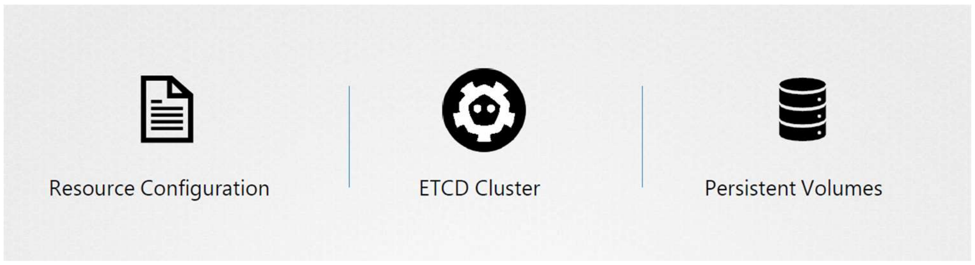
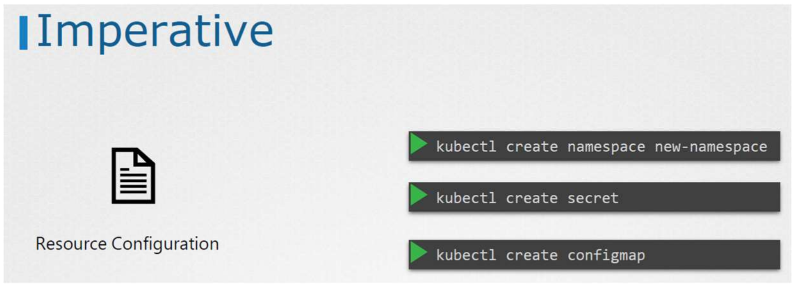
Let's start by looking at what you should consider backing up in Kubernetes cluster. So far in this course, we have deployed a number of different applications on our Kubernetes cluster using deployment, pods, and service definition files. We know that the etcd cluster is where all cluster-related information is stored, and if your applications are configured with persistent storage, then that is another candidate for backups.

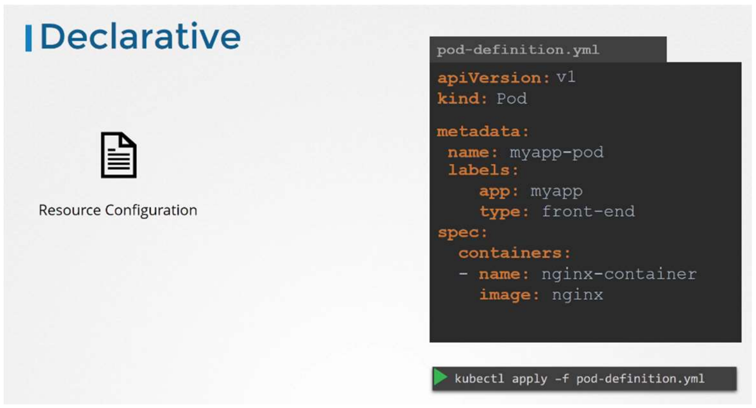


With respect to **resources** that we created in the cluster, at times we use the imperative way of creating an object by executing a command, such as while creating a namespace, or a secret, or config map, or, at times, for exposing applications.



```
kubectl create namespace new-namespace
kubectl create secret
kubectl create configmap
```

And at times, we used the declarative approach by first creating a definition file and then running the kubectl apply command on that file.



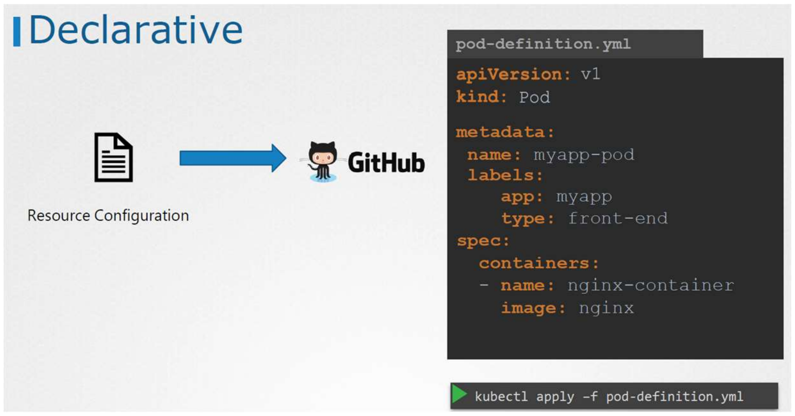
```
apiVersion: v1
kind: Pod

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

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

This is the preferred approach if you want to save your configuration, because now, you have all the objects required for a single application in the form of object definition files in a single folder. This can easily be reused at a later time, or shared with others. Of course, you must have a copy of these files saved at all times. A good practice is to store these on source code repositories, that way it can be maintained by a team. The source code repository should be configured with the right backup solutions.

With managed or public source code repositories like GitHub, you don't have to worry about this. With that, even when you lose your entire cluster, you can redeploy your application on the cluster by simply applying these configuration files on them.



## Declarative

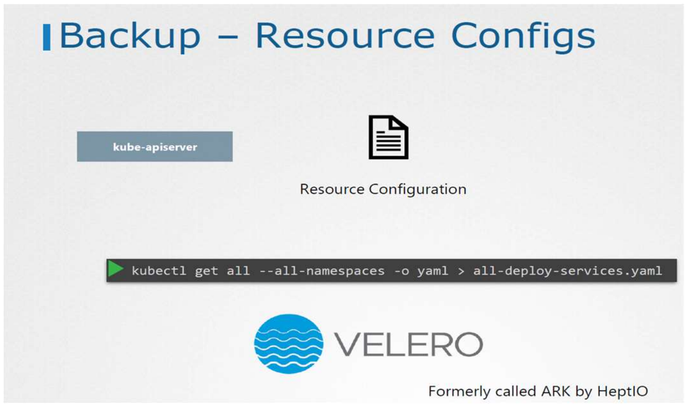
```
apiVersion: v1
kind: Pod

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

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

While the declarative approach is the preferred approach, it is not necessary that all of your team members stick to those standards. What if someone created an object the imperative way without documenting that information anywhere? So a better approach to backing up resource configuration is to query the Kube API server. Query the Kube API server using the kubectl, or by accessing the API server directly, and save all resource configurations for all objects created on the cluster as a copy.

For example, one of the commands that can be used in a backup script is to get all pods, and deployments, and services in all namespaces using the kubectl utility's get all command, and extract the output in a YAML format, then save that file. And that's just for a few resource group. There are many other resource groups that must be considered. Of course, you don't have to develop that solution yourself. There are tools like Ark, or now called Velero, by Heptio, that can do this for you. It can help in taking backups of your Kubernetes cluster using the Kubernetes API.



## Backup – Resource Configs

```
kubectl get all --all-namespaces -o yaml > all-deploy-services.yaml
```

Let us now move on to **etcd**. The etcd cluster stores information about the state of our cluster. So information about the cluster itself, the nodes and every other resources created within the cluster, are stored here. So instead of backing up resource as before, you may choose to back up the etcd server itself.

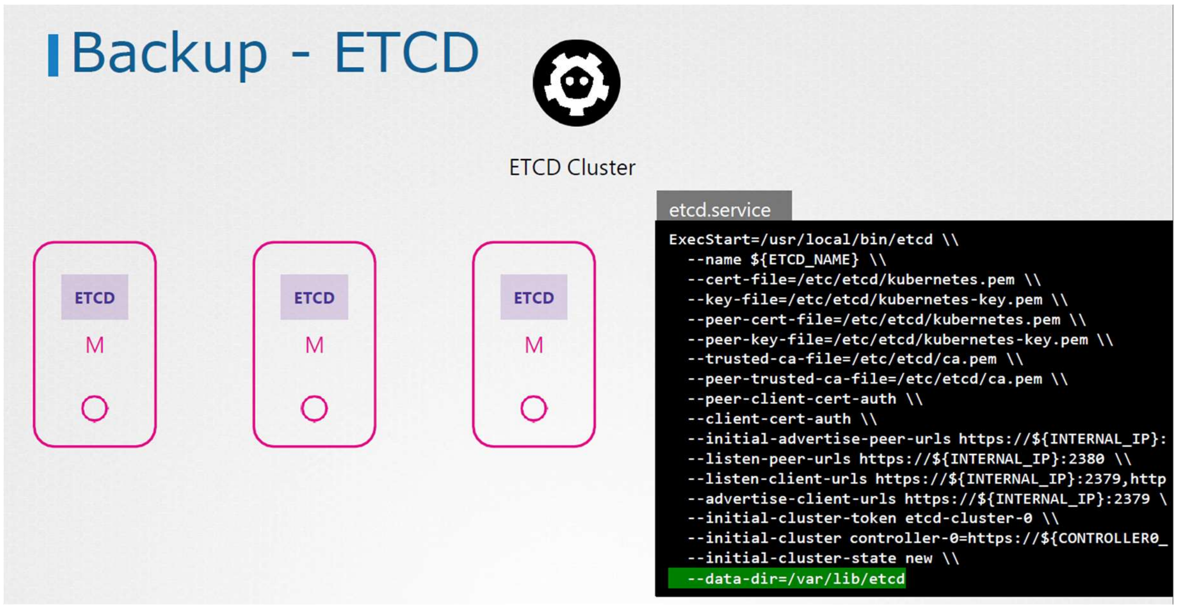
Here are some of the things that are stored in ETCD:

Cluster configuration: This includes information such as the cluster's name, the cluster's IP address, and the cluster's DNS name.

Pod state: This includes information such as the pod's name, the pod's IP address, and the pod's container images.

Service state: This includes information such as the service's name, the service's IP address, and the service's port number.

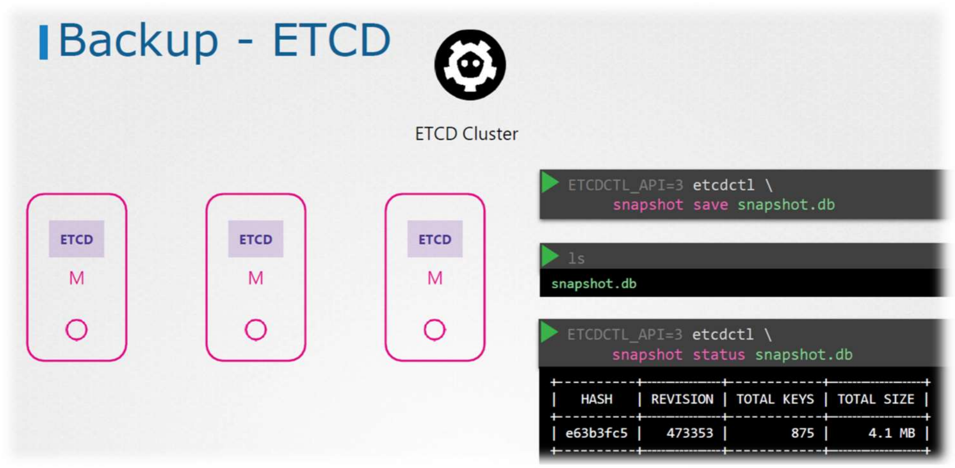
As we have seen, the etcd cluster is hosted on the master nodes. While configuring etcd, we specified a location where all the data would be stored, the data directory. That is the directory that can be configured to be backed up by your backup tool.



## Backup - ETCD

```
ExecStart=/usr/local/bin/etcd \\
  --name ${ETCD_NAME} \\
  --cert-file=/etc/etcd/kubernetes.pem \\
  --key-file=/etc/etcd/kubernetes-key.pem \\
  --peer-cert-file=/etc/etcd/kubernetes.pem \\
  --peer-key-file=/etc/etcd/kubernetes-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${INTERNAL_IP}:
  --listen-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-client-urls https://${INTERNAL_IP}:2379,http
  --advertise-client-urls https://${INTERNAL_IP}:2379 \
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster controller-0=https://${CONTROLLER0_
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
```

Etcd also comes with a built-in snapshot solution. You can take a snapshot of the etcd database by using the etcd control utility's snapshot save command. Give the snapshot a name, snapshot.db. A snapshot file is created by the name in the current directory. If you want it to be created in another location, specify the full path. You can view the status of the backup using the snapshot status command.



## Backup - ETCD

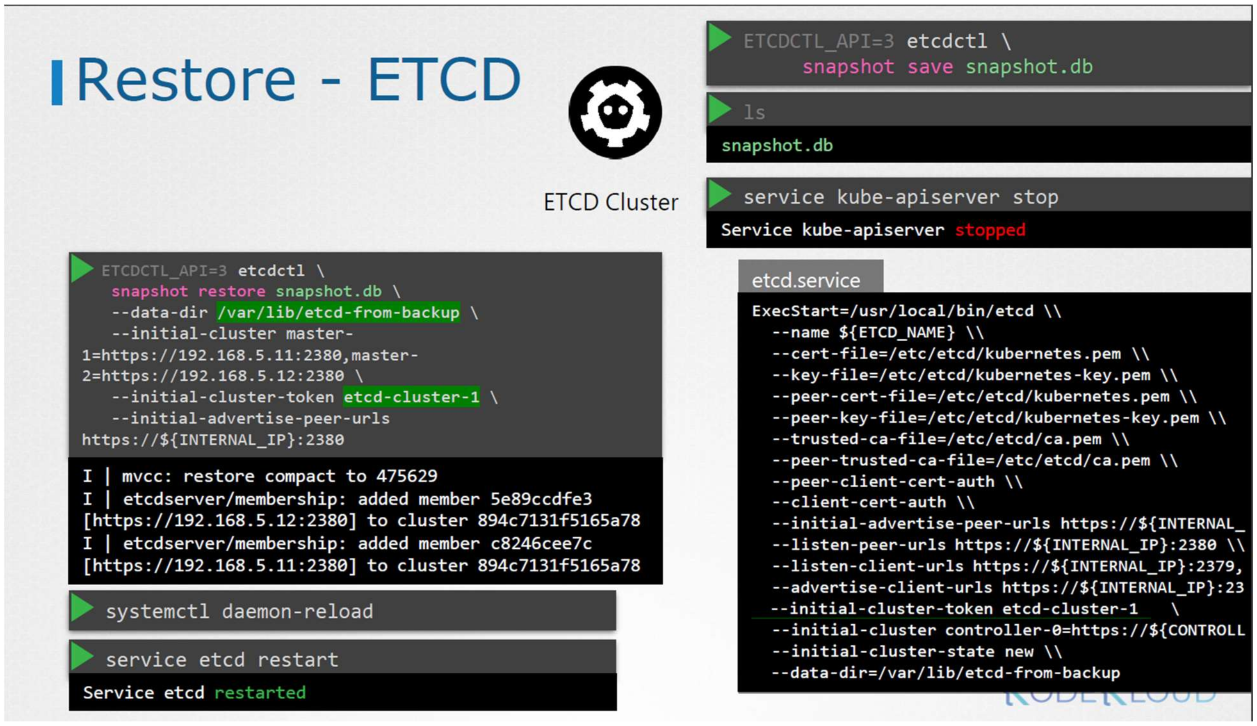
```
▶ ETCDCTL_API=3 etcdctl \
      snapshot save snapshot.db
```

```
▶ ls
snapshot.db
```

```
▶ ETCDCTL_API=3 etcdctl \
      snapshot status snapshot.db
+----------+----------+------------+------------+
|   HASH   | REVISION | TOTAL KEYS | TOTAL SIZE |
+----------+----------+------------+------------+
| e63b3fc5 |   473353 |        875 |     4.1 MB |
+----------+----------+------------+------------+
```

To restore the cluster from this backup at a later point in time, first, stop the Kube API server service, as the restore process will require you to restart the etcd cluster, and the Kube API server depends on it. Then, run the etcd controls snapshot restore command, with the path set to the path of the backup file, which is the snapshot.db file.

When etcd restores from a backup, it initializes a new cluster configuration and configures the members of etcd as new members to a new cluster. This is to prevent a new member from accidentally joining an existing cluster. On running this command, a new data directory is created. In this example, at location var lib etcd from backup. We then configure the etcd configuration file to use the new data directory. Then, reload the service demon and restart etcd service. Finally, start the Kube API server service. Your cluster should now be back in the original state.



## Restore - ETCD

```
▶ ETCDCTL_API=3 etcdctl \
      snapshot save snapshot.db
```

```
▶ ls
snapshot.db
```

```
▶ service kube-apiserver stop
Service kube-apiserver stopped
```

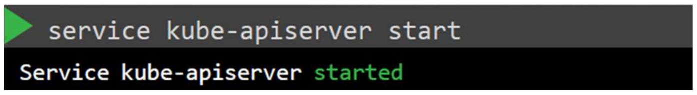
```
▶ ETCDCTL_API=3 etcdctl \
    snapshot restore snapshot.db \
    --data-dir /var/lib/etcd-from-backup \
    --initial-cluster master-
1=https://192.168.5.11:2380,master-
2=https://192.168.5.12:2380 \
    --initial-cluster-token etcd-cluster-1 \
    --initial-advertise-peer-urls
https://${INTERNAL_IP}:2380
I | mvcc: restore compact to 475629
I | etcdserver/membership: added member 5e89ccdfe3
[https://192.168.5.12:2380] to cluster 894c7131f5165a78
I | etcdserver/membership: added member c8246cee7c
[https://192.168.5.11:2380] to cluster 894c7131f5165a78
```

```
▶ systemctl daemon-reload
```

```
▶ service etcd restart
Service etcd restarted
```

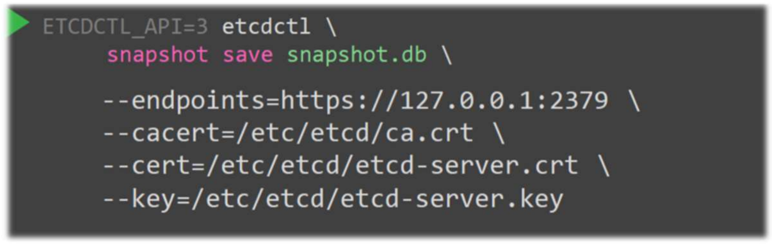
etcd.service

```
ExecStart=/usr/local/bin/etcd \\
  --name ${ETCD_NAME} \\
  --cert-file=/etc/etcd/kubernetes.pem \\
  --key-file=/etc/etcd/kubernetes-key.pem \\
  --peer-cert-file=/etc/etcd/kubernetes.pem \\
  --peer-key-file=/etc/etcd/kubernetes-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${INTERNAL_
  --listen-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-client-urls https://${INTERNAL_IP}:2379,
  --advertise-client-urls https://${INTERNAL_IP}:23
  --initial-cluster-token etcd-cluster-1    \
  --initial-cluster controller-0=https://${CONTROLL
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd-from-backup
```



```
service kube-apiserver start
Service kube-apiserver started
```

A quick note before I let you go. With all the etcd commands, remember to specify the certificate files for authentication, specify the endpoint to the etcd cluster and the CS certificate, the etcd server certificate, and the key.



```
ETCDCTL_API=3 etcdctl \
     snapshot save snapshot.db \

     --endpoints=https://127.0.0.1:2379 \
     --cacert=/etc/etcd/ca.crt \
     --cert=/etc/etcd/etcd-server.crt \
     --key=/etc/etcd/etcd-server.key
```

So we have seen two options, a backup using etcd, and a backup by querying the Kube API server. Now both of these have their pros and cons. Well, if you're using a managed Kubernetes environment, then, at times, you may not even access to the etcd cluster. In that case, backup by querying the Kube API server is probably the better way.