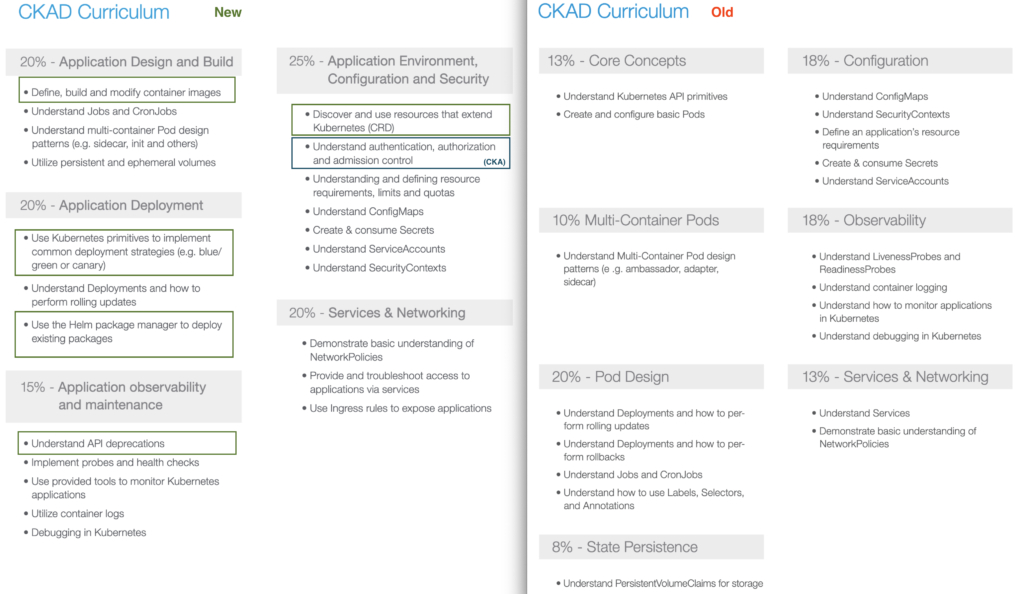
# Updates for Sep 2021 Changes

You might have heard about some of the upcoming changes for the CKAD exam objectives as [announced](https://training.linuxfoundation.org/ckad-program-change-2021/) by the Linux Foundation recently. These changes will go into effect on September 28. If you sit for the exam on the 28th of September or after it will be in the new format, irrespective of when you scheduled the exam or on what version your first attempt was (If you made a previous attempt). All exams after 28th of September will be in the new format.

I wanted to write to you to let you know what we are doing about it and what you should be doing about it. So read on.

#### What’s changing in the CKAD objectives?

Below is a summary of changes. The one on the left is the new exam curriculum and the one on the right is the old exam curriculum. I have highlighted the new exam objectives in green boxes. There are a few new concepts to learn such as building container images, blue/green or canary deployments, Helm, understanding API deprecations, discovering and using resources that extend Kubernetes (CRD), and some security concepts such as Authentication, Authorization and Admission Control. We have already covered the security concepts in the CKA course so you should be covered there if you have done CKA already. We will be adding these to the CKAD course soon.

CKAD Exam Changes September 2021

#### ****What are we doing to our CKAD Course?****

We will be updating our CKAD Preparation course with the additional contents within the next week or two. In this section named – “**Updates for Sep 2021 Changes**” we will add new topics as and when we produce them.

After September 28th we will re-organize the course structure to match the exam curriculum.

#### ****What should you do about preparing for the CKAD Certification?****

You have 2 options:

1. Give your best! Prepare and attempt the exam before the 28th of September.
2. If this is not sufficient time for you to prepare and you don’t think you can make it before the 28th, continue to prepare with the current contents. All existing contents are still part of the exam objectives. About 80% of the contents will remain the same. You may learn the new contents as and when we add them to the course.

We wish you all the best!

# 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

*# User File Contents*

*password123,user1,u0001*

*password123,user2,u0002*

*password123,user3,u0003*

*password123,user4,u0004*

*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

*apiVersion: v1*

*kind: Pod*

*metadata:*

*name: kube-apiserver*

*namespace: kube-system*

*spec:*

*containers:*

*- command:*

*- kube-apiserver*

*<content-hidden>*

*image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3*

*name: kube-apiserver*

*volumeMounts:*

*- mountPath: /tmp/users*

*name: usr-details*

*readOnly: true*

*volumes:*

*- hostPath:*

*path: /tmp/users*

*type: DirectoryOrCreate*

*name: usr-details*

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

*apiVersion: v1*

*kind: Pod*

*metadata:*

*creationTimestamp: null*

*name: kube-apiserver*

*namespace: kube-system*

*spec:*

*containers:*

*- command:*

*- kube-apiserver*

*- --authorization-mode=Node,RBAC*

*<content-hidden>*

*- --basic-auth-file=/tmp/users/user-details.csv*

Create the necessary roles and role bindings for these users:

*---*

*kind: Role*

*apiVersion: rbac.authorization.k8s.io/v1*

*metadata:*

*namespace: default*

*name: pod-reader*

*rules:*

*- apiGroups: [""] # "" indicates the core API group*

*resources: ["pods"]*

*verbs: ["get", "watch", "list"]*

*---*

*# This role binding allows "jane" to read pods in the "default" namespace.*

*kind: RoleBinding*

*apiVersion: rbac.authorization.k8s.io/v1*

*metadata:*

*name: read-pods*

*namespace: default*

*subjects:*

*- kind: User*

*name: user1 # Name is case sensitive*

*apiGroup: rbac.authorization.k8s.io*

*roleRef:*

*kind: Role #this must be Role or ClusterRole*

*name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to*

*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"

**Certification Tip – Imperative Commands!**

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=client 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 nginx --image=nginx

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

kubectl run nginx --image=nginx --dry-run=client -o yaml

Deployment

**Create a deployment**

kubectl create deployment --image=nginx nginx

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

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

**Generate Deployment with 4 Replicas**

kubectl create deployment nginx --image=nginx --replicas=4

You can also scale a deployment using the kubectl scale command.

kubectl scale deployment nginx --replicas=4

**Another way to do this is to save the YAML definition to a file and modify**

kubectl create deployment nginx --image=nginx--dry-run=client -o yaml > nginx-deployment.yaml

You can then update the YAML file with the replicas or any other field before creating the deployment.

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=client -o yaml

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

Or

kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -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 --type=NodePort --dry-run=client -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=client -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/>

# A quick note on editing PODs and Deployments

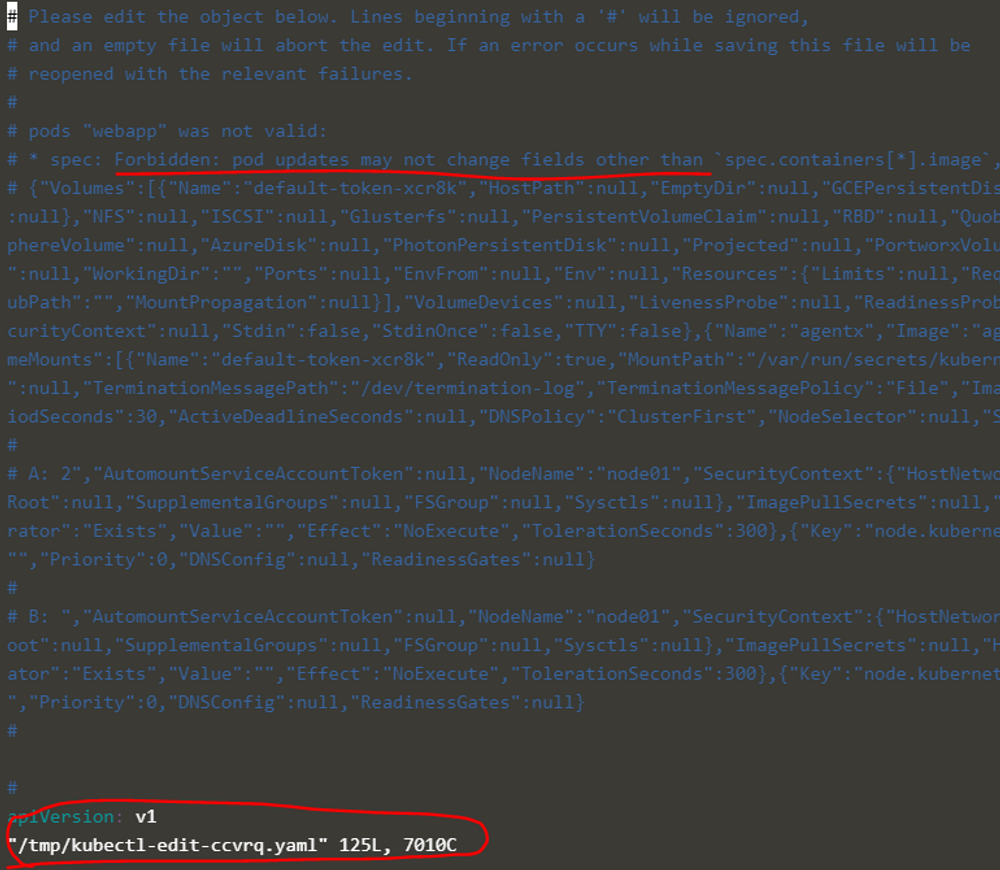
#### Edit a POD

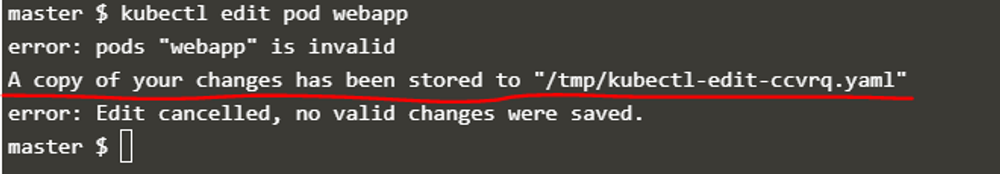
Remember, you CANNOT edit specifications of an existing POD other than the below.

* 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

# A quick 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) 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/). I hope to make a lecture on these in the future.