**Working with ETCDCTL**

etcdctl is a command line client for [**etcd**](https://github.com/coreos/etcd).

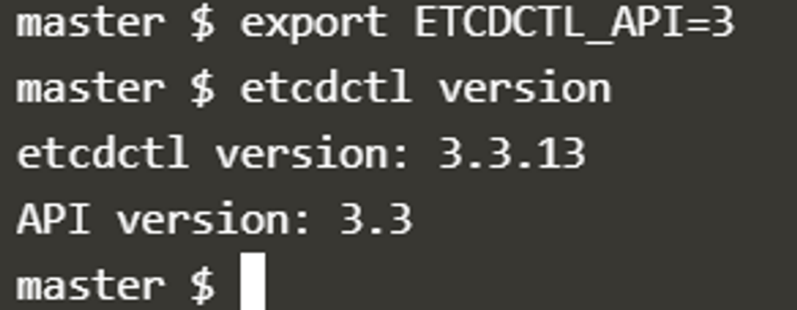
In all our Kubernetes Hands-on labs, the ETCD key-value database is deployed as a static pod on the master. The version used is v3.

To make use of etcdctl for tasks such as back up and restore, make sure that you set the ETCDCTL\_API to 3.

You can do this by exporting the variable ETCDCTL\_API prior to using the etcdctl client. This can be done as follows:

export ETCDCTL\_API=3

On the **Master Node**:



To see all the options for a specific sub-command, make use of the**-h or --help** flag.

For example, if you want to take a snapshot of etcd, use:

etcdctl snapshot save -h and keep a note of the mandatory global options.

Since our ETCD database is TLS-Enabled, the following options are mandatory:

--cacert                                                verify certificates of TLS-enabled secure servers using this CA bundle

--cert                                                    identify secure client using this TLS certificate file

--endpoints=[127.0.0.1:2379]          This is the default as ETCD is running on master node and exposed on localhost 2379.

--key                                                      identify secure client using this TLS key file

Similarly use the help option for **snapshot restore**to see all available options for restoring the backup.

etcdctl snapshot restore -h

For a detailed explanation on how to make use of the etcdctl command line tool and work with the -h flags, check out the solution video for the Backup and Restore Lab.

**Article on Setting up Basic Authentication**

Setup basic authentication on Kubernetes (Deprecated in 1.19)

Note: This is not recommended in a production environment. This is only for learning purposes. Also note that this approach is deprecated in Kubernetes version 1.19 and is no longer available in later releases

Follow the below instructions to configure basic authentication in a kubeadm setup.

Create a file with user details locally at /tmp/users/user-details.csv

1. # User File Contents
2. password123,user1,u0001
3. password123,user2,u0002
4. password123,user3,u0003
5. password123,user4,u0004
6. password123,user5,u0005

Edit the kube-apiserver static pod configured by kubeadm to pass in the user details. The file is located at /etc/kubernetes/manifests/kube-apiserver.yaml

1. apiVersion: v1
2. kind: Pod
3. metadata:
4. name: kube-apiserver
5. namespace: kube-system
6. spec:
7. containers:
8. - command:
9. - kube-apiserver
10. <content-hidden>
11. image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
12. name: kube-apiserver
13. volumeMounts:
14. - mountPath: /tmp/users
15. name: usr-details
16. readOnly: true
17. volumes:
18. - hostPath:
19. path: /tmp/users
20. type: DirectoryOrCreate
21. name: usr-details

Modify the kube-apiserver startup options to include the basic-auth file

1. apiVersion: v1
2. kind: Pod
3. metadata:
4. creationTimestamp: null
5. name: kube-apiserver
6. namespace: kube-system
7. spec:
8. containers:
9. - command:
10. - kube-apiserver
11. - --authorization-mode=Node,RBAC
12. <content-hidden>
13. - --basic-auth-file=/tmp/users/user-details.csv

Create the necessary roles and role bindings for these users:

1. ---
2. kind: Role
3. apiVersion: rbac.authorization.k8s.io/v1
4. metadata:
5. namespace: default
6. name: pod-reader
7. rules:
8. - apiGroups: [""] # "" indicates the core API group
9. resources: ["pods"]
10. verbs: ["get", "watch", "list"]
12. ---
13. # This role binding allows "jane" to read pods in the "default" namespace.
14. kind: RoleBinding
15. apiVersion: rbac.authorization.k8s.io/v1
16. metadata:
17. name: read-pods
18. namespace: default
19. subjects:
20. - kind: User
21. name: user1 # Name is case sensitive
22. apiGroup: rbac.authorization.k8s.io
23. roleRef:
24. kind: Role #this must be Role or ClusterRole
25. name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
26. apiGroup: rbac.authorization.k8s.io

Once created, you may authenticate into the kube-api server using the users credentials

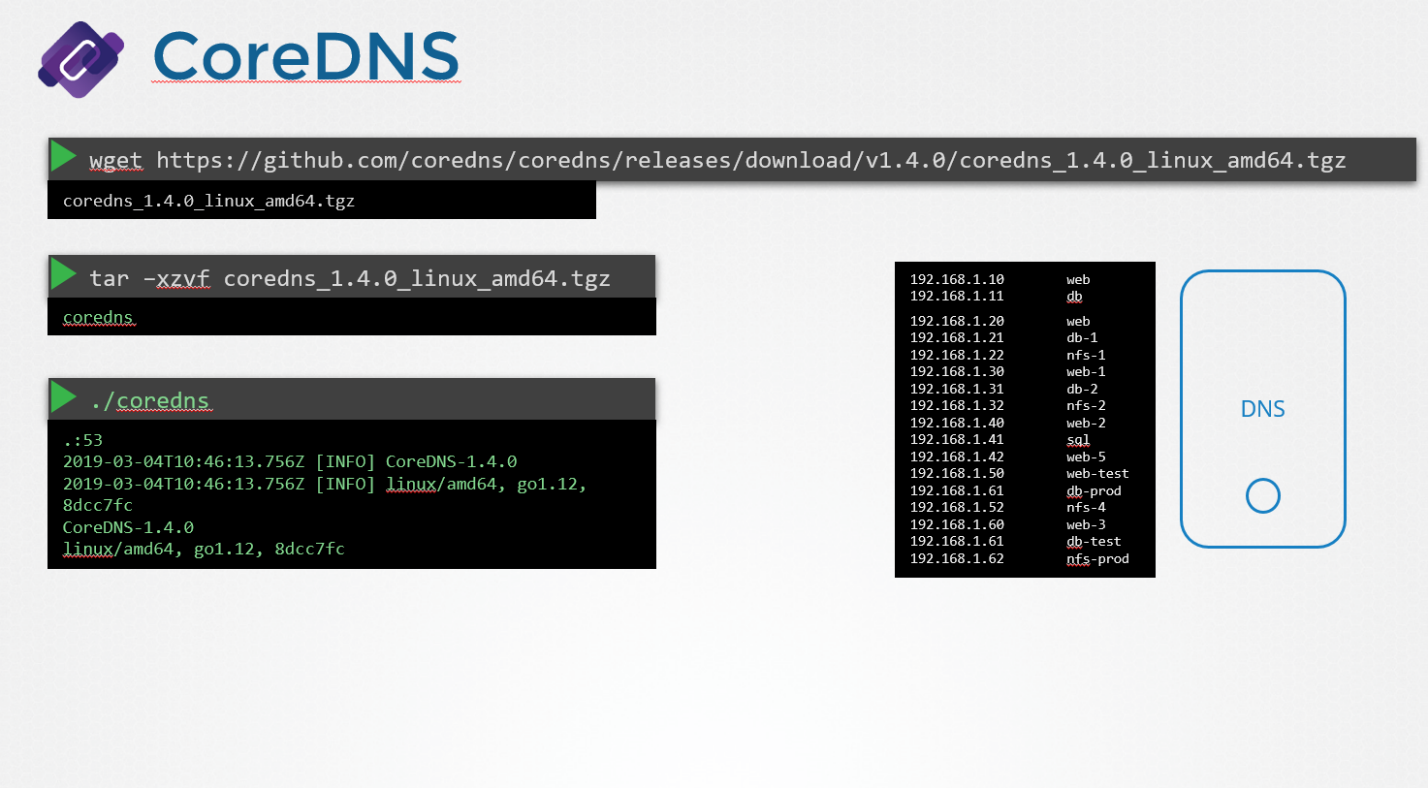
curl -v -k https://localhost:6443/api/v1/pods -u "user1:password123"

**Prerequisite - CoreDNS**

In the previous lecture we saw why you need a DNS server and how it can help manage name resolution in large environments with many hostnames and Ips and how you can configure your hosts to point to a DNS server. In this article we will see how to configure a host as a DNS server.

We are given a server dedicated as the DNS server, and a set of Ips to configure as entries in the server. There are many DNS server solutions out there, in this lecture we will focus on a particular one – CoreDNS.

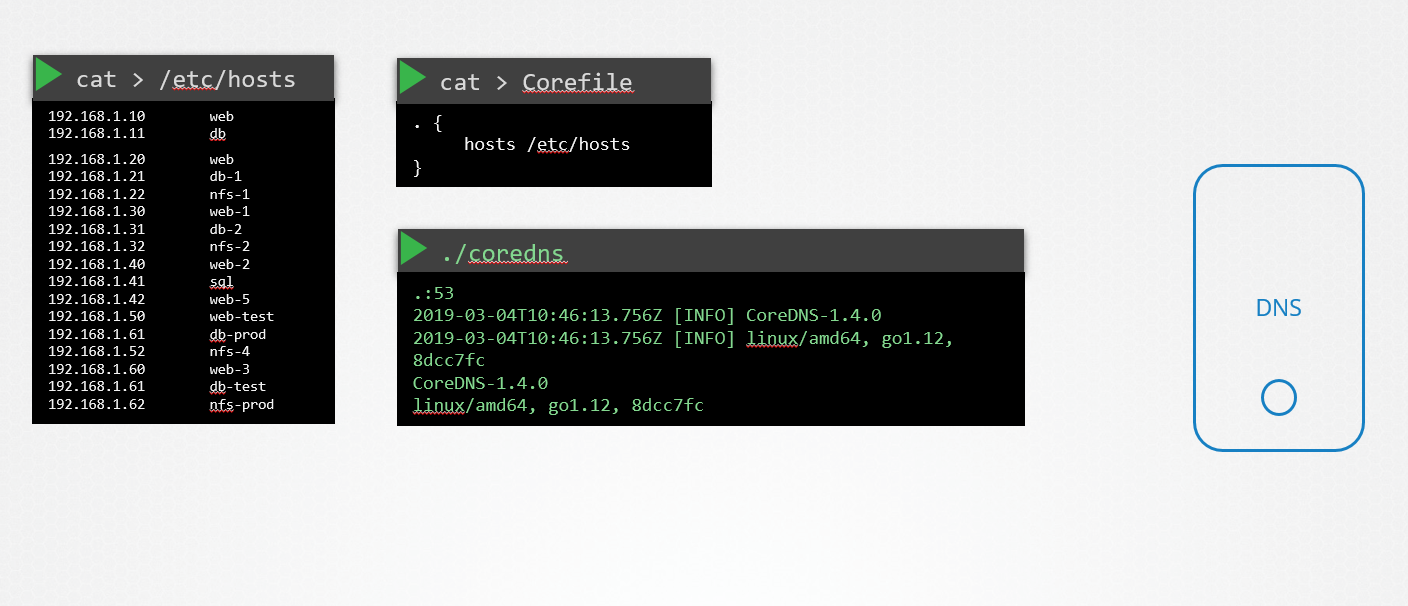
So how do you get core dns? CoreDNS binaries can be downloaded from their Github releases page or as a docker image. Let’s go the traditional route. Download the binary using curl or wget. And extract it. You get the coredns executable.



Run the executable to start a DNS server. It by default listens on port 53, which is the default port for a DNS server.

Now we haven’t specified the IP to hostname mappings. For that you need to provide some configurations. There are multiple ways to do that. We will look at one. First we put all of the entries into the DNS servers /etc/hosts file.

And then we configure CoreDNS to use that file. CoreDNS loads it’s configuration from a file named Corefile. Here is a simple configuration that instructs CoreDNS to fetch the IP to hostname mappings from the file /etc/hosts. When the DNS server is run, it now picks the Ips and names from the /etc/hosts file on the server.



CoreDNS also supports other ways of configuring DNS entries through plugins. We will look at the plugin that it uses for Kubernetes in a later section.

Read more about CoreDNS here:

<https://github.com/kubernetes/dns/blob/master/docs/specification.md>

<https://coredns.io/plugins/kubernetes/>

**Important Note about CNI and CKA Exam**

**An important tip about deploying Network Addons in a Kubernetes cluster.**

In the upcoming labs, we will work with Network Addons. This includes installing a network plugin in the cluster. While we have used weave-net as an example, please bear in mind that you can use any of the plugins which are described here:

[**https://kubernetes.io/docs/concepts/cluster-administration/addons/**](https://kubernetes.io/docs/concepts/cluster-administration/addons/)

[**https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model**](https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model)

In the CKA exam, for a question that requires you to deploy a network addon, unless specifically directed, you may use any of the solutions described in the link above.

**However,**the documentation currently does not contain a direct reference to the exact command to be used to deploy a third party network addon.

The links above redirect to third party/ vendor sites or GitHub repositories which cannot be used in the exam. This has been intentionally done to keep the content in the Kubernetes documentation vendor-neutral.

At this moment in time, there is still one place within the documentation where you can find the exact command to deploy weave network addon:

<https://v1-22.docs.kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/#steps-for-the-first-control-plane-node> (step 2)

**Article: Ingress**

As we already discussed Ingress in our previous lecture. Here is an update.

In this article, we will see what changes have been made in previous and current versions in Ingress.

Like in apiVersion, serviceName and servicePort etc.

Now, in k8s version 1.20+ we can create an Ingress resource from the imperative way like this:-

Format - kubectl create ingress <ingress-name> --rule="host/path=service:port"

Example - kubectl create ingress ingress-test --rule="wear.my-online-store.com/wear\*=wear-service:80"

Find more information and examples in the below reference link: -

https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#-em-ingress-em-

**References: -**

https://kubernetes.io/docs/concepts/services-networking/ingress

<https://kubernetes.io/docs/concepts/services-networking/ingress/#path-types>

**Ingress - Annotations and rewrite-target**

Different ingress controllers have different options that can be used to customise the way it works. NGINX Ingress controller has many options that can be seen [here](https://kubernetes.github.io/ingress-nginx/examples/). I would like to explain one such option that we will use in our labs. The [Rewrite](https://kubernetes.github.io/ingress-nginx/examples/rewrite/) target option.

Our watch app displays the video streaming webpage at http://<watch-service>:<port>/

Our wear app displays the apparel webpage at http://<wear-service>:<port>/

We must configure Ingress to achieve the below. When user visits the URL on the left, his request should be forwarded internally to the URL on the right. Note that the /watch and /wear URL path are what we configure on the ingress controller so we can forwarded users to the appropriate application in the backend. The applications don't have this URL/Path configured on them:  
  
http://<ingress-service>:<ingress-port>/watch --> http://<watch-service>:<port>/

http://<ingress-service>:<ingress-port>/wear --> http://<wear-service>:<port>/

Without the rewrite-target option, this is what would happen:

http://<ingress-service>:<ingress-port>/watch --> http://<watch-service>:<port>/watch

http://<ingress-service>:<ingress-port>/wear --> http://<wear-service>:<port>/wear

Notice watch and wear at the end of the target URLs. The target applications are not configured with /watch or /wear paths. They are different applications built specifically for their purpose, so they don't expect /watch or /wear in the URLs. And as such the requests would fail and throw a 404 not found error.

To fix that we want to "ReWrite" the URL when the request is passed on to the watch or wear applications. We don't want to pass in the same path that user typed in. So we specify the rewrite-target option. This rewrites the URL by replacing whatever is under rules->http->paths->path which happens to be /pay in this case with the value in rewrite-target. This works just like a search and replace function.

For example: replace(path, rewrite-target)  
In our case: replace("/path","/")

1. apiVersion: extensions/v1beta1
2. kind: Ingress
3. metadata:
4. name: test-ingress
5. namespace: critical-space
6. annotations:
7. nginx.ingress.kubernetes.io/rewrite-target: /
8. spec:
9. rules:
10. - http:
11. paths:
12. - path: /pay
13. backend:
14. serviceName: pay-service
15. servicePort: 8282

In another example given [here](https://kubernetes.github.io/ingress-nginx/examples/rewrite/), this could also be:

replace("/something(/|$)(.\*)", "/$2")

1. apiVersion: extensions/v1beta1
2. kind: Ingress
3. metadata:
4. annotations:
5. nginx.ingress.kubernetes.io/rewrite-target: /$2
6. name: rewrite
7. namespace: default
8. spec:
9. rules:
10. - host: rewrite.bar.com
11. http:
12. paths:
13. - backend:
14. serviceName: http-svc
15. servicePort: 80
16. path: /something(/|$)(.\*)