Certification Tip!

Here's a tip!

As you might have seen already, it is a bit difficult to create and edit YAML files. Especially in the CLI. During the exam, you might find it difficult to copy and paste YAML files from browser to terminal. Using the kubectl run command can help in generating a YAML template. And sometimes, you can even get away with just the kubectl run command without having to create a YAML file at all. For example, if you were asked to create a pod or deployment with specific name and image you can simply run the kubectl run command.

Use the below set of commands and try the previous practice tests again, but this time try to use the below commands instead of YAML files. Try to use these as much as you can going forward in all exercises

Reference (Bookmark this page for exam. It will be very handy):

<https://kubernetes.io/docs/reference/kubectl/conventions/>

**Create an NGINX Pod**

kubectl run --generator=run-pod/v1 nginx --image=nginx

**Generate POD Manifest YAML file (-o yaml). Don't create it(--dry-run)**

kubectl run --generator=run-pod/v1 nginx --image=nginx --dry-run -o yaml

**Create a deployment**

kubectl run --generator=deployment/v1beta1 nginx --image=nginx

**Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)**

kubectl run --generator=deployment/v1beta1 nginx --image=nginx --dry-run -o yaml

**Generate Deployment YAML file (-o yaml). Don't create it(--dry-run) with 4 Replicas (--replicas=4)**

kubectl run --generator=deployment/v1beta1 nginx --image=nginx --dry-run --replicas=4 -o yaml

**Save it to a file - (If you need to modify or add some other details before actually creating it)**

kubectl run --generator=deployment/v1beta1 nginx --image=nginx --dry-run --replicas=4 -o yaml > nginx-deployment.yaml

Certification Tips - Imperative Commands with Kubectl

While you would be working mostly the declarative way - using definition files, imperative commands can help in getting one time tasks done quickly, as well as generate a definition template easily. This would help save considerable amount of time during your exams.

Before we begin, familiarize with the two options that can come in handy while working with the below commands:

--dry-run: By default as soon as the command is run, the resource will be created. If you simply want to test your command , use the --dry-run option. This will not create the resource, instead, tell you whether the resource can be created and if your command is right.

-o yaml: This will output the resource definition in YAML format on screen.

Use the above two in combination to generate a resource definition file quickly, that you can then modify and create resources as required, instead of creating the files from scratch.

#### **POD**

**Create an NGINX Pod**

kubectl run --generator=run-pod/v1 nginx --image=nginx

**Generate POD Manifest YAML file (-o yaml). Don't create it(--dry-run)**

kubectl run --generator=run-pod/v1 nginx --image=nginx --dry-run -o yaml

#### **Deployment**

**Create a deployment**

kubectl run --generator=deployment/v1beta1 nginx --image=nginx

Or the newer recommended way:

kubectl create deployment --image=nginx nginx

**Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)**

kubectl run --generator=deployment/v1beta1 nginx --image=nginx --dry-run -o yaml

Or

kubectl create deployment --image=nginx nginx --dry-run -o yaml

**Generate Deployment YAML file (-o yaml). Don't create it(--dry-run) with 4 Replicas (--replicas=4)**

kubectl run --generator=deployment/v1beta1 nginx --image=nginx --dry-run --replicas=4 -o yaml

kubectl create deployment does not have a --replicas option. You could first create it and then scale it using the kubectl scale command.

**Save it to a file - (If you need to modify or add some other details)**

kubectl run --generator=deployment/v1beta1 nginx --image=nginx --dry-run --replicas=4 -o yaml > nginx-deployment.yaml

#### **Service**

**Create a Service named redis-service of type ClusterIP to expose pod redis on port 6379**

kubectl expose pod redis --port=6379 --name redis-service --dry-run -o yaml

(This will automatically use the pod's labels as selectors)

Or

kubectl create service clusterip redis --tcp=6379:6379 --dry-run -o yaml  (This will not use the pods labels as selectors, instead it will assume selectors as **app=redis.**[You cannot pass in selectors as an option.](https://github.com/kubernetes/kubernetes/issues/46191)So it does not work very well if your pod has a different label set. So generate the file and modify the selectors before creating the service)

**Create a Service named nginx of type NodePort to expose pod nginx's port 80 on port 30080 on the nodes:**

kubectl expose pod nginx --port=80 --name nginx-service --dry-run -o yaml

(This will automatically use the pod's labels as selectors, [but you cannot specify the node port](https://github.com/kubernetes/kubernetes/issues/25478). You have to generate a definition file and then add the node port in manually before creating the service with the pod.)

Or

kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run -o yaml

(This will not use the pods labels as selectors)

Both the above commands have their own challenges. While one of it cannot accept a selector the other cannot accept a node port. I would recommend going with the `kubectl expose` command. If you need to specify a node port, generate a definition file using the same command and manually input the nodeport before creating the service.

**Reference:**

<https://kubernetes.io/docs/reference/kubectl/conventions/>

* [Overview](https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/learn/lecture/15018998#overview)
* [Q&A](https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/learn/lecture/15018998#questions)
* [Bookmarks](https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/learn/lecture/15018998#bookmarks)
* [Announcements](https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/learn/lecture/15018998#announcements)

A quick note on editing PODs and Deployments

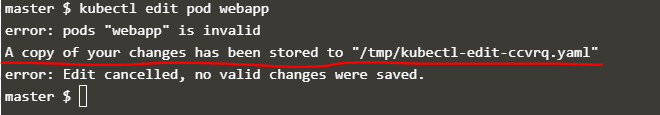
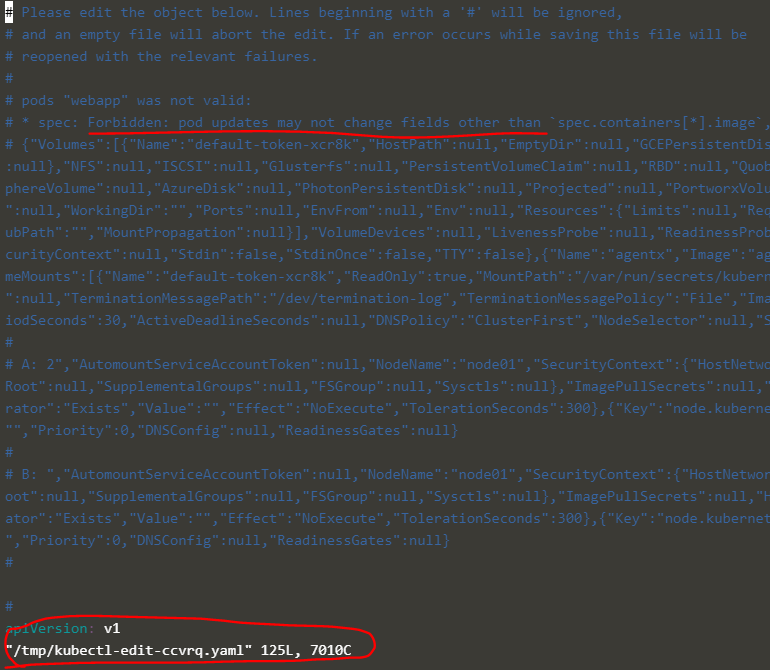
#### **Edit a POD**

Remember, you CANNOT edit specifications of an existing POD other than the below.

* spec.containers[\*].image
* spec.initContainers[\*].image
* spec.activeDeadlineSeconds
* spec.tolerations

For example you cannot edit the environment variables, service accounts, resource limits (all of which we will discuss later) of a running pod. But if you really want to, you have 2 options:

1. Run the kubectl edit pod <pod name> command.  This will open the pod specification in an editor (vi editor). Then edit the required properties. When you try to save it, you will be denied. This is because you are attempting to edit a field on the pod that is not editable.



A copy of the file with your changes is saved in a temporary location as shown above.

You can then delete the existing pod by running the command:

kubectl delete pod webapp

Then create a new pod with your changes using the temporary file

kubectl create -f /tmp/kubectl-edit-ccvrq.yaml

2. The second option is to extract the pod definition in YAML format to a file using the command

kubectl get pod webapp -o yaml > my-new-pod.yaml

Then make the changes to the exported file using an editor (vi editor). Save the changes

vi my-new-pod.yaml

Then delete the existing pod

kubectl delete pod webapp

Then create a new pod with the edited file

kubectl create -f my-new-pod.yaml

#### **Edit Deployments**

With Deployments you can easily edit any field/property of the POD template. Since the pod template is a child of the deployment specification,  with every change the deployment will automatically delete and create a new pod with the new changes. So if you are asked to edit a property of a POD part of a deployment you may do that simply by running the command

kubectl edit deployment my-deployment

* [Overview](https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/learn/lecture/14937592#overview)
* [Q&A](https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/learn/lecture/14937592#questions)
* [Bookmarks](https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/learn/lecture/14937592#bookmarks)
* [Announcements](https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/learn/lecture/14937592#announcements)

A note about Secrets!

Remember that secrets encode data in base64 format. Anyone with the base64 encoded secret can easily decode it. As such the secrets can be considered as not very safe.

The concept of safety of the Secrets is a bit confusing in Kubernetes. The [kubernetes documentation](https://kubernetes.io/docs/concepts/configuration/secret" \t "_blank) page and a lot of blogs out there refer to secrets as a "safer option" to store sensitive data. They are safer than storing in plain text as they reduce the risk of accidentally exposing passwords and other sensitive data. In my opinion it's not the secret itself that is safe, it is the practices around it.

Secrets are not encrypted, so it is not safer in that sense. However, some best practices around using secrets make it safer. As in best practices like:

* Not checking-in secret object definition files to source code repositories.
* [Enabling Encryption at Rest](https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/)for Secrets so they are stored encrypted in ETCD.

Also the way kubernetes handles secrets. Such as:

* A secret is only sent to a node if a pod on that node requires it.
* Kubelet stores the secret into a tmpfs so that the secret is not written to disk storage.
* Once the Pod that depends on the secret is deleted, kubelet will delete its local copy of the secret data as well.

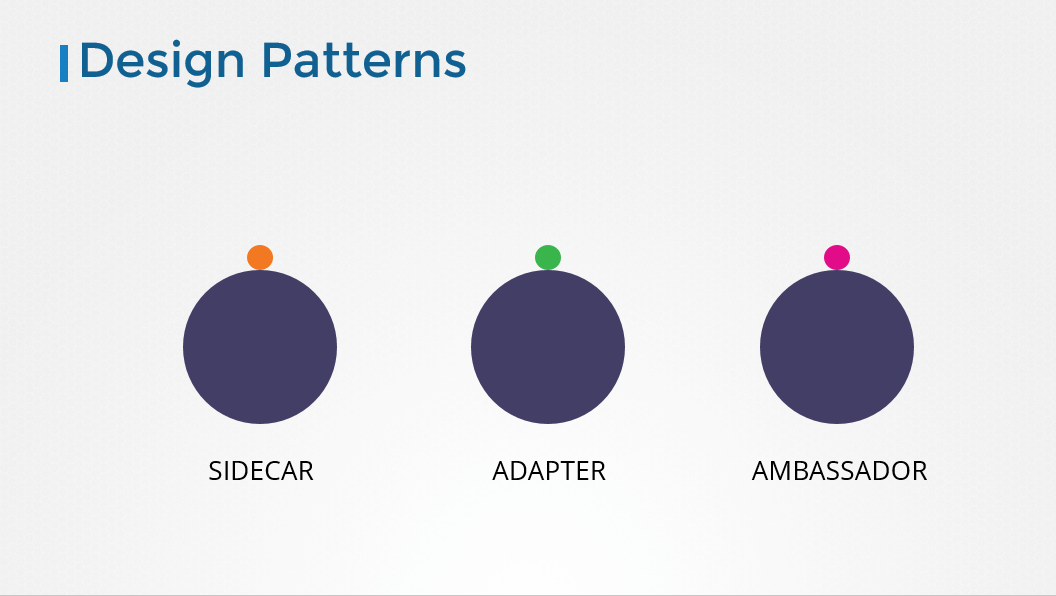
Read about the [protections](https://kubernetes.io/docs/concepts/configuration/secret/#protections)and [risks](https://kubernetes.io/docs/concepts/configuration/secret/#risks) of using secrets [here](https://kubernetes.io/docs/concepts/configuration/secret/#risks)

Having said that, there are other better ways of handling sensitive data like passwords in Kubernetes, such as using tools like Helm Secrets, [HashiCorp Vault](https://www.vaultproject.io/" \t "_blank). I hope to make a lecture on these in the future.

Multi-container PODs Design Patterns

There are 3 common patterns, when it comes to designing multi-container PODs. The first and what we just saw with the logging service example is known as a side car pattern. The others are the adapter and the ambassador pattern.

But these fall under the CKAD curriculum and are not required for the CKA exam. So we will be discuss these in more detail in the CKAD course.



InitContainers

In a multi-container pod, each container is expected to run a process that stays alive as long as the POD's lifecycle. For example in the multi-container pod that we talked about earlier that has a web application and logging agent, both the containers are expected to stay alive at all times. The process running in the log agent container is expected to stay alive as long as the web application is running. If any of them fails, the POD restarts.

But at times you may want to run a process that runs to completion in a container. For example a process that pulls a code or binary from a repository that will be used by the main web application. That is a task that will be run only  one time when the pod is first created. Or a process that waits  for an external service or database to be up before the actual application starts. That's where **initContainers**comes in.

An **initContainer**is configured in a pod like all other containers, except that it is specified inside a initContainers section,  like this:

1. apiVersion: v1
2. kind: Pod
3. metadata:
4. name: myapp-pod
5. labels:
6. app: myapp
7. spec:
8. containers:
9. - name: myapp-container
10. image: busybox:1.28
11. command: ['sh', '-c', 'echo The app is running! && sleep 3600']
12. initContainers:
13. - name: init-myservice
14. image: busybox
15. command: ['sh', '-c', 'git clone <some-repository-that-will-be-used-by-application> ; done;']

When a POD is first created the initContainer is run, and the process in the initContainer must run to a completion before the real container hosting the application starts.

You can configure multiple such initContainers as well, like how we did for multi-pod containers. In that case each init container is run **one at a time in sequential order**.

If any of the initContainers fail to complete, Kubernetes restarts the Pod repeatedly until the Init Container succeeds.

1. apiVersion: v1
2. kind: Pod
3. metadata:
4. name: myapp-pod
5. labels:
6. app: myapp
7. spec:
8. containers:
9. - name: myapp-container
10. image: busybox:1.28
11. command: ['sh', '-c', 'echo The app is running! && sleep 3600']
12. initContainers:
13. - name: init-myservice
14. image: busybox:1.28
15. command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
16. - name: init-mydb
17. image: busybox:1.28
18. command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']

Read more about initContainers here. And try out the upcoming practice test.

<https://kubernetes.io/docs/concepts/workloads/pods/init-containers/>

Self Healing Applications

Kubernetes supports self-healing applications through ReplicaSets and Replication Controllers. The replication controller helps in ensuring that a POD is re-created automatically when the application within the POD crashes. It helps in ensuring enough replicas of the application are running at all times.

Kubernetes provides additional support to check the health of applications running within PODs and take necessary actions through Liveness and Readiness Probes. However these are not required for the CKA exam and as such they are not covered here. These are topics for the Certified Kubernetes Application Developers (CKAD) exam and are covered in the CKAD course.

References

<https://kubernetes.io/docs/concepts/overview/kubernetes-api/>

Here is a link to kubernetes documentation if you want to learn more about this topic (You don't need it for the exam though):

<https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md>

<https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api_changes.md>

References

<https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#backing-up-an-etcd-cluster>

<https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/recovery.md>

<https://www.youtube.com/watch?v=qRPNuT080Hk>

Article on Setting up Basic Authentication

Setup basic authentication on kubernetes

Note: This is not recommended in a production environment. This is only for learning purposes.

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:

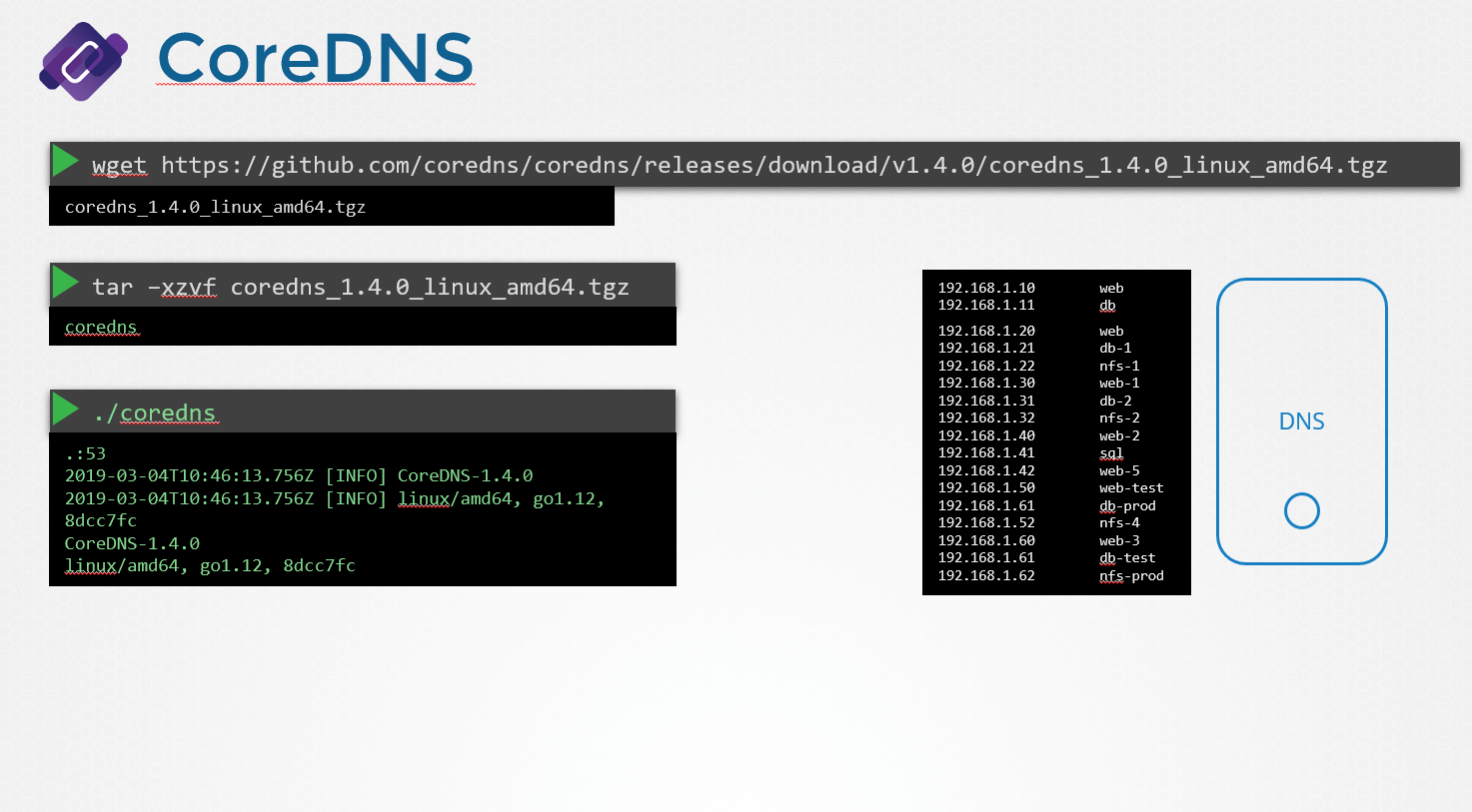
1. ---
2. kind: Role
3. apiVersion: rbac.authorization.k8s.io/v1
4. metadata:
5. namespace: default
6. name: pod-reader
7. rules:
8. - apiGroups: [""] # "" indicates the core API group
9. resources: ["pods"]
10. verbs: ["get", "watch", "list"]
12. ---
13. # This role binding allows "jane" to read pods 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

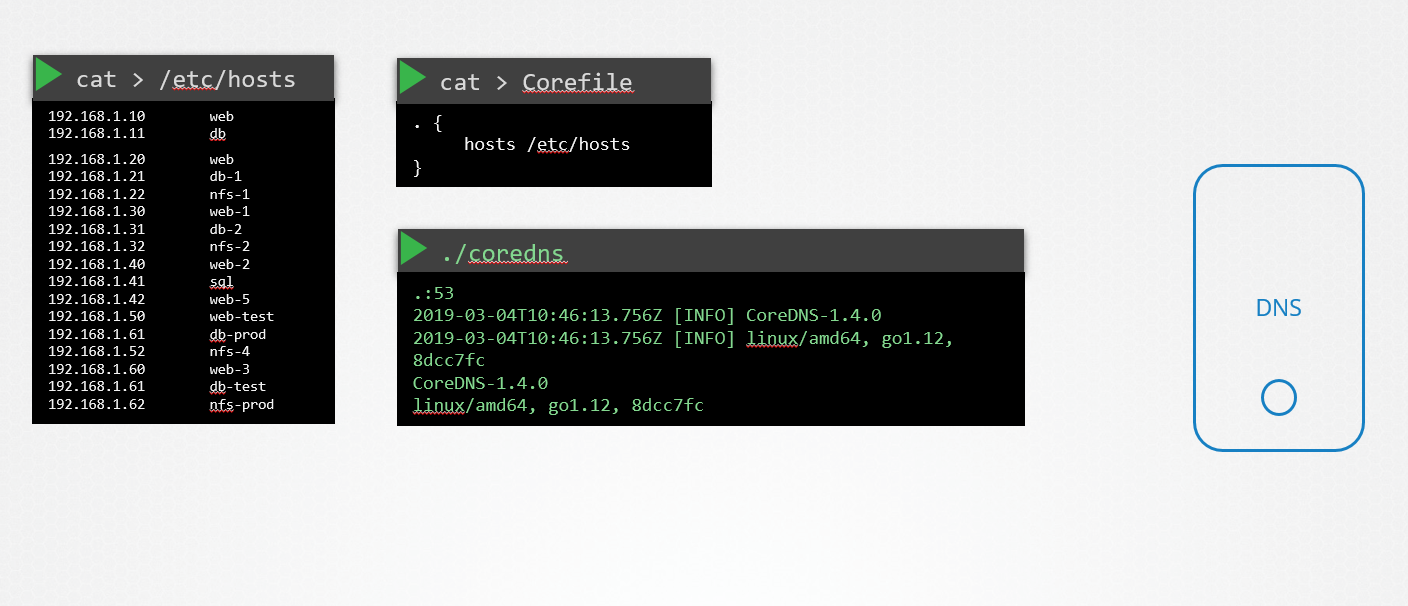
curl -v -k https://localhost:6443/api/v1/pods -u "user1:password123"

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/>

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: /$2
6. name: rewrite
7. namespace: default
8. spec:
9. rules:
10. - host: rewrite.bar.com
11. http:
12. paths:
13. - backend:
14. serviceName: http-svc
15. servicePort: 80
16. path: /something(/|$)(.\*)