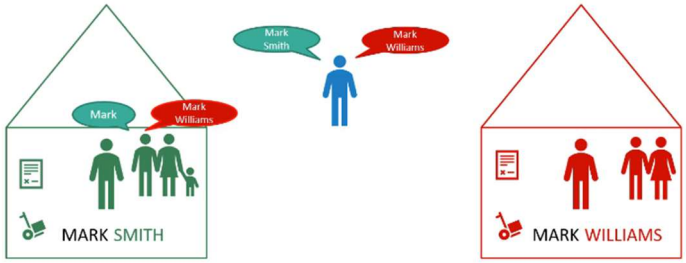
Namespace

In this lecture, we will discuss name spaces in Kubernetes. Let us begin with an analogy.

There are two boys named Mark. To differentiate them from each other, we call them by their last names, Smith and Williams. They come from different houses – the Smiths and the Williams. There are other members in the house. The individuals within the house address each other simply by their first names. For example, the father addresses Mark simply as Mark.

However, if the father wishes to address the Mark in the other house, he would use the full name. Someone outside of these houses would also use the full name to refer to the boys or anyone within these houses.

Each of these houses has its own set of rules that define who does what. Each of these houses has its own set of resources that they can consume.

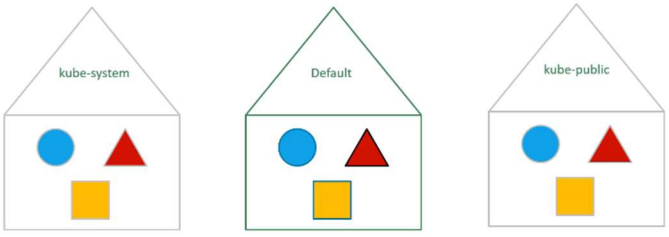


So far in this course, we've created objects such as pods, deployments, and services in our cluster. Whatever we have been doing, we have been doing within a namespace. We were inside a house all this while.

This namespace is known as the default namespace, and it is created automatically by Kubernetes when the cluster is first set up. Kubernetes creates a set of pods and services for its internal purposes, such as those required by the networking solution, the DNS service, etc.

To isolate these from the user and to prevent you from accidentally deleting or modifying these services, Kubernetes creates them under another namespace created at cluster startup named **kube-system**.

A third namespace created by Kubernetes automatically is called **kube-public**. This is where resources that should be made available to all users are created.

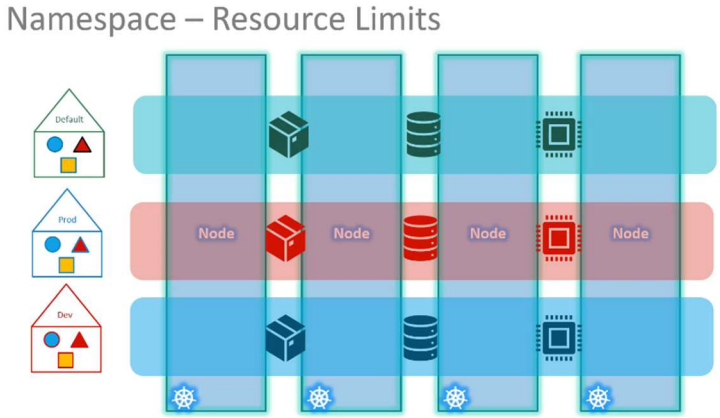


If your environment is small or you're learning and playing around with a small cluster, you shouldn't really have to worry about namespaces. You could continue to work in the default namespace.

However, as and when you grow and use a Kubernetes cluster for enterprise or production purposes, you may want to consider the use of namespaces. You can create your own namespaces as well.

For example, if you wanted to use the same cluster for both the development and production environments but, at the same time, isolate the resources between them, you can create a different namespace for each of them. That way, while working in the dev environment, you don't accidentally modify resources in production. Each of these namespaces can have its own set of policies that define who can do what.

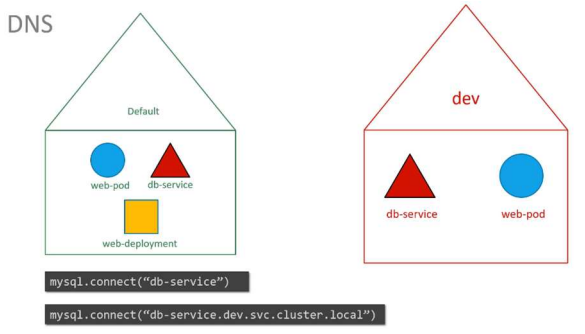
You can also assign quota of resources to each of these name spaces. That way, each name space is guaranteed a certain amount and does not use more than its allowed limit.



Going back to the default namespace that we have been working on. Just like how the members within a house refer to each other by their first names, the resources within a namespace can refer to each other simply by their names.

In this case, the web app pod can reach the DB service simply using the hostname "DB service."

If required, the web app pod can reach a service in another namespace as well. For this, you must append the name of the namespace to the name of the service. For example, for the web pod in the default namespace to connect to the database in the dev environment or namespace, use the "**servicename.namespace.svc.cluster.local**" format. That would be "**dbservice.dev.svc.cluster.local**." You're able to do this because when the service is created, a DNS entry is added automatically in this format.




```
mysql.connect("db-service.dev.svc.cluster.local")
```

Looking closely at the DNS name of the service, the last part, "cluster.local," is the default domain name of the Kubernetes cluster. "SVC" is the subdomain for service, followed by the namespace, and then the name of the service itself.

Let us now look at some of the operational aspects of namespaces. Let us start with the kubectl commands. For example, this command is used to list all the pods, but it only lists the pods in the default namespace. To list pods in another namespace, use the namespace option in the command along with the name of the namespace. In this case, kube-system.

```
> kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
Pod-1     1/1     Running   0          3d
Pod-2     1/1     Running   0          3d
```

```
> kubectl get pods --namespace=kube-system
NAME                              READY   STATUS    RESTAR
coredns-78fcdf6894-92d52          1/1     Running   7
coredns-78fcdf6894-jx25g          1/1     Running   7
etcd-master                       1/1     Running   7
kube-apiserver-master             1/1     Running   7
kube-controller-manager-master    1/1     Running   7
kube-flannel-ds-amd64-hz4cf       1/1     Running   14
kube-proxy-4b8tn                  1/1     Running   7
kube-proxy-98db4                  1/1     Running   7
kube-proxy-jjrbs                  1/1     Running   7
kube-scheduler-master             1/1     Running   7
```

Here, I have a pod definition file. When you create a pod using this file, the pod is created in the default namespace. To create the pod in another namespace, use the namespace option.

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

```
> kubectl create -f pod-definition.yml --namespace=dev
```

If you want to make sure that this pod gets created in the dev environment all the time, even if you don't specify the namespace in the command-line, you can move the namespace definition into the pod definition file.

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
  name: myapp-pod
  namespace: dev
  labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Like this, under the metadata section. This is a good way to ensure your resources are consistently created in the desired namespace.

So, how do you create a new namespace?

Like any other object, use a namespace definition file. The API version is V1, kind is namespace, and under metadata, specify the name. In this case, dev. Run the kubectl create command to create the namespace. Another way to create a namespace is by simply running the command kubectl create namespace, followed by the name of the namespace

```
namespace-dev.yml
apiVersion: v1
kind: Namespace
metadata:
    name: dev
```

```
> kubectl create -f namespace-dev.yml
namespace/dev created
```

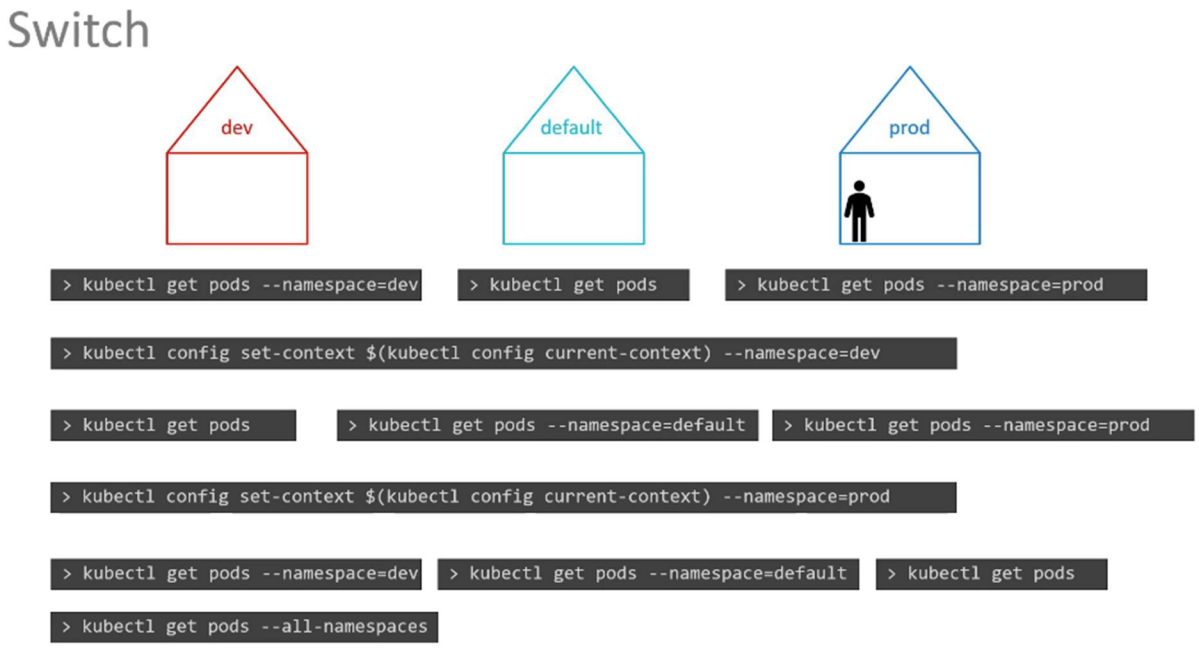
```
> kubectl create namespace dev
namespace/dev created
```

Now, say we are working in three namespaces. As we discussed before, by default, we are in the default namespace, which is why we can see the resources inside the default namespace using the "kubectl get pods" command. To view those in the dev namespace, we have to use the namespace option.

But what if we want to switch to the dev namespace permanently, so that we don't have to specify the namespace option anymore? Well, in that case, use the "kubectl config" command to set the namespace in the current context to dev.

You can then simply run the "kubectl get pods" command without the namespace option to list pods in the dev environment. But you will need to specify the option for other environments, such as default or prod.

Similarly, you can switch to the prod namespace the same way. Finally, to view pods in all namespaces, use the "all namespaces" option in the command. This will list all the pods in all of the namespaces.



To limit resources in a namespace, create a resource quota. To create one, start with a definition file for resource quota, specify the namespace for which you want to create the quota, and then under spec, provide your limits such as 10 pods, 10 CPU units, 10 GB of memory, etcetera.

Compute-quota.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
    name: compute-quota
    namespace: dev

spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi
```