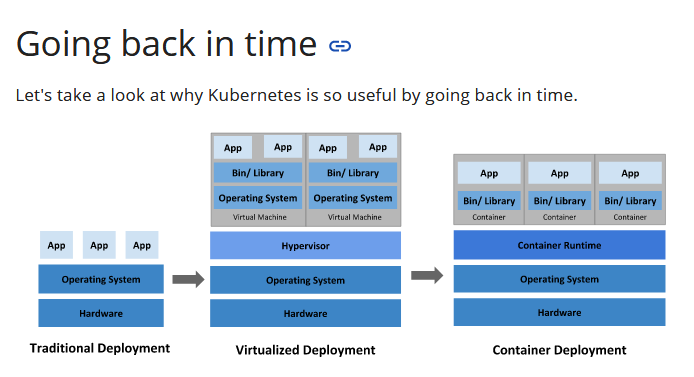
**What is Kubernetes?**

*Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.*

*The name Kubernetes originates from Greek, meaning helmsman or pilot. Google open-sourced the Kubernetes project in 2014. Kubernetes combines* [over 15 years of Google's experience](https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/) *running production workloads at scale with best-of-breed ideas and practices from the community*

**ORIGIN**



**Container deployment era:**

*Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.*

*Containers have become popular because they provide extra benefits, such as:*

* *Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.*
* *Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).*
* *Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.*
* *Observability not only surfaces OS-level information and metrics, but also application health and other signals.*
* *Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.*
* *Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.*
* *Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.*
* *Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.*
* *Resource isolation: predictable application performance.*
* *Resource utilization: high efficiency and density.*

**Why you need Kubernetes and what it can do**

*Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?*

*That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system.*

*Kubernetes provides you with:*

* ***Service discovery and load balancing*** *Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.*
* ***Storage orchestration*** *Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.*
* ***Automated rollouts and rollbacks*** *You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.*
* ***Automatic bin packing*** *You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.*
* ***Self-healing*** *Kubernetes restarts containers that fail, replaces containers, kills containers that don’t respond to your user-defined health check, and doesn’t advertise them to clients until they are ready to serve.*
* ***Secret and configuration management*** *Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration*

**What Kubernetes is not**

Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

Kubernetes:

* Does not limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
* Does not deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
* Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.
* Does not dictate logging, monitoring, or alerting solutions. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
* Does not provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
* Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.
* Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn’t matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

**Kubernetes Components**

*A Kubernetes cluster consists of the components that represent the control plane and a set of machines called nodes.*

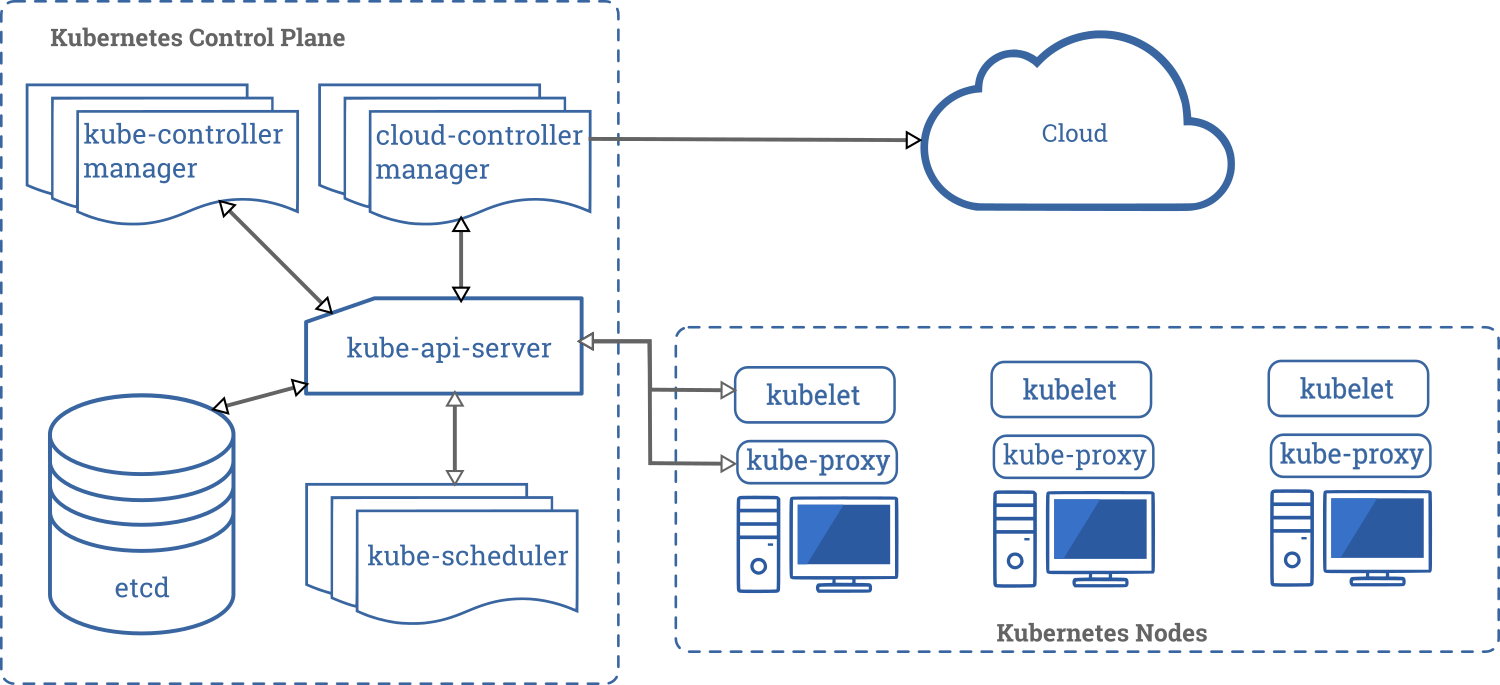
*When you deploy Kubernetes, you get a cluster.*

*A Kubernetes cluster consists of a set of worker machines, called* nodes*, that run containerized applications. Every cluster has at least one worker node.*

*The worker node(s) host the* Pods *that are the components of the application workload. The* control plane *manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.*

*This document outlines the various components you need to have a complete and working Kubernetes cluster.*

*Here's the diagram of a Kubernetes cluster with all the components tied together.*

**

***Control Plane Components***

*The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new* pod *when a deployment's replicas field is unsatisfied).*

*Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See* [Building High-Availability Clusters](https://kubernetes.io/docs/admin/high-availability/) *for an example multi-master-VM setup.*

***kube-apiserver***

*The API server is a component of the Kubernetes* control plane *that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.*

*The main implementation of a Kubernetes API server is* kube-apiserver*. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.*

***etcd***

*Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.*

*If your Kubernetes cluster uses etcd as its backing store, make sure you have a* [back up](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#backing-up-an-etcd-cluster) *plan for those data.*

*You can find in-depth information about etcd in the official* [documentation](https://etcd.io/docs/)*.*

***kube-scheduler***

*Control plane component that watches for newly created* Pods *with no assigned* node *, and selects a node for them to run on.*

*Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.*

***kube-controller-manager***

*Control Plane component that runs* controller *processes.*

*Logically, each* controller *is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.*

*These controllers include:*

* *Node controller: Responsible for noticing and responding when nodes go down.*
* *Replication controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.*
* *Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).*
* *Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.*

***cloud-controller-manager***

*A Kubernetes* control plane *component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that just interact with your cluster.*

*The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.*

*As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.*

*The following controllers can have cloud provider dependencies:*

* *Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding*
* *Route controller: For setting up routes in the underlying cloud infrastructure*
* *Service controller: For creating, updating and deleting cloud provider load balancers*

***Node Components***

*Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.*

***kubelet***

*An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.*

*The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn’t manage containers which were not created by Kubernetes.*

***kube-proxy***

*kube-proxy is a network proxy that runs on each* node *in your cluster, implementing part of the Kubernetes* Service *concept.*

kube-proxy *maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.*

*kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.*

***Container runtime***

*The container runtime is the software that is responsible for running containers.*

*Kubernetes supports several container runtimes:* [Docker](https://docs.docker.com/engine/) *,* [containerd](https://containerd.io/docs/) *,* [CRI-O](https://cri-o.io/#what-is-cri-o) *, and any implementation of the* [Kubernetes CRI (Container Runtime Interface)](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md)*.*

***Addons***

*Addons use Kubernetes resources (*[DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset" \t "_blank) *,* [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) *, etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the kube-system namespace.*

*Selected addons are described below; for an extended list of available addons, please see* [Addons](https://kubernetes.io/docs/concepts/cluster-administration/addons/)*.*

***DNS***

*While the other addons are not strictly required, all Kubernetes clusters should have* [cluster DNS](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/)*, as many examples rely on it.*

*Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.*

*Containers started by Kubernetes automatically include this DNS server in their DNS searches.*

***Web UI (Dashboard)***

[Dashboard](https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/) *is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.*

***Container Resource Monitoring***

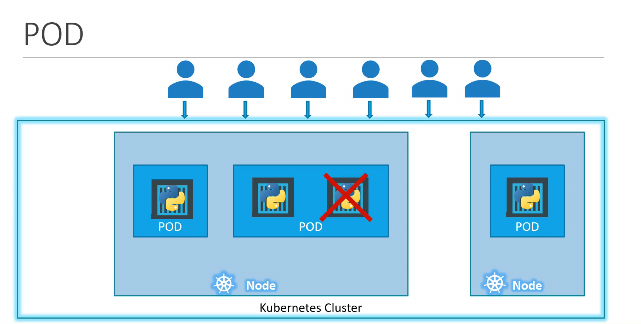
[Container Resource Monitoring](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/) *records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.*

***Cluster-level Logging***

*A* [cluster-level logging](https://kubernetes.io/docs/concepts/cluster-administration/logging/) *mechanism is responsible for saving container logs to a central log store with search/browsing interface.*

**PODS**

*Kubernetes our ultimate aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster. However Kubernetes does not deploy containers directly on the worker nodes. The containers are encapsulated into a Kubernetes object known as PODs. A POD is a single instance of an application. A POD is the smallest object that you can create in Kubernetes.*



Pod creation using YAML



**what is a replica and why do we need a replication controller. & ReplicaSet**

Let's go back to our first scenario where we had a single pod running our application. What if for some reason our application crashes and the part fails users will no longer be able to access our application to prevent users from losing access to our application. We would like to have more than one instance or pod running at the same time. That way if one failed we still have our application running on the other one. The replication controller helps us run multiple instances of a single part in the cabinet as cluster thus providing high availability so does that mean you can't use a replication controller if you plan to have a single pod.

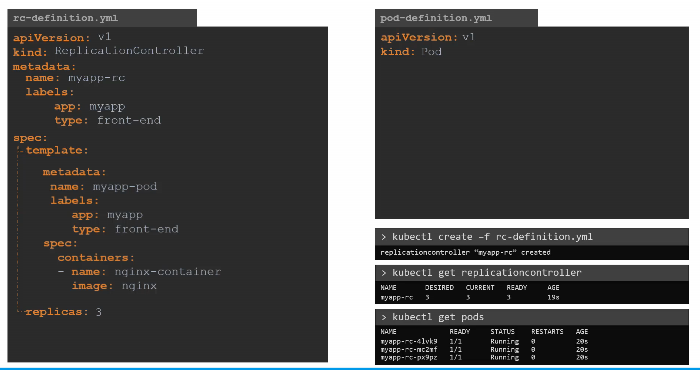
No.

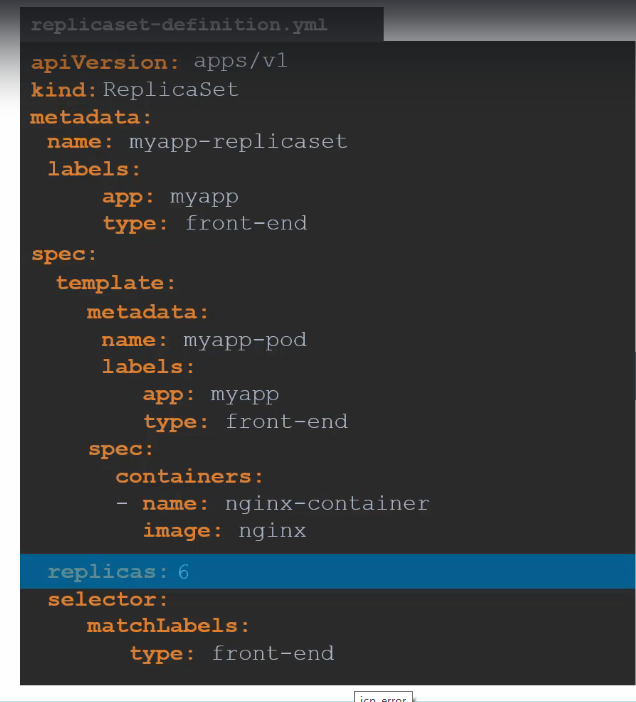
Even if you have a single pod the replication controller can help by automatically bringing up a new pod when the existing one fails does the replication controller ensures that the specified number of parts are running at all times even if it's just one or 100 another reason we need a replication controller is to create multiple paths to share the load across them.

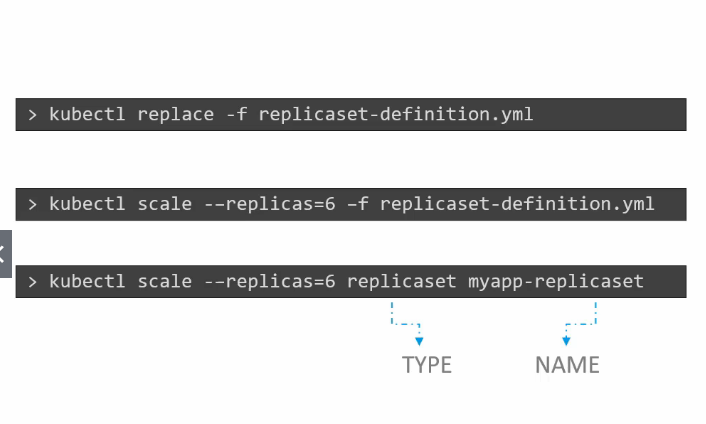
For example in this simple scenario we have a single pod serving a set of users when the number of users increase. We deploy additional pod to balance the load across the two pods if the demand further increases and if we were to run out of resources on the first node we could deploy additional pods across the other nodes in the cluster.

As you can see the replication controller spans across multiple nodes in the cluster. It helps us balance the load across multiple paths on different nodes as well as scale our application.

* Replica Controller is modified into Replica Set in the latest release, but most of the things remains same with some improvements.



Example for Replica Set



**Deployments**

*For a minute, let us forget about Pods and Replica Sets and other Kubernetes concepts and talk about how you might want to deploy your application in a production environment.Say for example you have a web server that needs to be deployed in a production environment.*

*You need not one but many such instances of the web server running. For obvious reasons. Secondly, whenever newer versions of application builds become available on the docker registry, you would like to upgrade your Docker instances seamlessly. However, when you upgrade your instances you do not want to upgrade all of them at once as we just did.*

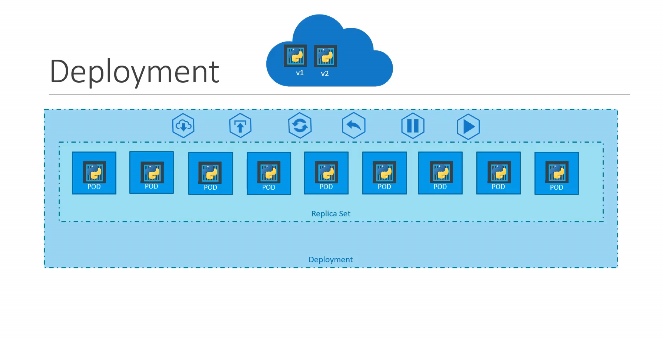
*This may impact users accessing your application so you might want to upgrade them one after the other and that kind of upgrade is known as rolling updates. Suppose one of the upgrades you performed resulted in an unexpected error and you're asked to undo the recent change you would like to be able to roll back the changes that were recently carried out.*

*Finally Say for example you would like to make multiple changes to your environment such as upgrading the underlying Web Server versions as well as Scaling the environment and also modifying the resource allocations etc.*

*You do not want to apply each change immediately after the command is run, instead you like to apply a pause to your environment. Make the changes and then resumes so that all the changes are rolled out together. All of these capabilities are available with the Kubernetes deployments.*

*So far in this course we discussed about PODs which deploys single instances of our application such as the web application in this case. Each container is encapsulated in PODs. Multiple such PODs are deployed using replication controllers or replica sets and then comes deployment which is a Kubernetes object that comes higher in the hierarchy.*

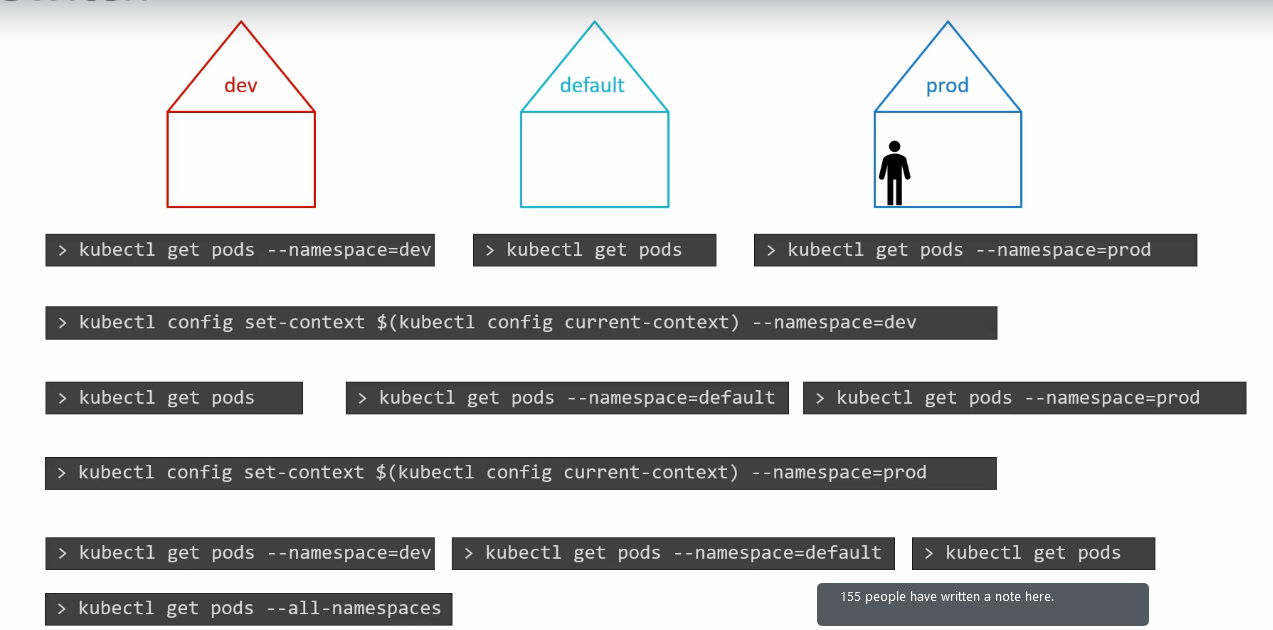
*The deployment provides us with the capability to upgrade the underlying instances seamlessly using rolling updates, undo changes and pause and resume changes as required.*

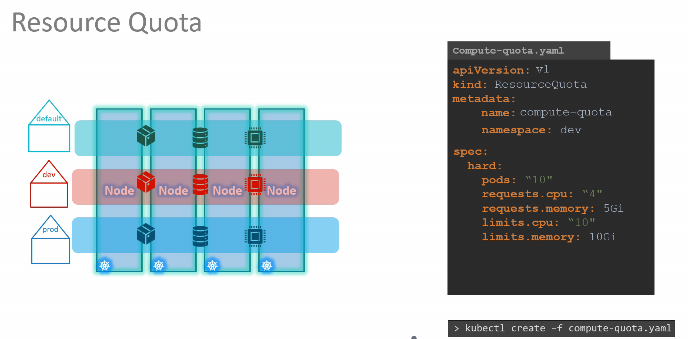


**NAMESPACES**

*Kubernetes Namespace is helpful for segregating the resources which belongs to which one. For an example, you are using the same k8s cluster for development & production. In such cases to separate each environment these namespaces will help.*

*$ >> Kubectl create namespace <<namepace name>>*





**Commands & Configurations**

*While going into the understanding of configurations & commands in kubernetes, let’s have a look on Docker commands which we are going to be used in our kubernetes pod configurations.*

*If you run a command “docker run ubuntu”, we can see a container get created and exited immediately because the Dockerfile of ubuntu doesn’t have any running process. A container will be running until the process inside that container is running. Every Docker image as a specified command which make a process to run continuously.*

*Example Dockerfile: <will check for the image of ubuntu 16 docker file to understand>*



*Ubuntu docker file has CMD[“bash”] which will try to trigger bash process in the terminal but by default is was not enabled. So, container goes down immediately after coming up. Now on top of that ubuntu you add your own things and create an image and deploy the container.*

*For example, if I added a sleep time of 10secs the container lives for that times and goes off what the process completes.*

***#create a ubuntu-sleep image from this Dockerfile.***

*From ubuntu:16.04*

*CMD [“sleep”,”10”]*

*Now the container remains for 10secs, to override that command you can use it with docker.*

* *Docker run ubuntu-sleep sleep 20*

*But seems this looks abit odd because image was already meant for sleep then why to again imply it in command. The change for this is ENTRYPOINT. It acts as an default executable or process to run and arguments only can be send in the CMD of a dockerfile.*

*#create a ubuntu-sleep image for permanent executable “sleep” from this Dockerfile.*

*From ubuntu:16.04*

*ENTRYPOINT [“sleep”]*

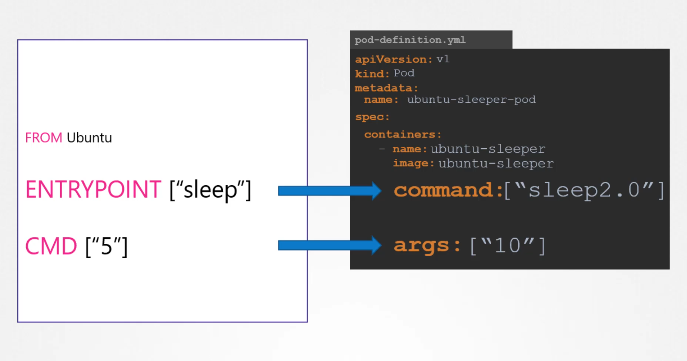
*CMD [“10”]*

***Here are the few sample dockerfile of “NGINX” & “MYSQLD”***



***Mapping of these commands to pod configurations:***

*Now to deploy the above created image in kubernetes as pods, the basic structure of the pods remains the same, the ENTRYPOINT & CMD will be mapped here as “command” & “args” will be added to the respective pods, remember carefully which is for what. Here is an example pod definition.*



***Editing the Pods & Deployments***

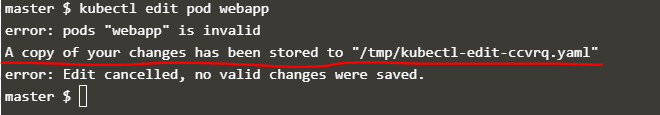
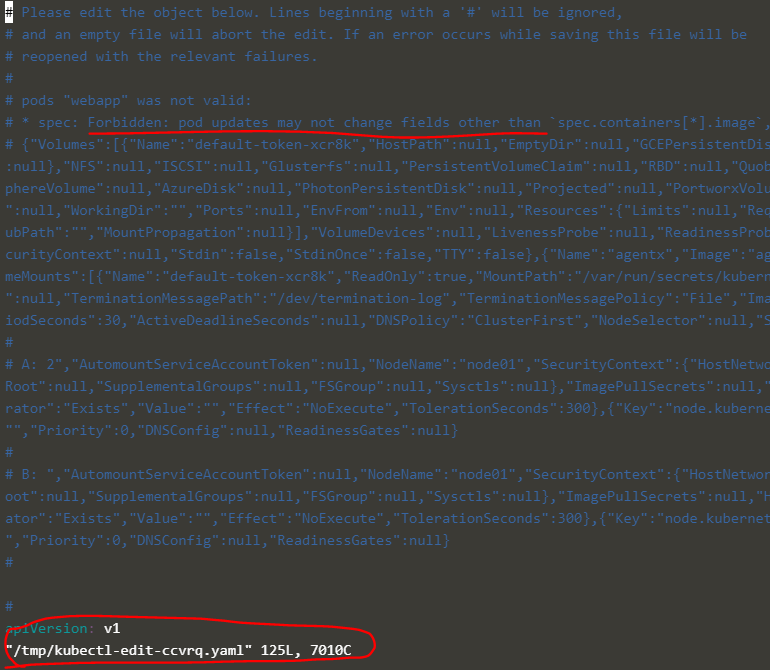
***Edit a POD***

*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***

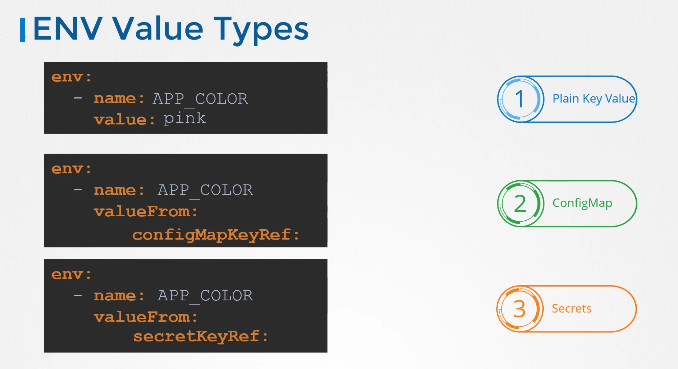
*With Deployments, you can easily edit any field/property of the POD template.* ***Since the pod template is a child of the deployment specification****, with every change the deployment will automatically delete and create a new pod with the new changes. So, if you are asked to edit a property of a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

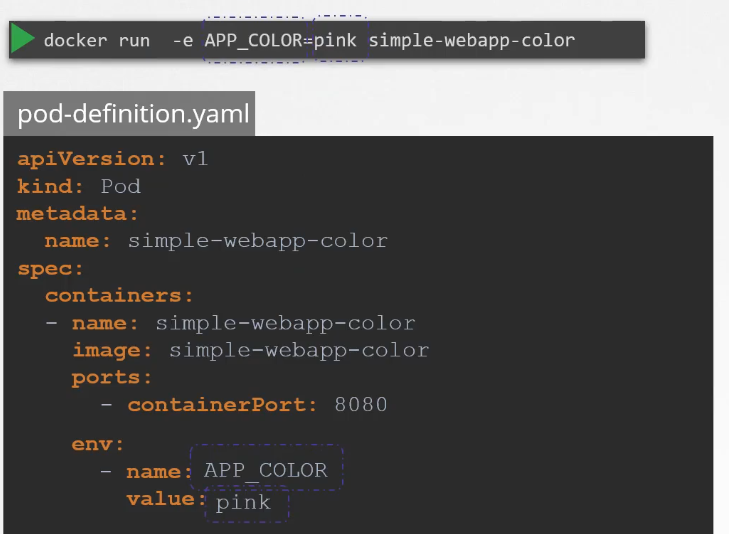
***>> kubectl edit deployment my-deployment***

**ENVIRONMENT VARIABLES**

*One can send Environment variables into the container running as a pod in kubernetes deployment, by 3 different ways, i.e.* ***Using env, Using ConfigMap, Using Secrets***

*Here is a small example for directly reading the ENV parameters from the pod-definition.yml file, under the spec section for each container, there will be an each env section which is an array of environment variables.*





**CONFIGMAPS**

*In the previous section, we saw how to define environment variables in a pod definition file. When you have a lot of pod definition files, it will become difficult to manage the environment data stored within the various files. We can take this information out of the pod definition file and manage it centrally using Configuration maps (i.e. ConfigMap).*

*Config maps are used to pass configuration data in the form of key value pairs in Kubernetes.*

*When it pod is created, inject the config map into the pod. So the key-value pairs are available as environment variables for the application hosted inside the container in the pod.*

*There are two phases involved in configuring config maps.* ***First create the ConfigMap*** *and* ***second inject them into the pod****.*

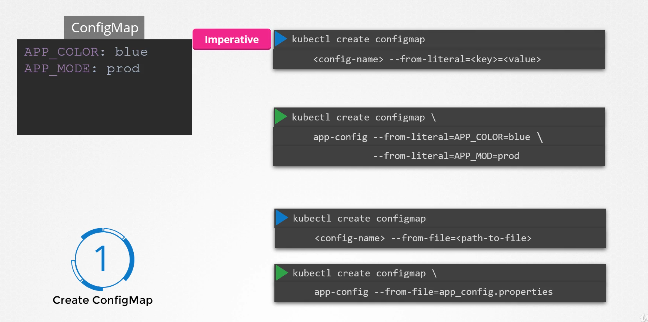
*Just like any other Kubernetes object, there are two ways of creating a config map. The imperative way without using a ConfigMap definition file and the declarative way by using a config map definition file.*

*If you do not wish to create a ConfigMap definition file you could simply use the kubectl create*

*configmap command and specify the required arguments. Let's look at that first. With this method, you can directly specify the key-value pairs in the command line. To create a complete map of the given values run the kubectl create configmap command. The command is followed by config name and the option --from-literal. The --from-literal option is used to specify the key value pairs in the command itself.*

*In this example, we are creating a config map by the name app-config with a key value pair of APP\_COLOR=blue. If you wish to add additional key value pairs simply specify the --from-literal option multiple times. However, this will get complicated when you have too many configuration items. Another way to input configuration data is through a file. Use the --from-file option to specify a path to the file that contains the required data.*

*The data from this file is read and stored under the name of the file.*

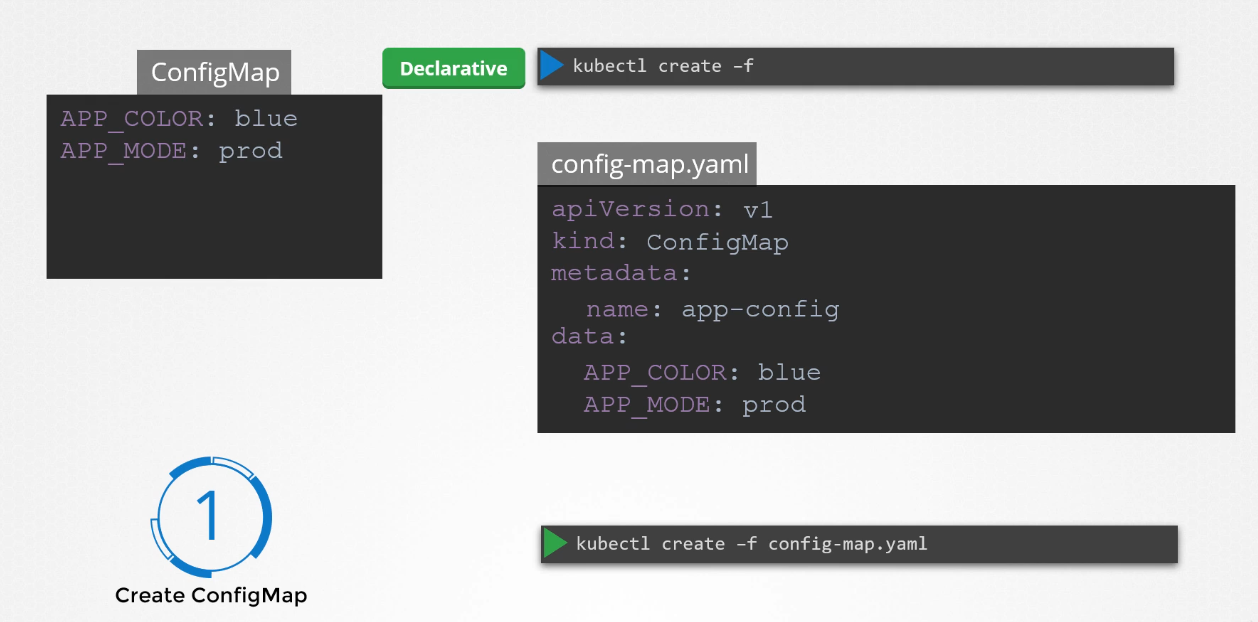


*Let us now look at the declarative approach. For this we create a definition file just like how we did for the pod. The file has an apiVersion, kind, metadata, hence instead of spec here we have data.*

*The apiVersion is v1, the kind ConfigMap. Under metadata we specify a name for the ConfigMap*

*we will call it app-config. Under data section, enter the configuration data in a key-value format.*

*>> kubectl create -f <config-definition filename>*



*You can create as many ConfigMaps as you need in the same way for various different purposes. Here we have one application, other for mySQL and another one for redis. So, it is important to name the configmaps appropriately as you will be using these names later while associating it with pods.*

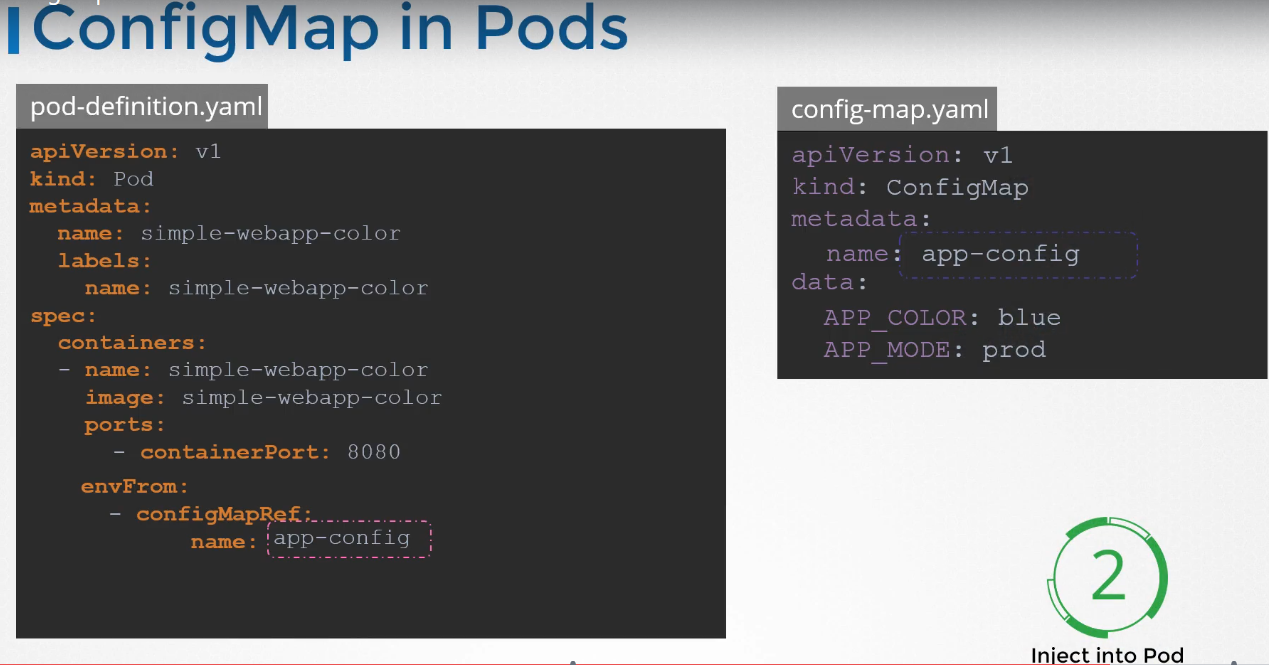


*To view configmaps run the command.*

***>> kubectl get configmaps***

***Now that we have the configmap created let us proceed with step 2 configuring it with a pod here****.*

*I have a simple pod definition file that runs my simple web application. To inject an environment variable, add a new property to the container called envFrom. The envFrom property is a list. so we can pass as many environment variables as required. Each item in the list corresponds to a configmap item. [These] specify the name of the config map we created earlier.*



*This is how we inject a specific configmap from the ones we created before****. What we just see in above figure was using config maps to inject environment variables.***

*There are other ways to inject configuration data into pods.*

*You can inject it as* ***a single environment variable*** *or you can inject the whole the data as* ***files in***

***a volume****.*

