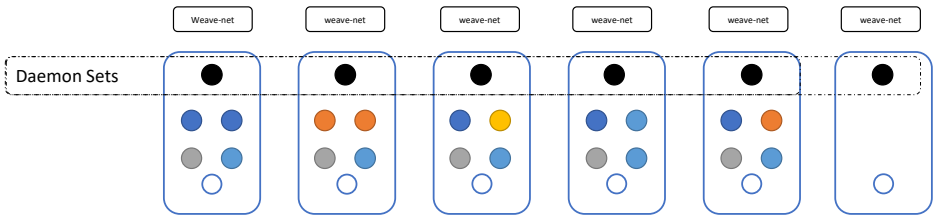
So what are some use cases of daemonsets? Say you would like to deploy a monitoring agent or log collector on each of your nodes in the cluster so you can monitor your cluster better. A daemon set is perfect for that as it can deploy your monitoring agent in the form of a POD in all the nodes in your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

Daemon Sets – UseCase – kube-proxy



Earlier while discussing the kubernetes architecture we learned that one of the worker node components that is required on every node in the cluster is a kube-proxy. That is one good use case of daemon Sets. The kubeproxy component can be deployed as a daemonSet in the cluster.

Daemon Sets – UseCase – Networking



Another use case is for networking. Networking solutions like weave-net requires an agent to be deployed on each node in the cluster. We will discuss about Networking concepts in much more detail later during this course, but I just wanted to point it out here for now.

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-definition.yaml
daemon-set Created
```

Creating a DaemonSet is similar to the ReplicaSet creation process. It has 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 apiVersion, kind, metadata and spec. The api version is apps/v1. 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. Ensure the labels in the selector matches the ones in the pod template.

Once ready create the daemonset using the kubectl create daemonset command.

View DaemonSets

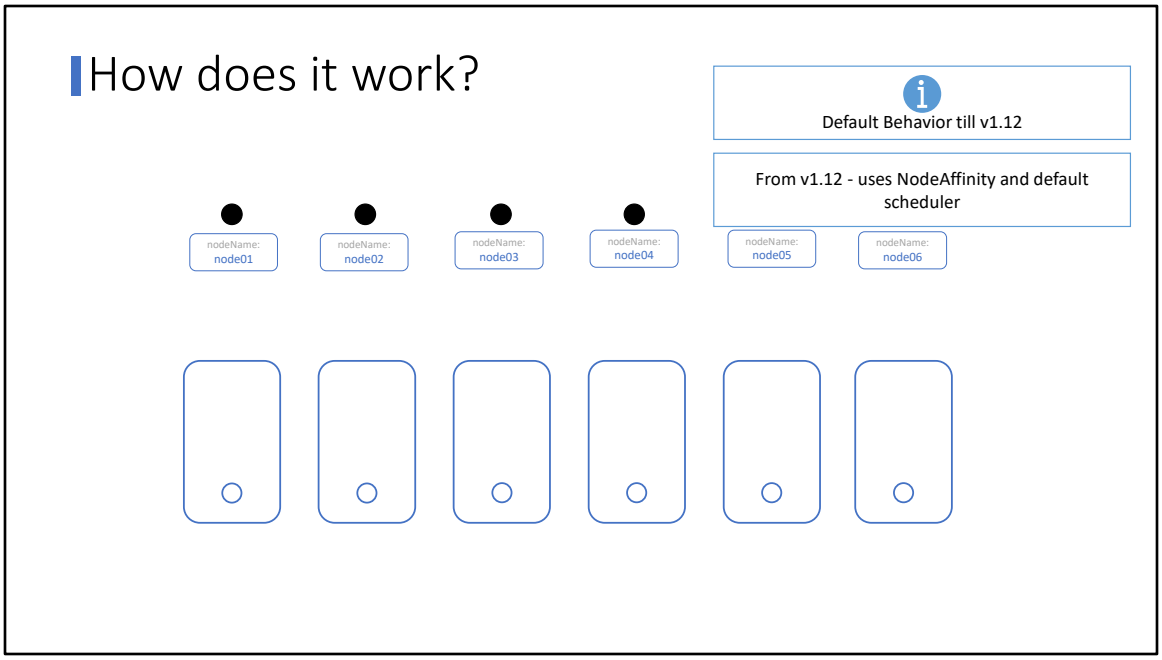
```
kubectl get daemonsets
```

```
       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:
```

To view the created daemonset run the kubectl get daemonset command. And of course to view more details run the kubectl describe command.



So how does a Daemon set work? How does it schedule pods on each node? If you were asked to schedule a pod on each node in the cluster how would you do it?

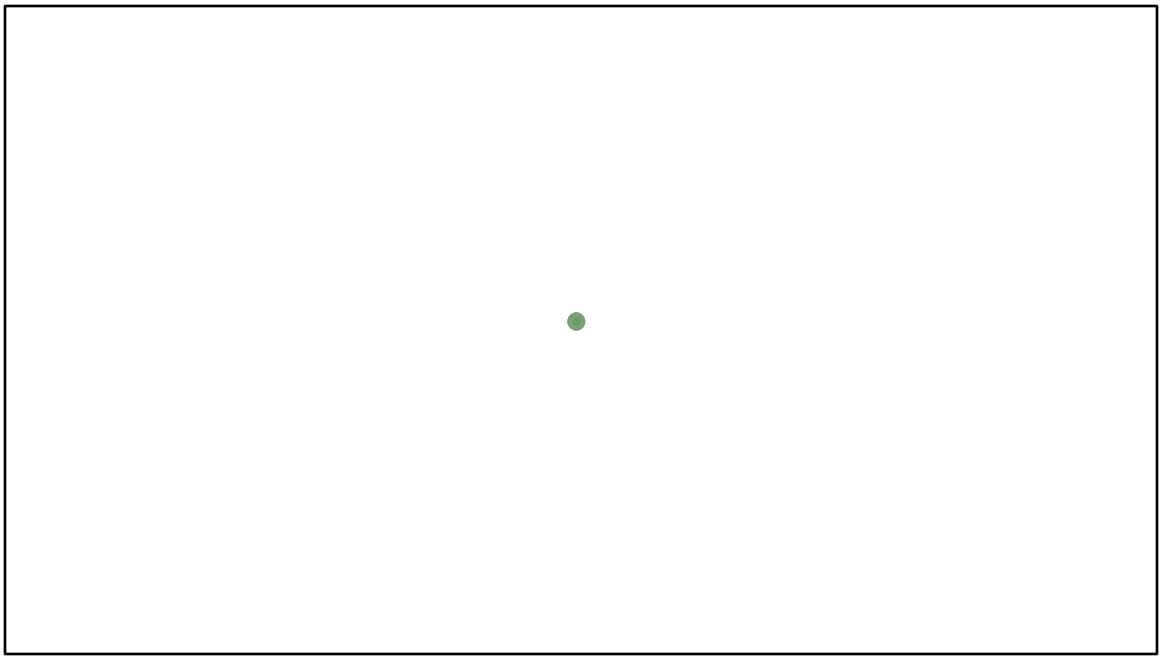
In one of the previous lectures we discussed that we could set the nodeName property on the pod to bypass the scheduler and get the pod placed on a node directly. So that's one approach. <c> On each pod, set the nodeName property in its specification before it is created and when they are created, they automatically land on the respective nodes.
So that's how it used to be until kubernetes version v1.12.

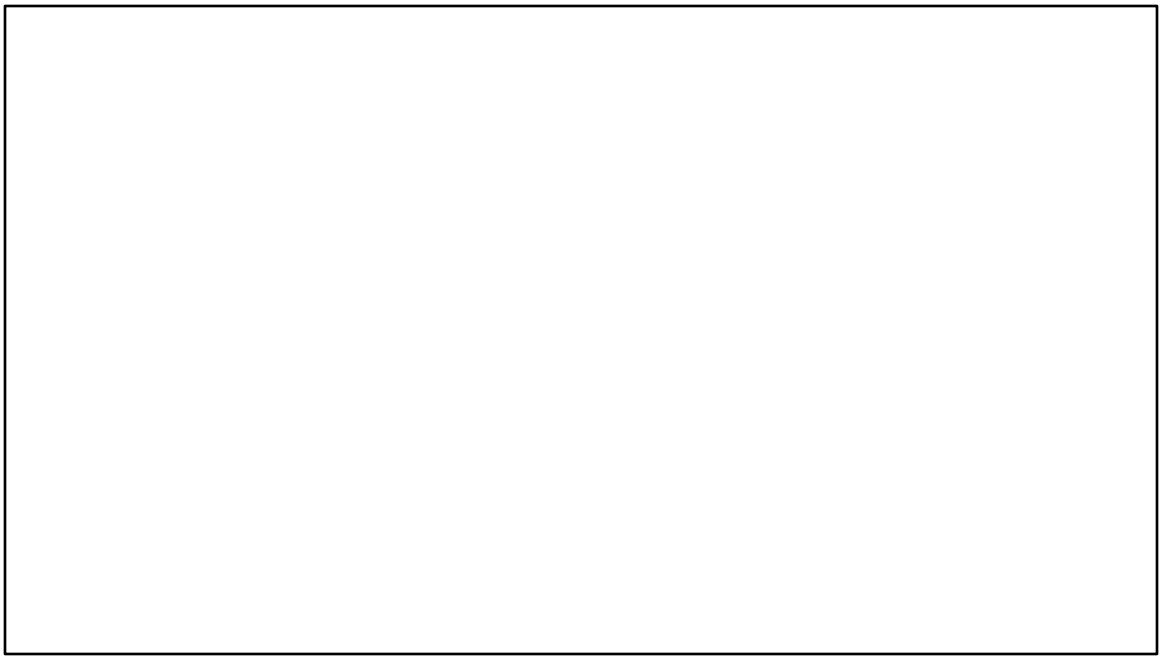
<c> From v1.12 onwards the Daemon set uses the default scheduler and node affinity rules that we learned in one of the previous lectures to schedule a pod on each node.

Practice Test

Head over to the practice test and practice working with DaemonSets.

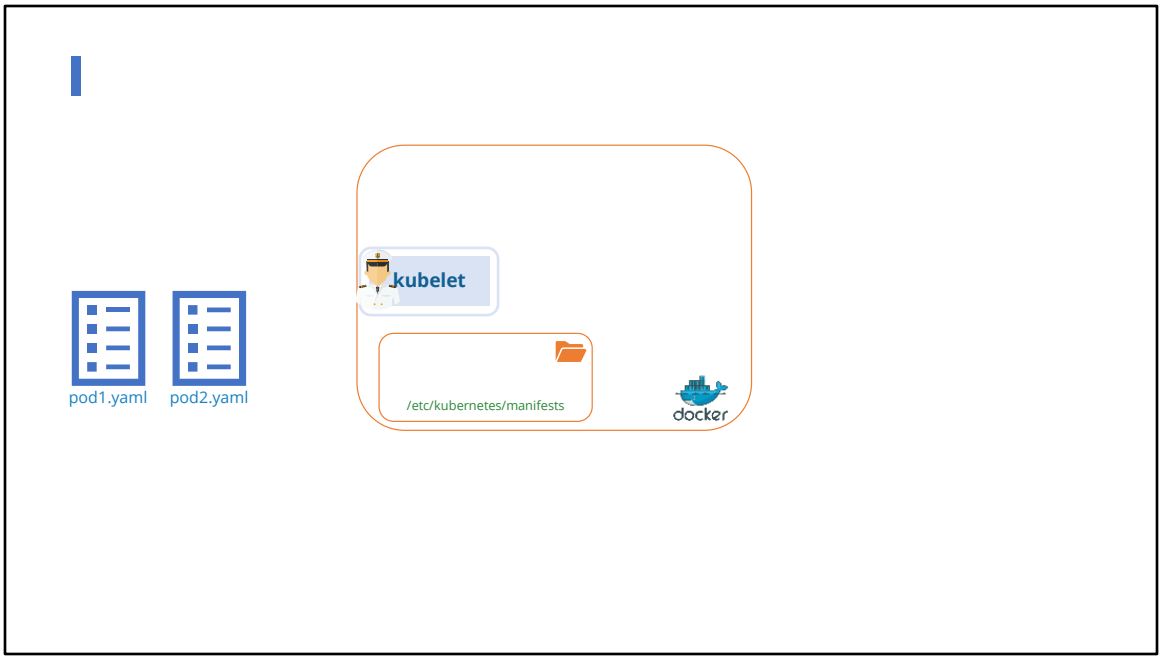
https://katacoda.com/mmumshad2/scenarios/kubernetes-cka-scheduler-daemonset





Hello and welcome to this lecture. In this lecture we discuss about Static Pods in Kubernetes.

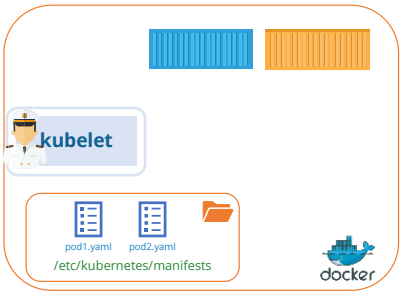
Earlier in this course we talked about the Architecture and how the kubelet functions as one of the many control plane components in Kubernetes. The kubelet relies on the kube-apiserver for instructions on what PODs to load on its node, which was based on a decision made by the kube-scheduler which was stored in the ETCD datastore. What if there was <c> no kube-apiserver, and <c> kube-scheduler and no controllers and no ETCD cluster. <c> What if there was no master at all. What if there were no other nodes. <c> What if you are all alone in the sea by yourself. Not part of any cluster. Is there anything that the kubelet can do as the captain on the ship? Can it operate as an independent node? If so who would provide the instructions required to create PODs?



Well, the kubelet can manage a node independently. On this ship…err..host, we have the kubelet installed. And of course we have Docker as well to run containers. There is no Kubernetes cluster. The one thing that the kubelet knows to do is create PODs. But we don't have an API server here to provide the POD details. <c> By now we know that to create a POD you need the details of the POD in a POD definition file. But how do you provide a pod definition file to the kubelet without a kube-api server?

You can configure the kubelet to read the pod definition files from a <c> directory on the server designated to store information about PODs. Place the pod definition files in this directory.

Static PODs



kubelet.service

```
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
```

The Kubelet periodically checks this directory for files, reads these files and <c> creates PODs on the host. Not only does it create the POD, it can 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 file within this directory, the kubelet recreates the pod for those changes to take affect. If you remove a file from this folder, the pod is deleted automatically.

So these PODs that are created by the kubelet on its own without the intervention from the API server or rest of the kuberentes cluster components are known as Static PODs.

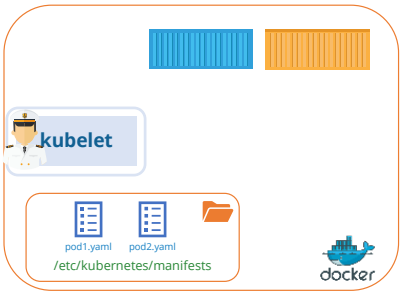
Remember you can only create PODs this way. You cannot create replicasets or deployments or services by placing a definition file in the designated directory. They are all concepts part of the whole Kubernetes architecture, that requires other control plane components like the replication and deployment controllers. The kubelet works at a POD level and can only understand PODs. Which is why it is able to create static pods this way.

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 in to the kubelet as a option while running the service. The option is named pod manifest path and here it is set to /etc/Kubernetes/manifests.

An alternate way to configure this is to use a separate YAML or JSON file as input to the kubelet.

Static PODs



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
```

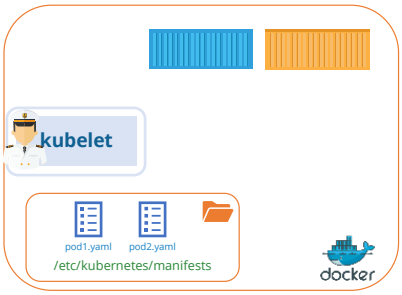
kubeconfig.yaml

```
staticPodPath: /etc/kubernetes/manifes
```

There is 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 staticPodPath in that file. Clusters setup by the kubeadmin tool uses this approach.

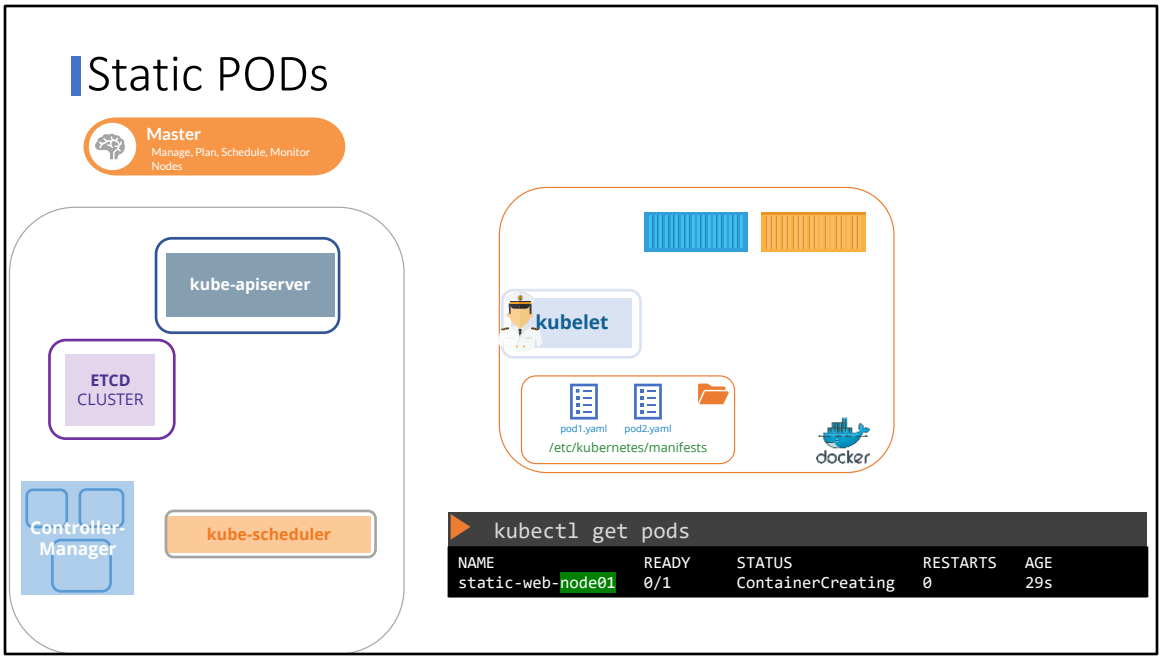
If you are inspecting an existing cluster, you should inspect this option of the kubelet to identify the path to the directory. You will then know where to place the definition file for your static pods. So keep this in mind when you go through the labs. You should know to view and configure this option irrespective of the method used to setup the cluster.

Static PODs




```
docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
PORTS               NAMES
8e5d4c4db7b6        busybox                "sh -c 'echo Hello K…"   20 seconds ago      Up 20 seconds
k8s_myapp-container_myapp-pod-host01_default_48e37fb432f2e06350e76786bd0bac66_0
f6737e1149cb        k8s.gcr.io/pause:3.1   "/pause"                 24 seconds ago      Up 23 seconds
k8s_POD_myapp-pod-host01_default_48e37fb432f2e06350e76786bd0bac66_0
```

Once the static pods are created, you can view them by running the Docker ps command. So why not the kubectl command as we have been doing so far? Remember we don't have the rest of the Kubernetes cluster. Kubectl utility works with the kube-apiserver. Since we don't have an API server now, no kubectl utility.

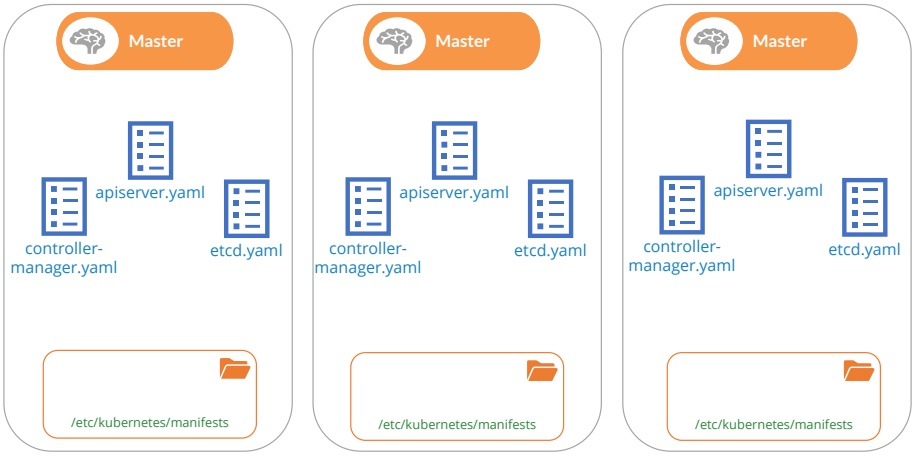


So then, how does it work when the node is part of a cluster? When there IS an API server requesting the kubelet to create PODs? Can the kubelet create both kinds of PODs at the same time?

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-apiserver provides input to kubelet. The kubelet can create both kinds of PODs – the staticpods and the ones from the api server - at the same time.

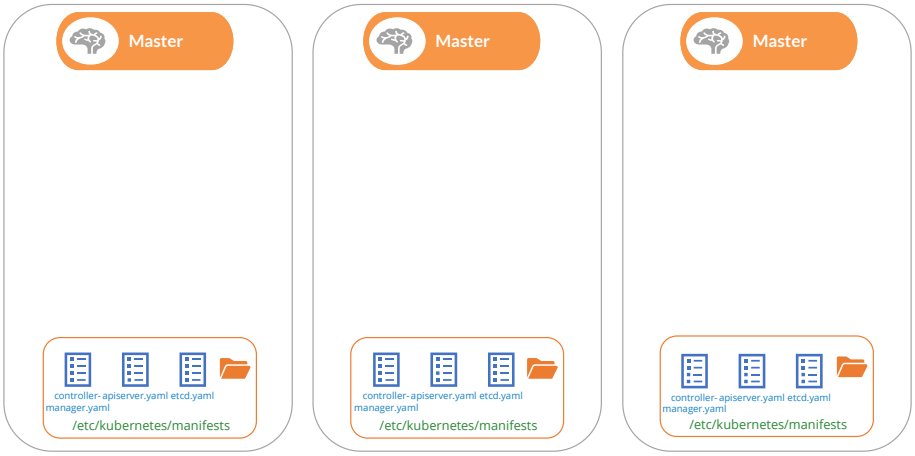
Well in that case is the API server aware of the static pods created by the kubelet? Yes it is. If you run the <c> kubectl get pods command on the master node, the static pods will be listed as any other pod. How is that happening? When the kubelet creates a static pod, if it is part of a cluster, it also creates a mirror object in the kubeapi server. What you see from the kube-apiserver 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 nodes manifest folder. <c> Note that the name of the pod is automatically appended with the node name. In this case node01.

Use Case



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. Start by installing kubelet on all the master nodes. Then create pod definition files that uses docker images of the various control plane components such as the api server, controller, etcd etc.

Use Case



Place the definition files in the designated manifests folder. And kubelet takes care of deploy 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 service crashing. If any of these services where to crash, since it's a static POD, it will be automatically restarted by the kubelet. Neat and simple. That's how the kubeadmin tool set's up a Kubernetes cluster.

Use Case

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

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-78fcdf6894-hwrq9	1/1	Running	0	16m
kube-system	coredns-78fcdf6894-rzhjr	1/1	Running	0	16m
kube-system	etcd-master	1/1	Running	0	15m
kube-system	kube-apiserver-master	1/1	Running	0	15m
kube-system	kube-controller-manager-master	1/1	Running	0	15m
kube-system	kube-proxy-lzt6f	1/1	Running	0	16m
kube-system	kube-proxy-zm5qd	1/1	Running	0	16m
kube-system	kube-scheduler-master	1/1	Running	0	15m
kube-system	weave-net-29z42	2/2	Running	1	16m
kube-system	weave-net-snmdl	2/2	Running	1	16m

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 kubeadmin tool. We will explore that setup in the upcoming practice test.

Static PODs vs DaemonSets

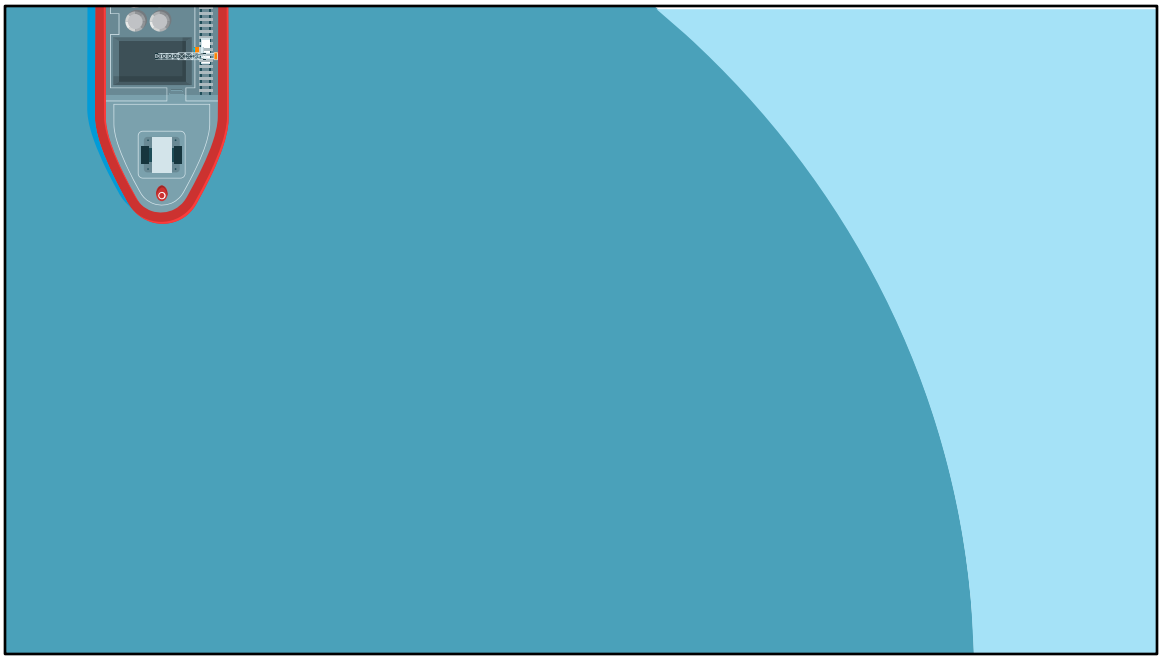
Static PODs	DaemonSets
Created by the Kubelet	Created by Kube-API server (DaemonSet Controller)
Deploy Control Plane components as Static Pods	Deploy Monitoring Agents, Logging Agents on nodes
Ignored by the Kube-Scheduler	

Before I let you go, one question that I get often is about the different between Static PODs and DaemonSets. DaemonSets as we saw earlier are used to ensure one instance of an application is available on all nodes in the cluster. It is handled by a daemonset controller through the kube-api server.

Whereas static pods, as we saw in this lecture, are created directly by the kubelet without any interference from the kube-api server or rest of the Kubernetes control plane components. Static pods can be used to deploy the Kubernetes control plane components itself. Both static pods and pods created by daemonsets are ignored by the kube-scheduler. The kube-scheduler has no affect on these pods.

Well that's it for this lecture. Head over to the practice test and practice working with Static Pods.

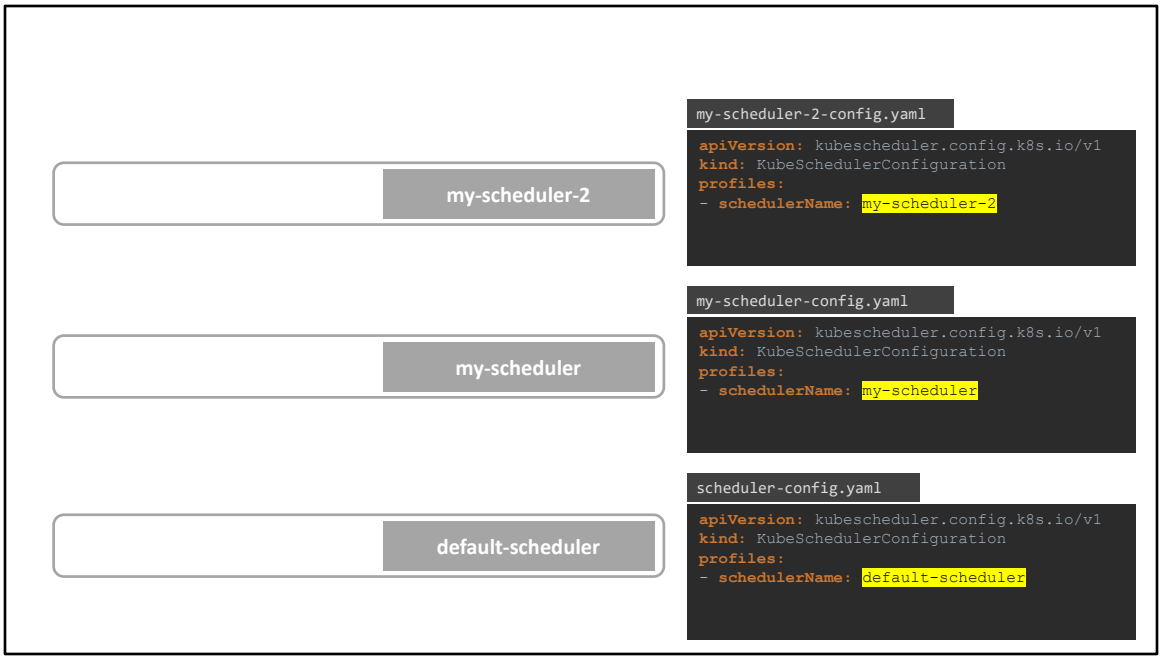
Hello and welcome to this lecture. In this lecture we look at deploying multiple schedulers in a Kubernetes cluster.



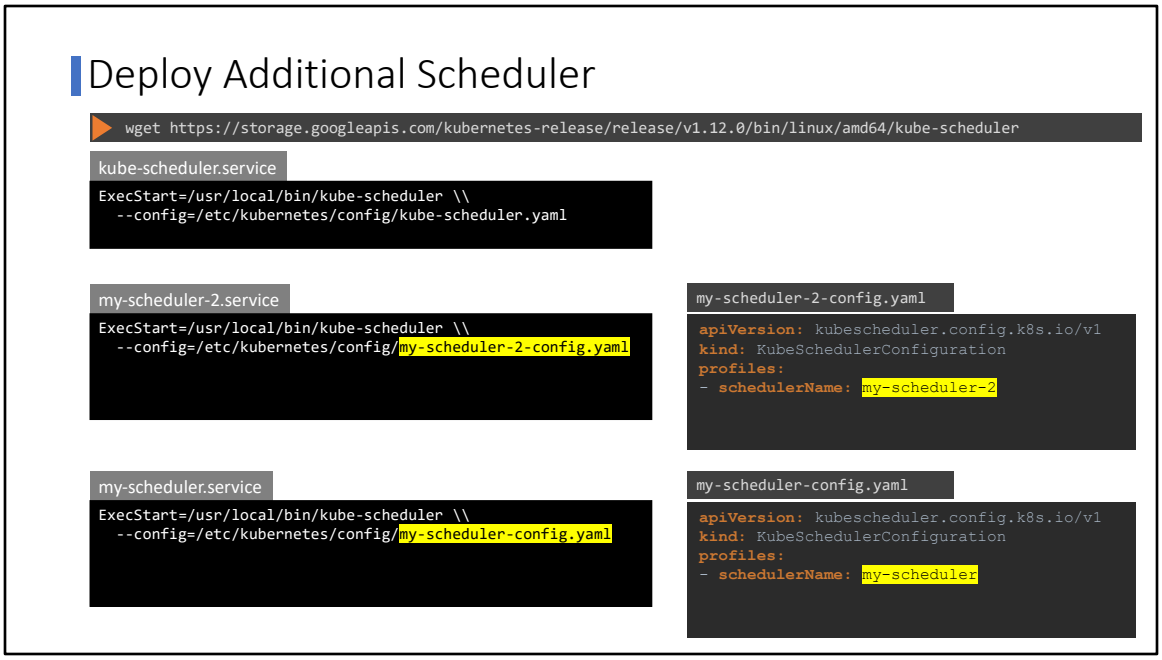
We have seen how the default-scheduler works in kubernetes in the previous lectures. It has an algorithm that distributes pods across nodes evenly as well as takes into consideration various conditions we specify through taints & tolerations and node affinity etc. But what if none of these satisfies your needs? Say you have a specific application that requires its components to be placed on nodes after performing some additional checks.

So you decide to have your own scheduling algorithm to place pods on nodes? So that you can add your own custom conditions and checks in it. Kubernetes is highly extensible. You can write your own kubernetes scheduler program, package it and deploy it as the default scheduler or as an additional scheduler in the kubernetes cluster. That way all of the other applications can go through the default scheduler, however one specific application can use your custom scheduler. Your kubernetes cluster can have multiple schedulers at the same time.

When creating a POD or a Deployment you can instruct kubernetes to have the POD scheduled by a specific scheduler.



When there are multiple schedulers they must have different names so we can identify them as separate schedulers. The default scheduler is named default-scheduler. This name is configured in a KubeSchedulerConfiguration file that looks like this. The default scheduler doesn't really need one because if you didn't specify a name it sets the name to default-scheduler, but this is how it would look if you created one. For the other schedulers we create a separate configuration file and set the scheduler name like this.



So let's start with the most simplest way of deploying an additional scheduler.

Earlier we saw how to deploy the kube-scheduler. We download the kube-scheduler binary and run it as a service with a set of options.

To deploy an additional scheduler, <c> you may use the same kube-scheduler binary or use one that you might have built for yourself, which is what you would do if you needed the scheduler to work differently. In this case we are going to use the same binary to deploy the additional scheduler. This time we point the configuration to the custom configuration file we created. So each scheduler uses a separate configuration file and with each file having it's own schedulerName.

Note that there are other options to be passed in such as the kubeconfig file to authenticate into the kubernetes API, but I am just skipping that for now to keep it simple.

This is not how you would deploy a custom scheduler 99% of the time today because with kubeadm deployment all the control plane components run as a pod or deployment within the kubernetes cluster. So let's check out how that's done.

Deploy Additional Scheduler as a Pod

my-custom-scheduler.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-custom-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --config=/etc/kubernetes/my-scheduler-config.yaml
    image: k8s.gcr.io/kube-scheduler-amd64:v1.11.3
    name: kube-scheduler
```

my-scheduler-config.yaml

```yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
- schedulerName: my-scheduler
leaderElection:
  leaderElect: true
  resourceNamespace: kube-system
  resourceName: lock-object-my-scheduler
```

Let’s take a look at how it works if you were to deploy the scheduler as a pod. <c> <c>
We create a pod definition file and specify the kube-config property which is the path to the scheduler.conf file that has the authentication information to connect to kubernetes API server. We then pass in our custom kubeschedulerconfiguration file as a config to the scheduler. Note that we have the schedulerName specified in this file and that's how the name gets picked up.

Finally an important option to look here is the leader-elect option. The leader-elect option is used when you have multiple copies of the scheduler running on different master nodes, as in a High Availability setup where you have multiple master nodes with the kube-scheduler process running on both of them. If multiple copies of the same scheduler are running on different nodes, only one can be active at a time. That’s where the leader-elect option helps in choosing a leader who will lead scheduling activities. We will discuss more about HA setup in another section. In case you do have multiple masters, you can pass in an additional parameter <c> to set a lock object name. This is to differentiate the new custom scheduler from the default during the leader election process.

Once done, create the pod using the kubectl create command.

Deploy Additional Scheduler as a Deployment

Let's take a look at how it works with the kubeadm tool. <c> The kubeadm tool deploys the scheduler as a POD. You can find the definition file it uses under the manifests folder. I have removed all the other details from the file so we can focus on the key parts. The command section has the command and associated options to start the scheduler.

<c>
We can create a custom scheduler by making a copy of the same file, and by changing the name of the scheduler. We set the name of the pod to my-custom-scheduler and we specify .

Finally an important option to look here is the leader-elect option. The leader-elect option is used when you have multiple copies of the scheduler running on different master nodes, as in a High Availability setup where you have multiple master nodes with the kube-scheduler process running on both of them. If multiple copies of the same scheduler are running on different nodes, only one can be active at a time. That's where the leader-elect option helps in choosing a leader who will lead scheduling activities. We will discuss more about HA setup in another section. In case you do have multiple masters, you can pass in an additional parameter <c> to set a

lock object name. This is to differentiate the new custom scheduler from the default during the leader election process.

Once done, create the pod using the kubectl create command.

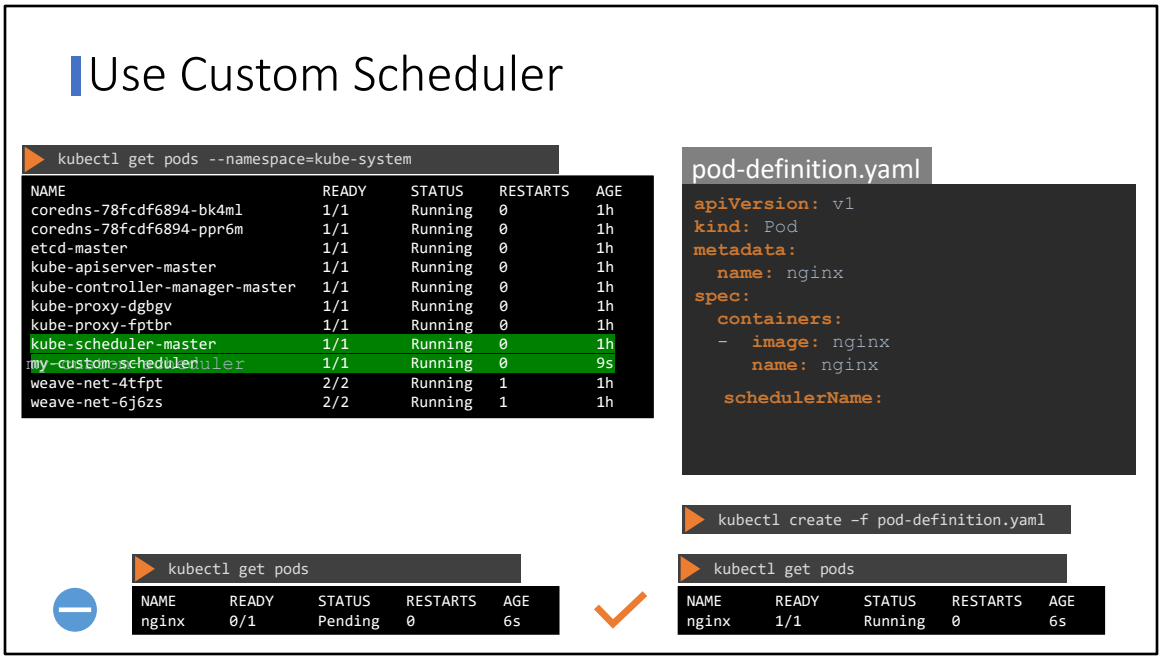
View Schedulers

```
kubectl get pods --namespace=kube-system
```

```
NAME                              READY     STATUS    RESTARTS   AGE
coredns-78fcdf6894-bk4ml          1/1       Running   0          1h
coredns-78fcdf6894-ppr6m          1/1       Running   0          1h
etcd-master                       1/1       Running   0          1h
kube-apiserver-master             1/1       Running   0          1h
kube-controller-manager-master    1/1       Running   0          1h
kube-proxy-dgbgv                  1/1       Running   0          1h
kube-proxy-fptbr                  1/1       Running   0          1h
kube-scheduler-master             1/1       Running   0          1h
my-custom-scheduler               1/1       Running   0          9s
weave-net-4tfpt                   2/2       Running   1          1h
weave-net-6j6zs                   2/2       Running   1          1h
```

Run the get pods command in the kube-system namespace and look for the new custom scheduler. Make sure its in a running state.

Use Custom Scheduler



The next step is to configure a new POD or a deployment to use the new scheduler. In the Pod specification add a new field called schedulerName and specify the name of the new scheduler. This way when the pod is created, the right scheduler picks it up to schedule.

Create the pod using the kubectl create command. If the scheduler was not configured correctly, then the pod will continue to remain in a Pending state. If everything is good, then the pod will be in a Running state.

View Events

```
kubectl get events –o wide
```

LAST SEEN	COUNT	NAME	KIND	TYPE	REASON	SOURCE	MESSAGE
9s	1	nginx.15	Pod	Normal	Scheduled	my-custom-scheduler	Successfully assigned default/nginx to node01
8s	1	nginx.15	Pod	Normal	Pulling	kubelet, node01	pulling image "nginx"
2s	1	nginx.15	Pod	Normal	Pulled	kubelet, node01	Successfully pulled image "nginx"
2s	1	nginx.15	Pod	Normal	Created	kubelet, node01	Created container
2s	1	nginx.15	Pod	Normal	Started	kubelet, node01	Started container

So how do you know which scheduler picked it up? View the events using the kubectl get events with the –o wide option command. This lists all the events in the current namespace. Look for the Scheduled events. As you can see the source of the event is the custom scheduler we created. And the message says successfully assigned default/nginx image.

View Scheduler Logs

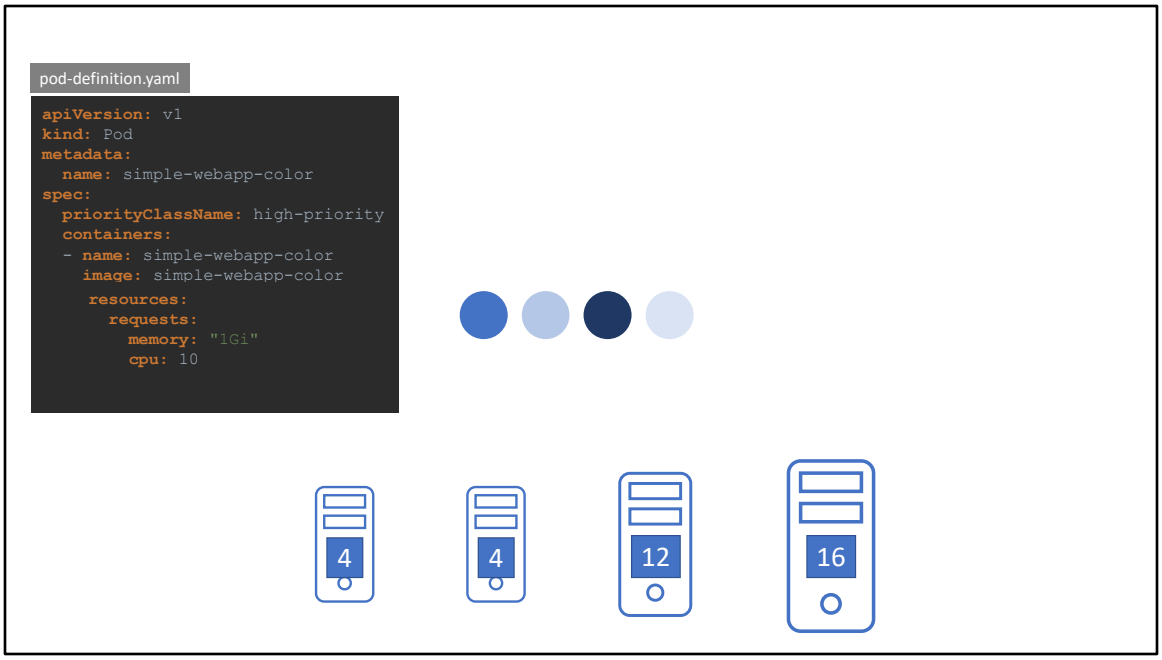
```
kubectl logs my-custom-scheduler --name-space=kube-system
```

```
I0204 09:42:25.819338       1 server.go:126] Version: v1.11.3
W0204 09:42:25.822720       1 authorization.go:47] Authorization is disabled
W0204 09:42:25.822745       1 authentication.go:55] Authentication is disabled
I0204 09:42:25.822801       1 insecure_serving.go:47] Serving healthz insecurely on 127.0.0.1:10251
I0204 09:45:14.725407       1 controller_utils.go:1025] Waiting for caches to sync for scheduler controller
I0204 09:45:14.825634       1 controller_utils.go:1032] Caches are synced for scheduler controller
I0204 09:45:14.825814       1 leaderelection.go:185] attempting to acquire leader lease  kube-system/my-custom-scheduler...
I0204 09:45:14.834953       1 leaderelection.go:194] successfully acquired lease kube-system/my-custom-scheduler
```

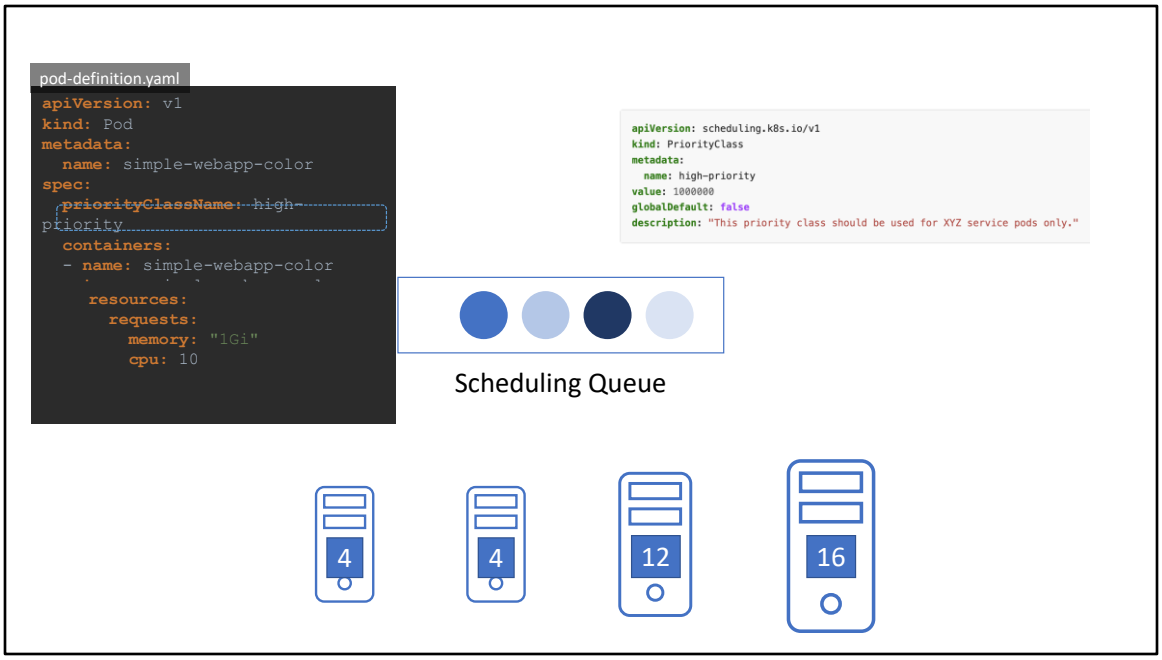
To view the logs of the scheduler, view the logs of the pod using the kubectl logs command. Specify the name of the scheduler and the correct namespace.

Well, that's it for this lecture. Head over to the practice exercises and practice deploying multiple schedulers.

Let us now look at what Scheduler Profiles are.



Let's first recap how the Kubernetes scheduler works using this example of scheduling a pod to one of these 4 Nodes. Here is our Pod definition file and there's our Pod. It is waiting to be scheduled on one of these nodes. It has a resource requirement of 10 CPU. and you can see the available CPU on all of these nodes. Now it is not alone, there are some others pods that are waiting to be scheduled as well.

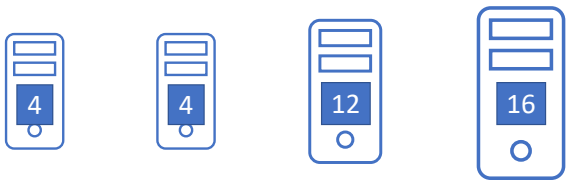


First the pods end up in the scheduling queue. This is where the pods wait to be scheduled. At this stage Pods are sorted based on the priority defined on the pods. In this case our Pod has a high-priority set. For this a priorityClass needs to be created first with the same name and a priority value set for it. In this case it's set to 1 million. So really high priority.

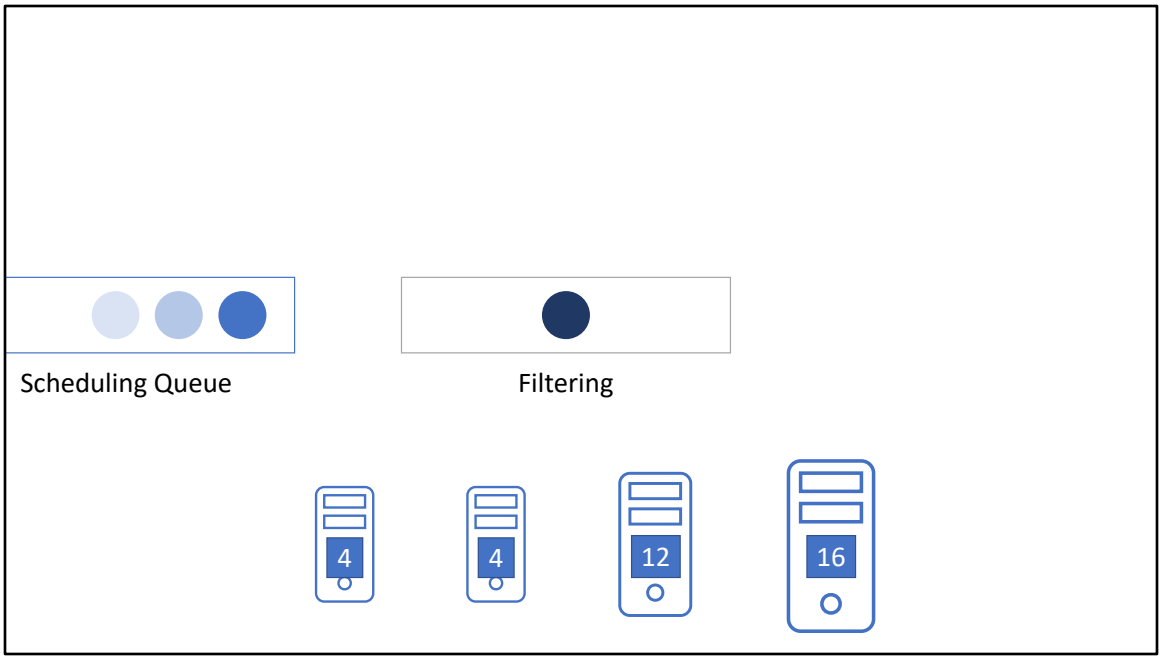
pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  priorityClassName: high-priority
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    resources:
      requests:
        memory: "1Gi"
        cpu: 10
```

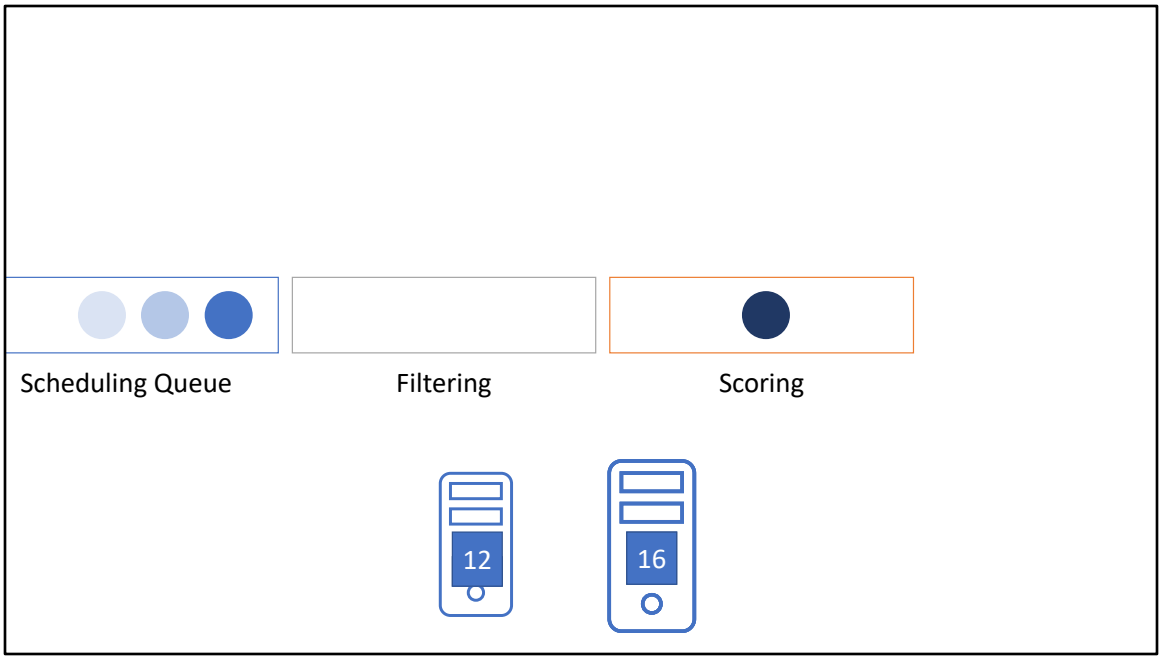
Scheduling Queue



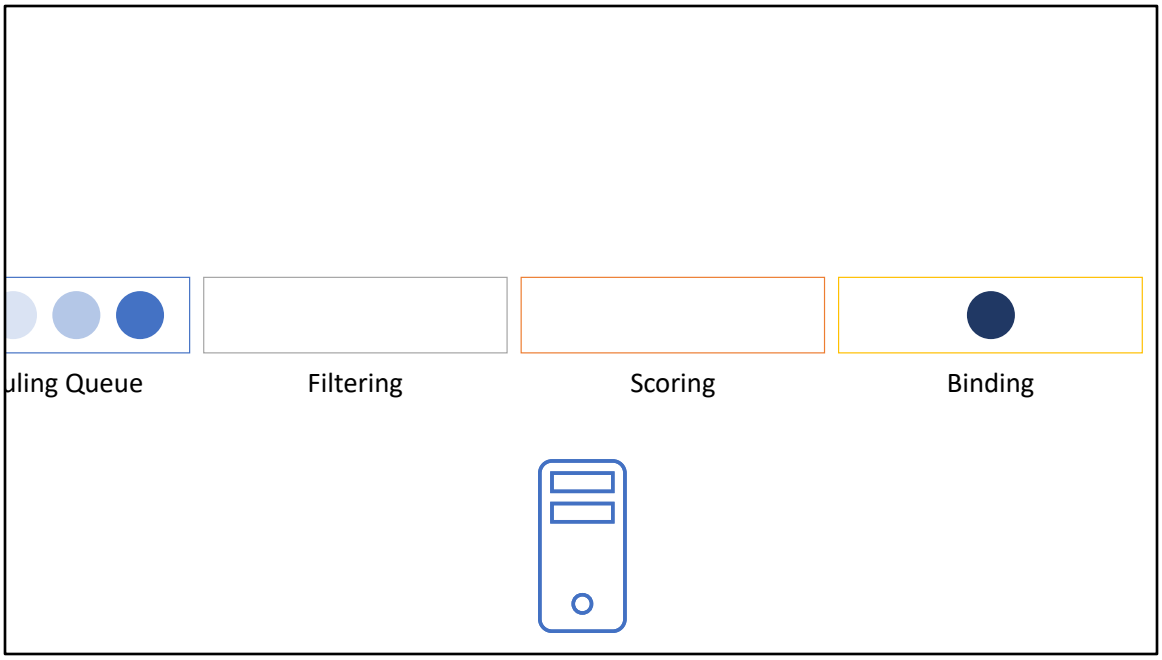
This is how pods with a higher priority gets to the beginning of the queue to be scheduled first.



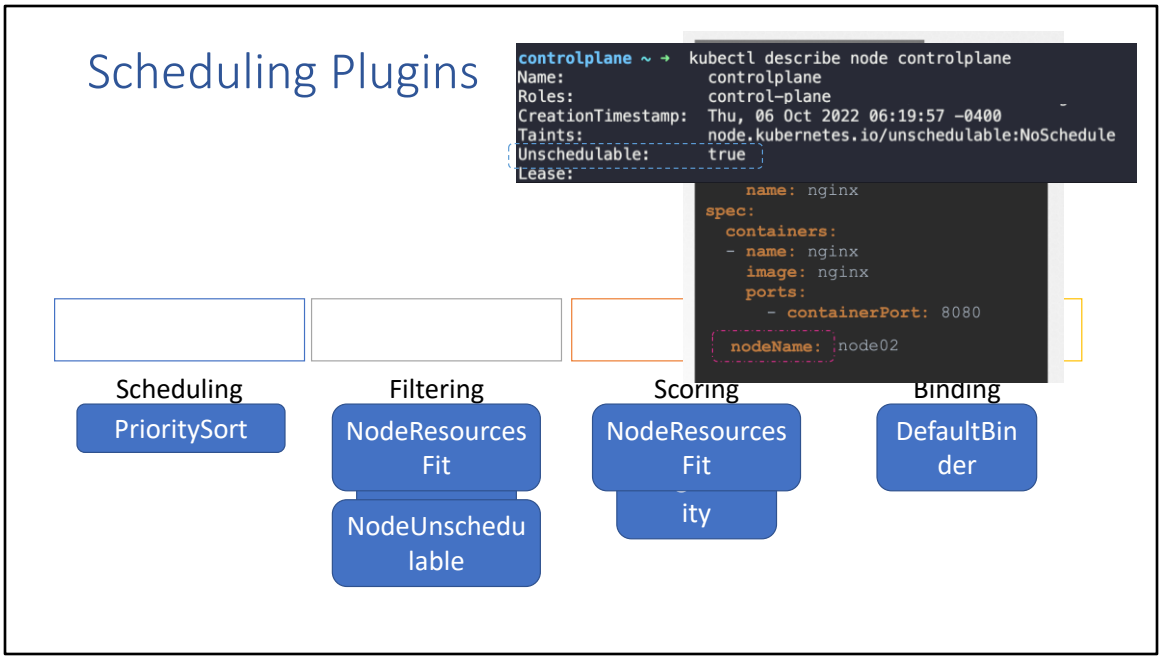
Then our pod enters the Filter phase. This is where nodes that cannot run the pod are filtered out. So in our case the first two nodes are out.



Then it enters the Scoring phase. This is where nodes are scored with different weights. From the two remaining nodes the scheduler associates a score to each node based on the free space that it will have after reserving the CPU required for that pod. In this case the first one has 2 left and second node has 6. So the second node gets a higher score.



And finally the binding phase. This is where the pod is finally bound to a node with the highest score.



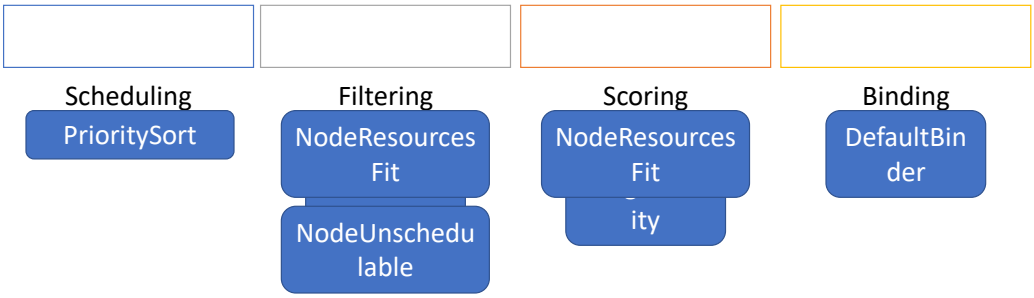
Now all of these operations are achieved with certain plugins. For example while in the scheduling queue, it's the prioritysort plugin that sorts the pods in an order based on the priority configured on the Pods. This is how pods with a priority class get's higher priority over other pods when scheduling. In the filtering stage it's the NodeResourcesFit plugin that identifies the nodes that has sufficient resources required by the pods and filters out the nodes that doesn't. Some other plugin examples are the nodename plugin that checks if a pod has a nodename mentioned in the pod spec and filters out all nodes that does not match this name. Another example is the nodeUnschedulable plugin that filters out nodes that has the unschedulable flag set to true.

In the scoring phase, again the NodeResorucesFit plugin associates a score to each node based on the resource available on it after the pod is allocated to it. So, as you can see a single plugin can be associated in multiple different phases. Another example of a plugin in this stage would be the ImageLocality plugin that associates a high score to the nodes that already has the container image used by the pods among the different nodes.

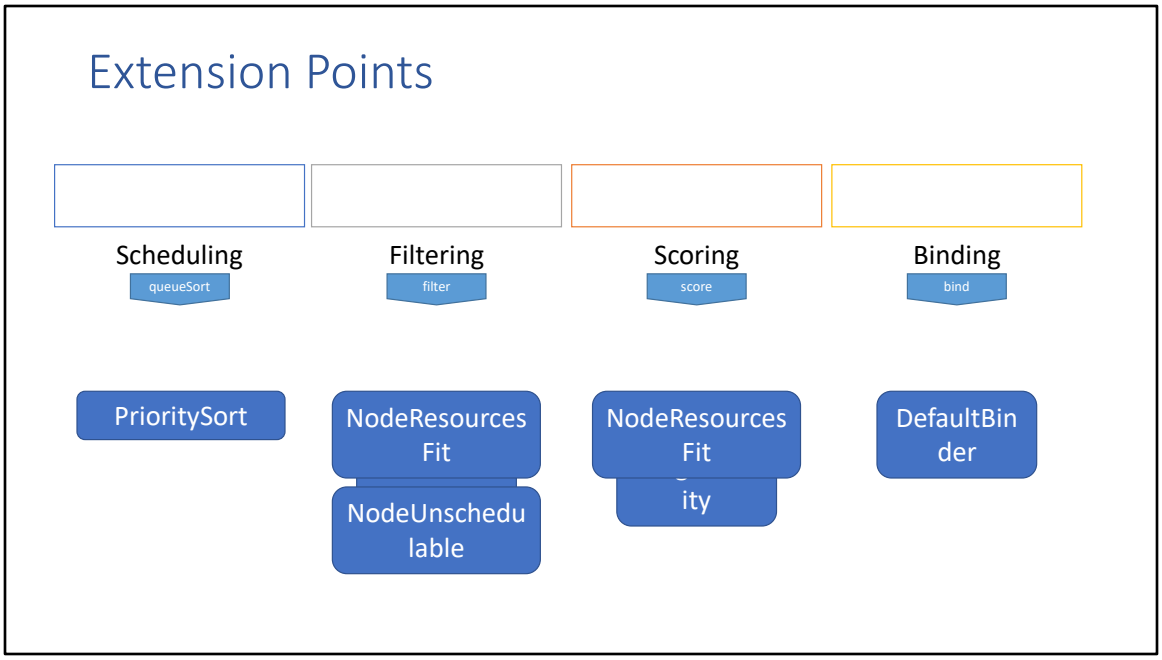
And finally in the binding phase you have the default binder plugin that provides

binding mechanism.

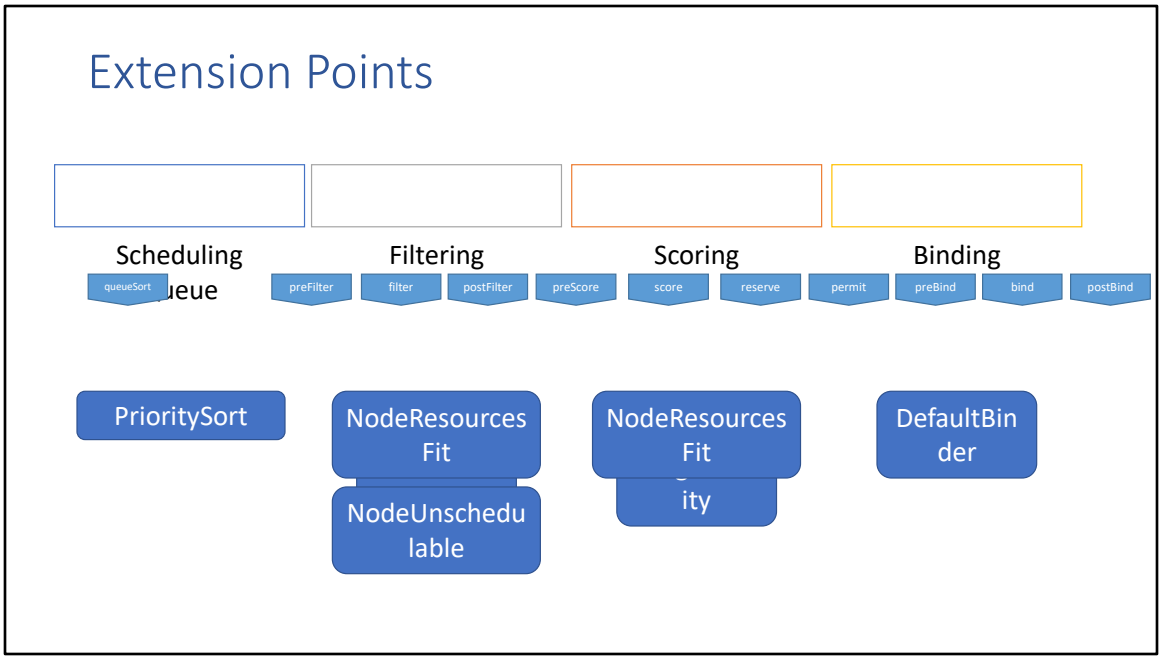
Scheduling Plugins



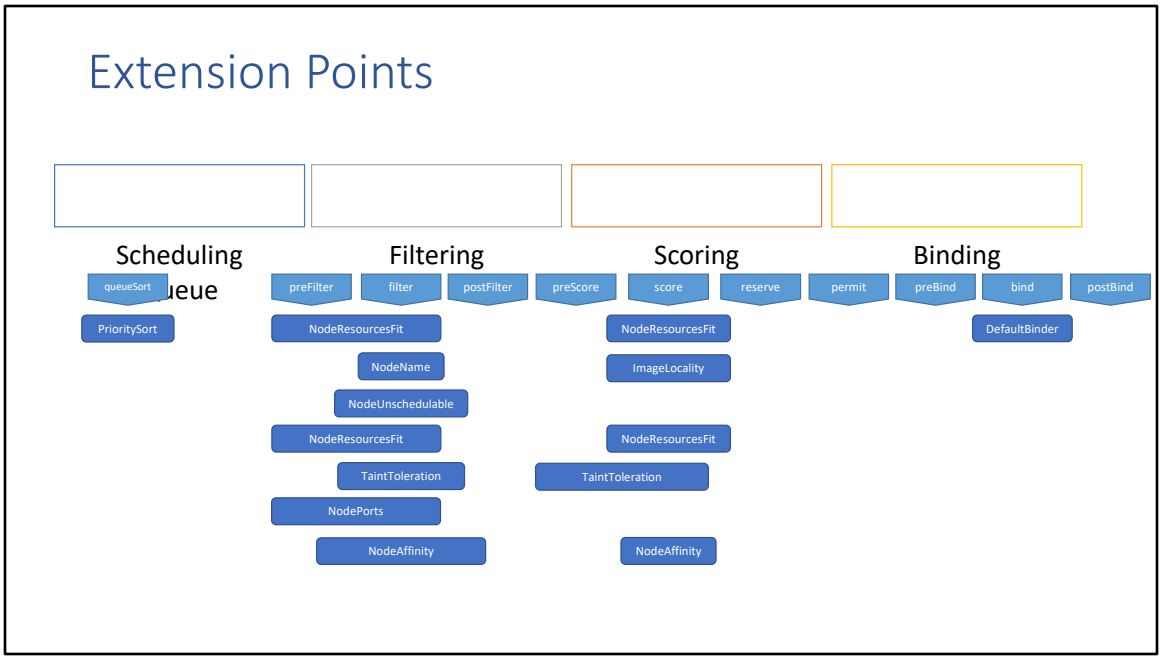
Now the highly extensible nature of Kubernetes makes it possible for us to customize what plugins go where. Meaning you can write your own plugin and plug them in here. And that is achieved



with the help of extension points. At each stage there is an extension point to which a plugin can be plugged to. In the scheduling queue we have a queueSort extension to which the priority sort plugin is plugged to. And then we have the filter extension, the score and the bind extension to which each of these plugins are plugged to.



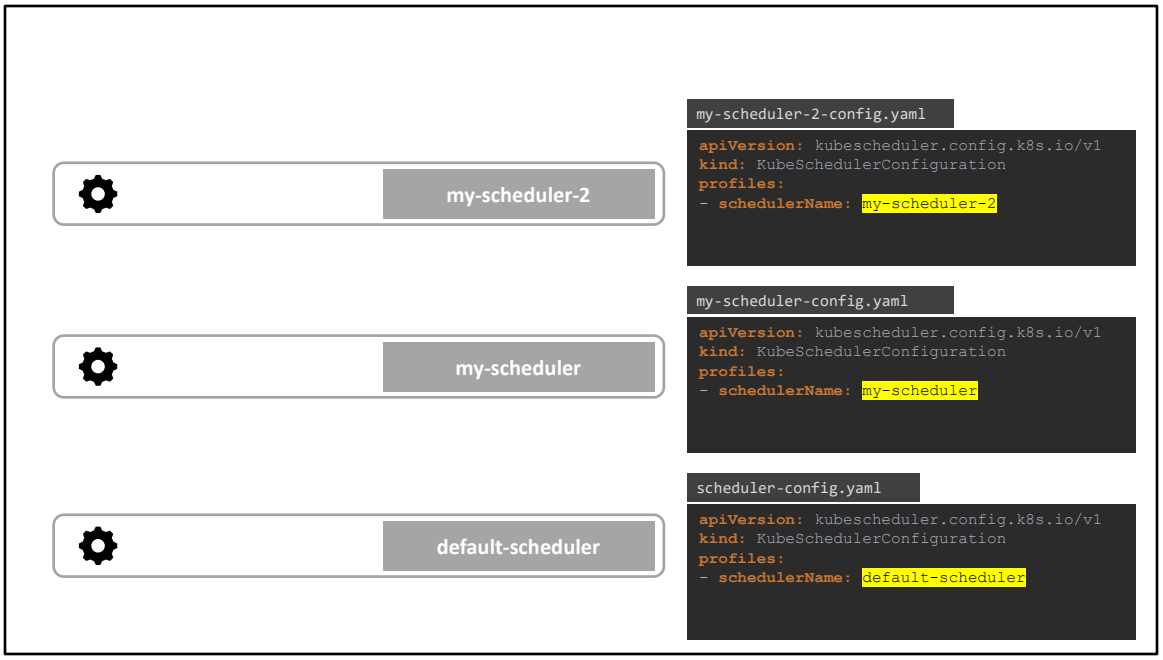
As a matter of fact there's more. There are extensions before entering the filter phase called the preFilter extension and after the filter phase called postFilter, and then there are preScore before the score extension point and reserve after the extension point. There is permit and preBind before bind and postBind after the binding phase. So you can get your own custom code to run anywhere in these points.



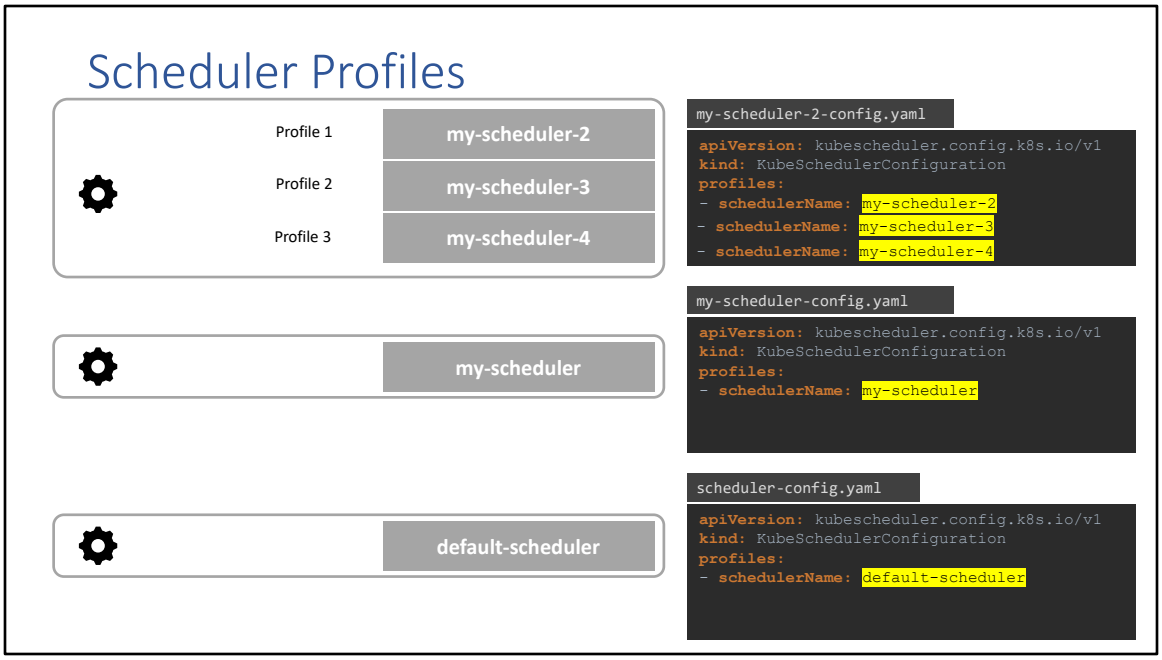
And here are some additional plugins that come by default that are associated with different extension points.

Now that's what scheduling plugins and extension points are. The highly extensible nature of Kubernetes allows us to customize the way these plugins are called and write our own scheduling plugin if needed.

Having learned that, let's look at how we can change the default behaviour of how these plugins are called and how we can get our plugins in there.

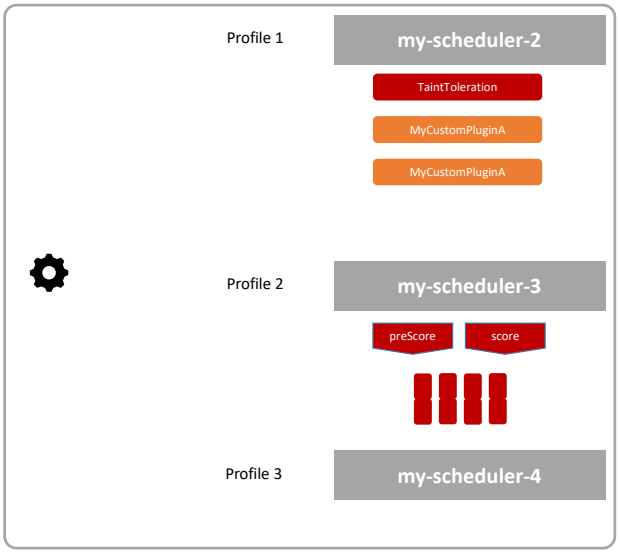


Earlier we talked about deploying 3 separate schedulers each with a separate scheduler binary. So, we have the default-scheduler, then my-scheduler and my-scheduler-2. <c> All of these are 3 separate scheduler binaries run with a separate scheduler-config file associated with each of them. Now that's one way to deploy multiple schedulers. The problem here is that since these are separate processes there is an additional effort required to maintain these separate processes. Also, since they are separate processes they may run into race conditions while making scheduling decisions. One scheduler may schedule a workload on a node without knowing that there's another scheduler scheduling a workload on that same node at that time.



And so, with the 1.18 release of Kubernetes, a feature to support multiple profiles in a single scheduler was introduced. Now you can configure multiple profiles within a single scheduler in the schedule configuration file by adding more entries to the list of profiles. For each profile specify a schedulerName. This creates a separate profile for each scheduler which acts as a separate scheduler itself. Except that now multiple schedulers run in the same binary.

Now how do you configure these different scheduler profiles to work differently?



my-scheduler-2-config.yaml

```yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
- schedulerName: my-scheduler-2
  plugins:
    score:
      disabled:
       - name: TaintToleration
      enabled:
       - name: MyCustomPluginA
       - name: MyCustomPluginB

- schedulerName: my-scheduler-3
  plugins:
    preScore:
      disabled:
       - name: '*'
    score:
      disabled:
       - name: '*'

- schedulerName: my-scheduler-4
```

Under each scheduler profile we can configure the plugins the way we want to. For example, for the my-scheduler-2 profile I am going to disable certain plugins like the TaintToleration and enable my own custom plugins.

For the my-scheduler-3 profile I am going to disable all preScore and score plugins. So this is how that's going to look. Under pugins section specify the extensionPoint and enable or disable the plugins by name or a pattern as shown in this case.

Well that's about it. I hope that gives you an overview of how schedulers and scheduler profiles work and how you can configure multiple scheduler profiles in kubernetes.